FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Modular Nonmonotonic Logic Programs

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Thomas Krennwallner
Registration Number 00026236

to the Faculty of Informatics
at the Vienna University of Technology
Advisor: Prof. Dr. Thomas Eiter

The dissertation has been reviewed by:

Docent Dr. Tomi Janhunen

Prof. Dr. Stefan Woltran

Wien, 15.08.2018

Thomas Krennwallner

Erklärung zur Verfassung der Arbeit

Thomas Krennwallner
Favoritenstraße 9–11, 1040 Wien

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.

Wien, 15.08.2018

Thomas Krennwallner

Abstract

Modular programming is common practice in software development, and the vast majority of general-purpose programming languages use modularity concepts to aid software engineers in designing and building complex systems based on reusable software components. Answer Set Programming (ASP), in comparison, is a popular paradigm for declarative programming and knowledge representation, but methods for reusing subprograms and program elements in ASP have not arrived for common use yet.

This thesis proposes Modular Nonmonotonic Logic Programs (MLPs), which are disjunctive logic programs under answer set semantics with modules that have contextualized input. Such programs incorporate a call by value mechanism and allow for unrestricted calls between modules—including mutual and self recursion—as a new approach to extend ASP with module constructs akin to those found in conventional programming. We define a model-theoretic semantics for this extended setting, show that many desired properties of ordinary logic programming generalize to modular ASP, and determine the computational complexity of the new formalism.

For the purpose of implementation, we consider rewriting techniques that make MLP semantics amenable to off-the-shelf ASP solvers. We present translations that take an MLP with module input and rewrite them in stages to a combined logic program without input that is evaluable with ASP reasoners. This operation comes at the price of inflating the program exponentially, but complexity-theoretic assumptions suggest that this is unavoidable. The alternative macro expansion technique applicable to syntactically restricted MLPs does not incur the blowup observable in the general setting, and we make use of it to develop an application by embedding hybrid Description Logic Programs into MLPs. This effectively unites MLP with established Datalog engines as backbone for the computation, which we experimentally evaluate.

We characterize answers sets in terms of classical (Herbrand) models of propositional, first-, and second-order sentences, extending a line of research for conventional logic programs. To this end, we lift on one side well-known loop formulas to MLPs, and otherwise augment ordered program completion for MLPs, which avoids explicit loop formula construction by auxiliary predicates. A further result is a study on the relationship of MLPs and DLP-functions, which is a notable formalism for compositional modular ASP with well-defined input/output interface. These investigations widen our understanding of MLPs and may prove beneficial for further semantic analysis and implementation perspectives.

Kurzfassung

Modulare Programmierung ist gängige Praxis in der Softwareentwicklung, und die überwiegende Mehrzahl universeller Programmiersprachen verwenden Modularitätskonzepte, um Softwareentwicklern beim Entwerfen und Konstruieren komplexer Systeme basierend auf wiederverwendbaren Softwarekomponenten zu helfen. Im Vergleich dazu sind Methoden zur erneuten Verwendung von Unterprogrammen und Programmelementen in der Answer Set Programmierung (ASP), einem weitverbreiteten Paradigma zur deklarativen Programmierung und Wissensrepräsentation, noch nicht im allgemeinem Gebrauch angekommen.

Diese Arbeit schlägt Modulare Nichtmonotone Logische Programme (MLP) vor, das heißt, disjunktive logische Programme unter Antwortmengen-Semantik mit Modulen, die kontextuellen Input unterstützen. Solche Programme integrieren einen Wertaufruf-Mechanismus und erlauben uneingeschränkte Aufrufe zwischen den Modulen—inklusive wechselseitiger sowie Selbst-Rekursion—, um ASP mit Konstrukten zu erweitern, die auch in konventioneller Programmierung aufzufinden sind. Wir definieren eine modell-theoretische Semantik für diese erweiterte Umgebung, zeigen, dass viele gewünschte Eigenschaften der gewöhnlichen logischen Programmierung sich verallgemeinern lassen hin zu modularem ASP, und bestimmen die Berechnungskomplexität des neuen Formalismus.

Zwecks Implementierung betrachten wir Umformungstechniken, um MLP zugänglich für gebrauchsfertige ASP Solver zu machen. Wir präsentieren Konvertierungen, die ein MLP mit Modul Input entgegennehmen und diese sukzessive in ein zusammengesetztes logisches Programm ohne Input umschreiben, sodass diese dann für ASP Reasoner auswertbar werden. Diese Operation hat ihren Preis in einem exponentiell aufgeblähtem Programm, jedoch legen Annahmen aus der Komplexitätstheorie nahe, dass dies unvermeidlich ist. Die alternative Makroexpansionstechnik anwendbar auf syntaktisch eingeschränkte MLP erleidet diese Vergrößerung, die im allgemeinen Rahmen wahrnehmbar ist, jedoch nicht. Daher benutzen wir diese, um eine Anwendung durch Einbetten von hybriden Beschreibungslogikprogrammen in MLP zu entwickeln. Dies vereint MLP mit bewährten Datalog Engines als Rückgrat zur Auswertung auf wirksame Weise, welche wir experimentell evaluieren.

Wir charakterisieren die Antwortmengen anhand von klassischen (Herbrand) Modellen von Sätzen in propositionaler Logik sowie Prädikatenlogik erster und zweiter Stufe, und erweitern dadurch eine Forschungsrichtung innerhalb der konventionellen

logischen Programmierung. Dazu generalisieren wir einerseits die bekannten Loop-Formeln, und andererseits erweitern wir geordnete Programmvervollständigung für MLP, die explizite Loop-Formelerzeugung durch Hilfsprädikate umgehen. Ein weiteres Ergebnis ist ein Studie über den Zusammenhang von MLP und DLP-Funktionen, ein bedeutender Formalismus für kompositionelles modulares ASP mit wohldefinierter Input/Output Schnittstelle. Diese Untersuchungen vertiefen unser Verständnis von MLP, welche sich als nützlich für weitere semantische Analysen und Implementierungsperspektiven erweisen könnten.

Acknowledgments

I wish to thank many persons who supported me with my research. I am truly grateful to my PhD advisors Thomas Eiter and Michael Fink for their guidance and help and for the time, knowledge, discussions, and feedback that they kindly gave me. We have had numerous meetings, which provided the basis for fleshing out technical details and eliminating theoretical and practical problems at every stage of development. They introduced me to the topic of modular logic programming and distributed reasoning, which sparked the idea of setting up a research proposal that obtained funding for pursuing this research. I want to thank hereby the Austrian Science Fund (FWF) for materializing project *Modular HEX-Programs (P20841)*.

I thank my colleague and friend DAO Tran Minh, with whom I not only did research in the aforementioned project, but also have the pleasure of working together at XIMES company. Together we had investigated theoretical aspects, written scientific papers, implemented algorithms, and traveled conferences. I am glad we can continue to work as a team.

I thank Tomi Janhunen and Stefan Woltran for accepting to serve on my dissertation committee, and for providing extensive and diligent feedback and comments on draft manuscripts of this thesis, which helped to produce the final version.

I would like to thank Axel Polleres for his hospitality in giving me the opportunity to work as research intern from October 2007 to May 2008 at DERI Galway before the research project came into being. During this collaboration, new research directions and ideas had flourished, which I had the great fortune not to miss.

I wish to express my gratitude to Giovambattista Ianni, with whom I not only could continually do research fruitfully, but also organize the Answer Set Programming Competition 2013, together with colleagues from University of Calabria and Vienna University of Technology. To mark their contributions, I list them here by name in alphabetical order: Mario Alviano, Francesco Calimeri, Günther Charwat, Carmine Dodaro, Martin Kronegger, Johannes Oetsch, Andreas Pfandler, Jörg Pührer, Christoph Redl, Francesco Ricca, Patrik Schneider, Martin Schwengerer, Lara Katharina Spendier, Johannes Peter Wallner, and Guohui Xiao. Moreover, I thank my fellow co-authors, colleagues, and students at the Knowledge Based Systems Group (in alphabetical order): Harald Beck, Uwe Egly, Magdalena Ortiz, Peter Schüller, Mantas Šimkus, Daria Stepanova, Hans Tompits, Antonius Weinzierl, and Lena Widl. I enjoyed our regu-

lar discussion, coffee breaks, research manuscripts preparations, organization duties, teaching courses, and travels to scientific conferences together.

I would like to thank Elfriede Nedoma, who painstakingly arranged travels and all sorts of administrative work. Moreover, I thank Matthias Schlögel for helping me keeping the machines for our experimental HTCondor distributed computation environment up and running. This thesis could not have been realized with all the fine software tools and fonts that are carefully handcrafted and designed for typesetting by able engineers and scientists, thus I dedicate a section at the end of this thesis to honor their contributions.

I want to give thanks to my parents for always supporting me and making it possible to pursue my bachelor and master studies. The courses I had taken during that time laid the foundation for this thesis.

Dedicated to the cats of catland

Contents

List of Figures

List of Tables

List of Results

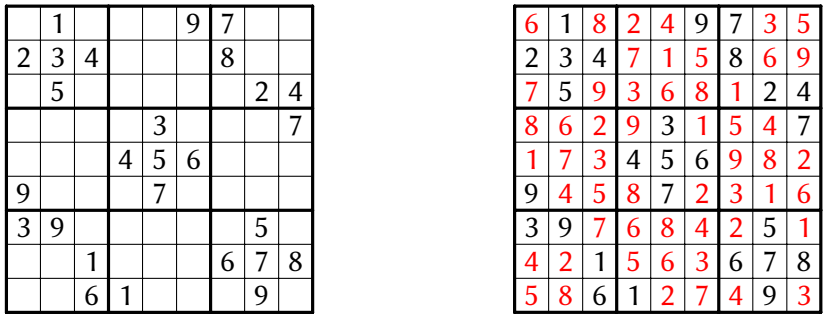
Modularity in Logic Programming

1

Introduction

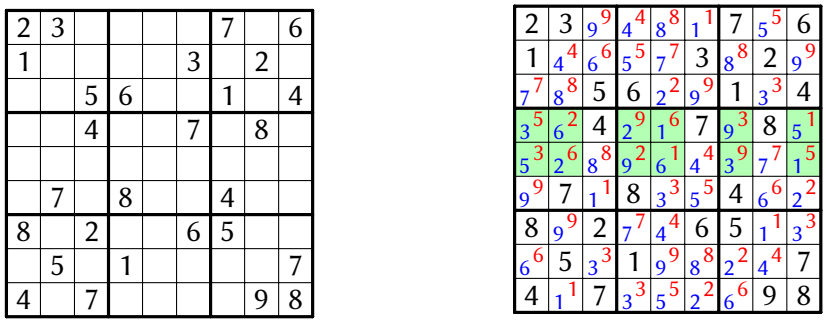
Answer set programming (ASP) is a well-established paradigm for declarative programming with roots to be found in Logic Programming and in Knowledge Representation and Reasoning, the branch of Artificial Intelligence concerned with explicitly representing information using logical formalisms. An advantage of ASP is to provide a versatile declarative modeling framework with many attractive features that allow turning problem statements of computationally hard problems with little to no effort into executable formal specifications, also called answer set programs. These programs can be used to describe and reason over problems in a large variety of domains, for example, commonsense and agent reasoning, diagnosis, deductive databases, software upgrade dependency handling, planning, product configuration, bioinformatics, scheduling, shift design, Markov network learning, and timetabling. See Brewka et al. (2011) for an overview article, the book by Gebser et al. (2013) for the practical details on how to implement answer set programs, and Brewka et al. (2016) for applications of answer set programming and further in-depth material. ASP has a close relationship to other declarative modeling paradigms and languages, such as SAT solving (Biere et al., 2009), Satisfiability Modulo Theories (SMT, see Barrett et al., 2009; Nieuwenhuis et al., 2006), automated theorem proving (Robinson and Voronkov, 2001), Constraint Programming (CP, see Rossi et al., 2006), and many others. All these formalisms have in common that they were designed for solving demanding problems, many of which arise in applications in Artificial Intelligence.

In ASP, problems are represented by nonmonotonic logic programs, such that the *stable models* (or *answer sets*) of the program represent the solutions to a given problem instance (Gelfond and Lifschitz, 1991). As an example for a problem that can be solved with ASP is the Sudoku puzzle, a highly successful number-placement riddle. Although simple to explain, Sudoku is neither easy to solve nor easy to implement efficiently with an imperative programming language. Given a 9×9 grid, the goal of the Sudoku game is to fill every cell in the grid with numbers in the range $D = 1, \ldots, 9$, such that each



(a) Puzzle 3 (b) Unique solution

Figure 1.1: Classic Sudoku Puzzle 3 from WSC 2016



(a) Puzzle 5 (b) Two solutions

Figure 1.2: Classic Sudoku Puzzle 5 from WSC 2016

$d \in D$ appears exactly once in every row, every column, and every 3×3 sub-grid that composes the grid. A Sudoku problem instance is a partially filled grid, which must be completed given the Sudoku constraints defined before.

Next, we will show three puzzles from the Official Practice Test for the 11th World Sudoku Championship (WSC) 2016 (Demiger, 2016). The first Sudoku instance, WSC Puzzle 3, is shown in Figure 1.1a. It has a single solution presented with red numbers in Figure 1.1b. But there are further kinds of Sudoku instances: one kind admits more than one solution, the other kind has no solution at all. Figure 1.2a shows WSC Puzzle 5, which has two solutions shown in Figure 1.2b: the first solution is displayed in blue numbers in the bottom-left corners of the cells, the second solution uses red numbers in the top-right corners of the cells. The cells with green background highlight the alternative parts of each solution. A Sudoku instance without solution is the one in Figure 1.3, WSC Puzzle 8; there is no way to fill the grid with numbers such that all Sudoku constraints are satisfied.

Numerous Sudoku solving software and algorithms have been posed. A popular approach to implement a Sudoku solver takes Knuth's *Dancing Link* technique to implement *Algorithm D* (Knuth, 2018), which is a nondeterministic procedure to find all

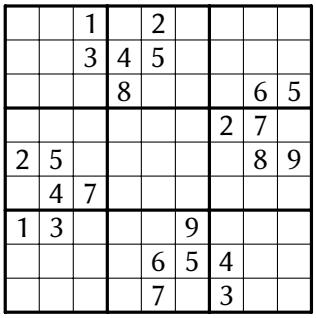


Figure 1.3: Classic Sudoku Puzzle 8 from WSC 2016 (no solutions)

solutions to the Exact Cover problem (Garey and D. S. Johnson, 1979). Norvig (2006) describes further methods based on constraint propagation and search. All implementations have in common that they use backtracking search to deal with the inherent nondeterminism of Sudoku. In fact, the generalized $n \times n$ grid Sudoku problem has been shown to be NP-complete (Yato and Seta, 2003), and this suggests that the approaches described before are a natural way to implement a Sudoku solver.

To show the merits of Answer Set Programming, we will provide an implementation based on Gringo (Gebser et al., 2014c) and Clasp (Gebser et al., 2012), two programs used to compute the answer set of a nonmonotonic logic program. Compared to an imperative implementation for Sudoku, which usually takes a few hundreds lines of code, the declarative ASP version is almost atomic in scale, yet powerful and full-featured. It consists of the four rules as given in Listing 1.1. The encoding uses many syntactic shortcuts supported by Gringo that makes it easier to write answer set programs. The formal definitions definitions for logic programs will be given in Chapter 2, but for the purpose of explaining the program above, we do not go into details here. Answer set programs consist of rules of the form

```
HEAD :- BODY1, ..., BODYn.
```

where `HEAD`, `BODY1`, ..., `BODYn` are atomic expressions. A rule should be read as `HEAD` *is satisfied if* `BODY1` *through* `BODYn` *is satisfied*; whenever `HEAD` is void, we call the rule a constraint that would forbid solutions whenever `BODY1` through `BODYn` are true. Atomic expressions are built from variables (which start with uppercase letters), constants (which start with lowercase letters or digits), and functions and relations such as addition and equality, respectively. Listing 1.1 mentions variables R, C, D, Q, B and constants $1, 2, 4, 7, 9$, the binary function $+$, and the binary relations $=, >$ and ternary relation *cell*.

The first rule of Listing 1.1 from line 4 through line 5 encodes all possible ways to fill all nine 3×3 sub-grids by using a *choice rule*. E.g., when producing a solution to WSC Puzzle 3 from Figure 1.1a an ASP solver would satisfy atoms $cell(1, 4, 2)$ and $cell(7, 3, 7)$, i.e., $cell(1, 4, 2)$ encodes that the cell specified by row 1 and column 4 holds

```
% (1) place exactly one digit D ∈ {1,...,9} for each cell identified by
% row R and column C on a 9×9 grid such that D is within a 3×3
% subgrid spanning (Q,B) to (Q+2,B+2), for Q,B ∈ {1,4,7}
{ cell(R,C,D) : R = Q..Q+2,  C = B..B+2 } = 1 :- D = 1..9,
                    Q = (1;4;7), B = (1;4;7).

% (2) cell (R,C) may contain at most one digit D ∈ {1,...,9}
:- R = 1..9, C = 1..9, { cell(R,C,1..9) } > 1.

% (3) a digit D in row R may be assigned in at most one column C
:- D = 1..9, R = 1..9, { cell(R,1..9,D) } > 1.

% (4) a digit D in column C may be assigned in at most one row R
:- D = 1..9, C = 1..9, { cell(1..9,C,D) } > 1.
```

Listing 1.1: ASP Sudoku Solver

digit 2, while atom $cell(7,3,7)$ states that row 7 and column 3 holds 7. Intuitively, the head

```
{ cell(R,C,D) : R = Q..Q+2,  C = B..B+2 } = 1
```

of the first rule expresses that the cardinality of the set of all atoms $cell(R,C,D)$ satisfying the constraints on the variables R,C after the colon must be exactly 1. Note that the semantics of above expression is similar to list comprehensions found in other programming languages and set-builder notation in mathematics. The exact range for the variables R,C,D,Q,B are given by the three body atoms `D = 1..9` (D is a number in the range 1 to 9), `Q = (1;4;7)` and `B = (1;4;7)` (there are three possible numbers $1,4,7$ for Q,B), as well as the two atoms

```
R = Q..Q+2,  C = B..B+2
```

from the choice construct in the head, which represents the sub-grid beginning at row Q and column B.

Typically, answer set solvers evaluate *ground* programs, i.e., programs whose variables have been replaced by constants. Grounders such as Gringo perform this task of intelligently replacing variables by constant symbols in all relevant ways. In the program above the first rule alone would generate 81 ground instances (nine possible values for D and three possible values each for Q and for B). E.g., one of the ground rules is the choice rule

```
{ cell(1,4,2); cell(1,5,2); cell(1,6,2);
  cell(2,4,2); cell(2,5,2); cell(2,6,2);
  cell(3,4,2); cell(3,5,2); cell(3,6,2)  } = 1.
```

that has been instantiated from the body whose variables are set to $D = 2$, $Q = 1$, and $B = 4$.

```
1 cell(1,2,1). cell(1,6,9). cell(1,7,7).
2 cell(2,1,2). cell(2,2,3). cell(2,3,4). cell(2,7,8).
3 cell(3,2,5). cell(3,8,2). cell(3,9,4).
4 cell(4,5,3). cell(4,9,7).
5 cell(5,4,4). cell(5,5,5). cell(5,6,6).
6 cell(6,1,9). cell(6,5,7).
7 cell(7,1,3). cell(7,2,9). cell(7,8,5).
8 cell(8,3,1). cell(8,7,6). cell(8,8,7). cell(8,9,8).
9 cell(9,3,6). cell(9,4,1). cell(9,8,9).
```

Listing 1.2: ASP Sudoku instance encoding WCS Puzzle 3 from Figure 1.1a

There are nine possible ways to pick exactly one atom from the set above, but only one of them can be correct. Together with the other 80 ground rules instantiated by the first rule, choosing a number placement produces a combinatorial explosion, as each of those rules require to pick exactly one atom. But for WCS Puzzle 3, there is only one solution that is consistent with the rules of Sudoku, thus further constraints are required to find that particular solution. Indeed, the last three rules in Listing 1.1 are constraints that give us the desired solution. The *counting aggregate*

```
{ cell(R,C,1..9) } > 1
```

in the body of the second rule in line 8 evaluates to true whenever for given R and C in the range $1, \ldots, 9$ as specified by the two body atoms

```
R = 1..9, C = 1..9
```

there are at least two atoms satisfied in the set $\{cell(R, C, 1), \ldots, cell(R, C, 9)\}$. As an example, after grounding one instance for the second rule would be

```
:- { cell(1,4,1); cell(1,4,2); cell(1,4,3);
     cell(1,4,4); cell(1,4,5); cell(1,4,6);
     cell(1,4,7); cell(1,4,8); cell(1,4,9) } > 1.
```

which imposes a constraint for the cell in row $R = 1$ and column $C = 4$; again, we would have 81 possible ground rules for the second rule. If other instantiations of the first rule would try to generate a solution that satisfies $cell(1, 4, 2)$ and $cell(1, 4, 3)$ simultaneously (i.e., the numbers 2 and 3 occur in the cell in row 1 and column 4), then the above counting aggregate would be true, and thus the constraint would invalidate the try to get a filled grid, as such an assignment would violate the rules of Sudoku.

Similarly, the other two constraints in line 11 and 14 would specify the regulations that each of the rows and each of the columns in the grid must not hold the same number more than once. As one can see, four ASP rules are sufficient to specify the Sudoku regulations, but how can we bring our Sudoku encoding to solve a particular problem instance? This is particularly easy in ASP, one needs to simply list all initial

```
% gringo sudoku.lp wcs3.lp | clasp
clasp version 3.3.3
Reading from stdin
Solving...
Answer: 1
cell(1,2,1) cell(1,6,9) cell(1,7,7) cell(2,1,2) cell(2,2,3) ↩
 cell(2,3,4) cell(2,7,8) cell(3,2,5) cell(3,8,2) cell(3,9,4) ↩
 cell(4,5,3) cell(4,9,7) cell(5,4,4) cell(5,5,5) cell(5,6,6) ↩
 cell(6,1,9) cell(6,5,7) cell(7,1,3) cell(7,2,9) cell(7,8,5) ↩
 cell(8,3,1) cell(8,7,6) cell(8,8,7) cell(8,9,8) cell(9,3,6) ↩
 cell(9,4,1) cell(9,8,9) cell(1,1,6) cell(1,3,8) cell(1,4,2) ↩
 cell(1,5,4) cell(1,8,3) cell(1,9,5) cell(2,4,7) cell(2,5,1) ↩
 cell(2,6,5) cell(2,8,6) cell(2,9,9) cell(3,1,7) cell(3,3,9) ↩
 cell(3,4,3) cell(3,5,6) cell(3,6,8) cell(3,7,1) cell(4,1,8) ↩
 cell(4,2,6) cell(4,3,2) cell(4,4,9) cell(4,6,1) cell(4,7,5) ↩
 cell(4,8,4) cell(5,1,1) cell(5,2,7) cell(5,3,3) cell(5,7,9) ↩
 cell(5,8,8) cell(5,9,2) cell(6,2,4) cell(6,3,5) cell(6,4,8) ↩
 cell(6,6,2) cell(6,7,3) cell(6,8,1) cell(6,9,6) cell(7,3,7) ↩
 cell(7,4,6) cell(7,5,8) cell(7,6,4) cell(7,7,2) cell(7,9,1) ↩
 cell(8,1,4) cell(8,2,2) cell(8,4,5) cell(8,5,9) cell(8,6,3) ↩
 cell(9,1,5) cell(9,2,8) cell(9,5,2) cell(9,6,7) cell(9,7,4) cell(9,9,3)
SATISFIABLE

Models       : 1
Calls        : 1
Time         : 0.007s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

Listing 1.3: Gringo-Clasp Solving Pipeline

cells with their numbers as *facts*, i.e., rules having a ground atom in the head, but whose body is void. For WCS Puzzle 3, this is encoded by the list of facts in Listing 1.2.

If we append Listing 1.1 (as `sudoku.lp`) to Listing 1.2 (as `wcs3.lp`) as input for the Gringo grounder, thus producing a ground program without variables, and then feed the result into the Clasp answer set solver, we would get the desired solution. Invoking Gringo and Clasp on the command line then results in the output shown in Listing 1.3. This particular encoding for Sudoku shows the Answer Set Programming paradigm in action (Janhunen and Niemelä, 2016) and can be used as a blueprint for practical implementations that solve search and optimization problems. Conceptually, Figure 1.4 shows a model for programming with this paradigm. In the left box, users first need to provide a formalization of the problem statement in ASP, and a concrete problem instance (usually as a set of facts) that can be extracted from instance data. Then, by running a *grounder*, we obtain a ground program without variables that encodes our problem instance (middle box). Next, we then try to find a solution to our problem instance by running a *solver*, which tries to search for answer sets of the ground pro-

Figure 1.4: Answer Set Programming Paradigm

gram instance (right box). The answer sets are a representation of the actual solutions, which can be extracted by transforming them into the proper solution format. If there is no answer set for the ground program instance, then there are no solutions to the problem instance.

The strategy outlined above hints already on a particular weak point of programming in ASP. The ASP paradigm does not aid in reusing answer set programs *per se*. Unlike any other programming language, there is no easy way to link a ready-made program to a new and larger program, as every ASP encoding for a problem statement must be tailored from scratch, possibly by manually assembling individual code fragments. This thesis tries to remedy this particular weakness of ASP by presenting a novel solution to modularity for answer set programs. Before we show the ideas of Modular ASP, we first describe Modular Programming in general, which is common and has spurred a lot of research in software engineering.

1.1 Modular Programming

A natural way to design large software systems for solving problems is to identify easier to handle subproblems that can be solved independently from each other, and then based on this analysis to craft corresponding software components that solve the subproblems. Software elements grouped together this way form a module, and the combination of such modules then gives an implementation for the whole problem and a working software system. Software modules thus provide a key concept in Software Engineering that helps developing software artifacts.

When following techniques based on structured design (Yourdon and Constantine, 1979), software elements building a module that are arranged based on relatedness of functionality have a tendency to be more reliable and robust, and are easier to reuse. Software design thus prefers systems with a high degree of *cohesion*, a software metric that reflects the relationship between functionally related software components. A software quality metric in contrast to cohesion is *coupling*, which is a measure for the interdependence of modules in software. Naturally, loose coupling is preferred over tight coupling, as loosely coupled modules tend to have increased reliability and robustness. Thus, software with high cohesion and low coupling is said to have higher structural quality. Testing software greatly benefits from structured programs, since it involves defining well-suited interfaces to the components, which in turn assists

writing test cases. When many programmers are working on a software project, the strict component-wise software design is the only way to success.

Two questions arise from these findings:

1. How to shape software with these desired traits?

2. Is there built-in support in programming languages that assists constructing modular software?

We will touch upon possible answers to these questions in the following paragraphs.

The first question is usually answered by creating software through careful architecture and design using sophisticated software engineering methodologies like Inversion of Control (R. E. Johnson and Foote, 1988) and dependency injection (Fowler, 2004), Aspect-Oriented Software Development (Murphy et al., 2001), or designing systems based on Service-Oriented Architecture (Papazoglou and van den Heuvel, 2007) and microservices (Pautasso et al., 2017a,b). Guidelines and examples for architectural patterns like the ones mentioned here are described by Bass et al. (2013), Lakos (2016), and Martin (2018), which give an in-detail account on software architecture and design.

The aforementioned strategies for building large software systems are state-of-the-art, but they come at a cost of increased complexity and potentially turning software applications into distributed systems, which adds further complexity to the runtime dynamics and may be disadvantageous when looking for reliability and robustness in software. That is, if we modularize software systems by building programs that communicate over computer networks, we must be aware of the following result, which demonstrates that designing usable and flexible distributed systems is a challenging task: Gilbert and Lynch (2002) show with their CAP Theorem that it is impossible to achieve three highly desirable properties in a distributed software system all at once, namely ***C****onsistency* (e.g, a read request must return the data that a preceding write request has written), ***A****vailability* (the system must always respond to a request), and ***P****artition-tolerance* (temporary disconnected networks must not result in a failing system). We can only choose at most two out of the three properties, i.e., CA, CP, and AP. Modern distributed software systems aim at achieving Availability and Partition-tolerance, but drop the Consistency property; they use techniques like *eventual consistency* (i.e., data updates over time eventually converge in the network of connected systems) as a remedy to the missing hard consistency constraint (see Brewer, 2012, for a discussion on remedial strategies).

In the context of Knowledge Representation and Logic Programming, frameworks based on Heterogeneous Nonmonotonic Multi-Context Systems (Brewka and Eiter, 2007) address issues arising in distributed computing with suitable algorithms (Dao-Tran et al., 2011, 2015) or asynchronous extensions to the equilibrium semantics (Dao-Tran and Eiter, 2017). See also the discussion in Chapter 10.

Programming languages usually address the second question by adding a module system to their core language. With such a feature, a compiler or interpreter can aid building modules already at compile-time by dependency tracking of modules and static code analysis. Most general-purpose languages have their own way to specifying modules. Techniques like information hiding, abstraction, and structured programming are well-established principles for breaking down subtasks and modules in imperative programming (Dahl et al., 1972; Liskov and Zilles, 1974; Parnas, 1972), and essentially any standard programming language has amenities that allow to define input/output interfaces to modules for easy code-reuse in implementations of possibly unrelated problems. Note that simple file inclusion mechanisms such as the C `#include` preprocessor directive do not facilitate the flexibility requirements that a module can cope with. Subtle issues such as duplicate definitions and circular `#include` dependencies may arise when the same file is included in different parts of a program.

In this thesis, we are not concerned with modularity in software design and architecture on the large scale, which is the target of the first question, rather, we aim at providing a possible answer to the second question of adding modularity features to the language level of nonmonotonic logic programming. We therefore continue to define programming paradigms, which we later use to provide showcase examples for modular programming.

1.1.1 Programming paradigms

Usually, one can broadly categorize a general-purpose programming language into two different paradigms: *imperative programming* and *declarative programming*. The imperative paradigm features programming languages whose expressions may have side-effects, i.e., calling a function with the same input values twice may lead to different results or errors, depending on the state of, e.g., global variables, the operating system, or programs running on a different computer. Such functions are said to be *referentially opaque*. In contrast, the declarative programming paradigm mostly forbids side-effects in their expressions, therefore calling functions in such languages behave in the mathematical sense, where the same input always produces the same result. Functions that behave in the mathematical sense are called *referentially transparent* or *pure*. A more in-detail presentation of programming paradigms is given by Van Roy and Haridi (2004), which provides further programming language classification criteria and their definitions.

Since we are concerned here with logic-based programming, we will classify programming languages into the following three broad categories:

- imperative programming languages,
- functional programming languages, and

- logic-based programming languages.

That is, we split declarative languages into functional and logic-based programming languages to make the distinction between their underlying model of computation explicit: functions form the basis for functional languages, while relations are the basis for logic programming languages (Hudak, 1989).

Programming languages of the imperative kind include procedural languages such as C, and object-oriented languages such as C++, Java, C#, etc. Both procedural and object-oriented languages are kept in the imperative paradigm because they both satisfy the property of having implicit states and side-effects in their expressions. In the functional programming realm, we include declarative languages such as Erlang, Haskell, or ML, who by and large disregard side-effects in their expressions (for concepts and historical evolution of functional programming see Hudak, 1989). For logic-based programming languages, we include declarative languages such as Prolog (Apt, 1990; Shapiro and Sterling, 1994) and Answer Set Programs (Marek and Truszczyński, 1999; Niemelä, 1999).

DeRemer and Kron (1975) identified the need for a formal specification language to describe the usage relationships between individual modules, a *module interconnection language* that allows to describe and integrate smaller software components for building a whole system. The authors distinguished *Programming-in-the-Large* and *Programming-in-the-Small*, the former being a method to design and specify the global architecture of a software system comprising of interconnected modules, and the latter a way to express and implement modules in a programming language. In a general-purpose programming language—and depending on the programming paradigm—a module thus can be viewed as a container artifact that consists of basic building blocks such as functions, variables, abstract data types, classes and their methods, class hierarchies and interfaces, and other concepts.

On the other hand, formal languages aiding Programming-in-the-Large to describe a complete software system as a portfolio of modules is less common. While well-established imperative programming languages such as C, C++, C#, or Java (below version 9) only have the means to specify individual modules in their core language, functional languages usually do have more versatile module systems built-in that allow to describe the connections between modules. This may be explained by the formal nature of functional programming, which makes it easier to clearly separate functionality and reason about individual components. This should come as no real surprise, as it is more natural to provide the formal means for modular programming in programming languages rooted in the lambda calculus, a formal system in mathematical logic. Working groups currently take remedial actions and specify standards and proposals for module systems in C++ (see Reis et al., 2016, which may be incorporated in the next C++20 standard) and the recent release of the Java 9 platform (Parlog, 2018; Reinhold, 2015, 2016), owing to the great demand for formal module systems in those imperative

languages. Before Java 9, the OSGi Alliance defined a module system on top of the Java core language (OSGi Alliance, 2014), which uses *bundles* as modular entities.

Before we take a look on the situation of modular programming in logic-based programming languages, we will next show how we can specify and interconnect modules in imperative and functional programming languages.

1.1.2 An Illustrative Example

We will now provide our running example that will be used throughout this thesis to illustrate modularity concepts, in particular the most general use of modules, i.e., modules that mutually refer to each other. As written in the C++ Standardization Committee technical report *A Module System for C++ (Revision 4)*:

> However, classes—and in general, most abstraction facilities—in real world programs don't necessarily maintain acyclic use relationship. When that happens, the cycle is typically "broken" by a forward declaration usually contained in one of the (sub)components. In a module world that situation needs scrutiny. (Reis et al. (2016))

For this purpose, we take the Even property of integers (or parity) as a well-known example in the literature to illustrate mutual recursion. A recursive definition for the set of *even* (respectively, *odd*) natural numbers is as follows. Given a natural number n, we define

1. $n = 0$ is even;
2. n is even, if its predecessor $n - 1$ is odd; and
3. n is odd if its predecessor $n - 1$ is even.

In the following, we will implement this property in various programming languages and paradigms, showing the use of modules and module systems.

1.1.3 Even in Imperative Languages

The C programming language (Kernighan and Ritchie, 1996) is the prime example for imperative programming and also one of the most wide-spread programming languages currently in use. The current C and C++ standards do not specify a formal way to express modules as first-class citizens, in fact, they do not have a module system built into their core language (see Reis et al., 2016, for the current working draft of a C++ module system). We therefore identify C modules as C functions that can be called from other C functions.

```c
/* forward declarations */
bool is_odd(int);
bool is_even(int);

bool is_odd(int N) {
  if (N == 0) return false;
  return is_even(N - 1);
}

bool is_even(int N) {
  if (N == 0) return true;
  return is_odd(N - 1);
}
```

Listing 1.4: Even in C

In order to implement the Even property with mutual recursion, one can define two functions `is_odd` and `is_even` that take an integer as input and return `bool` as presented in Listing 1.4. Note that the first two statements are necessary to declare the functions `is_odd(int)` and functions `is_even(int)` before their first application. This is due to the mutual usage in both functions, and allows to "break" the cycle between them (see also the quotation above from Reis et al. (2016)).

Now take, as an example, the integer `42`. In order to check whether `42` is an even number, we call `is_even(42)`, which in turn calls `is_odd(41)`. Then, `is_even(40)` will be called and this continues as before until we eventually reach the last call `is_even(0)` in this chain of mutually recursive calls, which will give us `true`. Thus, we get a sequence of alternating parities `is_odd(1)`, `is_even(2)`, ..., `is_odd(41)`, and finally `is_even(42)`, which all return `true`. This will also give us the final result that the integer `42` is indeed an even number.

Note that above evaluation strategy will not work for large integers `N`. Every invocation of `is_odd(int)` and `is_even(int)` in our call chain creates a fresh stack frame in the call stack, which for large `N` grows out of proportion. Thus, we will be unlucky and get a stack overflow when evaluating those functions for values of `N` larger than roughly `300000` on modern hardware, thus the program will crash with a segmentation fault. On closer inspection of Listing 1.4, we can realize that `is_odd(int)` and `is_even(int)` are *tail-recursive* functions, i.e., their final step is a function call that leads to a call in the call chain that will call itself again. Modern C compilers implement an optimization technique called *tail-call elimination*, which removes the need to adding a new stack frame to the call stack for every invocation of tail-recursive functions. For instance, C compilers such as GCC (`https://gcc.gnu.org/`) and Clang (`https://clang.llvm.org/`) accept the command-line options `-O -foptimize-sibling-calls` that turn on this particular optimization, and the compiled program will then be able

```
template<int N>
struct Even {
  static const bool is_even;
};

template<int N>
struct Odd {
  static const bool is_odd;
};

template <>
const bool Even<0>::is_even = true;

template <>
const bool Odd<0>::is_odd = false;

template<int N>
const bool Odd<N>::is_odd = Even<N - 1>::is_even;

template<int N>
const bool Even<N>::is_even = Odd<N - 1>::is_odd;
```

Listing 1.5: Even in C++

to evaluate the full range of integers.

Of course, implementing Even through mutual recursion serves here only to illustrate the concept. The classical way to check whether an integer `N` is even uses modular arithmetic modulo 2: the simple closed-form expression

```
N % 2
```
(1.1)

in the C programming language evaluates to `0` if and only if `N` is even. Alternatively, one can check whether the least-significant bit of an integer is set, and in this case that integer is odd. These constant time procedures are clearly favorable to evaluating a recursive procedure. But in contrast to (1.1), the mutually recursive strategy outlined above in the first paragraph is a more natural way to express the Even problem.

A further instance of the imperative programming paradigm is the C++ programming language (Stroustrup, 2013), an object-oriented variant of C that has support for various concepts to split functionality into components. Reis et al. (2016) note that the current C++ standard lacks direct language support for modules. Nevertheless, we may identify language constructs that help programmers to componentize their source code. Some of these constructs support compile-time modularity, i.e., “modules” will be instantiated at the compilation step of the program build process, and some of these concepts come into play when the built program is executed. The concept of template

```erlang
-module(odd).
-export([is_odd/1]).

is_odd(N) when N > 0 ->
  even:is_even(N - 1);

is_odd(0) ->
  false.
```

Listing 1.6: Even in Erlang (odd.erl)

```erlang
-module(even).
-export([is_even/1]).

is_even(N) when N > 0 ->
  odd:is_odd(N - 1);

is_even(0) ->
  true.
```

Listing 1.7: Even in Erlang (even.erl)

meta-programming (Alexandrescu, 2001) clearly belongs to the former class of structuring functionality into modules, and has been in widespread use in other programming languages as well.

In the following example given by the C++ program in Listing 1.5, we take a closer look into template meta-programming and exemplify modularity concepts of C++. Listing 1.5 rephrases our mutually recursive C program from Listing 1.4 above and computes the property of integers being even or odd at compile-time.

Now, when we want to know whether the integer `42` is an even number, we can instantiate the templates with parameter `N` set to `42` and access the Boolean member `Even<42>::is_even`, which will immediately evaluate to the constant value **`true`** when we run our program; when compiling our program, the C++ compiler will evaluate the mutual recursion for us and generate an instantiated template along with the constant member. Note that for large integer template parameters `N`, we will not be able to compile Listing 1.5. The C++11 standard limits the depth of template class instantiation to 1024, hence for template parameters `N` larger than 1024 we will not be able to compile the program.

1.1.4 Even in Functional Languages

Virtually all languages that belong to the functional programming paradigm have a module system built-in, but only some of these languages support cyclic dependencies

```
-module(odd,[N]).
-export([is_odd/0]).

is_odd() when N > 0 ->
  EvenMod = even:new(N - 1),
  EvenMod:is_even();

is_odd() when N =:= 0 ->
  false.
```

Listing 1.8: Even in Erlang with Parameterized Modules (odd.erl)

```
-module(even,[N]).
-export([is_even/0]).

is_even() when N > 0 ->
  OddMod = odd:new(N - 1),
  OddMod:is_odd();

is_even() when N =:= 0 ->
  true.
```

Listing 1.9: Even in Erlang with Parameterized Modules (even.erl)

between modules. Other languages theoretically allow dependencies, but you must specify directives that tell the compiler how to break cycles between modules.

One member of this branch of programming languages with module system that supports cyclic dependencies is Erlang (Armstrong, 2003, 2007). We will now recast our running example written in imperative languages above in the functional programming language Erlang. This time, albeit not necessary, we use two modules to emphasize mutual recursion over two modules in Listings 1.6 and 1.7.

The two listings above form essentially the same procedure as our C program from Listing 1.4, but this time using two modules `odd` and `even` to group related functionality in a named context. The evaluation will proceed similarly to the C version of the program.

Note that both functions `odd:is_odd/1` and `even:is_even/1` are tail-recursive, and since recursion is omnipresent in functional programming language, we can rely on tail-call elimination being always applied during program compilation, which in turn prevents the unlimited growth of the call stack during execution.

Parameterized Erlang modules (R. Carlsson, 2003) are an Erlang extension that is suitable for writing concise functional modules. We make use of this extension next in Listings 1.8 and 1.9 by adapting the respective Erlang modules from Listings 1.6 and 1.7 and parameterize both modules with an input parameter `N`, which shares some

similarity to the C++ template meta-programming example from Listing 1.5.

The argument `N` to functions `even:is_even/1` and `odd:is_odd/1` from Listings 1.6 and 1.7 have been shifted into parameter `N` for the modules `even,[N]` and `odd,[N]`. The functions `even:is_even/0` and `odd:is_odd/0` from Listings 1.8 and 1.9 now do not take arguments anymore. Instead of directly taking the integer `N` as input argument to compute the even and the odd property, both functions use parameter `N` from their respective module parameter `even,[N]` and `odd,[N]`, and make `even:is_even/0` and `odd:is_odd/0` applicable based on the evaluation of the function guards `when N > 0` and `when N =:= 0` (note that Erlang uses =:= to test for equality).

During program evaluation, `odd:is_odd/0` and `even:is_even/0` both explicitly instantiate a new module `even,[N-1]` and `odd,[N-1]`, respectively, using the module constructors `even:new(N-1)` and `odd:new(N-1)`. When binding the new modules to variables `EvenMod` and `OddMod`, they call `even:is_even/0` respectively `odd:is_odd/0` on that fresh instance, and use the return value as result.

If we take `N` to be `42` again, we can start the mutual recursion by calling `M = even:new(42), M:is_even()`, which in turn invokes `even:is_even/0` respectively `odd:is_odd/0` along a chain of instantiated modules `even,[42]`, `odd,[41]`, ..., `odd,[1]`, `even,[0]`. Again, the computation yields `true`, as expected.

Parameterizing modules have a long tradition in functional programming, and in fact, this model for modularization is prevalent. For example, such an abstraction mechanism is used in the following functional programming languages: in the R language through parameterized modules (Warnholz, 2017), and in ML (Milner et al., 1997), Objective Caml (Leroy et al., 2017), and Haskell (Shields and Jones, 2002) through the use of functors. Building on this idea seems to be a useful abstraction for modularity in answer set programming, but before we delve into this realm, we first provide a historic account on modular logic programming.

1.2 Modularity in Logic Programming

There is a long history of research in investigating modularity principles in logic programming. A good overview is provided by Brogi et al. (1994) and Bugliesi et al. (1994), which study modularity in the context of traditional definite Horn logic programming. In spirit of DeRemer and Kron (1975), the articles by Brogi et al. (1994) and Bugliesi et al. (1994) identify two directions for investigating modularity aspects in logic programming:

- *Programming-in-the-Large*, which introduces compositional operators to combine separate and independent modules; and
- *Programming-in-the-Small*, which builds upon abstraction and scoping mechanisms.

Early influential work on modularity in logic programming include Fitting (1987) and Gaifman and Shapiro (1989), where the former can be seen as an approach for Programming-in-the-Small, while the latter is a prototypical instance of Programming-in-the-Large.

For the Prolog programming language, there is a number of working implementations with module systems that allow to write Prolog programs with modules. The ISO Prolog standard (ISO-Prolog, 2000) standardizes the ISO-Prolog module system. Implementations of the Prolog like Ciao Prolog (Cabeza and Hermenegildo, 2000), SICStus Prolog (M. Carlsson and Mildner, 2012), SWI-Prolog (Wielemaker et al., 2012), or XSB Prolog (Swift and Warren, 2012) have a module system as part of their core language.

In contrast to the examples in §1.1 shown above, it is customary to view answer set programs as monolithic entities, i.e., one program is tailored to solve a particular problem without a clear separation of the subtasks, albeit the same principle of creating manageable pieces of knowledge will help users of ASP systems building knowledge bases. Having an explicit way to modularize knowledge in logic programs is thus needed and adding modularity principles to ASP has several advantages like easy knowledge base reuse by clean input/output interfaces and helping to model complex problem domains by focusing on smaller parts first. This issue has been identified and various notions for modularizing logic programs have been proposed to support testing logic programs, reusing and abstracting components, and maintaining program code.

However, there are obstacles that impede to bring such characteristics to ASP. Traditional answer set semantics has no module concept and there is no straightforward way that would allow that. It is not clear how a semantics should be defined that caters for modules, as the declarative nature of ASP does not distinguish between knowledge stored in different logic programs (when viewed as modules). Another issue is to allow for cyclic module systems, i.e., when modules mutually refer to each other. Modules that have such cyclic dependencies may bring in semantic issues like unfounded models that would not be present when viewing logic programs as single units. Both of these problems are related to the declarative nature of ASP, and any prospective model-theoretic semantics for modular ASP has to deal with unwanted semantic deficits. Methods that bring modularity aspects closer to ASP have not yet stood the test of time, and no single semantics has gained general acceptance.

Thus, there has been an increasing interest in studying modularity aspects of Answer Set Programming in the recent years, in order to ease the composition of program parts to an overall program. Since the early days of Datalog (Gottlob et al., 1989), modularity aspects have been recognized as an important issue, and already the seminal notion of stratification (Apt et al., 1988) builds on an evaluation of subprograms in an ordered way. This has been later largely elaborated to notions like modular stratification (Ross, 1994) and XY-stratification incorporated in the $\mathcal{LDL}$++ system (Arni et

al., 2003), and has been generalized to a syntactic notions of modularity for disjunctive Datalog programs (Eiter et al., 1994, 1997a,c) that, in the context of nonmonotonic logic programming, has been independently found as Splitting Sets (Lifschitz and Turner, 1994), which generalize stratification and proved to be a useful tool to decompose programs.

However, compared to the study of modularity in logic programming (Brogi et al., 1994; Bugliesi et al., 1994) (see Eiter et al. (1997b) for a historic account), work on modular ASP is still less developed. In the context of answer set semantics, whose focus lies in the treatment of negation-as-failure and disjunctive rules, several important proposals have been put forward.

Representatives of Programming-in-the-Large provide compositional operators for combining separate and independent modules based on standard semantics. This direction for modular logic programming has been followed in DLP-functions (Janhunen et al., 2009b) and modular SMODELS programs (Oikarinen and Janhunen, 2008), which focus on logic programs with Gaifman-Shapiro-style module architecture (Gaifman and Shapiro, 1989). Additional work generalizes their approach to a module-based framework for multi-language constraint modeling (Järvisalo et al., 2009) and to modular P-log programs that combines probabilistic reasoning with logic programs (Damásio and Moura, 2011). Recently, (Oikarinen and Janhunen, 2008) has been extended to lift syntactic restrictions on the dependencies between modules (Moura and Damásio, 2014, 2015), which allows to express positive cycles between modules by introducing fresh modules and rewriting output atoms. The work on abstract modular systems (Lierler and Truszczyński, 2016) studies general modular knowledge representation systems. Another proponent (Vennekens et al., 2006) is concerned with operator splitting similar in the vein of splitting sets (Lifschitz and Turner, 1994).

Programming-in-the-Small aims at enhancing ASP with abstraction and scoping mechanisms similar as in other programming paradigms. This direction has been widely considered, and modular extensions of answer set programs based on generalized quantifiers (Eiter et al., 1997b), macros (Baral et al., 2006), templates (Calimeri and Ianni, 2006), and for web rule bases (Analyti et al., 2011) have been proposed. On a broader scale, multi-agent scenarios with logic programs have been studied in social logic programs (Buccafurri and Caminiti, 2008) and communicating ASP (Bauters et al., 2011).

The two directions Programming-in-the-Large and Programming-in-the-Small are quite divergent, as Programming-in-the-Large requires to introduce new operators in the language. However the above concepts do not cater a module concept as familiar in conventional imperative and object-oriented languages, where procedures come with parameters that are passed on during the evaluation. To provide support for this, Eiter et al. (1997b) developed Modular ASP Programs, which are an early attempt to narrow the gap between imperative and declarative languages a bit. Such modular

(a) Imperative Programming (C-style)

- Declaration:

```
int fun(int, int);
```

- Definition:

```
int fun(int x, int y) {
  return [...];
}
```

- Use:

```
int z = fun(1,2);
```

(b) Functional Programming (Erlang-style)

- Declaration:

```
-spec fun(integer(),integer())
    -> integer().
```

- Definition:

```
fun(X,Y) -> [...].
```

- Use:

```
Z = fun(1,2).
```

(c) Modular Nonmonotonic Logic Programming

- Declaration:

  $m = (fun[p, q], R)$

  - fun is a module name
  - p, q are predicate names
  - R is a set of rules

- Definition:

  $R = \{o(X) \leftarrow p(X), q(X, Y), \cdots ; \ldots\}$

- Use:

  $z(X) \leftarrow fun[r, s].o(X)$

Figure 1.5: Modular Structuring in three Programming Language Paradigms

logic programs are based on an extension of logic programs with genuine generalized quantifiers, where modules can receive parametric input that is passed on in a *call by value* mode, in addition to the usual *call by reference* access to atoms in other modules. Strachey (2000) defines these two parameter calling modes as follows: a function which receives a parameter in call by value mode will receive the content (value) of a memory area as formal parameter, whereas a function with call by reference mode will receive the location of an area in memory as formal parameter; e.g., in C++, we may define a function `int f(int& x)` that takes a formal parameter `x` by reference, and calling `f(y)` allows `f` to change the content of `y`.

Figure 1.5 shows three functions expressed in different programming paradigms,

but all have in common that they receive their parameters in call by value mode. Figures 1.5a and 1.5b shows how to define, declare, and use call functions in C and Erlang, respectively. Both are essentially identical with only minor syntactic differences: the outcome of a function call `fun(1,2)` will be assigned to a variable `z`, where `fun` receives the values `1,2` as `x,y` in its function definition. Figure 1.5c shows the module of a Modular ASP program in comparison. Here, the formal input parameters p and q for module *fun* will be given the extension of the predicates r and s in the rule $z(X) \leftarrow fun[r, s].o(X)$, where r and s will be computed outside the context of *fun*. This gives rise to module instantiation for different values of p and q that are in spirit of the parameterized Erlang module example shown in Listings 1.8 and 1.9 and the C++ template meta-programming example in Listing 1.5.

General quantifiers are used as a tool to access a module P_i from another module P_j using *module atoms* of the form $P_i[\mathbf{p}].q(X)$ (in slightly different syntax), where $\mathbf{p}$ is a list of predicates and q is a predicate; intuitively, the module atom evaluates to true for X if, on input of the values of the predicates in $\mathbf{p}$ to the module P_i, the atom $q(X)$ will be concluded by P_i (under skeptical semantics). For a system $P_1[\mathbf{q}_1], \ldots, P_n[\mathbf{q}_n]$ of such modules, where each $\mathbf{q}_i$ is a (list of) formal input predicates, answer sets have been defined using a generalization of the Gelfond-Lifschitz reduct. The resulting framework is quite expressive, as it is EXPSPACE-complete in general.

But there are limitations and shortcomings in the seminal approach by Eiter et al. (1997b). As for the former, an important restriction adopted by Eiter et al. (1997b) is that *calls of modules must be acyclic*; that is, following the call chain, one may not return to a call of the same module. In fact, this condition was already imposed at the syntactic level and prohibits the use of recursion, which is a common and natural programming technique. Because of technical intricacies, also some other approaches to modular answer set programming have limited recursive module calls. Most prominently, DLP-functions (Janhunen et al., 2009b), which are disjunctive logic programs with a well-defined input/output interface, exclude recursive calls that involve positive recursion. The approach of Moura and Damásio (2014, 2015) shows how to allow positive recursion over modules as specified in the framework of Oikarinen and Janhunen (2008) by using rewriting techniques for mutual recursive modules and adding fresh modules, but there is no support for disjunctive logic programs. For more discussion, we refer to Chapter 11.

Furthermore, Eiter et al. (1997b) based their semantics on the Gelfond-Lifschitz reduct, which suffers from similar anomalies as answer sets for other extensions of logic programs defined in this way. And finally, it was more concerned with defining local models of a single module, by importing conclusions of other modules (where for a given input the "output" of a module is unique) rather than giving a model-based semantics to a collection $P_1, \ldots, P_n$ of modules, in which for the same input alternative outputs of a module are possible.

As we will see, the restrictions for modular logic programs (Eiter et al., 1997b) will be lifted in this thesis, with its contributions listed in §1.5. Next, we will look into the Even property again and try to model it using an answer set program.

1.2.1 Even in Nonmonotonic Logic Programming

In logic programming with an ordinary ASP realization, assuming data types are available and, in particular, the predecessor $n-1$, the Even property is easily expressed with recursive rules

$$\begin{aligned} even(N) &\leftarrow odd(N-1) \\ odd(N) &\leftarrow even(N-1) \\ even(0) &\leftarrow \end{aligned}$$

However, if no such predecessor is available, the task is more complicated. For example, if n is given as the cardinality of a set of elements, stored in a predicate q; i.e., we need to tell whether the set of facts over q has even cardinality. This problem is known as the Even-query in databases and has been studied intensively. In fact, it is well-known that this problem cannot be expressed in Datalog, and furthermore, even not without the use of a binary predicate (Chandra and Harel, 1982). This follows from Blass et al. (1986), which shows that logics with fixed-point operators like Datalog have the *0–1 Law* (Fagin, 1976; Glebskii et al., 1969). The same is true for ASP under stable model semantics, where we have negation as failure, see Proposition 3.1.

A common solution to this problem is, in order to realize the recursion scheme above, to guess an binary predecessor predicate $pred(x, y)$ over the elements in q that amounts to $x = y-1$. This works well for smaller sets, but suffers scalability problems as building the successor predicate is expensive (see also Abiteboul and Vianu, 1991, for the mismatch between the complexity of database computation and conventional Turing complexity).

If we can order the domain linearly, and provide this information together with the minimal and maximal elements, i.e., when the relations $pred/2$, $first/1$, and $last/1$ are stored in the extensional database, we may express the query with

$$\begin{aligned} even(Y) &\leftarrow pred(X, Y), odd(X) \\ odd(Y) &\leftarrow pred(X, Y), even(X) \\ odd(X) &\leftarrow first(X) \\ w &\leftarrow last(X), even(X) \end{aligned}$$

such that w is true whenever the domain contains an even number of elements.

Modular logic programming offers an alternative to express the Even query in a fashion that avoids to build a successor predicate, and retains the simple structure of

the program above. All we need to do is to determine the predecessor of n (which is given by q) in a predicate q' and then make a recursive call for q'. To this end, it is sufficient to drop some arbitrary element from q and let q' be the result. Dropping an element from q can be easily expressed by nondeterministic choice rules in ASP. In our formalism, this would lead to the following module, which shares some familiarity to parametric modules in functional programming.

Example 1.1 (Even query) Consider the following module $Parity[q/1]$, which consists of four rules that determine whether a set has an even respectively odd number of elements:

$$\begin{aligned} q'(X) \vee q'(Y) &\leftarrow q(X), q(Y), X \neq Y \\ skip &\leftarrow q(X), \text{not } q'(X) \\ odd &\leftarrow skip, Parity[q'].even \\ even &\leftarrow \text{not } odd \end{aligned}$$

Here, $q/1$ is a (formal) unary input predicate that stores the set. The first two rules have the effect, by the minimality of answer sets, that q becomes q' with one element arbitrarily removed (for which *skip* is true, as defined in the second rule). Do if q represents n, then q' represents $n-1$. The third rule determines recursively whether q stores an odd number of elements using the *module atom* $Parity[q'].even$, while the last rule defines *even* as the complement of *odd*. Intuitively, if we call the module *Parity* with a predicate p for input, then *even* is computed true, which is expressed by $Parity[p].even$, whenever p stores an even number of elements.

Intuitively, if we call *Parity* with a predicate p for input, then *even* is computed true (which is expressed by $Parity[p].even$), if p stores an even number of elements. Note that *Parity* is recursive, and for empty input p it calls itself with the same input.

As a matter of fact, the program module above does not use a binary predicate; only a guess of a unary predicate (for removing an element) is needed. In sense, the modular ASP encoding builds a predecessor predicate locally on the fly, while an ordinary ASP encoding builds it with a global guess, which should be less efficient. This intuition is in fact confirmed by experimental results (see Chapter 8).

Roughly, an MLP is a system $\mathbf{P} = (m_1, \ldots, m_n)$ of modules, where each module $m_i = (P_i[\mathbf{q}_i], R_i)$ has a module name P_i with an associated list $\mathbf{q}_i$ of formal input atoms, and an associated set of rules R_i (the "implementation"). A module m_i can access another module m_j using *module atoms* in the body of rules in R_i of the form $P_j[\mathbf{p}].o$. Intuitively, the module atom evaluates to true if, on input of the atoms in $\mathbf{p}$ to the module P_j, the atom o will be true in P_j. Such programs allow unrestricted cyclic calls between modules; they can be seen as a generalization of DLP-functions from propositional to Datalog programs that allow for positive cyclic calls between modules (including recursion), and provide a call by value mechanism.

Example 1.2 (Even MLP) For example, the following MLP $\mathbf{P} = (m_1, m_2, m_3)$ recursively checks whether the number of facts over predicate q in the main module $m_1 = (P_1[\mathbf{q}_1], R_1)$ is even. Note that m_1 has no input (i.e., $\mathbf{q}_1$ is empty) and uses the rules

$$R_1 = \left\{ \begin{array}{rcl} q(a) & \leftarrow & \\ q(b) & \leftarrow & \\ ok & \leftarrow & P_2[q].even \end{array} \right\},$$

as implementation. Intuitively, m_1 calls m_2 with a rule for the check, and assigns the result to *ok*. The module $m_2 = (P_2[\mathbf{q}_2], R_2)$ is mutual recursive with module $m_3 = (P_3[\mathbf{q}_3], R_3)$. They have the formal inputs $\mathbf{q}_2 = q_2$ and $\mathbf{q}_3 = q_3$, respectively, and the implementations

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

and

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

A call to m_2 "returns" *even*, if either the input q_2 to m_2 is empty (as then $skip_2$ is false), or the call of m_3 with q_2' resulting from q_2 by arbitrarily removing one element (then $skip_2$ is true) returns *odd*. Module m_3 returns *odd* for input q_3, if a call to m_2 with q_3' analogously constructed from q_3 returns *even*. In any answer set of $\mathbf{P}$, *ok* is true.

1.3 Goals

As described above, several semantics exist that deal with modularity in ASP. Virtually all semantics are defined such that mutual recursion between modules is disallowed. While this helps one to simplify the definitions of a semantics for modular ASP, in general this may bring issues when different, possibly independently developed modules are combined. Many natural problems exist that have an inherent cyclic flavor, and ruling out the chance to model problems using modules that depend on each other may be too restrictive in practice, or even force to use counter-intuitive encodings. We aim at defining a model-theoretic semantics that caters for this situation, investigate its semantic properties and computational complexity, and develop novel evaluation algorithms for such modular nonmonotonic logic programs. The next example illustrates

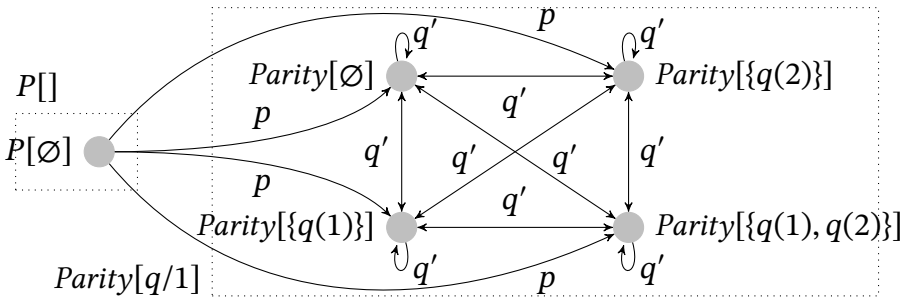


Figure 1.6: Call graph of instantiated modules in Example 1.3

cycles in modular logic programming using Modular Nonmonotonic Logic Programs (MLP) as defined in Chapter 3, a formalism that admits arbitrary nonground disjunctive nonmonotonic logic programs as modules. MLPs can be seen as a proponent of the Programming-in-the-Small approach to modular programming, as it is using module atoms as a language construct to access knowledge encoded in other modules. We sketch the basic building blocks of MLPs and refer to Chapter 3 for proper formal definitions.

Example 1.3 We demonstrate the use of *Parity* from Example 1.1 in an MLP with the (main) module $P[]$ with empty input, which calls *Parity* with a set p of two elements:

$$p(1) \leftarrow \qquad p(2) \leftarrow \qquad pev \leftarrow Parity[p].even$$

The combination of both modules gives the cyclic MLP $\mathbf{P} = (P[], Parity[q/1])$. On the surface, $\mathbf{P}$ can be seen as an "uninstantiated" modular program, whose semantics is given by characterizing models at modules which have been instantiated with a set of input facts: the *value calls*. Figure 1.6 depicts the *call graph* (the principle dependencies) of $\mathbf{P}$ with value calls as nodes and edges labeled with input predicates; e.g., value call $P[\varnothing]$ calls $Parity[\{q(1), q(2)\}]$ on input p. The dotted boxes highlight the modules from which the value calls on the inside have been generated.

Loosely speaking, MLPs encode schematic dependencies between modules, and instantiated modules then can be used to define a semantics that takes module input into account which is defined over possibly cyclic modules. Different interpretations of an MLP select different subgraphs of its call graph, and answer sets are defined based on the selected subgraphs. For instance, $\mathbf{P}$ has two answer sets in which *pev* is true at the main instantiation $P[\varnothing]$ and *even* is true at $Parity[\{q(1), q(2)\}]$ and $Parity[\varnothing]$, whereas *odd* is satisfied at $Parity[\{q(1)\}]$ and $Parity[\{q(2)\}]$. Both answer sets are symmetric on the guess of q' at $Parity[\{q(1), q(2)\}]$, but otherwise equal.

1.4 Methods

We have an advanced understanding of unexpected issues that arise when we allow for module cycles in MLPs. One key aspect is the use of the FLP-reduct (Faber et al., 2011) instead of the traditional GL-reduct (Gelfond and Lifschitz, 1991) to cure semantic issues when dealing with negation-as-failure over potential nonmonotonic module atoms. Roughly, given an interpretation of a program, the GL-reduct first removes each rule whose negative body is false in the interpretation, and then cut offs the negative literals from remaining rules. On the other hand, the FLP-reduct just removes rules whose body is unsatisfied in a given interpretation, which leaves negative literals in the result of this transformation. Applied to traditional answer set programs, both reducts are equivalent, but FLP-semantics is beneficial for language extensions of ASP such as logic programs with aggregates. In the context of MLPs, the FLP-semantics guarantees that models are minimal, thus we retain groundedness of the semantics and prohibit unfounded answer sets.

Another aspect of MLP is to contextualize module instantiation. Here, relevant instantiations are a concept to concentrate on the important part of all instantiated modules. In general, module instantiation plays a key role for the definition of a semantics for MLPs. Akin to the argument-passing semantics of imperative programming languages, the module instantiation employed in MLPs can be viewed as *call by value* mechanism, where module instantiations call other instantiations with explicit input facts, thus formal input arguments of modules cannot be changed after instantiation.

In the module framework of DLP-functions (Janhunen et al., 2009b), which can be classified as *call by reference* mechanism, input is given implicitly by the models of individual module, and composed modules may refer to input atoms as alias for atoms in the model. Here, the truth value of atoms are instantiated through the composed programs by a given model. To draw an analogy to the individual stages of ASP, MLPs instantiate modules by "grounding" and keep the instantiations fixed, while DLP-functions instantiate modules during model search.

Further results show that MLPs have an increase in computational complexity compared to standard ASP: propositional Horn-MLPs with unrestricted cyclic input over modules are EXP-complete, and nonground ones are 2EXP-complete. If we restrict propositional MLPs such that modules have no input predicates, we obtain for instance that checking satisfiability of normal propositional MLPs is NP-complete, and for disjunctive MLP it is Σ_2^p-complete. In general, checking answer set existence of arbitrary normal nonground MLPs is 2NEXP-complete, and 2NEXP^{NP}-complete for the disjunctive case.

1.5 Contributions

Concerning semantics, the use of the Gelfond-Lifschitz reduct effected that local models were in the spirit of Nash equilibria, viz., that a model is (locally) stable if assuming that all modules behave in the same way there is no need for the local program to switch to another model. Specifically, a program P_1 consisting of the clause $q \leftarrow P_2[q].p$, where the module $P_2[q_2]$ consists of the single clause $p \leftarrow q_2$, has two answer sets, viz., $\emptyset$ and $\{q\}$. The reason is that q can be concluded in a self-stabilizing way from the call $P_2[q].p$; however, arguably $\emptyset$ may be considered as the single answer set of P_1.

Such behavior can be excluded using alternative reducts, like the Faber-Leone-Pfeifer (FLP) reduct (Faber et al., 2011), which has been proposed in the context of ASP with aggregates to ensure that answer sets are minimal models (see Example 3.8 for the technical details). This reduct formed also the basis for defining the semantics of HEX-programs (Eiter et al., 2012a, 2006b), which generalized the semantics of logic programs with generalized quantifiers to the HiLog setting; however, the setting has been module-centric like (Eiter et al., 1997b), and no global semantics for a collection of modules is evident. MLPs overcome a restriction of a preliminary approach by Eiter et al. (1997b), in which module calls must be acyclic (which prohibits the use of recursion through modules), as well as anomalies of the semantics due to the Gelfond-Lifschitz reduct, which is replaced by the FLP reduct.

Motivated by these shortcomings, we reconsider modular ASP and make the following main contributions.

- We define a model theoretic semantics of a system $P_1[\mathbf{q}_1], \ldots, P_n[\mathbf{q}_n]$ of program modules, which are divided into one or multiple main modules P_i that have no input (i.e., $\mathbf{q}_i$ is void), and library modules which may have input (i.e., $\mathbf{q}_i$ can be void). Informally, the semantics assigns an answer set to each main module and module instance that is called by the program under a call by value mechanism (Eiter et al., 1997b); the answer set must be reproducible from the rules along its recursive computation.

Example 1.4 (cont'd) In Example 1.1 above, an answer set for the module instance of *Parity*$[q]$, whose input q stores $S = \{c_1, \ldots, c_n\}$, would have q' storing, for some permutation π of $\{1, \ldots, n\}$, the set $S_1 = S \setminus \{c_{\pi(1)}\}$ and call the instance of *Parity*$[q]$ with q storing S_1, whose answer set in turn stores $S_2 = S_1 \setminus \{c_{\pi(2)}\} = S \setminus \{c_{\pi(1)}, c_{\pi(2)}\}$ in q', etc. The value of *even* and *odd* in the answer sets of the instances is determined bottom up from the ground: for the instance of *Parity*$[q]$ where $q = \emptyset$, q' and *skip* are void, and thus *odd* must be necessarily false; hence, *even* is true. On the way back, *even* and *odd* are complemented with their values at the next recursion level.

While a naive definition of the semantics is straightforward, a more difficult question is to delineate the *relevant* instances of modules for the computation. Intuitively, many

(instances of) modules $P_i[\mathbf{q}_i]$ in a library might be completely irrelevant for determining the semantics of a particular collection of modules, but prevent the existence of a global semantics if locally, for some input value of $\mathbf{q}_i$, the instance has no answer set.

Example 1.5 (cont'd) Suppose in the module *Parity* in Example 1.1 there would also be a fact $r(a)$ and a rule $ok \leftarrow P'[r].nonempty$ where the module $P'[q/1]$ consists of the rules $nonempty \leftarrow \text{not } nonempty$ and $nonempty \leftarrow q(X)$. Then, an instance P' has an answer set precisely if its input is not empty. Thus, the call $P'[r].nonempty$ in the rule will always lead to an answer set in which *nonempty* is true, and hence we expect an answer set for the instance of *Parity* with input S. However, as P' has for empty input no answer set, there is no global answer set; intuitively, the instance of P' with empty input is irrelevant, and may be discarded.

To remedy this situation and to keep the semantics simple, we use here minimal models as an approximation of answer sets in module instances that are outside of a *context* (i.e., a *scope*), in which stability of models is strictly required. This context contains always at least the module instances along the call graph of the program and optionally further instances to increase in a sense the degree of stability. The smaller the context, the more permissive is the semantics. An alternative to using minimal models for ensuring consistency would, e.g., be to use paracoherent answer set semantics (Amendola et al., 2016; Sakama and Inoue, 1995); however the latter has higher computational complexity than ordinary answer set semantics.

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

- We characterize the computational complexity of the new formalism. Our modular ASP programs have the same complexity as ordinary ASP programs if the modules have no input, i.e., deciding answer set existence is Σ_2^p-complete in the propositional case and NEXP^{NP}-complete in the nonground (Datalog) case. For programs with arbitrary inputs, the complexity is exponentially higher, viz., NEXP^{NP}-complete and 2NEXP^{NP}-complete, respectively. Consequently, our formalism is (under common complexity hypothesis) more expressive than modular logic programs by Eiter et al. (1997b); the latter have EXPSPACE complexity, and EXPSPACE is believed to be strictly contained in 2NEXP^{NP}. The picture is analogous for deciding membership of an atom in the least model of a Horn program, which is P-complete for MLPs without input (respectively, EXP-complete for nonground programs), and EXP-complete for MLPs

with arbitrary input (respectively, 2EXP-complete in the Datalog setting). However, if the inputs are naturally bounded, then the complexity is the same as in the case without inputs, and thus as in ordinary ASP. We further investigate normal MLPs, both unrestricted and those which are acyclic with respect to their call graph, and show that in either case deciding whether normal or acyclic MLPs have an answer set is NEXP-complete.

- We provide two rewriting techniques for translating MLPs with module input into programs of simpler structure. The first one rewrites arbitrary MLPs to MLPs without module input, which may be transformed into logic programs without modules at all. This approach is costly in general and may generate exponentially larger programs. The second approach converts restricted MLPs to programs without modules, which will be applied to transfer Datalog-rewritable DL-programs (Heymans et al., 2010) to MLPs.

- We report a top-down evaluation procedure that expands only relevant module instantiations based on novel notions of input- and call-stratified MLPs, for which Splitting Set Theorem (Lifschitz and Turner, 1994) has been extended.

- We characterize the answer sets of MLPs in terms of classical models and explore the notion of *loop formulas* (Lin and Zhao, 2004) and *completion* (Clark, 1978) for MLPs. We further the work and study *ordered completion* exploring the recent approach of Asuncion et al. (2012).

- We analyze the relationship between our modular answer set programs and DLP-functions (Janhunen et al., 2009b), which are one of the premier formalisms for combining ASP modules. As it turns out, DLP-functions can be very naturally embedded into our formalism, by regarding DLP modules as MLP modules with empty input list; vice versa, a respective fragment of our modular ASP programs can be embedded into DLP functions. As our approach admits mutual recursion of calls with positive loops, and it furthermore also incorporates a call by value mechanism, it can be viewed as a generalization of DLP-functions with these features.

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

1.6 Organization

This thesis is organized into four parts. In Part I, we started with an introduction into modular programming and aspects of modularity in logic programming. In the next

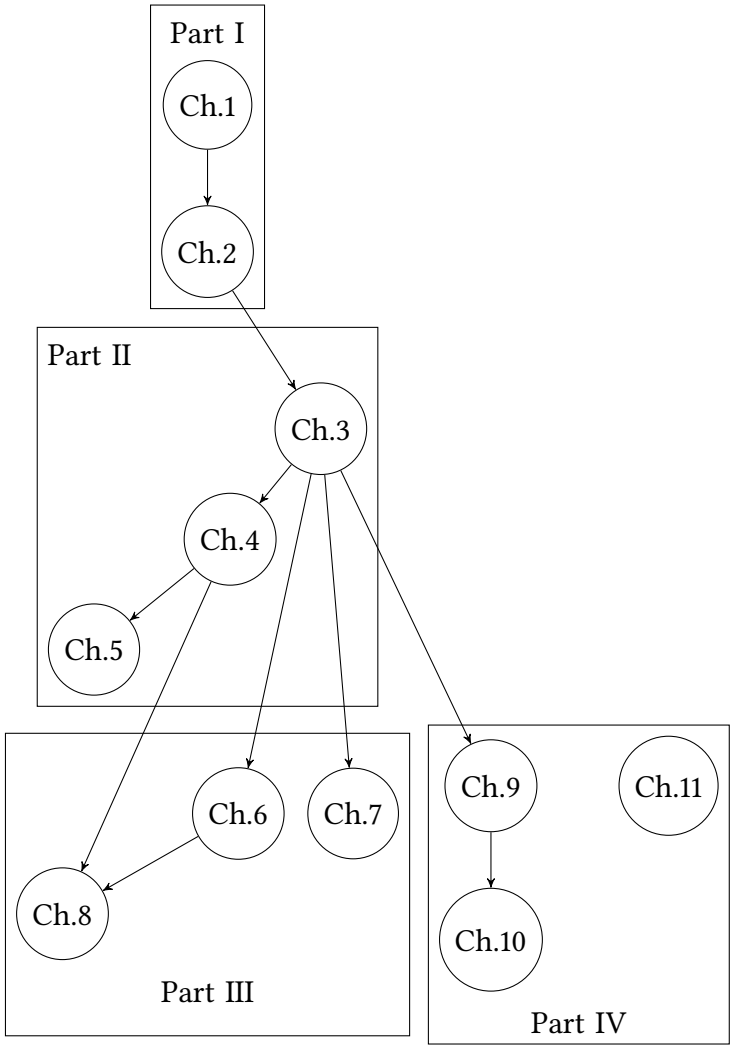


Figure 1.7: *Leitfaden*

Chapter 2 we give preliminaries for Answer Set Programming and review Generalized Quantifier Logic Programs.

The following Parts II–IV then elaborate on modular nonmonotonic logic programs (MLPs), our contribution to modularity in logic programming. In Part II, we start by introducing syntax and semantics of modular nonmonotonic logic programs in Chapter 3. We then study in Chapter 4 semantic properties and consider some important syntactic fragments of MLPs, before we proceed in Chapter 5 to analyze the computational complexity of the formalism.

The next Part III is concerned with characterizing MLPs using other logic formalisms. In Chapter 6 we investigate rewriting techniques for MLPs into programs of

simpler structure, namely into Datalog, which possibly completely removes all modules and thus potentially allows for easier program evaluation. This paves the way to show an application for MLPs, namely evaluating hybrid knowledge bases in the form of dl-programs with Datalog-rewritable description logics. Following this, Chapter 7 characterizes the semantics of MLPs in terms of classical models by adopting the notion of loop formulas and ordered completion to MLPs. In Chapter 8, we review MLP splitting sets and a top-down evaluation algorithm for MLPs. Moreover, we report the findings of an experimental evaluation for a benchmark using MLPs derived by the rewriting techniques developed for dl-programs in Chapter 6.

The final Part IV examines related work and concludes this thesis. We first establish a correspondence of a fragment of MLPs to DLP-functions (Janhunen et al., 2009b) in Chapter 9, and then provide further approaches to modular logic programming and their relationship to MLPs in Chapter 10. Chapter 11 considers potential future work, addresses possible applications and gives conclusions.

Figure 1.7 summarizes the conceptual dependencies between the chapters, where Chapter 11 implicitly depends on all chapters.

1.7 Publications Related with the Thesis

Dao-Tran et al. (2009a) devise a novel semantics for MLPs that allows for mutual recursion between modules. We have studied the semantic properties of MLPs, their computational complexity, and compared it to DLP-functions (Janhunen et al., 2009b); interestingly, DLP-functions can be seen as MLPs that have no module input parameters. MLPs conservatively extend ordinary logic programs, and many semantic properties of answer set programs generalize to MLPs. For instance, the important property that every answer set of an MLP is a minimal model implies that answer sets in the MLP setting are grounded (see discussion above). This thesis builds upon this work and gives detailed proofs and extends it in Chapters 3–5 and Chapter 9.

Eiter et al. (2009a) investigate the relationships between various semantics for modular logic programs and other nonmonotonic formalisms. We have provided a more systematic view of approaches in combining nonmonotonic knowledge bases and classified formalisms based on the program reduct and on the environment view, i.e., whether their semantics is defined in terms of local models for each individual knowledge base that implicitly converge to a semantics for the combined system, or whether the formalism has a global state using a collection of explicitly accessible local models.

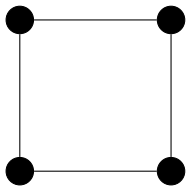
We developed a novel evaluation algorithm for MLPs (Dao-Tran et al., 2009b). Here, we concentrated on an MLP fragment called input- and call-stratified MLPs, whose stratification can be evaluated in a top-down fashion starting from uninstantiated modules. This way we could generalize the splitting set technique to MLPs and develop an

evaluation algorithm that traverses the call graph and instantiates modules on-the-fly. Example 1.3 above is input-call-stratified, and the techniques developed by Dao-Tran et al. (2009b) are applicable to it. In Chapter 8, we summarize their work there.

Krennwallner (2011) consolidates our work on Modular Nonmonotonic Logic Programs and pinpoints to issues that are present in cyclic module systems by highlighting how MLPs addresses them.

We worked on two characterizations of MLPs in terms of classical models by investigating the notions of loop formulas (Lin and Zhao, 2004) and ordered completion (Asuncion et al., 2012) in MLPs (Dao-Tran et al., 2011). The results include *modular loop formulas* based on loops over module instantiations, and *ordered completion for MLPs* without using explicit loop formulas. We generalized Clark's completion and positive dependency graph to MLPs with respect to different module instantiations. Based on these results, we defined modular loop formulas that capture MLP semantics. The second contribution was to explore ordered completion in the realm of MLPs. Here, fresh predicates ensure a derivation order, and program completion is only active for those predicates that do not participate in a positive loop, possibly involving module instantiations. Chapter 7 extends this work.

Eiter et al. (2012b) consider recent and ongoing work on combining rules and ontologies systems formalized in logic programming and description logics, respectively. Nonmonotonic description logic programs are a major formalism for a loose coupling of such combinations; this approach is attractive for combining systems, but the impedance mismatch between different reasoning engines and the API-style interfacing are an obstacle to efficient evaluation of dl-programs in general. Uniform evaluation circumvents this by transforming programs into a single formalism, which can be evaluated on a single reasoning engine. We use relational first-order logic (and thus relational database engines) and Datalog with negation as target formalisms, conducting experiments whose results show that significant performance gains are possible and suggest the potential of this approach.



2

Preliminaries and Previous Results

The objective of this chapter is to describe the formal specifications for two essential formalisms that provide the basis for this thesis. §2.1 describes the background for Answer Set Programs, i.e., disjunctive logic programs under stable model semantics. Beyond that, §2.2 describes logic programs with generalized quantifiers, which provide the theoretical underpinning for generalized quantifier modular logic programs (GQMLP), described in §2.3. Such GQMLPs constitute important previous results for modular logic programming in ASP and as such forms the intellectual predecessor for modular nonmonotonic logic programs.

2.1 Logic Programs under the Answer Set Semantics

Answer Set Programming stems from the stable model semantics of normal logic programs (Gelfond and Lifschitz, 1988) line of research (also known as *general logic programs*), which typically deals with negation as failure. This kind of negation is closely related to Reiter's *Default Logic* (Reiter, 1980), hence it is also known as *default negation* or *weak negation*. Since negation as failure is different from *classical negation* (or *strong negation*) in classical logic, Gelfond and Lifschitz proposed a logic programming approach that allows for both negations (Gelfond and Lifschitz, 1990). Subsequently, Gelfond and Lifschitz (1991) extended their semantics to disjunction in rule heads. Similar definitions for general logic programs and other classes of programs can be found in the literature (confer, e.g., (Lifschitz and Woo, 1992)). For an overview on other semantics for extended logic programs, see also (Dix, 1995).

Prominent systems for computing answer sets of are ASSAT (Lin and Zhao, 2004), Clasp (Gebser et al., 2011, 2012), Clingo (Gebser et al., 2017), Cmodels (Giunchiglia et al., 2006), DLV (Adrian et al., 2018; Leone et al., 2006), DLV2 (Alviano et al., 2017), DLVHEX (Eiter et al., 2018, 2006a, 2017), GnT (Janhunen et al., 2006), LP2* family (Janhunen, 2018), SMODELS (Niemelä, 1999; Simons et al., 2002), and WASP (Alviano et

Name	restriction
definite Horn	$k = 1, n = m$
Horn	$k \leq 1, n = m$
normal	$k \leq 1$
definite	$k \geq 1, n = m$
positive	$n = m$
disjunctive	no restriction

Table 2.1: Program classes

al., 2015a), which allow for efficient declarative problem solving. Some of these answer set solvers require ground input programs, which can be generated by sophisticated grounders like Gringo (Gebser et al., 2014c), $\mathcal{I}$-DLV (Calimeri et al., 2017), or Lparse (Syrjänen, 2001; Syrjänen, 2009).

2.1.1 Syntax of Answer Set Programs

Let $\mathcal{P}$, $\mathcal{C}$ and $\mathcal{X}$ be disjoint sets of predicate, constant, and variable symbols from a first-order vocabulary Φ, respectively, where $\mathcal{X}$ is infinite and $\mathcal{P}$ and $\mathcal{C}$ are countable. In accordance with common ASP solvers such as DLV, we assume that elements from $\mathcal{C}$ and $\mathcal{P}$ are string constants that begin with a lowercase letter or are double-quoted, where elements from $\mathcal{C}$ can also be integer numbers. Elements from $\mathcal{X}$ begin with an uppercase letter. A *term* is either a constant or a variable. Given $p \in \mathcal{P}$ an *atom* is defined as $p(t_1, \ldots, t_k)$, where k is called the arity of p and $t_1, \ldots, t_k$ are terms. Atoms of arity $k = 0$ are called *propositional atoms*.

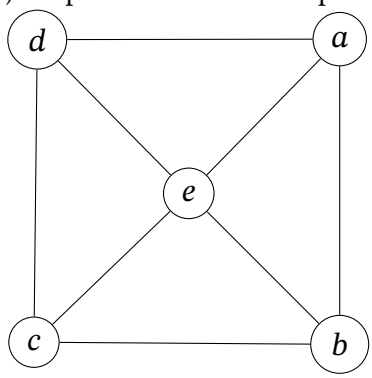
A *classical literal* (or simply *literal*) l is an atom p or a negated atom $\neg p$, where "$\neg$" is the symbol for true (classical) negation. Its *complementary* literal is $\neg p$ (respectively, p). A *negation as failure literal* (or *NAF-literal*) is a literal l or a default-negated literal not l. Negation as failure is an extension to classical negation, denoting a fact as false if all attempts to prove it fail. Thus, *not* l evaluates to *true* if it cannot be foundedly demonstrated that l is true, i.e., if either l is false or we do not know whether l is true or false.

A *rule* r is an expression of the form

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n \ , \tag{2.1}$$

where $k \geq 0$, $n \geq m \geq 0$, and $a_1, \ldots, a_k, b_1, \ldots, b_n$ are classical literals. We say that $a_1, \ldots, a_k$ is the *head* of r, while the conjunction $b_1, \ldots, b_m$, not $b_{m+1}, \ldots,$ not b_n is the *body* of r, where $b_1, \ldots, b_m$ (respectively, not $b_{m+1}, \ldots,$ not b_n) is the *positive* (respectively, *negative*) *body* of r. We use $H(r)$ to denote its head literals, and $B(r)$ to denote the set of all its body literals $B^+(r) \cup B^-(r)$, where $B^+(r) = \{b_1, \ldots, b_m\}$ and

(a) Graph instance for Example 2.1



(b) Legal 3-coloring of Example 2.1

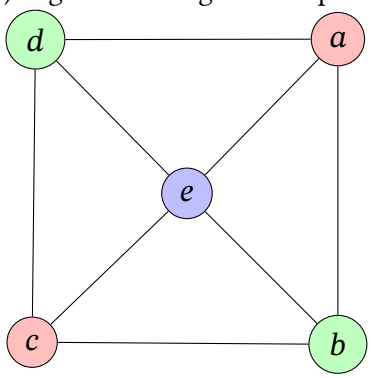


Figure 2.1: Graphs for Example 2.1

$B^-(r) = \{b_{m+1}, \ldots, b_n\}$. A rule r without head literals (i.e., $k = 0$) is an *integrity constraint*. A rule r with exactly one head literal (i.e., $k = 1$) is a *normal rule*. If the body of r is empty (that is, $m = n = 0$), then r is a *fact*, and we often omit "$\leftarrow$".[1] An *extended disjunctive logic program* (EDLP, or simply *program*) P is a finite set of rules r of the form (2.1).

Programs without disjunction in the heads of rules are called *extended logic programs* (ELPs). A program P without negation as failure, i.e., for all $r \in P$, $B^-(r) = \emptyset$ is called *positive logic program*. If, additionally, no strong negation occurs in P, i.e., the only form of negation is default negation in rule bodies, then P is called a *normal logic program* (NLP). The generalization of an NLP by allowing default negation in the heads of rules is called *generalized logic program* (GLP). Additional program classes of logic programs with the corresponding restrictions on the rules in a program are summarized in Table 2.1. Program classes based on dependency information such as stratified programs (Apt et al., 1988) are not considered here.

Next we will provide an answer set program as an example for specifying computational problems in a uniform way. This program will encode a problem from graph theory, namely the graph three-colorability (3COL) problem, which is a well-known NP-complete problem (Garey et al., 1976, give an accessible proof that reduces 3SAT to 3COL). The 3COL problem is defined as follows:
INSTANCE: Graph $G = (V, E)$.
QUESTION: Does G have a legal 3-coloring of its nodes, i.e., is there a mapping $f : V \to \{1, 2, 3\}$ such that if $(u, v) \in E$ then $f(u) \neq f(v)$?

The encoding given in Example 2.1 is uniform in the sense that it separates the problem specification from the concrete instance of a computational problem. That is, in the Answer Set Program, the instance of a problem is usually given as set of facts, while an additional set of rules that correspond to the problem specification is based

[1] In this thesis, we will use both forms "$a \leftarrow$" and "a," to denote that a is a fact in a logic program.

on the relations defined by the instance facts. This way, problem encodings become uniform, and we can abstract from concrete instances when defining the rules of the problem specification. This is the essence of the *Answer Set Programming Paradigm*, see Eiter et al. (2009b) and Janhunen and Niemelä (2016) for introductory material.

Example 2.1 (Graph Three-colorability (3COL)) Consider a graph G as shown in Figure 2.1a, which has six possible legal 3-colorings. One of them is depicted in Figure 2.1b, i.e., the mapping f such that $f(a) = f(c) = 1$ (using red nodes), $f(b) = f(d) = 2$ (using green nodes), and $f(e) = 3$ (in blue). If we would add the edge (a, c) to G, then G would not be three-colorable. Alternatively, adding (b, d) to G would have the same effect.

Now let P be an answer set program with the following set of rules:

$$
\begin{aligned}
col(X, red) \vee col(X, green) \vee col(X, blue) &\leftarrow node(X) \\
&\leftarrow col(X, C), col(Y, C), edge(X, Y) \\
node(X) &\leftarrow edge(X, Y) \\
node(X) &\leftarrow edge(Y, X)
\end{aligned}
$$

$$
\begin{aligned}
edge(a, b) &\leftarrow \\
edge(b, c) &\leftarrow \\
edge(c, d) &\leftarrow \\
edge(d, a) &\leftarrow \\
edge(a, e) &\leftarrow \\
edge(b, e) &\leftarrow \\
edge(c, e) &\leftarrow \\
edge(d, e) &\leftarrow
\end{aligned}
$$

This program is essentially split into two parts: one part comprising of the first four rules encodes the problem specification of 3COL, while the other part composed of the last eight facts encodes the graph from Figure 2.1a as a particular problem instance. Thus, P is a uniform encoding, as customary in Answer Set Programming. Note that the first rule is a disjunctive rule generating all 3-colorings using the colors $\{red, green, blue\}$ (instead of $\{1, 2, 3\}$), while the second rule is a constraint that forces 3-colorings to be legal as defined by the 3COL problem. The third and fourth rule use the auxiliary unary predicate *node* for specifying the set of nodes given the set of edges from the problem instance.

2.1.2 Semantics of Answer Set Programs

The semantics of extended disjunctive logic programs is defined for variable-free programs. Thus, we first define the *ground instantiation* of a program that eliminates its variables.

The *Herbrand universe* of a program P, denoted HU_P, is the set of all constant symbols $C \subset \mathcal{C}$ appearing in P. If there is no such constant symbol, then $HU_P = \{c\}$, where c is an arbitrary constant symbol from $\mathcal{C}$. As usual, terms, atoms, literals, rules, programs, etc. are *ground* iff they do not contain any variables. The *Herbrand base* of a program P, denoted HB_P, is the set of all ground (classical) literals that can be constructed from the predicate symbols appearing in P and the constant symbols in HU_P. A *ground instance* of a rule $r \in P$ is obtained from r by systematically replacing all instances of each variable that occurs in r by a constant symbol from HU_P. We use $ground(P)$ to denote the set of all ground instances of rules in P.

The semantics for EDLPs is defined first for positive ground programs. A set of literals $X \subseteq HB_P$ is *consistent* iff $\{p, \neg p\} \not\subseteq X$ for every atom $p \in HB_P$. An *interpretation* I relative to a program P is a consistent subset of HB_P. We say that a set of literals S *satisfies* a rule r if $H(r) \cap S \neq \emptyset$ whenever $B^+(r) \subseteq S$ and $B^-(r) \cap S = \emptyset$. A *model* of a positive program P is an interpretation $I \subseteq HB_P$ such that I satisfies all rules in P. An *answer set* of a positive program P is a minimal model of P with respect to set inclusion.

In order to extend this definition to programs with negation as failure, we define the *Gelfond-Lifschitz transform* (also often called the *Gelfond-Lifschitz reduct*) of a program P relative to an interpretation $I \subseteq HB_P$, denoted P^I, as the ground positive program that is obtained from $ground(P)$ by

1. deleting every rule r such that $B^-(r) \cap I \neq \emptyset$, and

2. deleting the negative body from every remaining rule.

An *answer set* of a program P is an interpretation $I \subseteq HB_P$ such that I is an answer set of P^I.

Example 2.2 Consider the following program P:

$$p \leftarrow \text{not } q$$
$$q \leftarrow \text{not } p$$

Let $I_1 = \{p\}$; then, $P^{I_1} = \{p \leftarrow\}$ with the unique minimal model $\{p\}$ and thus I_1 is an answer set of P. Likewise, P has an answer set $\{q\}$. However, the empty set $\emptyset$ is not an answer set of P, since the respective reduct would be $\{p \leftarrow; q \leftarrow\}$ with the minimal model $\{p, q\}$.

A constraint is used to eliminate "unwanted" models from the result, since its head is implicitly assumed to be *false*. A model that satisfies the body of a constraint is hence discarded from the set of answer sets.

Example 2.3 Let P be the program

$$p(X) \vee \neg p(X) \leftarrow q(X), \text{not}\, r(X)$$
$$q(c_1) \leftarrow$$
$$r(c_2) \leftarrow$$

The grounding of P is

$$p(c_1) \vee \neg p(c_1) \leftarrow q(c_1), \text{not}\, r(c_1)$$
$$p(c_2) \vee \neg p(c_2) \leftarrow q(c_2), \text{not}\, r(c_2)$$
$$q(c_1) \leftarrow$$
$$r(c_2) \leftarrow$$

This program has several models. For instance, $I_1 = \{q(c_1), r(c_1), r(c_2), p(c_1)\}$ is a model of P, since P^{I_1} is just

$$q(c_1) \leftarrow$$
$$r(c_2) \leftarrow$$

However, I_1 is not a minimal model of P^{I_1}. Now take $I_2 = \{q(c_1), r(c_2), p(c_1)\}$. We obtain P^{I_2} as

$$p(c_1) \vee \neg p(c_1) \leftarrow q(c_1)$$
$$q(c_1) \leftarrow$$
$$r(c_2) \leftarrow$$

Indeed, I_2 is a minimal model of P^{I_2}, hence it is an answer set of P. The other answer set is $I_3 = \{q(c_1), r(c_2), \neg p(c_1)\}$, as I_3 is a minimal model of $P^{I_3} = P^{I_2}$.

Example 2.4 Consider the 3COL example from above. The grounding of P is the program $ground(P)$:

$$col(a, red) \vee col(a, green) \vee col(a, blue) \leftarrow node(a)$$
$$col(b, red) \vee col(b, green) \vee col(b, blue) \leftarrow node(b)$$
$$col(c, red) \vee col(c, green) \vee col(c, blue) \leftarrow node(c)$$
$$col(d, red) \vee col(d, green) \vee col(d, blue) \leftarrow node(d)$$
$$col(e, red) \vee col(e, green) \vee col(e, blue) \leftarrow node(e)$$
$$col(blue, red) \vee col(blue, green) \vee col(blue, blue) \leftarrow node(blue)$$
$$col(green, red) \vee col(green, green) \vee col(green, blue) \leftarrow node(green)$$
$$col(red, red) \vee col(red, green) \vee col(red, blue) \leftarrow node(red)$$

$$
\begin{array}{rcl}
 & \leftarrow & col(a,1), col(a,1), edge(a,a) \\
 & \leftarrow & col(a,1), col(b,1), edge(a,b) \\
 & \vdots & \\
 & \leftarrow & col(a,2), col(a,2), edge(a,a) \\
 & \leftarrow & col(a,2), col(b,2), edge(a,b) \\
 & \vdots & \\
 & \leftarrow & col(a,a), col(a,a), edge(a,a) \\
 & \leftarrow & col(a,a), col(b,a), edge(a,b) \\
 & \vdots & \\
node(a) & \leftarrow & edge(a,a) \\
node(a) & \leftarrow & edge(a,b) \\
 & \vdots & \\
node(b) & \leftarrow & edge(a,b) \\
 & \vdots & \\
edge(a,b) & \leftarrow & \\
edge(b,c) & \leftarrow & \\
edge(c,d) & \leftarrow & \\
edge(d,a) & \leftarrow & \\
edge(a,e) & \leftarrow & \\
edge(b,e) & \leftarrow & \\
edge(c,e) & \leftarrow & \\
edge(d,e) & \leftarrow &
\end{array}
$$

Note that $ground(P)$ also contains unintuitive instances of rules from P such as

$$col(blue, red) \vee col(blue, green) \vee col(blue, blue) \leftarrow node(blue)$$

or

$$\leftarrow col(a,a), col(a,a), edge(a,a) \ .$$

Since P is positive, for each Herbrand interpretation I, $P^I = ground(P)$. Hence, the minimal models of $ground(P)$ and the answer sets of P coincide. One of them is the set

$$
\begin{aligned}
A = \{&edge(a,b), edge(b,c), edge(c,d), edge(d,a), \\
&edge(a,e), edge(b,e), edge(c,e), edge(d,e), \\
&node(a), node(b), node(c), node(d), node(e), \\
&col(a, red), col(b, green), col(c, red), col(d, green), col(e, blue)\} \ ,
\end{aligned}
$$

which corresponds to the 3-coloring shown in Figure 2.1b.

The main reasoning tasks that are associated with EDLPs under the answer-set semantics are the following:

- decide whether a given program P has an answer set;
- given a program P and a ground propositional formula ϕ, decide whether ϕ holds in every (respectively, some) answer set of P (*cautious* (respectively, *brave*) *reasoning*);
- given a program P and an interpretation $I \subseteq HB_P$, decide whether I is an answer set of P (*answer-set checking*); and
- enumerate the set of all answer sets of a given program P.

2.2 Generalized Quantifier Logic Programs

In this section, we recall definitions from Eiter et al. (1997b, 2000) for *Generalized Quantifier Logic Programs*, which form the basis for modular logic programs with Generalized Quantifiers (GQMLP) in §2.3. GQMLPs have been proposed as a logic programming formalism to support combining independent modules of logic programs. Such modular programs are nonmonotonic logic programs extended by the notion of generalized quantifiers (see Väänänen, 1999, for an introduction). In this approach, every module can be accessed via its generalized quantifier interface. GQLPs and GQMLPs are close in spirit to HEX-programs (Eiter et al., 2012a, 2006b, 2017), which are based on external atoms instead of generalized quantifier atoms.

2.2.1 Basic Concepts from Mathematical Logic

We start with defining notation. Letters $P, Q, \ldots$ denote predicates, lower case letters x, y, z variables, a, b, c, ... constants, and $f, g \ldots$ functions. The bold face version $\mathbf{P}$ of a predicate symbol P denotes a list $P_1, \ldots, P_m$ of predicate symbols, and similarly for variable, function, and constant symbols. Fraktur letters $\mathfrak{A}, \mathfrak{B}, \ldots$ denote logical structures. Sets of structures are denoted by capital letter C, and classes or mappings thereof by $\mathcal{M}, \mathcal{Q}, \ldots$; lower case Greek letters $\tau, \sigma, \ldots$ denote signatures.

Definition 2.1 (Signature).
A *signature* τ is a sequence $(P_1^{a_1}, \ldots, P_k^{a_k}, f_1^{b_1}, \ldots, f_l^{b_l}, c_1, \ldots, c_m)$ where the P_i are relational symbols of arity $a_i \geq 0$, the f_i are functions with $b_i \geq 1$ arguments, and the c_i are constants. τ is *relational*, if it contains only relational symbols.

Definition 2.2 (Structure).
A *structure* $\mathfrak{A}$ over τ is denoted by $(A, P_1^{\mathfrak{A}}, \dots, P_k^{\mathfrak{A}}, f_1^{\mathfrak{A}}, \dots, f_l^{\mathfrak{A}}, c_1^{\mathfrak{A}}, \dots, c_m^{\mathfrak{A}})$. The set A is called the *universe* or *domain* of $\mathfrak{A}$, and denoted $|\mathfrak{A}|$. $\mathfrak{A}$ is finite if $|\mathfrak{A}|$ is finite. The set of all structures over τ is denoted by $\mathrm{Struct}(\tau)$.

Let $\mathfrak{A}, \mathfrak{B} \in \mathrm{Struct}(\tau)$ such that $|\mathfrak{A}| = |\mathfrak{B}|$. Then, $\mathfrak{A} \subseteq \mathfrak{B}$, if $P_i^{\mathfrak{A}} \subseteq P_i^{\mathfrak{B}}$ for $1 \leq i \leq k$, $f_i^{\mathfrak{A}} = f_i^{\mathfrak{B}}$ for $1 \leq i \leq l$, and $c_i^{\mathfrak{A}} = c_i^{\mathfrak{B}}$ for $1 \leq i \leq m$.

For a relational signature τ and integer l, $\tau^{(l)} = (P_1^{la_1}, \dots, P_k^{la_k})$ is called the *l-ary vectorization* of τ.

Let $\mathfrak{A}$ be a relational structure, and $U \subseteq |\mathfrak{A}|$. Then the restriction of $\mathfrak{A}$ to universe U, in symbols $\mathfrak{A}|U$, is the structure $(U, P_1^{\mathfrak{A}} \cap U^{a_1}, \dots, P_k^{\mathfrak{A}} \cap U^{a_k})$. For a τ-structure $\mathfrak{A}$ and a signature τ_0 contained in τ, $\mathfrak{A}|\tau_0$ is the τ_0-structure obtained from $\mathfrak{A}$ by removing all relations, functions, and constants not contained in τ_0. Given a τ_0-structure $\mathfrak{B}$ such that $|\mathfrak{A}| = |\mathfrak{B}|$ and $\mathfrak{B} = \mathfrak{A}|\tau_0$, then $\mathfrak{B}$ is said to be the reduct of $\mathfrak{A}$ to τ_0, and conversely, $\mathfrak{A}$ is an expansion of $\mathfrak{B}$ to τ.

The set of all finite models of a formula Ψ is denoted by $\mathrm{Mod}(\Psi)$.

Let $\phi(x_1, \dots, x_n)$ be a formula with free variables $x_1, \dots, x_n$, and let $\mathfrak{A}$ be a structure. Then $\phi^{\mathfrak{A}}$ denotes the n-ary relation $\{(d_1, \dots, d_n) \in |\mathfrak{A}|^n \mid \mathfrak{A} \vDash \phi(d_1, \dots, d_n)\}$.

Let $\mathcal{L}$ be a syntactic fragment of first-order logic. Given signatures τ, σ and a natural number k, a k-ary interpretation I of τ into σ is a definition of the $\sigma^{(k)}$ relations in terms of τ, i.e., a tuple of $\mathcal{L}$ formulas, such that for each predicate symbol R in σ with arity r, I contains a formula ϕ_R over τ with $r \cdot k$ free variables which defines $R^{r \cdot k}$. For a structure $\mathfrak{A} \in \mathrm{Struct}(\tau)$, $I(\mathfrak{A})$ denotes the structure over $\sigma^{(k)}$ which is defined by I.

2.2.2 Generalized Quantifiers

Next, we define Generalized Quantifiers and their semantics.

Definition 2.3 (Generalized quantifiers).
Let C be a class of logical structures over a relational signature $\sigma = (R_1, \dots, R_n)$ with arities $a_1, \dots, a_n$ such that C is closed under isomorphism, i.e., if $\mathfrak{A} \cong \mathfrak{B}$ and $\mathfrak{A} \in C$, then $\mathfrak{B} \in C$. Such class C has an associated *generalized quantifier (GQ)* Q_C.

The intended semantics of a GQ Q_C is to check if a relation defined by the underlying logic belongs to a class of logical structures C.

Definition 2.4 (Extension of logics by a GQ).
The *extension* $\mathcal{L}(Q_C)$ *of a logic* $\mathcal{L}$ *by a GQ* Q_C is the closure of $\mathcal{L}$ under the following rule: If $\phi_1(\mathbf{x}_1), \dots, \phi_n(\mathbf{x}_n)$ are formulas of logic $\mathcal{L}$, where every ϕ_i has at least a_i free variables $\mathbf{x}_i$, then $Q_C \mathbf{x}_1 \cdots \mathbf{x}_n[\phi_1, \dots, \phi_n]$ is a formula of the extension $\mathcal{L}(Q_C)$, in which the occurrences of $\mathbf{x}_1, \dots, \mathbf{x}_n$ are bound.

For clarity, we shall often write the list of remaining free variables $\mathbf{y}$ after the formula. The semantics of a GQ Q_C is defined as follows.

Definition 2.5 (Semantics of GQs).
Let Θ be the formula

$$Q_C \mathbf{x}_1 \cdots \mathbf{x}_n[\phi_1, \ldots, \phi_n](\mathbf{y}),$$

and let $\mathfrak{A}$ be a structure and $\mathbf{b}$ a tuple over $|\mathfrak{A}|$ matching the arity of $\mathbf{y}$. Then $(\mathfrak{A}, \mathbf{b}) \models \Theta$, if and only if the structure $(A, \phi_1^{\mathfrak{A},\mathbf{b}}, \ldots, \phi_n^{\mathfrak{A},\mathbf{b}})$ belongs to C, where

$$\phi_i^{\mathfrak{A},\mathbf{b}} = \{\mathbf{a} \mid \mathfrak{A} \models \phi_i[\mathbf{a}, \mathbf{b}]\} \quad .$$

Example 2.5 Let $\mathfrak{A}$ be a structure with a domain $A = |\mathfrak{A}|$. The following list shows common generalized quantifiers and their semantics. Some of them will be used in further examples.

- $Q_\forall = \{(A, A)\}$ (universal quantifier)
- $Q_\exists = \{(A, U) \mid \varnothing \neq U \subseteq A\}$ (existential quantifier)
- $Q_\sim = \{(A, U, \{v\}) \mid v \in A \setminus U\}$ (complement quantifier)
- $Q_M = \{(A, U, V) \mid U, V \subseteq A, |U| > |V|\}$ (majority quantifier)
- $Q_k = \{(A, U) \mid |U| \equiv 0 \mod k\}$ (modularity quantifier)
- $Q_\cong = \left\{(A, E, F) \;\middle|\; \begin{array}{c}(A, E) \text{ and } (A, F)\\ \text{are isomorphic graphs}\end{array}\right\}$ (isomorphism quantifier)
- $Q_{TC} = \left\{(A, E, \{(u, v)\}) \;\middle|\; \begin{array}{l}\text{there is a path from } u \text{ to}\\ v \text{ in the graph } E \subseteq A \times A\end{array}\right\}$ (transitive closure quantifier)

2.2.3 Logic Programs with Generalized Quantifiers

Now we can define logic programs with generalized quantifiers.

Definition 2.6 (GQ atoms and literals).
Suppose that Q_C is a GQ defined over the signature $\sigma = (R_1, \ldots, R_n, R_{n+1})$ with associated arities $a_1, \ldots, a_n, a_{n+1} = l$, and that $S_1, \ldots, S_n$ are predicates from τ such that the arity of S_i equals a_i. Then, the formula

$$Q_C \mathbf{x}_1 \cdots \mathbf{x}_{n+1}[S_1(\mathbf{x}_1), \ldots, S_n(\mathbf{x}_n), \mathbf{x}_{n+1} = \mathbf{v}](\mathbf{v}) \tag{2.2}$$

is a *GQ-atom* with free variables $\mathbf{v} = v_1, \ldots, v_l$. A *GQ-literal* is a possibly negated GQ-atom. For brevity, we denote a GQ atom (2.2) by

$$Q_C[\mathbf{S}](\mathbf{v}) \qquad \text{(respectively, } Q_C[\mathbf{S}], \text{ if } \mathbf{v} \text{ is void)} \quad , \tag{2.3}$$

where $\mathbf{S} = S_1, \ldots, S_n$, and similarly for negative GQ-literals.

Notice that formula $\mathbf{x}_{n+1} = \mathbf{v}$ located in GQ-atom (2.2) defines the singleton relation $\{\mathbf{v}\}$, whose purpose is to transfer domain elements into the quantifier. For GQ-atoms of form (2.3), bound variables $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are implicitly understood.

Definition 2.7 (GQ logic programs).
Let τ_0 be a signature for describing the program input, and let τ be an extension of τ_0 by new relational symbols. A *logic program with GQs* (*GQLP*) on τ is a finite collection $\mathcal{P}$ of rules

$$A \leftarrow B_1, \ldots, B_m \tag{2.4}$$

where the head A is a τ-atom whose predicate does not occur in τ_0, and each body literal B_i is either a τ-literal or a generalized quantifier literal (GQ-literal) over τ.

For any collection $\mathcal{Q}$ of generalized quantifiers, we denote by $\tau^*_{\mathcal{Q}}$ the extension of τ by all predicate letters $Q_C[\mathbf{S}]$, where $Q_C \in \mathcal{Q}$ and $\mathbf{S} = S_1, \ldots, S_n$ is a list of predicate letters S_i from τ which matches the signature of Q_C; every such $Q_C[\mathbf{S}]$ is a *GQ-predicate*. Notice that $\tau^*_{\mathcal{Q}}$ is finite if τ and $\mathcal{Q}$ are finite. Then, a GQLP over τ is syntactically an ordinary logic program over the signature $\tau^*_{\mathcal{Q}}$.

Example 2.6 Consider the following program $\mathcal{P}$, which uses two GQs (see Example 2.5 for their definitions). One is the isomorphism GQ $Q_{\cong}[G_1, G_2]$, which tells whether G_1 and G_2 are isomorphic graphs, and the transitive closure GQ TC:

$$\begin{aligned} S(x,y) &\leftarrow Q_{TC}[E](x,y), Q_{TC}[E](y,x) \\ G_a(x,y) &\leftarrow E(x,y), S(a,x), S(a,y) \\ G_b(x,y) &\leftarrow E(x,y), S(b,x), S(b,y) \\ \mathit{Iso} &\leftarrow Q_{\cong}[G_a, G_b] \end{aligned}$$

Suppose that τ_0 contains the relation E and the constant symbols a and b. Given a graph $G = (V, E)$, and vertices $a, b \in V$, this program assigns the propositional letter *Iso* to true, if the strongly connected components in which a and b lie are isomorphic.

The semantics of a GQLP $\mathcal{P}$ is in spirit of the stable model semantics (Gelfond and Lifschitz, 1988). In the following, suppose that we have signatures τ_0 and τ as above, a GQLP $\mathcal{P}$ and a structure $\mathfrak{A} \in \mathrm{Struct}(\tau)$.

Definition 2.8 (Ground instantiation).
Then, the *ground instantiation* of $\mathcal{P}$ on $\mathfrak{A}$, denoted $\mathit{ground}(\mathcal{P}, \mathfrak{A})$, is the collection of all interpreted rules $C\theta$, where C is from $\mathcal{P}$ and θ is any ground substitution over $\mathfrak{A}$.

Definition 2.9 (Reduct).
Let $\mathcal{P}$ be a GQLP and $\mathfrak{A} \in \mathrm{Struct}(\tau)$. The *reduct of $\mathcal{P}$ with respect to* $\mathfrak{A}$, denoted $\mathit{red}(\mathcal{P}, \mathfrak{A})$, is the set of rules obtained from $\mathit{ground}(\mathcal{P}, \mathfrak{A})$ as follows.

1. Remove every rule r with a literal L in the body of r such that $\mathfrak{A} \not\models L$, where L is either negative or a GQ-literal.

2. Remove all negative literals and GQ-literals from the remaining rules.

Notice that $red(\mathcal{P}, \mathfrak{A})$ is a collection of interpreted Horn clauses, hence there is a least structure $\mathfrak{B}$, denoted $\mathfrak{A}_\infty(\mathcal{P})$, such that $\mathfrak{B}|\tau_0 = \mathfrak{A}|\tau_0$ (i.e., $\mathfrak{B}$ provides the same input to $\mathcal{P}$ as $\mathfrak{A}$) and $B \vDash r$, for every rule $r \in red(\mathcal{P}, \mathfrak{A})$. Since $red(\mathcal{P}, \mathfrak{A})$ is an ordinary logic program, the structure $\mathfrak{A}_\infty(\mathcal{P})$, also called the least model of $\mathcal{P}$ with respect to $\mathfrak{A}$, can be obtained as the least fixpoint of a monotonic operator, confer Lloyd (1987).

Definition 2.10 (GQ-stable models).
Let $\mathcal{P}$ be a GQLP and let $\mathfrak{A}_0 \in \text{Struct}(\tau_0)$. An expansion $\mathfrak{A} \in \text{Struct}(\tau)$ of $\mathfrak{A}_0$ is a *GQ-stable model of $\mathcal{P}$ on $\mathfrak{A}_0$*, iff it satisfies the fixpoint equation

$$\mathfrak{A} = \mathfrak{A}_\infty(\mathcal{P}).$$

The collection of all stable models of $\mathcal{P}$ is denoted by $\text{SM}(\mathcal{P}, \mathfrak{A}_0)$.

The *meaning* of $\mathcal{P}$ on $\mathfrak{A}_0$, denoted $\mathcal{M}^{st}_{\mathcal{P}}(\mathfrak{A}_0)$, is the structure which is the intersection of all GQ-stable models of $\mathcal{P}$ on $\mathfrak{A}_0$, i.e.,

$$\mathcal{M}^{st}_{\mathcal{P}}(\mathfrak{A}_0) = \bigcap_{\mathfrak{A} \in \text{SM}(\mathcal{P}, \mathfrak{A}_0)} \mathfrak{A} \ .$$

If $\text{SM}(\mathcal{P}, \mathfrak{A}_0) = \varnothing$, then $\mathcal{M}^{st}_{\mathcal{P}}(\mathfrak{A}_0)$ is the unique maximal structure $\mathfrak{B}$ such that $\mathfrak{B}|\tau_0 = \mathfrak{A}_0$.

Example 2.7 Consider the following program $\mathcal{P}$, which uses the majority quantifier Q_M and the modularity quantifier Q_2 (i.e., the even quantifier).

$$\begin{aligned} Q(x) &\leftarrow \neg S(x) \\ S(x) &\leftarrow \neg Q(x) \\ A(x) &\leftarrow Q_M[Q, S], S(x) \\ Q(x) &\leftarrow Q_2[A], S(x) \\ W(a, b) &\leftarrow \end{aligned}$$

Suppose that τ_0 contains merely the constant symbols a and b, and τ contains in addition the relation symbols W, A, Q, and S.

Intuitively, the first two clauses choose complementary extensions for Q and S; the third clause assures that S implies A, if Q holds on more individuals than S; similarly, the fourth clause assures that S implies Q, if A holds true on an even number of elements.

Consider Herbrand models on τ and let $\mathfrak{M}_1 = \{W(a, b), Q(a), Q(b)\}$. (We use the familiar notation for Herbrand models.) This interpretation is a GQ-stable model of

$\mathcal{P}$ (with respect to the unique Herbrand model $\mathfrak{M}_0 \in \text{Struct}(\tau_0)$). Indeed, the reduct $red(\mathcal{P}, \mathfrak{M}_1)$ consists of the clauses

$$\begin{array}{ll} W(a,b) \leftarrow & A(b) \leftarrow S(b) \\ Q(a) \leftarrow & Q(a) \leftarrow S(a) \\ Q(b) \leftarrow & Q(b) \leftarrow S(b) \\ A(a) \leftarrow S(a) & \end{array}$$

Program $\mathcal{P}$ has the least model $\mathfrak{M}_1$ with respect to $\mathfrak{M}_0$. Another GQ-stable model of $\mathcal{P}$ is $\mathfrak{M}_2 = \{W(a,b), S(a), S(b)\}$. The Herbrand model $\mathfrak{M}_3 = \{W(a,b), Q(a), S(b)\}$ is not a GQ-stable model of $\mathcal{P}$: $red(\mathcal{P}, \mathfrak{M}_3)$ contains the clauses $S(b) \leftarrow$ and $Q(b) \leftarrow S(b)$, which means that the least model of $red(\mathcal{P}, \mathfrak{M}_3)$ contains $Q(b)$. We obtain that $\text{SM}(\mathcal{P}, \mathfrak{M}_0) = \{\mathfrak{M}_1, \mathfrak{M}_2\}$, thus $\mathfrak{M}_1$ and $\mathfrak{M}_2$ are all GQ-stable models of $\mathcal{P}$ with respect to $\mathfrak{M}_0$.

2.3 Modular Logic Programming with GQLPs

Based on GQLPs, we define a semantics for modular logic programming in this section. This approach has been introduced by Eiter et al. (1997b, 2000), and we use their definitions here.

2.3.1 Syntax of modular logic programs

The syntax of *modular logic programs* (*GQMLPs*) is the one of GQLPs defined in §2.2.3, with the difference that the GQ-literals are intended to refer to a logic program, which is a *logic program module*. The similarity type of $LP[\mathbf{Q}]$ is the list of arities of predicates in $\mathbf{Q}$.

We shall refer to the calling program as the *main program*, and the called module as the *subprogram*; the GQ-literals in a GQMLP are termed *call literals*, and the GQ-predicates *call predicates*. An atom is a *call atom*, if its predicate is a call predicate. To distinguish ordinary predicates, atoms, and literals from call predicates, call atoms, and call literals, we call the former *standard* predicates (atoms, literals, respectively).

Definition 2.11 (Logic program module).
A *logic program module* μ is a pair $(LP[\mathbf{Q}], P)$ of a *module head* $LP[\mathbf{Q}]$, which has an associated integer $n \geq 0$ *(the arity)*, and an ordinary logic program P *(the body)*, in which the predicates $\mathbf{Q}$ are the input predicates and LP is the output predicate having arity n; syntactically, occurrences of the predicates $\mathbf{Q}$ in P are restricted to rule bodies. We require that each LP module is uniquely identified by its *name* LP and the list of the arities of the Q_i in $\mathbf{Q}$ (its *similarity type*).

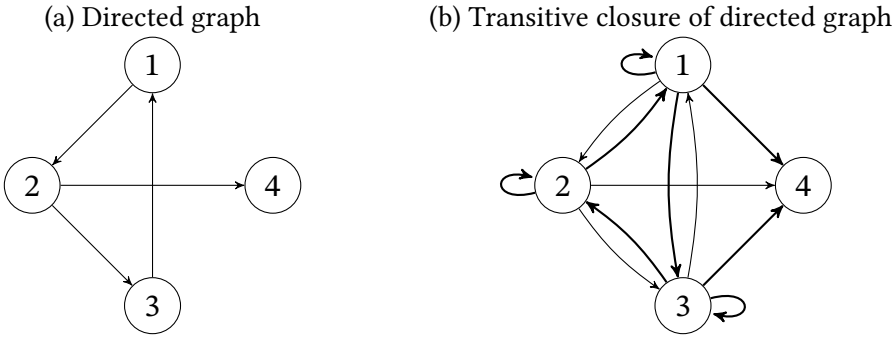


Figure 2.2: Graphs for Example 2.8

Definition 2.12 (Modular logic programs).
A *modular logic program* is a finite collection P of rules

$$A \leftarrow L_1, \ldots, L_m \ ,$$

where the head A is a standard atom and each body literal L_i is either a standard literal or a call literal, plus a collection $\mathcal{C}$ of logic program modules such that for each call literal $(\neg)LP[\mathbf{Q}](\mathbf{t})$ occurring in P, there is a module $LP[\mathbf{Q}']$ in $\mathcal{C}$, where LP has the arity of $\mathbf{t}$ and each $Q_i \in \mathbf{Q}$ has the arity of $Q_i' \in \mathbf{Q}'$.

2.3.2 Semantics of modular logic programs

Next, we define the meaning of a modular logic program.

Every LP module $\mu = (LP[\mathbf{Q}], P)$ under the semantics $\mathcal{M}^{st}$ can be seen as a GQ $Q_{C(\mu)}$ that is associated with the collection $C(\mu)$ of all structures $\mathfrak{A} = (A, \mathbf{Q}', R')$, where the $\mathbf{Q}'$ are relations for the predicates $\mathbf{Q}$ on A and $R' = \{\mathbf{a}\}$, for any tuple $\mathbf{a}$ over A such that

$$C(\mu) = \{(A, \mathbf{Q}', \{\mathbf{a}\}) \mid \mathcal{M}_P^{st}((A, \mathbf{Q}')) \models LP(\mathbf{a})\} \ .$$

Intuitively, P derives the atom $LP(\mathbf{a})$ on $\mathfrak{A}_0 = (A, \mathbf{Q}')$. We call $Q_{C(\mu)}$ the *module quantifier of μ under $\mathcal{M}^{st}$*.

Let P be a modular logic program. The meaning of P under $\mathcal{M}^{st}$ is defined as the meaning of P, viewed as GQLP over the collection of GQs associated with the LP modules used by P.

Example 2.8 Let $\mathfrak{M} = \{E(1,2), E(2,3), E(3,1), E(2,4), U(1)\}$ be a Herbrand model of the signature $\tau = \{E, U\}$ on the domain $\{1,2,3,4\}$. Then, $E^{\mathfrak{M}}$ is the digraph depicted in Figure 2.2a.

Consider the module $\mu = (TC[G], P_{TC})$, where P_{TC} is the program

$$TC(x, y) \leftarrow G(x, y)$$
$$TC(x, y) \leftarrow TC(x, z), G(z, y)$$

The extension of TC in the least fixpoint of P_{TC} is the transitive closure of the binary relation G.

The program P_{TC} has on $\mathfrak{A}_0 = \{G(1,2), G(2,3), G(3,1), G(2,4)\}$ the least model $\mathfrak{M}' = \{G(1,2), G(2,3), G(3,1), G(2,4)\} \cup \{TC(i,j), TC(i,4) \mid 1 \leq i, j \leq 3\}$. Figure 2.2b shows $TC^{\mathfrak{A}_0}$, where thick lines highlight those edges from graph $TC^{\mathfrak{A}_0}$ that do not belong to directed graph $G^{\mathfrak{A}_0}$. Thus, a call $TC[E]$ of μ, where E is defined in $\mathfrak{M}$, yields that, e.g., $TC[E](1,1)$ is true, while $TC[E](4,1)$ is false.

Note that compared to the GQLP in Example 2.6, we use μ to define a generalized quantifier that is equivalent to GQ Q_{TC} from Example 2.5.

2.3.3 Shortcomings of Generalized Quantifier Modular Logic Programs

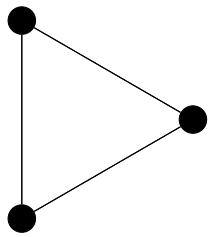
An important restriction for GQMLPs is that modules do not contain call literals themselves, i.e., the main program is the only part of a GQ modular logic program that can access subprograms through generalized quantifier calls, but subprograms are forbidden to have call literals. This essential restriction is important, as the semantics for GQMLPs would create semantic deficiencies like unfounded answer sets.

There is a way that allows one to make calls from subprograms to other subprograms, but they must be strictly hierarchical. In essence, the call graph of the modules must be acyclic and this way, one can only express finitely nested logic programs without mutual recursion (see the discussion by Eiter et al., 2000, Section 7). Similar to GQLPs and GQMLPs, Eiter et al. (2013) define nested HEX-programs, which give a semantics to hierarchical subprograms using external atoms; we defer the discussion to Chapter 10.

In the following chapters of this thesis, we will shed light on this problem and provide a solution to this restriction using modular nonmonotonic logic programs. To this end, we start with defining the principle framework for Modular Nonmonotonic Logic Programs next and establish basic semantic properties for them.

II

Elements of Modular Nonmonotonic Logic Programs



3

Modular Nonmonotonic Logic Programs

We start here with our framework for modular answer set programs, and define first syntax and then semantics of such programs. We assume that the reader is familiar with basic notions of logic programming and the answer set semantics of nonmonotonic logic programs (see Chapter 2 and Gelfond and Lifschitz, 1991). The syntax is based on disjunctive logic programs; our modular logic programs consist of modules as a way to structure logic programs. Moreover, such modules allow for input provided by other modules; it is safe to say that one module may call other modules and additionally provide input. We pose no essential restriction on the rules, and modules may mutually call each other in a recursive way, and, on top of that, provide mutual input.

The declarative semantics we provide for MLPs caters for this situation and is thus not straight-forward. By the very notion of module input, it is apparent that modules must be instantiated before they can be "used." When defining a declarative semantics, we abstract from the computational view of module calls and do not consider their operational semantics. This would require to consider module call chains, which may lead to infinite loops and thus necessitate loop checking. The MLP semantics is similar in nature to Kripke semantics (Blackburn and Benthem, 2007; Goranko and Otto, 2007): instead of worlds, there are so-called *value calls with input*, and the *call graph* of an MLP resembles the accessibility relation in a Kripke frame. In contrast to Kripke semantics, MLP semantics does not consider situations that are not modeled within the modular program, i.e., if a module accesses another module, then there will be a labeled edge in the call graph that records input information. This is different from Kripke semantics, which admits Kripke frames of any shape.

To this end, we delineate contexts of models that carry instantiations of modules and serve to define answer sets for modular programs. As noted by Eiter et al. (1997b), answer sets of modular programs based on Gelfond-Lifschitz-style reducts may be

weaker than answer sets of ordinary logic programs, we thus based the notion of answer sets on the FLP-reduct (Faber et al., 2011) in order to gain the desired property of minimality in answer sets.

3.1 Syntax of Modular Nonmonotonic Logic Programs

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

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

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

$$P[p_1, \ldots, p_k].o(t_1, \ldots, t_l) \ , k, l \geq 0, \tag{3.1}$$

where $P \in \mathcal{M}$ is a module name, $p_1, \ldots, p_k$ 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 from P, and $o \in \mathcal{P}$ is a predicate name with arity l such that for the list of terms $t_1, \ldots, t_l$, $o(t_1, \ldots, t_l)$ is an ordinary atom.

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

A *rule* r is of the form

$$\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_m, \text{not}\, \beta_{m+1}, \ldots, \text{not}\, \beta_n \ , \tag{3.2}$$

where $k \geq 1, n \geq m \geq 0$, $\alpha_1, \ldots, \alpha_k$ are atoms, and $\beta_1, \ldots, \beta_n$ are either atoms or module atoms. We define $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \ldots, \beta_m\}$ and $B^-(r) = \{\beta_{m+1}, \ldots, \beta_n\}$. If $B(r) = \emptyset$ and $H(r) \neq \emptyset$, then r is a *fact*; r is *ordinary*, if it contains only ordinary atoms. A rule r is called *positive* if it satisfies $m = n$; r is a *Horn* rule if r is positive and $k = 1$. If $k \leq 1$, then r is called *normal*. Rules without restrictions are called *disjunctive*.

We now formally define the syntax of modules.

Definition 3.1 (Module).
A *module* is a pair $m = (P[\mathbf{q}], R)$, where $P \in \mathcal{M}$ with associated formal input $\mathbf{q}$, and R is a finite set of rules. It is *ordinary*, if all rules in R are ordinary, and *ground*, if all rules in R are ground. A module m is either a *main module* or a *library module*; if it is a main module, then $|\mathbf{q}| = 0$.

We refer with $R(m)$ to the rule set of m, or simply identify R with P respectively $P[\mathbf{q}]$ if m is identified by P. When clear from the context, we omit empty [] and () from (main) modules and module atoms. E.g., the module $Parity[q]$ in Example 1.1 is a library module; further examples are given below.

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

Definition 3.2 (Modular logic program).
A *modular logic program* (MLP) $\mathbf{P}$ is an n-tuple of modules

$$(m_1, \ldots, m_n) \ , n \geq 1, \tag{3.3}$$

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

An MLP $\mathbf{P} = (m_1, \ldots, m_n)$ is called *positive* (respectively, *Horn* or *normal*) if for each module m_i of $\mathbf{P}$, all rules $r \in R(m_i)$ are positive (respectively, Horn or normal); $\mathbf{P}$ is said to have *empty inputs* if all the associated lists of inputs for each module m_i of $\mathbf{P}$ is empty.

Example 3.1 (cont'd) Suppose that we have a module $m_2 = (P[q], R_2)$, where R_2 is taken from the rules shown in Example 1.1. Besides m_2, we have further module $m_1 = (Q[], R_1)$, in which R_1 is the set of rules

$$\begin{aligned} s(a) &\leftarrow \\ s(b) &\leftarrow \\ s(c) &\leftarrow \\ s(d) &\leftarrow \\ s_1(X) \vee s_2(X) &\leftarrow s(X) \\ ok &\leftarrow P[s_1].even, P[s_2].even \\ ok &\leftarrow \text{not } ok \end{aligned}$$

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

The example above shows an interesting property of MLPs. Note that the predicates in Example 3.1 are at most unary, i.e., the program **P** uses only monadic relations, and that we have expressed the Even property with this program. It is well known that traditional logic programs using monadic predicates cannot express Even, and this also holds for programs with negation, which follows from Libkin (2004, Proposition 7.12). The following proposition formally shows this observation.

Proposition 3.1 (Even not in Monadic ASP)
Monadic Answer Set Programs cannot express Even.

Proof Let P be a monadic answer set program. Without loss of generality, we assume that variables in P are standardized apart, i.e., each rule in P has a unique set of variables, and that P does not contain constant symbols; every constant c can be replaced by a singleton unary relation. Let $A = \{p_1, \dots, p_\ell\}$ be the predicates occurring in P, and $\widetilde{A} = \{\widetilde{p_1}, \dots, \widetilde{p_\ell}\}$ be a set a fresh predicates.

We define Φ_P to be the following monadic second order logic (MSO) formula:

$$\exists A \left[\Gamma_P \wedge \forall \widetilde{A} \left(\left(\widetilde{A} < A \right) \supset \Xi_P \right) \right]$$

where

- for the set $X = \{X_1, \dots, X_y\}$ of variables occurring in P

$$\Gamma_P = \forall X \bigwedge_{r \in P} (B_r \wedge N_r \supset H_r)$$

and

$$\Xi_P = \exists X \bigvee_{r \in P} \neg \left(\widetilde{B_r} \wedge N_r \supset \widetilde{H_r} \right) \ ;$$

- for each rule r of form (2.1) in P,

$$\begin{aligned}
H_r &= a_1(X_{\rho(r,1)}) \vee \cdots \vee a_k(X_{\rho(r,k)}) \\
\widetilde{H_r} &= \widetilde{a_1}(X_{\rho(r,1)}) \vee \cdots \vee \widetilde{a_k}(X_{\rho(r,k)}) \\
B_r &= b_1(X_{\rho(r,k+1)}) \wedge \cdots \wedge b_m(X_{\rho(r,k+m)}) \\
\widetilde{B_r} &= \widetilde{b_1}(X_{\rho(r,k+1)}) \wedge \cdots \wedge \widetilde{b_m}(X_{\rho(r,k+m)}) \\
N_r &= \neg b_{m+1}(X_{\rho(r,k+m+1)}) \wedge \cdots \wedge \neg b_n(X_{\rho(r,k+n)}) \ ,
\end{aligned}$$

where $\rho(r, i)$ maps to a variable index for the set of variables X;

- $\widetilde{A} < A$ is short for $\widetilde{A} \leq A \wedge \neg(A \leq \widetilde{A})$, where for the fresh variable Y

$$\widetilde{A} \leq A = \forall Y\,(\widetilde{p_1}(Y) \supset p_1(Y)) \wedge \cdots \wedge \forall Y\,(\widetilde{p_\ell}(Y) \supset p_\ell(Y))$$

  and

$$A \leq \widetilde{A} = \forall Y\,(p_1(Y) \supset \widetilde{p_1}(Y)) \wedge \cdots \wedge \forall Y\,(p_\ell(Y) \supset \widetilde{p_\ell}(Y)) \quad .$$

Intuitively, Φ_P expresses answer set semantics of a monadic program P as an MSO formula:

- A and $\widetilde{A}$ stand for interpretations of the monadic predicates in P,
- Γ_P expresses that all rules $r \in P$ are satisfied by A,
- $\widetilde{A} < A$ states that $\widetilde{A}$ is a proper subset of A,
- Ξ_P is true whenever $\widetilde{A}$ does not satisfy at least one rule of the reduct P^A; note that the truth value of N_r is fixed by A.

Thus the intuitive reading of the MSO formula Φ_P gives us that there exists an interpretation A such that A is a model of P, and for all proper subsets $\widetilde{A}$ of A, $\widetilde{A}$ does not satisfy the reduct P^A.

We show now the following: Φ_P is satisfiable if and only if P has an answer set.

($\Leftarrow$) Let M be an answer set of the monadic answer set program P. We transform M to a relational interpretation $\mathfrak{M} = (U_{\mathfrak{M}}, \cdot^{\mathfrak{M}})$ for Φ_P as follows:

- for each $p_i(c) \in M$ we add c to $U_{\mathfrak{M}}$ and set $c^{\mathfrak{M}} = c$,
- for predicates p_i, $1 \leq i \leq \ell$, we set $p_i^{\mathfrak{M}} = \{c^{\mathfrak{M}} \mid p_i(c) \in M\}$, and
- for predicates $\widetilde{p_i}$, $1 \leq i \leq \ell$ we set $\widetilde{p_i}^{\mathfrak{M}}$ to any proper subset of $p_i^{\mathfrak{M}}$.

Intuitively, M gives us an extension for A and $\widetilde{A}$ encodes a proper subset of M. Since $M \models P$, it is easy to see that for an extension for A, the model $\mathfrak{M}$ of Φ_P satisfies Γ_P. By M being an answer set, M is a minimal model of P^M. Thus, for all proper subsets N of M, there exists a rule $r' \in P^M$ such that $N \not\models r'$, i.e., $N \models B^+(r')$ and $N \not\models H(r')$. Thus, if the set variable extensions in $\widetilde{A}$ happen to be a proper subset of the set variable extensions in A, which is encoded by $\widetilde{A} < A$, then at least one rule must be false in P^M: hence for an extension A such that for all extensions $\widetilde{A}$ the formula $\big(\widetilde{A} < A\big) \supset \Xi_P$ is true in $\mathfrak{M}$. And since M is minimal, this holds for all proper subsets of A, hence $\forall \widetilde{A}\,\big(\big(\widetilde{A} < A\big) \supset \Xi_P\big)$ is satisfied by $\mathfrak{M}$. Thus, Φ_P is satisfiable.

($\Rightarrow$) Let Φ_P be satisfiable. We can deduce that P has an answer set M that can be transformed from a model $\mathfrak{M} = (U_{\mathfrak{M}}, \cdot^{\mathfrak{M}})$ for Φ_P as follows:

$$M = \{p_i(c) \mid \text{ for } c \in \mathcal{C} \text{ and } p_i \in A \text{ such that } c^{\mathfrak{M}} \in p_i^{\mathfrak{M}}\} \ .$$

Thus, M corresponds to the extension of predicates from A. From $\mathfrak{M}$ satisfying the conjunct Γ_P for an extension A we can deduce $M \vDash P$ and $M \vDash P^M$. Now we show that since $\mathfrak{M}$ is a model of $\forall \widetilde{A} \left(\left(\widetilde{A} < A \right) \supset \Xi_P \right)$ for the fixed extension A, we get that M is a minimal model of P^M. Towards a contradiction, assume there exists an interpretation $N \subset M$ that is a model of P^M. Thus, for $\widetilde{A}$ corresponding to N the antecedent $\widetilde{A} < A$ forces that the consequent Ξ_P must be true in $\mathfrak{M}$, which means that for at least one $r' \in P^M$, the interpretation N corresponding to $\widetilde{A}$ does not satisfy r'. But this is a contradiction to $N \vDash P^M$. Consequently, M is a minimal model of P^M, and thus M is an answer set of P.

Since we can express answer set existence of P in MSO, it follows from Libkin (2004, Proposition 7.12) that Even cannot be expressed with monadic answer set programs. □

Related to the result above are frameworks for specifying nonmonotonic logics in second order logic have been presented by Bogaerts et al. (2016) and Egly et al. (2000), which use suitable encodings to capture the answer sets of logic programs by Quantified Boolean Formulas.

The next example demonstrates positive mutual recursion over modules.

Example 3.2 Take, as an example, the MLP $\mathbf{P} = (m_1, m_2)$, where both modules $m_1 = (P_1[], R_1)$ with $R_1 = \{p_1 \leftarrow P_2.p_2\}$ and $m_2 = (P_2[], R_2)$ with $R_2 = \{p_2 \leftarrow P_1.p_1\}$ are main modules. Intuitively, $\mathbf{P}$ amounts to the ordinary logic program $\{a \leftarrow b;\ b \leftarrow a\}$.

3.2 Semantics of Modular Nonmonotonic Logic Programs

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

The *Herbrand base* with respect to 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 $\mathrm{HB}_{\mathbf{P}}$. The grounding of a rule r is the set $gr(r)$ of all ground instances of r with respect to $\mathcal{C}$; the grounding of a rule set R is $gr(R) = \bigcup_{r \in R} gr(r)$, and the one of a module m, $gr(m)$, is defined by replacing the rules in $R(m)$ by $gr(R(m))$; the grounding of an MLP $\mathbf{P}$ is $gr(\mathbf{P})$, which is formed by grounding each module m_i of $\mathbf{P}$.

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

Let $S \subseteq \mathrm{HB}_{\mathbf{P}}$ 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 use the notation $S|_{\mathbf{p}} = \{p_i(\mathbf{c}) \in S \mid i \in \{1, \ldots, k\}\}$ and $S|_{\mathbf{p}}^{\mathbf{q}} = \{q_i(\mathbf{c}) \mid p_i(\mathbf{c}) \in S, i \in \{1, \ldots, k\}\}$.

Definition 3.3 (Value calls).
Let $\mathbf{P}$ be an MLP. For a module name $P \in \mathcal{M}$ with associated formal input $\mathbf{q}$ we say that $P[S]$ is a *value call with input* S, where $S \subseteq \mathrm{HB}_{\mathbf{P}}|_{\mathbf{q}}$. Let $\mathrm{VC}(\mathbf{P})$ denote the set of all value calls $P[S]$ with input S such that $P \in \mathcal{M}$.

Note that $\mathrm{VC}(\mathbf{P})$ is also used as index set here. Given MLP $\mathbf{P} = (m_1, \ldots, m_n)$, $\mathrm{VC}(\mathbf{P})$ is the set of indexes with value calls of the form $P_i[S]$, where i and S is used as a combined index. Here, i ranges from 1 to n, and S is a subset of $\mathrm{HB}_{\mathbf{P}}$ restricted to atoms with predicates from the input list $\mathbf{q}_i$ of module $m_i = (P_i[\mathbf{q}_i], R_i)$. For the given module m_i, $\mathrm{VC}(\mathbf{P})$ includes $\{P_i[\varnothing], \ldots, P_i[\mathrm{HB}_{\mathbf{P}}|_{\mathbf{q}_i}]\}$. In case where $m_i = (P_i[], R_i)$ is a main module or has empty input, i.e., the input list $\mathbf{q}_i$ is void, only $P_i[\varnothing]$ remains as value call in $\mathrm{VC}(\mathbf{P})$.

Based on value calls with inputs, we define module and program instantiations.

Definition 3.4 (Module and program instantiation).
A *rule base* is an (indexed) tuple $\mathbf{R} = (R_{P[S]} \mid P[S] \in \mathrm{VC}(\mathbf{P}))$ of sets of rules $R_{P[S]}$. For a module $m_i = (P_i[\mathbf{q}_i], R_i)$ from $\mathbf{P}$, its *instantiation with* $S \subseteq \mathrm{HB}_{\mathbf{P}}|_{\mathbf{q}_i}$ is $I_{\mathbf{P}}(P_i[S]) = R_i \cup S$. For an MLP $\mathbf{P}$, its *instantiation* is the rule base $I(\mathbf{P}) = (I_{\mathbf{P}}(P_i[S]) \mid P_i[S] \in \mathrm{VC}(\mathbf{P}))$.

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

Definition 3.5 (Interpretation).
An *interpretation* $\mathbf{M}$ of an MLP $\mathbf{P}$ is an (indexed) tuple $(M_i/S \mid P_i[S] \in \mathrm{VC}(\mathbf{P}))$, where all $M_i/S \subseteq \mathrm{HB}_{\mathbf{P}}$ contain only ordinary atoms.

Each M_i/S is essentially short-cut notation for identifying the element of tuple $\mathbf{M}$ that is indexed by value call $P_i[S]$. Both i and S give us the combined index in $\mathbf{M}$ for the set of atoms that is used to interpret the instantiation $I_{\mathbf{P}}(P_i[S])$ of module m_i relative to $P_i[S]$.

Based on interpretations, we define (classical) models of MLPs.

Definition 3.6 (Model).
Let $\mathbf{P}$ be an MLP, $P_i[S]$ be a value call from $\mathrm{VC}(\mathbf{P})$, and $m_k = (P_k[\mathbf{q}_k], R_k)$ be a module from $\mathbf{P}$. An interpretation $\mathbf{M}$ of $\mathbf{P}$ is a *model* of

- a ground atom $\alpha \in \mathrm{HB}_{\mathbf{P}}$ at $P_i[S]$, denoted $\mathbf{M}, P_i[S] \vDash \alpha$, if
  - $\alpha \in M_i/S$, in case α is an ordinary atom, and if
  - $o(\mathbf{c}) \in M_k/\big((M_i/S)|_{\mathbf{p}}^{\mathbf{q}_k}\big)$, in case α is a module atom $P_k[\mathbf{p}].o(\mathbf{c})$;

- a ground rule r at $P_i[S]$ ($\mathbf{M}, P_i[S] \vDash r$), if $\mathbf{M}, P_i[S] \vDash H(r)$ or $\mathbf{M}, P_i[S] \nvDash B(r)$, where
  - $\mathbf{M}, P_i[S] \vDash H(r)$ if $\mathbf{M}, P_i[S] \vDash \alpha$ for some $\alpha \in H(r)$, and
  - $\mathbf{M}, P_i[S] \vDash B(r)$ if $\mathbf{M}, P_i[S] \vDash \alpha$ for all $\alpha \in B^+(r)$ and $\mathbf{M}, P_i[S] \nvDash \alpha$ for all $\alpha \in B^-(r)$;
- a set of ground rules R at $P_i[S]$ ($\mathbf{M}, P_i[S] \vDash R$) iff $\mathbf{M}, P_i[S] \vDash r$ for all $r \in R$; and
- a ground rule base $\mathbf{R}$ ($\mathbf{M} \vDash \mathbf{R}$) iff $\mathbf{M}, P_i[S] \vDash R_{P_i[S]}$ for all $P_i[S] \in \mathrm{VC}(\mathbf{P})$.

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

Example 3.3 Consider $\mathbf{P}$ from Example 3.2, then $\mathbf{M} = (M_1/\emptyset, M_2/\emptyset)$ is a model of $\mathbf{P}$, where $M_1/\emptyset = \{p_1\}$ and $M_2/\emptyset = \{p_2\}$. Indeed, by Definition 3.6 we have $\mathbf{M}, P_1[\emptyset] \vDash p_1$; $\mathbf{M}, P_2[\emptyset] \vDash p_2$; $\mathbf{M}, P_1[\emptyset] \vDash P_2.p_2$; $\mathbf{M}, P_2[\emptyset] \vDash P_1.p_1$; hence $\mathbf{M}, P_1[\emptyset] \vDash p_1 \leftarrow P_2.p_2$; $\mathbf{M}, P_2[\emptyset] \vDash p_2 \leftarrow P_1.p_1$; thus $\mathbf{M}, P_1[\emptyset] \vDash I_{\mathbf{P}}(P_1[\emptyset])$ and $\mathbf{M}, P_2[\emptyset] \vDash I_{\mathbf{P}}(P_2[\emptyset])$; therefore $\mathbf{M} \vDash I(\mathbf{P})$ where $I(\mathbf{P}) = (I_{\mathbf{P}}(P_1[\emptyset]), I_{\mathbf{P}}(P_2[\emptyset]))$; and finally $\mathbf{M} \vDash \mathbf{P}$.

We next proceed to define answer sets of an MLP $\mathbf{P}$. To this end, we need to compare models and single out minimal models.

Definition 3.7 (Minimal models).
For any interpretations $\mathbf{M}$ and $\mathbf{M}'$ of $\mathbf{P}$, we define that $\mathbf{M} \leq \mathbf{M}'$, if for every $P_i[S] \in \mathrm{VC}(\mathbf{P})$ 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 $\mathbf{P}$ (respectively, a ground rule base $\mathbf{R}$) is *minimal*, if $\mathbf{P}$ (respectively, $\mathbf{R}$) has no model $\mathbf{M}'$ such that $\mathbf{M}' < \mathbf{M}$. The set of all minimal models of $\mathbf{P}$ (respectively, $\mathbf{R}$) is denoted by $\mathrm{MM}(\mathbf{P})$ (respectively, $\mathrm{MM}(\mathbf{R})$).

In order to focus on relevant modules, we introduce the formal notion of a call graph.

Definition 3.8 (Call graph).
Let $\mathbf{P}$ be an MLP, $P_i[S]$ and $P_k[T]$ be value calls from $\mathrm{VC}(\mathbf{P})$, and $m_i = (P_i[\mathbf{q}_i], R_i)$ and $m_k = (P_k[\mathbf{q}_k], R_k)$ be modules from $\mathbf{P}$. The *call graph of an MLP* $\mathbf{P}$ is a labeled digraph $\mathrm{CG}_{\mathbf{P}} = (V, E, l)$ with vertex set $V = \mathrm{VC}(\mathbf{P})$ 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* $\mathrm{CG}_{\mathbf{P}}(\mathbf{M}) = (V', E')$ *of* $\mathbf{P}$ *with respect to* $\mathbf{M}$ is the subgraph of $\mathrm{CG}_{\mathbf{P}}$ where E' contains all edges from $P_i[S]$ to $P_k[T]$ of $\mathrm{CG}_{\mathbf{P}}$ such that $(M_i/S)|_{l(e)}^{\mathbf{q}_k} = T$, and V' contains all $P_i[S]$ that are main module instantiations or induced by E'; any such $P_i[S]$ is called *relevant with respect to* $\mathbf{M}$.

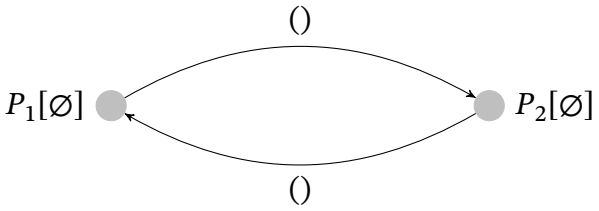


Figure 3.1: Call graph for Example 3.2

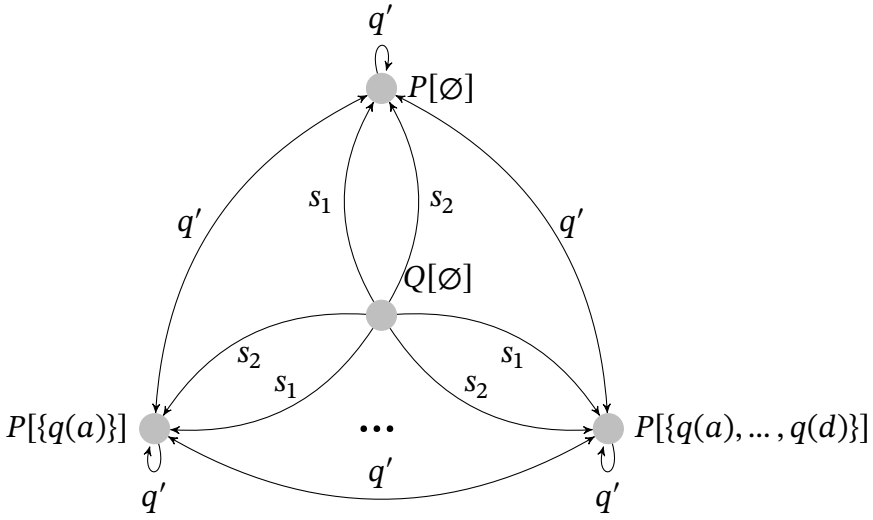


Figure 3.2: Call graph for Example 3.1

Example 3.4 Consider $\mathbf{P}$ from Example 3.2. Then $I(\mathbf{P}) = (I_{\mathbf{P}}(P_1[\varnothing]), I_{\mathbf{P}}(P_2[\varnothing]))$, and we obtain the call graph $\mathrm{CG}_{\mathbf{P}} = (\mathrm{VC}(\mathbf{P}), \{(P_1[\varnothing], P_2[\varnothing]), (P_2[\varnothing], P_1[\varnothing])\}, l)$, where l maps each edge to the void input list (see Figure 3.1). Both $P_1[\varnothing]$ and $P_2[\varnothing]$ are relevant, since they are main modules. Moreover, since both modules and all instantiations have empty input, we have that $\mathrm{CG}_{\mathbf{P}}(\mathbf{M}) = \mathrm{CG}_{\mathbf{P}}$ for any interpretation $\mathbf{M}$ of $\mathbf{P}$.

Example 3.5 Consider $\mathbf{P}$ from Example 3.1. The instantiation of $\mathbf{P}$ is

$$I(\mathbf{P}) = (I_{\mathbf{P}}(Q[\varnothing]), I_{\mathbf{P}}(P[\varnothing]), I_{\mathbf{P}}(P[\{q(a)\}]), \dots, I_{\mathbf{P}}(P[\{q(a), \dots, q(d)\}])) \quad ,$$

hence the graph shown in Figure 3.2 is the call graph of $\mathbf{P}$. The value call $Q[\varnothing]$ is always relevant (because it is main), the other value calls are only relevant in certain models. For instance, in a model $\mathbf{M} = (M_Q/\varnothing, M_P/\varnothing, M_P/\{q(a)\}, \dots)$ such that $M_P/\{q(a)\} = \{q(a), \mathit{skip}(a), \mathit{odd}\}$, we have that $P[\varnothing]$ is relevant as $(M_P/\{q(a)\})|^q_{q'} = \varnothing$.

We refer to the vertex and edge set of a graph G by $V(G)$ and $E(G)$, respectively. For defining answer sets, we use a reduct of the instantiated program as customary

in ASP. A suggestive way is to apply a traditional reduct to each module instance of $\mathbf{P}$; however, this is not fully satisfactory, as in practice $\mathbf{P}$ might contain module instantiations which have no answer sets for certain inputs, which compromises the existence of an answer set of $\mathbf{P}$. For this reason, we contextualize the notions of reduct and answer sets.

Definition 3.9 (Context-based reduct).
Let $\mathbf{M}$ be an interpretation of an MLP $\mathbf{P}$. A *context for* $\mathbf{M}$ is any set $C \subseteq \mathrm{VC}(\mathbf{P})$ such that $V(\mathrm{CG}_{\mathbf{P}}(\mathbf{M})) \subseteq C$. The *reduct of* $\mathbf{P}$ *at* $P[S]$ *with respect to* $\mathbf{M}$ *and* C is the rule set

$$f\,\mathbf{P}(P[S])^{\mathbf{M},C} = \begin{cases} \{r \in I_{gr(\mathbf{P})}(P[S]) \mid \mathbf{M}, P[S] \vDash B(r)\} & \text{if } P[S] \in C, \\ I_{gr(\mathbf{P})}(P[S]) & \text{otherwise.} \end{cases}$$

The *reduct of* $\mathbf{P}$ *with respect to* $\mathbf{M}$ *and* C is $f\,\mathbf{P}^{\mathbf{M},C} = (f\,\mathbf{P}(P[S])^{\mathbf{M},C} \mid P[S] \in \mathrm{VC}(\mathbf{P}))$.

That is, outside C the module instantiations of $\mathbf{P}$ (respectively, $gr(\mathbf{P})$) remain untouched, while inside C the FLP-reduct (Faber et al., 2011) is applied.

Definition 3.10 (Answer set).
Let $\mathbf{M}$ be an interpretation of a ground MLP $\mathbf{P}$. Then $\mathbf{M}$ is an *answer set of* $\mathbf{P}$ *with respect to a context* C *for* $\mathbf{M}$ if $\mathbf{M}$ is a minimal model of $f\,\mathbf{P}^{\mathbf{M},C}$.

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

Example 3.6 Let $\mathbf{P}$ be from Example 3.1. We have that $\mathbf{P}$ obtains answer sets of four different shapes, with each of them having exactly two instances of s_1 and two instances of s_2 for the model $M_Q/\emptyset$ of instantiation $Q[\emptyset]$. A specific answer set is

$$\big(M_Q/\emptyset, M_P/\emptyset, M_P/\{q(a)\}, M_P/\{q(b)\}, M_P/\{q(c)\}, \\ M_P/\{q(d)\}, M_P/\{q(a), q(c)\}, M_P/\{q(b), q(d)\}, \dots\big)$$

where

- $M_Q/\emptyset = \{s_1(a), s_2(b), s_1(c), s_2(d), ok, s(a), s(b), s(c), s(d)\}$,
- $M_P/\emptyset = \{even\}$,
- all models for instantiations whose input is a singleton set, i.e., for $M_P/\{q(a)\}$, $M_P/\{q(b)\}$, $M_P/\{q(c)\}$, and $M_P/\{q(d)\}$, contain *odd* and the respective *skip*'d element, and

- both $M_P/\{q(a), q(c)\}$ and $M_P/\{q(b), q(d)\}$ contain *even*.

Example 3.7 Consider $\mathbf{P}$ from Example 3.2. Let $\mathbf{M}_0 = (M_1^0/\varnothing, M_2^0/\varnothing)$ such that $M_1^0/\varnothing = M_2^0/\varnothing = \varnothing$, and $\mathbf{M}_1 = (M_1^1/\varnothing, M_2^1/\varnothing)$ such that $M_1^1/\varnothing = \{p_1\}$ and $M_2^1/\varnothing = \{p_2\}$, be interpretations for $\mathbf{P}$. One can verify that both are models of $\mathbf{P}$. Since we fixed the context C to VC($\mathbf{P}$), the reducts with respect to our models are

$$f\,\mathbf{P}^{\mathbf{M}_0} = \left(f\,\mathbf{P}(P_1[\varnothing])^{\mathbf{M}_0}, f\,\mathbf{P}(P_2[\varnothing])^{\mathbf{M}_0}\right) = (\varnothing, \varnothing)$$

and

$$f\,\mathbf{P}^{\mathbf{M}_1} = \left(f\,\mathbf{P}(P_1[\varnothing])^{\mathbf{M}_1}, f\,\mathbf{P}(P_2[\varnothing])^{\mathbf{M}_1}\right) = (I_{\mathbf{P}}(P_1[\varnothing]), I_{\mathbf{P}}(P_2[\varnothing])) \quad .$$

The minimal model of $f\,\mathbf{P}^{\mathbf{M}_0}$ is $\mathbf{M}_0$, hence it is an answer set of $\mathbf{P}$, whereas we have that the minimal model of $f\,\mathbf{P}^{\mathbf{M}_1}$ is also $\mathbf{M}_0$, i.e., $\mathbf{M}_1$ is not an answer set of $\mathbf{P}$.

A question that could arise is whether the FLP-reduct is really needed in order to obtain a well-behaved semantics for modular programs, or whether one could go with the standard GL-reduct. The main difference between the GL- and FLP-reduct is that the latter *treats all atoms as "black boxes"* (Faber et al., 2011), i.e., atoms need the definition of a proper satisfaction relation $\models$ for evaluation, whereas the GL-reduct simply needs set membership for verifying the value of an atom in an interpretation. The next example will clarify this point.

Example 3.8 Let $R_1 = \{q \leftarrow P_2[q].p\}$ and $R_2 = \{p \leftarrow q_2\}$ be two rule sets. Take, as an example, the MLP $\mathbf{P} = (m_1, m_2)$ with the main module $m_1 = (P_1[], R_1)$ and the library module $m_2 = (P_2[q_2], R_2)$. Let $\mathbf{M}_1 = (M_1^1/\varnothing, M_2^1/\varnothing, M_2^1/\{q_2\})$ and $\mathbf{M}_2 = (M_1^2/\varnothing, M_2^2/\varnothing, M_2^2/\{q_2\})$ be two interpretations for $\mathbf{P}$, where

- $M_1^1/\varnothing = M_2^1/\varnothing = M_2^2/\varnothing = \varnothing$,
- $M_2^1/\{q_2\} = M_2^2/\{q_2\} = \{p, q_2\}$, and
- $M_1^2/\varnothing = \{q\}$.

Note that $\mathbf{M}_1 < \mathbf{M}_2$. When we apply the FLP-reduct to the instantiations, we get

$$\begin{aligned} f\,\mathbf{P}(P_1[\varnothing])^{\mathbf{M}_1} &= \varnothing \\ f\,\mathbf{P}(P_1[\varnothing])^{\mathbf{M}_2} &= R_1 \\ f\,\mathbf{P}(P_2[\varnothing])^{\mathbf{M}_1} &= f\,\mathbf{P}(P_2[\varnothing])^{\mathbf{M}_2} = R_2 \\ f\,\mathbf{P}(P_2[\{q_2\}])^{\mathbf{M}_1} &= f\,\mathbf{P}(P_2[\{q_2\}])^{\mathbf{M}_2} = R_2 \cup \{q_2\}. \end{aligned}$$

We have that both $\mathbf{M}_1$ and $\mathbf{M}_2$ are models of $f\,\mathbf{P}^{\mathbf{M}_1}$ and $f\,\mathbf{P}^{\mathbf{M}_2}$, respectively. Note that only $\mathbf{M}_1$ is the single answer set of $\mathbf{P}$ as $\mathrm{MM}\left(f\,\mathbf{P}^{\mathbf{M}_1}\right) = \mathrm{MM}\left(f\,\mathbf{P}^{\mathbf{M}_2}\right) = \{\mathbf{M}_1\}$.

For any reasonable definition of GL-reduct for MLPs, $\mathbf{P}^{\mathbf{M}_1}$ and $\mathbf{P}^{\mathbf{M}_2}$, we would have that the rule $q \leftarrow P_2[q].p$ would have a particular "fixed" instance in the reduct. That is, in $\mathbf{P}^{\mathbf{M}_1}$, $P_2[q].p$ would refer to p from $P_2[\varnothing]$, and in $\mathbf{P}^{\mathbf{M}_2}$, $P_2[q].p$ would refer to p from $P_2[\{q_2\}]$. This means that $\mathbf{M}_1$ is not the minimal model of $\mathbf{P}^{\mathbf{M}_2}$ anymore, and we get that $\mathbf{M}_1$ and $\mathbf{M}_2$ are both answer sets of $\mathbf{P}$.

This shows that the FLP semantics is a reasonable choice and that the GL-transform is insensitive to positive loops in the modular setting.

3.3 Basic Semantic Properties

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

Proposition 3.2 (Conservativity)
Let R be an ordinary logic program. Then, M is an answer set of R iff $\mathbf{M} = (M_1/\varnothing \coloneqq M)$ is an answer set of the MLP (m_1), where $m_1 = (P_1[\,], R)$ is a main module and P_1 is a module name.

Proof This proposition can be easily seen as the GL-reduct R^M is equivalent to FLP-reduct $f\,(m_1)(P_1[\varnothing])^{\mathbf{M},C}$, since there is only one context $C = \{P_1[\varnothing]\}$. □

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

Lemma 3.3
If $\mathbf{M} \models f\,\mathbf{P}^{\mathbf{M},C}$ for some context C for $\mathbf{M}$, then $\mathbf{M} \models \mathbf{P}$.

Proof Let $\mathbf{M} \models f\,\mathbf{P}^{\mathbf{M},C}$ for some context C for $\mathbf{M}$. Thus, $\mathbf{M}, P_i[S] \models f\,\mathbf{P}(P_i[S])^{\mathbf{M},C}$ for all $P_i[S] \in \mathrm{VC}(\mathbf{P})$. We show now that for all $P_i[S] \in \mathrm{VC}(\mathbf{P})$, $\mathbf{M} \models I_{gr(\mathbf{P})}(P_i[S])$. Consider $P_i[S] \notin C$, then by definition we get that $f\,\mathbf{P}(P_i[S])^{\mathbf{M},C} = I_{gr(\mathbf{P})}(P_i[S])$. Hence, $\mathbf{M}, P_i[S] \models f\,\mathbf{P}(P_i[S])^{\mathbf{M},C}$ implies $\mathbf{M} \models I_{gr(\mathbf{P})}(P_i[S])$. In case $P_i[S] \in C$, the FLP-reduct is defined to be $f\,\mathbf{P}(P_i[S])^{\mathbf{M},C} = \{r \in I_{gr(\mathbf{P})}(P_i[S]) \mid \mathbf{M}, P_i[S] \models B(r)\}$. Since $\mathbf{M}, P_i[S] \models f\,\mathbf{P}(P_i[S])^{\mathbf{M},C}$, we can deduce that $\mathbf{M}, P_i[S] \models r$ for all rules r in $f\,\mathbf{P}(P_i[S])^{\mathbf{M},C}$. Now let $r \in I_{gr(\mathbf{P})}(P_i[S])$ such that $r \notin f\,\mathbf{P}(P_i[S])^{\mathbf{M},C}$. By definition of FLP-reduct we derive $\mathbf{M}, P_i[S] \not\models B(r)$, and thus $\mathbf{M}, P_i[S] \models r$. Therefore, all $r \in I_{gr(\mathbf{P})}(P_i[S])$ are satisfied at $P_i[S]$ by $\mathbf{M}$, and the result follows. □

Lemma 3.4
If $\mathbf{M} \models \mathbf{P}$, then $\mathbf{M} \models f\,\mathbf{P}^{\mathbf{M}',C}$ for any interpretation $\mathbf{M}'$ and context C.

Proof Let $\mathbf{M} \models \mathbf{P}$. By definition, $\mathbf{M} \models I(gr(\mathbf{P}))$ and for all $P_i[S] \in \mathrm{VC}(\mathbf{P})$, $\mathbf{M} \models I_{gr(\mathbf{P})}(P_i[S])$. Now let $\mathbf{M}'$ be an interpretation of $\mathbf{P}$ and $C \subseteq \mathrm{VC}(\mathbf{P})$ be a context. We have to show that $\mathbf{M} \models f\,\mathbf{P}(P_i[S])^{\mathbf{M}',C}$ for all $P_i[S] \in \mathrm{VC}(\mathbf{P})$.

Consider $P_i[S] \notin C$, then by definition $f\,\mathbf{P}(P_i[S])^{\mathbf{M}',C} = I_{gr(\mathbf{P})}(P_i[S])$. From $\mathbf{M} \models \mathbf{P}$ we derive $\mathbf{M} \models I_{gr(\mathbf{P})}(P_i[S])$. Hence, $\mathbf{M} \models f\,\mathbf{P}(P_i[S])^{\mathbf{M}',C}$. For the case $P_i[S] \in C$, by definition $f\,\mathbf{P}(P_i[S])^{\mathbf{M}',C} = \{r \in I_{gr(\mathbf{P})}(P_i[S]) \mid \mathbf{M}', P_i[S] \models B(r)\}$. Hence, we have that the FLP-reduct $f\,\mathbf{P}(P_i[S])^{\mathbf{M}',C} \subseteq I_{gr(\mathbf{P})}(P_i[S])$. Since $\mathbf{M} \models I_{gr(\mathbf{P})}(P_i[S])$ we conclude that $\mathbf{M} \models f\,\mathbf{P}(P_i[S])^{\mathbf{M}',C}$. Hence, for all $P_i[S]$, we have that $\mathbf{M} \models f\,\mathbf{P}(P_i[S])^{\mathbf{M}',C}$, and thus $\mathbf{M} \models f\,\mathbf{P}^{\mathbf{M}',C}$. □

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

Proposition 3.5 (Minimal models)
If $\mathbf{M}$ is an answer set of $\mathbf{P}$ with respect to context C, then $\mathbf{M} \in \mathrm{MM}(\mathbf{P})$.

Proof Let $\mathbf{M}$ be an answer set of $\mathbf{P}$ with respect to C. Then $\mathbf{M} \in \mathrm{MM}\left(f\,\mathbf{P}^{\mathbf{M},C}\right)$, which implies that $\mathbf{M}$ is a model of $\mathbf{P}$ by Lemma 3.3. We prove that it is a minimal model of $\mathbf{P}$. Towards a contradiction assume that $\mathbf{M} \notin \mathrm{MM}(\mathbf{P})$. Then, there exists $\mathbf{M}' < \mathbf{M}$, such that $\mathbf{M}' \models \mathbf{P}$. By Lemma 3.4, we conclude that $\mathbf{M}' \models f\,\mathbf{P}^{\mathbf{M},C}$. However, this is a contradiction to $\mathbf{M} \in \mathrm{MM}\left(f\,\mathbf{P}^{\mathbf{M},C}\right)$. Therefore, $\mathbf{M}$ is a minimal model of $\mathbf{P}$. □

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

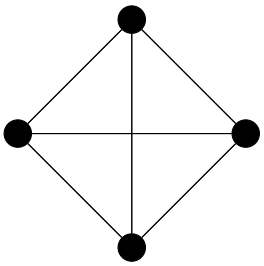
Proposition 3.6 (Context refinement)
If $\mathbf{M}$ is an answer set of $\mathbf{P}$ with respect to context $C \subseteq \mathrm{VC}(\mathbf{P})$, then $\mathbf{M}$ is an answer set of $\mathbf{P}$ with respect to every context $C' \subseteq C$ for $\mathbf{M}$, i.e., $V(\mathrm{CG}_{\mathbf{P}}(\mathbf{M})) \subseteq C' \subseteq C$.

Proof Towards a contradiction, assume that $\mathbf{M}$ is an answer set of $\mathbf{P}$ with respect to context C, but not with respect to context C' for $\mathbf{M}$, $V(\mathrm{CG}_{\mathbf{P}}(\mathbf{M})) \subseteq C' \subseteq C$. Since $\mathbf{M} \models \mathbf{P}$, we conclude that there exists $\mathbf{M}' < \mathbf{M}$, such that $\mathbf{M}' \models f\,\mathbf{P}^{\mathbf{M},C'}$. We prove that $\mathbf{M}' \models f\,\mathbf{P}^{\mathbf{M},C}$. Consider any $P_i[S] \in C$. If $P_i[S]$ is in C', then $f\,\mathbf{P}(P_i[S])^{\mathbf{M},C} = f\,\mathbf{P}(P_i[S])^{\mathbf{M},C'}$, otherwise $f\,\mathbf{P}(P_i[S])^{\mathbf{M},C} \subseteq f\,\mathbf{P}(P_i[S])^{\mathbf{M},C'}$. Therefore, in both cases $\mathbf{M}', P_i[S] \models f\,\mathbf{P}(P_i[S])^{\mathbf{M},C'}$ implies $\mathbf{M}', P_i[S] \models r$ for all $r \in f\,\mathbf{P}(P_i[S])^{\mathbf{M},C}$. This proves that $\mathbf{M}' \models f\,\mathbf{P}^{\mathbf{M},C}$, and since $\mathbf{M}' < \mathbf{M}$, this contradicts the assumption that $\mathbf{M}$ is an answer set of $\mathbf{P}$ with respect to context C. Hence, $\mathbf{M}$ is an answer set of $\mathbf{P}$ with respect to context C' as well. □

Finally, it would be appreciated if we would have a syntactic property that face no inconsistency in the scope of instantiations that are relevant to them. Let $ord(\mathbf{P})$ denote the result of deleting from an MLP $\mathbf{P}$ every rule that contains some module atom. If

$V(\mathrm{CG}_{\mathbf{P}}(\mathbf{M}')) \subseteq C$ for all $\mathbf{M}' \leq \mathbf{M}$ such that $\mathbf{M}' \vDash f\ ord(\mathbf{P})^{\mathbf{M},C}$, i.e., each decrease of $\mathbf{M}$ to a model $\mathbf{M}'$ of the ordinary rules in the reduct does not lead to a call of an instance outside the scope C then we would have a *safe scope* that gives us the desired property, but this is difficult to check and thus a computationally hard problem.

From now on, we do not consider contextual reducts of MLPs $\mathbf{P}$ and focus on the case where the context ranges over the whole set of value calls $\mathrm{VC}(\mathbf{P})$.



4

Semantic Properties of Modular Nonmonotonic Logic Programs

Semantic properties of MLPs provide the basis to find attractive features like lower computational complexity, which can range from P-complete to 2EXP-complete in general. We will postpone the concrete elaboration of details on the computational complexity landscape of modular nonmonotonic logic programs to the following Chapter 5, and concentrate in this chapter on the inspection of several program classes that have a unique model property.

The first class of such programs are Horn Modular Nonmonotonic Logic Programs, which will be defined in §4.1. We will show that they possess a canonical answer set, which equals their unique minimal model, as models of Horn MLPs are closed under model intersection.

Then, we will look into bottom up fixed-point computation for Horn programs in §4.2, and define an appropriate monotone and continuous operator $T_{\mathbf{P}}$ for Horn MLPs $\mathbf{P}$, such that applying the Kleene Fixed-Point Theorem establishes that the least fixpoint of $T_{\mathbf{P}}$ gives us the unique answer set of $\mathbf{P}$.

In §4.3, we will define stratified MLPs, which extend Horn MLPs, and show that such programs have a unique answer set that can be computed in ω steps for the limit ordinal ω. To this end, we will define an appropriate notion of program stratification and define the $T^L_{\mathbf{P}}$ operator, which provides the means for iterated fixed-point computation of the answer set for stratified MLPs.

4.1 Horn Modular Nonmonotonic Logic Programs

Recall that an MLP $\mathbf{P}$ is called Horn if each module of $\mathbf{P}$ only consists of Horn rules, i.e., rules whose body consists only of positive (module) literals and one atom in the head. Obviously, answer sets coincide with the naive semantics if $V(\mathrm{CG}_{\mathbf{P}}(\mathbf{M})) = \mathrm{VC}(\mathbf{P})$ for

all interpretations $\mathbf{M}$ of $\mathbf{P}$, in particular, when all modules are main. Moreover, also for positive MLPs the semantics coincides with the naive semantics. Just like in ordinary logic programs, it behaves like the minimal model semantics in absence of negation.

Proposition 4.1 (Minimal models in positive MLPs)
Let $\mathbf{P}$ be positive. Then, the answer sets of $\mathbf{P}$ coincide with $\mathrm{MM}(\mathbf{P})$.

Proof According to Proposition 3.5, every answer set of $\mathbf{P}$ is a minimal model of $\mathbf{P}$. We prove the converse direction for positive $\mathbf{P}$. Let $\mathbf{M} \in \mathrm{MM}(\mathbf{P})$. Then, by Lemma 3.4, $\mathbf{M} \models f\,\mathbf{P}^{\mathbf{M}}$. Towards a contradiction assume that there exists $\mathbf{M}' < \mathbf{M}$, such that $\mathbf{M}' \models f\,\mathbf{P}^{\mathbf{M}}$. Then $\mathbf{M}', P_i[S] \models I_{gr(\mathbf{P})}(P_i[S])$ for all $P_i[S] \in \mathrm{VC}(\mathbf{P}) \setminus V(\mathrm{CG}_{\mathbf{P}}(\mathbf{M}))$. Moreover, if $P_i[S] \in V(\mathrm{CG}_{\mathbf{P}}(\mathbf{M}))$, then $\mathbf{M}', P_i[S] \models f\,\mathbf{P}(P_i[S])^{\mathbf{M}}$ and $\mathbf{M}', P_i[S] \not\models B(r)$ for all $r \in I_{gr(\mathbf{P})}(P_i[S]) \setminus f\,\mathbf{P}(P_i[S])^{\mathbf{M}}$. Therefore, we conclude that $\mathbf{M}' \models I(\mathbf{P})$, i.e., $\mathbf{M}' \models \mathbf{P}$. Since $\mathbf{M}' < \mathbf{M}$, this contradicts the assumption that $\mathbf{M} \in \mathrm{MM}(\mathbf{P})$. Hence, $\mathbf{M}$ is an answer set of $\mathbf{P}$. □

By monotonicity of all module instances, one can easily show that the models of a Horn MLP $\mathbf{P}$ are closed under a suitable notion of intersection.

Definition 4.1 (Intersection).
Given two interpretations $\mathbf{M}$ and $\mathbf{N}$ of the MLP $\mathbf{P} = (m_1, \ldots, m_n)$, let their *intersection* be the interpretation denoted $\mathbf{M} \cap \mathbf{N}$ such that

$$(M \cap N)_i/S = \bigcap_{S' \supseteq S} (M_i/S' \cap N_i/S') \ ,$$

for every $S \subseteq \mathrm{HB}_{\mathbf{P}}|_{\mathbf{q}_i}$ and $i = 1, \ldots, n$.

Thus, the intersection $\mathbf{M} \cap \mathbf{N}$ builds a component-wise intersection for all supersets S' of input S. Note that the intersection reduces to $(M \cap N)_i/S = M_i/S \cap N_i/S$ if $S \subseteq S'$ implies $M_i/S \subseteq M_i/S'$ and $N_i/S \subseteq N_i/S'$. This generalizes the usual intersection of Horn logic programs for MLPs with empty inputs, as in this case the condition $S' \supseteq S$ requires that $S = S' = \emptyset$ and thus $(M \cap N)_i/\emptyset = M_i/\emptyset \cap N_i/\emptyset$.

Based on intersection, we can prove the following statement.

Proposition 4.2 (Model intersection)
Suppose $\mathbf{M} \models \mathbf{P}$ and $\mathbf{N} \models \mathbf{P}$, where $\mathbf{P}$ is Horn. Then $\mathbf{M} \cap \mathbf{N} \models \mathbf{P}$.

Proof Towards a contradiction, assume $\mathbf{M} \models \mathbf{P}$ and $\mathbf{N} \models \mathbf{P}$, but $\mathbf{M} \cap \mathbf{N} \not\models \mathbf{P}$. Hence, there exists some $P_i[S] \in \mathrm{VC}(\mathbf{P})$ and some rule $r \in I_{gr(\mathbf{P})}(P_i[S])$ such that $\mathbf{M} \cap \mathbf{N}, P_i[S] \models B(r)$ and $\mathbf{M} \cap \mathbf{N}, P_i[S] \not\models H(r)$. By definition,

$$(M \cap N)_i/S = \bigcap_{S' \supseteq S} (M_i/S' \cap N_i/S') \ ,$$

and as $H(r)$ is an ordinary atom, without loss of generality $H(r) \notin M_i/S^*$ for some $S^* \supseteq S$, i.e., $\mathbf{M}, P_i[S^*] \not\models H(r)$. However, for every ordinary atom $\alpha \in B(r)$ it holds that $\mathbf{M}, P_i[S^*] \models \alpha$. We show that also for every module atom $\alpha = P_j[\mathbf{p}].o(\mathbf{c})$ in $B(r)$, $\mathbf{M}, P_i[S^*] \models \alpha$ holds.

Let $S_\mathbf{p} = (M \cap N)_i/S|_\mathbf{p}$ be the input value of α in $P_i[S]$. As $\mathbf{M} \cap \mathbf{N}, P_i[S] \models \alpha$, $o(\mathbf{c}) \in (M \cap N)_j/S_\mathbf{p}$ holds. Now let $S^*_\mathbf{p} = (M \cap N)_i/S^*|_\mathbf{p}$ be the input value of α in $P_i[S^*]$. It holds that $S^*|_\mathbf{p} \supseteq S|_\mathbf{p}$; hence, by definition of $\mathbf{M} \cap \mathbf{N}$, $M_j/S^*|_\mathbf{p} \supseteq (M \cap N)_j/S|_\mathbf{p}$; the latter means that $o(\mathbf{c}) \in M_j/S^*|_\mathbf{p}$, and hence $\mathbf{M}, P_i[S^*] \models \alpha$.

In summary, this shows that $\mathbf{M}, P_i[S^*] \models B(r)$. From $\mathbf{M} \models \mathbf{P}$, it follows that $\mathbf{M}, P_i[S^*] \models H(r)$, which is a contradiction. □

Proposition 4.2 generalizes for any collection $\mathbb{M}$ of models for $\mathbf{P}$ such that

$$\left(\bigcap_{\mathbf{M},\mathbf{N} \in \mathbb{M}} \mathbf{M} \cap \mathbf{N} \right) \models \mathbf{P} \ .$$

As a consequence, a Horn MLP has a canonical answer set. In the following, we say that a given model $\mathbf{M}$ of an MLP $\mathbf{P}$ is called *least model* iff $\mathbf{M} \leq \mathbf{N}$ for all models $\mathbf{N}$ of $\mathbf{P}$.

Corollary 4.3 (Canonical model)
If $\mathbf{P}$ is Horn, then it has a unique answer set, which coincides with its least model.

4.2 Fixed-Point Characterization

Like for ordinary programs, we can compute the answer set of a Horn MLP by means of a bottom up fixed-point computation. With this end in mind, we provide formal definitions from lattice theory next (Gierz et al., 2003).

Definition 4.2 (Partially ordered set).
A *partial order* is a binary relation $\leq$ over a set V that satisfies for all a, b, c in V

- $a \leq a$ (reflexivity);
- if $a \leq b$ and $b \leq a$, then $a = b$ (antisymmetry); and
- if $a \leq b$ and $b \leq c$, then $a \leq c$ (transitivity).

A set V with a partial order $\leq$ is called *partially ordered set* $(V, \leq)$.

Definition 4.3 (Lower and upper bounds).
A *lower bound* (respectively, *upper bound*) of a subset W of a partially ordered set $(V, \leq)$ is an element a of V such that for all $x \in W$, $a \leq x$ (respectively, $a \geq x$). A lower bound (respectively, upper bound) a of W is called *greatest lower bound* (respectively, *least upper bound*) of W if for all lower bounds (respectively, upper bounds) y of W in V, $y \leq a$ (respectively, $y \geq a$).

Definition 4.4 (Lattice).
A *complete lattice* is a partially ordered set $(V, \leq)$ such that each subset $W \subseteq V$ has a least upper bound $\text{lub}(W)$ and a greatest lower bound $\text{glb}(W)$.

For our purposes, the partially ordered set $(V, \leq)$, where V is the set of all MLP interpretations of a program $\mathbf{P}$ and $\leq$ is defined as in Definition 3.7, is a complete lattice. An *operator* on a complete lattice $(V, \leq)$ is a mapping $T\colon V \to V$.

We can now define an operator for Horn MLPs, which is used for fixed-point computation.

Definition 4.5 (Immediate consequence operator for Horn MLPs).
Given a Horn MLP $\mathbf{P}$ and an interpretation $\mathbf{M}$ of $\mathbf{P}$, we define the operator $T_{\mathbf{P}}(\mathbf{M})$ componentwise as follows:

$$T_{\mathbf{P}}(\mathbf{M}) = \left(T_{\mathbf{P}}(\mathbf{M})_{P_i[S]} \mid P_i[S] \in \text{VC}(\mathbf{P})\right) \ ,$$

where

$$T_{\mathbf{P}}(\mathbf{M})_{P_i[S]} = M_i/S \cup \left\{H(r) \ \mid \ r \in I_{gr(\mathbf{P})}(P_i[S]) \text{ and } \mathbf{M}, P_i[S] \models B(r)\right\} \ .$$

The $T_{\mathbf{P}}$ operator for a Horn MLP $\mathbf{P}$ is the inflationary variant (Abiteboul et al., 1995) for the T_P operator for ordinary Horn programs P (Emden and Kowalski, 1976; Lloyd, 1987), and generalizes T_P in order to take module input into account. Where T_P is defined for Herbrand interpretations, $T_{\mathbf{P}}$ is defined for interpretations over MLPs $\mathbf{P}$. The main distinction is to introduce module instantiations into its definition and the pointwise application of $T_{\mathbf{P}}$, essentially splitting $T_{\mathbf{P}}$ based on value calls $P_i[S] \in \text{VC}(\mathbf{P})$.

In the following, we will show that $T_{\mathbf{P}}$ is a monotone and continuous operator. We start with the former property and define monotone operators.

Definition 4.6 (Monotone operators).
An operator $T\colon V \to V$ on partially ordered set $(V, \leq)$ is *monotone*, if for all $x, y \in V$,

$$x \leq y \text{ implies } T(x) \leq T(y) \ .$$

Proposition 4.4 (Monotonicity)
The $T_{\mathbf{P}}$ operator for a Horn MLP $\mathbf{P}$ is monotone.

Proof Let $\mathbf{M}$ and $\mathbf{N}$ be interpretations for $\mathbf{P}$ such that $\mathbf{M} \leq \mathbf{N}$. We show now that $T_{\mathbf{P}}(\mathbf{M}) \leq T_{\mathbf{P}}(\mathbf{N})$. Without loss of generality, let $M_i/S \subseteq N_i/S$ for a value call $P_i[S]$ of $\mathbf{P}$, and let $M_j/T = N_j/T$ for all value calls $P_j[T] \neq P_i[S]$. We get that

$$\left\{H(r) \ \mid \ r \in I_{gr(\mathbf{P})}(P_i[S]) \text{ and } \mathbf{M}, P_i[S] \models B(r)\right\} \subseteq \left\{H(r) \ \mid \ r \in I_{gr(\mathbf{P})}(P_i[S]) \text{ and } \mathbf{N}, P_i[S] \models B(r)\right\} \ ,$$

as $\mathbf{P}$ is Horn and $M_i/S \subseteq N_i/S$. Therefore,

$$M_i/S \cup \{H(r) \mid r \in I_{gr(\mathbf{P})}(P_i[S]) \text{ and } \mathbf{M}, P_i[S] \models B(r)\} \subseteq N_i/S \cup \{H(r) \mid r \in I_{gr(\mathbf{P})}(P_i[S]) \text{ and } \mathbf{N}, P_i[S] \models B(r)\} \ ,$$

and this implies that $T_{\mathbf{P}}(\mathbf{M}) \leq T_{\mathbf{P}}(\mathbf{N})$. □

Monotone operators enjoy useful fixpoint properties, such as the following.

Knaster-Tarski Theorem (Tarski, 1955)Any monotone operator T on a complete lattice $(V, \leq)$ has a least fixpoint

$$\text{lfp}(T) = \text{glb}(\{x \in V \mid T(x) \leq x\}) \ .$$

To show that $T_{\mathbf{P}}$ is also continuous, we need to define continuous operators. We start with directed sets of partially ordered sets.

Definition 4.7 (Directed sets).
Given the partially ordered set $(V, \leq)$, we call a nonempty subset $W \subseteq V$ *directed*, if for each pair $x, y \in W$ there exists some $z \in W$ such that $x \leq z$ and $y \leq z$.

The least upper bound of a directed set W is contained in W.

Lemma 4.5
Let $(V, \leq)$ be a partially ordered set and $W \subseteq V$. If W is directed then $\text{lub}(W) \in W$.

Proof Towards a contradiction, assume that $\text{lub}(W) \notin W$. From $\text{lub}(W)$ being an upper bound, we can infer that there must exist $x, y \in W$ with $x \leq \text{lub}(W)$ and $y \leq \text{lub}(W)$ such that $x \not\leq y$ and $y \not\leq x$ and there is no $z \in W$ such that $x \leq z$ and $y \leq z$. Hence, x and y are two upper bounds in W. Then, by W being directed, we have that for all $u, v \in W$ there exists $z \in W$ such that $u \leq z$ and $v \leq z$. Now both x and y do not have a z in W such that $x \leq z$ and $y \leq z$, which contradicts our assumption that $\text{lub}(W) \notin W$. □

Definition 4.8 (Continuous operator).
An operator $T \colon V \to V$ on a complete lattice $(V, \leq)$ is *continuous*, if for every directed set $W \subseteq V$,

$$T(\text{lub}(W)) = \text{lub}(T(W)) \ ,$$

where $T(W) = \{T(x) \mid x \in W\}$.

Intuitively, directed models converge, as we can build a chain $\mathbf{M}_0 < \mathbf{M}_1 < \cdots$. Note that continuous operators are also monotone.

The following Lemma is useful for proving that $T_{\mathbf{P}}$ is a continuous operator.

Lemma 4.6
Let $\mathbf{P}$ be a Horn MLP, let W be a directed set of interpretations for $\mathbf{P}$, let $P_i[S] \in \mathrm{VC}(\mathbf{P})$, and let $B = \{b_1, \dots, b_n\}$ be a set of atoms. Then, $\mathrm{lub}(W), P_i[S] \models B$ iff $\mathbf{M}, P_i[S] \models B$ for some $\mathbf{M} \in W$.

Proof ($\Rightarrow$) Let $\mathrm{lub}(W), P_i[S] \models B$. By Lemma 4.5, we obtain that $\mathrm{lub}(W) \in W$ and thus we immediately get that $\mathbf{M}, P_i[S] \models B$ for some $\mathbf{M} \in W$.

($\Leftarrow$) Let $\mathbf{M}, P_i[S] \models B$ for some $\mathbf{M} \in W$. We have that $\mathbf{N} \leq \mathrm{lub}(W)$ for all $\mathbf{N} \in W$, and thus $\mathbf{M} \leq \mathrm{lub}(W)$. Hence, $\mathrm{lub}(W), P_i[S] \models B$. □

We can now show the following.

Proposition 4.7 (Continuous operator)
The $T_{\mathbf{P}}$ operator for a Horn MLP $\mathbf{P}$ is continuous.

Proof Let W be a directed set of interpretations, and let $P_i[S] \in \mathrm{VC}(\mathbf{P})$. We show that $T_{\mathbf{P}}(\mathrm{lub}(W)) = \mathrm{lub}(T_{\mathbf{P}}(W))$. Then,

$$
\begin{aligned}
& T_{\mathbf{P}}(\mathrm{lub}(W)), P_i[S] \models a \\
& \iff a \leftarrow b_1, \dots, b_n \in I_{gr(\mathbf{P})}(P_i[S]) \text{ and} \\
& \quad \mathrm{lub}(W), P_i[S] \models b_j \text{ for} j \in \{1, \dots, n\} && \text{by Definition 4.5} \\
& \iff a \leftarrow b_1, \dots, b_n \in I_{gr(\mathbf{P})}(P_i[S]) \text{ and for some } \mathbf{M} \in W, \\
& \quad \mathbf{M}, P_i[S] \models b_j \text{ for } j \in \{1, \dots, n\} && \text{by Lemma 4.6} \\
& \iff T_{\mathbf{P}}(\mathbf{M}), P_i[S] \models a \text{ for some } \mathbf{M} \in W && \text{by Definition 4.5} \\
& \iff \mathrm{lub}(T_{\mathbf{P}}(W)), P_i[S] \models a && \text{by Definition 4.4 .}
\end{aligned}
$$
□

A stronger Theorem than Knaster-Tarski Theorem holds for continuous operators.

Kleene Fixed-Point Theorem (Kleene, 1952)Any continuous operator T on a complete lattice $(V, \leq)$ has a least fixpoint

$$\mathrm{lfp}(T) = \mathrm{lub}(\{T^i \mid i \geq 0\}) \ ,$$

where $T^0 = \mathrm{glb}(V)$ and $T^{i+1} = T(T^i)$, for all integers $i \geq 0$.

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

Lemma 4.8
An interpretation $\mathbf{M}$ is a pre-fixpoint of $T_{\mathbf{P}}$ iff $\mathbf{M}$ is a model of $\mathbf{P}$.

Proof ($\Rightarrow$) Let $T_{\mathbf{P}}(\mathbf{M}) \leq \mathbf{M}$. For all $P_i[S] \in \mathrm{VC}(\mathbf{P})$ and all $r \in I_{gr(\mathbf{P})}(P_i[S])$, it holds that $\mathbf{M}, P_i[S] \models B(r)$ implies $\mathbf{M}, P_i[S] \models H(r)$. Thus, $\mathbf{M} \models r$ for all rules r appearing in $I(\mathbf{P})$ and so we can conclude that $\mathbf{M}$ is a model of $\mathbf{P}$.

($\Leftarrow$) Let $\mathbf{M}$ be a model of $\mathbf{P}$. Thus, $\mathbf{M} \models r$ for all $r \in I_{gr(\mathbf{P})}(P_i[S])$ and all $P_i[S] \in \mathrm{VC}(\mathbf{P})$, and thus for any r in $I_{\mathbf{P}}(P_i[S])$, $\mathbf{M}, P_i[S] \models B(r)$ implies $\mathbf{M}, P_i[S] \models H(r)$. Hence, $T_{\mathbf{P}}(\mathbf{M}) \leq \mathbf{M}$. □

We obtain the following result.

Proposition 4.9 (Least fix point)
For a Horn MLP $\mathbf{P}$, $\mathrm{lfp}(\mathbf{P})$ is the unique answer set of $\mathbf{P}$.

Proof Since $\mathrm{lfp}(\mathbf{P}) = T_{\mathbf{P}}\uparrow_{\omega}(\mathbf{M}_{\emptyset})$ is a fixpoint of $T_{\mathbf{P}}$, it is also a pre-fixpoint and by Lemma 4.8 it is a model of $\mathbf{P}$. We show now that $\mathbf{M} = \mathrm{lfp}(\mathbf{P})$ is also the least model of $\mathbf{P}$. Let $\mathbf{N}$ be an interpretation such that $\mathbf{M} \leq \mathbf{N}$ and let $\mathbf{M}^k = T_{\mathbf{P}}\uparrow_k(\mathbf{M}_{\emptyset})$ for all $k \geq 0$. We show that if $\mathbf{N} \models \mathbf{P}$ then $\mathbf{M}^i \leq \mathbf{N}$ for all integers $i \geq 0$. We proceed by induction on i. In the base case, we set $i = 0$ and obtain that $\mathbf{M}^0 = \mathbf{M}_{\emptyset}$, hence $\mathbf{M}^0 \leq \mathbf{N}$. For the inductive step, let $i > 0$ and let $\mathbf{M}^i \leq \mathbf{N}$ be our inductive hypothesis. We show now that $\mathbf{M}^{i+1} \leq \mathbf{N}$. By definition, $\mathbf{M}^{i+1} = T_{\mathbf{P}}\uparrow_{i+1}(\mathbf{M}_{\emptyset}) = T_{\mathbf{P}}(T_{\mathbf{P}}\uparrow_i(\mathbf{M}_{\emptyset})) = T_{\mathbf{P}}(\mathbf{M}^i)$, and so we derive $\mathbf{M}^i \leq \mathbf{M}^{i+1}$. From $\mathbf{M}$ being a fixpoint of $T_{\mathbf{P}}$, we conclude that $\mathbf{M}^i \leq \mathbf{M}$ and $\mathbf{M}^{i+1} \leq \mathbf{M}$. As a consequence of $\mathbf{M} \leq \mathbf{N}$, we can now infer $\mathbf{M}^{i+1} \leq \mathbf{N}$, what was to be shown. □

4.3 Stratified Modular Nonmonotonic Logic Programs

A useful class of ordinary logic programs are called stratified logic programs, which are normal logic programs that forbid recursion over negative literals (Apt et al., 1988). Stratified programs extend Horn programs: both enjoy having a unique model that can be computed by iterated application of a suitable operator, but unlike Horn programs, stratified programs do allow a "safe use" of negation in the body. For this purpose, Apt et al. (1988) define stratifications for normal logic programs P, and if there exists a stratification for P, then P is called a stratified program.

It is thus worthwhile to define stratified MLPs with similar properties as stratified ordinary logic programs. Necessary to that end is to define stratifications for normal MLPs. We will thus generalize this concept as follows. Intuitively, the usual notion of the dependency graph of a program is extended by nodes standing for the module atoms appearing in $\mathbf{P}$, which serve to take care of the dependencies between input to the module and module output. Furthermore, we assume that each predicate occurs in ordinary atoms of at most one module.

Definition 4.9 (Dependency graph).
Let $\mathbf{P} = (m_1, \dots, m_n)$ be an MLP. The *dependency graph of* $\mathbf{P}$ is the following directed graph $G_\mathbf{P} = (V, E)$. The vertex set V contains all $p \in \mathcal{P} \cup \mathcal{E}$, with p appearing somewhere in $\mathbf{P}$, and $\mathcal{E}$ is the set of module atoms in $\mathbf{P}$. The edge set E is as follows:

- Let $r \in R(m_i)$. There is a $\star$-edge $p \to^\star q$ in $G_\mathbf{P}$, $\star \in \{+, -\}$, if one of items 1–3 holds:
  1. $p(\mathbf{t}_1) \in H(r)$ and $q(\mathbf{t}_2) \in B^\star(r)$;
  2. $p(\mathbf{t}_1), q(\mathbf{t}_2) \in H(r)$ and $\star = -$; and
  3. $p(\mathbf{t}_1) \in H(r)$ and q is a module atom in $B^\star(r)$.
- Let $\alpha \in \mathcal{E}$ be of the form $P_j[\mathbf{p}].o(\mathbf{t})$ in $R(m_i)$. There is a $+$-edge $a \to^+ b$ in $G_\mathbf{P}$ if one of items 4–6 holds:
  4. $a = \alpha$ and $b = o$;
  5. $a = \alpha$ and b appears in $\mathbf{q}_j$ of $P_j[\mathbf{q}_j]$; or
  6. $a = q_\ell$ and $b = p_\ell$, where q_ℓ appears in $\mathbf{q}_j$ of $P_j[\mathbf{q}_j]$ and p_ℓ appears in $\mathbf{p}$.

Based on the dependency graph of an MLP, we can now defined stratified MLPs.

Definition 4.10 (Stratified MLP).
We say that an MLP $\mathbf{P}$ is *stratified* if no cycle in $G_\mathbf{P}$ has $-$-edges.

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

Let k be the maximal value assigned by a particular such labeling function, and let for $0 \leq i \leq k$ the set $St_i = \{a \in \mathrm{HB}_\mathbf{P} \mid l(a) = i\}$ be a stratum for $\mathbf{P}$. Then a partitioning $St_0, \dots, St_k$ of $\mathrm{HB}_\mathbf{P}$ is a stratification.

Example 4.1 Let $\mathbf{P} = (m_1, m_2, m_3)$ be an MLP with modules

$$
\begin{aligned}
P_1[]: \quad & a_1 \leftarrow \mathrm{not}\, b_1 \\
& c_1 \leftarrow P_3[a_1].a_3 \\
P_2[]: \quad & a_2 \leftarrow \mathrm{not}\, P_1.b_1 \\
P_3[q_3]: \quad & a_3 \leftarrow q_3
\end{aligned}
$$

The unique answer set $\mathbf{M}$ is determined by

$$M_1/\emptyset = \{a_1, c_1\} \qquad M_2/\emptyset = \{a_2\} \qquad M_3/\emptyset = \emptyset$$
$$M_3/\{q_3\} = \{a_3, q_3\}$$

The dependency graph of $\mathbf{P}$ is $G_\mathbf{P} = (V, E)$, where $V = \{a_1, b_1, c_1, a_2, a_3, q_3\}$ and E consists of the edges

$$\begin{aligned} a_1 &\to^- b_1 \\ c_1 &\to^+ P_3[a_1].a_3 \\ P_3[a_1].a_3 &\to^+ a_3 \\ P_3[a_1].a_3 &\to^+ q_3 \\ q_3 &\to^+ a_1 \\ a_2 &\to^- P_1.b_1 \\ P_1.b_1 &\to^+ b_1 \\ a_3 &\to^+ q_3 \end{aligned}$$

We have a stratification $St_0 = \{b_1, P_1.b_1\}$, $St_1 = \{a_1, c_1, P_3[a_1].a_3, a_2, q_3, a_3\}$ of $\mathrm{HB}_\mathbf{P}$.

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

Definition 4.11 (Immediate consequence operator for stratified MLPs).
Given a normal MLP $\mathbf{P}$, a subset L of $\mathrm{HB}_\mathbf{P}$, and an interpretation $\mathbf{M}$ of $\mathbf{P}$, we define the operator $T_\mathbf{P}^L(\mathbf{M})$ as the operator $T_\mathbf{P}(\mathbf{M})$, where $T_\mathbf{P}(\mathbf{M})_{P_i[S]}$ has been replaced with

$$T_\mathbf{P}^L(\mathbf{M})_{P_i[S]} = M_i/S \cup \{H(r) \in L \mid r \in I_{gr(\mathbf{P})}(P_i[S]) \text{ and } \mathbf{M}, P_i[S] \models B(r)\} \quad .$$

Thus, compared to $T_\mathbf{P}$, the $T_\mathbf{P}^L$-operator has a certain set L as an additional parameter used to project those atoms from $\mathrm{HB}_\mathbf{P}$ that belong L, i.e., $T_\mathbf{P} = T_\mathbf{P}^{\mathrm{HB}_\mathbf{P}}$ and for a subset $L \subseteq \mathrm{HB}_\mathbf{P}$, we have $T_\mathbf{P}^L(\mathbf{M})_{P_i[S]} = M_i/S \cup \left(T_\mathbf{P}(\mathbf{M})_{P_i[S]} \cap L\right)$.

By $T_\mathbf{P}^L{\uparrow_\omega}(\mathbf{M})$, we denote the application of $T_\mathbf{P}^L$ in ω steps, starting with $\mathbf{M}$. Furthermore, let $\mathbf{M}^0 = \mathbf{M}_\emptyset$ be the empty interpretation, i.e., where $M_i/S = \emptyset$ for every value call $P_i[S] \in \mathrm{VC}(\mathbf{P})$. For any stratification $St_0, \ldots, St_k$ of $\mathrm{HB}_\mathbf{P}$ such that k is the maximal value assigned by a labelling function $l\colon \mathrm{HB}_\mathbf{P} \to \mathbb{N}$, we let $L_i = \bigcup_{0 \le j \le i} St_j$ and inductively define $\mathbf{M}^{i+1} = T_\mathbf{P}^{L_{i+1}}{\uparrow_\omega}(\mathbf{M}^i)$, for $0 \le i < k$.

Proposition 4.10 (Stratified answer set)
Let $\mathbf{P}$ be normal and stratified. Then $\mathbf{M}^k$ is an answer set of $\mathbf{P}$, for any stratification $St_0, \ldots, St_k$ of $\mathrm{HB}_\mathbf{P}$.

Proof Let $m_i = (P_i[\mathbf{q}_i], R_i)$ be a ground module of $\mathbf{P}$. For a $j \leq k$, we denote by $m_i^j = (P_i[\mathbf{q}_i], R_i^j)$, where $R_i^j = \{r \in R_i \mid H(r) \in L_j\}$, and by $\mathbf{P}^j = (m_1^j, \ldots, m_n^j)$.

We show now that $\mathbf{M}^k$ is an answer set of $\mathbf{P}^k$ $(= \mathbf{P})$. We proceed by induction on j such that $0 \leq j \leq k$.

Let $j = 0$ in the base case. We get by having only a single stratum St_0 that $I_{\mathbf{P}^0}(P_i[S])$ is Horn and coincides with $I_{\mathbf{P}}(P_i[S])$. Hence, $T_{\mathbf{P}^0}(\mathbf{M}) = T_{\mathbf{P}}(\mathbf{M})$ for any $\mathbf{M}$, in particular $T_{\mathbf{P}^0}(\mathbf{M}^0) = T_{\mathbf{P}}(\mathbf{M}^0)$. Thus, $T_{\mathbf{P}^0}\uparrow_\omega(\mathbf{M}^0) = T_{\mathbf{P}}\uparrow_\omega(\mathbf{M}^0) = \mathrm{lfp}(\mathbf{P})$.

Let $j > 0$ in the inductive step and assume that $\mathbf{M}^{j-1}$ is an answer set of $\mathbf{P}^{j-1}$ with stratification $St_0, \ldots, St_{j-1}$. We first show that $\mathbf{M}^j$ is a model of $\mathbf{P}^j$ with stratification $St_0, \ldots, St_j$. Towards a contradiction, assume that $\mathbf{M}^j \not\models \mathbf{P}^j$, i.e., $\mathbf{M}^j \not\models I(\mathbf{P}^j)$. There is a rule $r \in I_{\mathbf{P}^j}(P_i[S])$ such that $\mathbf{M}^j, P_i[S] \not\models r$, hence $\mathbf{M}^j, P_i[S] \models B(r)$ and $\mathbf{M}^j, P_i[S] \not\models H(r)$. Since $\mathbf{M}^{j-1} \models \mathbf{P}^{j-1}$, this r must not appear in m_i^{j-1}, thus $H(r) \notin L_{j-1}$. By definition of m_i^j, we have $H(r) \in L_j$, hence $H(r) \in L_j \setminus L_{j-1}$ and thus $H(r) \in St_j$. Therefore we must have $H(r) \in T_{\mathbf{P}^j}^{L_j}(\mathbf{M}^{j-1})$, which is a contradiction to our assumption that $\mathbf{M}^j \not\models \mathbf{P}^j$. Thus, $\mathbf{M}^j$ is a model of $\mathbf{P}^j$.

Now we show that $\mathbf{M}^j$ is a minimal model of $f\,\mathbf{P}^j(P_i[S])^{\mathbf{M}^j}$. Towards a contradiction, assume there exists an $\mathbf{M}' < \mathbf{M}^j$ that is a model of $f\,\mathbf{P}^j(P_i[S])^{\mathbf{M}^j}$. Since $\mathbf{M}^{j-1}$ is an answer set of $\mathbf{P}^{j-1}$ with stratification $St_0, \ldots, St_{j-1}$, there must exist an atom $a \in M_i^j/S$ such that $a \notin M_i^{j-1}/S$ for a particular $P_i[S]$, otherwise a would be missing from M^{j-1}/S, as $\mathbf{M}^{j-1}$ is an answer set for $\mathbf{P}^{j-1}$. Hence, a is from $L_j \setminus L_{j-1}$, thus $a \in St_j$. From $\mathbf{M}^j = T_{\mathbf{P}}^{L_j}\uparrow_\omega(\mathbf{M}^{j-1})$ we can conclude that there exists an $r \in I_{gr(\mathbf{P})}(P_i[S])$ with $H(r) = a$, and since a is from St_j, we must have that $\mathbf{M}^{j-1}, P_i[S] \models B(r)$. Now, $\mathbf{M}^j, P_i[S] \models H(r)$, but $\mathbf{M}', P_i[S] \not\models H(r)$, thus we arrive at a contradiction for $\mathbf{M}'$ being a model of $f\,\mathbf{P}^j(P_i[S])^{\mathbf{M}^j}$. □

Example 4.2 (cont'd) Given $\mathbf{P}$ from the previous example, we have $L_0 = St_0$ and $L_1 = St_0 \cup St_1$. The answer set $\mathbf{M} = \mathbf{M}^1$ can be obtained from $\mathbf{M}^1 = T_{\mathbf{P}}^{L_1}\uparrow_\omega(\mathbf{M}^0)$.

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

Proposition 4.11 (Stratified call graph)
Let $\mathbf{P}$ be a stratified normal MLP and $St_0, \ldots, St_k$ be an arbitrary stratification of $\mathrm{HB}_{\mathbf{P}}$. Then, for every answer set $\mathbf{M}$ of $\mathbf{P}$, it holds that

1. $V(\mathrm{CG}_{\mathbf{P}}(\mathbf{M})) = V(\mathrm{CG}_{\mathbf{P}}(\mathbf{M}^k))$, and
2. $M_i/S = M_i^k/S$, for all $P_i[S] \in \mathrm{VC}(\mathbf{P})$ and any stratification $St_0, \ldots, St_k$ of $\mathrm{HB}_{\mathbf{P}}$.

Proof Let $St'_0, \ldots, St'_{\ell_1}$ and $St''_0, \ldots, St''_{\ell_2}$ be two stratifications for $\mathrm{HB}_{\mathbf{P}}$. Observe that for $i < j \leq \ell_1 \leq \ell_2$, no stratum St'_j depends on a stratum St'_i and no stratum St''_j depends on a stratum St''_i, respectively, which is guaranteed by the definition of $G_{\mathbf{P}}$.

Following the proof for (Apt et al., 1988, Theorem 11), we can transform both $St'_0, \ldots, St'_{\ell_1}$ and $St''_0, \ldots, St''_{\ell_2}$ into a single stratification $St_0, \ldots, St_k$ by grouping clusters of rules together; clusters are nonempty subsets of rules in $\mathbf{P}$ that are the unions of a maximal collection of definitions that define relations depending on each other. For this purpose, one can define a partial order on clusters that order them based on their dependencies. Clusters that are unrelated with respect to this partial order can then be rearranged while preserving their answer set.

Hence, we get that

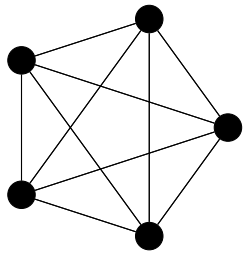
$$\bigcup_{0 \leq i < \ell_1} T_{\mathbf{P}}^{L'_{i+1}} \uparrow_{\omega}(\mathbf{M}^i) = \bigcup_{0 \leq i < k} T_{\mathbf{P}}^{L_{i+1}} \uparrow_{\omega}(\mathbf{M}^i)$$

and

$$\bigcup_{0 \leq i < \ell_2} T_{\mathbf{P}}^{L''_{i+1}} \uparrow_{\omega}(\mathbf{M}^i) = \bigcup_{0 \leq i < k} T_{\mathbf{P}}^{L_{i+1}} \uparrow_{\omega}(\mathbf{M}^i) \ .$$

Thus, for any stratification the answer set coincide, and both (1) and (2) hold. □

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



5

Computational Complexity of Modular Nonmonotonic Logic Programs

This chapter discusses the computational costs of MLPs in the propositional and the nonground settings. We study the complexity of MLPs with and without module input and provide completeness results for the problem of deciding whether a ground atom is contained in the unique answer set for Horn MLPs, and for deciding answer set existence for normal and disjunctive MLPs. All results are compactly summarized in Tables 5.1–5.3.

The complexity results have been obtained using Turing machine simulations and bounded domino tiling problems; see also Börger et al. (1997) for an in-detail investigation on these techniques. Our findings in §5.2 show that allowing unrestricted module input in MLPs increases the computational complexity by an exponential factor already in the Horn case. Similarly, as we show in §5.3, deciding whether a normal MLP has an answer set is NEXP-complete even in the propositional case, which matches the complexity of nonground normal logic programs. As shown in §5.4, this holds even for acyclic normal MLPs, i.e., normal MLPs whose call graph is acyclic. §5.5 then presents complexity results for general MLPs without restrictions. Propositional disjunctive MLPs match the complexity of nonground disjunctive logic programs (NEXP^{NP}-complete). In the nonground case, the complexity jumps again by an exponential factor: nonground Horn MLPs are complete for 2EXP, while normal (respectively, disjunctive) MLPs are complete for 2NEXP (respectively, 2NEXP^{NP}). If we bound the arities of the input predicates by a constant, then the complexity drops by an exponential factor and matches that of ordinary logic programs: nonground Horn MLPs with bounded predicates are complete for EXP, while nonground normal (respectively, disjunctive) MLPs are then complete for NEXP (respectively, NEXP^{NP}).

MLP **P**	Computing lfp(**P**)	Deciding $\alpha \in \text{lfp}(\mathbf{P})$
propositional, empty inputs	polynomial time	P-complete
propositional	exponential time	EXP-complete
nonground, bounded predicates	exponential time	EXP-complete
nonground	double exponential time	2EXP-complete

Table 5.1: Complexity of Horn MLPs (α is a ground atom)

MLP **P**	Answer set existence
normal, empty inputs	NP-complete
empty inputs	Σ_2^p-complete
normal	NEXP-complete
acyclic	NEXP-complete
unrestricted	NEXP^{NP}-complete

Table 5.2: Complexity of answer set existence for propositional MLPs

MLP **P**	Answer set existence
normal, bounded predicates	NEXP-complete
normal	2NEXP-complete
bounded predicates	NEXP^{NP}-complete
unrestricted	2NEXP^{NP}-complete

Table 5.3: Complexity of answer set existence for nonground MLPs

Figure 5.1 illustrates the complexity landscape for the studied syntactic classes of MLPs as an directed acyclic graph, where each node $S\colon C$ consists of the name S of the syntactic class on the left and the complexity class C on the right of the colon. There is an edge from $S_1\colon C_1$ to $S_2\colon C_2$ whenever S_1 contains S_2. Here, $\mathbb{N}_r$ and $\mathbb{N}_r^b$, for $r \in \{u, n, h\}$, denote the classes of nonground MLPs with arbitrary module input (superscript is void) and bounded predicate input (superscript b), respectively, such that the rule sets consists of unrestricted rules (u), normal rules (n), and Horn rules (h). For the syntactic classes of propositional MLPs we let $\mathbb{P}_r$ and $\mathbb{P}_r^{\varnothing}$, for $r \in \{u, n, a, h\}$, denote the classes of propositional MLPs with arbitrary module input (where the superscript is void) and empty module input (with $\varnothing$ as superscript), respectively, such that the rule sets consists of unrestricted rules (u), normal rules (n), acyclic rules (a), and Horn rules (h).

We start with a recapitulation of the most important definitions from complexity theory in the next section.

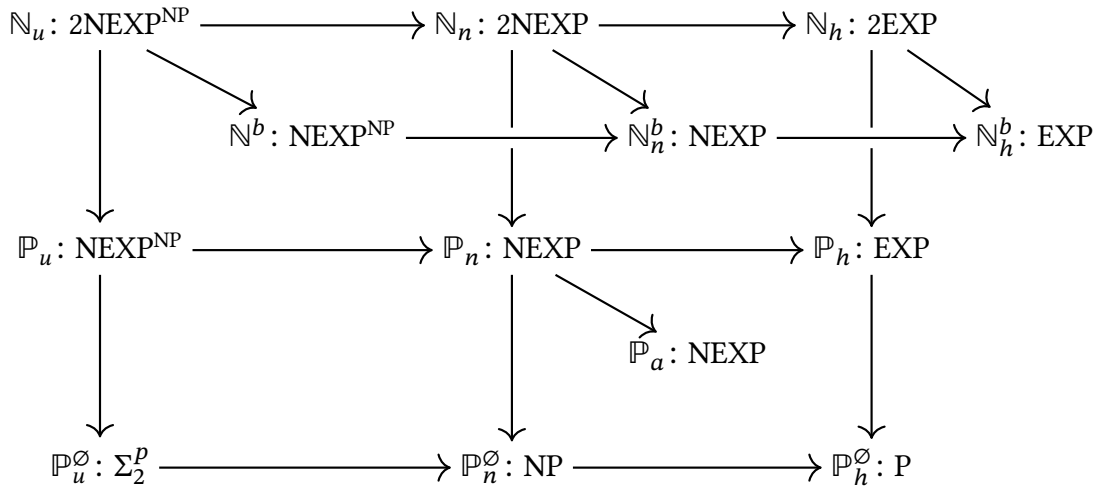


Figure 5.1: Complexity landscape of Modular Nonmonotonic Logic Programs

5.1 Alternating Turing Machines and Complexity Classes

This section builds upon results and definitions by Chandra et al. (1981) and Dantsin et al. (2001). Further material and details on complexity theory is provided by Papadimitriou (1994), by Börger et al. (1997), and by Garey and D. S. Johnson (1979).

Some of our results in this chapter rely on alternating Turing machine simulations. This type of Turing machine can be considered as a generalization of the classical deterministic and nondeterministic Turing machine, hence we base our definitions on alternating Turing machines, and then provide the restrictions necessary for deterministic and nondeterministic machines.

5.1.1 Alternating Turing Machines

An *alternating Turing machine* (ATM) T is a quintuple $(S, \Sigma, \delta, s_0, g)$, where S is a finite set of *states*, Σ is a finite alphabet of *input and tape symbols*, δ is the *transition relation*, $s_0 \in S$ is the *initial state*, and $g\colon S \to \{\exists, \forall\}$ is a mapping that assigns each state in S a *state identifier*. The transition relation δ is defined as

$$\delta \subseteq (S \times \Sigma) \times ((S \cup \{yes, no\}) \times \Sigma \times \{-1, 0, +1\}) \quad ,$$

where the states *yes*, *no* do not occur in S and $-1, 0, +1$ denote *motion directions*. We denote $\textvisiblespace \in \Sigma$ to be the blank tape symbol.

We say that T is a *nondeterministic* Turing machine (NTM) iff for each $s \in S$ it holds that $g(s) = \exists$, and T is called a *deterministic* Turing machine (DTM), iff it is a NTM and δ is a functional relation over $(S \times \Sigma)$. Since g is not important for DTMs

and NTMs, we simply identify them as quadruple (S, Σ, δ, s_0). Thus, an NTM may be considered to be an ATM without universal states, and a DTM is an NTM whose transition relation δ is functional.

A *configuration* γ of ATM T is a triple $(s, w, u) \in S \cup \{yes, no\} \times \Sigma^* \times \Sigma^*$, where s represents the current state of T, and w and u represent the contents of the tape left and right from the read-write head on the tape, respectively. Given an input string I, we call configuration $\gamma_0 = (s_0, \sqcup, I)$ the *initial configuration* of T.

Let $\gamma = (s, w, u)$ be a configuration for the ATM T. If $g(s) = \forall$, then γ is said to be a *universal* configuration, and for $g(s) = \exists$, we say that γ is an *existential* configuration. We call γ *accepting* if $s = yes$, and *rejecting* if $s = no$. A *halting* configuration is either an accepting or a rejecting configuration.

For an ATM $T = (S, \Sigma, \delta, s_0, g)$ and configurations γ, γ' of T, we call γ' a *successor* of γ if γ' can be reached from γ in one step according to the transition relation δ. A *computation* of T is a sequence of configurations $\gamma_0, \gamma_1, \gamma_2, \ldots$ such that for each pair γ_i and γ_{i+1}, $i \geq 0$, γ_{i+1} is a successor of γ_i.

We can now define accepting and rejecting configurations given universal and existential configurations. Let $\gamma = (s, w, u)$ be a configuration such that $g(s) \in \{\forall, \exists\}$. We call γ an *accepting* configuration if γ is universal and all successors γ' of γ are accepting, or if γ is existential and there exists a successor γ' of γ that is accepting. We say that the universal (respectively, existential) configuration γ is *rejecting* if some successor γ' of γ is rejecting (respectively, all successors γ' of γ are rejecting).

The ATM T *accepts* input I if the initial configuration γ_0 is accepting, and *rejects* input I if γ_0 is rejecting. An ATM *halts* on an input I if its initial configuration γ_0 is accepting or rejecting. We say that an ATM T *decides a language* L if T accepts all strings $I \in L$ and rejects all strings $I \notin L$.

5.1.2 Complexity Classes

Based on alternating Turing machines, we define now the complexity classes that are used in our results. For a given function $g\colon \mathbb{N} \to \mathbb{N}$, we denote by $O(g(n))$ the set of functions $\{f(n) \mid \exists c, k \in \mathbb{N} \setminus \{0\}$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq k\}$. A function $f\colon \mathbb{N} \to \mathbb{N}$ is called a *proper complexity function* if for all $n \in \mathbb{N}$, $f(n+1) \geq f(n)$, and there exists a DTM T_f that for a given input string I such that $n = |I|$ writes exactly $f(n)$ blank symbols $\sqcup$ on the tape and whose computation has length $O(n + f(n))$ and touches $O(f(n))$ tape cells and halts. Let f be a proper complexity function on positive integers. Let ATIME $(f(n))$ be the class of all languages that are decided by some ATM, whose computations halt on input I with length $n = |I|$ in at most $f(n)$ steps. We define ASPACE $(f(n))$ to be the class of all languages that are decided by some ATM, whose computations halt on input I with length $n = |I|$ with at most $f(n)$ of cells visited. The definitions for deterministic (DTIME $(f(n))$, DSPACE $(f(n))$) and

nondeterministic (NTIME $(f(n))$, NSPACE $(f(n))$) resource bounds classes are analogous, but instead of ATMs, they use DTMs and NTMs, respectively.

We can now define the main complexity classes for alternating computations.

$$\begin{aligned}
\text{ALOGSPACE} &= \text{ASPACE}\,(\log n) \\
\text{AP} &= \bigcup_{k>0} \text{ATIME}\left(n^k\right) \\
\text{APSPACE} &= \bigcup_{k>0} \text{ASPACE}\left(n^k\right) \\
\text{AEXP} &= \bigcup_{k>0} \text{ATIME}\left(2^{n^k}\right) \\
\text{AEXPSPACE} &= \bigcup_{k>0} \text{ASPACE}\left(2^{n^k}\right)
\end{aligned}$$

The deterministic and nondeterministic complexity classes for deterministic Turing machines (LOGSPACE, P, PSPACE, EXP, EXPSPACE) and nondeterministic Turing machines (NLOGSPACE, NP, NPSPACE, NEXP, NEXPSPACE) are defined accordingly using DTIME $(\cdot)$ and DSPACE $(\cdot)$ respectively NTIME $(\cdot)$ and NSPACE $(\cdot)$ instead.

For a language L over alphabet $\Sigma \setminus \{\textvisiblespace\}$ we let $\overline{L}$ denote its *complement language* $(\Sigma \setminus \{\textvisiblespace\})^* \setminus L$. A complexity class C has a *complementary class co-C* defined as the set $\{\overline{L} \mid L \in C\}$.

We define now oracle Turing machines. Let $L \subseteq \Sigma^*$ be a language. The computation of an *oracle machine* T^L *with oracle* L proceeds like an ordinary Turing machine with an additional write-only *query tape* and additional three states q_{query}, q_{yes}, q_{no}. Whenever T^L is not in state q_{query} the computation proceeds as a standard Turing machine, and T^L may also write to the query tape. Let I_q be the string written on the query tape. If T^L is in state q_{query}, then T^L switches its state either to q_{yes} in case $I_q \in L$, or to q_{no} in case $I_q \notin L$, and then erases the query tape content. Note that we did not fix the machine type here: whenever T is a DTM, we call T^L an deterministic oracle Turing machine, and for T being a NTM, we say that T^L is a nondeterministic oracle Turing machine. The time and space resource bounds are defined analogously as in standard Turing machines, except that additionally the number of steps and required space on the query tape is taken into account. Let C be any deterministic or nondeterministic time complexity class, we define C^L to be the class of all languages accepted by a DTM or NTM (whenever C is deterministic or nondeterministic, respectively) with equal time bound as in C such that the machine has an oracle for language L. For a set of languages A, we define $C^A = \bigcup_{L \in A} C^L$.

The *polynomial hierarchy* (Stockmeyer, 1976; Wrathall, 1976) consists of complexity

classes Δ_i^p, Σ_i^p, and Π_i^p recursively defined as follows:

$$\begin{aligned}\Delta_0^p = \Sigma_0^p = \Pi_0^p &= \mathrm{P}\\ \Delta_{i+1}^p &= \mathrm{P}^{\Sigma_i^p}\\ \Sigma_{i+1}^p &= \mathrm{NP}^{\Sigma_i^p}\\ \Pi_{i+1}^p &= \text{co-}\Sigma_{i+1}^p\end{aligned}$$

for all $i \geq 0$.

Further, we define complexity classes mEXP and mNEXP, for $m \geq 1$, as follows. For $n \geq 0$, let ${}^n a$ denote the iterated exponentiation

$${}^n a = \begin{cases} 1 & n = 0, \\ a^{({}^{(n-1)}a)} & n > 0. \end{cases}$$

Then

$$m\mathrm{EXP} = \mathrm{DTIME}\left(({}^m 2)^{n^k}\right)$$

and

$$m\mathrm{NEXP} = \mathrm{NTIME}\left(({}^m 2)^{n^k}\right) \quad .$$

As an example, set $m = 2$: then $2\mathrm{EXP} = \mathrm{DTIME}\left(2^{2^{n^k}}\right)$ and $2\mathrm{NEXP} = \mathrm{NTIME}\left(2^{2^{n^k}}\right)$, which will be used in our complexity results. Note that for $m = 1$, $1\mathrm{EXP} = \mathrm{EXP}$ and $1\mathrm{NEXP} = \mathrm{NEXP}$.

The *weak EXP hierarchy* (Hemachandra, 1989) consists of complexity classes Σ_i^e and Π_i^e recursively defined for all $i \geq 0$ as follows:

$$\begin{aligned}\Sigma_0^e &= \mathrm{EXP}\\ \Sigma_{i+1}^e &= \mathrm{NEXP}^{\Sigma_i^p}\\ \Pi_{i+1}^e &= \text{co-}\Sigma_{i+1}^e\end{aligned}$$

For our purposes, we simply use EXP, NEXP, and $\mathrm{NEXP}^{\mathrm{NP}}$. We can also go one level higher and define a double exponential hierarchy, but instead of EXP and NEXP as base class, we use 2EXP and 2NEXP instead. We will later show that in general, MLPs match the complexity class $2\mathrm{NEXP}^{\mathrm{NP}}$.

Next, we define reductions and completeness for complexity classes. We say that *language L_1 can be reduced to language L_2* if there is a function $f\colon \Sigma^* \to \Sigma^*$ (called *reduction* from L_1 to L_2) computable by a DTM in polynomial time such that for all

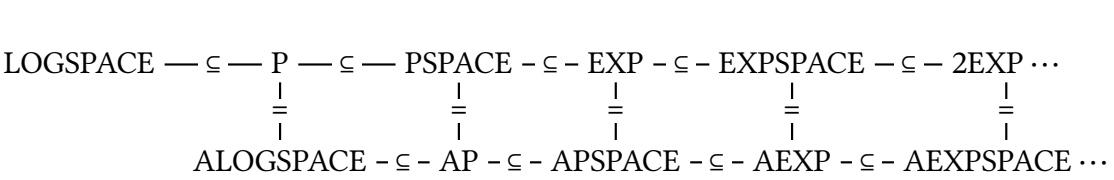


Figure 5.2: Relationships between deterministic and alternating hierarchies

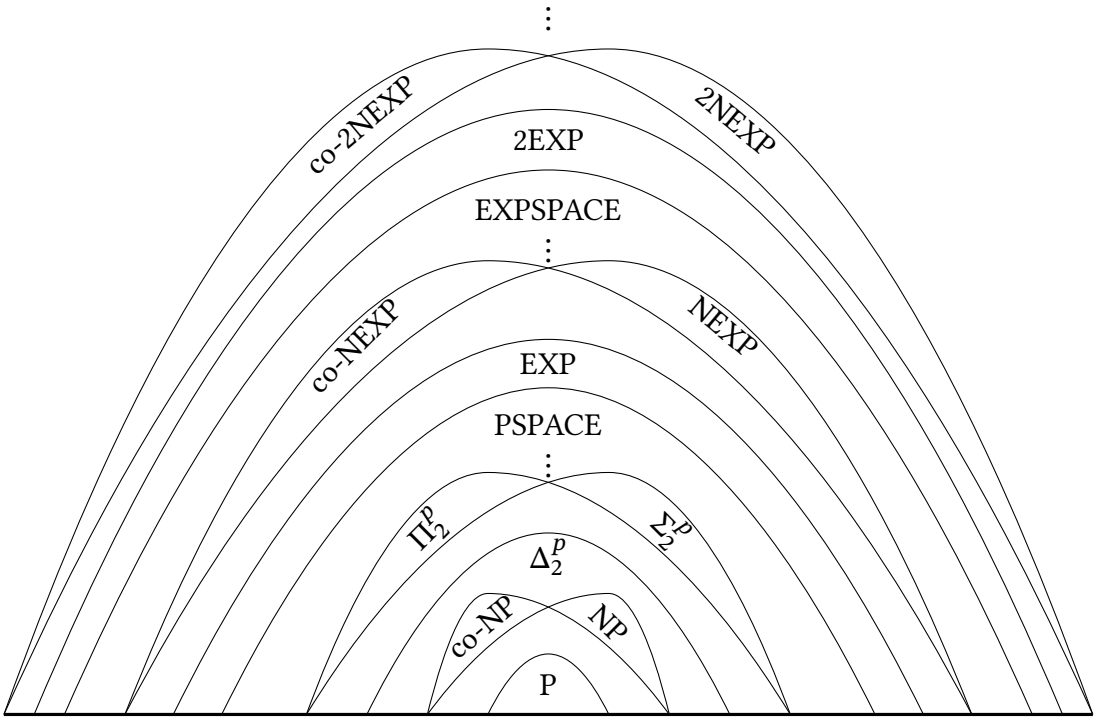


Figure 5.3: Relationships between complexity classes

strings $I \in \Sigma^*$, $I \in L_1$ iff $f(I) \in L_2$. Let C be a complexity class and L be a language from C. We define L to be *C-hard* if for all $L' \in C$, L' can be reduced to L. A language L is called *C-complete* if L is C-hard and $L \in C$.

As shown by Chandra et al. (1981), the deterministic hierarchy shifts exactly by one level in alternating complexity classes, and their relationships to deterministic complexity classes can be summarized by Figure 5.2 (Grädel, 2007). Figure 5.3 reviews the relationships of complexity classes used in our results (Papadimitriou, 1994).

5.2 Propositional MLPs without Input

To begin with, let us restrict our attention to Horn MLPs. Considering the propositional case, if the modules $m_i = (P_i[\mathbf{q}_i], R_i)$ in $\mathbf{P}$ have no input (i.e., $\mathbf{q}_i$ is void),

then $I(\mathbf{P})$ has polynomial size and lfp($\mathbf{P}$) is computable in polynomial time. For arbitrary propositional $\mathbf{P}$ with no inputs, we can guess and verify an answer set $\mathbf{M}$ of $\mathbf{P}$ in polynomial time with an NP oracle. As shown in Proposition 3.2, MLPs subsume ordinary logic programs, we thus obtain by known results the same complexity; compare Dantsin et al. (2001). Note that the results here stay the same for stratified MLPs. Indeed, the lower bounds in this section could have been obtained by reductions from ordinary logic programs to MLPs with a single module without input. Instead we use QBF encodings and Turing machine simulations, which we build upon in §5.3 and §5.5. See also the discussion on simulating Turing machines by logical deduction in Dantsin et al. (2001, Section 4.1).

With slight abuse of notation, for a ground atom α and an interpretation $\mathbf{M}$ of $\mathbf{P}$, we write $\alpha \in \mathbf{M}$ if $\alpha \in M_i/S$ for a given $P_i[S] \in \mathrm{VC}(\mathbf{P})$ in the following.

Theorem 5.1 (Computational complexity of propositional MLPs without input) Given a propositional MLP $\mathbf{P} = ((P_1[], R_1), \ldots, (P_n[], R_n))$,

1. if $\mathbf{P}$ is Horn, the unique answer set $\mathbf{M} = \mathrm{lfp}(\mathbf{P})$ of $\mathbf{P}$ is computable in polynomial time and to decide whether $\alpha \in \mathbf{M}$ for a ground atom α is P-complete;
2. if $\mathbf{P}$ is normal, to decide whether $\mathbf{P}$ has an answer set is NP-complete; and
3. to decide whether $\mathbf{P}$ has an answer set is Σ_2^p-complete.

Proof of Theorem 5.1, item 1 We first show membership in P. Since every interpretation of $\mathbf{P}$ is of form $\mathbf{M} = (M_1/\emptyset, \ldots, M_n/\emptyset)$, we have that the least fixpoint of the $T_\mathbf{P}$ operator can be computed in polynomial time: if we exhaustively apply $T_\mathbf{P}(M_i/\emptyset)$, $1 \leq i \leq n$, we reach the fixpoint after at most $(m+1) \cdot n$ application steps, where m is the number of rules in $\mathbf{P}$. Each application of $T_\mathbf{P}$ can be done in polynomial time. This shows that the unique answer set $\mathbf{M} = \mathrm{lfp}(\mathbf{P})$ can be computed in time polynomial in the size of $\mathbf{P}$.

Next we show P-hardness. A language $L \in \mathrm{P}$ can be decided by a deterministic Turing machine T in polynomially many steps. We can translate each instance I of L with $m = |I|$ to a Horn MLP with empty input modules that encodes T. Let $\mathbf{P} = ((P_1[], R_1))$. The rules in R_1 are the initialization facts, transition rules, inertia rules, and accept rules from Dantsin et al. (2001, Section 4.1). We have now that $T_\mathbf{P}{\uparrow}_0(\mathbf{M}_\emptyset) = \mathbf{M}_\emptyset$ for $\mathbf{M}_\emptyset = (M_1/\emptyset) = (\emptyset)$ and that $M_1/\emptyset$ in $T_\mathbf{P}{\uparrow}_1(\mathbf{M}_\emptyset)$ consists of all initialization facts from R. The least fixpoint lfp($\mathbf{P}$) is reached at $T_\mathbf{P}{\uparrow}_{m+2}(\mathbf{M}_\emptyset)$ and contains *accept* in $M_1/\emptyset$ iff T accepts I in at most m steps. This is analogous to Dantsin et al. (2001, Lemma 4.1). The reduction can be done in logarithmic space in the length of I (see proof of Dantsin et al. (2001, Theorem 4.2)), thus our P-hardness result follows. We obtain that $\alpha \in \mathbf{M}$ is P-complete. □

Proof of Theorem 5.1, item 2 Membership in NP follows from the following observation. The set of value calls $\mathrm{VC}(\mathbf{P}) = \{P_1[\varnothing], \dots, P_n[\varnothing]\}$ contains only empty input sets, thus for every interpretation $\mathbf{M}$ of $\mathbf{P}$, we have that $M_k/\left((M_i/\varnothing)|_{()}^{()}\right) = M_k/\varnothing$ for every edge $P_k[\varnothing] \to P_i[\varnothing]$ in $\mathrm{CG}_\mathbf{P}$. The reduct $f\,\mathbf{P}^\mathbf{M}$ thus can then be computed in time polynomial in the size of $\mathbf{P}$ by just applying the FLP-reduction in all $f\,\mathbf{P}(P_i[\varnothing])^\mathbf{M}$. An algorithm that checks whether $\mathbf{P}$ is consistent works as follows. We first guess an interpretation $\mathbf{M} = (M_1/\varnothing, \dots, M_n/\varnothing)$, which can be done in time polynomial in the size of $\mathbf{P}$. Then we compute $f\,\mathbf{P}^\mathbf{M}$ and check whether $\{\mathbf{M}\} = \mathrm{MM}\left(f\,\mathbf{P}^\mathbf{M}\right)$ in time polynomial in the size of $\mathbf{P}$ (negated atoms that survive the reduction $f\,\mathbf{P}^\mathbf{M}$ cannot be in $\mathbf{M}$, thus this amounts to checking a positive program). Hence, checking consistency is in NP.

Next we show NP-hardness, which is shown similar to item 1. A language $L \in \mathrm{NP}$ can be decided by a nondeterministic Turing machine M in polynomially many steps. We can translate each instance I of L with $m = |I|$ to a normal MLP with empty input modules that encodes M. Let $\mathbf{P} = ((P_1[], R_1))$. Similarly to the hardness proof for ordinary ASP, the rules in R_1 are the initialization facts, transition rules, inertia rules, and accept rules used in the proof of Dantsin et al. (2001, Theorem 5.7). Now every answer set $\mathbf{M} = (M_1/\varnothing)$ of $\mathbf{P}$ must contain *accept*, since the transition rules encode the nondeterministic choice of the NTMs M state transition. Analog to Dantsin et al. (2001, Theorem 5.7), $\mathbf{M}$ corresponds to accepting computations of M in at most m steps. The reduction can be done in time polynomial in the size of the instance I, thus our completeness result follows. □

Proof of Theorem 5.1, item 3 We start with showing membership in Σ_2^p. As shown before in item 2, $\mathrm{VC}(\mathbf{P}) = \{P_1[\varnothing], \dots, P_n[\varnothing]\}$ and the reduct $f\,\mathbf{P}^\mathbf{M}$ can then be computed in time polynomial in the size of $\mathbf{P}$ by just applying the FLP-reduction in all $f\,\mathbf{P}(P_i[\varnothing])^\mathbf{M}$. An algorithm that checks whether $\mathbf{P}$ is consistent works as follows. We first guess an interpretation $\mathbf{M} = (M_1/\varnothing, \dots, M_n/\varnothing)$ and compute $f\,\mathbf{P}^\mathbf{M}$ in time polynomial in the size of $\mathbf{P}$. Then, we check whether $\mathbf{M}$ is a minimal model with respect to $\leq$, i.e., there is no interpretation $\mathbf{N} < \mathbf{M}$ that satisfies $f\,\mathbf{P}^\mathbf{M}$. This check can be done using a co-NP oracle, thus by $\mathrm{NP}^{\text{co-NP}} = \mathrm{NP}^{\mathrm{NP}}$ the Σ_2^p upper bound follows.

The hardness part can be shown similar to item 1. Let $\phi = \exists X \forall Y \psi$ be a QBF, where ψ is in 3-DNF. The problem of deciding whether ϕ is valid is a Σ_2^p-complete problem. We can reduce this problem to answer set existence of propositional MLPs without input analog to the reduction given in the proof of Eiter and Gottlob (1995, Theorem 3.1). Instead of a DLP P we use an MLP $\mathbf{P} = ((P[], R))$ such that R contains the rules of P. The reduction generates $\mathbf{P}$ in time polynomial in the size of ϕ and works as expected: ϕ is valid iff $\mathbf{P}$ has an answer set. Thus we have a Σ_2^p lower bound for item 3, and therefore deciding whether $\mathbf{P}$ has an answer set is Σ_2^p-complete. □

5.3 Propositional MLPs with Input

The results in §5.2 generalize to the case where the module inputs in $\mathbf{P}$ have bounded length, i.e., $|\mathbf{q}_i| \leq k$ for some constant k, as $I(\mathbf{P})$ and $\mathbf{M}$ have polynomial size. For unrestricted inputs, however, $I(\mathbf{P})$ and $\mathbf{M}$ are exponential and we get a respective blowup.

The hardness parts are shown by encodings of Turing machines, which adapt constructions by Dantsin et al. (2001). Roughly speaking, we use modules with three groups of input predicates of the form $P[\mathbf{c}, \mathbf{c}', \mathbf{t}]$, where $\mathbf{c}$ and $\mathbf{c}'$ amount to tape cell indexes, and $\mathbf{t}$ to a time stamp during a computation. With $|\mathbf{c}| = |\mathbf{c}'| = |\mathbf{t}| = \ell$, we can model 2^ℓ cells and 2^ℓ time stamps. Further atoms store the cell contents, state of the machine, and the position of the read-write head. The transition function is encoded by rules with access to the contents of neighboring cells, which is realized by respective (recursive) module calls; neighboring cells and time stamps are computed using local rules. Inertia rules are used to keep the tape content for cells that have not been used in a computation step. This process requires to scan for the current position of the read-write head relative to an unused position on the tape. Since we can have 2^ℓ cells, adding all possible cell positions in relation to the current head position would blow up the encoding and thus the reduction would not be polynomial. Using cell positions $\mathbf{c}$ and $\mathbf{c}'$ as module input and their relative position $\mathbf{c} < \mathbf{c}'$ or $\mathbf{c}' < \mathbf{c}$ on a grid, we can encode inertia rules in the required reduction time bounds.

Note that the Turing machine encodings given in this section are incremental, i.e., we build upon the Turing machine simulation of the syntactic class with lower complexity and reuse them in the MLP fragment with higher computational costs.

We can therefore state our next result.

Theorem 5.2 (Computational complexity of propositional MLPs with input) Given a propositional MLP $\mathbf{P} = (P_1[\mathbf{q}_1], \ldots, P_n[\mathbf{q}_n])$,

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

In the following, we will now prove this result for every item.

5.3.1 Proof of Theorem 5.2, item 1

We first show membership in EXP. Since $|\mathrm{HB}_\mathbf{P}|$ is linear in the size of $\mathbf{P}$, we have that every interpretation $\mathbf{M}$ of $\mathbf{P}$ consists of at most $n \cdot 2^{|\mathrm{HB}_\mathbf{P}|}$ components, thus the least fixpoint of the $T_\mathbf{P}$ operator can be computed in exponential time: if we exhaustively apply $T_\mathbf{P}(M_i/S)$, $1 \leq i \leq n$ and $S \subseteq \mathrm{HB}_\mathbf{P}|_{\mathbf{q}_i}$, we reach the fixpoint after at most

$(m+1)\cdot n\cdot 2^{|\mathrm{HB_P}|}$ application steps, where m is the number of rules in $\mathbf{P}$. Each application of $T_{\mathbf{P}}$ can be done in polynomial time. This shows that the unique answer set $\mathbf{M} = \mathrm{lfp}(\mathbf{P})$ can be computed in time exponential in the size of $\mathbf{P}$.

We show now EXP-hardness. Given a deterministic Turing machine T which halts in less than $N = 2^{m^k}$ steps for an input I such that $m = |I|$, we can simulate T by an MLP consisting of three modules. Without loss of generality the encoding considers Turing machines without input. Since we require that the running time is exponential in the length of the input, we can adapt the transition function δ of an arbitrary Turing machine T with respect to I, and encode the input in a transition function δ' by writing down the input as a first step before the actual computation is done.

Turing machines that run in exponential time can potentially touch exponential many tape cells at exponential many different time points. For that reason, we cannot simply encode time points and cell positions in the atoms, but have to encode time instants and positions as binary numbers and use a counter to address the correct configuration of the machine. We can do this by using the input mechanism of MLPs and encode the bits of a nonnegative number $n \in \{0, \dots, 2^\ell - 1\}$ as a sequence of atoms $b_1, \dots, b_\ell, \overline{b_1}, \dots, \overline{b_\ell}$ (short $\mathbf{b}$). If b_i is true in an interpretation then the i^{th} bit of n is 1 and if $\overline{b_j}$ is true then bit j of n is set to 0; intuitively, b_i and $\overline{b_i}$ must have complementary truth values in a model. This way we can represent 2^ℓ positions and time points.

In the following, let $T = (S, \Sigma, \delta, s_0)$ be a deterministic Turing machine and let I be an input string encoded in δ. We use propositional atoms listed below in our modules to encode the states and the tape content of T during the computation. Note that we do not need to add indexes for cell positions and time points in the atoms below, as the formal input parameters to the modules are used to encode cells and time points. This way, the value calls can be identified as cell-time reference to Turing machine configurations. Let $m = |I|$ be the length of I, $\ell = m^k$ for some constant k, and $N = 2^\ell$ be the time bound for T.

- *init*, $\overline{\mathit{init}}$, and *start* mean that in a run of T we are at time point 0, in between time points $1, \dots, N-1$, and at the beginning of the tape, respectively;
- at time point t, the atoms t_i^- and $\overline{t_i^-}$ represent the i^{th} bit of time point $t-1$;
- at cell c, the atoms c_i^-, $\overline{c_i^-}$ and c_i^+, $\overline{c_i^+}$ represent the i^{th} bit of $c-1$ and $c+1$, respectively;
- at cell position c', which is used to have a relative position for tape cell c, the atoms $c_i'^-$, $\overline{c_i'^-}$ and $c_i'^+$, $\overline{c_i'^+}$ represent the i^{th} bit of $c'-1$ and $c'+1$, respectively;
- atom s representing that at time point t, the machine is in state $s \in S$;
- atom σ indicates that symbol $\sigma \in \Sigma$ is written on cell c at time point t on the tape of T;

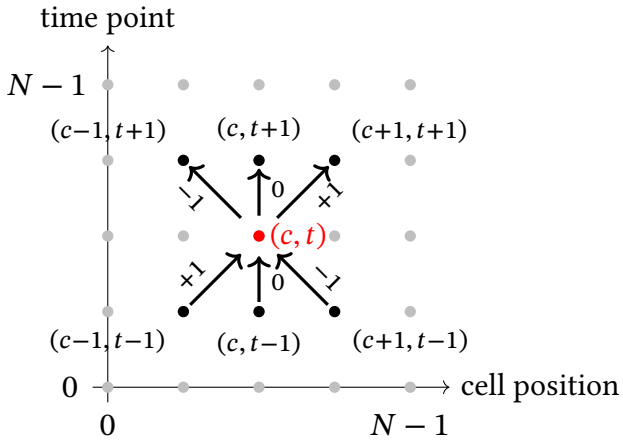


Figure 5.4: Turing machine configurations on the cell-time grid

- atoms $\sigma'^{-1}, \sigma'^{+1}$ encode that symbol $\sigma' \in \Sigma$ will be written on cell $c-1$ or $c+1$ at time point t whenever the read/write head is at c on the tape of T for transitions with motion directions from $\{-1,+1\}$;
- *head* has the intuitive meaning that the read/write-head is at cell position c at time point t;
- m^{-1}, m^{0}, m^{+1} means that the read/write-head at cell position c at time point t moved to the left, stay at the current, or moved to the right position from time point $t-1$, respectively;
- atoms $\leq, =^i$, and $\neq$ that represent for two cell positions c and c' whether $c \leq c'$, $c = c'$, or $c \neq c'$ hold, respectively; and
- the atom *accept*, which means that T accepts input I.

Note that the atoms $\sigma'^{-1}, \sigma'^{+1}$ capture that when we move from cell c at time point $t-1$ to $c+d$ at t for motion directions $d \in \{-1,+1\}$, the symbol σ on c at $t-1$ under the read/write head will remain and, depending on direction d, only the successor/predecessor tape cell will get a new symbol at time point t.

Next we define the MLP $\mathbf{D}(T,N) = (m_1^{\mathbf{D}}, m_2^{\mathbf{D}}, m_3, m_4)$ that, given DTM T and time bound N, simulates computations of T. Intuitively, the main module $m_1^{\mathbf{D}}$ computes in *accept* the acceptance of I, while library module $m_2^{\mathbf{D}}$ encodes transition rules for δ (and therefore I). The library modules m_3 and m_4 help $m_2^{\mathbf{D}}$ with the computation of successors and predecessors of cell positions and time points, and for computing a linear order $\leq$ for cell positions.

Figure 5.4 visualizes possible configuration changes of a Turing machine based on cell positions and time points using appropriate successor and predecessor con-

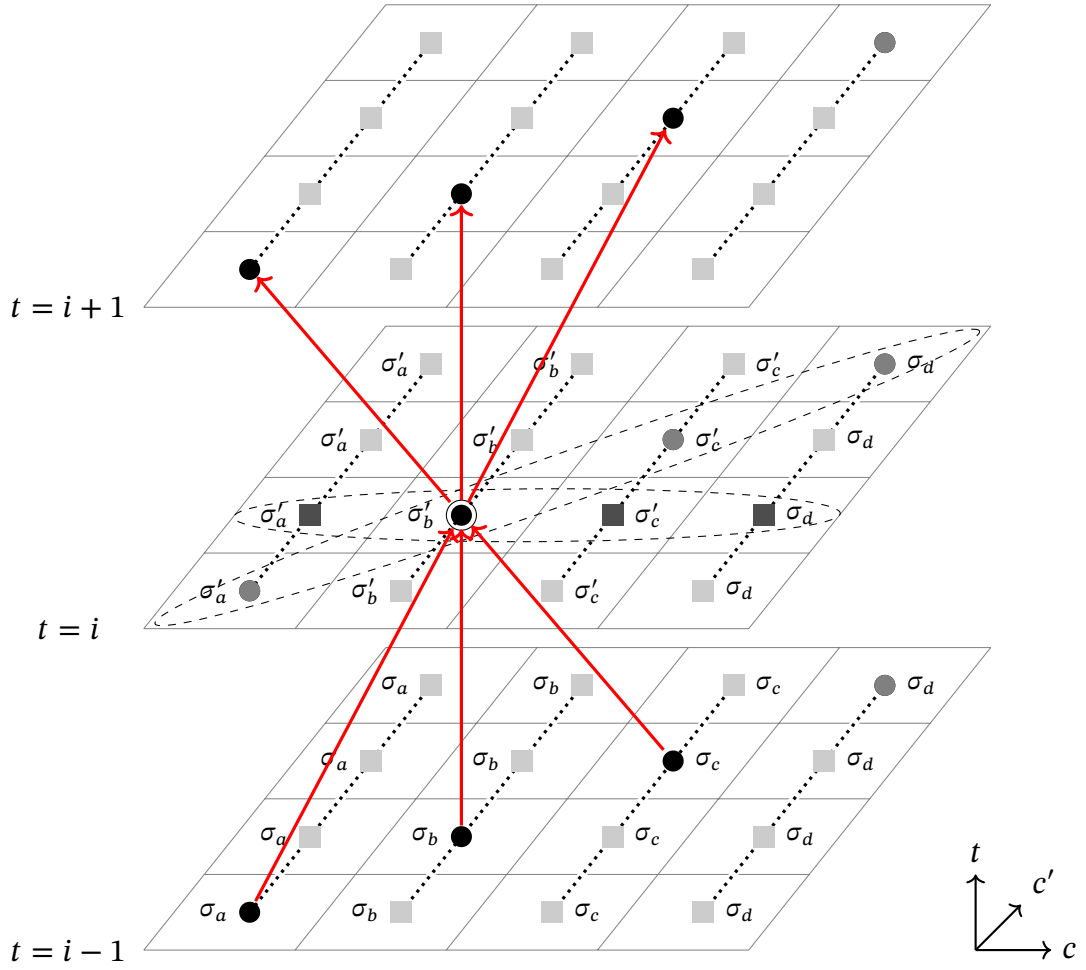


Figure 5.5: Turing machine motions in the cell-time-cube

figurations of cell c at time t. It highlights cell-time point (c, t) and shows possible predecessor and successor positions with thick arrows depending on the motion directions $-1, 0, +1$ of the read/write head: the computation may have come from cell position $c-1$, c, or from $c+1$ at the direct predecessor time point $t-1$ in the past, and looking into the future, we may continue with our computation at the direct successor time point $t+1$ at cell position $c-1$, at c, or at $c+1$.

But using a two-dimensional grid as intuition for the Turing machine encoding is not enough. In order to capture that cell contents remain unchanged on the tape whenever they are not involved in a state transition, we use reference cell positions c' that are used to scan for the head position for all positions c such that $c < c' - \max(d, 0)$ and $c' - \min(d, 0) < c$ depending on the motion directions $d \in \{-1, 0, +1\}$ of the head. Whenever (c', c') does not contain the machine head, we copy to position (c, c') at time

point t the symbol σ from the direct predecessor time point $t-1$. Then, we copy the tape contents of $(c, c+1)$ and $(c, c-1)$ at current time point t, thus distributing cell contents over the whole grid and thus filling the computation tape along the diagonal.

Figure 5.5 illustrates this process. At time points $t = i-1, i, i+1$, we span a grid of cell positions (c, c'). The cells along the dotted lines store the same tape content, while the actual computation takes place in the diagonal of each grid, indicated by filled circles. At time point $t = i$ we have a cell highlighted as fish-eye bullet that has three incoming and outgoing red edges. Whenever the machine head is located on this position, we may have come from three configurations shown as black circles located along the diagonal at time point $t = i-1$. The tape content at $t = i$ depends on the motion direction: for motion 0, we have $\sigma'_a = \sigma_a$ and $\sigma'_c = \sigma_c$, for motion -1, we have $\sigma'_a = \sigma_a$ and $\sigma'_b = \sigma_b$, and for motion $+1$ we have $\sigma'_b = \sigma_b$ and $\sigma'_c = \sigma_c$; in any case, σ_d at $t = i$ remains unchanged. In the next step, the machine can only continue to three configurations in the diagonal of time point $t = i+1$. The computation is analog to Figure 5.4, but this time, the grid of configuration covers all configurations located in the diagonal for each grid $t = 0, \ldots, N-1$. For our Turing machine encoding, both cell positions c and c' and time point t will be given as formal input parameters to module $m_2^{\mathbf{D}}$.

We setup the modules of $\mathbf{D}(T, N) = (m_1^{\mathbf{D}}, m_2^{\mathbf{D}}, m_3, m_4)$ as follows.

- The main module $m_1^{\mathbf{D}} = (dtm[\,], R_1^{\mathbf{D}})$, where $R_1^{\mathbf{D}}$ is the set of rules

$$o_1 \leftarrow \quad \cdots \quad o_\ell \leftarrow \tag{5.1}$$

$$accept \leftarrow conf[\mathbf{o}, \mathbf{o}, \mathbf{o}].yes \tag{5.2}$$

  Note that $\mathbf{o} = o_1, \ldots, o_\ell, \overline{o_1}, \ldots, \overline{o_\ell}$, i.e., input $\mathbf{o}, \mathbf{o}, \mathbf{o}$ represents the triple $(c, c', t) = (N{-}1, N{-}1, N{-}1)$ in binary, since for $i = 1, \ldots, \ell$ we must have o_i true and $\overline{o_i}$ false in all models.

- The library module

$$m_2^{\mathbf{D}} = (conf[c_1, \ldots, c_\ell, \overline{c_1}, \ldots, \overline{c_\ell}, c'_1, \ldots, c'_\ell, \overline{c'_1}, \ldots, \overline{c'_\ell}, t_1, \ldots, t_\ell, \overline{t_1}, \ldots, \overline{t_\ell}], R_2^{\mathbf{D}}) \ ,$$

  where $R_2^{\mathbf{D}}$ consists of the following groups of rules:

  offset rules for $1 \leq i \leq \ell$:

$$t_i^- \leftarrow op[\mathbf{t}].b_i^+ \qquad \overline{t_i^-} \leftarrow op[\mathbf{t}].\overline{b_i^+} \tag{5.3}$$

$$\begin{aligned} c_i^- &\leftarrow op[\mathbf{c}].b_i^+ & \overline{c_i^-} &\leftarrow op[\mathbf{c}].\overline{b_i^+} \\ c_i^+ &\leftarrow op[\mathbf{c}].b_i^- & \overline{c_i^+} &\leftarrow op[\mathbf{c}].\overline{b_i^-} \end{aligned} \tag{5.4}$$

$$\begin{aligned} c_i'^- &\leftarrow op[\mathbf{c}'].b_i^+ & c_i'^+ &\leftarrow op[\mathbf{c}'].b_i^- \\ \overline{c_i'^-} &\leftarrow op[\mathbf{c}'].\overline{b_i^+} & \overline{c_i'^+} &\leftarrow op[\mathbf{c}'].\overline{b_i^-} \end{aligned} \tag{5.5}$$

auxiliary rules (distinguish initialization phase from computation phase):

$$\mathit{init} \leftarrow \overline{t_1}, \dots, \overline{t_\ell} \qquad \mathit{start} \leftarrow \overline{c_1}, \dots, \overline{c_\ell}, \overline{c'_1}, \dots, \overline{c'_\ell} \tag{5.6}$$

$$\overline{\mathit{init}} \leftarrow t_1 \qquad \cdots \qquad \overline{\mathit{init}} \leftarrow t_\ell \tag{5.7}$$

initial rules:

$$\mathit{head} \leftarrow \mathit{init}, \mathit{start} \qquad s_0 \leftarrow \mathit{init}, \mathit{start} \qquad \text{␣} \leftarrow \mathit{init} \tag{5.8}$$

transition rules for $(s, \sigma, s', \sigma', +1) \in \delta$:

$$\mathit{head} \leftarrow \overline{\mathit{init}}, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].s, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].\sigma, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].\mathit{head} \tag{5.9}$$

$$s' \leftarrow \overline{\mathit{init}}, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].s, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].\sigma, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].\mathit{head} \tag{5.10}$$

$$\sigma \leftarrow \overline{\mathit{init}}, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].s, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].\sigma, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].\mathit{head} \tag{5.11}$$

$$\sigma'^{-1} \leftarrow \overline{\mathit{init}}, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].s, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].\sigma, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].\mathit{head} \tag{5.12}$$

$$\sigma' \leftarrow \overline{\mathit{init}}, \mathit{conf}[\mathbf{c}^+, \mathbf{c}^+, \mathbf{t}].\sigma'^{-1} \tag{5.13}$$

$$m^{+1} \leftarrow \overline{\mathit{init}}, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].s, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].\sigma, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].\mathit{head} \tag{5.14}$$

transition rules for $(s, \sigma, s', \sigma', 0) \in \delta$:

$$\mathit{head} \leftarrow \overline{\mathit{init}}, \mathit{conf}[\mathbf{c}, \mathbf{c}, \mathbf{t}^-].s, \mathit{conf}[\mathbf{c}, \mathbf{c}, \mathbf{t}^-].\sigma, \mathit{conf}[\mathbf{c}, \mathbf{c}, \mathbf{t}^-].\mathit{head} \tag{5.15}$$

$$s' \leftarrow \overline{\mathit{init}}, \mathit{conf}[\mathbf{c}, \mathbf{c}, \mathbf{t}^-].s, \mathit{conf}[\mathbf{c}, \mathbf{c}, \mathbf{t}^-].\sigma, \mathit{conf}[\mathbf{c}, \mathbf{c}, \mathbf{t}^-].\mathit{head} \tag{5.16}$$

$$\sigma' \leftarrow \overline{\mathit{init}}, \mathit{conf}[\mathbf{c}, \mathbf{c}, \mathbf{t}^-].s, \mathit{conf}[\mathbf{c}, \mathbf{c}, \mathbf{t}^-].\sigma, \mathit{conf}[\mathbf{c}, \mathbf{c}, \mathbf{t}^-].\mathit{head} \tag{5.17}$$

$$m^0 \leftarrow \overline{\mathit{init}}, \mathit{conf}[\mathbf{c}, \mathbf{c}, \mathbf{t}^-].s, \mathit{conf}[\mathbf{c}, \mathbf{c}, \mathbf{t}^-].\sigma, \mathit{conf}[\mathbf{c}, \mathbf{c}, \mathbf{t}^-].\mathit{head} \tag{5.18}$$

transition rules for $(s, \sigma, s', \sigma', -1) \in \delta$:

$$head \leftarrow \overline{init}, conf[\mathbf{c}^+, \mathbf{c}^+, \mathbf{t}^-].s, conf[\mathbf{c}^+, \mathbf{c}^+, \mathbf{t}^-].\sigma, conf[\mathbf{c}^+, \mathbf{c}^+, \mathbf{t}^-].head \tag{5.19}$$

$$s' \leftarrow \overline{init}, conf[\mathbf{c}^+, \mathbf{c}^+, \mathbf{t}^-].s, conf[\mathbf{c}^+, \mathbf{c}^+, \mathbf{t}^-].\sigma, conf[\mathbf{c}^+, \mathbf{c}^+, \mathbf{t}^-].head \tag{5.20}$$

$$\sigma \leftarrow \overline{init}, conf[\mathbf{c}^+, \mathbf{c}^+, \mathbf{t}^-].s, conf[\mathbf{c}^+, \mathbf{c}^+, \mathbf{t}^-].\sigma, conf[\mathbf{c}^+, \mathbf{c}^+, \mathbf{t}^-].head \tag{5.21}$$

$$\sigma'^{+1} \leftarrow \overline{init}, conf[\mathbf{c}^+, \mathbf{c}^+, \mathbf{t}^-].s, conf[\mathbf{c}^+, \mathbf{c}^+, \mathbf{t}^-].\sigma, conf[\mathbf{c}^+, \mathbf{c}^+, \mathbf{t}^-].head \tag{5.22}$$

$$\sigma' \leftarrow \overline{init}, conf[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}].\sigma'^{+1} \tag{5.23}$$

$$m^{-1} \leftarrow \overline{init}, conf[\mathbf{c}^+, \mathbf{c}^+, \mathbf{t}^-].s, conf[\mathbf{c}^+, \mathbf{c}^+, \mathbf{t}^-].\sigma, conf[\mathbf{c}^+, \mathbf{c}^+, \mathbf{t}^-].head \tag{5.24}$$

inertia rules for each $\sigma \in \Sigma$ (covering $c = 0, \dots, c' - 2, c' + 1, \dots, N - 1$):

$$\sigma \leftarrow \overline{init}, conf[\mathbf{c}, \mathbf{c}', \mathbf{t}^-].\sigma, conf[\mathbf{c}', \mathbf{c}', \mathbf{t}].head, conf[\mathbf{c}', \mathbf{c}', \mathbf{t}].m^{-1}, \\ ord[\mathbf{c}^+, \mathbf{c}'].\leq, ord[\mathbf{c}, \mathbf{c}'].\neq \tag{5.25}$$

$$\sigma \leftarrow \overline{init}, conf[\mathbf{c}, \mathbf{c}', \mathbf{t}^-].\sigma, conf[\mathbf{c}', \mathbf{c}', \mathbf{t}].head, conf[\mathbf{c}', \mathbf{c}', \mathbf{t}].m^{-1}, \\ ord[\mathbf{c}', \mathbf{c}].\leq, ord[\mathbf{c}, \mathbf{c}'].\neq \tag{5.26}$$

inertia rules for each $\sigma \in \Sigma$ (covering $c = 0, \dots, c' - 1, c' + 1, \dots, N - 1$):

$$\sigma \leftarrow \overline{init}, conf[\mathbf{c}, \mathbf{c}', \mathbf{t}^-].\sigma, conf[\mathbf{c}', \mathbf{c}', \mathbf{t}].head, conf[\mathbf{c}', \mathbf{c}', \mathbf{t}].m^{0}, \\ ord[\mathbf{c}, \mathbf{c}'].\leq, ord[\mathbf{c}, \mathbf{c}'].\neq \tag{5.27}$$

$$\sigma \leftarrow \overline{init}, conf[\mathbf{c}, \mathbf{c}', \mathbf{t}^-].\sigma, conf[\mathbf{c}', \mathbf{c}', \mathbf{t}].head, conf[\mathbf{c}', \mathbf{c}', \mathbf{t}].m^{0}, \\ ord[\mathbf{c}', \mathbf{c}].\leq, ord[\mathbf{c}, \mathbf{c}'].\neq \tag{5.28}$$

inertia rules for each $\sigma \in \Sigma$ (covering $c = 0, \dots, c' - 1, c' + 2, \dots, N - 1$):

$$\sigma \leftarrow \overline{init}, conf[\mathbf{c}, \mathbf{c}', \mathbf{t}^-].\sigma, conf[\mathbf{c}', \mathbf{c}', \mathbf{t}].head, conf[\mathbf{c}', \mathbf{c}', \mathbf{t}].m^{+1}, \\ ord[\mathbf{c}, \mathbf{c}'].\leq, ord[\mathbf{c}, \mathbf{c}'].\neq \tag{5.29}$$

$$\sigma \leftarrow \overline{init}, conf[\mathbf{c}, \mathbf{c}', \mathbf{t}^-].\sigma, conf[\mathbf{c}', \mathbf{c}', \mathbf{t}].head, conf[\mathbf{c}', \mathbf{c}', \mathbf{t}].m^{+1}, \\ ord[\mathbf{c}', \mathbf{c}^-].\leq, ord[\mathbf{c}, \mathbf{c}'].\neq \tag{5.30}$$

inertia rules for each $\sigma \in \Sigma$ and state *yes*:

$$\sigma \leftarrow \overline{init}, conf[\mathbf{c}, \mathbf{c}'^{+}, \mathbf{t}].\sigma \tag{5.31}$$
$$\sigma \leftarrow \overline{init}, conf[\mathbf{c}, \mathbf{c}'^{-}, \mathbf{t}].\sigma \tag{5.32}$$
$$yes \leftarrow \overline{init}, conf[\mathbf{c}, \mathbf{c}, \mathbf{t}^{-}].yes \tag{5.33}$$
$$yes \leftarrow \overline{init}, conf[\mathbf{c}, \mathbf{c}'^{+}, \mathbf{t}].yes \tag{5.34}$$
$$yes \leftarrow \overline{init}, conf[\mathbf{c}, \mathbf{c}'^{-}, \mathbf{t}].yes \tag{5.35}$$

- The library module $m_3 = (op[b_1, \ldots, b_\ell, \overline{b_1}, \ldots, \overline{b_\ell}], R_3)$, where R_3 consists of the following groups of rules:

successor rules for $1 \leq i < \ell$ and $1 \leq j \leq \ell$:

$$inv_1 \leftarrow \begin{array}{lll} \overline{inv_{i+1}} \leftarrow inv_i, \overline{b_i} & & \overline{b_j^+} \leftarrow b_j, inv_j \\ inv_{i+1} \leftarrow inv_i, b_i & & b_j^+ \leftarrow \overline{b_j}, inv_j \\ \overline{inv_{i+1}} \leftarrow \overline{inv_i} & & b_j^+ \leftarrow b_j, \overline{inv_j} \\ & & \overline{b_j^+} \leftarrow \overline{b_j}, \overline{inv_j} \end{array} \tag{5.36}$$

predecessor rules for $1 \leq i < \ell$:

$$\begin{array}{lll} b_1^- \leftarrow \overline{b_1} & \overline{b_{i+1}^-} \leftarrow b_{i+1}, \overline{c_i} & b_{i+1}^- \leftarrow \overline{b_{i+1}}, \overline{c_i} \\ \overline{c_1} \leftarrow \overline{b_1} & c_{i+1} \leftarrow b_{i+1}, \overline{c_i} & \overline{c_{i+1}} \leftarrow \overline{b_{i+1}}, \overline{c_i} \\ \overline{b_1^-} \leftarrow b_1 & b_{i+1}^- \leftarrow b_{i+1}, c_i & \overline{b_{i+1}^-} \leftarrow \overline{b_{i+1}}, c_i \\ c_1 \leftarrow b_1 & c_{i+1} \leftarrow b_{i+1}, c_i & c_{i+1} \leftarrow \overline{b_{i+1}}, c_i \end{array} \tag{5.37}$$

- and the library module

$$m_4 = (ord[x_1, \ldots, x_\ell, \overline{x_1}, \ldots, \overline{x_\ell}, y_1, \ldots, y_\ell, \overline{y_1}, \ldots, \overline{y_\ell}], R_4) \ ,$$

where R_4 consists of the following groups of rules:

inequality rules for $1 \leq i \leq \ell$:

$$\neq \ \leftarrow x_i, \overline{y_i} \qquad\qquad \neq \ \leftarrow \overline{x_i}, y_i \tag{5.38}$$

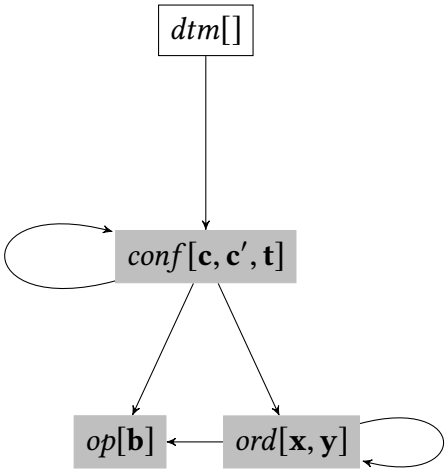


Figure 5.6: Module dependencies of a deterministic Turing machine simulation

equality rules for $1 < i \leq \ell$:

$$\begin{aligned} =^1 &\leftarrow x_1, y_1 & =^1 &\leftarrow \overline{x_1}, \overline{y_1} \\ =^i &\leftarrow x_i, y_i, =^{i-1} & =^i &\leftarrow \overline{x_i}, \overline{y_i}, =^{i-1} \end{aligned} \tag{5.39}$$

successor rules for $1 \leq i \leq \ell$:

$$z_i \leftarrow op[\mathbf{x}].b_i^+ \tag{5.40}$$

$$\overline{z_i} \leftarrow op[\mathbf{x}].\overline{b_i^+} \tag{5.41}$$

order rules:

$$\leq \leftarrow =^\ell \tag{5.42}$$

$$\leq \leftarrow ord[\mathbf{z}, \mathbf{y}].\leq \tag{5.43}$$

Figure 5.6 shows the inter-module dependencies of the modules in $\mathbf{D}(T, N)$: the modules represent nodes in the directed graph, and there exists an edge from a module m_i to a module m_j whenever m_i has a module call to m_j. The graph shows the main module $m_1^{\mathbf{D}}$ in white, while the library modules $m_2^{\mathbf{D}}$, m_3, m_4 with input are shown in gray. The structural dependencies make it clear that $\mathbf{D}(T, N)$ is cyclic: both $m_2^{\mathbf{D}}$ and m_4 call themselves, while m_3 is a sink module that do not call any other module.

We show now that we can simulate the computation of a deterministic Turing machine T on input I with $\mathbf{D}(T,N)$ and prove that T accepts input I within $N = 2^{m^k}$ steps if and only if $\mathit{accept} \in \mathrm{lfp}(\mathbf{D}(T,N))$.

($\Rightarrow$) Suppose T accepts input I within N steps. First, we get that $T_{\mathbf{D}(T,N)}\uparrow_0(\mathbf{M}_\varnothing) = \mathbf{M}_\varnothing$. Let $X \subseteq \mathrm{HB}_{\mathbf{P}}|_{\mathbf{c},\mathbf{c}',\mathbf{t}}$, $Y \subseteq \mathrm{HB}_{\mathbf{P}}|_{\mathbf{x},\mathbf{y}}$, and $B \subseteq \mathrm{HB}_{\mathbf{P}}|_{\mathbf{b}}$. Then, $T_{\mathbf{D}(T,N)}\uparrow_1(\mathbf{M}_\varnothing)$ contains at $\mathit{dtm}[\varnothing]$ all the facts o_i, at each $\mathit{conf}[X]$ it is equal to X, at each $\mathit{op}[B]$ the facts $B \cup \{\mathit{inv}_1\}$, and at each $\mathit{ord}[Y]$ the facts Y. Note that some instantiations may also contain invalid bit representations S for time points and cell positions, thus also the sets M_i/S of a model $\mathbf{M}$. This does not harm, in fact, we can partition the least fixpoint computation into two parts: one part that contains only instantiations that represent valid cells and time points, and one part that has arbitrary outcome. We defer the proof for that claim until later and continue to look into instantiations with valid bit vectors only.

Let $C_i \subseteq \mathrm{HB}_{\mathbf{P}}|_{\mathbf{c}}$ and $C'_j \subseteq \mathrm{HB}_{\mathbf{P}}|_{\mathbf{c}'}$ and $T_k \subseteq \mathrm{HB}_{\mathbf{P}}|_{\mathbf{t}}$ be valid bit representations for cell positions c_i and c'_j and time points t_k. Now, $T_{\mathbf{D}(T,N)}\uparrow_2(\mathbf{M}_\varnothing)$ contains at $\mathit{conf}[C_0 \cup C'_0 \cup T_0]$ the fact init, at $\mathit{conf}[C_0 \cup C'_0 \cup T_i]$ for $0 \leq i < N$ the fact start, and at $\mathit{conf}[C_i \cup C'_j \cup T_k]$ for $1 \leq i \leq j \leq k < N$ the fact $\overline{\mathit{init}}$. Moreover, we have that $T_{\mathbf{D}(T,N)}\uparrow_2(\mathbf{M}_\varnothing)$ contains for a valid bit representations B at each $\mathit{op}[B]$ the successor and predecessor representation of B stored in the atoms $b_i^+, \overline{b_i^+}, b_i^-, \overline{b_i^-}$, and at each $\mathit{ord}[X \cup Y]$ for valid bit representation X and Y for integers $x, y \in \{0, \dots, N-1\}$ the fact $\neq$ if $x \neq y$, or alternatively the facts $=^\ell$ and $\leq$ when $x = y$. Hence, we have computed the binary relation $\neq$ and $=$ on the set $\{0, \dots, N-1\}$, and the subset of the binary relation $\leq$ on $\{0, \dots, N-1\}$, viz. all the pairs along the diagonal of $\{0, \dots, N-1\}^2$. Thereafter, the initial rules will become applicable.

In the next step, $T_{\mathbf{D}(T,N)}\uparrow_3(\mathbf{M}_\varnothing)$ contains at $\mathit{conf}[C_0 \cup C'_0 \cup T_0]$ then the facts head and s_0, and at $\mathit{conf}[C_i \cup C'_j \cup T_0]$ for $0 \leq i \leq j < N$ the fact $\llcorner\!\lrcorner$, thus the blank input tape, whose content consists only of blank tape symbols $\llcorner\!\lrcorner$, is now stored in our interpretation. Moreover, for valid bit representation X, Y, Z for integers $x, y, z \in \{0, \dots, N-1\}$ such that $z = x+1$ and $z = y$, our interpretation contains at $\mathit{ord}[X \cup Y]$ the facts from $\mathbf{z}$ that represent the successor of $\mathbf{x}$, and the fact $\leq$. Now the interpretation additionally stores $x \leq x+1$ for the binary relation $\leq$ on $\{0, \dots, N-1\}$. The following steps $T_{\mathbf{D}(T,N)}\uparrow_{3+i}(\mathbf{M}_\varnothing)$ for $1 < i < N$ continue to add to the binary relation $\leq$ further line segments $(x, x+i)$ located above and in parallel to the diagonal until we reach $T_{\mathbf{D}(T,N)}\uparrow_{3+N-1}(\mathbf{M}_\varnothing)$, where we have for integers $x, y, z \in \{0, \dots, N-1\}$ such that $z = x + N - 1$ and $z = y$ the fact $\leq$, hence the interpretation contains now all pairs of the binary relation $\leq$ on $\{0, \dots, N-1\}$.

Interlaced with the computation for $\leq$, the steps that follow $T_{\mathbf{D}(T,N)}\uparrow_3(\mathbf{M}_\varnothing)$ compute the outcome of the Turing machine run according to the transition relation δ, i.e., we have $\overline{\mathit{init}}$ true in all value calls corresponding to time point t_i for $i > 0$, hence the transition and some inertia rules become applicable. And since we can infer all $\leq$ at

$ord[X \cup Y]$ after we have reached step $T_{\mathbf{D}(T,N)}\uparrow_{3+N-1}(\mathbf{M}_{\varnothing})$, we are then in the position to apply all inertia rules.

In the first phase of the Turing machine run, the input I is prepared. The transition rules only examine configurations in the diagonal of the grid for $t-1$ of form $(c-1, c-1, t-1)$, $(c, c, t-1)$, or $(c+1, c+1, t-1)$, i.e., they inspect the former time point $t-1$ and thus infer the current state s' and symbol σ' depending on the motion direction. Hence, the actual tape for the computation can only be found in the diagonal $(0,0), (1,1), \ldots, (N-1, N-1)$ for each time point t.

The inertia rules (5.25)–(5.35) are more sophisticated. They scan for the head in the diagonal (c', c') and copy σ depending on the motion direction $d \in \{-1, 0, +1\}$ from the interpretation encoding the previous time point $t-1$ whenever $c' - \min(d, 0) < c$ (rules (5.25), (5.27), and (5.29)) or $c < c' - \max(d, 0)$ (rules (5.26), (5.28), and (5.30)). This way we copy all σ that have not been changed by the computation into time point t at all (c, c') whenever the head is located at (c', c'). Hence unchanged tape content has been copied into the interpretation horizontally connected through (c', c'), which stores *head* and depending on the motion direction contains $\sigma', \sigma'^{-1}, \sigma'^{+1}$ derived from the transition rules. In a next step, inertia rules (5.31)–(5.32) become applicable, whose purpose is to copy at time point t every symbol σ that is located at $(c, c'-1)$ or $(c, c'+1)$ into the interpretation for (c, c', t), i.e., we distribute the same tape symbol σ vertically at c. This implies that we will also fill the diagonal (c, c, t) for every c and thus the tape that is used for the transition rules is complete and can be used for the next transition.

After I has been prepared using δ, the run of T on I is simulated and at $conf[C_i \cup C'_j \cup T_k]$ for $0 \leq i \leq j \leq k < N$ we get the respective configuration of T for cell position c_i, c'_j and time point t_k. Since T accepts I, there is an accepting configuration $\gamma = (yes, w, \sigma' \cdots)$ for the run of T on I given by a transition $(s, \sigma, yes, \sigma', \cdot) \in \delta$. This transition is encoded by a transition rule with rule head *yes*, hence $\mathrm{lfp}(\mathbf{D}(T,N)), conf[C_i \cup C'_i \cup T_k] \vDash yes$ for an i and k such that $0 \leq i \leq k < N$ and from inertia rules (5.33)–(5.35) we eventually get $\mathrm{lfp}(\mathbf{D}(T,N)), conf[C_{N-1} \cup C'_{N-1} \cup T_{N-1}] \vDash yes$. This makes rule (5.2) applicable and we end up with $\mathrm{lfp}(\mathbf{D}(T,N)), dtm[\varnothing] \vDash accept$. item 1 showed that the least fixpoint needs exponential time to compute, hence the claim follows.

($\Leftarrow$) Suppose $\mathrm{lfp}(\mathbf{D}(T,N)), dtm[\varnothing] \vDash accept$. For each $conf[C_i \cup C'_i \cup T_k]$ such that $0 \leq i \leq k < N$, we can extract a configuration $\gamma_k = (s, \sigma^0 \cdots \sigma^{i-1}, \sigma^i \cdots \sigma^{N-1})$ for the machine T whenever $\mathrm{lfp}(\mathbf{D}(T,N)), conf[C_i \cup C'_i \cup T_k] \vDash s$ and $\mathrm{lfp}(\mathbf{D}(T,N)), conf[C_i \cup C'_i \cup T_k] \vDash head$ and for all $0 \leq j < N$, $\mathrm{lfp}(\mathbf{D}(T,N)), conf[C_j \cup C'_j \cup T_k] \vDash \sigma^j$. Since we can infer *accept*, we must have $\mathrm{lfp}(\mathbf{D}(T,N)), conf[C_{N-1} \cup C'_{N-1} \cup T_{N-1}] \vDash yes$, and as the transition rules encode the relation δ of T, there must be a transition $(s, \sigma, yes, \sigma', \cdot)$ in δ that fires the respective transition rule. Hence the computation $\gamma_0\gamma_1 \cdots \gamma_{N-1}$ is accepting, as we can reach a configuration γ_k from γ_0 such that $0 \leq k < N$ and $\gamma_k = (yes, w, u)$. Thus, T accepts I within $N = 2^{\ell}$ steps and halts in state *yes*.

It remains to show that the least fixpoint computation can be partitioned into a valid and an invalid part. If B is a valid representation of the bit vector $b_1,\dots,b_\ell, \overline{b_1},\dots,\overline{b_\ell}$, then $T_{\mathbf{D}(T,N)}\uparrow_2(\mathbf{M}_\varnothing)$ contains at each $op[B]$ the successor and predecessor representation of B by the atoms $b_i^+, \overline{b_i^+}, b_i^-, \overline{b_i^-}$. If B is not a valid representation, i.e., $b_i, \overline{b_i} \in B$ or $b_i, \overline{b_i} \notin B$ for some $i \in \{1,\dots,\ell\}$, then $T_{\mathbf{D}(T,N)}\uparrow_2(\mathbf{M}_\varnothing)$ at $op[B]$ will either

1. contain at least a pair $b_i^+, \overline{b_i^+}$ or $b_i^-, \overline{b_i^-}$ of complementary bits, or
2. both $b_i^+, \overline{b_i^+}$ or $b_i^-, \overline{b_i^-}$ are not contained.

Since all transition and inertia rules contain module atoms of form $conf[\mathbf{c}^\star, \mathbf{c}^\star, \cdot].p$, $conf[\mathbf{c}', \mathbf{c}', \cdot].p$, $conf[\mathbf{c}^\star, \mathbf{c}'^\star, \cdot].p$, $ord[\mathbf{c}, \mathbf{c}'].p$, or $ord[\mathbf{c}', \mathbf{c}].p$ such that $\star$ is either void or from $\{+,-\}$, hence taking an ill-formed $\mathbf{c}$, $\mathbf{c}'$, and $\mathbf{t}$ or an ill-formed successor or predecessor for $\mathbf{c}$, $\mathbf{c}'$, and $\mathbf{t}$ as input, each ill-valued instantiation will only take truth values of ill-valued parts of an interpretation, and each valid instantiation will only take truth values of valid parts of the interpretation. Since rules (5.1) represent a valid bit vector, the rule (5.2) is applicable only in a valid instantiation, hence our result follows.

The construction of $\mathbf{D}(T,N)$ is feasible in time polynomial in the size of I, thus deciding whether $\alpha \in \mathrm{lfp}(\mathbf{P})$ is EXP-complete. □

5.3.2 Proof of Theorem 5.2, item 2

Showing membership in NEXP works as follows. We first guess an interpretation $\mathbf{M}$ for a normal MLP $\mathbf{P}$. Every interpretation $\mathbf{M}$ of $\mathbf{P}$ uses at most $n \cdot 2^{|\mathrm{HB}_\mathbf{P}|}$ value calls, thus checking that all rules of $I(\mathbf{P})$ are satisfied and whether $\mathbf{M}$ is a minimal model for $f\,\mathbf{P}^\mathbf{M}$ takes exponentially many steps.

Hardness can be shown by adapting the Turing machine encoding of item 1 in §5.3.1. Given a nondeterministic TM T, we can make small modifications to the encoding $\mathbf{D}(T,N)$ from above and use an MLP

$$\mathbf{N}(T,N) = (m_1^\mathbf{N}, m_2^\mathbf{N}, m_3, m_4, m_5^\mathbf{N})$$

with the additional library module $m_5^\mathbf{N}$ that encodes branches in a nondeterministic computation tree.

We use the following additional propositional atoms in our modules $m_2^\mathbf{N}$ and $m_5^\mathbf{N}$ to encode branches in a computation of T:

- the atom $b_{s,\sigma,i}$ to encode that at time point t, the Turing machine chooses branch i for transition $(s,\sigma,s_i',\sigma_i',d_i) \in \delta$; and

- the atoms $\overline{\mathit{final}}$ and bad to encode that T is not at the final time point $N-1$ and whether $\mathbf{t}$ encodes an invalid binary representation for a time point t, respectively.

We use the following modules in our encoding:

- the main module $m_1^{\mathbf{N}} = (\mathit{ntm}[], R_1^{\mathbf{N}})$ such that $R_1^{\mathbf{N}}$ is the set of rules $R_1^{\mathbf{D}}$ and the additional rule

$$\mathit{accept} \leftarrow \text{not } \mathit{accept} \tag{5.44}$$

- the library module

$$m_2^{\mathbf{N}} = (\mathit{conf}[c_1, \dots, c_\ell, \overline{c_1}, \dots, \overline{c_\ell}, c'_1, \dots, c'_\ell, \overline{c'_1}, \dots, \overline{c'_\ell}, t_1, \dots, t_\ell, \overline{t_1}, \dots, \overline{t_\ell}], R_2^{\mathbf{N}}) \ ,$$

where $R_2^{\mathbf{N}}$ consists of $R_2^{\mathbf{D}}$ with the following modifications for the transition and inertia rules: (5.9)–(5.35) get for a transition $(s, \sigma, s'_i, \sigma'_i, d_i) \in \delta$ the additional body atom

$$\mathit{branch}[\mathbf{t}^-].b_{s,\sigma,i} \tag{5.45}$$

Take, as an example, the rule (5.10) from $m_2^{\mathbf{D}}$. Then (5.10) in $m_2^{\mathbf{N}}$ amounts to the rules

$$\begin{aligned} s'_1 \leftarrow\ & \overline{\mathit{init}}, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].s, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].\sigma, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].\mathit{head}, \\ & \mathit{branch}[\mathbf{t}^-].b_{s,\sigma,1} \\ & \vdots \\ s'_j \leftarrow\ & \overline{\mathit{init}}, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].s, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].\sigma, \mathit{conf}[\mathbf{c}^-, \mathbf{c}^-, \mathbf{t}^-].\mathit{head}, \\ & \mathit{branch}[\mathbf{t}^-].b_{s,\sigma,j} \end{aligned}$$

for all $+1$-transitions $(s, \sigma, s'_1, \sigma'_1, +1), \dots, (s, \sigma, s'_j, \sigma'_j, +1) \in \delta$ such that $1 \leq j \leq k$, where k is the number of all (s, σ)-transitions of form $(s, \sigma, s'_i, \sigma'_i, d_i) \in \delta$.

- the library module m_3 ($\mathit{op}[b_1, \dots, b_\ell, \overline{b_1}, \dots, \overline{b_\ell}]$) from $\mathbf{D}(T, N)$;
- the library module m_4 ($\mathit{ord}[x_1, \dots, x_\ell, \overline{x_1}, \dots, \overline{x_\ell}, y_1, \dots, y_\ell, \overline{y_1}, \dots, \overline{y_\ell}]$) from $\mathbf{D}(T, N)$;
- and the library module $m_5^{\mathbf{N}} = (\mathit{branch}[t_1, \dots, t_\ell, \overline{t_1}, \dots, \overline{t_\ell}], R_5^{\mathbf{N}})$, where $R_5^{\mathbf{N}}$ is the following groups of rules.

auxiliary rules for $1 \leq i \leq \ell$:

$$\overline{\mathit{final}} \leftarrow \overline{t_i} \qquad \mathit{bad} \leftarrow t_i, \overline{t_i} \qquad \mathit{bad} \leftarrow \text{not } t_i, \text{not } \overline{t_i} \tag{5.46}$$

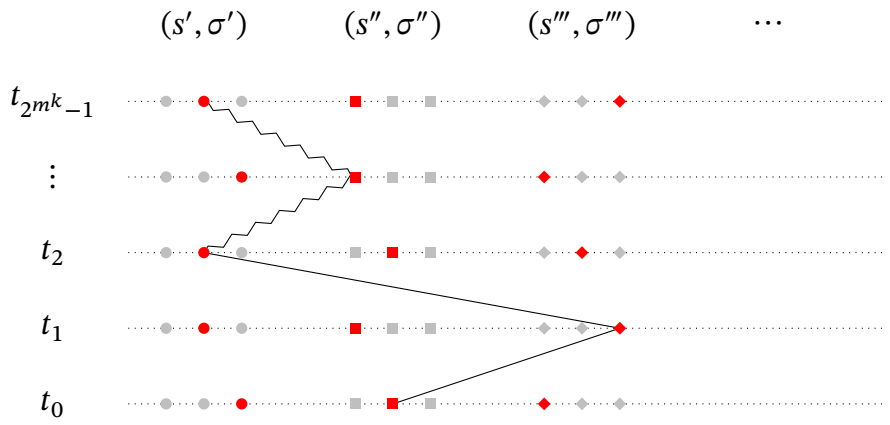


Figure 5.7: nondeterministic Turing machine run

branching rules for $(s, \sigma, s'_i, \sigma'_i, d_i) \in \delta$ such that $1 \leq i \leq k$:

$$b_{s,\sigma,i} \leftarrow \overline{final}, \text{not } bad, \text{not } b_{s,\sigma,1}, \ldots, \text{not } b_{s,\sigma,i-1}, \text{not } b_{s,\sigma,i+1}, \ldots, \text{not } b_{s,\sigma,k} \tag{5.47}$$

Note that $k = 1$ for deterministic transitions and thus rules (5.47) collapse to the single rule $b_{s,\sigma,1} \leftarrow \overline{final}$, not bad, i.e., the transition is determined simply by s and σ.

Since Turing machine computations start in state s_0, connecting possible successor states form a computation tree. Each level of the tree represents a time point in a computation. In an actual run $\Gamma = \gamma_0, \ldots, \gamma_{N-1}$ of an NTM we will have one transition made for each successor configuration γ_j to γ_{j+1}, thus a run is a path from root s_0 to a leaf state in the computation tree. Each successor configuration in a run is thus determined by the chosen i^{th} transition $(s, \sigma, s'_i, \sigma'_i, d_i) \in \delta$. The branching rules above generate all possible branches in the computation tree for each time point t.

Figure 5.7 shows shapes along dotted lines, which stand for all possible choices of transitions $(s, \sigma, s'_i, \sigma'_i, d_i) \in \delta$ that can be taken in a particular time point t, where groups of equal shape share the same pair (s, σ). At time point t, we must therefore select for each group (s, σ) exactly one transition $(s, \sigma, s'_i, \sigma'_i, d_i) \in \delta$ when the machine is in state s and the head is on σ. The set of all those transitions (s, σ) are then the red-colored shapes for each time point t. When we connect the red shapes that have been chosen to go from a configuration γ_j to γ_{j+1}, we can see the path that has been chosen by a run Γ in the computation tree of the NTM. The red shapes connected by lines then record the actual transition that has been taken in a run Γ (in this figure, the computation starts in (s'', σ'') at time point 0 and then continues in (s''', σ''') at time point 1, then continues at (s', σ') at time point 2, and via a path to (s'', σ'') it ends up in (s', σ') at time point $N - 1$).

Our encoding satisfies the requirement that only one choice i for each pair (s, σ) can be made for a given time point t. Intuitively, at each time point t we guess a branch

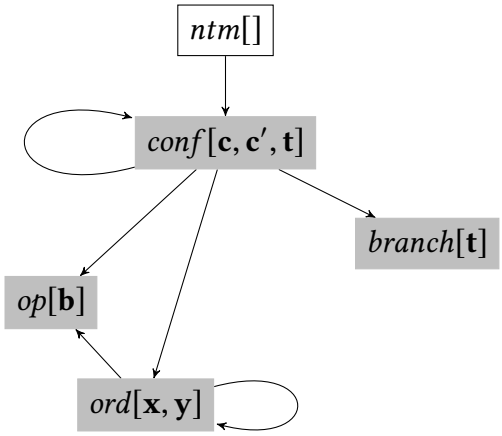


Figure 5.8: Module dependencies of a nondeterministic Turing machine simulation

of the nondeterministic computation in $m_5^{\mathbf{N}}$ and take then according to the transition relation of the pair (s, σ) all possible k branches of the computation into account, and eventually kill all nonaccepting branches using the additional constraint (5.44) in $m_1^{\mathbf{N}}$.

Figure 5.8 shows the inter-module dependencies of the modules in $\mathbf{N}(T, N)$. The graph shows the main module $m_1^{\mathbf{N}}$ in white, while the library modules $m_2^{\mathbf{N}}$, m_3, m_4, $m_5^{\mathbf{N}}$ with input are shown in gray. Compared to the graph for $\mathbf{D}(T, N)$ in Figure 5.6, the graph $\mathbf{N}(T, N)$ uses the adapted modules $m_1^{\mathbf{N}}$ and $m_2^{\mathbf{N}}$, and adds the module $m_5^{\mathbf{N}}$ with an additional edge from $m_2^{\mathbf{N}}$ to $m_5^{\mathbf{N}}$. Hence, $\mathbf{N}(T, N)$ is also cyclic: both $m_2^{\mathbf{N}}$ and m_4 call themselves, while m_3 and $m_5^{\mathbf{N}}$ are sink modules that does not call any other module.

We show now that we can simulate the computation of a nondeterministic Turing machine T on input I with $\mathbf{N}(T, N)$ and prove that T accepts input I and halts in state *yes* in less than $N = 2^{m^k}$ steps if and only if $\mathbf{N}(T, N)$ has an answer set.

($\Rightarrow$) Suppose T accepts input I within N steps and halts in state *yes*. Then there is a sequence of chosen configurations $\Gamma = \gamma_0, \ldots, \gamma_{N-1}$ such that the final configuration γ_{N-1} has state *yes*. We show now that we can build an answer set for $\mathbf{N}(T, N)$ from Γ.

Let $\mathbf{M}^{\mathbf{N}}$ denote the answer set for $\mathbf{N}(T, N)$ that will be obtained from Γ as follows: set $M_1^{\mathbf{N}}/\emptyset = \{o_1, \ldots, o_\ell, \mathit{accept}\}$ and $M_3^{\mathbf{N}}/B$ to the successor and predecessor of the set B encoding a bit vector $\mathbf{b}$. For sets X, Y, Z that encode bit vectors $\mathbf{x}, \mathbf{y}, \mathbf{z}$ representing integers $x, y, z \in \{0, \ldots, N-1\}$ in binary, we set $M_4^{\mathbf{N}}/(X \cup Y) = X \cup Y \cup Z \cup O_{X,Y,Z}$, where $O_{X,Y,Z}$ stores the bits of $\mathbf{z}$, the atom $\leq$ whenever $x \leq y$, the atom $=^i$ whenever the i^{th} bits of x and y agree on their truth values, and the atom $\neq$ whenever $x = y$.

Then, for a configuration $\gamma_i = (s, \hat{w}\hat{\sigma}, \sigma\hat{u})$ in Γ, let γ_{i+1} be one of the successor configurations $(s'_j, \hat{w}\hat{\sigma}, \sigma'_j\hat{u})$, $(s'_j, \hat{w}\hat{\sigma}\sigma'_j, \hat{u})$, or $(s'_j, \hat{w}, \hat{\sigma}\sigma'_j\hat{u})$ that can be obtained from a

transition $\tau_i = (s, \sigma, s'_j, \sigma'_j, d_j) \in \delta$ at time point i. We can now set

- $M_5^{\mathbf{N}}/T_i = \{\overline{final}\} \cup \{b_{s,\sigma,j}\} \cup \{b_{s',\sigma',1} \mid s' \neq s \wedge \sigma' \neq \sigma\}$, for T_i representing time points i such that $0 \leq i < N-1$,
- $M_5^{\mathbf{N}}/T_{N-1} = T_{N-1}$, and
- $M_5^{\mathbf{N}}/S = S \cup \{bad\} \cup \{\overline{final} \mid \overline{t_j} \in S \text{ for } j \in \{1, \ldots, \ell\}\}$ for invalid bit representations S.

Given a configuration $\gamma_i = (s, w, u)$ of the sequence Γ, i represents the time point t, and $|w|$ is the position c of the tape head. We can form the input for module $m_2^{\mathbf{N}}$ as follows. First encode i in binary as $\mathbf{t}$ using the atoms t_j and $\overline{t_j}$, for $1 \leq j \leq \ell$. Then encode $|w|$ in binary as $\mathbf{c}$ and for $|w| \leq c'$, encode c' as $\mathbf{c}'$ using, for $1 \leq j \leq \ell$, atoms $c_j, \overline{c_j}$ and $c'_j, \overline{c'_j}$, respectively. Let the resulting set of atoms be $C_{\gamma_i}^{c'}$ consisting of atoms from $\mathbf{c}$, $\mathbf{c}'$, and $\mathbf{t}$. Then, setup

- $M_2^{\mathbf{N}}/C_{\gamma_i}^{c'}$, for $1 \leq i < N$ and $c \leq c'$, to $C_{\gamma_i}^{c'}$ and all atoms such that it satisfies the offset and transition rules according to the transition τ_i that can be obtained from Γ, and
- $M_2^{\mathbf{N}}/C_{\gamma_0}^{c'}$ to $C_{\gamma_0}^{c'} \cup \{head, s_0, \sqcup, init, start\} \cup O_{c,c',t}$, where $O_{c,c',t}$ store the atoms representing $\mathbf{c}'^{-}$, $\mathbf{c}'^{+}$, and $\mathbf{t}^{-}$ for $c-1$, $c+1$ and $t-1$, respectively.

Whenever we have an invalid bit representation S we set $M_2^{\mathbf{N}}/S$ accordingly; here, the transition rules are not applicable.

One can verify that the interpretation $\mathbf{M}^{\mathbf{N}}$ is a model of $\mathbf{N}(T, N)$. We show now that it is also a minimal model of $f\,\mathbf{N}(T, N)^{\mathbf{M}^{\mathbf{N}}}$. Towards a contradiction, assume that there exists an interpretation $\mathbf{M}' < \mathbf{M}^{\mathbf{N}}$ such that $\mathbf{M}'$ is a model of $f\,\mathbf{N}(T, N)^{\mathbf{M}^{\mathbf{N}}}$. From the construction of $\mathbf{M}^{\mathbf{N}}$ and the rules in $\mathbf{N}(T, N)$, we can only have that M_5'/T_i does not have an atom $b_{s,\sigma,j}$ which is in $M_5^{\mathbf{N}}/T_i$. But we must have a rule of form (5.47) in $f\,\mathbf{N}(T, N)^{\mathbf{M}^{\mathbf{N}}}$, which contradicts the assumption that $\mathbf{M}'$ is a model. Thus, $\mathbf{M}^{\mathbf{N}}$ is an answer set of $\mathbf{N}(T, N)$.

($\Leftarrow$) Suppose $\mathbf{M}^{\mathbf{N}}$ is an answer set of $\mathbf{N}(T, N)$. We can extract a configuration sequence $\Gamma = \gamma_0, \gamma_1, \ldots, \gamma_{N-1}$ for the machine T in a similar way as we have shown in item 1. Set $\gamma_0 = (s_0, \sqcup, \sqcup)$, and let successor configurations γ_k for $k > 0$ be of the form (s_j, w, u).

For each time point $0 < k < N$ and cell position $0 \leq i < N$ we inspect value calls $conf[C_i \cup C'_i \cup T_k]$ and their predecessors $conf[C_{i-1} \cup C'_{i-1} \cup T_{k-1}]$, $conf[C_i \cup C'_i \cup T_{k-1}]$, and $conf[C_{i+1} \cup C'_{i+1} \cup T_{k-1}]$. Whenever $\mathbf{M}^{\mathbf{N}}$, $conf[C_i \cup C'_i \cup T_k] \models head$

and $\mathbf{M}^{\mathbf{N}}, conf[C_{i+d} \cup C'_{i+d} \cup T_{k-1}] \models head$ for $d \in \{-1, 0, +1\}$, we can pinpoint the transition from δ that brought us from γ_{k-1} to γ_k as follows.

First we generate the words left and right from the tape head at time point k. As the head is in position i, the word on the left is of the form $w = \sigma^0 \cdots \sigma^{i-1}$ and the word at the head position is $u = \sigma^i \sigma^{i+1} \cdots \sigma^{N-1}$. For $0 \leq i' < i$ and $i < i' < N$, we get $\sigma^{i'}$ from $\mathbf{M}^{\mathbf{N}}, conf[C_{i'} \cup C'_{i'} \cup T_k] \models \sigma^{i'}$, and at the head position i, we let $\sigma^i = \sigma_j$ for $\mathbf{M}^{\mathbf{N}}, conf[C_i \cup C'_i \cup T_k] \models \sigma_j$.

The state is given by $\mathbf{M}^{\mathbf{N}}, conf[C_i \cup C'_i \cup T_k] \models s_j$, and we determine the motion direction $d_j \in \{-1, 0, +1\}$ by reading $\mathbf{M}^{\mathbf{N}}, conf[C_i \cup C'_i \cup T_k] \models m^{-1}$, $\mathbf{M}^{\mathbf{N}}, conf[C_i \cup C'_i \cup T_k] \models m^0$, or $\mathbf{M}^{\mathbf{N}}, conf[C_i \cup C'_i \cup T_k] \models m^{+1}$, respectively.

Now given s_j, σ_j, d_j, there must exist a transition $(\hat{s}, \hat{\sigma}, s_j, \sigma_j, d_j) \in \delta$, which we can locate from the single atom $b_{\hat{s},\hat{\sigma},j}$ such that $\mathbf{M}^{\mathbf{N}}, branch[T_{k-1}] \models b_{\hat{s},\hat{\sigma},j}$.

Since we can infer $\mathbf{M}^{\mathbf{N}}, ntm[\varnothing] \models accept$ as required by rule (5.44), and as $\mathbf{M}^{\mathbf{N}}$ is a minimal model, we also must have $\mathbf{M}^{\mathbf{N}}, conf[C_{N-1} \cup C'_{N-1} \cup T_{N-1}] \models yes$ as required by rule (5.2). Thus there must be an accepting configuration γ in the computation Γ generated by $\mathbf{M}^{\mathbf{N}}$. Since we have inferred all configurations by transitions from δ, this means that Γ must be accepting and that T halts in state *yes* on input I, thus T accepts I within 2^{m^k} steps.

The NTM encoding $\mathbf{N}(T, N)$ can be generated in time polynomial in the size of I, hence deciding whether a normal $\mathbf{P}$ has an answer set is NEXP-complete. □

5.3.3 Proof of Theorem 5.2, item 3

An algorithm for checking whether there exists an answer set $\mathbf{M}$ for a disjunctive $\mathbf{P}$ is as follows. First we guess an interpretation $\mathbf{M}$ for $\mathbf{P}$. Since $\mathbf{M}$ uses at most $n \cdot 2^{|\mathrm{HB_P}|}$ value calls, we check that all rules from $I(\mathbf{P})$ are satisfied and compute $f\,\mathbf{P}^{\mathbf{M}}$ in time exponential in the size of $\mathbf{P}$. The check whether whether $\mathbf{M}$ is a minimal model for $f\,\mathbf{P}^{\mathbf{M}}$ by using an oracle for co-NP, thus we arrive at an $\mathrm{NEXP}^{\mathrm{NP}}$ upper bound.

The hardness part of this proof can be shown by an encoding of alternating Turing machines. An exponential-time bounded ATM T that starts in an $\exists$-state s_0 and have exactly one alternation to a $\forall$-state solve the instances of problems in $\mathrm{NEXP}^{\mathrm{NP}}$: let Σ_k and Π_k denote k-alternation bounded ATMs as defined by Chandra et al. (1981), and $A\Sigma_k^e$ and $A\Pi_k^e$ denote the class of sets accepted by Σ_k and Π_k machines which accept in exponential time, respectively, then following Chandra et al. (1981) we obtain $\Sigma_k^e = A\Sigma_k^e$ and $\Pi_k^e = A\Pi_k^e$.

For the purpose of showing $\mathrm{NEXP}^{\mathrm{NP}}$-hardness, we will adapt the encoding $\mathbf{N}(T, N)$ from item 2 from §5.3.2 and use the MLP

$$\mathbf{A}(T, N) = (m_1^{\mathbf{A}}, m_2^{\mathbf{A}}, m_3, m_4, m_5^{\mathbf{A}})$$

where, compared to $\mathbf{N}(T,N)$, $m_1^{\mathbf{A}}$, $m_2^{\mathbf{A}}$, and $m_5^{\mathbf{A}}$ receive minor modifications to encode universal configuration sequences for an ATM computation.

We use the following additional propositional atom in modules $m_1^{\mathbf{A}}$, $m_2^{\mathbf{A}}$, and $m_5^{\mathbf{A}}$ to capture the alternation to a universal configuration sequence in a computation of T:

- atom *forall* encodes that starting from a time point t such that $0 < t < N$, Turing machine configurations are universal.

The modules of $\mathbf{A}(T,N)$ have the following structure:

- the main module $m_1^{\mathbf{A}} = (atm[], R_1^{\mathbf{A}})$, where $R_1^{\mathbf{A}}$ is the set of rules $R_1^{\mathbf{N}}$, where rule (5.2) is replaced by rule

$$accept \leftarrow conf[\mathbf{o},\mathbf{o},\mathbf{o}].yes, conf[\mathbf{o},\mathbf{o},\mathbf{o}].forall \tag{5.48}$$

- the library module

$$m_2^{\mathbf{A}} = (conf[c_1,\dots,c_\ell,\overline{c_1},\dots,\overline{c_\ell},c'_1,\dots,c'_\ell,\overline{c'_1},\dots,\overline{c'_\ell},t_1,\dots,t_\ell,\overline{t_1},\dots,\overline{t_\ell}], R_2^{\mathbf{A}}) \ ,$$

where $R_2^{\mathbf{A}}$ consists of $R_2^{\mathbf{N}}$ and the following two groups of rules:

alternation rules for all states $s \in S$ such that $g(s) = \forall$:

$$forall \leftarrow \overline{init}, s \tag{5.49}$$
$$forall \leftarrow \overline{init}, conf[\mathbf{c},\mathbf{c}',\mathbf{t}^-].forall \tag{5.50}$$
$$forall \leftarrow \overline{init}, conf[\mathbf{c}^+,\mathbf{c}',\mathbf{t}].forall \tag{5.51}$$
$$forall \leftarrow \overline{init}, conf[\mathbf{c}^-,\mathbf{c}',\mathbf{t}].forall \tag{5.52}$$
$$forall \leftarrow \overline{init}, conf[\mathbf{c},\mathbf{c}'^+,\mathbf{t}].forall \tag{5.53}$$
$$forall \leftarrow \overline{init}, conf[\mathbf{c},\mathbf{c}'^-,\mathbf{t}].forall \tag{5.54}$$

saturation rules for all states $s \in S$ such that $g(s) = \forall$ and all $\sigma \in \Sigma$:

$$head \leftarrow \overline{init}, atm.accept, forall \tag{5.55}$$
$$s \leftarrow \overline{init}, atm.accept, forall \tag{5.56}$$
$$\sigma \leftarrow \overline{init}, atm.accept, forall \tag{5.57}$$

Take note that once we arrive at a universal state s at time point t we will derive *forall* with rule (5.49). In following time points, the alternation rules keep deriving *forall*. This is guaranteed by forcing *forall* to be true in every position of the grid (c,c') at time point t with rules (5.51)–(5.54), and then continuing to derive *forall* once it is true in a past time point $t-1$ at any position (c,c') with rule (5.50).

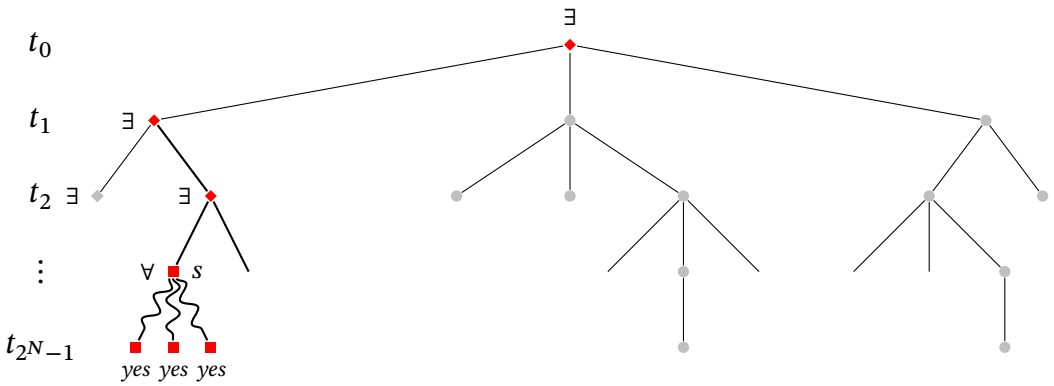


Figure 5.9: Alternating Turing machine computation tree

- the library module m_3 ($op[b_1, \ldots, b_\ell, \overline{b_1}, \ldots, \overline{b_\ell}]$) from $\mathbf{D}(T, N)$;
- the library module m_4 ($ord[x_1, \ldots, x_\ell, \overline{x_1}, \ldots, \overline{x_\ell}, y_1, \ldots, y_\ell, \overline{y_1}, \ldots, \overline{y_\ell}]$) from $\mathbf{D}(T, N)$;
- and the library module $m_5^{\mathbf{A}} = (branch[t_1, \ldots, t_\ell, \overline{t_1}, \ldots, \overline{t_\ell}], R_5^{\mathbf{A}})$, where $R_5^{\mathbf{A}}$ is the set of auxiliary rules (5.46) from $R_5^{\mathbf{N}}$ and the following two groups of rules:

branching rules for $(s, \sigma, s_i', \sigma_i', d_i) \in \delta$ such that $1 \leq i \leq k$:

$$b_{s,\sigma,1} \vee \cdots \vee b_{s,\sigma,k} \leftarrow \overline{final}, \text{not } bad \tag{5.58}$$

saturation rules for $s \in S$ such that $g(s) = \forall$ and $(s, \sigma, s_i', \sigma_i', d_i) \in \delta$ for $1 \leq i \leq k$:

$$b_{s,\sigma,i} \leftarrow \overline{final}, \text{not } bad, atm.accept, conf[\mathbf{t}, \mathbf{t}, \mathbf{t}].forall \tag{5.59}$$

Note that the module atom $conf[\mathbf{t}, \mathbf{t}, \mathbf{t}].forall$ in the saturation rules (5.59) uses $\mathbf{t}$ as input for $\mathbf{c}$ and $\mathbf{c}'$ for module $m_2^{\mathbf{A}}$. The alternation rules of $m_2^{\mathbf{A}}$ allow us to undifferentiatedly access atom *forall* in any time point t whose corresponding configuration is universal.

In analogy to nondeterministic Turing machines, alternating Turing machine computations form a computation tree as shown in Figure 5.9. Accepting runs start at time point t_0 in an existential state s_0 until they reach a universal configuration with state s from which only universal configurations can follow for ATMs that have only one alternation. Here, the difference of the acceptance condition to classical Turing machines come into play and we need to assure that all configurations of the subtree rooted at s are accepting, i.e., all configurations of the subtree lead to a leaf with state *yes*.

The ATM encoding $\mathbf{A}(T, N)$ is essentially our NTM encoding $\mathbf{N}(T, N)$ from Theorem 5.2, item 2, with the following modifications:

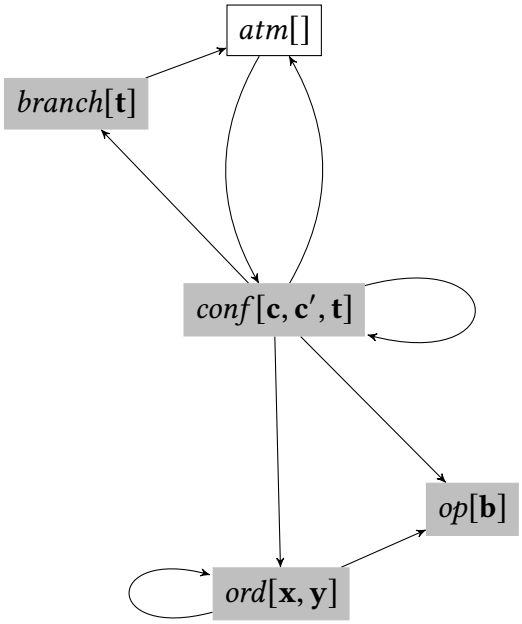


Figure 5.10: Module dependencies of an alternating Turing machine simulation

- the acceptance rule (5.48) from $m_1^{\mathbf{A}}$ additionally checks whether $conf[\mathbf{o}, \mathbf{o}, \mathbf{o}].forall$ is satisfied,
- $m_2^{\mathbf{A}}$ superinduces alternation and saturation rules,
- $m_5^{\mathbf{A}}$ uses disjunctive branching rules and adds saturation rules.

In $\mathbf{A}(T, N)$ we introduce the atom *forall* that will be true in all value calls $conf[C \cup C' \cup T_j]$ for $m_2^{\mathbf{A}}$ for any cell positions c and c' represented by C and C', respectively, and all time points t_j encoded by T_j such that for an $i > 0$, the machine T is in an existential configuration at t_{i-1} and in a universal configuration at t_i, and $t_j \geq t_i$.

Figure 5.10 shows the inter-module dependencies of the modules in $\mathbf{A}(T, N)$. The graph shows the main module $m_1^{\mathbf{A}}$ in white, while the library modules $m_2^{\mathbf{A}}$, m_3, m_4, $m_5^{\mathbf{A}}$ with input are shown in gray. The graph for $\mathbf{N}(T, N)$ in Figure 5.8 is almost identical to the graph for $\mathbf{A}(T, N)$: modules $m_1^{\mathbf{N}}$, $m_2^{\mathbf{N}}$, and $m_5^{\mathbf{N}}$ have been replaced by modules $m_1^{\mathbf{A}}$, $m_2^{\mathbf{A}}$, and $m_5^{\mathbf{A}}$, respectively, with the additional dependencies from $m_2^{\mathbf{A}}$ and $m_5^{\mathbf{A}}$ to $m_1^{\mathbf{A}}$. Hence, $\mathbf{A}(T, N)$ contains four cycles: both $m_2^{\mathbf{A}}$ and m_4 call themselves, $m_2^{\mathbf{A}}$ and the main module $m_1^{\mathbf{A}}$ are mutually recursive, and $m_5^{\mathbf{A}}$ is in a cycle with $m_1^{\mathbf{A}}$ and $m_2^{\mathbf{A}}$. As before, module m_3 is a sink module.

We will show now that we can simulate the computation of an alternating Turing machine T with bounded alternation from existential to universal on input I with

$\mathbf{A}(T,N)$ and prove that T accepts input I and halts in state *yes* in less than $N = 2^{m^k}$ steps if and only if $\mathbf{A}(T,N)$ has an answer set.

($\Rightarrow$) Suppose ATM T accepts input I within N steps and halts in state *yes*. There is a computation $\Gamma = \gamma_0, \ldots, \gamma_{N-1}$ such that for $i > 0$, $\Gamma_\exists = \gamma_0, \ldots, \gamma_{i-1}$ are existential configurations, $\Gamma_\forall = \gamma_i, \ldots, \gamma_{N-1}$ are universal configurations such that all possible successors γ' of γ_i are accepting, and the final configuration γ_{N-1} has state *yes*. We can build an interpretation $\mathbf{M^A}$ for $\mathbf{A}(T,N)$ based on $\Gamma = \Gamma_\exists\Gamma_\forall$ similar to the proof of Theorem 5.2, item 2 with $\mathbf{M^N}$ for $\mathbf{N}(T,N)$ and show, *mutatis mutandis*, that $\mathbf{M^A}$ is a model for $\mathbf{A}(T,N)$ and minimal model of $f\,\mathbf{A}(T,N)^{\mathbf{M^A}}$.

We obtain $\mathbf{M^A}$ from Γ as follows: set $M_1^\mathbf{A}/\varnothing$, $M_3^\mathbf{A}/B$, and $M_4^\mathbf{A}/(X \cup Y)$ to the same values as $M_1^\mathbf{N}/\varnothing$, $M_3^\mathbf{N}/B$, and $M_4^\mathbf{N}/(X \cup Y)$ from $\mathbf{M^N}$ in the proof of Theorem 5.2, item 2. Since $\mathbf{N}(T,N)$ is an encoding for ATMs that contain only existential configurations, we can reuse the existential part $\Gamma_\exists$ of Γ for $\mathbf{A}(T,N)$ and consider all γ_j from $\Gamma_\exists = \gamma_0, \ldots, \gamma_{i-1}$ and set

- $M_2^\mathbf{A}/C_{\gamma_j}^{c'} = M_2^\mathbf{N}/C_{\gamma_j}^{c'}$ and
- $M_5^\mathbf{A}/T_j = M_5^\mathbf{N}/T_j \cup \{b_{s,\sigma,1} \mid (s,\sigma,s_i',\sigma_i',d_i) \in \delta \text{ such that } g(s) = \forall\}$.

To complete the interpretation $\mathbf{M^A}$ we use the universal configurations $\Gamma_\forall$ from Γ as follows. We set for all γ_j from $\Gamma_\forall = \gamma_i, \ldots, \gamma_{N-1}$

- $M_2^\mathbf{A}/C_{\gamma_j}^{c'} = \{\mathit{forall}, \mathit{head}\} \cup \{s \mid s \in S \text{ such that } g(s) = \forall\} \cup \Sigma$,
- $M_5^\mathbf{A}/T_j = T_j \cup \{\overline{\mathit{final}}\} \cup \{b_{s,\sigma,1}, \ldots, b_{s,\sigma,k} \mid s \in S \text{ and } \sigma \in \Sigma \text{ such that } g(s) = \forall\} \cup \{b_{s,\sigma,1} \mid (s,\sigma,s_i',\sigma_i',d_i) \in \delta \text{ such that } g(s) = \exists\}$, for all T_j representing time point j such that $i \leq j < N-1$,
- $M_5^\mathbf{A}/T_{N-1} = T_{N-1}$, and
- $M_5^\mathbf{A}/S = S \cup \{\mathit{bad}\} \cup \{\overline{\mathit{final}} \mid \overline{t_j} \in S \text{ for } j \in \{1, \ldots, \ell\}\}$ for invalid bit representations S.

Now $\mathbf{M^A}$ is an interpretation for $\mathbf{A}(T,N)$. One can verify that the interpretation $\mathbf{M^A}$ is a model of $\mathbf{A}(T,N)$. We show now that it is also a minimal model of $f\,\mathbf{A}(T,N)^{\mathbf{M^A}}$.

Towards a contradiction, assume that there exists an interpretation $\mathbf{M}' < \mathbf{M^A}$ such that $\mathbf{M}'$ is a model of $f\,\mathbf{A}(T,N)^{\mathbf{M^A}}$. From the construction of $\mathbf{M^A}$ and the rules in $\mathbf{A}(T,N)$, we can only have that M_5'/T_i does not have an atom $b_{s,\sigma,j}$ which is in $M_5^\mathbf{A}/T_i$. But we must have a rule of form (5.58) in $f\,\mathbf{A}(T,N)^{\mathbf{M^A}}$, which contradicts the assumption that $\mathbf{M}'$ is a model. Thus, $\mathbf{M^A}$ is an answer set of $\mathbf{A}(T,N)$.

($\Leftarrow$) Suppose $\mathbf{M}^{\mathbf{A}}$ is an answer set of MLP $\mathbf{A}(T, N)$. We can extract an accepting computation $\Gamma = \gamma_0, \ldots, \gamma_{N-1}$ for the machine T from $\mathbf{M}^{\mathbf{A}}$ such that for $i > 0$, $\Gamma_\exists = \gamma_0, \ldots, \gamma_{i-1}$ are existential configurations, $\Gamma_\forall = \gamma_i, \ldots, \gamma_{N-1}$ are universal configurations such that all possible successors γ' of γ_i are accepting, and the final configuration γ_{N-1} has state *yes*. By rules (5.44) and (5.48), we must have that $accept \in M_1^{\mathbf{A}}/\emptyset$, hence there exists a maximal i such that $0 < i < N - 1$ and $\gamma_0, \ldots, \gamma_{i-1}$ are all existential configurations.

Set $\gamma_0 = (s_0, \sqcup, \sqcup)$, and let successor configurations γ_k for $k > 0$ be of the form (s_j, w, u). We can now extract existential successor configurations starting from γ_0 similar to item 2 of the proof for NTMs, with the distinction that we have disjunctive branching rules (5.58). Since we start in an existential phase the bodies of the saturation rules (5.59) are not satisfied as *forall* cannot be inferred from time point t_0 up to and including a time point t_{i-1}, where $0 < i < N - 1$. Once we can infer *forall* at time point t_i, the machine had performed an alternation, and we have arrived at the universal computation $\Gamma_\forall$. Thus, the answer set $\mathbf{M}^{\mathbf{A}}$ gives us first the desired existential configuration sequence $\Gamma_\exists = \gamma_0, \ldots, \gamma_{i-1}$. Then, starting from i, we can build $\Gamma_\forall = \gamma_i, \ldots, \gamma_{N-1}$ by following those universal transitions from δ that follow γ_i This works from the observation that our answer set $\mathbf{M}^{\mathbf{A}}$ is a minimal model and any nonaccepting universal configuration that follow our fixed $\Gamma_\exists$ would have killed $\mathbf{M}^{\mathbf{A}}$ by rule (5.44). This is not the case, and so all universal configurations are accepting in the computation $\Gamma_\forall$ generated by $\mathbf{M}^{\mathbf{A}}$. Since we have inferred all configurations by transitions from δ, this means that Γ must be accepting and that all configurations of T halt in state *yes* on input I, thus T accepts I within 2^{m^k} steps.

We can build the ATM encoding $\mathbf{A}(T, N)$ in polynomial time in the size of I, thus we have shown that deciding whether a given disjunctive propositional MLP $\mathbf{P}$ has an answer set is $\mathrm{NEXP}^{\mathrm{NP}}$-complete. □

5.4 Acyclic MLPs

In this section, we turn our attention to a restricted class of normal propositional MLPs with input and define acyclic MLPs. As the name suggests, this class of MLPs do not allow recursive module calls. Interestingly, acyclic MLPs are NEXP-complete, just like unrestricted normal propositional MLPs. As an example of modular logic programming, we show now how a domino problem can be encoded using a propositional MLP with negation, using several modules. First, we define acyclic propositional MLPs.

Definition 5.1 (Acyclic MLP).
We say that a propositional normal MLP $\mathbf{P}$ is *acyclic*, if the call graph $\mathrm{CG}_{\mathbf{P}}$ is acyclic.

Next, we define domino systems, tilings, and the exponential square tiling problem, which will be used in showing the hardness part for our complexity result. We follow the definitions from Grädel (1989) and Savelsbergh and van Emde Boas (1984).

Definition 5.2 (Domino systems).
A *domino system* is a triple $\mathcal{D} = (D, H, V)$ such that $D = \{d_1, \dots, d_k\}$ is a finite set whose elements are called *dominoes*, and $H, V \subseteq D \times D$ are binary relations. Let $R = \{0, \dots, 2^n - 1\}$ be a set of integers given in binary digits of length n, we say that $\mathcal{D}$ *tiles* $R \times R$, if there is a *tiling* $\tau \colon R \times R \to D$ such that, for each $i, j \in R$,

(T1) if $\tau(i, j) = d$ and $\tau(i + 1, j) = d'$, then $(d, d') \in H$; and

(T2) if $\tau(i, j) = d$ and $\tau(i, j + 1) = d'$, then $(d, d') \in V$.

The *exponential square tiling problem* consists of all pairs $(\mathcal{D}, R)$ such that $\mathcal{D}$ tiles $R \times R$.

Lewis (1978) has shown the following result.

Exponential Square Tiling (Lewis, 1978)The exponential square tiling problem is NEXP-complete.

Example 5.1 (Domino system with square tiling) Let $\mathcal{D} = (D, H, V)$ be a domino system with dominoes $D = \{\oplus, \ominus, \otimes, \oslash\}$, let

$$H = \{(\otimes, \oslash), (\oslash, \otimes), (\oplus, \ominus), (\ominus, \oplus)\}$$

and

$$V = \{(\otimes, \oplus), (\oplus, \oslash), (\oslash, \ominus), (\ominus, \otimes)\} \ ,$$

be binary relations, and let $R = \{0, \dots, 15\}$, i.e., R ranges from 0 to $2^n - 1$ for $n = 4$. The domino system $\mathcal{D}$ tiles $R \times R$ with the following tiling τ as a witness:

i is even:

$$\tau(i, j) = \begin{cases} \otimes & j \equiv 0 \mod 4 \\ \oplus & j \equiv 1 \mod 4 \\ \oslash & j \equiv 2 \mod 4 \\ \ominus & j \equiv 3 \mod 4 \end{cases}$$

i is odd:

$$\tau(i, j) = \begin{cases} \oslash & j \equiv 0 \mod 4 \\ \ominus & j \equiv 1 \mod 4 \\ \otimes & j \equiv 2 \mod 4 \\ \oplus & j \equiv 3 \mod 4 \end{cases}$$

The placement of the dominoes on $R \times R$ is shown in Figure 5.11.

We can now show the following.

Theorem 5.3 (Computational complexity of acyclic MLPs)
Given an acyclic MLP $\mathbf{P} = (P_1[\mathbf{q}_1], \dots, P_n[\mathbf{q}_n])$, to decide whether $\mathbf{P}$ has an answer set is NEXP-complete.

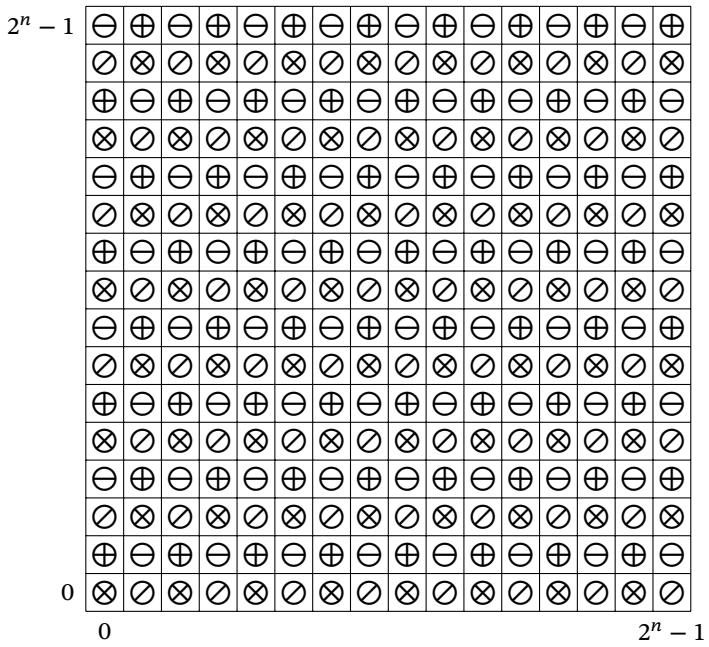


Figure 5.11: A domino system tiles $R \times R$

Proof Membership in NEXP follows from the results in Theorem 5.2, as propositional normal MLPs are NEXP-complete.

To show NEXP-hardness, we encode the tiling problem as acyclic MLP. Given a domino system $\mathcal{D} = (D, H, V)$ and $R = \{0, \dots, 2^n - 1\}$ as above, we can reduce the problem whether $\mathcal{D}$ tiles $R \times R$ to an acyclic MLP

$$\mathbf{T}(\mathcal{D}, R) = (m_0, m_1, \dots, m_{2n+1}, n_1, n_2, n_3, n_4)$$

as follows:

- The main module $m_0 = (main[], R_0)$ has one rule

$$R_0 = \{kill \leftarrow \text{not } gen_1.ok, \text{not } kill\} \ . \tag{5.60}$$

- To encode integers in the range R, we use lists $\mathbf{b} = b_1, \dots, b_n$ of propositional atoms b_i. The positions on the grid R^2 are then encoded by pairs $(\mathbf{x}, \mathbf{y})$ of such lists.

  The library modules $m_1, \dots, m_{2n+1}$ issue the checks for each position on the grid.

  For the range $i = 1, \dots, 2n$, this is done by cascading calls of modules

$$m_i = (gen_i[b_1, \dots, b_{i-1}], R_i),$$

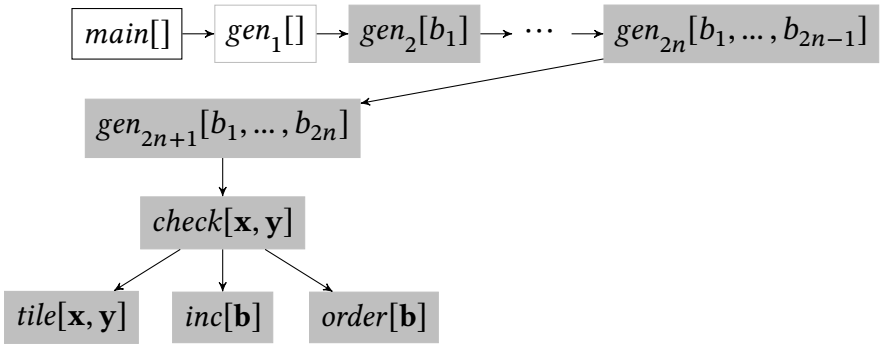


Figure 5.12: Module dependencies of a domino system encoding

where both values for the i^{th} bit are considered in the rules R_i:

$$b_i \leftarrow \tag{5.61}$$

$$\overline{b_i} \leftarrow \overline{b_i} \tag{5.62}$$

$$ok \leftarrow gen_{i+1}[b_1, \ldots, b_{i-1}, b_i].ok, gen_{i+1}[b_1, \ldots, b_{i-1}, \overline{b_i}].ok \tag{5.63}$$

Note that the dummy rule $\overline{b_i} \leftarrow \overline{b_i}$ lets $\overline{b_i}$ to be false in all minimal models. For $i = 2n + 1$, we then call $check[\mathbf{x}, \mathbf{y}]$ in module

$$m_{2n+1} = (gen_{2n+1}[b_1, \ldots, b_{2n}], R_{2n+1}),$$

where R_{2n+1} consists of the rule

$$ok \leftarrow check[b_1, \ldots, b_{2n}].ok \tag{5.64}$$

such that $\mathbf{x} = b_1, \ldots, b_n$ and $\mathbf{y} = b_{n+1}, \ldots, b_{2n}$.

- The library module $n_1 = (tile[\mathbf{x}, \mathbf{y}], Q_1)$ assigns a position $(\mathbf{x}, \mathbf{y})$ on the grid $R \times R$ arbitrarily a tile from D such that $k = |D|$:

$$Q_1 = \bigcup_{i \in \{1,\ldots,k\}} \{d_i \leftarrow \text{not } d_1, \ldots, \text{not } d_{i-1}, \text{not } d_{i+1}, \ldots, \text{not } d_k\} \ . \tag{5.65}$$

- On top of this, we have to check where the arbitrarily chosen tiling τ is legal, i.e., obeys the relations H and V. We do this using the library module

$$n_2 = (check[\mathbf{x}, \mathbf{y}], Q_2)$$

that receives as input a position $(\mathbf{x}, \mathbf{y})$ and has the rules Q_2:

$$ok \leftarrow \text{not } violated \tag{5.66}$$

$$yup_i \leftarrow \text{not } order[\mathbf{y}].last, inc[\mathbf{y}].b_i^+ \quad \text{for } i = 1, \dots, n \tag{5.67}$$

$$violated \leftarrow \text{not } order[\mathbf{y}].last, tile[\mathbf{x}, \mathbf{y}].d_i, tile[\mathbf{x}, \mathbf{yup}].d_j \quad \text{for } (d_i, d_j) \notin V \text{ such that } i, j \in \{1, \dots, |D|\} \tag{5.68}$$

$$xright_i \leftarrow \text{not } order[\mathbf{x}].last, inc[\mathbf{x}].b_i^+ \quad \text{for } i = 1, \dots, n \tag{5.69}$$

$$violated \leftarrow \text{not } order[\mathbf{x}].last, tile[\mathbf{x}, \mathbf{y}].d_i, tile[\mathbf{xright}, \mathbf{y}].d_j \quad \text{for } (d_i, d_j) \notin H \text{ such that } i, j \in \{1, \dots, |D|\} \tag{5.70}$$

- the module $n_3 = (inc[\mathbf{b}], Q_3)$ calculates the increment of an input number $\mathbf{b}$ by one in $\mathbf{b}^+ = b_1^+, \dots, b_n^+$ analog to the successor rules from module $m_3 = (op[\mathbf{b}], R_3)$ in the proof of Theorem 5.2, item 1, and
- module $n_4 = (order[\mathbf{b}], Q_4)$ tells in *last* whether the input number $\mathbf{b}$ is the maximum in R:

$$last \leftarrow b_1, \dots, b_n, \text{not } \overline{b_1}, \dots, \text{not } \overline{b_n} \tag{5.71}$$

Figure 5.12 shows the inter-module dependencies of the modules in the domino encoding $\mathbf{T}(\mathcal{D}, R) = (m_0, m_1, \dots, m_{2n+1}, n_1, n_2, n_3, n_4)$. The main module m_0 and library module without input m_1 is shown in white, and the library modules with input $m_2, \dots, m_{2n+1}$ and $n_1, \dots, n_4$ are gray. The structural dependencies show that $\mathbf{T}(\mathcal{D}, R)$ is acyclic: there is a chain of calls from m_0 to n_1, and starting from there n_1 calls three sink modules n_2, n_3, n_4 that do not call further modules.

The MLP $\mathbf{T}(\mathcal{D}, R)$ can be constructed in polynomial time from $\mathcal{D}$ and R, and the call graph $\mathrm{CG}_{\mathbf{T}(\mathcal{D},R)}$ is acyclic: rooted at m_1, the call graph is spanning a binary tree of value calls $gen_i[S] \to gen_{i+1}[S \cup \{b_i\}]$ and $gen_i[S] \to gen_{i+1}[S]$, whose leaves $gen_{2n+1}[S']$ have an edge for $X = S'|_{b_1,\dots,b_n}^{x_1,\dots,x_n}$ and $Y = S'|_{b_{n+1},\dots,b_{2n}}^{y_1,\dots,y_n}$ to the value call $check[X \cup Y]$. From $check[X \cup Y]$ only three value calls follow: $tile[X \cup Y]$, $inc[X \cup Y]$, $order[X]$, and $order[Y]$, all of which do not possess any outgoing edges in $\mathrm{CG}_{\mathbf{T}(\mathcal{D},R)}$. Therefore, $\mathbf{T}(\mathcal{D}, R)$ is an acyclic MLP. We show now that $\mathbf{T}(\mathcal{D}, R)$ has an answer set iff $\mathcal{D}$ tiles $R \times R$.

($\Rightarrow$) Suppose $\mathbf{T}(\mathcal{D}, R)$ has an answer set $\mathbf{M}$. We show now that $\mathbf{M}$ can be transformed into a tiling $\tau\colon R \times R \to D$ such that (T1) and (T2) hold. Let $M_{n_1}/(X \cup Y)$ be the part of $\mathbf{M}$ for module $n_1 = (tile[\mathbf{x}, \mathbf{y}], Q_1)$ such that X and Y contains only atoms from $\mathbf{x}$ and $\mathbf{y}$ being true in $M_{n_1}/(X \cup Y)$, respectively. Let $i, j \in R$ be integers that correspond to $\mathbf{x}$ and $\mathbf{y}$, then we set the tiling $\tau(i, j) = d_\ell$ whenever $d_\ell \in M_{n_1}/(X \cup Y)$.

Since module n_1 can only infer a single $d_\ell \in D$ for a particular pair $(\mathbf{x}, \mathbf{y})$, we get that τ is a function. From module m_0 we force the cascade of modules $m_1, \ldots, m_{2n+1}$ to have $ok \in M_{m_i}/S$ for $1 \leq i \leq 2n+1$, hence $violated \notin M_{m_i}/S$ for $i \leq 2n$. Moreover, we get that $ok \in M_{n_2}/(X \cup Y)$ from $ok \in M_{m_{2n+1}}/S$ such that $S = X \cup Y$. The lists of atoms **xright** and **yup** give us the successors for $\mathbf{x}$ and $\mathbf{y}$ in n_2, hence there is no pair $(d, d') \in H$ that violates (T1) and there is no pair $(d, d') \in V$ that violates (T2), which is enforced by the rules (5.70) and (5.68), respectively. Therefore, τ is indeed a tiling satisfying (T1) and (T2), and thus $\mathcal{D}$ tiles $R \times R$.

($\Leftarrow$) Let $\mathcal{D}$ tile $R \times R$, i.e., there exists a tiling $\tau\colon R \times R \to D$ such that (T1) and (T2) hold. We show now that τ can be transformed into an answer set $\mathbf{M}$ of $\mathbf{T}(\mathcal{D}, R)$. We set $\mathbf{M}$ as follows:

- $M_0/\emptyset = \emptyset$;
- for $1 \leq i \leq 2n$ and $gen_i[S] \in \mathrm{VC}(\mathbf{T}(\mathcal{D}, R))$, $M_i/S = \{ok, b_i\} \cup S$;
- $M_{2n+1}/S = \{ok\} \cup S$;
- for $i, j \in R$ such that $\tau(i, j) = d_\ell$, let $\mathbf{x}$ and $\mathbf{y}$ correspond to i and j, and let X and Y contain only those atoms from $\mathbf{x}$ and $\mathbf{y}$ that are true, we set $M_{n_1}/(X \cup Y)$ to $\{d_\ell\} \cup X \cup Y$;
- for $i, j \in R$, let $\mathbf{x}$ and $\mathbf{y}$ correspond to i and j, let X and Y contain only those atoms from $\mathbf{x}$ and $\mathbf{y}$ that are true, let **xright** and **yup** correspond to $i+1$ and $j+1$, and let X_{right} and Y_{up} contain only those atoms from **xright** and **yup** that are true, respectively, we set $M_{n_2}/(X \cup Y) = \{ok\} \cup X \cup Y \cup X_{right} \cup Y_{up}$; and
- we set M_{n_3}/B to the successor of the bits in B and M_{n_4}/B stores *last* whenever B is the maximum in R.

It is easy to see that $\mathbf{M}$ is a model for $\mathbf{T}(\mathcal{D}, R)$. As τ is a tiling, we have at exactly one $d \in D$ for all $i, j \in R$ such that $\tau(i, j) = d$, hence module instances of n_1 are satisfied.

Next, we show that $\mathbf{M}$ is a minimal model for $f\,\mathbf{T}(\mathcal{D}, R)^{\mathbf{M}}$. Towards a contradiction, assume to the contrary that there exists an interpretation $\mathbf{N} < \mathbf{M}$ of $\mathbf{T}(\mathcal{D}, R)$ such that $\mathbf{N} \vDash f\,\mathbf{T}(\mathcal{D}, R)^{\mathbf{M}}$. Since $M_0/\emptyset = \emptyset$, we must have ok and b_i in M_i/S for $1 \leq i \leq 2n$, and ok in M_{2n+1}/S. Therefore, M_{n_2}/S must contain ok. The successor rules for **xright** and **yup** in n_2 force that the atoms for the given $S = X \cup Y$ are present, and since τ is a tiling, n_1 requires that appropriate $d_i, d_j \in D$ are true in $M_{n_1}/(X \cup Y)$, $M_{n_1}/(X_{right} \cup Y)$, and $M_{n_1}/(X \cup Y_{up})$, respectively. Since $violated \notin M_{n_2}/S$, removing ok from any value call in $\mathbf{N}$ violates that $\mathbf{N} \vDash f\,\mathbf{T}(\mathcal{D}, R)^{\mathbf{M}}$. Hence, $\mathbf{N}$ is not a model of $f\,\mathbf{T}(\mathcal{D}, R)^{\mathbf{M}}$ as $\mathbf{N} < \mathbf{M}$, which contradicts our assumption. Therefore, $\mathbf{M}$ is a minimal model for $f\,\mathbf{T}(\mathcal{D}, R)^{\mathbf{M}}$ and so we proved that $\mathbf{M}$ is an answer set for $\mathbf{T}(\mathcal{D}, R)$.

We proved that deciding whether an acyclic MLP has an answer set is a NEXP-hard problem, using a polynomial reduction from the tiling problem, which is NEXP-complete by Exponential Square Tiling. Together with the NEXP upper bound we obtain that deciding acyclic MLP consistency is NEXP-complete. □

5.5 General MLPs

In this section, we first remove all syntactic restrictions and examine the computational complexity of MLPs in general case, i.e., we admit nonground Datalog rules, allow n-ary predicates as input, and cyclic module calls. Then, we restrict MLPs with bounded input predicate arities in §5.5.4.

In the Datalog setting, we get for MLPs a similar picture as for ordinary logic programs, where the complexity of Datalog programs is exponentially higher than the one of propositional programs. Intuitively, the process of grounding may introduce exponentially many ground instances of an atom, which in turn may result in double exponentially many module instances; thus, $I(\mathbf{P})$ and interpretations $\mathbf{M}$ have double exponential size in general. Computing $\mathrm{lfp}(\mathbf{P})$ for Horn MLPs $\mathbf{P}$ may thus take double exponential time, and a guess for an answer set has double exponential size. For MLPs with bounded input predicate arities we get that the complexity stays on the same level as the one of ordinary logic programs.

The hardness parts can be shown by lifting the Turing machine constructions for the propositional case from §5.3. Here, ℓ-ary predicates $q(X_1, \ldots, X_\ell)$ are used to store 2^ℓ bits of a number, such that a range of 2^{2^ℓ} tape cells and time stamps can be spanned via module inputs $\mathbf{q} = q, \overline{q}$.

We get the following results.

Theorem 5.4 (Computational complexity of general MLPs)
Given a nonground MLP $\mathbf{P} = (P_1[\mathbf{q}_1], \ldots, P_n[\mathbf{q}_n])$,

1. if $\mathbf{P}$ is Horn, the unique answer set $\mathbf{M} = \mathrm{lfp}(\mathbf{P})$ of $\mathbf{P}$ is computable in double exponential time and to decide whether $\alpha \in \mathbf{M}$ for a ground atom α is 2EXP-complete;

2. if $\mathbf{P}$ is normal, to decide whether $\mathbf{P}$ has an answer set is 2NEXP-complete; and

3. to decide whether $\mathbf{P}$ has an answer set is $2\mathrm{NEXP}^{\mathrm{NP}}$-complete.

The next sections will show this result.

5.5.1 Proof of Theorem 5.4, item 1

We first show membership in 2EXP. Since $|\mathrm{HB}_{\mathbf{P}}|$ is exponential in the length of $\mathbf{P}$, we have that every interpretation $\mathbf{M}$ of $\mathbf{P}$ consists of at most $n \cdot 2^{|\mathrm{HB}_{\mathbf{P}}|}$ components, we have that the least fixpoint of the $T_{\mathbf{P}}$ operator can be computed in double exponential time: if we exhaustively apply $T_{\mathbf{P}}(M_i/S)$, $1 \leq i \leq n$ and $S \subseteq \mathrm{HB}_{\mathbf{P}}|_{\mathbf{q}_i}$, we reach the fixpoint after at most $(m+1) \cdot n \cdot 2^{|\mathrm{HB}_{\mathbf{P}}|}$ application steps, where m is the number of ground rules in $gr(\mathbf{P})$ (which is exponential in the size of $\mathbf{P}$). Each application of $T_{\mathbf{P}}$ can thus done in exponential time. This shows that the unique answer set $\mathbf{M} = \mathrm{lfp}(\mathbf{P})$ can be computed in time double exponential in the size of $\mathbf{P}$.

We show now 2EXP-hardness. Given a deterministic Turing machine T which halts within $N = 2^{2^\ell}$ steps for an input I such that $m = |I|$ and $\ell = m^k$ for some constant k, we can simulate T by an MLP $\widehat{\mathbf{D}}(T,N) = (n_1^{\mathbf{D}}, n_2^{\mathbf{D}}, n_3, n_5, n_4)$ consisting of four modules. Similar to $\mathbf{D}(T,N)$ from §5.3.1, the main module $n_1^{\mathbf{D}}$ computes in *accept* the acceptance of I, while library module $n_2^{\mathbf{D}}$ encodes transition rules for δ and thus input I. The library module n_5 defines a linear order $\leq^i$ for $\{0,1\}^i$, which will be used by library module n_4 to define successor and predecessor rules for cell-time movements.

To address $N = 2^{2^\ell}$ steps, we make use of ℓ-ary predicates in the input of modules and encode the bits of a nonnegative number $n \in \{0, \ldots, 2^{2^\ell} - 1\}$ as a sequence of predicates $b, \overline{b}$. In $\mathbf{D}(T,N)$ we use the atoms b_i and $\overline{b_i}$ to encode whether the i^{th} bit is true or false for a nonnegative integer $n \in \{0, \ldots, 2^\ell - 1\}$. In $\widehat{\mathbf{D}}(T,N)$ we lift index i from atoms b_i and $\overline{b_i}$ and encode $i \in \{0, \ldots, 2^\ell - 1\}$ in binary in the arguments of $b(X_1, \ldots, X_\ell)$ and $\overline{b}(X_1, \ldots, X_\ell)$: if the j^{th} bit of i is 1, we set $X_j = 1$ in $b(X_1, \ldots, X_\ell)$, otherwise if the j^{th} bit of i is 0, we set $X_j = 0$ in $\overline{b}(X_1, \ldots, X_\ell)$. Now, for a fixed bit sequence $\mathbf{t}$ of length ℓ, $b(t_1, \ldots, t_\ell)$ and $\overline{b}(t_1, \ldots, t_\ell)$ must have complementary truth values in a model. This way we can represent 2^{2^ℓ} cell positions and time points.

We set up the modules of $\widehat{\mathbf{D}}(T,N) = (n_1^{\mathbf{D}}, n_2^{\mathbf{D}}, n_3, n_4, n_5)$ up as follows. We have

- a main module $n_1^{\mathbf{D}} = (dtm[\,], Q_1^{\mathbf{D}})$, where $Q_1^{\mathbf{D}}$ is the set of rules

$$o(\mathbf{X}) \leftarrow succ.first^\ell(\mathbf{X}) \tag{5.72}$$

$$o(\mathbf{Y}) \leftarrow o(\mathbf{X}), succ.succ^\ell(\mathbf{X}, \mathbf{Y}) \tag{5.73}$$

$$accept \leftarrow conf[o, \overline{o}, o, \overline{o}, o, \overline{o}].yes \tag{5.74}$$

Note that $Q_1^{\mathbf{D}}$ is essentially $R_1^{\mathbf{D}}$ with ℓ-ary predicates: the rules (5.1) are replaced by rules (5.72)–(5.73) and rule (5.2) replaced by (5.74). For a number $N = 2^{2^\ell}$ and a bit sequence $\mathbf{t}$ of length ℓ such that $\mathbf{t}$ encodes a nonnegative integer i in binary, $o(\mathbf{t})$ stands

for the i^{th} bit being 1 in nonnegative integer $n \in \{0, \dots, N-1\}$ encoded in binary, thus input $(o, \overline{o}, o, \overline{o}, o, \overline{o})$ represents integers $(N-1, N-1, N-1)$.

- a library module $n_2^{\mathbf{D}} = (conf[c, \overline{c}, c', \overline{c'}, t, \overline{t}], Q_2^{\mathbf{D}})$, where $Q_2^{\mathbf{D}}$ consists of the rules from the propositional encoding in $R_2^{\mathbf{D}}$ with minor modifications: we drop the auxiliary rules from $R_2^{\mathbf{D}}$ and keep
  - the offset rules (5.3)–(5.5)
  - the initial rules (5.8),
  - the transition rules (5.9)–(5.24), and
  - the inertia rules (5.25)–(5.35),

  and apply the following customization to all rules: we drop all indices i from the atoms $b_i^+, b_i^-, \overline{b_i^+}, \overline{b_i^-}, t_i^-, c_i^-, c_i^+, c_i'^-, c_i'^+$ and then use them as ℓ-place predicates b^+, b^-, $\overline{b^+}$, $\overline{b^-}$, t^-, c^-, c^+, c'^-, c'^+ with the variables $\mathbf{X} = X_1, \dots, X_\ell$ as parameters. Furthermore, instead of the auxiliary rules of $R_2^{\mathbf{D}}$, the set $Q_2^{\mathbf{D}}$ contains the

auxiliary rules:

$$\widehat{init}(\mathbf{X}) \leftarrow \overline{t}(\mathbf{X}), succ.first^{\ell}(\mathbf{X}) \tag{5.75}$$

$$\widehat{init}(\mathbf{Y}) \leftarrow \widehat{init}(\mathbf{X}), \overline{t}(\mathbf{Y}), succ.succ^{\ell}(\mathbf{X}, \mathbf{Y}) \tag{5.76}$$

$$\widehat{start}(\mathbf{X}) \leftarrow \overline{c}(\mathbf{X}), succ.first^{\ell}(\mathbf{X}) \tag{5.77}$$

$$\widehat{start}(\mathbf{Y}) \leftarrow \widehat{start}(\mathbf{X}), \overline{c}(\mathbf{Y}), succ.succ^{\ell}(\mathbf{X}, \mathbf{Y}) \tag{5.78}$$

$$\overline{init} \leftarrow t(\mathbf{X}) \tag{5.79}$$

$$init \leftarrow \widehat{init}(\mathbf{X}), succ.last^{\ell}(\mathbf{X}) \tag{5.80}$$

$$start \leftarrow \widehat{start}(\mathbf{X}), succ.last^{\ell}(\mathbf{X}) \tag{5.81}$$

Intuitively, $\widehat{init}$ and $\widehat{start}$ compute all successor bits of the least significant bit for the initial time point and cell position 0, respectively. Thus, *init* and *start* is true whenever we have all possible $\overline{t}(\mathbf{t})$ and $\overline{c}(\mathbf{t})$ true, respectively.

To show the modifications for the offset rules in an exemplary way, the rules for $c_i'^-$ and $c_i'^+$ in (5.5) from $m_2^{\mathbf{D}}$ is in $n_2^{\mathbf{D}}$ replaced by the rules

$$c'^-(\mathbf{X}) \leftarrow op[\mathbf{c}'].b^+(\mathbf{X}) \qquad c'^+(\mathbf{X}) \leftarrow op[\mathbf{c}'].b^-(\mathbf{X})$$

$$\overline{c'^-}(\mathbf{X}) \leftarrow op[\mathbf{c}'].\overline{b^+}(\mathbf{X}) \qquad \overline{c'^+}(\mathbf{X}) \leftarrow op[\mathbf{c}'].\overline{b^-}(\mathbf{X})$$

- a library module $n_3 = (succ[], Q_3)$, where Q_3 consists of the following groups of rules.

initial index successor rules:

$$succ^1(0,1) \leftarrow \qquad first^1(0) \leftarrow \qquad last^1(1) \leftarrow \tag{5.82}$$

$$val(0) \leftarrow \qquad val(1) \leftarrow \tag{5.83}$$

index successor rules for $1 \leq i < \ell$ ($\mathbf{X}_i = X_1, \ldots, X_i$ and $\mathbf{Y}_i = Y_1, \ldots, Y_i$):

$$succ^{i+1}(Z, \mathbf{X}_i, Z, \mathbf{Y}_i) \leftarrow succ^i(\mathbf{X}_i, \mathbf{Y}_i), val(Z) \tag{5.84}$$

$$succ^{i+1}(Z, \mathbf{X}_i, Z', \mathbf{Y}_i) \leftarrow succ^1(Z, Z'), last^i(\mathbf{X}_i), first^i(\mathbf{Y}_i) \tag{5.85}$$

$$first^{i+1}(Z, \mathbf{X}_i) \leftarrow first^1(Z), first^i(\mathbf{X}_i) \tag{5.86}$$

$$last^{i+1}(Z, \mathbf{X}_i) \leftarrow last^1(Z), last^i(\mathbf{X}_i) \tag{5.87}$$

- a library module $n_4 = (op[b, \overline{b}], Q_4^{\mathrm{D}})$, where Q_4^{D} consists of the following groups of rules.

successor rules ($\mathbf{X} = X_1, \ldots, X_\ell$ and $\mathbf{Y} = Y_1, \ldots, Y_\ell$):

$$inv(\mathbf{X}) \leftarrow succ.first^\ell(\mathbf{X}) \tag{5.88}$$

$$\overline{inv}(\mathbf{Y}) \leftarrow \overline{inv}(\mathbf{X}), succ.succ^\ell(\mathbf{X}, \mathbf{Y}) \tag{5.89}$$

$$\overline{inv}(\mathbf{Y}) \leftarrow inv(\mathbf{X}), \overline{b}(\mathbf{X}), succ.succ^\ell(\mathbf{X}, \mathbf{Y}) \tag{5.90}$$

$$inv(\mathbf{Y}) \leftarrow inv(\mathbf{X}), b(\mathbf{X}), succ.succ^\ell(\mathbf{X}, \mathbf{Y}) \tag{5.91}$$

$$\overline{b^+}(\mathbf{X}) \leftarrow b(\mathbf{X}), inv(\mathbf{X}) \tag{5.92}$$

$$b^+(\mathbf{X}) \leftarrow b(\mathbf{X}), \overline{inv}(\mathbf{X}) \tag{5.93}$$

$$b^+(\mathbf{X}) \leftarrow \overline{b}(\mathbf{X}), inv(\mathbf{X}) \tag{5.94}$$

$$\overline{b^+}(\mathbf{X}) \leftarrow \overline{b}(\mathbf{X}), \overline{inv}(\mathbf{X}) \tag{5.95}$$

predecessor rules ($\mathbf{X} = X_1, \dots, X_\ell$ and $\mathbf{Y} = Y_1, \dots, Y_\ell$):

$$b^-(\mathbf{X}) \leftarrow \overline{b}(\mathbf{X}), succ.first^\ell(\mathbf{X}) \tag{5.96}$$

$$\overline{b^-}(\mathbf{X}) \leftarrow b(\mathbf{X}), succ.first^\ell(\mathbf{X}) \tag{5.97}$$

$$c(\mathbf{X}) \leftarrow b(\mathbf{X}), succ.first^\ell(\mathbf{X}) \tag{5.98}$$

$$\overline{c}(\mathbf{X}) \leftarrow \overline{b}(\mathbf{X}), succ.first^\ell(\mathbf{X}) \tag{5.99}$$

$$\overline{b^-}(\mathbf{Y}) \leftarrow b(\mathbf{Y}), \overline{c}(\mathbf{X}), succ.succ^\ell(\mathbf{X}, \mathbf{Y}) \tag{5.100}$$

$$c(\mathbf{Y}) \leftarrow b(\mathbf{Y}), \overline{c}(\mathbf{X}), succ.succ^\ell(\mathbf{X}, \mathbf{Y}) \tag{5.101}$$

$$b^-(\mathbf{Y}) \leftarrow b(\mathbf{Y}), c(\mathbf{X}), succ.succ^\ell(\mathbf{X}, \mathbf{Y}) \tag{5.102}$$

$$c(\mathbf{Y}) \leftarrow b(\mathbf{Y}), c(\mathbf{X}), succ.succ^\ell(\mathbf{X}, \mathbf{Y}) \tag{5.103}$$

$$b^-(\mathbf{Y}) \leftarrow \overline{b}(\mathbf{Y}), \overline{c}(\mathbf{X}), succ.succ^\ell(\mathbf{X}, \mathbf{Y}) \tag{5.104}$$

$$\overline{c}(\mathbf{Y}) \leftarrow \overline{b}(\mathbf{Y}), \overline{c}(\mathbf{X}), succ.succ^\ell(\mathbf{X}, \mathbf{Y}) \tag{5.105}$$

$$\overline{b^-}(\mathbf{Y}) \leftarrow \overline{b}(\mathbf{Y}), c(\mathbf{X}), succ.succ^\ell(\mathbf{X}, \mathbf{Y}) \tag{5.106}$$

$$c(\mathbf{Y}) \leftarrow \overline{b}(\mathbf{Y}), c(\mathbf{X}), succ.succ^\ell(\mathbf{X}, \mathbf{Y}) \tag{5.107}$$

Note that n_4 lifts m_3 from $\mathbf{D}(T, N)$ by moving indexes i into predicate arguments, thus we can compute successors and predecessors of integers encoded by $b, \overline{b}$ in the range $\{0, \dots, N-1\}$.

- and a library module $n_5 = (ord[x, \overline{x}, y, \overline{y}], Q_5)$, where Q_5 consists of the following groups of rules.

inequality rules ($\mathbf{Z} = Z_1, \dots, Z_\ell$):

$$\neq \leftarrow x(\mathbf{Z}), \overline{y}(\mathbf{Z}) \qquad\qquad \neq \leftarrow \overline{x}(\mathbf{Z}), y(\mathbf{Z}) \tag{5.108}$$

equality rules ($\mathbf{Z} = Z_1, \dots, Z_\ell$ and $\mathbf{Z}' = Z'_1, \dots, Z'_\ell$):

$$\widehat{=}(\mathbf{Z}) \leftarrow x(\mathbf{Z}), y(\mathbf{Z}), succ.first^\ell(\mathbf{Z}) \tag{5.109}$$

$$\widehat{=}(\mathbf{Z}) \leftarrow \widehat{=}(\mathbf{Z}'), succ.succ^\ell(\mathbf{Z}', \mathbf{Z}), x(\mathbf{Z}), y(\mathbf{Z}) \tag{5.110}$$

$$\widehat{=}(\mathbf{Z}) \leftarrow \overline{x}(\mathbf{Z}), \overline{y}(\mathbf{Z}), succ.first^\ell(\mathbf{Z}) \tag{5.111}$$

$$\widehat{=}(\mathbf{Z}) \leftarrow \widehat{=}(\mathbf{Z}'), succ.succ^\ell(\mathbf{Z}', \mathbf{Z}), \overline{x}(\mathbf{Z}), \overline{y}(\mathbf{Z}) \tag{5.112}$$

$$= \leftarrow \widehat{=}(\mathbf{Z}), succ.last^\ell(\mathbf{Z}) \tag{5.113}$$

successor rules ($\mathbf{X} = X_1, \ldots, X_\ell$):

$$z(\mathbf{X}) \leftarrow op[x, \overline{x}].b^+(\mathbf{X}) \tag{5.114}$$

$$\overline{z}(\mathbf{X}) \leftarrow op[x, \overline{x}].\overline{b^+}(\mathbf{X}) \tag{5.115}$$

order rules:

$$\leq \leftarrow = \tag{5.116}$$

$$\leq \leftarrow ord[z, \overline{z}, y, \overline{y}].\leq \tag{5.117}$$

Note that n_5 lifts m_4 from $\mathbf{D}(T, N)$ by moving index i into predicate arguments. Thus, we can order pairs of integers in the range $\{0, \ldots, N-1\}$ encoded by inputs $x, \overline{x}$ and $y, \overline{y}$.

Figure 5.13 shows the inter-module dependencies of the modules in $\widehat{\mathbf{D}}(T, N) = (n_1^{\mathrm{D}}, n_2^{\mathrm{D}}, n_3, n_4, n_5)$. Compared to the dependencies for $\mathbf{D}(T, N)$ in Figure 5.6, $\widehat{\mathbf{D}}(T, N)$ has an additional sink library module n_3 for computing the successor relation and which is used by all library modules $n_2^{\mathrm{D}}, n_4, n_5$. Just like $\mathbf{D}(T, N)$, $\widehat{\mathbf{D}}(T, N)$ has cyclic dependencies: n_2^{D} and n_5 call themselves.

We show now that we can simulate the computation of a deterministic Turing machine T on input I with $\widehat{\mathbf{D}}(T, N)$ and prove that T accepts input I within $N = 2^{2^{m^k}}$ steps if and only if *accept* $\in \mathrm{lfp}(\widehat{\mathbf{D}}(T, N))$. The argument that $\widehat{\mathbf{D}}(T, N)$ gives us the desired outcome is the essentially the same as the argument in §5.3.1 (Theorem 5.2, item 1) with $\mathbf{D}(T, N)$. We have changed the time step and cell position addressing in our encoding $\widehat{\mathbf{D}}(T, N)$ by introducing predicate arguments that play the role of the indexes in $\mathbf{D}(T, N)$. To this end we have introduced module n_3, which takes care of computing a successor relation for all pairs of integers (n, n') such that $n, n' \in \{0, \ldots, 2^\ell\}$. The transition rules are essentially identical, with the difference that module input take ℓ-place predicates for storing cell positions and time points.

From a given DTM T and bound N we can build $\widehat{\mathbf{D}}(T, N)$ in polynomial time in the size of I, therefore deciding whether $\alpha \in \mathrm{lfp}(\mathbf{P})$ for a nonground Horn MLP $\mathbf{P}$ is 2EXP-complete. □

5.5.2 Proof of Theorem 5.4, item 2

Showing membership in 2NEXP is similar to the proof of Theorem 5.2, item 2, from §5.3.2. An algorithm that checks whether a nonground normal $\mathbf{P}$ has an answer set

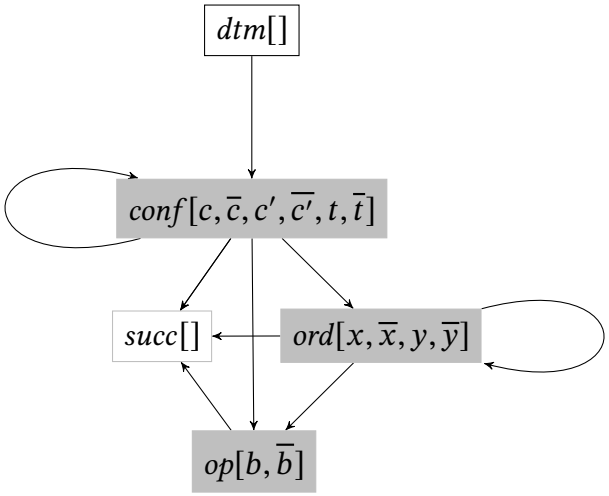


Figure 5.13: Module dependencies of a deterministic Turing machine simulation

starts by guessing an interpretation $\mathbf{M}$ for $\mathbf{P}$. Every interpretation $\mathbf{M}$ of $\mathbf{P}$ uses at most $n \cdot 2^{2^{|\mathrm{HB}_\mathbf{P}|}}$ value calls, thus checking that all rules of $I(\mathbf{P})$ are satisfied and whether $\mathbf{M}$ is a minimal model for $f\,\mathbf{P}^\mathbf{M}$ takes double-exponentially many steps.

Hardness can be shown by adapting the Turing machine encoding of item 1 from §5.5.1. We setup the modules of $\widehat{\mathbf{N}}(T, N) = (n_1^\mathbf{N}, n_2^\mathbf{N}, n_3, n_4, n_5, n_6^\mathbf{N})$ as follows. We have

- a main module $n_1^\mathbf{N} = (ntm[], Q_1^\mathbf{N})$, where $Q_1^\mathbf{N}$ is the set of rules $Q_1^\mathbf{D}$ with the additional rule (5.44) from the propositional module $m_1^\mathbf{N}$;
- a library module $n_2^\mathbf{N} = (conf[c, \overline{c}, c', \overline{c'}t, \overline{t}], Q_2^\mathbf{N})$, where $Q_2^\mathbf{N}$ consists of the initial rules, transition rules, inertia rules from $Q_2^\mathbf{D}$ with the following modifications for the transition and inertia rules: (5.9)–(5.35) get for a transition $(s, \sigma, s_i', \sigma_i', d_i) \in \delta$ the additional body atom

$$branch[t^-, \overline{t^-}].b_{s,\sigma,i} \tag{5.118}$$

As an example, the rule (5.10) from $n_2^\mathbf{D}$ is in $n_2^\mathbf{N}$ replaced by the rules

$$\begin{aligned} s_1' \leftarrow\ & \overline{init}, conf[c^-, \overline{c^-}, c^-, \overline{c^-}, t^-, \overline{t^-}].s, conf[c^-, \overline{c^-}, c^-, \overline{c^-}, t^-, \overline{t^-}].\sigma, \\ & conf[c^-, \overline{c^-}, c^-, \overline{c^-}, t^-, \overline{t^-}].head, branch[t^-, \overline{t^-}].b_{s,\sigma,1} \\ \vdots\ & \\ s_j' \leftarrow\ & \overline{init}, conf[c^-, \overline{c^-}, c^-, \overline{c^-}, t^-, \overline{t^-}].s, conf[c^-, \overline{c^-}, c^-, \overline{c^-}, t^-, \overline{t^-}].\sigma, \\ & conf[c^-, \overline{c^-}, c^-, \overline{c^-}, t^-, \overline{t^-}].head, branch[t^-, \overline{t^-}].b_{s,\sigma,j} \end{aligned}$$

for all $+1$-transitions $(s, \sigma, s_1', \sigma_1', +1), \ldots, (s, \sigma, s_j', \sigma_j', +1) \in \delta$ such that $1 \leq j \leq k$, where k is the number of all (s, σ)-transitions of form $(s, \sigma, s_i', \sigma_i', d_i) \in \delta$.

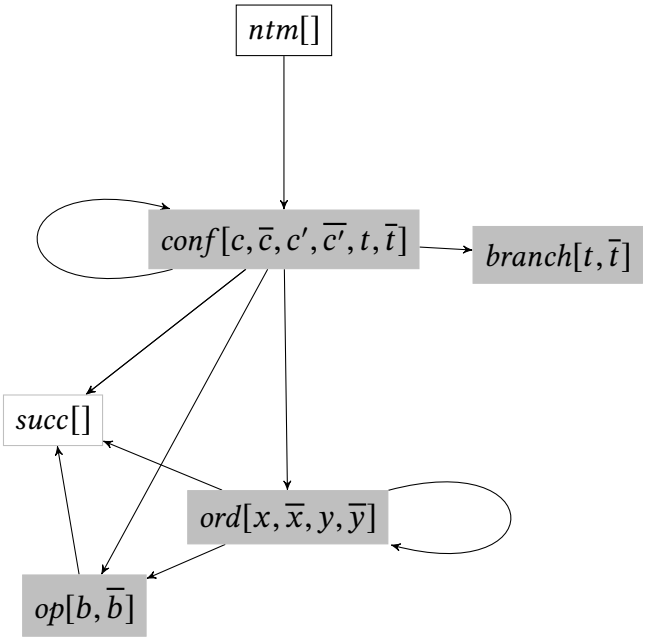


Figure 5.14: Module dependencies of a nondeterministic Turing machine simulation

- the library module n_3 ($succ[]$) from $\widehat{\mathbf{D}}(T, N)$;
- the library module n_4 ($op[b, \overline{b}]$) from $\widehat{\mathbf{D}}(T, N)$;
- the library module n_5 ($ord[x, \overline{x}, y, \overline{y}]$) from $\widehat{\mathbf{D}}(T, N)$;
- and the library module $n_6^{\mathbf{N}} = (branch[t, \overline{t}], Q_5^{\mathbf{N}})$, where $Q_5^{\mathbf{N}}$ are the branching rules from $R_4^{\mathbf{N}}$ such that all module input lists replace $\mathbf{t} = t_1, \ldots, t_\ell, \overline{t_1}, \ldots, \overline{t_\ell}$ with ℓ-ary input predicates $t, \overline{t}$. Furthermore, $Q_5^{\mathbf{N}}$ has

  auxiliary rules ($\mathbf{X} = X_1, \ldots, X_\ell$):

$$\overline{final} \leftarrow t(\mathbf{X}) \qquad bad \leftarrow t(\mathbf{X}), \overline{t}(\mathbf{X}) \qquad bad \leftarrow \text{not}\, t(\mathbf{X}), \text{not}\, \overline{t}(\mathbf{X}) \tag{5.119}$$

The inter-module dependencies for $\widehat{\mathbf{N}}(T, N) = (n_1^{\mathbf{N}}, n_2^{\mathbf{N}}, n_3, n_4, n_5, n_6^{\mathbf{N}})$ are shown in Figure 5.14. Compared to the dependencies for $\mathbf{N}(T, N)$ in Figure 5.8, $\widehat{\mathbf{N}}(T, N)$ has an additional sink library module n_3, and since $\widehat{\mathbf{N}}(T, N)$ adapts $\widehat{\mathbf{D}}(T, N)$ for NTMs, it contains the branching module $n_6^{\mathbf{N}}$ just like $\mathbf{N}(T, N)$ contains $m_5^{\mathbf{N}}$; the two self-cycles related with $n_2^{\mathbf{N}}$ and n_5 remain intact.

We show now that we can simulate the computation of a nondeterministic Turing machine T on input I with $\widehat{\mathbf{N}}(T, N)$ and claim that T accepts input I and halts in state *yes* within $N = 2^{2^{m^k}}$ steps if and only if $\widehat{\mathbf{N}}(T, N)$ has an answer set. The argument

follows the one for $\mathbf{N}(T,N)$ in §5.3.2 (Theorem 5.2, item 2), as only minor modifications to the input predicates have to be taken into account for addressing at most $2^{2^{m^k}}$ time points and cell positions.

The algorithm outlined above for generating $\widehat{\mathbf{N}}(T,N)$ is a polynomial reduction from Turing machine T on input I, thus we have shown that deciding whether a normal nonground MLP $\mathbf{P}$ has an answer set is 2NEXP-complete. □

5.5.3 Proof of Theorem 5.4, item 3

We show membership in 2NEXP^{NP} akin to the proof of Theorem 5.2, item 3, from §5.3.3. An algorithm for answer set checking given a nonground disjunctive $\mathbf{P}$ works as follows. First we guess an interpretation $\mathbf{M}$ for $\mathbf{P}$. Since $\mathbf{M}$ uses at most $n \cdot 2^{2^{|\text{HB}_\mathbf{P}|}}$ value calls, we verify that all rules from $I(\mathbf{P})$ are satisfied and compute $f\,\mathbf{P}^\mathbf{M}$ in time double-exponential in the size of $\mathbf{P}$. The check whether whether $\mathbf{M}$ is a minimal model for $f\,\mathbf{P}^\mathbf{M}$ uses an oracle for co-NP, thus we reach a 2NEXP^{NP} upper bound.

Hardness can be shown by adapting the Turing machine encoding of item 2 from §5.5.2. We setup the modules of $\widehat{\mathbf{A}}(T,N) = (n_1^\mathbf{A}, n_2^\mathbf{A}, n_3, n_4, n_5, n_6^\mathbf{A})$ as follows. We have

- a main module $n_1^\mathbf{A} = (atm[], Q_1^\mathbf{A})$, where $Q_1^\mathbf{A}$ is the set of rules $R_1^\mathbf{A}$ with the rules (5.1) replaced by (5.72)–(5.73).
- a library module $n_2^\mathbf{A} = (conf[c,\overline{c},t,\overline{t}], Q_2^\mathbf{A})$, where $Q_2^\mathbf{A}$ consists of the initial rules, transition rules, inertia rules, saturation rules from $R_2^\mathbf{A}$, and the offset and auxiliary rules from $Q_2^\mathbf{D}$. For the rules from $R_2^\mathbf{A}$, we apply the usual index modification to all rules: we drop all indices i from the atoms $b_i^+, b_i^-, \overline{b_i^+}, \overline{b_i^-}, t_i^-, c_i^-, c_i^+, c_i'^-, c_i'^+$ and then use ℓ-place predicates b^+, b^-, $\overline{b^+}$, $\overline{b^-}$, t^-, c^-, c^+, c'^-, c'^+ with the variables $\mathbf{X} = X_1, \ldots, X_\ell$ as parameters.
- the library module n_3 ($succ[]$) from $\widehat{\mathbf{D}}(T,N)$;
- the library module n_4 ($op[b,\overline{b}]$) from $\widehat{\mathbf{D}}(T,N)$;
- the library module n_5 ($ord[x,\overline{x},y,\overline{y}]$) from $\widehat{\mathbf{D}}(T,N)$;
- and the library module $n_6^\mathbf{A} = (branch[t,\overline{t}], Q_5^\mathbf{A})$, where $Q_5^\mathbf{A}$ are the branching rules and saturation rules from $R_4^\mathbf{A}$ and the auxiliary rules from $Q_5^\mathbf{N}$. In all module atoms, we replace input $\mathbf{t} = t_1, \ldots, t_\ell, \overline{t_1}, \ldots, \overline{t_\ell}$ with ℓ-ary input predicates $t, \overline{t}$.

The MLP $\widehat{\mathbf{A}}(T,N) = (n_1^\mathbf{A}, n_2^\mathbf{A}, n_3, n_4, n_5, n_6^\mathbf{A})$ is the most complex one in this chapter. Its dependencies are shown in Figure 5.15, which shows the close relationship to $\mathbf{A}(T,N)$ in Figure 5.10: $\widehat{\mathbf{A}}(T,N)$ has an additional sink library module n_3 like all nonground MLP encodings before, and since $\widehat{\mathbf{A}}(T,N)$ adapts $\widehat{\mathbf{N}}(T,N)$ for ATMs, it contains

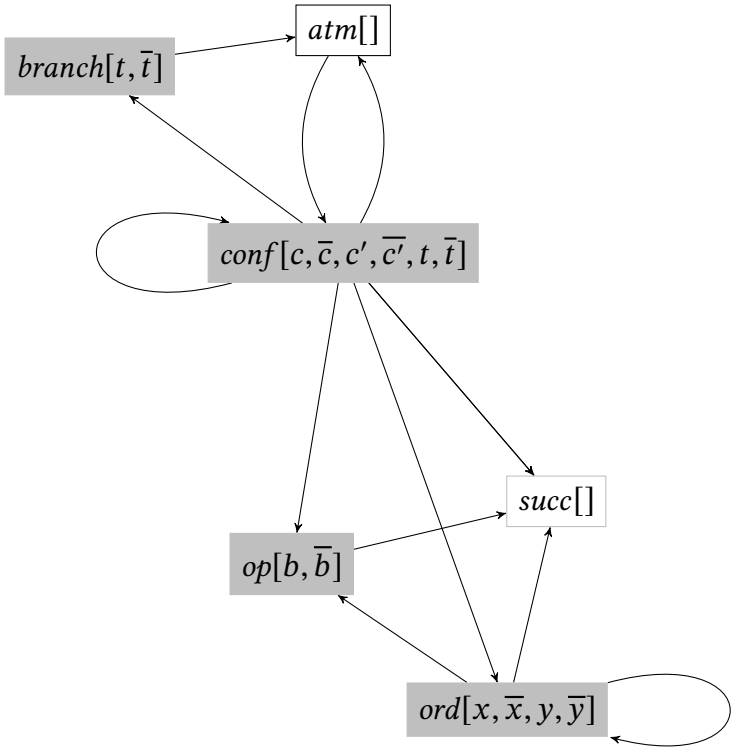


Figure 5.15: Module dependencies of an alternating Turing machine simulation

the branching module $n_6^{\mathbf{A}}$ which calls $n_1^{\mathbf{A}}$, thereby creating an additional cycle in the dependencies. The module $n_2^{\mathbf{A}}$ adds another cycle through $n_1^{\mathbf{A}}$. The self-loops $n_2^{\mathbf{A}}$ and n_5 are present, too.

We will show now that we can simulate the computation of an alternating Turing machine T with bounded alternation from existential to universal on input I with $\widehat{\mathbf{A}}(T, N)$ and prove that T accepts input I and halts in state *yes* in less than $N = 2^{2^{m^k}}$ steps if and only if $\widehat{\mathbf{A}}(T, N)$ has an answer set. The argument follows the one for $\mathbf{A}(T, N)$ in §5.3.3 (Theorem 5.2, item 3), with the minor modifications to the ℓ-place input predicates for addressing at most $2^{2^{m^k}}$ time points and cell positions.

The ATM encoding $\widehat{\mathbf{A}}(T, N)$ can be built from T and N in polynomial time in the size of I, which shows that deciding whether a disjunctive nonground MLP $\mathbf{P}$ has an answer set is $2\mathrm{NEXP}^{\mathrm{NP}}$-complete. □

5.5.4 Complexity of MLPs with bounded predicate arities

Finally, we note that the complexity drops by an exponential to the one of ordinary logic programs if the arities of input predicates are bounded by a constant, as then

$I(\mathbf{P})$ and $\mathbf{M}$ have single exponential size. Using the results in this chapter, the next statement then follows immediately.

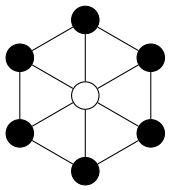
Corollary 5.5 (Complexity of general MLPs with bounded predicate arities) Given a nonground MLP $\mathbf{P} = (P_1[\mathbf{q}_1], \ldots, P_n[\mathbf{q}_n])$ whose predicate arities of $\mathbf{q}_i$ for $i = 1, \ldots, n$ are bounded by a constant,

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

Compared to answer set programs with bounded predicate arities (Eiter et al., 2007a), which shows a similar drop in complexity, we are an exponential higher in complexity. This is of no surprise, as modules with input have to be taken into account. Techniques based on (Eiter et al., 2010) may prove to be useful for implementing a reasoner for MLPs with bounded-predicate arities.

III

Characterizing Modular Nonmonotonic Logic Programs



6

Translation of Modular Nonmonotonic Logic Programs to Datalog

In this chapter, we investigate rewriting techniques for translating Modular Nonmonotonic Logic Programs with module input into programs of simpler structure. We are mainly concerned here with logic programs in the Datalog setting (Gottlob et al., 1989), i.e., we try to shift a modular logic program by rewriting the modules or a subset thereof into a Datalog program, so that large portions or even the complete MLP can be evaluated using a Datalog reasoner. As a first technique, we will show how to rewrite programs with multiple modules with empty input into an MLP in normal form in §6.2. Next, we present a general technique in §6.3 that is concerned with rewriting arbitrary MLPs into ones that have no module input such that they can, if desired, be transformed into logic programs without modules at all. This approach comes at a cost; in the worst case, the translation is exponentially larger than the original MLP. The macro expansion technique presented in §6.4 deals with a restricted syntactic class of MLPs that do not incur this blow-up. While macro expansion cannot be applied to general MLPs, macros are still important in the context of converting, e.g., Datalog-rewritable dl-programs (Heymans et al., 2010) into modular logic programs, which we will show in §6.5.

Without loss of generality, we consider MLPs of form $\mathbf{P} = (m_1, \ldots, m_n)$ such that each module $m_i = (P_i[q], R_i)$ has at most one formal input parameter. This restriction is immaterial and only serves the purpose of simplifying the rewriting rules in this chapter; essentially, one can always transform a module with a positive number of input predicates to one that has exactly one input predicate using *module input reification*. We will show how to reify modules with an input list of arbitrary length in the first §6.1.

6.1 Module Input Reification

We start this chapter by showing how to translate an MLP that contains modules with more than one formal input parameter into an equivalent MLP whose modules have at most one input parameter.

Intuitively, we can reify a module $m = (P[\mathbf{q}], R)$ into a module $m' = (P[q'], R')$ by using a fresh predicate symbol q' whose arity is 1 plus the maximal arity of all input predicate symbols of $\mathbf{q}$. Then, the set of rules R' contains

- module input rules that transfer input from q' back to the predicates from $\mathbf{q}$,
- module atom input rules that reify input in a single predicate p^e for each module atom e appearing in m, and
- a reified version of R such that each module atom $e = Q[\mathbf{p}].o(\mathbf{t})$ is replaced by its reified module atom $Q[p^e].o(\mathbf{t})$.

This technique makes the restriction we made in the beginning of this chapter insignificant, i.e., requiring that all modules having at most one input parameter, as every MLP can be converted into a reified MLP while preserving their answer sets.

In the following, let $\mathbf{P} = (m_1, \dots, m_n)$ be an arbitrary MLP such that each m_i of $\mathbf{P}$ is of form $(P_i[\mathbf{q}_i], R_i)$. Let p be a predicate symbol from $\mathbf{P}$, we define its associated arity as $a(p)$. For a list of predicate symbols $\mathbf{p} = p_1, \dots, p_\ell$ we let $a(\mathbf{p}) = \max_{p \text{ is from } \mathbf{p}} a(p)$. We define the set of module atoms appearing in module m (respectively, in a rule $r \in R(m)$) as $ma(m)$ (respectively, $ma(r)$). Let ϵ be a fresh constant symbol not appearing in $\mathbf{P}$.

Whenever a module atom has more than one input parameter, we need module atom input rules that compress all input predicates into a single reified input parameter.

Definition 6.1 (Module atom input rules and reified module atom).
For a module atom $e = P_j[\mathbf{p}].o(\mathbf{t})$ from $\mathbf{P}$ such that $|\mathbf{p}| > 1$, we define the *module atom input rules of* e as the set of rules

$$\begin{aligned} p^e(1, X_1, \dots, X_{a(p_1)}, \underbrace{\epsilon, \dots, \epsilon}_{a(\mathbf{p})-a(p_1)}) &\leftarrow p_1(X_1, \dots, X_{a(p_1)}) \\ &\vdots \\ p^e(\ell, X_1, \dots, X_{a(p_\ell)}, \underbrace{\epsilon, \dots, \epsilon}_{a(\mathbf{p})-a(p_\ell)}) &\leftarrow p_\ell(X_1, \dots, X_{a(p_\ell)}) \end{aligned}$$

where p^e is a fresh predicate symbol not appearing in $\mathbf{P}$ with arity $a(p^e) = a(\mathbf{p}) + 1$. The *reified module atom for* e is the module atom $P_j[p^e].o(\mathbf{t})$.

Note that ϵ is used to fill up missing positions when defining $p^e(i, \dots)$ for predicates p_i such that $a(p_i) < a(\mathbf{p})$.

Whenever a module has more than one input parameter, we need module input rules that transfer the reified input parameter back to their original input parameters.

Definition 6.2 (Module input rules).
Given a list of formal input parameters $\mathbf{q}_i = q_1, \dots, q_k$ of a library module m_i of $\mathbf{P}$ such that $k > 1$, we define the *module input rules of* m_i as the set of rules

$$\begin{aligned} q_1(X_1, \dots, X_{a(q_1)}) &\leftarrow q_i'(1, X_1, \dots, X_{a(q_1)}, \underbrace{\epsilon, \dots, \epsilon}_{a(\mathbf{q})-a(q_1)}) \\ &\vdots \\ q_k(X_1, \dots, X_{a(q_k)}) &\leftarrow q_i'(k, X_1, \dots, X_{a(q_k)}, \underbrace{\epsilon, \dots, \epsilon}_{a(\mathbf{q})-a(q_k)}) \end{aligned}$$

The following definition gives us reified MLPs, whose modules have at most one input parameter.

Definition 6.3 (Reified module and reified MLP).
For any module m_i from $\mathbf{P}$, the *reified module for* m_i is the module

$$m_i' = \begin{cases} (P_i[q_i'], R_i') & \text{if} |\mathbf{q}_i| > 1 \\ (P_i[\mathbf{q}_i], R_i') & \text{if } |\mathbf{q}_i| \leq 1 \end{cases}$$

where q_i' is a fresh predicate symbol with arity $a(q_i') = a(\mathbf{q}_i) + 1$, and R_i' consists of

- the module input rules of m_i in case $|\mathbf{q}_i| > 1$,
- module atom input rules for each module atom $Q[\mathbf{p}].o(\mathbf{t})$ appearing in R_i whenever $|\mathbf{p}| > 1$, and
- all rules from R_i such that each module atom $Q[\mathbf{p}].o(\mathbf{t}) \in ma(m_i)$ with $|\mathbf{p}| > 1$ is replaced by its reified module atom.

The *reified MLP* $\mathbf{P}'$ is the MLP $(m_1', \dots, m_n')$, where m_i' is the reified module for m_i.

To show that reified MLPs are equivalent to general MLPs, we define the following functions and interpretations. Let $A \subseteq \mathrm{HB}_{\mathbf{P}}$ be a set ground atoms, let $\mathbf{p} = p_1, \dots, p_\ell$ be a list predicates, and let q be a predicate symbol. We define

$$r(A, \mathbf{p}, q) = \bigcup_{p_i \text{ is from } \mathbf{p}} \left\{ q(i, \mathbf{c}, \underbrace{\epsilon, \dots, \epsilon}_{a(\mathbf{p})-a(p_i)}) \;\middle|\; p_i(\mathbf{c}) \in A \right\}.$$

and

$$\hat{r}(A, q, \mathbf{p}) = \bigcup_{p_i \text{ is from } \mathbf{p}} \left\{ p_i(\mathbf{c}) \;\middle|\; q(i, \mathbf{c}, \underbrace{\epsilon, \dots, \epsilon}_{a(\mathbf{p})-a(p_i)}) \in A \right\}.$$

Let $\mathbf{M}$ be an interpretation for an MLP $\mathbf{P}$, we define $\mathit{incl}(\mathbf{M})$ to be an interpretation $\mathbf{M}'$ for the reified MLP $\mathbf{P}'$ such that for all $P_i[S] \in \mathrm{VC}(\mathbf{P})$, we set

- $M_i'/S = M_i/S$ in $\mathbf{M}'$ whenever $|\mathbf{q}_i| \leq 1$ in module $m_i = (P_i[\mathbf{q}_i], R_i)$ from $\mathbf{P}$, and
- otherwise, we set

$$M_i'/\big(r(S, \mathbf{q}_i, q_i')\big) = M_i/S \cup r(S, \mathbf{q}_i, q_i') \cup \bigcup_{e=Q[\mathbf{p}].o(\mathbf{t})\in ma(R_i)} r(M_i/S, \mathbf{p}, p^e) \ .$$

Let $\mathbf{M}'$ be an interpretation for the reified MLP $\mathbf{P}'$. We let $excl(\mathbf{M}')$ to be the interpretation for the MLP $\mathbf{P}$ such that for all $P_i[S] \in \mathrm{VC}(\mathbf{P}')$, we set

- $M_i/S = M_i'/S$ in $\mathbf{M}$ whenever $|\mathbf{q}_i| \leq 1$ in module $m_i = (P_i[\mathbf{q}_i], R_i)$ from $\mathbf{P}$, and
- otherwise, we set

$$M_i/\big(\hat{r}(S, q_i', \mathbf{q}_i)\big) = \Big(M_i'/S \setminus HB_{\mathbf{P}'}|_{\mathbf{q}_i'}\Big) \cup \hat{r}(S, q_i', \mathbf{q}_i) \cup \bigcup_{e=Q[\mathbf{p}].o(\mathbf{t})\in ma(R_i)} \hat{r}(M_i'/S, p^e, \mathbf{p}) \ .$$

We can now show the following.

Proposition 6.1 (Module input reification)
Let $\mathbf{P}$ be an arbitrary MLP. Then, the answer sets of $\mathbf{P}$ correspond one-to-one to the answer sets of the reified MLP $\mathbf{P}'$.

Proof Let $\mathbf{M}$ and $\mathbf{M}'$ be interpretations for $\mathbf{P}$ and $\mathbf{P}'$, respectively. We first show that both $incl(\mathbf{M})$ and $excl(\mathbf{M}')$ are interpretations for $\mathbf{P}'$ and $\mathbf{P}$, respectively. Whenever $|\mathbf{q}_i| \leq 1$ in a module $m_i = (P_i[\mathbf{q}_i], R_i)$ from $\mathbf{P}$, module input reification does not change m_i', thus $P_i[S] \in \mathrm{VC}(\mathbf{P})$ if and only if $P_i[S] \in \mathrm{VC}(\mathbf{P}')$. For the case $|\mathbf{q}_i| > 1$, the reified module m_i' has only one input predicate q_i' derived from $\mathbf{q}_i$. There is a one-to-one correspondence between the value calls $P_i[S] \in \mathrm{VC}(\mathbf{P})$ and $P_i[S'] \in \mathrm{VC}(\mathbf{P}')$, concretely $S' = r(S, \mathbf{q}_i, q_i')$ and $S = \hat{r}(S', q_i', \mathbf{q}_i)$. Hence, $P_i[S] \in \mathrm{VC}(\mathbf{P})$ if and only if $P_i[S'] \in \mathrm{VC}(\mathbf{P}')$. Furthermore, the call graphs $\mathrm{CG}_{\mathbf{P}}$ and $\mathrm{CG}_{\mathbf{P}'}$ are isomorphic: edge $(u, v) \in E(\mathrm{CG}_{\mathbf{P}})$ if and only if edge $(f(u), f(v)) \in E(\mathrm{CG}_{\mathbf{P}'})$ for the bijection $f : V(\mathrm{CG}_{\mathbf{P}}) \to V(\mathrm{CG}_{\mathbf{P}'})$ such that

$$f(P_i[S]) = \begin{cases} P_i[r(S, \mathbf{q}_i, q_i')] & \text{if } |\mathbf{q}_i| > 1 \\ P_i[S] & \text{otherwise} \end{cases}$$

and

$$f^{-1}(P_i[S']) = \begin{cases} P_i[\hat{r}(S', q_i', \mathbf{q}_i)] & \text{if } |\mathbf{q}_i| > 1 \\ P_i[S'] & \text{otherwise} \end{cases} \ .$$

Rules without module atoms are left unchanged in reified MLPs, and using the Horn module input rules, the rules agree on their truth values in both $\mathbf{M}$ and $\mathbf{M}'$. Now,

for rules with module atoms of form $P_j[\mathbf{p}].o(\mathbf{c}) \in ma(gr(R_i))$ such that $\mathbf{q}_j$ is the input list of $P_j[\mathbf{q_j}]$, we distinguish the case $|\mathbf{q}_j| \leq 1$, and $|\mathbf{q}_j| > 1$. In the former case, module input reification leaves the module atom unchanged, therefore they use the same value call in the respective call graph to determine their truth value, and thus $\mathbf{M}, P_i[S] \models P_j[\mathbf{p}].o(\mathbf{c})$ if and only if $\mathbf{M}', P_i[S] \models P_j[\mathbf{p}].o(\mathbf{c})$. In the case $|\mathbf{q}_j| > 1$, module input reification changes $P_j[\mathbf{p}].o(\mathbf{c}) \in ma(gr(R_i))$ to $P_j[p^e].o(\mathbf{c}) \in ma(gr(R_i'))$ for $e = P_j[\mathbf{p}].o(\mathbf{t}) \in ma(R_i)$. The Horn module atom input rules transfer the extension of each predicate p in $\mathbf{p}$ to the predicate p^e, and given that the call graphs are isomorphic using f, we arrive at identical value calls in both $\mathbf{P}$ and $\mathbf{P}'$. The Horn module input rules then dissect the extension of predicate q_j' to the respective predicate q from $\mathbf{q}_j$, and we get that corresponding rules are true in both $\mathbf{M}$ and $\mathbf{M}'$ for $\mathbf{P}$ and $\mathbf{P}'$, respectively. Hence, the truth of a ground module atom $P_j[\mathbf{p}].o(\mathbf{c}) \in ma(gr(R_i))$ in $\mathbf{M}$ for $\mathbf{P}$ coincides with the truth of the corresponding ground module atom $P_j[p^e].o(\mathbf{c}) \in ma(R_i')$ in $\mathbf{M}'$ for $\mathbf{P}'$. Thus, the rules in $f\,\mathbf{P}(P_i[S])^{\mathbf{M}}$ coincide with the rules in $f\,\mathbf{P}'(P_i[S'])^{\mathbf{M}'}$, and thus $\mathbf{M}$ is an answer set for $\mathbf{P}$ if and only if $\mathbf{M}'$ is an answer set for $\mathbf{P}'$. □

Note that applying module input reification to an MLP $\mathbf{P} = (m_1, \ldots, m_n)$ adds at most $n-1$ new predicate symbols q_i' and their module input rules for all library modules m_i of $\mathbf{P}$. If a library module m_i has input $\mathbf{q}_i$ such that $|\mathbf{q}_i| = j$, then there will be j module input rules for m_i. Then, if the modules of $\mathbf{P}$ contain k module atoms $e_1, \ldots, e_k$, at most k new predicate symbols $p^{e_1}, \ldots, p^{e_k}$ will be introduced together with their module atom input rules. Module atoms e_i having input list $\mathbf{p}_i$ such that $|\mathbf{p}_i| = \ell$, give us ℓ additional module atom input rules in a module of $\mathbf{P}$. The modification of module atoms to take p^{e_i} as module input does not increase the overall complexity.

Main modules and library modules with at most one input parameter do not change their input signature and will not receive additional module input rules. The same is true for rules with module atoms with at most one input parameter, which do not change and therefore do not require to add module atom input rules to our MLP. Therefore, whenever an MLP has only modules with at most one formal input parameter, module input reification leaves the MLP untouched.

6.2 Rewriting Modules without Input

In this section, we define a simple syntactic restriction for modular logic programs: there is exactly one module without input, the main module. This does not impose a strong limitation in writing MLPs, as we will see below that every MLP can be reduced with a linear rewriting step into its normal form. The rewriting methodology generally comes in two stages: we combine all rules from all the input-less modules to a fresh main module m_0, and then exchange in the rules of the remaining modules calls to

the former input-less modules to calls to the fresh main module m_0. We now begin by defining a normal form for modular logic programs.

Definition 6.4 (Normal form).
An MLP $\mathbf{P} = (m_1, \dots, m_n)$ is in *normal form*, if exactly one module m_i of $\mathbf{P}$ has no formal input parameters, i.e., $m_i = (P_i[], R_i)$.

Note that Definition 3.2 then implies that module m_i must be the main module for $\mathbf{P}$.

In order to rewrite MLPs to their normal form, we first define a normalization procedure for atoms and rules of a module.

Definition 6.5 (Module normalization).
For an atom a appearing in a module $m = (P[q], R)$ of MLP $\mathbf{P} = (m_1, \dots, m_n)$, we define

$$\mathcal{N}(a) = \begin{cases} a & a \text{ is of form } p(\mathbf{t}) \text{ or } P_i[p].o(\mathbf{t}) \ , \\ P_0.o(\mathbf{t}) & a \text{ is of form } P_j.o(\mathbf{t}) \ . \end{cases}$$

Given a rule $r \in R$ of form (3.2) from a module $m = (P[q], R)$, we define the rule

$$\begin{aligned} \mathcal{N}(r) = \alpha_1 \vee \dots \vee \alpha_k \leftarrow\ & \mathcal{N}(\beta_1), \dots, \mathcal{N}(\beta_m), \\ & \text{not } \mathcal{N}(\beta_{m+1}), \dots, \text{not } \mathcal{N}(\beta_n), \ , \end{aligned} \tag{6.1}$$

and for a set of rules R we define $\mathcal{N}(R) = \{\mathcal{N}(r) \mid r \in R\}$. For the module m we let

$$\mathcal{N}(m) = (P[q], \mathcal{N}(R)) \ .$$

The next definition shows how to normalize an MLP $\mathbf{P}$.

Definition 6.6 (MLP normalization).
Let $\mathbf{P} = (m_1, \dots, m_n)$ be an MLP such that each module $m_i = (P_i[q_i], R_i)$ from $\mathbf{P}$ has at most one input parameter q_i, and let $I_0 = \{m_{i_1}, \dots, m_{i_k}\}$ be the set of all modules from $\mathbf{P}$ that have no formal input, and $I_1 = \{m_{j_1}, \dots, m_{j_{n-k}}\}$ be the modules with exactly one input parameter. We define the *MLP normalization* $\mathcal{N}(\mathbf{P})$ *of MLP* $\mathbf{P}$ to be the MLP

$$\mathcal{N}(\mathbf{P}) = (m_0, \mathcal{N}(m_{j_1}), \dots, \mathcal{N}(m_{j_{n-k}})) \ ,$$

where m_0 is the fresh module $(P_0[], R_0)$ with $R_0 = \bigcup_{m \in I_0} \mathcal{N}(R(m))$.

Note that I_0 is always nonempty, as every MLP has a main module. Applying module normalization on an MLP that is already in normal form does not change the call structure, it simply replaces the single main module, say m_i, with a new module m_0 and replaces each call to m_i with a call to m_0. That is, normalizing simply renames $P_i[]$ to $P_0[]$.

We can now show that every MLP $\mathbf{P}$ can be transformed into an equivalent MLP $\mathcal{N}(\mathbf{P})$ that is in normal form. Without loss of generality, we assume for the next result that each ordinary atom from $\mathbf{P}$ has a label identifying the rule base it appears in: for a module m_j, each ordinary atom in $R(m_j)$ is of form $p^j(\mathbf{t})$.

Lemma 6.2
The answer set of MLP $\mathbf{P}$ correspond one-to-one to the answer sets of MLP $\mathcal{N}(\mathbf{P})$, i.e.,

- for every answer set $\mathbf{M}$ of $\mathbf{P}$, there exists an answer set $\mathbf{M}'$ of $\mathcal{N}(\mathbf{P})$ such that

$$M'_0/\varnothing = \bigcup_{m_{i_j} \in I_0} M_{i_j}/\varnothing, \text{ and} \tag{6.2}$$

$$M'_i/S = M_i/S \text{ for } m_i \in I_1 \tag{6.3}$$

- for every answer set $\mathbf{M}'$ of $\mathcal{N}(\mathbf{P})$ there exists an answer set $\mathbf{M}$ of $\mathbf{P}$ such that

$$M_i/S = M'_i/S \text{ for } m_i \in I_1, \text{ and} \tag{6.4}$$

$$M_{i_j}/\varnothing = \{a^{i_j} \in M'_0/\varnothing\} \text{ for } m_{i_j} \in I_0 \ . \tag{6.5}$$

Proof ($\Rightarrow$) Let $\mathbf{M}$ be an answer set of $\mathbf{P}$. We show now that there exists an answer set $\mathbf{M}'$ of $\mathcal{N}(\mathbf{P})$ such that (a) $\mathbf{M}' \vDash f\,\mathcal{N}(\mathbf{P})^{\mathbf{M}'}$, and that (b) $\mathbf{M}'$ is a minimal model of $f\,\mathcal{N}(\mathbf{P})^{\mathbf{M}'}$.

We start with item a. Let $r' \in f\,\mathcal{N}(\mathbf{P})(P_i[S])^{\mathbf{M}'}$ for $m_i \in I_1$. For the rule r such that $\mathcal{N}(r) = r'$ and $r = r'$, all calls in r' and r do not access module m_0 respectively some $m_{i_j} \in I_0$. By construction of $\mathbf{M}'$, we can conclude that $r \in f\,\mathbf{P}(P_i[S])^{\mathbf{M}}$. In case $r \neq r'$, one module atom $a' = P_0.o(\mathbf{c})$ must appear in $B(r')$. Let $a = P_{i_j}.o(\mathbf{c})$ be the atom such that $\mathcal{N}(a) = a'$, we have that $m_{i_j} \in I_0$. From (6.2) we can conclude that $o(\mathbf{c}) \in M'_0/\varnothing$ iff $o(\mathbf{c}) \in M_{i_j}/\varnothing$. Therefore, $r \in f\,\mathbf{P}(P_i[S])^{\mathbf{M}}$. In both cases, we can conclude that $\mathbf{M}' \vDash f\,\mathcal{N}(\mathbf{P})(P_i[S])^{\mathbf{M}'}$ since $\mathbf{M}$ is a model for all rules in $f\,\mathbf{P}^{\mathbf{M}}$.

Let $r' \in f\,\mathcal{N}(\mathbf{P})(P_0[\varnothing])^{\mathbf{M}'}$ such that $r' = \mathcal{N}(r)$ for a rule r from $m_{i_j} \in I_0$. All atoms in $B(r')$ are satisfied by $M'_0/\varnothing$. We have that for atoms $a' \in B(r')$ with the corresponding atom $a \in B(r)$ such that $\mathcal{N}(a) = a'$, either $a = a'$ or $a \neq a'$. In the former case, a is ordinary and from m_{i_j}, or a is a module atom $P_i[p].o(\mathbf{c})$ calling $m_i \in I_1$ from $m_{i_j} \in I_0$. Thus, $\mathbf{M}, P_{i_j}[\varnothing] \vDash a$ iff $\mathbf{M}', P_0[\varnothing] \vDash a'$, which follows from (6.2) and (6.3). In case $a \neq a'$, a' is of form $P_0.o(\mathbf{c})$ for a module atom a of form $P_k.o(\mathbf{c})$ such that $m_k \in I_0$. From (6.2) we can conclude that $o(\mathbf{c}) \in M'_0/\varnothing$ iff $o(\mathbf{c}) \in M_k/\varnothing$. Therefore, $\mathbf{M}', P_0[\varnothing] \vDash a'$ iff $\mathbf{M}, P_k[\varnothing] \vDash a$. In both cases, we can now conclude that $r \in f\,\mathbf{P}(P_{i_j}[\varnothing])^{\mathbf{M}}$, and since $\mathbf{M} \vDash f\,\mathbf{P}^{\mathbf{M}}$, we get $\mathbf{M}' \vDash f\,\mathcal{N}(\mathbf{P})(P_0[\varnothing])^{\mathbf{M}'}$.

Since $\mathbf{M}' \vDash f\,\mathcal{N}(\mathbf{P})(P_i[S])^{\mathbf{M}'}$ for all $P_i[S] \in \mathrm{VC}(\mathcal{N}(\mathbf{P}))$, we have shown that $\mathbf{M}' \vDash f\,\mathcal{N}(\mathbf{P})^{\mathbf{M}'}$.

Next, we consider item b. To show that $\mathbf{M}'$ is a minimal model of $f\,\mathcal{N}(\mathbf{P})^{\mathbf{M}'}$, we must ensure that there is no interpretation $\mathbf{M}''$ such that $\mathbf{M}'' < \mathbf{M}'$ and $\mathbf{M}'' \vDash f\,\mathcal{N}(\mathbf{P})^{\mathbf{M}'}$.

Towards a contradiction, assume $\mathbf{M}''$ satisfies $f\,\mathcal{N}(\mathbf{P})^{\mathbf{M}'}$. As $\mathbf{M}'' < \mathbf{M}'$, we consider the following cases: (1) for some M_i''/S with $m_i \in I_1$, we have that $M_i''/S \subset M_i'/S$; or (2) we have that $M_0''/\varnothing \subset M_0'/\varnothing$.

Let $\mathbf{N}$ denote an interpretation for $\mathbf{P}$ such that $N_i/S = M_i''/S$ for $m_i \in I_1$, thus $N_i/S = M_i'/S = M_i/S$. Furthermore, for $m_{i_j} \in I_0$, let $N_{i_j}/\varnothing = \{a^{i_j} \in M_0''/\varnothing\}$.

From the construction of M_i''/S in case (1), and of $M_0''/\varnothing$ in case (2), we can now deduce that $N_i/S \subset M_i/S$ for case (1), and that $N_{i_j}/\varnothing \subset M_{i_j}/\varnothing$ in case (2). Hence $\mathbf{N} < \mathbf{M}$, and from $\mathbf{M}$ being a minimal model of $f\,\mathbf{P}^{\mathbf{M}}$, we get in both cases that $\mathbf{N}, P_i[S] \not\models f\,\mathbf{P}^{\mathbf{M}}$ and $\mathbf{N}, P_{i_j}[\varnothing] \not\models f\,\mathbf{P}^{\mathbf{M}}$, respectively. There is a rule $r \in f\,\mathbf{P}(P_i[S])^{\mathbf{M}}$ (respectively, $r \in f\,\mathbf{P}(P_{i_j}[\varnothing])^{\mathbf{M}}$) such that $\mathbf{N}, P_i[S] \not\models r$ (respectively, $\mathbf{N}, P_{i_j}[\varnothing] \not\models r$). From the construction of M_i'/S (respectively, $M_0'/\varnothing$), we can deduce that there must be a corresponding rule $r' \in f\,\mathcal{N}(\mathbf{P})(P_i[S])^{\mathbf{M}'}$ (respectively, $r' \in f\,\mathcal{N}(\mathbf{P})(P_0[\varnothing])^{\mathbf{M}'}$) such that $r' = \mathcal{N}(r)$. But now we arrive at a contradiction, as in case (1) $\mathbf{M}'', P_i[S] \not\models r'$ and in case (2) $\mathbf{M}'', P_0[\varnothing] \not\models r'$. Thus, $\mathbf{M}''$ does not satisfy $f\,\mathcal{N}(\mathbf{P})(P_i[S])^{\mathbf{M}'}$ (respectively, $f\,\mathcal{N}(\mathbf{P})(P_0[\varnothing])^{\mathbf{M}'}$).

Therefore, in both cases (1) and (2) we can conclude that $\mathbf{M}'$ is a minimal model of $f\,\mathcal{N}(\mathbf{P})^{\mathbf{M}'}$.

($\Leftarrow$) Let $\mathbf{M}'$ be an answer set of $\mathcal{N}(\mathbf{P})$. We show now that there exists a corresponding answer set $\mathbf{M}$ of $\mathbf{P}$ such that (a) $\mathbf{M} \models f\,\mathbf{P}^{\mathbf{M}}$, and that (b) $\mathbf{M}$ is a minimal model of $f\,\mathbf{P}^{\mathbf{M}}$.

Consider item a. Let $r \in f\,\mathbf{P}(P_i[S])^{\mathbf{M}}$ for $m_i \in I_1$. For the rule r' such that $\mathcal{N}(r) = r'$ and $r = r'$, all module calls in r' and r do not access module m_0 respectively some $m_{i_j} \in I_0$. By construction of $\mathbf{M}$ in (6.4), we can conclude that $r' \in f\,\mathcal{N}(\mathbf{P})(P_i[S])^{\mathbf{M}'}$. In case $r \neq r'$, one module atom $a' = P_0.o(\mathbf{c})$ must appear in $B(r')$. Let $a = P_{i_j}.o^{i_j}(\mathbf{c})$ be the atom such that $\mathcal{N}(a) = a'$, we have that $m_{i_j} \in I_0$. From (6.5) we can conclude that $o^{i_j}(\mathbf{c}) \in M_0'/\varnothing$ iff $o^{i_j}(\mathbf{c}) \in M_{i_j}/\varnothing$. Therefore, $r' \in f\,\mathcal{N}(\mathbf{P})(P_i[S])^{\mathbf{M}'}$. In both cases, we can conclude that $\mathbf{M} \models f\,\mathbf{P}(P_i[S])^{\mathbf{M}}$ since $\mathbf{M}'$ is a model for all rules in $f\,\mathcal{N}(\mathbf{P})^{\mathbf{M}'}$.

Let $r \in f\,\mathbf{P}(P_{i_j}[\varnothing])^{\mathbf{M}}$ such that $r' = \mathcal{N}(r)$ for a rule r from $m_{i_j} \in I_0$. All atoms in $B(r)$ are satisfied by $M_{i_j}/\varnothing$. We have that for atoms $a \in B(r)$ with the corresponding atom $a' \in B(r')$ such that $\mathcal{N}(a) = a'$, either $a = a'$ or $a \neq a'$.

In the former case, a is ordinary and from m_{i_j}, or a is a module atom $P_i[p].o(\mathbf{c})$ calling $m_i \in I_1$ from $m_{i_j} \in I_0$. Thus, $\mathbf{M}, P_{i_j}[\varnothing] \models a$ iff $\mathbf{M}', P_0[\varnothing] \models a'$, which follows from (6.4) and (6.5).

In case $a \neq a'$, a' is of form $P_0.o^k(\mathbf{c})$ for a module atom a of form $P_k.o^k(\mathbf{c})$ such that $m_k \in I_0$. From (6.5) we can conclude that $o^k(\mathbf{c}) \in M_0'/\varnothing$ iff $o^k(\mathbf{c}) \in M_k/\varnothing$. Therefore, $\mathbf{M}', P_0[\varnothing] \models a'$ iff $\mathbf{M}, P_k[\varnothing] \models a$. In both cases, we can now conclude that $r' \in f\,\mathcal{N}(\mathbf{P})(P_{i_j}[\varnothing])^{\mathbf{M}'}$, and since $\mathbf{M}' \models f\,\mathcal{N}(\mathbf{P})^{\mathbf{M}'}$, we get $\mathbf{M} \models f\,\mathbf{P}(P_{i_j}[\varnothing])^{\mathbf{M}}$.

Since $\mathbf{M} \models f\,\mathbf{P}(P_i[S])^{\mathbf{M}}$ for all $P_i[S] \in \mathrm{VC}(\mathbf{P})$, we have shown that $\mathbf{M} \models f\,\mathbf{P}^{\mathbf{M}}$.

We show item b next. To show that $\mathbf{M}$ is a minimal model of $f\,\mathbf{P}^{\mathbf{M}}$, we must ensure that there is no interpretation $\mathbf{M}''$ such that $\mathbf{M}'' < \mathbf{M}$ and $\mathbf{M}'' \models f\,\mathbf{P}^{\mathbf{M}}$.

Towards a contradiction, assume $\mathbf{M}''$ satisfies $f\,\mathbf{P}^{\mathbf{M}}$. As $\mathbf{M}'' < \mathbf{M}$, we consider the following cases: (1) for some M_i''/S with $m_i \in I_1$, we have that $M_i''/S \subset M_i/S$; or (2) we have that for $m_{i_j} \in I_0$, $M_{i_j}''/\varnothing \subset M_{i_j}/\varnothing$.

Let $\mathbf{N}$ denote an interpretation for $\mathcal{N}(\mathbf{P})$ such that $N_i/S = M_i''/S$ for $m_i \in I_1$, thus $N_i/S = M_i'/S = M_i/S$. Furthermore, let $N_0/\varnothing = \bigcup_{m_{i_j} \in I_0} \{a^{i_j} \in M_{i_j}''/\varnothing\}$.

From the construction of M_i''/S in case (1), and of $M_{i_j}''/\varnothing$ in case (2), we can now deduce that $N_i/S \subset M_i'/S$ for case (1), and that $N_0/\varnothing \subset M_0'/\varnothing$ in case (2). Hence $\mathbf{N} < \mathbf{M}'$, and from $\mathbf{M}'$ being a minimal model of $f\,\mathcal{N}(\mathbf{P})^{\mathbf{M}'}$, we get in both cases that $\mathbf{N}, P_i[S] \not\models f\,\mathcal{N}(\mathbf{P})^{\mathbf{M}'}$ and $\mathbf{N}, P_0[\varnothing] \not\models f\,\mathcal{N}(\mathbf{P})^{\mathbf{M}'}$, respectively. There is a rule $r' \in f\,\mathcal{N}(\mathbf{P})(P_i[S])^{\mathbf{M}'}$ (respectively, $r' \in f\,\mathcal{N}(\mathbf{P})(P_0[\varnothing])^{\mathbf{M}'}$) such that $\mathbf{N}, P_i[S] \not\models r'$ (respectively, $\mathbf{N}, P_0[\varnothing] \not\models r'$). From the construction of M_i/S respectively $M_{i_j}/\varnothing$, we can deduce that there must be a corresponding rule $r \in f\,\mathbf{P}(P_i[S])^{\mathbf{M}}$ respectively $r \in f\,\mathbf{P}(P_{i_j}[\varnothing])^{\mathbf{M}}$ such that $r' = \mathcal{N}(r)$.

But now we arrive at a contradiction, as in case (1) $\mathbf{M}'', P_i[S] \not\models r$ and in case (2) $\mathbf{M}'', P_{i_j}[\varnothing] \not\models r$. Thus, $\mathbf{M}''$ does not satisfy $f\,\mathbf{P}(P_i[S])^{\mathbf{M}}$ (respectively, $f\,\mathbf{P}(P_{i_j}[\varnothing])^{\mathbf{M}}$).

Therefore, in both cases (1) and (2) we can conclude that $\mathbf{M}$ is a minimal model of $f\,\mathbf{P}^{\mathbf{M}}$. □

Example 6.1 Consider the MLP $\mathbf{P} = (m_1, m_2, m_3)$, where $m_1 = (P_1[], R_1)$ is a main module, $m_2 = (P_2[], R_2)$ is an input-less module, and $m_3 = (P_3[q/1], R_3)$ is a library module. The rules are

- $R_1 = \{a \leftarrow P_2.o, \quad c \leftarrow P_3[a].b\}$;
- $R_2 = \{o \leftarrow \text{not } p, \quad p \leftarrow \text{not } o\}$; and
- $R_3 = \{b \leftarrow P_2.o\}$.

The normalization of $\mathbf{P}$ is the MLP $\mathcal{N}(\mathbf{P}) = (m_0, \mathcal{N}(m_3))$, where $m_0 = (P_0[], R_0)$ is the new main module, and $\mathcal{N}(m_3) = (P_3[q/1], \mathcal{N}(R_3))$ is the normalized library module; the rule bases are as follows:

- $R_0 = \{a \leftarrow P_0.o, \quad c \leftarrow P_3[a].b \quad o \leftarrow \text{not } p, \quad p \leftarrow \text{not } o\}$ and
- $\mathcal{N}(R_3) = \{b \leftarrow P_0.o\}$.

6.3 General MLP Rewriting Techniques to Datalog

This section is split into three parts, each of them describe a rewriting technique for MLPs: *instance rewriting*, *call rewriting*, and *module removal of connected closed call sets*.

The approach for the instance rewriting translation is the following. Given a set S of modules from an MLP $\mathbf{P}$, we create for each module of S a fresh module without input parameters. Fresh modules get auxiliary rules and rewritten rules from their original modules such that the adapted rules encode the value calls of the original module using extra parameters in each atom. Additionally, in case a module from S calls another module from S, the module atom will be adapted to call the fresh module without input parameters. Thus, the outcome of the instance rewriting will be an extension of $\mathbf{P}$ that includes a clone of S, which preserves the call structure in S. The call rewriting translation goes one step further and separates the original modules from the cloned modules completely, thus the result of applying call rewriting to $\mathbf{P}$ will be an MLP that has two parts: a cloned part, whose modules have no input parameters and resemble the original MLP in call structure and answer sets, and a degenerated original part, whose call structure has now been torn apart. After call rewriting, we can take off the original part from $\mathbf{P}$ and receive an MLP without input parameters using the module removal technique.

6.3.1 Instance Rewriting

We now start to define the instance rewriting translation. Let $m = (P[q], R)$ be a module of $\mathbf{P}$ and $\ell > 0$. We define for the list of distinct variables $\mathbf{B} = B_0, \dots, B_{\ell-1}$ the list of unary atoms

$$\mathtt{bit}(\mathbf{B}) = bit(B_0), \dots, bit(B_{\ell-1}) \ ,$$

where bit is a fresh predicate symbol not appearing in $\mathbf{P}$. We define the set of module atoms appearing in module m (respectively, in a rule $r \in R(m)$) as $ma(m)$ (respectively, $ma(r)$).

For the integer $k \geq 0$, we define $\mathtt{bpos}_k \colon \mathcal{C}^k \to \{0, \dots, |\mathcal{C}^k| - 1\}$ to be a bijective function that sends a k-tuple $\mathbf{c} \in \mathcal{C}^k$ to an integer from $\{0, \dots, |\mathcal{C}^k| - 1\}$. When clear from context, we omit k from $\mathtt{bpos}_k$. We define

$$\mathtt{bv}(\mathbf{c}) = B_0, \dots, B_{j-1}, b_j, B_{j+1}, \dots, B_{|\mathcal{C}^k|-1}$$

to be the list of terms such that $b_j = 1$ for $j = \mathtt{bpos}_k(\mathbf{c})$ and for each $i_1 \neq i_2$ we have that B_{i_1} and B_{i_2} are pairwise distinct variables.

Intuitively, both $\mathtt{bit}(\mathbf{B})$ and $\mathtt{bv}(\mathbf{c})$ allow us to inflate ordinary predicates appearing in a module such that the inflated tuple pinpoints to a value call in the call graph $\mathrm{CG}_{\mathbf{P}}$ for an MLP $\mathbf{P}$. While the atoms $\mathtt{bit}(\mathbf{B})$ will be used to bind the variables B_i to 0 and 1, the list of terms $\mathtt{bv}(\mathbf{c})$ is then added to each inflated ordinary atom from the input list q of the module m.

Example 6.2 Let $\mathbf{c} = (a_7, a_8, a_9)$ be a triple formed from $\mathcal{C} = \{a_0, a_1, \dots, a_9\}$. Hence, $|\mathcal{C}| = 10$ and the set of all triples over $\mathcal{C}$ is of size $|\mathcal{C}^3| = 10^3 = 1000$. We can now

use $\texttt{bpos}_3(\cdot)$ to encode $\mathbf{c}$ as integer as follows: given the triple $(a_{i_1}, a_{i_2}, a_{i_3}) \in \mathcal{C}^3$, we let $\texttt{bpos}_3(a_{i_1}, a_{i_2}, a_{i_3}) = 10^2 \cdot i_1 + 10^1 \cdot i_2 + 10^0 \cdot i_3$ and its inverse be $\texttt{bpos}_3^{-1}(N) = \left(\left\lfloor \frac{N}{10^2} \right\rfloor \bmod 10, \left\lfloor \frac{N}{10^1} \right\rfloor \bmod 10, \left\lfloor \frac{N}{10^0} \right\rfloor \bmod 10\right)$. Thus, we get $\texttt{bpos}_3(a_7, a_8, a_9) = 789$, and for the inverse function $\texttt{bpos}_3^{-1}(789) = (a_7, a_8, a_9)$. Moreover, the list $\texttt{bv}(\mathbf{c})$ has length 1000 and consists of the terms $B_0, \dots, B_{788}, 1, B_{790}, \dots, B_{999}$.

In general, given the set of constant symbols $\mathcal{C} = \{a_0, \dots, a_{n-1}\}$ we can encode k-tuples $\mathbf{c} = (a_{i_1}, \dots, a_{i_k})$ with

$$\texttt{bpos}_k(a_{i_1}, \dots, a_{i_k}) = \sum_{j=1}^{k} i_j \cdot n^{k-j}$$

and

$$\texttt{bpos}_k^{-1}(N) = \left(\left\lfloor \frac{N}{n^{k-1}} \right\rfloor \bmod n, \left\lfloor \frac{N}{n^{k-2}} \right\rfloor \bmod n, \dots, \left\lfloor \frac{N}{n^1} \right\rfloor \bmod n, \left\lfloor \frac{N}{n^0} \right\rfloor \bmod n\right) \ .$$

As a first step, we define functions that, given a module m of $\mathbf{P}$, produce rules that prepare the input for m with $\mathcal{I}(m)$, and generate rules that mimic the value calls with $\mathcal{VR}(m)$, which is based on the module atoms appearing in m. Both $\mathcal{I}(m)$ and $\mathcal{VR}(m)$ generate new auxiliary rules not appearing in $R(m)$.

For the following definitions, let $\mathbf{B} = B_0, \dots, B_{|\mathcal{C}^{|q|}|-1}$ be a list of pairwise distinct variables of length $|\mathcal{C}^{|q|}|$.

Definition 6.7 (Input rules and value rules).
Given module $m = (P[q], R)$, we define the *input rules of* m as the set of rules

$$\mathcal{I}(m) = \left\{ \ q(\mathbf{c}, \texttt{bv}(\mathbf{c})) \leftarrow \texttt{bit}(\mathbf{B}) \ \mid \mathbf{c} \in \mathcal{C}^{|q|} \right\} \ .$$

For a module m, we define the *value rules of* m as the set of rules

$$\mathcal{VR}(m) = \bigcup_{e \in ma(m)} \mathcal{VR}_m(e) \ ,$$

where

$$\mathcal{VR}_m(P[p].o(\mathbf{t})) = \left\{ \begin{array}{l} val_p(\mathbf{c}, 1, \mathbf{B}) \leftarrow p(\mathbf{c}, \mathbf{B}), \texttt{bit}(\mathbf{B}) \\ val_p(\mathbf{c}, 0, \mathbf{B}) \leftarrow \text{not } p(\mathbf{c}, \mathbf{B}), \texttt{bit}(\mathbf{B}) \end{array} \mid \mathbf{c} \in \mathcal{C}^{|p|} \right\} \ .$$

Example 6.3 (cont'd) Consider an MLP $\mathbf{P} = (m_1, \dots, m_n)$ with $\mathcal{C}$ as in Example 6.2. Let $m_j = (P_j[q], R_j)$ be a module from $\mathbf{P}$ such that R_j consists of the following two rules:

$$o \leftarrow$$
$$p(X_1, X_2, X_3) \leftarrow P_j[p].o, q(X_1, X_2, X_3)$$

We get for $\mathcal{I}(m_j)$ the following set of rules:

$$
\begin{aligned}
q(a_0, a_0, a_0, \quad 1, \; B_1, B_2, \ldots, B_{997}, \; B_{998}, \; B_{999}) &\leftarrow \mathtt{bit}(B_0), \ldots, \mathtt{bit}(B_{999}) \\
q(a_0, a_0, a_1, \; B_0, \quad 1, B_2, \ldots, B_{997}, \; B_{998}, \; B_{999}) &\leftarrow \mathtt{bit}(B_0), \ldots, \mathtt{bit}(B_{999}) \\
&\vdots \\
q(a_9, a_9, a_8, \; B_0, \; B_1, B_2, \ldots, B_{997}, \quad 1, \; B_{999}) &\leftarrow \mathtt{bit}(B_0), \ldots, \mathtt{bit}(B_{999}) \\
q(a_9, a_9, a_9, \; B_0, \; B_1, B_2, \ldots, B_{997}, \; B_{998}, \quad 1) &\leftarrow \mathtt{bit}(B_0), \ldots, \mathtt{bit}(B_{999})
\end{aligned}
$$

and for the module atom $e = P_j[p].o$, we get the following rules for $\mathcal{VR}_{m_j}(e)$:

$$
\begin{aligned}
val_p(a_0, a_0, a_0, 1, B_0, \ldots, B_{999}) \leftarrow \quad & p(a_0, a_0, a_0, B_0, \ldots, B_{999}), \\
& \mathtt{bit}(B_0), \ldots, \mathtt{bit}(B_{999}) \\
val_p(a_0, a_0, a_0, 0, B_0, \ldots, B_{999}) \leftarrow \quad & \text{not } p(a_0, a_0, a_0, B_0, \ldots, B_{999}), \\
& \mathtt{bit}(B_0), \ldots, \mathtt{bit}(B_{999}) \\
& \vdots \\
val_p(a_9, a_9, a_9, 1, B_0, \ldots, B_{999}) \leftarrow \quad & p(a_9, a_9, a_9, B_0, \ldots, B_{999}), \\
& \mathtt{bit}(B_0), \ldots, \mathtt{bit}(B_{999}) \\
val_p(a_9, a_9, a_9, 0, B_0, \ldots, B_{999}) \leftarrow \quad & \text{not } p(a_9, a_9, a_9, B_0, \ldots, B_{999}), \\
& \mathtt{bit}(B_0), \ldots, \mathtt{bit}(B_{999})
\end{aligned}
$$

Next, we define the translation $\mathcal{B}(m)$ that takes the rules of module m and rewrites them to rules of the same form, but with atoms of higher arity, whose purpose is to encode module instantiations of the value calls.

Definition 6.8.
For an atom a appearing in a module $m = (P[q], R)$ of MLP $\mathbf{P} = (m_1, \ldots, m_n)$, we define

$$
\mathcal{B}_m(a) = \begin{cases} p(\mathbf{t}, \mathbf{B}) & a \text{ is of form } p(\mathbf{t}) \ , \\ P_{n+j}.o(\mathbf{t}, \mathbf{V}^a) & a \text{ is of form } P_j[p].o(\mathbf{t}), \text{where } \mathbf{V}^a \\ & \text{is a list of distinct variables of length } |\mathcal{C}^{|p|}| \ , \\ a & a \text{ is of form } P_j.o(\mathbf{t}) \ . \end{cases}
$$

Note that $\mathcal{B}_m(a)$ distinguishes between module atoms a with and without input: an atom from the former category will be rewritten to a module atom that calls a module not present in $\mathbf{P}$, while a module atom from the latter class will not be rewritten and still calls the same module from $\mathbf{P}$.

Definition 6.9.
For a module atom $e = P_j[p].o(\mathbf{t})$ appearing in m, we define the list of atoms

$$\mathcal{VC}_m(e) = val_p(\mathbf{c}_1, V^e_{\mathtt{bpos}(\mathbf{c}_1)}, \mathbf{B}), \dots, val_p(\mathbf{c}_u, V^e_{\mathtt{bpos}(\mathbf{c}_u)}, \mathbf{B}) \ ,$$

where $\mathcal{C}^{|p|} = \{\mathbf{c}_1, \dots, \mathbf{c}_u\}$. Given a rule $r \in R(m)$ of form (3.2), we can now define the rule

$$\begin{aligned} \mathcal{B}_m(r) = \mathcal{B}_m(\alpha_1) \vee \dots \vee \mathcal{B}_m(\alpha_k) \leftarrow\ & \mathcal{B}_m(\beta_1), \dots, \mathcal{B}_m(\beta_m), \\ & \text{not } \mathcal{B}_m(\beta_{m+1}), \dots, \text{not } \mathcal{B}_m(\beta_n), \\ & \mathtt{bit}(\mathbf{B}), \mathcal{VC}_m(e_1), \dots, \mathcal{VC}_m(e_u) \ , \end{aligned} \tag{6.6}$$

where $ma(r) = \{e_1, \dots, e_u\}$. Furthermore, we let

$$\mathcal{B}(m) = \{\mathcal{B}_m(r) \mid r \in R(m)\} \ .$$

The module instance rewriting of m is then given as $\mathcal{T}(m)$.

Definition 6.10 (Module instance rewriting).
For a module m, we let the *module instance rewriting* $\mathcal{T}(m)$ *of* m be the set of rules

$$\mathcal{T}(m) = \{bit(0) \leftarrow, bit(1) \leftarrow\} \cup \mathcal{I}(m) \cup \mathcal{VR}(m) \cup \mathcal{B}(m) \ .$$

Example 6.4 (cont'd) Continuing our example above, we get for $\mathcal{B}(m_j)$ the set of rules

$$o(B_0, \dots, B_{999}) \leftarrow bit(B_0), \dots, bit(B_{999})$$

$$\begin{aligned} p(X_1, X_2, X_3, B_0, \dots, B_{999}) \leftarrow\ & P_{n+j}[p].o(V^e_0, \dots, V^e_{999}), q(X_1, X_2, X_3, B_0, \dots, B_{999}), \\ & bit(B_0), \dots, bit(B_{999}), \\ & val_p(\mathbf{c}_0, V^e_{\mathtt{bpos}_3(\mathbf{c}_1)}, B_0, \dots, B_{999}), \\ & \vdots \\ & val_p(\mathbf{c}_{999}, V^e_{\mathtt{bpos}_3(\mathbf{c}_{999})}, B_0, \dots, B_{999}) \end{aligned}$$

Note that $\mathcal{C}^3 = \{\mathbf{c}_0, \dots, \mathbf{c}_{999}\} = \{(a_0, a_0, a_0), \dots, (a_9, a_9, a_9)\}$, thus we obtain for

$$\mathtt{bpos}_3(\mathbf{c}_0), \dots, \mathtt{bpos}_3(\mathbf{c}_{999})$$

the sequence $0, \dots, 999$. Then, $\mathcal{T}(m_j)$ contains the facts

$$\begin{aligned} bit(0) \leftarrow \\ bit(1) \leftarrow \end{aligned}$$

all rules from Example 6.3, and $\mathcal{B}(m_j)$. Intuitively, for the module atom e from module m_j, the atoms of form $val_p(\mathbf{c}_i, V^e_{\mathtt{bpos}_3(\mathbf{c}_i)}, B_0, \dots, B_{999})$ ($i \in \{0, \dots, 999\}$) in this example encode with the list of variables $\mathbf{V}^e = V^e_0, \dots, V^e_{999}$ and $\mathbf{B} = B_0, \dots, B_{999}$ the self-call of m_j, that is, that the value call $P_j[T]$ calls $P_j[T']$. This means that $\mathbf{B}$ encodes T in the calling $P_j[T]$, and $\mathbf{V}^e$ encodes T' in the called $P_j[T']$.

The module instance rewriting $\mathcal{T}(m)$ is a set of rules that might contain module atoms of form $P_{n+j}.o(\mathbf{t}, \mathbf{V}^a)$, i.e., a call to a module that does not belong to the MLP $\mathbf{P}$. Thus, without the perspective of $\mathbf{P}$, and without further translated modules from $\mathbf{P}$ that are getting called by $\mathcal{T}(m)$, the module instance rewriting does not complete the picture of the instance rewriting translation. Hence, we further need the means to specify inter-module dependencies in an MLP, i.e., those modules from $\mathbf{P}$ that are getting called by certain modules, which in turn might call further modules from $\mathbf{P}$. Intuitively, given a module m_i of $\mathbf{P}$, we will refer to a set of modules as closure with respect to m_i when it fulfills certain conditions for calling other modules, but never call modules that do not belong to the closure. A (connected) closed call set is then based on the closure, and further requires all those modules from $\mathbf{P}$ to be included in the closed call set whenever they call modules from the closure. These notions will then be used to define instance and call rewriting below, and are necessary to rewrite modules with $\mathcal{T}(\cdot)$ in order to build a new MLP that contains $\mathbf{P}$ and the rewritten modules with their dependencies. We define now formally closure and closed call sets. In the following let $pc(m)$ denote the set of modules $\{m_j \mid P_j[p].o(\mathbf{t}) \in ma(m)\}$.

Definition 6.11 (Closure and (connected) closed call set).
For a given module m_i of an MLP $\mathbf{P}$ we define the *closure* $cl_{\mathbf{P}}(m_i)$ *of* $\mathbf{P}$ *with respect to* m_i as the smallest set S such that

- $m_i \in S$, and
- for all $m_j \in S$, $pc(m_j) \subseteq S$.

Given an MLP $\mathbf{P}$ and a module m_i of $\mathbf{P}$, a *closed call set with respect to* m_i is a set of modules S of $\mathbf{P}$ such that

- $m_i \in S$, and
- for all $m_j \in S$, $cl_{\mathbf{P}}(m_j) \subseteq S$.

The *connection graph of* $\mathbf{P}$ is the undirected graph $C_{\mathbf{P}} = (V, E)$ such that $V \subset \{m_1, \ldots, m_n\}$ consists all library modules of $\mathbf{P}$ and $E = \{(m_i, m_j) \in V \times V \mid m_j \in pc(m_i)\}$. A closed call set S is defined to be connected if S is a connected component of $C_{\mathbf{P}}$.

Note that for a given module m_i, $pc(m_i)$ only contains modules with a nonempty input list (i.e., library modules). The closure $cl_{\mathbf{P}}(m_i)$ and closed call sets based on it contain modules with empty input list only if m_i has no input. Connected closed call sets never contain modules without input, as the vertices of the connection graph $C_{\mathbf{P}}$ only contain library modules; thus, the vertices of $C_{\mathbf{P}}$ can only be a proper subset of the modules of $\mathbf{P}$, as at least one module of $\mathbf{P}$ must be a main module. The closure

$cl_{\mathbf{P}}(m_i)$ of the MLP $\mathbf{P}$ with respect to the module m_i is the least closed call set with respect to m_i.

Example 6.5 Let $\mathbf{P} = (m_1, m_2, m_3, m_4, m_5)$ be an MLP such that

$$\begin{aligned} m_1 &= (P_1[], \quad R_1 = \{a \leftarrow P_2[a].b, P_3[a].c\}) \\ m_2 &= (P_2[q_2], R_2 = \emptyset) \\ m_3 &= (P_3[q_3], R_3 = \{c \leftarrow P_2[c].b, P_4.d\}) \\ m_4 &= (P_4[], \quad R_4 = \{d \leftarrow P_2[d].b\}) \\ m_5 &= (P_5[q_5], R_5 = \{e \leftarrow P_2[e].b\}) \end{aligned}$$

Figure 6.1a shows the call structure of the modules in $\mathbf{P}$: modules represent nodes in the directed graph, and edges from module m_i to module m_j are present whenever m_i has a module call to m_j. The graph shows main modules as white nodes, while library modules are gray.

Given module m_1, the nodes inside the dotted area in Figure 6.1a represent the closure $cl_{\mathbf{P}}(m_1)$, which is the set of modules $\{m_1, m_2, m_3\}$. The nodes attached to thick edges display all the modules that are members that belong to one of $pc(m_1)$, $pc(m_2)$, and $pc(m_3)$. The reason for $m_4 \notin cl_{\mathbf{P}}(m_1)$ is that m_4 has no input list and therefore it is not captured by $pc(m_3)$, whereas module $m_5 \notin cl_{\mathbf{P}}(m_1)$ because m_5 has no incoming call from modules in $cl_{\mathbf{P}}(m_1)$.

We can now define the instance rewriting with respect to a closed call set.

Definition 6.12 (Instance rewriting).
Let $\mathbf{P} = (m_1, \dots, m_n)$ be an MLP, let m_i be a module from $\mathbf{P}$, and let $S = \{m_{i_1}, \dots, m_{i_h}\}$ be a closed call set with respect to m_i. The *instance rewriting of MLP* $\mathbf{P}$ *with respect to closed call set* S is the MLP

$$IR_{\mathbf{P}}(S) = (m_1, \dots, m_n, m'_{n+1}, \dots, m'_{n+h}) \ ,$$

where $m'_{n+j} = (P_{n+j}[], \mathcal{T}(m_{i_j}))$.

We will refer to the fresh modules m'_{n+j} from $IR_{\mathbf{P}}(S)$ as *shadow modules*, and call the set $\{m'_{n+1}, \dots, m'_{n+h}\}$ the *shadow of* $IR_{\mathbf{P}}(S)$.

Example 6.6 (cont'd) Let $\mathbf{P} = (m_1, \dots, m_5)$ be the MLP from Example 6.5 and let $S = \{m_1, m_2, m_3, m_5\}$ be a closed call set with respect to module m_1. The instance rewriting of $\mathbf{P}$ with respect to S is the MLP $IR_{\mathbf{P}}(S) = (m_1, m_2, m_3, m_4, m_5, m'_1, m'_2, m'_3, m'_5)$, where $m'_j = (P_{n+j}[], \mathcal{T}(m_j))$ (for simplicity, we will not use the index i_j in S to name m'_j). Figure 6.1b shows the inter-module dependencies of $IR_{\mathbf{P}}(S)$, where $S = cl_{\mathbf{P}}(m_1) \cup \{m_5\}$ is shown using the dash-dotted area, and the shadow modules m'_1, m'_2, m'_3, m'_5 for

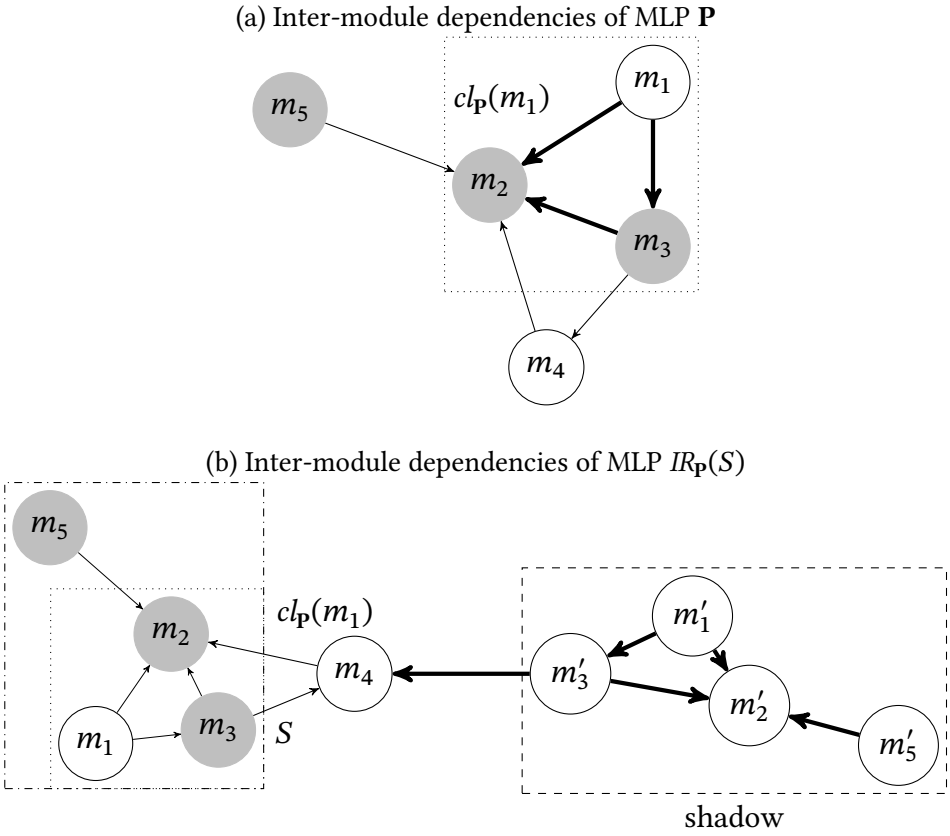


Figure 6.1: Instance Rewriting

S are within the dashed area. As can be seen from the graph, the sub-structure defined by S from the original MLP **P** is preserved in the shadow, except for the incoming call from m_4 to m'_2.

In order to show the next lemma, we define for a module $m_j = (P_j[q_j], R_j)$ the bijective function $\mathtt{bs}_j \colon \{0,1\}^{|q_j|} \to 2^{\mathrm{HB}_{\mathbf{P}}|_{q_j}}$ that sends $\mathbf{b} \in \{0,1\}^{|q_j|}$ to the set of atoms

$$\mathtt{bs}_j(\mathbf{b}) = \{q_j(\mathbf{c}) \mid \mathbf{c} \in \mathcal{C}^{|q_j|} \wedge k = \mathtt{bpos}(\mathbf{c}) \wedge b_k = 1\}$$

and $\mathtt{bs}_j^{-1}(A) = \mathbf{b}$ be its inverse function for a set of atoms $A \subseteq \mathrm{HB}_{\mathbf{P}}|_{q_j}$. Intuitively, $\mathtt{bs}_j(\mathbf{b})$ includes only those $q_j(\mathbf{c})$ where bit b_k for $\mathbf{c}$ is 1, whereas $\mathtt{bs}_j^{-1}(A)$ gives us the list of bits corresponding to the set A. We can now show the following.

Lemma 6.3
If **M** is an answer set of MLP **P** then there exists an answer set **M**′ of the MLP $IR_{\mathbf{P}}(S)$ such that **M**′ coincides with **M** on the value calls of **P**, where

- $M'_j/T = M_j/T$, for all $P_j[T] \in \mathrm{VC}(\mathbf{P})$; and
- $M'_{n+j}/\emptyset$, for $j \in \{1, \ldots, h\}$, consists of

$$\{bit(0), bit(1)\} \cup \tag{6.7}$$

$$\left\{q_{i_j}(\mathbf{c}, \mathbf{b}) \mid \mathbf{c} \in \mathcal{C}^{|q_{i_j}|} \wedge \mathbf{b} \in \{0,1\}^* \wedge b_{\mathtt{bpos}(\mathbf{c})} = 1\right\} \cup \tag{6.8}$$

$$\left\{a(\mathbf{c}, \mathbf{b}) \mid \mathbf{b} \in \{0,1\}^* \wedge T = \mathtt{bs}_{i_j}(\mathbf{b}) \wedge a(\mathbf{c}) \in M_{i_j}/T\right\} \cup \tag{6.9}$$

$$\bigcup_{P[p].o(\mathbf{t}) \in ma(m_{i_j})} \left\{val_p(\mathbf{c}, 1, \mathbf{b}) \mid \begin{array}{l} \mathbf{c} \in \mathcal{C}^{|p|} \wedge \mathbf{b} \in \{0,1\}^* \wedge \\ T = \mathtt{bs}_{i_j}(\mathbf{b}) \ \wedge p(\mathbf{c}) \in M_{i_j}/T \end{array}\right\} \cup \tag{6.10}$$

$$\bigcup_{P[p].o(\mathbf{t}) \in ma(m_{i_j})} \left\{val_p(\mathbf{c}, 0, \mathbf{b}) \mid \begin{array}{l} \mathbf{c} \in \mathcal{C}^{|p|} \wedge \mathbf{b} \in \{0,1\}^* \wedge \\ T = \mathtt{bs}_{i_j}(\mathbf{b}) \wedge p(\mathbf{c}) \notin M_{i_j}/T \end{array}\right\} . \tag{6.11}$$

Proof Let $\mathbf{M}$ be an answer set of $\mathbf{P}$. We show now that there exists an answer set $\mathbf{M}'$ of $IR_{\mathbf{P}}(S)$ such that (a) $\mathbf{M}' \vDash f\, IR_{\mathbf{P}}(S)^{\mathbf{M}'}$, and that (b) $\mathbf{M}'$ is a minimal model of $f\, IR_{\mathbf{P}}(S)^{\mathbf{M}'}$.

We begin with item a. By construction of $IR_{\mathbf{P}}(S)$, $\mathrm{CG}_{\mathbf{P}}$ is a strict subgraph of $\mathrm{CG}_{IR_{\mathbf{P}}(S)}$ such that some value calls from $\mathrm{VC}(IR_{\mathbf{P}}(S)) \setminus \mathrm{VC}(\mathbf{P})$ have edges going to $\mathrm{VC}(\mathbf{P})$, but $\mathrm{CG}_{IR_{\mathbf{P}}(S)}$ cannot have an edge from a value call in $\mathrm{VC}(\mathbf{P})$ that ends in $\mathrm{VC}(IR_{\mathbf{P}}(S))$. Thus we can split $IR_{\mathbf{P}}(S)$ into a part M'_j/T of $\mathbf{M}'$ that corresponds to $\mathbf{P}$, and a part $M'_{n+j}/\emptyset$, $1 \leq j \leq h$, of the shadow $\{m'_{n+1}, \ldots, m'_{n+h}\}$ for the closed call set $S = \{m_{i_1}, \ldots, m_{i_h}\}$.

Hence, $\mathbf{M}' \vDash f\, IR_{\mathbf{P}}(S)^{\mathbf{M}'}$, as the part that is identical to $\mathbf{M}$ satisfies $\mathbf{M}', P_j[T] \vDash f\, IR_{\mathbf{P}}(S)^{\mathbf{M}'}$ for all $P_j[T] \in \mathrm{VC}(\mathbf{P})$. Furthermore, the part $M'_{n+1}/\emptyset, \ldots, M'_{n+h}/\emptyset$ of $\mathbf{M}'$ satisfies $\mathbf{M}', P_{n+j}[\emptyset] \vDash f\, IR_{\mathbf{P}}(S)^{\mathbf{M}'}$ corresponding to shadow modules $m'_{n+j} = (P_{n+j}[], \mathcal{T}(m_{i_j}))$ for $1 \leq j \leq h$:

- for each m'_{n+j} we have that (6.7) satisfies $\{bit(0) \leftarrow, bit(1) \leftarrow\}$ from $\mathcal{T}(m_{i_j})$;
- by (6.8), all rules from $\mathcal{I}(m_{i_j})$ are satisfied: let $r' \in grnd(\mathcal{I}(m_{i_j}))$, as $\{bit(0), bit(1)\} \subseteq M'_{n+j}/\emptyset$, we have that $B(r')$ is satisfied by $\mathbf{M}'$. To see that $H(r')$ is also satisfied, consider a ground substitution θ such that $B(r') = \mathtt{bit}(\mathbf{B})\theta = \mathtt{bit}(\mathbf{b})$. Hence, for $\mathbf{c} \in \mathcal{C}^{|q_{i_j}|}$, we have that $\mathtt{bv}(\mathbf{c})\theta = \mathbf{b}$ such that $b_{\mathtt{bpos}(\mathbf{c})} = 1$. This means that $H(r') = q_{i_j}(\mathbf{c}, \mathtt{bv}(\mathbf{c}))\theta = q_{i_j}(\mathbf{c}, \mathbf{b})$, and by construction of (6.8), we get that $H(r')$ is also satisfied by $\mathbf{M}'$.

- $f\ \mathcal{VR}(m_{i_j})^{\mathbf{M}'}$ is satisfied by $\mathbf{M}'$: let p be the input predicate from module atoms appearing in m_{i_j}. In case $p(\mathbf{c}) \in M_{i_j}/T$ by (6.10) we get that $val_p(\mathbf{c}, 1, \mathbf{b}) \in M'_{n+j}/\varnothing$ such that $T = \mathtt{bs}_{i_j}(\mathbf{b})$. Otherwise, for $p(\mathbf{c}) \notin M_{i_j}/T$, we have that $val_p(\mathbf{c}, 0, \mathbf{b}) \in M'_{n+j}/\varnothing$ by (6.11) such that $T = \mathtt{bs}_{i_j}(\mathbf{b})$. By (6.9) we have $p(\mathbf{c}) \in M_{i_j}/T$ iff $p(\mathbf{c}, \mathbf{b}) \in M'_{n+j}/\varnothing$ such that $T = \mathtt{bs}_{i_j}(\mathbf{b})$. Consider $r' \in f\ \mathcal{VR}(m_{i_j})^{\mathbf{M}'}$, and a ground substitution θ such that $B(r') \supset \mathtt{bit}(\mathbf{B})\theta = \mathtt{bit}(\mathbf{b})$. As both atoms $\{bit(0), bit(1)\} \subseteq M'_{n+j}/\varnothing$, we have that $\mathtt{bit}(\mathbf{b})$ from $B(r')$ is satisfied by $\mathbf{M}'$. We distinguish two cases: (1) r' is of form $val_p(\mathbf{c}, 1, \mathbf{b}) \leftarrow p(\mathbf{c}, \mathbf{b}), \mathtt{bit}(\mathbf{b})$, or (2) r' is of form $val_p(\mathbf{c}, 0, \mathbf{b}) \leftarrow \text{not } p(\mathbf{c}, \mathbf{b}), \mathtt{bit}(\mathbf{b})$. In case (1), we have $p(\mathbf{c}, \mathbf{b}) \in M'_{n+j}/\varnothing$ from $f\ IR_{\mathbf{P}}(S)^{\mathbf{M}'}$ and therefore $p(\mathbf{c}) \in M_{i_j}/T$ by the set (6.9). Hence, by (6.10), we get that $val_p(\mathbf{c}, 1, \mathbf{b})$ is contained in $M'_{n+j}/\varnothing$, thus $\mathbf{M}', P_{n+j}[\varnothing] \vDash r'$. Otherwise, in case (2), we have $p(\mathbf{c}, \mathbf{b}) \notin M'_{n+j}/\varnothing$ from $f\ IR_{\mathbf{P}}(S)^{\mathbf{M}'}$ and therefore $p(\mathbf{c}) \notin M_{i_j}/T$ by (6.9). Thus, we can deduce from (6.11) that $val_p(\mathbf{c}, 0, \mathbf{b}) \in M'_{n+j}/\varnothing$, and so $\mathbf{M}', P_{n+j}[\varnothing] \vDash r'$. Therefore, all rules from $f\ \mathcal{VR}(m_{i_j})^{\mathbf{M}'}$ are satisfied by $\mathbf{M}'$.

- we have that (6.9)–(6.11) satisfy all rules from $f\ \mathcal{B}(m'_{n+j})^{\mathbf{M}'}$: let r' be a rule from $f\ \mathcal{B}(m'_{n+j})^{\mathbf{M}'}$ such that $\{\mathtt{bit}(\mathbf{b})\} \subseteq B^+(r')$. By construction of $\mathcal{B}(m'_{n+j})$, $r' = \mathcal{B}_{m_{i_j}}(r)$ for a ground rule $r \in grnd(R(m_{i_j}))$. As $\mathbf{M}', P_{n+j}[\varnothing] \vDash B(r')$, we need to show that $\mathbf{M}', P_{n+j}[\varnothing] \vDash H(r')$ in order to get that $\mathbf{M}', P_{n+j}[\varnothing] \vDash r'$.

  First, we show $\mathbf{M}', P_{i_j}[T] \vDash B(r)$ and $\mathbf{M}', P_{i_j}[T] \vDash H(r)$ for $T = \mathtt{bs}_{i_j}(\mathbf{b})$. Let $a \in B(r)$. We consider three cases: (1) a is of form $a(\mathbf{c})$, (2) a is of form $P_k[p].o(\mathbf{c})$, or (3) a is of form $P_k.o(\mathbf{c})$. In case (1), we can deduce that there is a corresponding atom $a(\mathbf{c}, \mathbf{b}) \in B(r')$, therefore by (6.9), we get $\mathbf{M}', P_{n+j}[\varnothing] \vDash a(\mathbf{c}, \mathbf{b})$ iff $\mathbf{M}', P_{i_j}[T] \vDash a(\mathbf{c})$. For case (2), we have a corresponding module atom $a' = P_{n+k}.o(\mathbf{c}, \mathbf{v}) \in B(r')$ such that for the ground substitution θ with $\{\mathtt{bit}(\mathbf{B})\}\theta = \{\mathtt{bit}(\mathbf{b})\} \subseteq B^+(r')$ and for the ground substitution σ mapping each variable V^a_u to v_u from $\mathbf{v}$ for $u \in \{1, \ldots, |\mathcal{C}^{|p|}|\}$ it holds that

$$\mathcal{VC}_{m_{i_j}}(a)\theta\sigma = \left\{val_p(\mathbf{c}', V^a_{\mathtt{bpos}(\mathbf{c}')}, \mathbf{b}) \mid \mathbf{c}' \in \mathcal{C}^{|p|}\right\}\sigma\ .$$

  Since $\mathbf{M}', P_{n+j}[\varnothing] \vDash v$ for all atoms $v \in \mathcal{VC}_{m_{i_j}}(a)\theta\sigma$, in case $a' \in B^+(r')$, $\mathbf{M}', P_{n+j}[\varnothing] \vDash a'$, and thus we get $\mathbf{M}', P_{n+k}[\varnothing] \vDash o(\mathbf{c}, \mathbf{v})$, whereas in case $a' \in B^-(r')$, we have $\mathbf{M}', P_{n+j}[\varnothing] \nvDash a'$ and thus we get $\mathbf{M}', P_{n+k}[\varnothing] \nvDash o(\mathbf{c}, \mathbf{v})$. As $\mathbf{M}', P_{n+k}[\varnothing] \vDash o(\mathbf{c}, \mathbf{v})$ iff $\mathbf{M}', P_k[T'] \vDash o(\mathbf{c})$ for $T' = \mathtt{bs}_k(\mathbf{v})$, we can deduce that $\mathbf{M}', P_{i_j}[T] \vDash a$ for $a \in B^+(r)$ (respectively, $\mathbf{M}', P_{i_j}[T] \nvDash a$ for $a \in B^-(r)$) with the following argument. From (6.10) (respectively, (6.11)), it follows that $p(\mathbf{c}) \in M_{i_j}/T$ iff $val_p(\mathbf{c}, 1, \mathbf{b}) \in M'_{n+j}/\varnothing$ (respectively, $p(\mathbf{c}) \notin M_{i_j}/T$ iff $val_p(\mathbf{c}, 0, \mathbf{b}) \in$

$M'_{n+j}/\emptyset$). Hence, $(M_{i_j}/T)|_p^{q_k} = \mathtt{bs}_k(\mathbf{v})$, as for all $v_u = 1$ from $\mathbf{v}$ we get $q_k(\mathbf{c}') \in \mathtt{bs}_k(\mathbf{v})$ such that $u = \mathtt{bpos}(\mathbf{c}')$, and for each $val_p(\mathbf{c}', 1, \mathbf{b}) \in M'_{n+j}/\emptyset$ we get that $p(\mathbf{c}') \in M_{i_j}/T$ and so $q_k(\mathbf{c}') \in (M_{i_j}/T)|_p^{q_k}$. Thus, $\mathbf{M}', P_{n+k}[\emptyset] \models o(\mathbf{c}, \mathbf{v})$ iff $\mathbf{M}', P_k[T'] \models o(\mathbf{c})$, and therefore $\mathbf{M}', P_{n+j}[\emptyset] \models a'$ iff $\mathbf{M}', P_{i_j}[T] \models a$. In case (3), we have that $a \in B(r')$. Now from $\mathbf{M}', P_{n+j}[\emptyset] \models a$ we immediately get that $\mathbf{M}', P_{i_j}[T] \models a$ as for any T, $(M_{i_j}/T)|_{()}^{()} = \emptyset$, and thus $o(\mathbf{c}) \in M'_k/\emptyset$ iff $\mathbf{M}', P_{n+j}[\emptyset] \models a$.

Now, as $\mathbf{M}', P_{i_j}[T] \models B(r)$, and since $\mathbf{M}$ is identical to $\mathbf{M}'$ for each $P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})$, we can deduce that $r \in f\,\mathbf{P}(P_{i_j}[T])^{\mathbf{M}}$. And since $\mathbf{M}$ is an answer set of $\mathbf{P}$, we must have that $\mathbf{M}$ and thus $\mathbf{M}'$ is a model for r as $\mathbf{M}', P_{i_j}[T] \models B(r)$, we can conclude that $\mathbf{M}', P_{i_j}[T] \models H(r)$ as was the claim. Now from this, since $T = \mathtt{bs}_{i_j}(\mathbf{b})$, we get that $\mathbf{M}', P_{n+j}[\emptyset] \models H(r')$, as for any $a(\mathbf{c}) \in H(r)$ such that $a(\mathbf{c}) \in M_{i_j}/T$, we have $a(\mathbf{c}, \mathbf{b}) \in H(r')$ by construction of $\mathcal{B}(m'_{n+j})$ and $a(\mathbf{c}, \mathbf{b}) \in M_{n+j}/\emptyset$ by (6.9).

In conclusion, we have that both $\mathbf{M}', P_{n+j}[\emptyset] \models B(r')$ and $\mathbf{M}', P_{n+j}[\emptyset] \models H(r')$, hence $\mathbf{M}', P_{n+j}[\emptyset] \models r'$ for all $r' \in f\,\mathcal{B}(m'_{n+j})^{\mathbf{M}'}$.

We consider item b next. To show that $\mathbf{M}'$ is a minimal model of $f\,IR_{\mathbf{P}}(S)^{\mathbf{M}'}$, we must ensure that there is no interpretation $\mathbf{M}''$ such that $\mathbf{M}'' < \mathbf{M}'$ and $\mathbf{M}'' \models f\,IR_{\mathbf{P}}(S)^{\mathbf{M}'}$.

Towards a contradiction, assume $\mathbf{M}''$ satisfies $f\,IR_{\mathbf{P}}(S)^{\mathbf{M}'}$. As $\mathbf{M}'' < \mathbf{M}'$, we consider the following cases: (1) for some M''_k/T with $k \leq n$ we have that $M''_k/T \subset M'_k/T$; or (2) we have that $M''_{n+j}/\emptyset \subset M'_{n+j}/\emptyset$ for some $j \in \{1, \ldots, h\}$ (recall that j corresponds to an index for a module from the closed call set $S = \{m_{i_1}, \ldots, m_{i_h}\}$).

In case (1), let $\mathbf{N}$ denote an interpretation for $\mathbf{P}$ such that $N_k/T = M''_k/T$ for all $P_k[T] \in \mathrm{VC}(\mathbf{P})$: hence we have that $\mathbf{N} < \mathbf{M}$. As the modules $m_1, \ldots, m_n$ from $IR_{\mathbf{P}}(S)$ do not call the shadow modules m'_{n+j} for $j \in \{1, \ldots, h\}$, it holds by construction of $\mathbf{M}'$ that for all $P_k[T] \in \mathrm{VC}(\mathbf{P})$, $f\,\mathbf{P}(P_k[T])^{\mathbf{M}} = f\,IR_{\mathbf{P}}(S)(P_k[T])^{\mathbf{M}'}$ and $\mathbf{N}, P_k[T] \models f\,\mathbf{P}(P_k[T])^{\mathbf{M}}$; it follows that $\mathbf{N} \models f\,\mathbf{P}^{\mathbf{M}}$, which contradicts minimality.

In case (2), let $\mathbf{N}$ denote the interpretation for $\mathbf{P}$ such that for all $P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})$ with $m_{i_j} \in S$, we set

$$N_{i_j}/T = \{a(\mathbf{c}) \in \mathrm{HB}_{\mathbf{P}} \mid a(\mathbf{c}, \mathbf{b}) \in M''_{n+j}/\emptyset \wedge \mathtt{bs}_k(\mathbf{b}) = T\} \tag{6.12}$$

and $N_k/T = M_k/T$ for all modules m_k from $\mathbf{P}$ such that $m_k \notin S$. From our assumption that $\mathbf{M}''$ satisfies $f\,IR_{\mathbf{P}}(S)^{\mathbf{M}'}$, if $\{bit(0), bit(1)\} \nsubseteq M''_{n+j}/\emptyset$ or there exists an atom $a \in M'_{n+j}/\emptyset$ of form $val_p(\mathbf{c}, 0, \mathbf{b})$ or $val_p(\mathbf{c}, 1, \mathbf{b})$ such that $a \notin M''_{n+j}/\emptyset$, then we get a contradiction for $\mathbf{M}'' \models f\,IR_{\mathbf{P}}(S)^{\mathbf{M}'}$, as $\mathbf{M}''$ would not satisfy all the rules from

$\{bit(0) \leftarrow, bit(1) \leftarrow\}$ respectively $f\ \mathcal{VR}(m_{i_j})^{\mathbf{M}'}$ of $\mathcal{T}(m_{i_j})$. Therefore $\mathbf{N} < \mathbf{M}$, as there must be an atom $a(\mathbf{c}) \in M_{i_j}/T$ such that $a(\mathbf{c}) \notin N_{i_j}/T$, which follows from $\mathbf{M}'' < \mathbf{M}'$ in case (2) and the construction of $\mathbf{N}$ in (6.12). For $\mathbf{b} = \mathtt{bs}_{i_j}^{-1}(T)$, we have now that atom $a(\mathbf{c}, \mathbf{b}) \in M'_{n+j}/\varnothing$ and $a(\mathbf{c}, \mathbf{b}) \notin M''_{n+j}/\varnothing$. This atom $a(\mathbf{c}, \mathbf{b})$ must be from (6.9). Since $\mathbf{M}$ is a minimal model of $f\,\mathbf{P}^{\mathbf{M}}$ and $\mathbf{N} < \mathbf{M}$, we must have that $\mathbf{N} \not\models f\,\mathbf{P}^{\mathbf{M}}$. Hence, there is a rule $r \in f\,\mathbf{P}(P_{i_j}[T])^{\mathbf{M}}$ such that $\mathbf{N} \not\models r$, thus $\mathbf{N}, P_{i_j}[T] \vDash B(r)$ and $\mathbf{N}, P_{i_j}[T] \not\models H(r)$. As $\mathbf{M}'$ is a model of $f\,IR_{\mathbf{P}}(S)^{\mathbf{M}'}$ and by construction of $\mathbf{M}'$, there must exist a rule $r' \in f\,IR_{\mathbf{P}}(S)(P_{n+j}[\varnothing])^{\mathbf{M}'}$ such that for the ground substitution θ with $\{\mathtt{bit}(\mathbf{B})\}\theta = \{\mathtt{bit}(\mathbf{b})\}$, we have that $r' \in grnd(\mathcal{B}_{m_{i_j}}(r)\theta)$. As $\mathbf{N}, P_{i_j}[T] \vDash B(r)$, we get by construction of $\mathbf{N}$ that $\mathbf{M}'', P_{n+j}[\varnothing] \vDash B(r')$, and as $\mathbf{N}, P_{i_j}[T] \not\models H(r)$ we can derive that $\mathbf{M}'', P_{n+j}[\varnothing] \not\models H(r')$. Therefore, $\mathbf{M}'', P_{n+j}[\varnothing] \not\models r'$. But since $r' \in f\,IR_{\mathbf{P}}(S)(P_{n+j}[\varnothing])^{\mathbf{M}'}$, we conclude $\mathbf{M}'' \not\models f\,IR_{\mathbf{P}}(S)^{\mathbf{M}'}$, which contradicts our assumption that $\mathbf{M}''$ is a model for $f\,IR_{\mathbf{P}}(S)^{\mathbf{M}'}$. We therefore deduce that $\mathbf{M}'$ is a minimal model for $f\,IR_{\mathbf{P}}(S)^{\mathbf{M}'}$. □

6.3.2 Call Rewriting

This section is concerned with the call rewriting translation for MLPs, which is an adaption of instance rewriting that allows us to completely isolate the modules from a connected closed call set S with modules that are not contained in S. In the following, we let $\mathbf{P} = (m_1, \ldots, m_n)$ be an MLP, $m_i = (P_i[\mathbf{q}_i], R_i)$ be a module from $\mathbf{P}$ such that $|\mathbf{q}_i| \leq 1$, and $S = \{m_{i_1}, \ldots, m_{i_h}\}$ is a connected closed call set with respect to m_i (recall Definition 6.11).

Definition 6.13 (Value rules).
Let m be a module of $\mathbf{P}$, we define the *value rules of m with respect to a connected closed call set S* as the set of rules

$$\mathcal{VR}^S(m) = \bigcup_{m_{i_j} \in S} \bigcup_{P_{i_j}[p].o(\mathbf{t}) \in ma(m)} \mathcal{VR}_m(P_{i_j}[p].o(\mathbf{t})) \ .$$

Note that we only consider module calls $P_{i_j}[p].o(\mathbf{t})$ with one input parameter, i.e., for m calling library modules $m_{i_j} \in S$.

Next, we define call redirection, which replaces calls to modules from the connected closed call set S with calls to their accompanying shadow modules.

Definition 6.14 (Call redirection).
Given an atom a appearing in a module $m = (P[], R)$ without input parameters, we define

$$\mathcal{C}^S_m(a) = \begin{cases} a & \begin{array}{l} a \text{ is of form } p(\mathbf{t}) \text{ or of form } P_k[\mathbf{p}].o(\mathbf{t}), |\mathbf{p}| \leq 1, \\ \text{such that } m_k \notin S \ , \end{array} \\ P_{n+j}.o(\mathbf{t}, \mathbf{V}^a) & \begin{array}{l} a \text{ is of form } P_{i_j}[\mathbf{p}].o(\mathbf{t}), |\mathbf{p}| \leq 1, \text{ such that } m_{i_j} \in S, \\ \text{where } \mathbf{V}^a \text{ is a list of distinct variables of length } |\mathcal{C}^{|\mathbf{p}|}| \ . \end{array} \end{cases}$$

Given a rule $r \in R$ of form (3.2), we can now define the rule

$$\begin{aligned} \mathcal{C}^S_m(r) \ = \ \alpha_1 \vee \cdots \vee \alpha_k \leftarrow \ & \mathcal{C}^S_m(\beta_1), \dots, \mathcal{C}^S_m(\beta_m), \text{not } \mathcal{C}^S_m(\beta_{m+1}), \dots, \text{not } \mathcal{C}^S_m(\beta_n), \\ & \mathcal{VC}_m(e_1), \dots, \mathcal{VC}_m(e_u) \ , \end{aligned}$$

where $e_1, \dots, e_u$ are those module atoms from $ma(r)$ such that e_k is of form $P_{i_j}[p].o(\mathbf{t})$ with $m_{i_j} \in S$. For a module m, we let

$$\mathcal{C}^S(m) = \{\mathcal{C}^S_m(r) \mid r \in R(m)\} \ .$$

Note that we only consider module calls $e_1, \dots, e_u$ in $\mathcal{C}^S_m(r)$ that have an input parameter, which are library modules by definition. Module calls e_k to input-less (main or library) modules from S do not need to be guarded with $\mathcal{VC}_m(e)$, as $\mathcal{C}^S_m(e)$ is already fixed and does not need to select the right module instance for accessing the output atom. Such modules can only occur in S if they are used to build a closed call set; for instance, module m_1 from Example 6.5 is such a module.

The module call rewriting of module m and connected closed call set S now combines $\mathcal{VR}^S(m)$ and $\mathcal{C}^S(m)$.

Definition 6.15 (Module call rewriting).
For a module m and a connected closed call set S, we let the *module call rewriting* $\mathcal{TC}^S(m)$ be the following set of rules

$$\mathcal{TC}^S(m) = \mathcal{VR}^S(m) \cup \mathcal{C}^S(m) \ .$$

Next, we formally define the access set of a connected closed call set S.

Definition 6.16 (Access set of a connected closed call set).
Let $\mathbf{P}$ be an MLP, let m_i be a module of $\mathbf{P}$, and let S be a connected closed call set of $\mathbf{P}$. We define the *access set of S with respect to m_i* as the set of modules

$$acc_{\mathbf{P}}(S) = \left\{ m_k \mid \begin{array}{l} m_k = (P_k[], R_k) \text{ appears in } \mathbf{P} \wedge m_k \notin S \wedge \\ m_{i_j} \in S \wedge P_{i_j}[p].o(\mathbf{t}) \in ma(m_k) \end{array} \right\} \ .$$

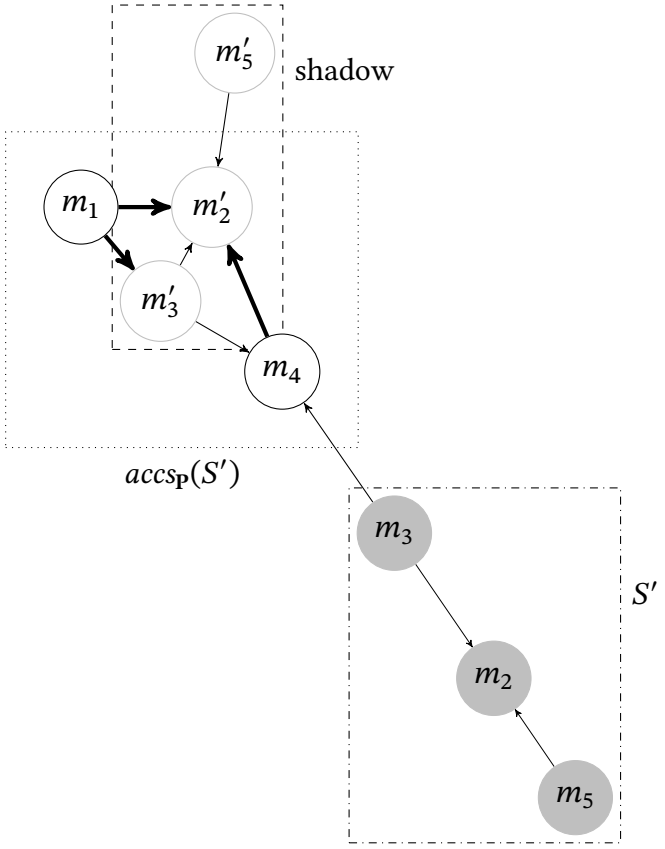


Figure 6.2: Call Rewriting

Intuitively, the access set gives us all modules m_k without input parameters that are not contained in S, such that m_k calls modules from S. Thus we may rewrite the calls from m_k to S using module call rewriting to access the shadow modules instead of the modules of S.

Example 6.7 Let $\mathbf{P}$ be the MLP from Example 6.5. The connection graph $C_{\mathbf{P}} = (V, E)$ has the set of vertices $V = \{m_2, m_3, m_5\}$ and the edges $E = \{(m_3, m_2), (m_5, m_2)\}$. Thus, $C_{\mathbf{P}}$ consists of one connected component $S' = \{m_2, m_3, m_5\}$, which will be the connected closed call set we use for the call rewriting. Then, we get that $accs_{\mathbf{P}}(S') = \{m_1, m_4\}$, as m_1 calls both m_2 and m_3, and m_4 calls m_2.

We can now define the call rewriting with respect to a connected closed call set.

Definition 6.17 (Call rewriting).
Let $\mathbf{P} = (m_1, \dots, m_n)$ be an MLP such that for a module $m_i = (P_i[\mathbf{q}_i], R_i)$ from $\mathbf{P}$ we have $|\mathbf{q}_i| \leq 1$ and $S = \{m_{i_1}, \dots, m_{i_h}\}$ is a connected closed call set with respect to m_i. We define the *call rewriting of MLP* $\mathbf{P}$ *with respect to connected closed call set* S as the

MLP

$$CR_{\mathbf{P}}(S) = (m'_1, \ldots, m'_n, m'_{n+1}, \ldots, m'_{n+h}) \ ,$$

where for $k \in \{1, \ldots, n\}$,

$$m'_k = \begin{cases} (P_k[], \mathcal{TC}^S(m_k)) & \text{if } m_k \in accs_{\mathbf{P}}(S), \\ m_k & \text{otherwise,} \end{cases}$$

and $m'_{n+j} = (P_{n+j}[], \mathcal{T}(m_{i_j}))$ for $j \in \{1, \ldots, h\}$.

Intuitively, we leave modules $m_{i_j} \in S$ untouched, and clone them as shadow modules m'_{n+j}, $j \in \{1, \ldots, h\}$, by applying instance rewriting $\mathcal{T}(m_{i_j})$ on them. Then, modules $m_k \in accs_{\mathbf{P}}(S)$ will be rewritten to call the shadow modules instead of modules from S.

To show the next lemma we define for each M_k/T from $\mathbf{M}$ the set $val(M_k/T)$ consisting of

$$val(M_k/T) = \bigcup_{m_{i_j} \in S} \bigcup_{P_{i_j}[p].o(\mathbf{t}) \in ma(m_k)} \left\{ val_p(\mathbf{c}, 1, \mathbf{b}) \mid \begin{array}{l} \mathbf{c} \in \mathcal{C}^{|p|} \wedge \\ p(\mathbf{c}) \in M_k/T \wedge \\ \mathtt{bs}_k^{-1}(T) = \mathbf{b} \end{array} \right\} \cup \tag{6.13}$$

$$\bigcup_{m_{i_j} \in S} \bigcup_{P_{i_j}[p].o(\mathbf{t}) \in ma(m_k)} \left\{ val_p(\mathbf{c}, 0, \mathbf{b}) \mid \begin{array}{l} \mathbf{c} \in \mathcal{C}^{|p|} \wedge \\ p(\mathbf{c}) \notin M_k/T \wedge \\ \mathtt{bs}_k^{-1}(T) = \mathbf{b} \end{array} \right\} \tag{6.14}$$

and $flat(M_k/T)$ be the set

$$flat(M_k/T) = \{a(\mathbf{c}, \mathbf{b}) \mid a(\mathbf{c}) \in M_k/T \wedge \mathtt{bs}_k^{-1}(T) = \mathbf{b}\} \ . \tag{6.15}$$

For a module $m_{i_j} \in S$ from $IR_{\mathbf{P}}(S)$ such that $P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})$ and a model $M_{n+j}/\emptyset$ from $\mathbf{M}$ for the shadow module m'_{n+j} of $IR_{\mathbf{P}}(S)$, we let $lift(M_{n+j}/\emptyset, T)$ be the set

$$lift(M_{n+j}/\emptyset, T) = \{a(\mathbf{c}) \in \mathrm{HB}_{\mathbf{P}} \mid a(\mathbf{c}, \mathbf{b}) \in M_{n+j}/\emptyset \wedge \mathtt{bs}_{i_j}(\mathbf{b}) = T\} \ . \tag{6.16}$$

We can now show the following.

Lemma 6.4
The answer sets of the MLP $IR_{\mathbf{P}}(S)$ correspond one-to-one to the answer sets of the MLP $CR_{\mathbf{P}}(S)$, that is,

- for each answer set $\mathbf{M}$ of $IR_{\mathbf{P}}(S)$ there exists an answer set $\mathbf{M}'$ for $CR_{\mathbf{P}}(S)$ such that

  - $M'_k/T = M_k/T$ for $P_k[T] \in \mathrm{VC}(\mathbf{P})$ such that $m_k \notin S \cup accs_{\mathbf{P}}(S)$;
  - $M'_{i_j}/T = lift(M_{n+j}/\emptyset, T)$ for all $P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})$ such that $m_{i_j} \in S$;
  - for each module $m_{i_j} \in S$,

$$M'_{n+j}/\emptyset = \{bit(0), bit(1)\} \cup \bigcup_{P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})} \Big(val(M_{i_j}/T) \cup flat(M_{i_j}/T) \Big) \ ;$$

  - $M'_k/\emptyset = M_k/\emptyset \cup val(M_k/\emptyset)$ for each module $m_k \in accs_{\mathbf{P}}(S)$; and

- for each answer set $\mathbf{M}'$ of $CR_{\mathbf{P}}(S)$ there exists an answer set $\mathbf{M}$ of $IR_{\mathbf{P}}(S)$ such that

  - $M_k/T = M'_k/T$ for $P_k[T] \in \mathrm{VC}(\mathbf{P})$ such that $m_k \notin S \cup accs_{\mathbf{P}}(S)$;
  - $M_{i_j}/T = lift(M'_{n+j}/\emptyset, T)$ for all $P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})$ such that $m_{i_j} \in S$;
  - for each module $m_{i_j} \in S$,

$$M_{n+j}/\emptyset = \{bit(0), bit(1)\} \cup \bigcup_{P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})} \Big(val(M'_{i_j}/T) \cup flat(M'_{i_j}/T) \Big) \ ;$$

  - $M_k/\emptyset = M'_k/\emptyset \cup val(M'_k/\emptyset)$ for each module $m_k \in accs_{\mathbf{P}}(S)$.

Proof In case $accs_{\mathbf{P}}(S) = \emptyset$, i.e., there are no modules outside S that call modules from S, it follows that $IR_{\mathbf{P}}(S) = CR_{\mathbf{P}}(S)$, thus our claim holds. For the case that $accs_{\mathbf{P}}(S)$ contains at least one module m_k we show now that for each answer set of $IR_{\mathbf{P}}(S)$ there is a corresponding answer set of $CR_{\mathbf{P}}(S)$, and vice versa. Intuitively, an answer set of $IR_{\mathbf{P}}(S)$ can be converted to an answer set of $CR_{\mathbf{P}}(S)$ by exchanging the part of the answer set that correspond to S with the part that correspond to the shadow modules. The same holds when converting answer sets of $CR_{\mathbf{P}}(S)$ to answer sets of $IR_{\mathbf{P}}(S)$.

($\Rightarrow$) Let $\mathbf{M}$ be an answer set of $IR_{\mathbf{P}}(S)$. We show now that there exists an answer set $\mathbf{M}'$ of $CR_{\mathbf{P}}(S)$ such that (a) $\mathbf{M}' \vDash f\, CR_{\mathbf{P}}(S)^{\mathbf{M}'}$, and that (b) $\mathbf{M}'$ is a minimal model of $f\, CR_{\mathbf{P}}(S)^{\mathbf{M}'}$. Intuitively, $\mathbf{M}'$ is the result of swapping the part of $\mathbf{M}$ for the shadow modules with the part of $\mathbf{M}$ for the modules from S, while modules from $accs_{\mathbf{P}}(S)$ get additional atoms of form $val_p(\mathbf{c}, 1)$ (respectively, $val_p(\mathbf{c}, 0)$); modules that belong to neither part are kept the same.

Let us start with item a. In case module $m_k \notin S \cup accs_{\mathbf{P}}(S)$ for value calls $P_k[T] \in \mathrm{VC}(\mathbf{P})$, we have that $M'_k/T = M_k/T$ and $m'_k = m_k$, thus

$$f\, CR_{\mathbf{P}}(S)(P_k[T])^{\mathbf{M}'} = f\, IR_{\mathbf{P}}(S)(P_k[T])^{\mathbf{M}} \ ,$$

and since $\mathbf{M}$ satisfies $f\,IR_{\mathbf{P}}(S)^{\mathbf{M}}$, we get that $\mathbf{M}', P_k[T] \models f\,CR_{\mathbf{P}}(S)(P_k[T])^{\mathbf{M}'}$.

In case that a module $m_{i_j} \in S$, it holds that $M'_{i_j}/T = lift(M_{n+j}/\emptyset, T)$ for all $P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})$ such that $m_{i_j} \in S$. As $\mathbf{M}$ is an answer set for $IR_{\mathbf{P}}(S)$, we have that $\mathbf{M} \models f\,IR_{\mathbf{P}}(S)^{\mathbf{M}}$, and in particular $\mathbf{M}, P_{n+j}[\emptyset] \models f\,IR_{\mathbf{P}}(S)(P_{n+j}[\emptyset])^{\mathbf{M}}$. We show now that $\mathbf{M}', P_{i_j}[T] \models f\,CR_{\mathbf{P}}(S)(P_{i_j}[T])^{\mathbf{M}'}$ for all $P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})$ such that $m_{i_j} \in S$. Let $r \in f\,IR_{\mathbf{P}}(S)(P_{n+j}[\emptyset])^{\mathbf{M}}$, therefore $\mathbf{M}, P_{n+j}[\emptyset] \models B(r)$ as well as $\mathbf{M}, P_{n+j}[\emptyset] \models r$. We distinguish the following cases:

- In case r is from $\mathcal{I}(m_{i_j})$, we have that r is of form $q_{i_j}(\mathbf{c}, \mathbf{b}) \leftarrow \mathtt{bit}(\mathbf{b})$. Thus, for $T = \mathtt{bs}_{i_j}(\mathbf{b})$, we get $q_{i_j}(\mathbf{c}) \in lift(M_{n+j}/\emptyset, T)$ and so $q_{i_j}(\mathbf{c}) \in M'_{i_j}/T$, which is a requirement for $\mathbf{M}'$ being a model for $f\,CR_{\mathbf{P}}(S)(P_{i_j}[T])^{\mathbf{M}'}$.

- For the case that r is from $\mathcal{B}(m_{i_j})$, we have that r is of form (6.6). Thus, for $\{\mathtt{bit}(\mathbf{b})\} \subseteq B(r)$ such that $T = \mathtt{bs}_{i_j}(\mathbf{b})$, we show now that there is a rule $r' \in grnd(R(m_{i_j}))$ such that $r = \mathcal{B}_{m_{i_j}}(r')$, and $r' \in f\,CR_{\mathbf{P}}(S)(P_{i_j}[T])^{\mathbf{M}'}$. The rule r' must be of form (3.2), i.e., whenever there is an inflated atom $a(\mathbf{c}, \mathbf{b}) \in H(r) \cup B(r)$, we have $a(\mathbf{c}) \in H(r') \cup B(r')$, and for module atoms $P_{n+k}.o(\mathbf{c}, \mathbf{v}) \in B(r)$ (respectively, $P_k.o(\mathbf{c}) \in B(r)$) we have the corresponding $P_k[p].o(\mathbf{c}) \in B(r')$ (respectively, $P_k.o(\mathbf{c}) \in B(r')$). By construction of $M'_{i_j}/T = lift(M_{n+j}/\emptyset, T)$, we can deduce that $a(\mathbf{c}) \in M'_{i_j}/T$ iff $a(\mathbf{c}, \mathbf{b}) \in M_{n+j}/\emptyset$ such that $a(\mathbf{c}) \in \mathrm{HB}_{\mathbf{P}}$. Since $\mathbf{M}, P_{n+j}[\emptyset] \models B(r)$, we get for ordinary atoms $a(\mathbf{c}) \in B^+(r')$ that $\mathbf{M}', P_{i_j}[T] \models a(\mathbf{c})$ (respectively, for ordinary atoms $a(\mathbf{c}) \in B^-(r')$ that $\mathbf{M}', P_{i_j}[T] \not\models a(\mathbf{c})$). For module atoms $a' = P_k.o(\mathbf{c}) \in B^+(r')$ we get that $\mathbf{M}', P_{i_j}[T] \models a'$ (respectively, for $a' \in B^-(r')$ that $\mathbf{M}', P_{i_j}[T] \not\models a'$), as

$$M_k/(M_{n+j}/\emptyset)|^{()}_{()} = M_k/(M'_{i_j}/T)|^{()}_{()} = M_k/\emptyset$$

for any $P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})$. Considering module atoms $a' = P_k[p].o(\mathbf{c}) \in B(r')$, we get that $a = P_{n+k}.o(\mathbf{c}, \mathbf{v}) \in B(r)$ for the ground substitution σ mapping each variable V^a_u to v_u from $\mathbf{v}$ for $u \in \{1, \ldots, |\mathcal{C}^{|p|}|\}$ such that

$$\mathcal{VC}_{m_{i_j}}(a')\sigma = \left\{ val_p(\mathbf{c}', V^a_{\mathtt{bpos}(\mathbf{c}')}) \mid \mathbf{c}' \in \mathcal{C}^{|p|} \right\}\sigma \subseteq B^+(r) \ .$$

It holds that $o(\mathbf{c}, \mathbf{v}) \in M_{n+k}/(M_{n+j}/\emptyset)|^{()}_{()} = M_{n+k}/\emptyset$ for an $a \in B^+(r)$ (respectively, $o(\mathbf{c}, \mathbf{v}) \notin M_{n+k}/\emptyset$ for an $a \in B^-(r)$). Thus, for the module $m_{i_k} \in S$ and for $T' = \mathtt{bs}_{i_k}(\mathbf{v})$, we have that $o(\mathbf{c}) \in M'_{i_k}/T'$ iff $a \in B^+(r)$. Now as $\mathbf{M}, P_{n+j}[\emptyset] \models v$ for all $v \in \mathcal{VC}_{m_{i_j}}(a')\sigma$, we get that for $p(\mathbf{c}', \mathbf{b}) \in M_{n+j}/\emptyset$ iff $v_u = 1$ for v_u from $\mathbf{v}$ such that $u = \mathtt{bpos}(\mathbf{c}')$, and therefore, by construction of

$lift(M_{n+j}/\emptyset, T)$, we get $p(\mathbf{c}') \in M'_{i_j}/T$ iff $v_u = 1$. Hence, $q_k(\mathbf{c}') \in (M'_{i_j}/T)|_p^{q_k}$ iff $v_u = 1$, which means that

$$(M'_{i_j}/T)|_p^{q_k} = \mathtt{bs}_{i_k}(\mathbf{v}) = T' \ .$$

Therefore, we can now link $P_{i_j}[T]$ to $P_{i_k}[T']$ in $\mathbf{M}'$, as it holds in case $a' \in B^+(r')$ that $\mathbf{M}', P_{i_j}[T] \models a'$, and for $a' \in B^-(r')$, we have that $\mathbf{M}', P_{i_j}[T] \not\models a'$.

Therefore, $\mathbf{M}', P_{i_j}[T] \models B(r')$, and we get that $r' \in f\ CR_{\mathbf{P}}(S)(P_{i_j}[T])^{\mathbf{M}'}$. By construction of $\mathbf{M}'$ and as $\mathbf{M}, P_{n+j}[\emptyset] \models H(r)$, we have for $a(\mathbf{c}) \in H(r')$ such that $\mathbf{M}, P_{n+j}[\emptyset] \models a(\mathbf{c}, \mathbf{b})$ that $\mathbf{M}', P_{i_j}[T] \models a(\mathbf{c})$. Thus, $\mathbf{M}', P_{i_j}[T] \models H(r')$, and so we deduce $\mathbf{M}', P_{i_j}[T] \models r'$.

Hence, $\mathbf{M}', P_{i_j}[T] \models f\ CR_{\mathbf{P}}(S)(P_{i_j}[T])^{\mathbf{M}'}$ for all value calls $P_{i_j}[T]$ such that $m_{i_j} \in S$.

Next, we consider value calls $P_{n+j}[\emptyset]$ for $m_{i_j} \in S$. In this case, we have that $M'_{n+j}/\emptyset = \bigcup_{P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})} val(M_{i_j}/T) \cup flat(M_{i_j}/T)$. As $\mathbf{M}$ is an answer set for $IR_{\mathbf{P}}(S)$, we have that $\mathbf{M} \models f\ IR_{\mathbf{P}}(S)^{\mathbf{M}}$, and in particular $\mathbf{M}, P_{i_j}[T] \models f\ IR_{\mathbf{P}}(S)(P_{i_j}[T])^{\mathbf{M}}$ for all $P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})$ such that $m_{i_j} \in S$. In $CR_{\mathbf{P}}(S)$, we have $m'_{i_j} = m_{i_j}$ and $m'_{n+j} = (P_{n+j}[], \mathcal{T}(m_{i_j}))$, just as in $IR_{\mathbf{P}}(S)$. To show that $\mathbf{M}', P_{n+j}[\emptyset] \models f\ CR_{\mathbf{P}}(S)(P_{n+j}[\emptyset])^{\mathbf{M}'}$, we can reuse the argument from part (a) of the proof for Lemma 6.3, which works *mutatis mutandis* by considering $\mathbf{M}'$ as defined here and showing that each part from $\mathcal{T}(m_{i_j})$ of $f\ CR_{\mathbf{P}}(S)(P_{n+j}[S])^{\mathbf{M}'}$ is satisfied by $\mathbf{M}'$ accordingly, that is, the part $\mathcal{I}(m_{i_j})$, $f\ \mathcal{VR}(m_{i_j})^{\mathbf{M}'}$, and $f\ \mathcal{B}(m'_{n+j})^{\mathbf{M}'}$.

The last case considers the modules m_k from $accs_{\mathbf{P}}(S)$. Here, $M'_k/\emptyset = M_k/\emptyset \cup val(M_k/\emptyset)$, and $CR_{\mathbf{P}}(S)$ defines $m'_k = (P_k[], \mathcal{TC}^S(m_k))$. As $\mathbf{M}$ is an answer set for $IR_{\mathbf{P}}(S)$, we have that $\mathbf{M} \models f\ IR_{\mathbf{P}}(S)^{\mathbf{M}}$, and in particular $\mathbf{M}, P_k[\emptyset] \models f\ IR_{\mathbf{P}}(S)(P_k[\emptyset])^{\mathbf{M}}$ such that $m_k \in accs_{\mathbf{P}}(S)$. We show now that $\mathbf{M}', P_k[\emptyset] \models f\ CR_{\mathbf{P}}(S)(P_k[\emptyset])^{\mathbf{M}'}$.

- the atoms from (6.13) and (6.14) satisfy $f\ \mathcal{VR}^S(m_k)^{\mathbf{M}'}$: let $r' \in f\ \mathcal{VR}^S(m_k)^{\mathbf{M}'}$, thus $\mathbf{M}', P_k[\emptyset] \models B(r')$. By construction of $\mathcal{VR}^S(m)$, we get that r' is of form $val_p(\mathbf{c}, 1) \leftarrow p(\mathbf{c})$ in case $p(\mathbf{c}) \in M'_k/\emptyset$, otherwise r' is of form $val_p(\mathbf{c}, 0) \leftarrow p(\mathbf{c})$ for $p(\mathbf{c}) \notin M'_k/\emptyset$. In the former case, we get from (6.13) that $val_p(\mathbf{c}, 1) \in M'_k/\emptyset$, while the latter allows us to conclude from (6.14) that $val_p(\mathbf{c}, 0) \in M'_k/\emptyset$. Therefore we can derive that $\mathbf{M}', P_k[\emptyset] \models H(r')$ and so we have that $\mathbf{M}', P_k[\emptyset] \models r'$ for all rules $r' \in f\ \mathcal{VR}^S(m_k)^{\mathbf{M}'}$.

- all $f\ \mathcal{C}^S(m_k)^{\mathbf{M}'}$ are satisfied: let $r' \in f\ \mathcal{C}^S(m_k)^{\mathbf{M}'}$, thus $\mathbf{M}', P_k[\emptyset] \models B(r')$. We show now that $r \in f\ IR_{\mathbf{P}}(S)(P_k[\emptyset])^{\mathbf{M}}$ such that $r' = \mathcal{C}^S_{m_k}(r)$. Let $a \in B(r)$. As

$\mathcal{C}^S_{m_k}(a) = a$ for all ordinary atoms a or for module atoms $a = P_j[p].o(\mathbf{c})$ such that $m_j \notin S$, we get that $\mathbf{M}, P_k[\varnothing] \vDash a$ iff $\mathbf{M}', P_k[\varnothing] \vDash a$. For module atoms $a \in B(r)$ such that $a = P_{i_j}[p].o(\mathbf{c})$ with $m_{i_j} \in S$, we have that $\mathcal{C}^S_{m_k}(a) = a' = P_{n+j}.o(\mathbf{c}, \mathbf{v})$ for the ground substitution σ mapping each variable V^a_u to v_u from $\mathbf{v}$ for $u \in \{1, \ldots, |\mathcal{C}^{|p|}|\}$ such that $\mathcal{VC}_{m_k}(a)\sigma = \left\{val_p(\mathbf{c}', V^a_{\mathtt{bpos}(\mathbf{c}')}) \mid \mathbf{c}' \in \mathcal{C}^{|p|}\right\}\sigma$. As $\mathbf{M}', P_k[\varnothing] \vDash B(r')$, we distinguish: (1) $a' \in B^+(r')$, or (2) $a' \in B^-(r')$.

In case (1), $\mathbf{M}', P_k[\varnothing] \vDash a'$, hence $o(\mathbf{c}, \mathbf{v}) \in M'_{n+j}/\varnothing$ such that $T' = \mathtt{bs}_{i_j}(\mathbf{v})$. We must have that $o(\mathbf{c}, \mathbf{v}) \in \mathit{flat}(M_{i_j}/T')$ by construction of $M'_{n+j}/\varnothing$, hence $o(\mathbf{c}) \in M_{i_j}/T'$. Since $q_{i_j}(\mathbf{c}') \in T'$ iff $v_u = 1$ such that $u = \mathtt{bpos}(\mathbf{c}')$, we get by (6.13) and (6.14) that $T' = (M'_k/\varnothing)|_p^{q_{i_j}}$, as whenever $p(\mathbf{c}') \in M'_k/\varnothing$ we have $val_p(\mathbf{c}', 1) \in M'_k/\varnothing$, and for $p(\mathbf{c}') \notin M'_k/\varnothing$ we have $val_p(\mathbf{c}', 0) \in M'_k/\varnothing$, thus encoding T' by means of $\mathbf{v}$. But since $M_k/\varnothing \subseteq M'_k/\varnothing$ and both differ only in atoms of form $val_p(\mathbf{c}', 0)$ and $val_p(\mathbf{c}', 0)$, we have that $(M'_k/\varnothing)|_p^{q_{i_j}} = (M_k/\varnothing)|_p^{q_{i_j}}$, and thus $T' = (M_k/\varnothing)|_p^{q_{i_j}}$, whence we conclude that $\mathbf{M}, P_k[\varnothing] \vDash a$.

In case (2), $\mathbf{M}', P_k[\varnothing] \nvDash a'$, hence $o(\mathbf{c}, \mathbf{v}) \notin M'_{n+j}/\varnothing$ such that $T' = \mathtt{bs}_{i_j}(\mathbf{v})$. We must have that $o(\mathbf{c}, \mathbf{v}) \notin \mathit{flat}(M_{i_j}/T')$ by construction of $M'_{n+j}/\varnothing$, hence $o(\mathbf{c}) \notin M_{i_j}/T'$. With a similar argument as in case (1), we can deduce that $\mathbf{M}, P_k[\varnothing] \nvDash a$.

Therefore, $\mathbf{M}, P_k[\varnothing] \vDash B(r)$, and so $r \in f\,IR_{\mathbf{P}}(S)(P_k[\varnothing])^{\mathbf{M}}$. Now as $\mathbf{M}$ satisfies $\mathbf{M} \vDash f\,IR_{\mathbf{P}}(S)^{\mathbf{M}}$, we must have that $\mathbf{M}, P_k[\varnothing] \vDash H(r)$, and since $H(r') = H(r)$, we get $\mathbf{M}', P_k[\varnothing] \vDash H(r')$ and thus $\mathbf{M}', P_k[\varnothing] \vDash r'$. Therefore, we can deduce that $\mathbf{M}', P_k[\varnothing] \vDash f\,\mathcal{C}^S(m_k)^{\mathbf{M}'}$.

We can derive now that for all value calls $P_k[T] \in \mathrm{VC}(CR_{\mathbf{P}}(S))$ we have that $\mathbf{M}', P_k[T] \vDash f\,CR_{\mathbf{P}}(S)^{\mathbf{M}'}$, therefore $\mathbf{M}' \vDash f\,CR_{\mathbf{P}}(S)^{\mathbf{M}'}$.

We turn our attention to item b. To show that $\mathbf{M}'$ is a minimal model of $f\,CR_{\mathbf{P}}(S)^{\mathbf{M}'}$, we must ensure that there is no interpretation $\mathbf{M}''$ such that $\mathbf{M}'' < \mathbf{M}'$ and $\mathbf{M}'' \vDash f\,CR_{\mathbf{P}}(S)^{\mathbf{M}'}$.

Towards a contradiction, assume $\mathbf{M}''$ satisfies $f\,CR_{\mathbf{P}}(S)^{\mathbf{M}'}$. As $\mathbf{M}'' < \mathbf{M}'$, we consider the following cases:

1. for some M''_k/T such that $m_k \notin S \cup \mathit{accs}_{\mathbf{P}}(S)$ we have that $M''_k/T \subset M'_k/T$;
2. for some $M''_k/\varnothing$ such that $m_k \in \mathit{accs}_{\mathbf{P}}(S)$ we have that $M''_k/\varnothing \subset M'_k/\varnothing$;
3. for some M''_{i_j}/T such that for $m_{i_j} \in S$ and $P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})$ we have that $M''_{i_j}/T \subset M'_{i_j}/T$; or

4. for some $M''_{n+j}/\emptyset$ such that $m_{i_j} \in S$ we have $M''_{n+j}/\emptyset \subset M'_{n+j}/\emptyset$.

Let $\mathbf{N}$ denote an interpretation for $IR_{\mathbf{P}}(S)$ such that

- $N_k/T = M''_k/T$ for all $P_k[T] \in \mathrm{VC}(\mathbf{P})$ such that $m_k \notin S \cup accs_{\mathbf{P}}(S)$,
- $N_k/\emptyset = M''_k/\emptyset \setminus val(M_k/\emptyset)$ for each module $m_k \in accs_{\mathbf{P}}(S)$,
- $N_{i_j}/T = lift(M''_{n+j}/\emptyset, T)$ for all $P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})$ such that $m_{i_j} \in S$, and
- for each module $m_{i_j} \in S$,

$$N_{n+j}/\emptyset = \{bit(0), bit(1)\} \cup \bigcup_{P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})} val(M''_{i_j}/T) \cup flat(M''_{i_j}/T) \ . \qquad (6.17)$$

If one of (1)–(4) is true, we get that $\mathbf{N} < \mathbf{M}$. For case (1), we have for $m_k \notin S \cup accs_{\mathbf{P}}(S)$, $m'_k = m_k$ by definition of $CR_{\mathbf{P}}(S)$ and $M'_k/T = M_k/T$. Thus, for $P_k[T] \in \mathrm{VC}(\mathbf{P})$ the reduct $f\, IR_{\mathbf{P}}(S)(P_k[T])^{\mathbf{M}}$ is equal to $f\, CR_{\mathbf{P}}(S)(P_k[T])^{\mathbf{M}'}$. Since $M''_k/T \subset M'_k/T$ and $M'_k/T = M_k/T$, we get $N_k/T \subset M_k/T$. By assumption $\mathbf{M}'' \models f\, CR_{\mathbf{P}}(S)^{\mathbf{M}'}$, we have that $\mathbf{M}'', P_k[T] \models f\, CR_{\mathbf{P}}(S)^{\mathbf{M}'}$ for all $P_k[T] \in \mathrm{VC}(CR_{\mathbf{P}}(S))$, and thus $\mathbf{N}, P_k[T] \models f\, CR_{\mathbf{P}}(S)^{\mathbf{M}'}$ for all $P_k[T] \in \mathrm{VC}(IR_{\mathbf{P}}(S))$. But this contradicts $\mathbf{M}$ being a minimal model of $f\, IR_{\mathbf{P}}(S)^{\mathbf{M}}$, therefore $\mathbf{N}$ is not a model of $f\, IR_{\mathbf{P}}(S)^{\mathbf{M}}$ and thus $\mathbf{M}''$ is not a model of $f\, CR_{\mathbf{P}}(S)^{\mathbf{M}'}$.

In case (2), $M''_k/\emptyset \subset M'_k/\emptyset$, so we get that $N_k/\emptyset \subset M_k/\emptyset$, as from our assumption that $\mathbf{M}''$ is a model for $f\, CR_{\mathbf{P}}(S)^{\mathbf{M}'}$, we cannot have that atoms from $val(M_k/\emptyset)$ are missing from $M''_k/\emptyset$. Therefore, $\mathbf{N} < \mathbf{M}$ and by minimality of $\mathbf{M}$ we have that $\mathbf{N}$ does not satisfy $f\, IR_{\mathbf{P}}(S)^{\mathbf{M}}$. We must have a rule $r \in f\, IR_{\mathbf{P}}(S)(P_k[\emptyset])^{\mathbf{M}}$ such that $\mathbf{N}, P_k[\emptyset] \not\models r$, i.e., $\mathbf{N}, P_k[\emptyset] \models B(r)$ but $\mathbf{N}, P_k[\emptyset] \not\models H(r)$. From $r \in f\, IR_{\mathbf{P}}(S)(P_k[\emptyset])^{\mathbf{M}}$ we can deduce that there exists an $r' \in f\, CR_{\mathbf{P}}(S)(P_k[\emptyset])^{\mathbf{M}'}$ such that $r' = \mathcal{C}^S_{m_k}(r)$ and by definition of the FLP-reduct, $\mathbf{M}'', P_k[\emptyset] \models B(r')$. But now we arrive at a contradiction, as we must have that $\mathbf{M}'', P_k[\emptyset] \not\models H(r')$ and therefore $\mathbf{M}'', P_k[\emptyset] \not\models r'$, which is necessary to satisfy $f\, CR_{\mathbf{P}}(S)^{\mathbf{M}'}$.

For the case (3), $M''_{i_j}/T \subset M'_{i_j}/T$, we have that (6.17) is true. The subset-relationship $\mathbf{N} < \mathbf{M}$ follows from $M''_{i_j}/T \subset M'_{i_j}/T$, where M'_{i_j}/T is $lift(M_{n+j}/\emptyset, T)$, and thus $N_{n+j}/\emptyset \subset M_{n+j}/\emptyset$. Therefore, $\mathbf{N} < \mathbf{M}$ holds and by minimality of $\mathbf{M}$ we have that $\mathbf{N}$ does not satisfy $f\, IR_{\mathbf{P}}(S)^{\mathbf{M}}$. We must have a rule $r' \in f\, IR_{\mathbf{P}}(S)(P_{n+j}[\emptyset])^{\mathbf{M}}$ such that $\mathbf{N}, P_{n+j}[\emptyset] \not\models r'$, that is $\mathbf{N}, P_{n+j}[\emptyset] \models B(r')$ but $\mathbf{N}$ does not satisfy $H(r')$ at $P_{n+j}[\emptyset]$. As r' appears in the shadow modules it must be from the set $grnd(\mathcal{B}_{m_{i_j}}(r))$, where $r \in R(m_{i_j})$. Since $r' \in f\, IR_{\mathbf{P}}(S)(P_{n+j}[\emptyset])^{\mathbf{M}}$, we must have $\mathbf{M}, P_{n+j}[\emptyset] \models B(r')$, and

by construction of $\mathbf{M}'$ and $\mathbf{N}$, we must have $\mathbf{M}', P_{i_j}[T] \models B(r)$, where $T = \mathtt{bs}_{i_j}(\mathbf{b})$ for $\{\mathtt{bit}(\mathbf{b})\} \subseteq B(r')$. Therefore, $r \in f\ CR_{\mathbf{P}}(S)(P_{i_j}[T])^{\mathbf{M}'}$. As $\mathbf{N}, P_{n+j}[\varnothing] \not\models H(r')$, no $a(\mathbf{c}, \mathbf{b}) \in H(r')$ is true in $\mathbf{N}$, and from the construction of $\mathbf{N}$, we can conclude that for all $a(\mathbf{c}) \in H(r)$ we have $a(\mathbf{c}) \notin M''_{i_j}/T$. Hence, $\mathbf{M}'', P_{i_j}[T] \not\models H(r)$, so we get that $\mathbf{M}'', P_{i_j}[T] \not\models r$, which contradicts our assumption that $\mathbf{M}''$ satisfies $f\ CR_{\mathbf{P}}(S)(P_{i_j}[T])^{\mathbf{M}'}$.

For the case (4), $M''_{n+j}/\varnothing \subset M'_{n+j}/\varnothing$, we have that $N_{i_j}/T = \mathit{lift}(M''_{n+j}/\varnothing, T)$ for all $P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})$ such that $m_{i_j} \in S$. For some particular $P_{i_j}[T]$, we have that $N_{i_j}/T \subset M_{i_j}/T$, which follows from the construction of $M'_{n+j}/\varnothing$ from one of the M_{i_j}/T. Now we have that $\mathbf{N} < \mathbf{M}$ by minimality of $\mathbf{M}$ we conclude that $\mathbf{N}$ does not satisfy $f\ IR_{\mathbf{P}}(S)^{\mathbf{M}}$. Now we can apply the same line of reasoning as in case (3), this time only reversing the role of the shadow modules and the modules from S. We then arrive at $\mathbf{M}''$ not satisfying $f\ CR_{\mathbf{P}}(S)(P_{n+j}[\varnothing])^{\mathbf{M}'}$, a contradiction.

($\Leftarrow$) Let $\mathbf{M}'$ be an answer set of $CR_{\mathbf{P}}(S)$. We show now that there exists a corresponding answer set $\mathbf{M}$ of $IR_{\mathbf{P}}(S)$ such that (a) $\mathbf{M} \models f\ IR_{\mathbf{P}}(S)^{\mathbf{M}}$, and that (b) $\mathbf{M}$ is a minimal model of $f\ IR_{\mathbf{P}}(S)^{\mathbf{M}}$. Intuitively, we convert the shadow part from $\mathbf{M}'$ to the S-part of interpretation $\mathbf{M}$, and the S-part from $\mathbf{M}'$ to the shadow part from $\mathbf{M}$. The proof now works, *mutatis mutandis*, as the proof for ($\Rightarrow$), hence both (a) and (b) are true, and $\mathbf{M}$ is an answer set of $IR_{\mathbf{P}}(S)$. □

6.3.3 Module Removal of Connected Closed Call Sets

We are now able to show that we can remove a connected closed call set from an MLP $\mathbf{P}$ in order to obtain a rewritten MLP without input parameters.

We first define module removal, whose aim is to prune off modules from an MLP. In later sections, we will employ module removal again for simplifying MLPs. We therefore define a general notion of module removal, and use it specifically for removing connected closed call sets here.

Definition 6.18 (Module removal).
Let $\mathbf{P} = (m_1, \ldots, m_n)$ be an MLP, let m_k be a module from $\mathbf{P}$, and let $S = \{m_{i_1}, \ldots, m_{i_h}\}$ be a set of modules from $\mathbf{P}$. We define

$$\mathbf{P} - m_k = (m_1, \ldots, m_{k-1}, m_{k+1}, \ldots, m_n)$$

to be the *reduced MLP* $\mathbf{P}$ *with respect to module* m_k and

$$\mathbf{P} - S = \big(\cdots\big((\mathbf{P} - m_{i_1}) - m_{i_2}\big)\cdots\big) - m_{i_h}$$

to be the *reduced MLP* $\mathbf{P}$ *with respect to a set of modules* S.

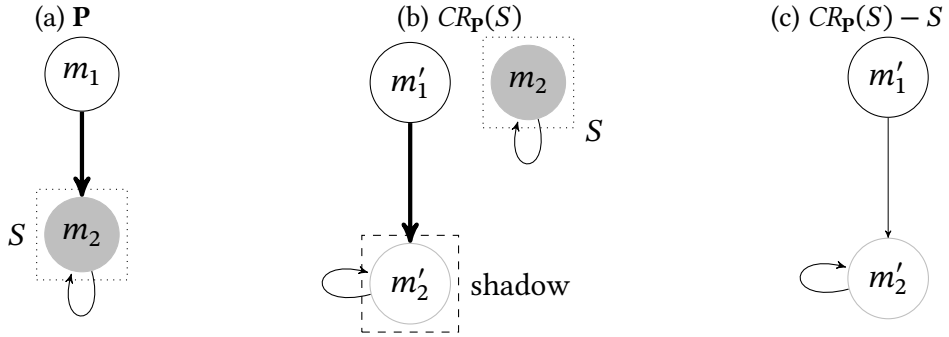


Figure 6.3: Module Removal

Intuitively, removing modules from an MLP $\mathbf{P}$ might lead to a program that is not well-formed, i.e., it could be that some modules in $\mathbf{P} - m_k$ or $\mathbf{P} - S$ depend on m_k or on modules in S. But under certain conditions, we can prune superfluous modules of S from an MLP and get an equivalent MLP with fewer modules.

One possibility is to prune the call rewriting $CR_{\mathbf{P}}(S)$, where S is a connected closed call set with respect to a module m_i of the MLP $\mathbf{P}$. Since S is a connected closed call set, $CR_{\mathbf{P}}(S) - S$ is guaranteed to have no dependencies from $CR_{\mathbf{P}}(S) - S$ to any module from S, which allows us to remove superfluous modules from the MLP $CR_{\mathbf{P}}(S)$.

The next example illustrates module removal using the even module.

Example 6.8 Consider the MLP $\mathbf{P} = (m_1, m_2)$, where $m_1 = (main[], R_1)$ is a main module with the set of rules R_1

$$
\begin{aligned}
&p(a) \leftarrow \\
&p(b) \leftarrow \\
&\quad r \leftarrow P[p].even
\end{aligned}
$$

and $m_2 = (P[q/1], R_2)$ is the module from Example 3.1. Let $S = \{m_2\}$ be a connected closed call set with respect to m_1. Figure 6.3a shows the inter-module dependencies of $\mathbf{P}$.

Then, the call rewriting $CR_{\mathbf{P}}(S)$ is given by the MLP (m_1', m_2, m_2'), where m_1' and m_2' are shown below. Figure 6.3b shows the shadow module m_2' and the disconnected module m_2. Applying module removal of $CR_{\mathbf{P}}(S)$ with respect to S yields $CR_{\mathbf{P}}(S) - S =$

(m_1', m_2') with the main module $m_1' = (main[], R_1')$, where R_1' is the set of rules

$$p(a) \leftarrow$$

$$p(b) \leftarrow$$

$$\begin{aligned} val_p(a,1) &\leftarrow p(a) \\ val_p(a,0) &\leftarrow \text{not } p(a) \\ val_p(b,1) &\leftarrow p(b) \\ val_p(b,0) &\leftarrow \text{not } p(b) \end{aligned}$$

$$r \leftarrow P.even(V_{1,1}, V_{1,2}), val_p(a, V_{1,1}), val_p(b, V_{1,2})$$

and the library module $m_2' = (P[], R_2')$, where R_2' is the set of rules

$$\begin{aligned} bit(0) &\leftarrow \\ bit(1) &\leftarrow \end{aligned}$$

$$\begin{aligned} q(a,1,B_{1,2}) &\leftarrow bit(B_{1,1}), bit(B_{1,2}) \\ q(b,B_{1,1},1) &\leftarrow bit(B_{1,1}), bit(B_{1,2}) \end{aligned}$$

$$\begin{aligned} q'(X,B_{1,1},B_{1,2}) \vee q'(Y,B_{1,1},B_{1,2}) \leftarrow\ & q(X,B_{1,1},B_{1,2}), q(Y,B_{1,1},B_{1,2}), X \neq Y, \\ & bit(B_{1,1}), bit(B_{1,2}) \end{aligned}$$

$$\begin{aligned} skip(X,B_{1,1},B_{1,2}) \leftarrow\ & q(X,B_{1,1},B_{1,2}), \text{not } q'(X,B_{1,1},B_{1,2}), \\ & bit(B_{1,1}), bit(B_{1,2}) \end{aligned}$$

$$\begin{aligned} val_{q'}(a,1,B_{1,1},B_{1,2}) &\leftarrow q'(a,B_{1,1},B_{1,2}), bit(B_{1,1}), bit(B_{1,2}) \\ val_{q'}(a,0,B_{1,1},B_{1,2}) &\leftarrow \text{not } q'(a,B_{1,1},B_{1,2}), bit(B_{1,1}), bit(B_{1,2}) \\ val_{q'}(b,1,B_{1,1},B_{1,2}) &\leftarrow q'(b,B_{1,1},B_{1,2}), bit(B_{1,1}), bit(B_{1,2}) \\ val_{q'}(b,0,B_{1,1},B_{1,2}) &\leftarrow \text{not } q'(b,B_{1,1},B_{1,2}), bit(B_{1,1}), bit(B_{1,2}) \end{aligned}$$

$$\begin{aligned} odd(B_{1,1},B_{1,2}) \leftarrow\ & skip(X,B_{1,1},B_{1,2}), P.even(V_{1,1},V_{1,2}), \\ & bit(B_{1,1}), bit(B_{1,2}), \\ & val_{q'}(a,V_{1,1},B_{1,1},B_{1,2}), \\ & val_{q'}(b,V_{1,2},B_{1,1},B_{1,2}) \end{aligned}$$

$$even(B_{1,1},B_{1,2}) \leftarrow \text{not } odd(B_{1,1},B_{1,2}), bit(B_{1,1}), bit(B_{1,2})$$

Note that $CR_{\mathbf{P}}(S) - S$ consists of modules without input and preserves the call structure of $\mathbf{P}$, as shown in Figure 6.3c.

In the following, we let $\mathbf{P} = (m_1, \dots, m_n)$ be an MLP, $m_i = (P_i[\mathbf{q}_i], R_i)$ be a module from $\mathbf{P}$ such that $|\mathbf{q}_i| \leq 1$, and $S = \{m_{i_1}, \dots, m_{i_h}\}$ is a connected closed call set with respect to m_i. We are now able to show the following.

Proposition 6.5 (Module Removal)
The answer sets of the MLP $\mathbf{P}$ correspond one-to-one to the answer sets of the MLP $CR_{\mathbf{P}}(S) - S$.

Proof As S is a connected closed call set with respect to m_i, the modules in S call either modules in S or main modules from $\mathbf{P}$ that do not show up in S. Hence, $CR_{\mathbf{P}}(S) - S$ is a proper MLP, as the remaining modules in $CR_{\mathbf{P}}(S) - S$ do not call any module in S anymore.

What is left to be shown is that the reduced MLP $CR_{\mathbf{P}}(S) - S$ has the same answer sets as $\mathbf{P}$. Let $\mathbf{M}$ be an answer set for $\mathbf{P}$, and let $\mathbf{M}^i$ be the answer set $\mathbf{M}'$ for $IR_{\mathbf{P}}(S)$ that is obtained from the proof of Lemma 6.3. Furthermore, let $\mathbf{M}^c$ be the answer set $\mathbf{M}'$ for $CR_{\mathbf{P}}(S)$ that is obtained from using $\mathbf{M}^i$ as answer set of $IR_{\mathbf{P}}(S)$ in the proof of Lemma 6.4. We obtain an interpretation $\widehat{\mathbf{M}}$ for the MLP $CR_{\mathbf{P}}(S) - S$ from $\mathbf{M}^c$ by removing all $M^c_{i_j}/T$ from $\mathbf{M}^c$ such that $m_{i_j} \in S$ and $P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})$. Since no module from $CR_{\mathbf{P}}(S) - S$ calls any module from S, we immediately get that $\widehat{\mathbf{M}}$ is an answer set for $CR_{\mathbf{P}}(S) - S$.

Now let $\widehat{\mathbf{M}}$ be an answer set for $CR_{\mathbf{P}}(S) - S$. We receive an interpretation $\mathbf{M}$ for $\mathbf{P}$ by first removing all $\widehat{M_{n+j}}/\varnothing$ from $\widehat{\mathbf{M}}$ such that $m_{i_j} \in S$, and then adding for each $P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})$ the set $M_{i_j}/T = \mathit{lift}(\widehat{M_{n+j}}/\varnothing, T)$ to $\mathbf{M}$, where $\mathit{lift}(\cdot,\cdot)$ is the function defined in (6.16).

To see that $\mathbf{M}$ is an answer set of $\mathbf{P}$, we let $\mathbf{M}^c$ be an interpretation of $CR_{\mathbf{P}}(S)$ that is obtained from $\widehat{\mathbf{M}}$ by adding $M^c_{i_j}/T = \mathit{lift}(\widehat{M_{n+j}}/\varnothing, T)$ for all $m_{i_j} \in S$ and $P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})$. It is easy to see that for each $P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})$ such that $m_{i_j} \in S$, each rule $r \in f\, CR_{\mathbf{P}}(S)(P_{i_j}[T])^{\widehat{\mathbf{M}}}$ has a corresponding rule $r' \in f\,(CR_{\mathbf{P}}(S) - S)(P_{n+j}[\varnothing])^{\mathbf{M}^c}$, thus $\mathbf{M}^c$ must be an answer set of $CR_{\mathbf{P}}(S)$. Now let $\mathbf{M}^i$ be the answer set for $IR_{\mathbf{P}}(S)$ that is obtained from $\mathbf{M}^c$ in the proof of Lemma 6.3. Note that we have now all the original rules r from $m_k \in \mathit{accs}_{\mathbf{P}}(S)$ from the rewritten rules $\mathcal{C}^S_{m_k}(r)$ for the module m'_k from $CR_{\mathbf{P}}(S)$ also satisfied in $IR_{\mathbf{P}}(S)$. Now as $\mathbf{P}$ does not call modules m_{n+j} from $IR_{\mathbf{P}}(S)$, we immediately get that $\mathbf{M}$ is also an answer set of $\mathbf{P}$, since $f\,\mathbf{P}(P_{i_j}[T])^{\mathbf{M}} = f\,IR_{\mathbf{P}}(S)(P_{i_j}[T])^{\mathbf{M}^i}$ for all $P_{i_j}[T] \in \mathrm{VC}(\mathbf{P})$. □

6.4 Macro Expansion of Modular Logic Programs

In this section, we develop a macro expansion rewriting technique that does not impose blowing up the arity of predicates. Here, we copy the rules from a particular module into their calling module. Note that this approach to rewriting modular logic programs is only applicable to modules whose rules are Horn, and whose call graph is acyclic. In the following §6.5, we will then show how this technique can be applied to dl-programs (Eiter et al., 2008).

6.4.1 Module Copy Rewriting

Module copy rewriting is split into two parts, one for the module callee and one for the module caller. The first one takes a module that is being called by another one and rewrites its atoms by attaching the associated module call to its predicates, as well as adding new rules that deal with module input. The second part of the translation takes care of the caller module by replacing the module call by the fresh predicate created in the first part of the translation. Adding the callee translation to the module caller then gives us a new module that has one module call less and the rules of the called module added to the caller module. When applying this method to all the module calls, we can remove module atoms, thus creating an MLP with one or more disconnected library modules, which in a subsequent step can be safely removed.

We begin with the definitions for rewriting module callees.

Definition 6.19 (Module callee input).
Let $m = (Q[q], R)$ be a module and p be a predicate symbol matching the arity of q. We define the *module callee input rule* as

$$\mathcal{CI}(m, Q[p]) = \quad q^{Q[p]}(\mathbf{X}) \leftarrow p(\mathbf{X}) \quad .$$

The rules of a called module are rewritten based on the following translation.

Definition 6.20 (Module callee rewriting).
Let $m = (Q[q], R)$ be a module and let p be a predicate symbol matching the arity of q. For an atom a appearing in m, we define

$$\mathcal{CT}(a, Q[p]) = \begin{cases} b^{Q[p]}(\mathbf{t}) & \text{if } a \text{ is of form } b(\mathbf{t}) \\ P_j[b^{Q[p]}].o(\mathbf{t}) & \text{if } a \text{ is of form } P_j[b].o(\mathbf{t}) \end{cases} .$$

Given a rule r of form (3.2), we can now define $\mathcal{CT}(r, Q[p])$ to be the rule

$$\mathcal{CT}(\alpha_1, Q[p]) \vee \cdots \vee \mathcal{CT}(\alpha_k, Q[p]) \leftarrow \mathcal{CT}(\beta_1, Q[p]), \ldots, \mathcal{CT}(\beta_m, Q[p]), \\ \text{not } \mathcal{CT}(\beta_{m+1}, Q[p]), \ldots, \text{not } \mathcal{CT}(\beta_n, Q[p]) .$$

For the set of rules R in m, we let

$$\mathcal{CT}(R, Q[p]) = \{\mathcal{CT}(r, Q[p]) \mid r \in R\} \ .$$

The next part of the translation takes a calling module and a called module, and rewrites the module atom of the called module. A module m_k calls a module m_i if there is a rule in $R(m_k)$ that has a module atom of form $P_i[p].o(\mathbf{t})$ in its body.

Definition 6.21 (Module caller rewriting).
Given two modules $m_k = (P_k[q_k], R_k)$ and $m_i = (P_i[q_i], R_i)$ such that m_k calls m_i, we define

$$\mathcal{CM}(a, P_i[p]) = \begin{cases} o^{P_i[p]}(\mathbf{t}) & \text{if } a \text{ is of form } P_i[p].o(\mathbf{t}) \\ a & \text{otherwise} \end{cases} ,$$

and for a rule r of form (3.2), we define $\mathcal{CM}(r, P_i[p])$ to be the rule

$$\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \mathcal{CM}(\beta_1, P_i[p]), \ldots, \mathcal{CM}(\beta_m, P_i[p]), \\ \text{not } \mathcal{CM}(\beta_{m+1}, P_i[p]), \ldots, \text{not } \mathcal{CM}(\beta_n, P_i[p]) \ .$$

For a set of rules R and the module atom $P_i[p]$, we let

$$\mathcal{CM}(R, P_i[p]) = \{\mathcal{CM}(r, P_i[p]) \mid r \in R\} \ .$$

Given m_k and m_i as in Definition 6.21, we can now incorporate m_i into m_k such that m_k does not call m_i anymore. We therefore formally define:

Definition 6.22 (Module copy rewriting).
The *module copy rewriting of the MLP* $\mathbf{P}$ *with respect to the modules* m_k *and* m_i *and module atom* $P_i[p].o(\mathbf{t})$ is the MLP

$$MCR(\mathbf{P}, m_k, m_i, P_i[p]) = (m_1, \ldots, m_{k-1}, m'_k, m_{k+1}, \ldots, m_n) \ ,$$

where

$$m'_k = (P_k[q_k], \mathcal{CM}(R(m_k), P_i[p]) \cup \mathcal{CT}(R(m_i), P_i[p]) \cup \mathcal{CI}(m_i, P_i[p])) \ .$$

Example 6.9 Let $\mathbf{P} = (m_0, m_1, m_2, m_3)$ be an MLP with the main module $m_0 = (P_0[], R_0)$, and the three library modules $m_1 = (P_1[q_1], R_1)$, $m_2 = (P_2[q_2], R_2)$, and $m_3 = (P_3[q_3], R_3)$. The rules of $\mathbf{P}$ are as follows:

$$R_0 = \left\{ \begin{array}{l} a \leftarrow \\ c \leftarrow P_1[a].o_1, P_2[b].o_2 \\ d \leftarrow P_1[b].o_1 \end{array} \right\}$$

$$R_1 = \left\{ \begin{array}{l} o_1 \leftarrow \text{not } P_2[q_1].o_2 \\ o_1 \leftarrow P_3[q_1].o_3 \end{array} \right\}$$

$$R_2 = \left\{ \ o_2 \leftarrow P_3[q_2].o_3 \ \right\}$$

$$R_3 = \left\{ \ o_3 \leftarrow q_3 \ \right\}$$

For the modules m_2 and m_3 of the MLP $\mathbf{P}$, the module copy rewriting is given by

$$MCR(\mathbf{P}, m_2, m_3, P_3[q_2]) = (m_0, m_1, m'_2, m_3) \ ,$$

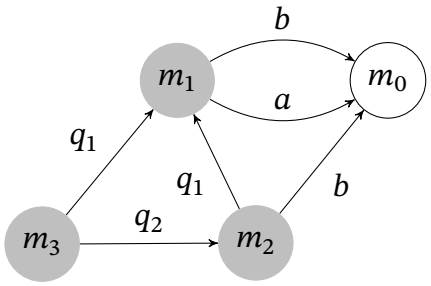


Figure 6.4: Directed connection graph $MC_{\mathbf{P}}$

where $m_2' = (P_2[q_2], R_2')$ such that R_2' is

$$\begin{aligned} \mathcal{CI}(m_2, P_3[q_2]): &\quad q_3^{P_3[q_2]} \leftarrow q_2 \\ \mathcal{CM}(R(m_2), P_3[q_2]): &\quad o_2 \leftarrow o_3^{P_3[q_2]} \\ \mathcal{CT}(R(m_2), P_3[q_2]): &\quad o_3^{P_3[q_2]} \leftarrow q_3^{P_3[q_2]} \end{aligned}$$

In order to capture module dependencies for the rewriting, we define the following dependency graph.

Definition 6.23 (Directed connection graph).
The *directed connection graph of the MLP* $\mathbf{P}$ is the directed labeled graph $MC_{\mathbf{P}} = (V, E)$ with vertex set $V = \{m_1, \dots, m_n\}$ consisting of all modules of $\mathbf{P}$ and edge set $E = \{(m_i, m_k, p) \mid (m_i, m_k) \in V \times V \text{ and } P_i[p].o(\mathbf{t}) \in ma(m_k)\}$.

That is, a labeled edge (m_i, m_k, p) appears in $MC_{\mathbf{P}}$ from m_i to m_k capturing the dependency from the callee module m_i to the caller module m_k for every module input predicate p of a module atom $P_i[p].o(\mathbf{t})$ in m_k. Note that the labels p allow to have multiple edges from m_i to m_k.

Example 6.10 (cont'd) The directed connection graph $MC_{\mathbf{P}}$ for the MLP $\mathbf{P}$ from Example 6.9 is shown in Figure 6.4.

For a set of ground atoms M and module tag $Q[p]$, we define the functions

$$\begin{aligned} tag(M, Q[p]) &= \{a^{Q[p]}(\mathbf{c}) \mid a(\mathbf{c}) \in M\} \ , \\ untag(M, Q[p]) &= \{a(\mathbf{c}) \mid a^{Q[p]}(\mathbf{c}) \in M\} \ , \text{ and} \\ notag(M, Q[p]) &= M \setminus \{a^{Q[p]}(\mathbf{c}) \in M\} \end{aligned}$$

to show the following.

Lemma 6.6
Let $\mathbf{P} = (m_1, \dots, m_n)$ be an MLP such that for the modules m_k and m_i from $\mathbf{P}$

- the module atom $P_i[p].o(\mathbf{t})$ appears in m_k,
- m_i is a Horn module, and
- no strongly connected component of $MC_\mathbf{P}$ contains both m_i and m_k.

Then,

- for an answer set $\mathbf{M}$ of $\mathbf{P}$ there exists an answer set $\mathbf{M}'$ of the module copy rewriting $MCR(\mathbf{P}, m_k, m_i, P_i[p])$ such that for all $P_j[T] \in \mathrm{VC}(\mathbf{P})$ with $S = (M_j/T)|_p^{q_i}$,

$$M'_j/T = \begin{cases} M_k/T \cup tag(M_i/S, P_i[p]) & \text{if } j = k, \\ M_j/T & \text{otherwise;} \end{cases} \tag{6.18}$$

- for an answer set $\mathbf{M}'$ of $MCR(\mathbf{P}, m_k, m_i, P_i[p])$ there exists an answer set $\mathbf{M}$ of the MLP $\mathbf{P}$ such that for all $P_j[T] \in \mathrm{VC}(\mathbf{P})$,

$$M_j/T = \begin{cases} notag(M'_j/T, P_i[p]) & \text{if } j = k, \\ M'_j/T & \text{otherwise.} \end{cases} \tag{6.19}$$

Proof Intuitively, the answer sets of $\mathbf{P}$ and $MCR(\mathbf{P}, m_k, m_i, P_i[p])$ correspond one-to-one to each other. An answer set $\mathbf{M}$ of $\mathbf{P}$ can be mapped to an answer set of $MCR(\mathbf{P}, m_k, m_i, P_i[p])$ by copying the model M_i/S and tagging the atoms with a module atom label to M_k/T for value calls $P_i[S]$ that get called from $P_k[T]$. In the other direction, we can remove those additional atoms from M'_k/T.

($\Rightarrow$) Let $\mathbf{M}$ be an answer set of $\mathbf{P}$. We show now that $\mathbf{M}'$—constructed from $\mathbf{M}$ as defined above—is an answer set of the MLP $MCR(\mathbf{P}, m_k, m_i, P_i[p])$, i.e., we show that (a) $\mathbf{M}' \vDash f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])^{\mathbf{M}'}$, and that (b) $\mathbf{M}'$ is a minimal model of the reduct $f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])^{\mathbf{M}'}$.

We show item a now. We immediately get that $\mathbf{M}', P_j[T] \vDash f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])^{\mathbf{M}'}$ for $j \neq k$, as

$$f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])(P_j[T])^{\mathbf{M}'} = f\,\mathbf{P}(P_j[T])^{\mathbf{M}}$$

in this case. From the definition of module copy rewriting, we can see that

$$\begin{aligned} &\mathcal{CM}(f\,\mathbf{P}(P_k[T])^{\mathbf{M}}, P_i[p]) \cup \mathcal{CT}(f\,\mathbf{P}(P_k[T])^{\mathbf{M}}, P_i[p]) \\ &\subseteq f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])(P_k[T])^{\mathbf{M}'} \\ &\subseteq \mathcal{CM}(f\,\mathbf{P}(P_k[T])^{\mathbf{M}}, P_i[p]) \cup \mathcal{CT}(f\,\mathbf{P}(P_k[T])^{\mathbf{M}}, P_i[p]) \cup grnd(\mathcal{CI}(m_i, P_i[p])) \ . \end{aligned}$$

All rules of $\mathcal{CT}(f\,\mathbf{P}(P_k[T])^{\mathbf{M}}, P_i[p])$ are satisfied, as we have that $tag(M_i/S, P_i[p])$ is contained in M'_k/T. Furthermore, $grnd(\mathcal{CI}(m_i, P_i[p]))$ is satisfied by M'_k/T, as for M_i/S such that $S = (M_k/T)|_p^{q_i}$, we have that $S \subseteq M_i/S$ by definition of an interpretation for an MLP $\mathbf{P}$, hence all rules

$$q_i^{P_i[p]}(\mathbf{c}) \leftarrow p(\mathbf{c}) \in grnd(\mathcal{CI}(m_i, P_i[p]))$$

such that $p(\mathbf{c}) \in M_k/T$ are true in M'_k/T. Thus, all rules of $\mathcal{CM}(f\,\mathbf{P}(P_k[T])^{\mathbf{M}}, P_i[p])$ without a module atom $P_i[p].o(\mathbf{c})$ in their bodies are satisfied, as M'_k/T contains all atoms from M_k/T, which is a model of $f\,\mathbf{P}(P_k[T])^{\mathbf{M}}$. Now let $r \in f\,\mathbf{P}(P_k[T])^{\mathbf{M}}$ such that there is a module atom $a = P_i[p].o(\mathbf{c}) \in B(r)$. Then, there is the corresponding rule $r' \in \mathcal{CM}(f\,\mathbf{P}(P_k[T])^{\mathbf{M}}, P_i[p])$ such that $o^{P_i[p]}(\mathbf{c}) \in B(r')$. For $\mathbf{M}, P_k[T] \models a$ we must have that $\mathbf{M}, P_i[S] \models o(\mathbf{c})$ for $S = (M_k/T)|_p^{q_i}$, and for $\mathbf{M}, P_k[T] \not\models a$ we get $\mathbf{M}, P_i[S] \not\models o(\mathbf{c})$. Hence, $o(\mathbf{c}) \in M_i/S$ (respectively, $o(\mathbf{c}) \notin M_i/S$) and so we have that $o^{P_i[p]}(\mathbf{c}) \in M'_k/T$ (respectively, $o^{P_i[p]}(\mathbf{c}) \notin M'_k/T$), which proves that $\mathbf{M}'$ satisfies $B(r')$. As $H(r) = H(r')$, we also have that $\mathbf{M}'$ satisfies $H(r')$, since $\mathbf{M}$ is a model of r. Hence, $\mathbf{M}' \models f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])^{\mathbf{M}'}$.

We continue with item b. To show that $\mathbf{M}'$ is a minimal model of

$$f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])^{\mathbf{M}'} \quad ,$$

we must ensure that there is no smaller interpretation $\mathbf{M}'' < \mathbf{M}'$ such that $\mathbf{M}''$ is a model of $f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])^{\mathbf{M}'}$.

Towards a contradiction, assume $\mathbf{M}'' \models f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])^{\mathbf{M}'}$. As $\mathbf{M}'' < \mathbf{M}'$, we consider the following cases:

1. for some M''_j/U such that $j \neq k$ and $j \neq i$, we have $M''_j/U \subset M'_j/U$;
2. for some M''_k/T we have $M''_k/T \subset M'_k/T$; and
3. for some M''_i/S we have $M''_i/S \subset M'_i/S$.

Let $\mathbf{N}$ denote an interpretation for $\mathbf{P}$ such that

- $N_j/U = M''_j/U$ for all $P_j[U] \in \mathrm{VC}(\mathbf{P})$ such that $j \notin \{i, k\}$,
- $N_k/T = notag(M''_k/T, P_i[p])$ for each $P_k[T] \in \mathrm{VC}(\mathbf{P})$,
- for each value call $P_k[T] \in \mathrm{VC}(\mathbf{P})$ such that $S'' = (M''_k/T)|_p^{q_i}$, we set $N_i/S'' = untag(M''_k/T, P_i[p])$, and
- for all $P_i[S] \in \mathrm{VC}(\mathbf{P})$ such that $S \neq S''$ for any S'' from above, we set $N_i/S = M''_i/S$.

If one of (1)–(3) is true, then $\mathbf{N} < \mathbf{M}$. In case (1) and (3), we have that

$$f\,\mathbf{P}(P_j[U])^{\mathbf{M}} = f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])(P_j[U])^{\mathbf{M}'}$$

and

$$f\,\mathbf{P}(P_i[S])^{\mathbf{M}} = f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])(P_i[S])^{\mathbf{M}'} \ ,$$

respectively. As $\mathbf{M}$ is an answer set of $\mathbf{P}$, we get that $\mathbf{N}, P_j[U] \not\models f\,\mathbf{P}(P_j[U])^{\mathbf{M}}$ in case (1), and $\mathbf{N}, P_i[S] \not\models f\,\mathbf{P}(P_i[S])^{\mathbf{M}}$ in case (3). Therefore, in either case, we get

$$\mathbf{M}'', P_j[U] \not\models f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])(P_j[U])^{\mathbf{M}'}$$

and

$$\mathbf{M}'', P_i[S] \not\models f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])(P_i[S])^{\mathbf{M}'} \ ,$$

respectively, which contradicts $\mathbf{M}''$ being a model for $f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])^{\mathbf{M}'}$.

In case (2), there must exist an atom $a \in M_k'/T$ such that $a \notin M_k''/T$. We distinguish the following cases: (i) a is of form $a^{P_i[p]}(\mathbf{c})$, or (ii) a is from $notag(M_k'/T, P_i[p])$.

In the case (i), a must be from $tag(M_i/S, P_i[p])$, where $S = (M_k/T)|_p^{q_i}$. Hence, for $a(\mathbf{c}) \in M_i/S$, we have that $a(\mathbf{c}) \notin N_i/S''$ for $S'' = (M_k''/T)|_p^{q_i}$. Since m_i is Horn and by assumption that both $\mathbf{M}$ and $\mathbf{N}$ are models of $f\,\mathbf{P}(P_i[S''])^{\mathbf{M}}$, we must have that $\mathbf{M} \cap \mathbf{N}, P_i[S''] \models f\,\mathbf{P}(P_i[S''])^{\mathbf{M}}$. As $N_i/S'' \subset M_i/S''$, we must have that $\mathbf{M} \cap \mathbf{N} < \mathbf{M}$, but since $\mathbf{M}$ is a minimal model which must be unique on the Horn module m_i, we arrive at a contradiction that $\mathbf{N}$ is a model of $f\,\mathbf{P}(P_i[S''])^{\mathbf{M}}$. Since $f\,\mathbf{P}(P_i[S''])^{\mathbf{M}} = f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])(P_i[S''])^{\mathbf{M}'}$ we also get that $\mathbf{M}''$ does not satisfy $f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])(P_i[S''])^{\mathbf{M}'}$, hence $\mathbf{M}'' \not\models f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])^{\mathbf{M}'}$ and we must have that $\mathbf{M}'$ is a minimal model for $f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])^{\mathbf{M}'}$.

In case (ii), where $a \in notag(M_k'/T, P_i[p])$, we have that $a \notin M_k''/T$, hence $a \notin N_k/T$. As $N_k/T \subset M_k/T$, by minimality of $\mathbf{M}$ we can infer that there is a rule $r \in f\,\mathbf{P}(P_k[T])^{\mathbf{M}}$ such that $\mathbf{N}, P_k[T] \not\models r$, therefore both $\mathbf{N}, P_k[T] \models B(r)$ and $\mathbf{N}, P_k[T] \not\models H(r)$ hold. For the rule $r' = \mathcal{CM}(r, P_i[p])$ both ordinary and module literals in $B(r')$ are satisfied by $\mathbf{M}'$, as well as literals of form $o^{P_i[p]}(\mathbf{c}) = \mathcal{CM}(P_i[p].o(\mathbf{c}), P_i[p])$ (respectively, not $o^{P_i[p]}(\mathbf{c})$). Thus, $o(\mathbf{c}) \in M_i/S$ (respectively, $o(\mathbf{c}) \notin M_i/S$), so we have that $o^{P_i[p]}(\mathbf{c}) \in tag(M_i/S, P_i[p])$ (respectively, $o^{P_i[p]}(\mathbf{c}) \notin tag(M_i/S, P_i[p])$), and therefore we conclude that $\mathbf{M}', P_k[T] \models B(r')$, which means that

$$r' \in f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])(P_k[T])^{\mathbf{M}'} \ .$$

Hence we get $\mathbf{M}'', P_k[T] \not\models r'$ as $H(r) = H(r')$ and $\mathbf{M}'', P_k[T] \not\models H(r')$; therefore we have a contradiction for $\mathbf{M}''$ being a model for $f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])^{\mathbf{M}'}$.

($\Leftarrow$) Let $\mathbf{M}'$ be an answer set of $MCR(\mathbf{P}, m_k, m_i, P_i[p])$. We show now that there exists an answer set $\mathbf{M}$ of $\mathbf{P}$ that can be constructed from $\mathbf{M}'$, i.e., we show that (a) $\mathbf{M} \vDash f\,\mathbf{P}^{\mathbf{M}}$, and that (b) $\mathbf{M}$ is a minimal model of $f\,\mathbf{P}^{\mathbf{M}}$.

Let us consider item a. We immediately get that $\mathbf{M}, P_j[T] \vDash f\,\mathbf{P}^{\mathbf{M}}$ for $j \neq k$, as

$$f\,\mathbf{P}(P_j[T])^{\mathbf{M}} = f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])(P_j[T])^{\mathbf{M}'}$$

in this case. Otherwise, from the definition of module copy rewriting, we can see that if $r \in f\,\mathbf{P}(P_k[T])^{\mathbf{M}}$, then there exists the rule $r' \in f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])(P_k[T])^{\mathbf{M}'}$ such that $r' = \mathcal{CM}(r, P_i[p])$. For those rules, we have $\mathbf{M}', P_k[T] \vDash B(r')$. Ordinary literals are satisfied in $B(r)$ as well; it remains to show that literals with module atoms of form $P_i[p].o(\mathbf{c}) \in B(r)$ are also satisfied by $\mathbf{M}$. This follows from m_i being Horn, since for a rewritten atom $o^{P_i[p]}(\mathbf{c}) = \mathcal{CM}(P_i[p].o(\mathbf{c}), P_i[p]) \in M'_k/T$ there must be a rule $\hat{r} \in f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])(P_k[T])^{\mathbf{M}'}$ such that $H(\hat{r}) = o^{P_i[p]}(\mathbf{c})$. As m_i is Horn, it has a unique model on the value calls for m_i, thus there must exist a rule $\bar{r} \in f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])(P_i[S])^{\mathbf{M}'}$ with $S = (M'_k/T)|_p^{q_i}$ such that $H(\bar{r}) = o(\mathbf{c})$ and $\hat{r} = \mathcal{CT}(\bar{r}, P_i[p])$. Now as $\mathbf{M}', P_i[S] \vDash B(\bar{r})$ and since $f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])(P_i[S])^{\mathbf{M}'} = f\,\mathbf{P}(P_i[S])^{\mathbf{M}}$, we have $\bar{r} \in f\,\mathbf{P}(P_i[S])^{\mathbf{M}}$, and since $M_i/S = M'_i/S$ we also have $H(\bar{r}) = o(\mathbf{c}) \in M_i/S$, thus $\mathbf{M}, P_i[S] \vDash H(\bar{r})$. Now we can conclude that $\mathbf{M}, P_k[T] \vDash P_i[p].o(\mathbf{c})$, and so we get that all literals from $B(r)$ are satisfied by $\mathbf{M}$, and since $H(r) = H(r')$, we have $\mathbf{M}, P_k[T] \vDash H(r)$ from $\mathbf{M}', P_k[T] \vDash r'$. Therefore, $\mathbf{M}, P_k[T] \vDash r$ and thus $\mathbf{M}, P_k[T] \vDash f\,\mathbf{P}^{\mathbf{M}}$.

Now, we have that all value calls from $f\,\mathbf{P}^{\mathbf{M}}$ are satisfied by $\mathbf{M}$, hence $\mathbf{M}$ is a model for $f\,\mathbf{P}^{\mathbf{M}}$.

Next, we consider item b. To show that $\mathbf{M}$ is a minimal model of $f\,\mathbf{P}^{\mathbf{M}}$, we must ensure that there is no interpretation $\mathbf{N}$ such that $\mathbf{N} < \mathbf{M}$ and $\mathbf{N} \vDash f\,\mathbf{P}^{\mathbf{M}}$.

Towards a contradiction, assume $\mathbf{N}$ satisfies $f\,\mathbf{P}^{\mathbf{M}}$. As $\mathbf{N} < \mathbf{M}$, we consider the following cases:

1. for some N_j/U such that $j \neq k$, we have $N_j/U \subset M_j/U$; and
2. for some N_k/T we have $N_k/T \subset M_k/T$.

Let $\mathbf{M}''$ denote an interpretation for $MCR(\mathbf{P}, m_k, m_i, P_i[p])$ such that

- $M''_j/U = N_j/U$ for all $P_j[U] \in \mathrm{VC}(\mathbf{P})$ such that $j \neq k$,
- $M''_k/T = N_k/T \cup tag(N_i/S, P_i[p])$ such that $S = (N_k/T)|_p^{q_i}$ for each $P_k[T] \in \mathrm{VC}(\mathbf{P})$,

If one of (1) or (2) is true, then $\mathbf{M}'' < \mathbf{M}'$. In case (1), we have that

$$f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])(P_j[U])^{\mathbf{M}'} = f\,\mathbf{P}(P_j[U])^{\mathbf{M}} \quad .$$

As $\mathbf{M}'$ is an answer set of $MCR(\mathbf{P}, m_k, m_i, P_i[p])$, we get that

$$\mathbf{M}'', P_j[U] \not\models f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])(P_j[U])^{\mathbf{M}'} \quad .$$

Therefore we get $\mathbf{N}, P_j[U] \not\models f\,\mathbf{P}(P_j[U])^{\mathbf{N}}$, which contradicts $\mathbf{N}$ being a model for $f\,\mathbf{P}^{\mathbf{M}}$.

In case (2), there must exist an atom $a \in M_k/T$ such that $a \notin N_k/T$. Thus, by minimality of $\mathbf{M}'$, we get that $\mathbf{M}'' \not\models f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])(P_k[T])^{\mathbf{M}'}$, and as a must be from $notag(M'_k/T, P_i[p])$, there must be a rule $r' \in f\,MCR(\mathbf{P}, m_k, m_i, P_i[p])(P_k[T])^{\mathbf{M}'}$ such that $\mathbf{M}'', P_k[T] \models B(r')$ but $\mathbf{M}'', P_k[T] \not\models H(r')$. So we get that for the rule r' both literals from m_k as well as literals of form $o^{P_i[p]}(\mathbf{c}) = \mathcal{CM}(P_i[p].o(\mathbf{c}), P_i[p])$ are satisfied in the body of r' by $\mathbf{M}'$. Now let r be a rule from $\mathbf{P}$ such that $r' = \mathcal{CM}(r, P_i[p])$. Since $\mathbf{M}'', P_k[T] \models o^{P_i[p]}$ (respectively, $\mathbf{M}'', P_k[T] \not\models o^{P_i[p]}$), we get that also $\mathbf{M}, P_k[T] \models P_i[p].o(\mathbf{c})$ (respectively, $\mathbf{M}, P_k[T] \not\models P_i[p].o(\mathbf{c})$). Thus, $r \in f\,\mathbf{P}(P_k[T])^{\mathbf{M}}$. Now we have that $\mathbf{N}, P_k[T] \models B(r)$, and since $H(r) = H(r')$, we conclude $\mathbf{N}, P_k[T] \not\models r$. Therefore, we arrive at a contradiction to our assumption $\mathbf{N} \models f\,\mathbf{P}^{\mathbf{M}}$, and $\mathbf{M}$ must be a minimal model for $f\,\mathbf{P}^{\mathbf{M}}$. □

In the following, we consider sets of modules that can be rewritten using module copy rewriting. We start with defining the topological sort of a subgraph of $MC_{\mathbf{P}}$.

Definition 6.24 (Topological sort).
The *topological sort of* $MC_{\mathbf{P}} = (V, E)$ *with respect to a set* S *of modules of* $\mathbf{P}$ is a linear ordering $\prec \subseteq S \times S$ of vertices $S \subseteq V$ such that for all $m_i, m_k \in S$, $m_i \prec m_k$ whenever m_k is reachable from m_i in $MC_{\mathbf{P}}$.

Example 6.11 (cont'd) Continuing with Example 6.9, let $S = \{m_1, m_2, m_3\}$ be a set of modules from $\mathbf{P}$ and $MC_{\mathbf{P}}$ the directed connection graph of $\mathbf{P}$ as shown in Figure 6.4. There exists only one topological sort $\prec$ of $MC_{\mathbf{P}}$ with respect to S: $\{(m_3, m_1), (m_3, m_2), (m_2, m_1)\}$, viz., $m_3 \prec m_2 \prec m_1$.

We now define rewriting sequences, which are used to fix a sequence of rewriting steps for the MLP $\mathbf{P}$. Such sequences do not guarantee that we end up in a "rewriting fix point," but for certain sequences—called admissible—we can guarantee that the outcome of the applied rewriting steps end up in an MLP that has a disconnected part ready to be removed from that program.

Definition 6.25 (Rewriting sequence).
A *module call rewriting step* $\sigma = (s, t, p)$ *for MLP* $\mathbf{P}$ is the rewriting function

$$\sigma(\mathbf{P}) = \mathit{MCR}(\mathbf{P}, m_t, m_s, P_s[p]) \ ,$$

where $s, t \in \{1, \dots, n\}$ and p is a predicate symbol. A *rewriting sequence* θ is a sequence of module call rewriting steps $\sigma_1 \cdots \sigma_{\ell-1}\sigma_\ell$ such that $\theta(\mathbf{P}) = \sigma_\ell(\sigma_{\ell-1}(\cdots \sigma_1(\mathbf{P})))$.

Example 6.12 (cont'd) The rewriting steps $\sigma_1 = (m_3, m_1, q_1)$ and $\sigma_2 = (m_3, m_2, q_2)$ and $\sigma_3 = (m_2, m_1, q_1)$ are the rewriting functions $\sigma_1(\mathbf{P}) = \mathit{MCR}(\mathbf{P}, m_1, m_3, P_3[q_1])$, $\sigma_2(\mathbf{P}) = \mathit{MCR}(\mathbf{P}, m_2, m_3, P_3[q_2])$, and $\sigma_3(\mathbf{P}) = \mathit{MCR}(\mathbf{P}, m_1, m_2, P_2[q_1])$, respectively. Put together, there are several possible rewriting sequences $\theta_{i,j,k}$ of the form $\sigma_i\sigma_j\sigma_k$, for $i, j, k \in \{1, 2, 3\}$, as well as sub-sequences thereof.

Based on a topological sort $\prec$ of $\mathit{MC}_\mathbf{P}$ with respect to a set of modules S, we define admissible rewriting sequences next.

Definition 6.26 (Admissible rewriting sequence).
Let S be a set of modules from the MLP $\mathbf{P}$ and $\prec$ be a topological sort of $\mathit{MC}_\mathbf{P} = (V, E)$ with respect to S. A rewriting sequence $\theta = \sigma_1 \cdots \sigma_\ell$ of $\mathbf{P}$ is called *admissible with respect to* S, if the following conditions hold:

1. for each $\sigma_i = (s, t, p)$ of θ, both m_s and m_t are from S,
2. if $m_i \prec m_j$ and $m_j \prec m_k$ such that for all pairs $\sigma_a = (i, j, p_a), \sigma_b = (j, k, p_b)$ in θ, then σ_a appears before σ_b in θ,
3. for $(m_i, m_k, p) \in E$ such that $m_i, m_k \in S$ implies there exists a $\sigma_j = (i, k, p)$ in θ, and
4. for each $\sigma_i = (s_i, t_i, p_i)$ from θ, the module m_{s_i} is a Horn module.

Example 6.13 (cont'd) Let $S = \{m_1, m_2, m_3\}$ be a set of modules such that $m_3 \preceq m_2 \preceq m_1$ is a topological sort of $\mathit{MC}_\mathbf{P}$ for the MLP $\mathbf{P}$ from Example 6.9. Then, there are three admissible rewriting sequences $\sigma_1\sigma_2\sigma_3$, $\sigma_2\sigma_1\sigma_3$, and $\sigma_2\sigma_3\sigma_1$ with respect to S, as defined in Example 6.12.

Intuitively, (1) requires that all rewriting steps must work on the set of fixed modules S. Condition (2) ensures that the rewriting sequence θ respects the topological sort $\prec$ of $\mathbf{P}$. Condition (3) requires that all modules in S will be completely rewritten; both (2) and (3) then guarantee that in a particular module no call is left before the next module is rewritten. The last condition (4) makes sure that all modules except the root module are Horn, i.e., for the rewriting step $\sigma_\ell = (s_\ell, t_\ell, p_\ell)$, the module m_{t_ℓ} of $\mathbf{P}$ is not required to be a Horn module. This allows to rewrite a sub-dag of the transposed

graph of $MC_{\mathbf{P}}$ (i.e., every edge has reverse orientation) rooted in a generic module, where every leaf and every inner node consists of Horn modules to be rewritten until we reach the root module.

Following an admissible rewriting sequence θ with respect to a set of modules S, we can rewrite $\mathbf{P}$ such that we create an MLP $\theta(\mathbf{P})$ with a separated part S.

Proposition 6.7 (Module separation)
Let $\mathbf{P} = (m_1, \dots, m_n)$ be an MLP, let S be a set of modules from $\mathbf{P}$ and let $\theta = \sigma_1 \cdots \sigma_\ell$ be an admissible rewriting sequence of $\mathbf{P}$ with respect to S. Then, the answer sets of $\mathbf{P}$ correspond one-to-one to the answer sets of $\theta(\mathbf{P})$.

Proof We proceed by induction on h for $1 \leq h \leq \ell$ such that $\theta^h = \sigma_1 \cdots \sigma_h$. In the base case, we set $h = 1$ and have $\theta^1 = \sigma_1$ to be an admissible rewriting sequence with respect to S. Then, for a pair $m_i, m_k \in S$ such that $m_i \prec m_k$, we have $\sigma_1 = (i, k, p)$. Hence, $\sigma_1(\mathbf{P}) = MCR(\mathbf{P}, m_k, m_i, P_i[p])$, and by Lemma 6.6, we get that the answer sets of $\mathbf{P}$ and $\theta^1(\mathbf{P})$ correspond.

In the inductive step, we let $h > 1$ and assume that our proposition holds for all $j < h$. We prove that if the answer sets of $\mathbf{P}$ and $\theta^j(\mathbf{P})$ for all admissible rewriting sequences $\theta^j = \sigma_1 \cdots \sigma_j$ for $j < h$ coincide, then the answer sets of $\mathbf{P}$ are in one-to-one correspondence to $\theta^{j+1}(\mathbf{P})$ for the admissible rewriting sequence $\theta^{j+1} = \sigma_1 \cdots \sigma_j \sigma_{j+1}$. Now, let $\sigma_{j+1} = (i, r, p)$ for a pair $m_i, m_r \in S$ such that $m_i \prec m_r$. As $\sigma_{j+1}(\theta^j(\mathbf{P})) = MCR(\theta^j(\mathbf{P}), m_r, m_i, P_i[p])$, we get by Lemma 6.6 that the answer sets of $\theta^{j+1}(\mathbf{P})$ match the ones of $\theta^j(\mathbf{P})$, and by our assumption that $\theta^j(\mathbf{P})$ and $\mathbf{P}$ have corresponding answer sets, we get that the answer sets of $\theta^{j+1}(\mathbf{P})$ and $\mathbf{P}$ coincide. □

In general, module copy rewriting requires the addition of exponentially many rule copies in order to exhaustively rewrite an MLP. This can be seen as follows: only acyclic parts of the MLP can be rewritten, but we need to clone the rules of modules along all possible paths in the directed connection graph; in the worst case, we may have exponentially many paths. Take, as an example, a stack of diamond-shaped MLPs similar to the one of Example 6.9—i.e., when a top-most module of a lower diamond becomes the sink of an upper diamond—, we would need to copy the sink module into more than one module along the edges of $MC_{\mathbf{P}}$ in direction to the main module. When an anchor point module sharing two diamonds in the stack then joins its callee modules, it needs to copy also the two clones of the sink module. Continuing upwards towards the top-module of two diamond subgraphs, we end up in four clones of the sink module. Note that n module calls from one module to another module also requires to replicate the callee module n times. If the number of paths in the directed connection graph of an MLP is bounded by a polynomial, or the length of the paths is polynomially bounded, then module copy rewriting increases the size of the MLP only by polynomially many module clones. For instance, when the directed connection

graph of an MLP forms a binary tree, we can rewrite leaf modules to their successor modules and when we arrive at the top-most root module, we will have collected only one copy for each preceding module in the root module.

6.4.2 Module Removal of Separated Modules

Once we have (exhaustively) applied module copy rewriting on an MLP, we potentially end up with a rewritten MLP that has all source modules being separated from the target part of the MLP. That is, the part of the rewritten MLP that contains the last target module m_{t_ℓ} in the last rewriting step $\sigma_\ell = (s_\ell, t_\ell, p_\ell)$ of an admissible rewriting sequence $\theta = \sigma_1 \cdots \sigma_\ell$ is disconnected from all modules m_{s_1} to m_{s_ℓ}. All those m_{s_i} serve no purpose anymore when no other module is calling them, as the rewritten m_{t_ℓ} would then contain all the rule copies from m_{s_1} up to m_{s_ℓ}.

By inspecting the connected components of the *underlying graph* of the directed connection graph $MC_{\theta(\mathbf{P})}$—i.e., the undirected variant of $MC_{\theta(\mathbf{P})}$—, we can remove those modules of $\theta(\mathbf{P})$ that appear in singleton connected components. This means we can prune a module m_i whenever its connected component is $\{m_i\}$ in the underlying graph of $MC_{\theta(\mathbf{P})}$.

We now formally define the basic concepts for removing separated modules from a rewritten MLP $\theta(\mathbf{P})$. The *underlying graph* of an directed graph G is the graph UG obtained by replacing each directed edge of G by a corresponding undirected edge. Let S be a set of modules from an MLP $\mathbf{P}$ and let θ be an admissible rewriting sequence of $\mathbf{P}$ with respect to S. Let $UMC_{\theta(\mathbf{P})}$ be the underlying graph of $MC_{\theta(\mathbf{P})} = (V, E)$ for the MLP $\theta(\mathbf{P})$. A module m from $\theta(\mathbf{P})$ is called *free to prune* if the singleton set $\{m\}$ is a connected component of $UMC_{\theta(\mathbf{P})}$ and the MLP $\mathbf{P}_m = (m)$ has at least one minimal model, i.e., $\mathrm{MM}(\mathbf{P}_m) \neq \emptyset$. The set of modules $P \subseteq S$ is called a *pruning set with respect to* θ if for a rewriting step $\sigma = (s, t, p)$ from θ, every module $m_s \in P$ is free to prune, and no module $m' \in S$ is free to prune such that $m' \notin P$.

We can now show the following.

Lemma 6.8
Let $\mathbf{P} = (m_1, \ldots, m_n)$ be an MLP, let S be a set of free to prune modules from $\mathbf{P}$. Then, if $\mathbf{M}$ is an answer set of $\mathbf{P}$ there exists an answer set $\mathbf{M}'$ of $\mathbf{P} - S$ such that $M'_j/T = M_j/T$ for each $m_j \notin S$.

Proof We show now that $\mathbf{M}'$ is an answer set of $\mathbf{P} - S$, i.e., we show that both (a) $\mathbf{M}' \vDash f\,(\mathbf{P} - S)^{\mathbf{M}'}$, and that (b) $\mathbf{M}'$ is a minimal model of $f\,(\mathbf{P} - S)^{\mathbf{M}'}$.
We commence with item a. As $\mathbf{M}$ is an answer set of $\mathbf{P}$, it satisfies $\mathbf{M}, P_i[T] \vDash f\,\mathbf{P}^{\mathbf{M}}$ for each $P_i[T] \in \mathrm{VC}(\mathbf{P})$. Thus, $\mathbf{M}', P_i[T] \vDash f\,(\mathbf{P} - S)^{\mathbf{M}'}$.
Next, in item b, since all $m_i \in S$ are free to prune, there are no calls of form $P_i[p].o(\mathbf{c})$ in the rules from m_j of $\mathbf{P} - S$. This follows from $\{m_i\}$ being a connected component from

$UMC_{\mathbf{P}}$. Furthermore, m_i must have a minimal model, and $\mathbf{M}$ captures one of them with the interpretation $(M_i/T \mid P_i[T] \in \mathrm{VC}(\mathbf{P}))$ for m_i. Towards a contradiction, assume that there exists an interpretation $\mathbf{M}'' < \mathbf{M}'$ such that $\mathbf{M}'' \models f(\mathbf{P} - S)^{\mathbf{M}'}$. Now let $\mathbf{N}$ be an interpretation for $\mathbf{P}$ such that $N_i/T = M_i/T$ for each $m_i \in S$ and $M_j/T = M''_j/T$ for each $m_j \notin S$. It is clear that $\mathbf{N} < \mathbf{M}$, and since $\mathbf{M}$ is a minimal model of $f\,\mathbf{P}^{\mathbf{M}}$, there must be a rule r from some module $m_j \notin S$ such that for the value call $P_j[T]$, $\mathbf{N}, P_j[T] \not\models r$. The same rule r must exist in $f(\mathbf{P} - S)(P_j[T])^{\mathbf{M}'}$. Hence, $\mathbf{M}'', P_j[T] \not\models r$ in $\mathbf{P} - S$, and thus $\mathbf{M}'', P_j[T] \not\models f(\mathbf{P} - S)(P_j[T])^{\mathbf{M}'}$, which contradicts our assumption that $\mathbf{M}'' \models f(\mathbf{P} - S)^{\mathbf{M}'}$. We can now conclude that $\mathbf{M}'$ is a minimal model of $f(\mathbf{P} - S)^{\mathbf{M}'}$. □

Lemma 6.9
Let $\mathbf{P} = (m_1, \ldots, m_n)$ be an MLP such that for the modules m_k and m_i from $\mathbf{P}$

- the module atom $P_i[p].o(\mathbf{t})$ appears in m_k,
- m_i is a Horn module, and
- $UMC_{MCR(\mathbf{P},m_k,m_i,P_i[p])}$ has the connected component $\{m_i\}$.

Then,

1. for an answer set $\mathbf{M}$ of $MCR(\mathbf{P}, m_k, m_i, P_i[p])$ there exists an answer set $\mathbf{M}'$ of the MLP $MCR(\mathbf{P}, m_k, m_i, P_i[p]) - \{m_i\}$ such that for all $P_j[T] \in \mathrm{VC}(\mathbf{P})$, $j \neq i$,

$$M'_j/T = M_j/T \ ; \tag{6.20}$$

2. for an answer set $\mathbf{M}'$ of $MCR(\mathbf{P}, m_k, m_i, P_i[p]) - \{m_i\}$ there exists an answer set $\mathbf{M}$ of the MLP $MCR(\mathbf{P}, m_k, m_i, P_i[p])$ such that for all $P_j[T] \in \mathrm{VC}(\mathbf{P})$,

$$M_j/T = \begin{cases} untag(M'_k/S, P_i[p]) & \text{if } j = i \text{ and } T = (M_k/S)|_p^{q_i}, \\ M'_j/T & \text{otherwise.} \end{cases} \tag{6.21}$$

Proof Note that item 1 follows from Lemma 6.8, since m_i is free to prune in module call rewriting $\mathbf{P}' = MCR(\mathbf{P}, m_k, m_i, P_i[p])$, as m_i is a Horn module and m_i is disconnected in $\mathbf{P}'$.

What remains to show is item 2. As $\mathbf{M}'$ is equal to $\mathbf{M}$ on the part without value calls $P_i[T]$, what need to show that $\mathbf{M}^h = (M_i/T \mid P_i[T] \in \mathrm{VC}(\mathbf{P}))$ is a minimal model for the MLP (m_i), since m_i is disconnected in $MCR(\mathbf{P}, m_k, m_i, P_i[p])$ and therefore m_i does not call other modules. Hence, the rules in module m_k from $MCR(\mathbf{P}, m_k, m_i, P_i[p])$ that

correspond to m_i do neither call other modules. Therefore, as m_i is Horn, the rules in the reduct $f(m_i)^{\mathbf{M}^h}$ correspond one-to-one to the rules in the reduct

$$f(MCR(\mathbf{P}, m_k, m_i, P_i[p]) - \{m_i\})(P_k[S])^{\mathbf{M}'} \quad ,$$

i.e., $r \in f(m_i)(P_i[T])^{\mathbf{M}^h}$ if and only if

$$\mathcal{CT}(r, P_i[p]) \in f(MCR(\mathbf{P}, m_k, m_i, P_i[p]) - \{m_i\})(P_k[S])^{\mathbf{M}'} \quad .$$

As $\mathbf{M}'$ is a model for the MLP $f\,MCR(\mathbf{P}, m_k, m_i, P_i[p]) - \{m_i\}(P_k[S])^{\mathbf{M}'}$, we get that $\mathbf{M}^h$ is a model for $f(m_i)^{\mathbf{M}^h}$. We show now that $\mathbf{M}^h$ is minimal. Towards a contradiction, assume that $\mathbf{N}^h$ is an interpretation of (m_i) such that $\mathbf{N}^h < \mathbf{M}^h$. Let $\mathbf{M}''$ be an interpretation for the MLP $MCR(\mathbf{P}, m_k, m_i, P_i[p]) - \{m_i\}$ such that $M''_j/T = M'_j/T$ for all $j \neq k$, and $M''_k/T = notag(M'_k/T, P_i[p]) \cup N^h_i/S$ such that $S = (M_k/T)|^{q_i}_p$. Therefore, $\mathbf{M}'' < \mathbf{M}'$ and thus $\mathbf{M}'', P_k[T] \not\models f\,MCR(\mathbf{P}, m_k, m_i, P_i[p]) - \{m_i\}^{\mathbf{M}'}$. There must be a rule r' contained in the MLP $f\,MCR(\mathbf{P}, m_k, m_i, P_i[p]) - \{m_i\}(P_k[T])^{\mathbf{M}'}$ such that $\mathbf{M}'', P_k[T] \not\models r'$. Since $\mathbf{M}''$ differs from $\mathbf{M}'$ in the atoms that correspond to m_i, we get that $r' = \mathcal{CT}(r, P_i[p])$ for a rule r from m_i. Now as the rules in the reduct $f(m_i)^{\mathbf{M}^h}$ and the rules in the reduct $f\,MCR(\mathbf{P}, m_k, m_i, P_i[p]) - \{m_i\}(P_k[T])^{\mathbf{M}'}$ corresponding to m_i match each other using $\mathcal{CT}(\cdot,\cdot)$, we have that $\mathbf{N}^h, P_i[S] \not\models r'$, and hence $\mathbf{N}^h$ is not a model for $f(m_i)^{\mathbf{M}^h}$, which contradicts our assumption. Hence, $\mathbf{M}^h$ is a minimal model of $f(m_i)^{\mathbf{M}^h}$. Since m_i is Horn, we get that $\mathbf{M}$ is an answer set of MLP $MCR(\mathbf{P}, m_k, m_i, P_i[p])$. □

Rewriting a set of modules requires to ensure that all calls from a calling module to another callee module will be completely rewritten, before we can apply macro rewriting to the next module. To this end, we refine the definition for admissible rewriting sequences and define module-oriented rewriting sequences next.

Definition 6.27 (Module-oriented rewriting sequence).
Let $\theta = \sigma_1 \cdots \sigma_\ell$ be an admissible rewriting sequence for MLP $\mathbf{P}$ with respect to a set of modules S, then θ is called a *module-oriented rewriting sequence with respect to* S, if for all subsequences $\sigma_i \cdots \sigma_j \cdots \sigma_k$ of θ such that $i \leq j \leq k$ and $\sigma_\star = (s_\star, t_\star, p_\star)$ for $\star \in \{i, j, k\}$, if $s_i = s_k$ then the following two hold:

1. $s_i = s_j \wedge t_i = t_j = t_k$
2. $s_i \neq s_j \rightarrow t_i \neq t_k$

Intuitively, module-oriented rewriting sequences enforce that all calls from a particular module to another module will be rewritten using a continuous sequence of

rewriting steps. Condition (1) is a safeguard that there is no intermediate rewriting step σ_j that copies a module m_{s_j} different from m_{s_i} into module m_{t_i} before all calls from m_{t_i} to m_{s_i} have been rewritten. The second condition (2) guarantees that whenever there are two different modules m_{s_i} and m_{s_j} that get called by m_{t_i} and m_{t_k}, then m_{t_i} and m_{t_k} must be different.

For the following results, let $\mathbf{P} = (m_1, \ldots, m_n)$ be an MLP, let S be a set of modules from $\mathbf{P}$, let $\theta = \sigma_1 \cdots \sigma_\ell$ be a module-oriented rewriting sequence of $\mathbf{P}$ with respect to S, and let P be a pruning set with respect to θ. Note that module-oriented rewriting sequences rewrite Horn modules, thus even though $\theta(\mathbf{P})$ and $\theta(\mathbf{P}) - P$ do not share modules from P, this does not create additional answer sets.

Proposition 6.10 (Module pruning)
The answer sets of $\theta(\mathbf{P})$ correspond one-to-one to the answer sets of $\theta(\mathbf{P}) - P$.

Proof We proceed by induction on h for $1 \leq h \leq \ell$ such that $\theta^h = \sigma_1 \cdots \sigma_h$. In the base case, we set $h = 1$ and have $\theta^1 = \sigma_1$ to be a module-oriented rewriting sequence with respect to S. Then, for a pair $m_i, m_k \in S$ such that $m_i \prec m_k$, we have $\sigma_1 = (i, k, p)$ and $P^1 = \{m_i\}$. Hence, $\sigma_1(\mathbf{P}) = \mathit{MCR}(\mathbf{P}, m_k, m_i, P_i[p])$, and by Lemma 6.9, we get that the answer sets of $\theta^1(\mathbf{P})$ and $\theta^1(\mathbf{P}) - P^1$ correspond.

In the inductive step, we let $h > 1$ and assume that our proposition holds for all $j < h$. We prove that if the answer sets of $\theta^j(\mathbf{P})$ and $\theta^j(\mathbf{P}) - P^j$ coincide for all module-oriented rewriting sequences $\theta^j = \sigma_1 \cdots \sigma_j$ for $j < h$ such that $P^j = \{m_{s_1}, \ldots, m_{s_j}\}$ is a pruning set with respect to θ^j, then the answer sets of $\theta^{j+1}(\mathbf{P})$ are in one-to-one correspondence to $\theta^{j+1}(\mathbf{P}) - P^{j+1}$ for the module-oriented rewriting sequence $\theta^{j+1} = \sigma_1 \cdots \sigma_j \sigma_{j+1}$ and the pruning set $P^{j+1} = P^j \cup \{m_{s_{j+1}}\}$. Now, let $\sigma_{j+1} = (i, r, p)$ for a pair $m_i, m_r \in S$ such that $m_i \prec m_r$. As $\sigma_{j+1}(\theta^j(\mathbf{P})) = \mathit{MCR}(\theta^j(\mathbf{P}), m_r, m_i, P_i[p])$, we get by Lemma 6.6 that the answer sets of $\theta^{j+1}(\mathbf{P})$ match the ones of $\theta^j(\mathbf{P})$, and by our assumption that $\theta^j(\mathbf{P})$ and $\theta^j(\mathbf{P}) - P^j$ have corresponding answer sets, we get that the answer sets of $\theta^{j+1}(\mathbf{P})$ and $\theta^{j+1}(\mathbf{P}) - P^{j+1}$ coincide. □

The next result then immediately follows from Proposition 6.7 and Proposition 6.10.

Corollary 6.11 (Separated module removal)
There exists a one-to-one correspondence between the set of answer sets of $\mathbf{P}$ and the set of answer sets of $\theta(\mathbf{P}) - P$.

6.5 Application: Description Logic Programs

In this section, we will show how to use MLP macro expansion from §6.4 as an application to implement dl-programs (Eiter et al., 2008), which is a prominent formalism for representing hybrid knowledge bases based on Description Logics (DLs) and logic

programs. To this end, we first show how to translate dl-programs based on the class of Datalog-rewritable Description Logics into MLPs. Datalog-rewritable DLs have been studied intensively over the last years as an efficient means to implement DL reasoning by translating ontologies and queries into Datalog programs. This class of DLs comprises of DLs such as $\mathcal{LDL}^+$ (Heymans et al., 2010), Horn-$\mathcal{SHIQ}$ (Eiter et al., 2012c), $\mathcal{RL}$ (Krötzsch et al., 2013), $\mathcal{SROEL}(\sqcap, \times)$ (Krötzsch, 2011), and more (see also (Krötzsch et al., 2015; Xiao, 2013) for an overview on Datalog-rewritable DLs). Several studies on ontology-based data access (Poggi et al., 2008) use Datalog (Bienvenu et al., 2013; Gottlob et al., 2014) as a basis. Following that, we apply macro expansion to the translated dl-programs and thus show how to efficiently evaluate dl-programs using MLPs. Interestingly, the results here reduce in principle to the rewritings and optimization techniques developed by Xiao (2013), the main difference being that their translations are ad hoc for Datalog-rewritable DLs. We assume familiarity with Description Logics and dl-programs as defined by Eiter et al. (2008).

6.5.1 Rewriting Description Logic Programs to MLPs

Let $KB = (L, P)$ be a dl-program, let $\lambda = S_1 op_1 p_1, \dots, S_m op_m p_m$ be an input list appearing in a dl-atom from P, and let $\mathbf{C} = \{C_1, \dots, C_k\}$ and $\mathbf{R} = \{R_1, \dots, R_\ell\}$ be the set of atomic concepts and the set of atomic roles from L, respectively.

We start with the definitions for rewriting the Description Logic knowledge base L.

Definition 6.28 (DL module).
The *DL module for* L is defined as the module

$$m_L = (P_L[C_1, \dots, C_k, R_1, \dots, R_\ell], \Psi(L)) \ ,$$

such that $\Psi(L) = \Phi(L) \cup T_P$, where

- $\Phi(L)$ is a transformation from DL knowledge base L to a Datalog program, and
- T_P is the set of facts $\mathsf{T}(a)$ and $\mathsf{T}^2(a, b)$ for each constant a, b in the Herbrand universe of P.

Note that Φ is a generic transformation for Datalog-rewritable Description Logics L; confer Xiao (2013, Definition 4.1) for the details. For simplicity, we assume that all Description Logics knowledge bases are using the $\mathcal{LDL}^+$ Description Logic (Xiao, 2013), which is a Datalog-rewritable Description Logic. Thus, in the rest of this section, we identify Φ with the transformation $\Phi_{\mathcal{LDL}^+}$ defined by Xiao (2013, Section 4.2.2).

Example 6.14 Let $KB = (L, P)$ be a dl-program over $\mathcal{LDL}^+$ Description Logic knowledge base $L = \{C \sqsubseteq D, D \sqsubseteq \forall R.E\}$, where C, D, E are atomic concepts and R is an

atomic role, and the logic program

$$P = \left\{ \begin{array}{l} c(a) \leftarrow \\ r(a,b) \leftarrow \\ q(X) \leftarrow \mathrm{DL}[C \uplus c, R \uplus r; E](X) \end{array} \right\} .$$

Then, $\Psi(L) = \Phi(L) \cup T_P$ consists of

$$\Phi(L) = \left\{ \begin{array}{l} D(X) \leftarrow C(X) \\ E(Y) \leftarrow D(X), R(X,Y) \end{array} \right\} .$$

and $T_P = \{\top(a), \top(b), \top^2(a,b), \top^2(b,a)\}$, and the DL module for L is

$$m_L = (P_L[C, D, E, R], \Psi(L)) .$$

Next, we give definitions for rewriting input lists.

Definition 6.29 (Transfer module).
For a concept or role S_i from a input list λ, we define the *transfer rule* $\rho(S_i)$ as

$$\rho(S_i) = \begin{cases} c_{S_i}(X) \leftarrow q_i(X) & \text{if } S_i \text{ is an atomic concept} \\ r_{S_i}(X,Y) \leftarrow q_i(X,Y) & \text{if } S_i \text{ is an atomic role} \end{cases} .$$

The transfer rules for λ is the set of rules $\rho(\lambda) = \{\rho(S_i) \mid S_i \text{ is from } \lambda\}$.

For a dl-atom $q = \mathrm{DL}[\lambda; Q](\mathbf{t})$, we define the *query rule* $\vartheta(q)$ as the rule

$$Q(\mathbf{t}) \leftarrow P_L[c_{C_1}, \ldots, c_{C_k}, r_{R_1}, \ldots, r_{R_\ell}].Q(\mathbf{t}) .$$

The query rules for λ is the set of rules

$$\vartheta(\lambda) = \{\vartheta(q) \mid \text{dl-atom } q \text{ appears in } P \text{ with input list } \lambda\} .$$

Given an input list $\lambda = S_1 op_1 p_1, \ldots, S_m op_m p_m$ from a rule appearing in the dl-program $KB = (L, P)$, we define the *transfer module for* λ as $m_\lambda = (P_\lambda[q_1, \ldots, q_m], R_\lambda)$, where q_i is an input predicate matching the arity of p_i and $R_\lambda = \rho(\lambda) \cup \vartheta(\lambda)$.

Intuitively, the module atom $P_L[c_{C_1}, \ldots, c_{C_k}, r_{R_1}, \ldots, r_{R_\ell}].Q(\mathbf{t})$ stands for the DL atom $q = \mathrm{DL}[\lambda; Q](\mathbf{t})$, and $\vartheta(q)$ then assigns a truth value to Boolean DL queries $Q(\mathbf{c})$ based on the truth values for ordinary atoms $Q(\mathbf{c})$ in DL module m_L for the input list λ encoded by the module input $c_{C_1}, \ldots, c_{C_k}, r_{R_1}, \ldots, r_{R_\ell}$. Note that we can use $Q(\mathbf{t})$ in m_L since Φ is defined to be a preserving transformation (Xiao, 2013), i.e., concept and roles in L are mapped to identically named predicates in $\Phi(L)$.

Example 6.15 (cont'd) For the input list $\lambda = C \uplus c, R \uplus r$ of dl-atom

$$a = \mathrm{DL}[C \uplus c, R \uplus r; E](X)$$

from KB in the previous example, the transfer rules are

$$\rho(\lambda) = \left\{ \begin{array}{c} c_C(X) \leftarrow q_c(X) \\ r_R(X,Y) \leftarrow q_r(X,Y) \end{array} \right\} ,$$

the query rule for a is

$$\vartheta(a) = E(X) \leftarrow P_L[c_C, c_D, c_E, r_R].E(X) \quad ,$$

and the query rules for λ are $\vartheta(\lambda) = \{\vartheta(a)\}$. The transfer module m_λ is the MLP module $(P_\lambda[q_c, q_r], R_\lambda)$, where $R_\lambda = \rho(\lambda) \cup \vartheta(\lambda)$.

We can now formally define the dl-program rewriting.

Definition 6.30 (dl-program rewriting).
Let $\Lambda = \{\lambda_1, \ldots, \lambda_j\}$ be the set of all input lists that appear in a dl-atom from the dl-program $KB = (L, P)$. The *dl-program rewriting of KB* is the MLP

$$\Delta(KB) = (m_P, m_L, m_{\lambda_1}, \ldots, m_{\lambda_j})$$

such that $m_P = (P[], P')$ is the main module, where P' is the program P with each dl-atom $\mathrm{DL}[\lambda; Q](\mathbf{t})$ replaced by the module atom $P_\lambda[p_1, \ldots, p_m].Q(\mathbf{t})$ such that $\lambda = S_1 op_1 p_1, \ldots, S_m op_m p_m$.

Example 6.16 (cont'd) The dl-program rewriting $\Delta(KB)$ for our running example is the MLP (m_P, m_L, m_λ), where $m_P = (P[], P')$ such that

$$P' = \left\{ \begin{array}{c} c(a) \leftarrow \\ r(a,b) \leftarrow \\ q(X) \leftarrow P_\lambda[c,r].E(X) \end{array} \right\} .$$

For the next results, we let I be an interpretation for dl-program KB and define I^Ψ in analogy to Xiao (2013, Lemma 4.5) and let

$$S(\lambda, I) = \{q_i(\mathbf{c}) \mid S_i \uplus p_i \text{ is in } \lambda \text{ and } p_i(\mathbf{c}) \in I\} \ .$$

Lemma 6.12
Let I be an interpretation for dl-program KB and $Q(\mathbf{c})$ be a Boolean DL query. Then, $I^\Psi \vDash Q_\lambda(\mathbf{c})$ if and only if $\mathbf{M}, P_\lambda[S] \vDash Q(\mathbf{c})$ such that $S = S(\lambda, I)$, where

- $M_P/\varnothing = I$,

- $M_L/T = \mathrm{MM}\,(\Psi(L) \cup T)$ for all $P_L[T]$, and
- for $\rho(T) = \{c_{S_i}(a) \mid q_i(a) \in T\} \cup \{r_{S_i}(a,b) \mid q_i(a,b) \in T\}$, we let

$$M_\lambda/T = T \cup \rho(T) \cup \left\{ Q(\mathbf{c}) \;\middle|\; Q(\mathbf{c}) \in M_L/S \text{ such that } S = \rho(T)|^{C_1,\ldots,C_k,R_1,\ldots,R_\ell}_{c_{C_1},\ldots,c_{C_k},r_{R_1},\ldots,r_{R_\ell}} \right\}$$

for all $P_\lambda[T]$.

Proof We prove both directions.

$$\begin{aligned}
I^\Psi \vDash Q_\lambda(\mathbf{c}) &\Leftrightarrow Q_\lambda(\mathbf{c}) \in \mathrm{MM}\left(\Psi(L_\lambda \cup \{S_{i_\lambda}(\mathbf{c}_i) \mid p_i(\mathbf{c}_i) \in I\})\right) && (6.22)\\
&\Leftrightarrow Q(\mathbf{c}) \in \mathrm{MM}\,(\Psi(L) \cup \{S_i(\mathbf{c}_i) \mid p_i(\mathbf{c}_i) \in I\}) && (6.23)\\
&\Leftrightarrow Q(\mathbf{c}) \in M_L/T \text{ for } T = \{S_i(\mathbf{c}_i) \mid p_i(\mathbf{c}_i) \in I\} && (6.24)\\
&\Leftrightarrow Q(\mathbf{c}) \in M_\lambda/T \text{ for } T = \{q_i(\mathbf{c}_i) \mid p_i(\mathbf{c}_i) \in I\} && (6.25)\\
&\Leftrightarrow \mathbf{M}, P_\lambda[S] \vDash Q(\mathbf{c}) \text{ for } S = S(\lambda, I) && (6.26)
\end{aligned}$$

The first equivalence (6.22) follows from Xiao (2013, Lemma 4.5). In (6.23) we use the modularity property of Ψ of Xiao (2013, Definition 4.1) and that Ψ preserves names in L. Equivalence (6.24) holds since $M_L/T = \mathrm{MM}\,(\Psi(L) \cup T)$. For (6.25) we use that M_λ/T first translates q_i to the concept and role atoms c_{S_i} and r_{S_i} with $\rho(T)$ and then maps them as input to $P_L[C_1, \ldots, C_k, R_1, \ldots, R_\ell]$, thus $q_i(\mathbf{c}) \in T$ if and only if $p_i(\mathbf{c}_i) \in I$. The last equivalence (6.26) then follows immediately by setting S to T. □

We can now show the following.

Proposition 6.13 (DL-rewriting)
Let $KB = (L, P)$ be a dl-program over a Datalog-rewritable Description Logic. Then, the answer sets of KB correspond one-to-one to the answer sets of $\Delta(KB)$.

Proof Based on Xiao (2013, Lemma 4.5), we can extend an answer set M from KB to an answer set M^Ψ for $\Psi(KB)$. Let $\mathbf{M}^\Delta$ be an interpretation based on M^Ψ for $\Delta(KB)$ as defined in Lemma 6.12. We get that $M^\Psi \vDash Q_\lambda(\mathbf{c})$ if and only if $\mathbf{M}, P_\lambda[S] \vDash Q(\mathbf{c})$ such that $S = S(\lambda, M)$. Hence, the FLP-reduct $f\,\Delta(KB)(P[\varnothing])^{\mathbf{M}^\Delta}$ is equivalent to the strong dl-transform sP_L^M, which follows from $M_P/\varnothing = M$. □

6.5.2 Macro Expansion for dl-Programs

In this section we show how to apply macro expansion introduced in §6.4 to dl-program rewriting. The key to macro expansion is defining a DL module rewriting sequence for macro expansion. Figure 6.5 shows the module dependencies of the MLP $\Delta(KB)$

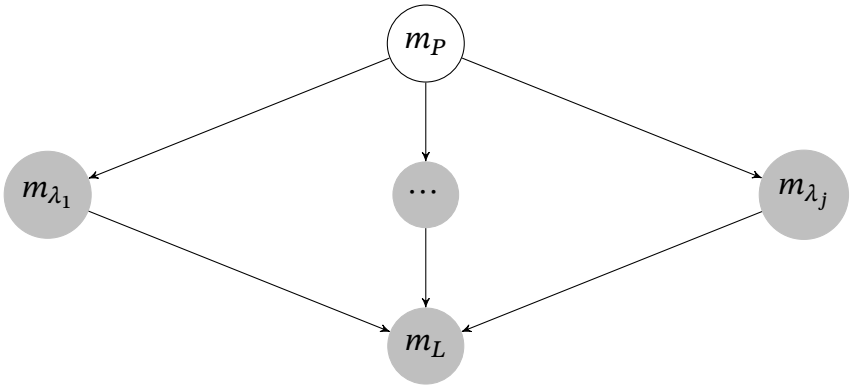


Figure 6.5: Module dependencies of dl-program rewriting

for a given dl-program KB. Clearly, the module dependencies are acyclic, hence we can copy the DL module m_L into transfer modules m_{λ_i}, and then copy the result into m_P. When we do this for all input lists $\lambda_1, \ldots, \lambda_j$ appearing in KB, we end up in an MLP that consists of a rewritten main module m_P containing j copies of m_L and all transfer modules m_{λ_i}. After applying module removal using modules $m_L, m_{\lambda_1}, \ldots, m_{\lambda_j}$ as separated part as shown in §6.4.2, we have a single main module MLP without input as result, which can be evaluated in a Datalog engine.

Let $KB = (L, P)$ be a dl-program over a Datalog-rewritable Description Logic L. In the following, we assume that the module calls of $\Delta(KB)$ have been reified using the technique from §6.1 such that each library module has exactly one input parameter, i.e., the DL module $m_L = (P_L[q_L], \Psi(L))$ has the input parameter q_L and transfer modules $m_\lambda = (P_\lambda[q_\lambda], R_\lambda)$ have input parameter q_λ. Furthermore, the module calls in m_λ have p_L as predicate parameter, and the main module m_P use p_λ as parameter.

Example 6.17 (cont'd) Given KB and $\Delta(KB)$ from Example 6.16 such that module atom $e_1 = P_\lambda[c, r].E(X)$ and module atom $e_2 = P_L[c_C, c_D, c_E, r_R].E(X)$, the module input reified MLP $\Delta(KB)' = (m'_P, m'_L, m'_\lambda)$, where main module $m'_P = (P[], P')$ such that

$$P' = \left\{ \begin{array}{r} c(a) \leftarrow \\ r(a,b) \leftarrow \\ p^\lambda(1, X, \epsilon) \leftarrow c(X) \\ p^\lambda(2, X, Y) \leftarrow r(X, Y) \\ q(X) \leftarrow P_\lambda[p^\lambda].E(X) \end{array} \right\},$$

transfer module $m'_\lambda = (P_\lambda[q_\lambda], R_\lambda)$ with

$$R_\lambda = \rho(\lambda) \cup \left\{ \begin{array}{r} q_C(X) \leftarrow q_\lambda(1, X, \epsilon) \\ q_R(X, Y) \leftarrow q_\lambda(2, X, Y) \\ p^L(1, X, \epsilon) \leftarrow c_C(X) \\ p^L(2, X, \epsilon) \leftarrow c_D(X) \\ p^L(3, X, \epsilon) \leftarrow c_E(X) \\ p^L(4, X, Y) \leftarrow c_R(X, Y) \\ E(X) \leftarrow P_L[p^L].E(X) \end{array} \right\},$$

and DL module $m'_L = (P_L[q_L], R_L)$, where

$$R_L = \Psi(L) \cup \left\{ \begin{array}{r} C(X) \leftarrow q_L(1, X, \epsilon) \\ D(X) \leftarrow q_L(2, X, \epsilon) \\ E(X) \leftarrow q_L(3, X, \epsilon) \\ R(X, Y) \leftarrow q_L(4, X, Y) \end{array} \right\}.$$

Definition 6.31 (DL module rewriting sequence).
Let $\Lambda = \{\lambda_1, \ldots, \lambda_j\}$ be the set of all input lists that appear in a dl-atom from dl-program $KB = (L, P)$. We say that

$$\theta = \sigma_{1,P}\sigma_{1,L}\sigma_{2,P}\sigma_{2,L} \cdots \sigma_{j-1,P}\sigma_{j-1,L}\sigma_{j,P}\sigma_{j,L}$$

is a *DL module rewriting sequence*, where each rewriting step is defined as

- $\sigma_{i,L} = (m_L, m_{\lambda_i}, p_L)$ for $i \in \{1, \ldots, j\}$, and
- $\sigma_{i,P} = (m_{\lambda_i}, m_P, p_{\lambda_i})$ for $i \in \{1, \ldots, j\}$.

In the following, let $P_{KB} = \{m_L, m_{\lambda_1}, \ldots, m_{\lambda_j}\}$ be the set of modules from $\Delta(KB)$ without m_P. We can now show the following results.

Proposition 6.14 (DL-module separation)
The answer sets of MLP $\Delta(KB)$ correspond one-to-one to the answer sets of MLP $\theta(\Delta(\mathbf{P}))$.

Proof This follows from Proposition 6.7, since θ is an admissible rewriting sequence with respect to P_{KB}. □

The next result shows that we can remove all modules from P_{KB}, thus only m_P with the appropriate module copies is required to evaluate $\Delta(KB)$.

Proposition 6.15 (DL-module pruning)
The answer sets of MLP $\theta(\Delta(KB))$ correspond one-to-one to the answer sets of MLP $\theta(\Delta(\mathbf{P})) - P_{KB}$.

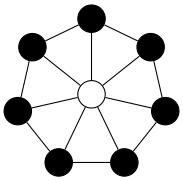
Proof This follows from Proposition 6.10, since θ is a module-oriented rewriting sequence and P_{KB} is a pruning set with respect to θ. □

The next result follows immediately from Proposition 6.13, Proposition 6.14, and Proposition 6.15.

Corollary 6.16 (Separated DL-module removal)
The answer sets of dl-program KB correspond one-to-one to the answer sets of the MLP $\theta(\Delta(KB)) - P_{KB}$.

One can extend this technique to rewrite only parts of $\Delta(KB)$. For instance, we may copy only the Horn-parts into m_P and leave the remainder alone. Subsequent module removals then only discards the transfer modules that were involved in the Horn-parts of $\Delta(KB)$.

Standard dl-programs only allow to access a single ontology L. As a mild extension, we may introduce dl-programs $KB = (P, L_1, \dots, L_k)$ such that P has dl-atoms accessing different ontologies L_i. With MLPs, such an extension can be swiftly captured by introducing k different DL modules $m_{L_1}, \dots, m_{L_k}$, which do not call any further modules. Transfer modules than prepare the input for each of them. Similarly, we may add further MLP modules to $\Delta(KB)$, which can be used in P from KB.



7

Representing Modular Nonmonotonic Logic Programs with Classical Logic

We further the work on MLPs and turn to characterizing answer sets in terms of classical models in this chapter, which is in line with recent research in Answer Set Programming. To this end, we first explore the notion of *loop formulas* to MLPs. Lin and Zhao (2004) first used loop formulas to characterize the answer sets of normal, i.e., disjunction-free, propositional logic programs by the models of a propositional formula comprised of the Clark completion (Clark, 1978) and of additional formulas for each positive loop in the dependency graph of the program. They built on this result by developing the ASP solver ASSAT, which uses a SAT solver for answer set computation (Lin and Zhao, 2004). The loop formula characterization has subsequently been extended to disjunctive logic programs (Lee and Lifschitz, 2003), and to general propositional theories under a generalized notion of answer set (Ferraris et al., 2006). In the latter work, the notion of a loop has been adapted to include trivial loops (singletons) in order to recast Clark's completion as loop formulas. Besides their impact on ASP solver development, loop formulas are a viable means for the study of semantic properties of ASP programs, as they allow to resort to classical logic for characterization. For instance, in the realm of modular logic programming, loop formulas have recently been fruitfully extended to DLP-functions (Janhunen et al., 2009b), simplifying some major proofs.

The expedient properties of MLPs, however, render a generalization of loop formulas more involved. Due to the module input mechanism, it is necessary to keep track of different module instantiations. Furthermore, because of unlimited recursion in addition to loops that occur inside a module, loops across module boundaries, i.e., when modules refer to each other by module atoms, have to be captured properly. To cope with this requirements,

- we adapt Clark's completion for module atoms with respect to different module instantiations;
- we provide a refined version of the positive dependency graph for an MLP, the *modular dependency graph*, and *cyclic instantiation signature*: the combination then relates module instantiations with the atoms of a module;
- based on it, we define *modular loops* and their external support formulas; and
- eventually, we define *modular loop formulas*, and show that the conjunction of all modular loop formulas for an MLP characterizes the answer sets of $\mathbf{P}$ in its (Herbrand) models.

Second, we explore the recent approach of Asuncion et al. (2012) to modify the Clark completion in order to characterize answer set semantics of nonmonotonic logic programs with finite Herbrand universes but without using loop formulas explicitly. The idea is to introduce predicates of the form $D^{q,P_k[T]}_{p,P_i[S]}(\mathbf{y},\mathbf{x})$ which intuitively holds when $q(\mathbf{y})$ at value call $P_k[T]$ is used to derive $p(\mathbf{x})$ at value call $P_i[S]$, and to respect a derivation order. The completion is allowed to take effect only if no positive loop is present, which is ensured by adding $D^{q,P_k[T]}_{p,P_i[S]}(\mathbf{y},\mathbf{x}) \wedge \neg D^{p,P_i[S]}_{q,P_k[T]}(\mathbf{x},\mathbf{y})$ in the completion of rules with head $p(\mathbf{x})$ and $q(\mathbf{y})$ in the positive body. For that to work, we must ensure that $D^{q,P_k[T]}_{p,P_i[S]}(\mathbf{y},\mathbf{x})$ respects transitive derivations, i.e., the composition of $D^{p,P_i[S]}_{q,P_j[T]}(\mathbf{x},\mathbf{y})$ and $D^{q,P_j[T]}_{r,P_k[U]}(\mathbf{y},\mathbf{z})$ must be contained in $D^{p,P_i[S]}_{r,P_k[U]}(\mathbf{x},\mathbf{z})$. The resulting translation is called *ordered completion*.

An advantage of this approach is that, at the cost of fresh (existential) predicates, constructing the (possible exponentially) many loop formulas can be avoided, while answer sets may be extracted from the (Herbrand) models of a first-order sentence, which may be fed into a suitable theorem prover. This similarly applies to MLPs, where unrestricted call by value however leads to an unavoidable blowup, which may be avoided by resorting to higher-order logic. Independent of computational perspectives, the novel characterizations widen our understanding of MLPs and they may prove useful for semantic investigations, similarly to those by Janhunen et al. (2009b).

In this chapter, we restrict our investigations to normal MLPs. There is no principal obstacle to extend the loop formula encoding given here to disjunctive MLPs, and doing this would require to change just aspects of the propositional formulas given here without changing the structure of the formulas. In contrast, ordered completion formulas for disjunctive MLPs and already ordinary logic programs need further work; they may require a blowup given that ordinary disjunctive Datalog programs are $\mathrm{NEXP}^{\mathrm{NP}}$-complete.

In the sequel, we will characterize the answer sets of normal MLPs

- via loop formulas and program completion,

- via second-order logic formulation, and
- via ordered completion.

Such characterizations consist of the following parts:

- the completion, which singles out classical models, which is studied in §7.1;
- the loop formulas, which take care of minimality (foundedness) aspects; this will be considered in §7.2;
- a translational semantics using second-order logic, which is based on completion and minimality formulation in §7.3.1;
- alternatively, the completion can be made ordered, which we present in §7.3.2.

7.1 Program Completion for MLPs

We start with adapting the classical Clark completion (Clark, 1978) to cater for module atoms. The intuition behind this adaption is to replace every module atom $\beta(\mathbf{y}) = P_k[\mathbf{p}].o(\mathbf{y})$ in module m_i by a formula $\mu(P_i[S], \beta(\mathbf{y}))$, which selects the value call $P_k[T]$ for m_k and the truth value for $o(\mathbf{c})$ based on the value of the input atoms $\mathbf{p}$ for value call $P_i[S]$.

Throughout this chapter we assume that predicate names appearing in the modules of an MLP $\mathbf{P}$ are not shared in two different modules of $\mathbf{P}$. Furthermore, we assume that the variables in the head of all rules in $\mathbf{P}$ mentioning the predicate symbol a share the same distinct variables $\mathbf{x}$, i.e., $H(r) = \{a(\mathbf{x})\}$ for all rules r with predicate a in the head. Additionally, we assume that $\mathbf{P}$ does not contain constant symbols; every constant in $\mathbf{P}$ can be replaced by a singleton unary relation.

Given a set $S \subseteq \mathrm{HB}_\mathbf{P}$ of ordinary atoms, we assume that S is enumerated, i.e., $S = \{a_1, \dots, a_n\}$ where $n = |S|$. We identify subsets B of S with their characteristic function $\chi^B : S \to \{0, 1\}$ such that $\chi^B(a) = 1$ iff $a \in B$.

For any ordinary atom $a(\mathbf{x})$ and any set of ground ordinary atoms A, let $a^A(\mathbf{x})$ denote a fresh atom, and for any set B of ordinary atoms, let $B^A = \{a^A(\mathbf{x}) \mid a(\mathbf{x}) \in B\}$. Let $\neg.A = \{\neg a(\mathbf{x}) \mid a(\mathbf{x}) \in A\}$ and, as usual, $\bigvee F = \bigvee_{f \in F} f$ and $\bigwedge F = \bigwedge_{f \in F} f$. Note that $\bigvee \varnothing = \bot$ and $\bigwedge \varnothing = \top$.

Given a rule r of form (3.2), for $\star \in \{+, -\}$, let $B^\star_o(r)$ and $B^\star_m(r)$ be the sets of ordinary and module atoms appearing in $B^\star(r)$, respectively.

For the following definitions, let $\mathbf{P}$ be a normal MLP, and let $m_i = (P_i[\mathbf{q}_i], R_i)$ and $m_k = (P_k[\mathbf{q}_k], R_k)$ be two modules from $\mathbf{P}$.

Definition 7.1 (Module atom completion).
Let $\beta(\mathbf{z}) = P_k[\mathbf{p}].o(\mathbf{z})$ be a module atom appearing in a rule from module m_i, let $\mathbf{q}_k = q_{k,1}, \ldots, q_{k,\ell}$ be the formal input parameter for module m_k, and let $P_i[S]$ and $P_k[T]$ be two value calls from $\mathrm{VC}(\mathbf{P})$. We define

$$\epsilon(P_i[S], P_k[T]) = \bigwedge_{j=1}^{\ell} \bigwedge_{\chi^T(q_{k,j}(\mathbf{c}))=1} p_j^S(\mathbf{c}) \wedge \bigwedge_{\chi^T(q_{k,j}(\mathbf{c}))=0} \neg p_j^S(\mathbf{c}) \ .$$

The *module atom completion formulas* are defined as

$$\mu(P_i[S], \beta(\mathbf{z})) = \bigvee_{P_k[T] \in \mathrm{VC}(\mathbf{P})} \left(\epsilon(P_i[S], P_k[T]) \wedge o^T(\mathbf{z})\right)$$

and

$$\bar{\mu}(P_i[S], \beta(\mathbf{z})) = \bigvee_{P_k[T] \in \mathrm{VC}(\mathbf{P})} \left(\epsilon(P_i[S], P_k[T]) \wedge \neg o^T(\mathbf{z})\right) \ .$$

Intuitively, $\epsilon(P_i[S], P_k[T])$ encodes the module input for calls from value call $P_i[S]$ to $P_k[T]$. The module atom completion formulas $\mu(P_i[S], \beta(\mathbf{z}))$ and $\bar{\mu}(P_i[S], \beta(\mathbf{z}))$ select from a given value call $P_i[S]$ (the calling module) one of the target value calls $P_k[T]$ such that some instance of $o^T(\mathbf{z})$ is true in $\mu(P_i[S], \beta(\mathbf{z}))$ whenever $\beta(\mathbf{z})$ appears in the positive body of a rule, and if $\beta(\mathbf{z})$ appears in the negative body of a rule, then we use $\bar{\mu}(P_i[S], \beta(\mathbf{z}))$ to address an instance of $o^T(\mathbf{z})$ being false.

Next we define support rules.

Definition 7.2 (Support rules).
Let R be a set of normal rules, then the *support rules* of R with respect to an ordinary atom $a(\mathbf{t})$ is

$$\mathrm{SR}(a(\mathbf{t}), R) = \{r \in R \mid H(r) = \{a(\mathbf{t})\}\} \ .$$

We can now define modular completion, which relates instantiations of the rules in a module to propositional formulas. We will reuse some of the formulas later in a nonground setting, thus the definitions apply to nonground programs.

Definition 7.3 (Modular completion).
Let $r \in R(m_i)$ be a rule, let $P_i[S] \in \mathrm{VC}(\mathbf{P})$ be a value call, and let $\mathbf{y}$ be the free variables in the body of r. We define

$$\beta(P_i[S], r) = \exists \mathbf{y} \left[\bigwedge B_o^+(r)^S \wedge \bigwedge_{\beta(\mathbf{z}) \in B_m^+(r)} \mu(P_i[S], \beta(\mathbf{z})) \wedge \bigwedge \neg . B_o^-(r)^S \wedge \bigwedge_{\beta(\mathbf{z}) \in B_m^-(r)} \bar{\mu}(P_i[S], \beta(\mathbf{z})) \right]$$

and for an atom $a(\mathbf{x})$ appearing in some rule head, we let

$$\gamma(P_i[S], a(\mathbf{x})) = \forall \mathbf{x} \left[\bigvee_{r \in \mathrm{SR}(a(\mathbf{x}), R(m_i))} \beta(P_i[S], r) \supset a^S(\mathbf{x}) \right]$$

and

$$\sigma(P_i[S], a(\mathbf{x})) = \forall \mathbf{x} \left[a^S(\mathbf{x}) \supset \bigvee_{r \in \mathrm{SR}(a(\mathbf{x}), R(m_i))} \beta(P_i[S], r) \right] \ .$$

For any value call $P_i[S]$ of module m_i, let

$$\gamma(\mathbf{P}, P_i[S]) = \bigwedge_{r \in R(m_i) \wedge a(\mathbf{x}) \in H(r)} \gamma(P_i[S], a(\mathbf{x})) \wedge \bigwedge_{\chi^S(q_{i,j}(\mathbf{c}))=1} q^S_{i,j}(\mathbf{c})$$

and

$$\sigma(\mathbf{P}, P_i[S]) = \bigwedge_{r \in R(m_i) \wedge a(\mathbf{x}) \in H(r)} \sigma(P_i[S], a(\mathbf{x})) \ ,$$

and for an MLP $\mathbf{P}$ we define

$$\gamma(\mathbf{P}) = \bigwedge_{P_i[S] \in \mathrm{VC}(\mathbf{P})} \gamma(\mathbf{P}, P_i[S])$$

and

$$\sigma(\mathbf{P}) = \bigwedge_{P_i[S] \in \mathrm{VC}(\mathbf{P})} \sigma(\mathbf{P}, P_i[S]) \ .$$

The intuition behind formulas $\beta(P_i[S], r)$ is to state that the whole body of a rule r from $I_{\mathbf{P}}(P_i[S])$ is satisfied, which is going to be used in $\gamma(P_i[S], a(\mathbf{x}))$ to encode that for all $\mathbf{x}$ if one body of a rule with $a(\mathbf{x})$ in the head is satisfied, then $a^S(\mathbf{x})$ must be true. Formula $\sigma(P_i[S], a(\mathbf{x}))$ gives the other direction that for all $\mathbf{x}$ if $a(\mathbf{x})$ is true, then one of the bodies must be satisfied. Formulas $\gamma(\mathbf{P}, P_i[S])$ and $\sigma(\mathbf{P}, P_i[S])$ then extend $\gamma(P_i[S], a(\mathbf{x}))$ and $\sigma(P_i[S], a(\mathbf{x}))$ for all possible rules from $I_{\mathbf{P}}(P_i[S])$; note that $\gamma(P_i[S], a(\mathbf{x}))$ additionally adds all atoms from S in value call $P_i[S]$. The formulas $\gamma(\mathbf{P})$ and $\sigma(\mathbf{P})$ then encode the whole MLP $\mathbf{P}$ by iterating over all possible value calls from $\mathrm{VC}(\mathbf{P})$.

In the rest of this section, we assume that $\mathbf{P}$ is a ground normal MLP, thus the list of terms $\mathbf{t}$ in Definition 7.2 amount to a list of constant symbols $\mathbf{c}$ from $\mathcal{C}$.

We now give our two running examples that we use throughout this section.

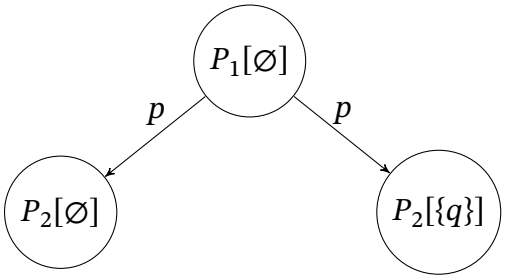


Figure 7.1: Callgraph for Example 7.1

Example 7.1 Let $m_1 = (P_1[], R_1)$ with $R_1 = \{p \leftarrow P_2[p].r\}$ and $m_2 = (P_2[q], R_2)$ with $R_2 = \{r \leftarrow q\}$. Then $\mathbf{P} = (m_1, m_2)$ is a normal MLP with the main module m_1. The program $\mathbf{P}$ has the single answer set $(M_1/\varnothing \coloneqq \varnothing, M_2/\varnothing \coloneqq \varnothing, M_2/\{q\} \coloneqq \{r, q\})$.

The modular completion of $\mathbf{P}$ gives us the following formulas, where $S_1 = \varnothing$, $S_2^0 = \varnothing$, and $S_2^1 = \{q\}$:

- $\gamma(\mathbf{P}, P_1[\varnothing]) = \left((\neg p^{S_1} \wedge r^{S_2^0}) \vee (p^{S_1} \wedge r^{S_2^1}) \supset p^{S_1}\right) \wedge \top$,
- $\gamma(\mathbf{P}, P_2[\varnothing]) = \left(q^{S_2^0} \supset r^{S_2^0}\right) \wedge \top$,
- $\gamma(\mathbf{P}, P_2[\{q\}]) = \left(q^{S_2^1} \supset r^{S_2^1}\right) \wedge q^{S_2^1}$;

and

- $\sigma(\mathbf{P}, P_1[\varnothing]) = p^{S_1} \supset (\neg p^{S_1} \wedge r^{S_2^0}) \vee (p^{S_1} \wedge r^{S_2^1})$,
- $\sigma(\mathbf{P}, P_2[\varnothing]) = r^{S_2^0} \supset q^{S_2^0}$,
- $\sigma(\mathbf{P}, P_2[\{q\}]) = r^{S_2^1} \supset q^{S_2^1}$.

The conjunction of the first three formulas yields $\gamma(\mathbf{P})$, and the last three give us $\sigma(\mathbf{P})$. Note that $\top$ in $\gamma(\mathbf{P}, P_1[\varnothing])$ and $\gamma(\mathbf{P}, P_2[\varnothing])$ stems from input $\varnothing$.

Example 7.2 For the MLP $\mathbf{P}$ in Example 3.2, we get the following modular completion formulas:

- $\gamma(\mathbf{P}) = \left(p_2^{\varnothing} \supset p_1^{\varnothing}\right) \wedge \left(p_1^{\varnothing} \supset p_2^{\varnothing}\right)$
- $\sigma(\mathbf{P}) = \left(p_1^{\varnothing} \supset p_2^{\varnothing}\right) \wedge \left(p_2^{\varnothing} \supset p_1^{\varnothing}\right)$

For readability, we dropped $\top$ encoding the inputs $\varnothing$ of $\gamma(\mathbf{P}, P_1[\varnothing])$ and $\gamma(\mathbf{P}, P_2[\varnothing])$.

The callgraphs for the programs defined in Examples 7.1 and 3.2 are shown in Figures 7.1 and 3.1, respectively. The former is acyclic, whereas that latter manifests cyclic module calls. In the following §7.2, we will show that a rule like $p \leftarrow P_2[p].r$ from Example 7.1 gives us an intriguing cyclic dependency, which does not materializes in the call graph alone.

As a first result, we can now show that for a normal MLP $\mathbf{P}$, formula $\gamma(\mathbf{P})$ captures the classical models of $\mathbf{P}$.

Lemma 7.1
The models of $\gamma(\mathbf{P})$ correspond one-to-one to the models of $\mathbf{P}$. That is,

1. if $M \models \gamma(\mathbf{P})$, then $\mathbf{M} \models \mathbf{P}$, where $M_i/S = \{p(\mathbf{c}) \in \mathrm{HB}_\mathbf{P} \mid p^S(\mathbf{c}) \in M \wedge p \in \mathcal{P}_i\}$, for all $P_i[S]$, and
2. if $\mathbf{M} \models \mathbf{P}$, then $M \models \gamma(\mathbf{P})$, where

$$M = \bigcup_{P_i[S] \in \mathrm{VC}(\mathbf{P})} (M_i/S)^S \ .$$

Proof For showing item 1, suppose that $M \models \gamma(\mathbf{P})$, and let $\mathbf{M}$ be as defined. We need to show that $\mathbf{M}, P_i[S] \models r$ for each $r \in I_\mathbf{P}(P_i[S]) = R(m_i) \cup S$ and $P_i[S] \in \mathrm{VC}(\mathbf{P})$. If r is a fact $q_j(\mathbf{c}) \leftarrow$ for a formal input parameter q_j of $P_i[\mathbf{q}]$, then $q_j(\mathbf{c}) \in S$ and, by $M \models \gamma(\mathbf{P}, P_i[S])$, we have $M \models q_j^S(\mathbf{c})$; hence, $q_j(\mathbf{c}) \in M_i/S$, and thus $\mathbf{M}, P_i[S] \models r$. Otherwise, $r \in R(m_i)$. As $M \models \gamma(\mathbf{P}, P_i[S])$, we have that M satisfies $\gamma(P_i[S], \alpha)$ for all rules $r' \in \mathrm{SR}(\alpha, R(m_i))$ such that $H(r') = \{\alpha\}$, thus also for the rule r such that $H(r) = \{a(\mathbf{c})\}$, hence $M \models \gamma(P_i[S], a(\mathbf{c}))$. By construction, for each ordinary atom $\beta(\mathbf{c})$ in r, we have $M \models \beta^S(\mathbf{c})$ implies $\mathbf{M}, P_i[S] \models \beta(\mathbf{c})$; furthermore, $M \models \mu(P_i[S], \beta(\mathbf{c}))$ for $P_k[\mathbf{p}].o(\mathbf{c})$ implies $M \models o^T(\mathbf{c})$, where $T \subseteq \mathrm{HB}_\mathbf{P}|_{\mathbf{q}_k}$ is the unique set T such that $M \models \bigwedge_j (p_j^S(\mathbf{c}) \equiv q_{k,j}^T(\mathbf{c}))$. That is, $M \models \mu(P_i[S], \beta(\mathbf{c}))$ implies $\mathbf{M}, P_i[S] \models \beta(\mathbf{c})$. This gives us that $M \models \beta(P_i[S], r)$ implies $\mathbf{M}, P_i[S] \models B(r)$. Moreover, since $M \models \gamma(P_i[S], a(\mathbf{c}))$ such that $H(r) = \{a(\mathbf{c})\}$, either $M \models a^S(\mathbf{c})$, or $M \not\models \bigvee \beta(P_i[S], r')$ for all $r' \in \mathrm{SR}(a(\mathbf{c}), R(m_i))$. Thus, $\mathbf{M}, P_i[S] \models H(r)$, or $\mathbf{M}, P_i[S] \not\models B(r)$. Hence, it follows that $\mathbf{M}, P_i[S] \models r$ for any r and $P_i[S]$ from $\mathbf{P}$, thus $\mathbf{M} \models \mathbf{P}$.

To prove item 2, let $\mathbf{M} \models \mathbf{P}$, and let $M = \bigcup_{P_i[S] \in \mathrm{VC}(\mathbf{P})} (M_i/S)^S$. To show that $M \models \gamma(\mathbf{P})$, we prove that $M \models \gamma(\mathbf{P}, P_i[S])$ for all $P_i[S]$. As $S \subseteq I_\mathbf{P}(P_i[S])$ and $\mathbf{M}, P_i[S] \models I_\mathbf{P}(P_i[S])$, all conjuncts $q_j^S(\mathbf{c})$ (representing the formal input) in $\gamma(\mathbf{P}, P_i[S])$ are satisfied by M; thus it remains to show $M \models \bigwedge \gamma(P_i[S], a(\mathbf{c}))$ for all $a(\mathbf{c})$ appearing in some rule head in $R(m_i)$. Let $r \in R(m_i)$ such that $H(r) = \{a(\mathbf{c})\}$. For each ordinary atom $\beta(\mathbf{c})$ in r, we have by construction of M that $M \models \beta^S(\mathbf{c})$ if $\mathbf{M}, P_i[S] \models \beta(\mathbf{c})$; furthermore, for each module atom $\beta(\mathbf{c}) = P_k[\mathbf{p}].o(\mathbf{c})$ in r, we have that $M \models \mu(P_i[S], \beta(\mathbf{c}))$ whenever $M \models o^T(\mathbf{c})$, which follows from $o(\mathbf{c}) \in M_k/T$, where $T \subseteq \mathrm{HB}_\mathbf{P}|_{\mathbf{q}_k}$ contains $q_{k,j}(\mathbf{c})$ if

$M \models p_j^S(\mathbf{c})$ whenever $p_j(\mathbf{c}) \in M_i/S$. Thus, $M \models \mu(P_i[S], \beta(\mathbf{c}))$ if $o(\mathbf{c}) \in M_k/(M_i/S)|_{\mathbf{P}}^{\mathbf{q}_k}$, which we obtain from $\mathbf{M}, P_i[S] \models o(\mathbf{c})$. Now we have shown that $\mathbf{M}, P_i[S] \models B(r)$ implies $M \models \beta(P_i[S], r)$. As $\mathbf{M}, P_i[S] \models r$, we have that $\mathbf{M}, P_i[S] \models H(r)$ or $\mathbf{M}, P_i[S] \not\models B(r)$. Therefore, $M \models \gamma(P_i[S], a(\mathbf{c}))$. We have shown that this holds for any $a(\mathbf{c})$ and $P_i[S]$, thus $M \models \gamma(\mathbf{P})$. □

Apt et al. (1988) define *supported models* for logic programs. For a normal MLP $\mathbf{P}$, we can show that answer sets are supported minimal models of $\mathbf{P}$, which has been shown for normal logic programs by Marek and Subrahmanian (1992). We first generalize the definition to MLPs.

Definition 7.4 (Supported models for MLPs).
A model $\mathbf{M}$ of a normal MLP $\mathbf{P}$ is called *supported* if for every atom $\alpha \in M_i/S$, where $P_i[S] \in \mathrm{VC}(\mathbf{P})$, there is some rule $r \in \mathrm{SR}(\alpha, I_{\mathbf{P}}(P_i[S]))$ such that $\mathbf{M}, P_i[S] \models B(r)$.

The following result shows that answer sets of $\mathbf{P}$ are supported models of $\mathbf{P}$. The reverse direction does not hold in general.

Proposition 7.2 (Supported models)
If $\mathbf{M}$ is an answer set of a normal MLP $\mathbf{P}$, then $\mathbf{M}$ is a supported minimal model of $\mathbf{P}$.

Proof Since $\mathbf{M}$ is an answer set, it is a minimal model of $\mathbf{P}$ by Proposition 3.5. Towards a contradiction, suppose $\mathbf{M}$ is not supported. There is an atom $\alpha \in M_i/S$ such that all rules $r \in \mathrm{SR}(\alpha, I_{\mathbf{P}}(P_i[S]))$ satisfy $\mathbf{M}, P_i[S] \not\models B(r)$. Hence, there is no rule $r \in \mathrm{SR}(\alpha, I_{\mathbf{P}}(P_i[S]))$ such that $r \in f\,\mathbf{P}(P_i[S])^{\mathbf{M}}$, contradicting $\mathbf{M}$ being a minimal model of $f\,\mathbf{P}(P_i[S])^{\mathbf{M}}$. Therefore, $\mathbf{M}$ is a supported model of $\mathbf{P}$. □

Example 7.3 In Example 3.2, the following models of $\mathbf{P}$ are supported models of $\mathbf{P}$:

- $\mathbf{M}_1 = (M_1/\emptyset := \emptyset, M_2/\emptyset := \emptyset)$ and
- $\mathbf{M}_2 = (M_1/\emptyset := \{p_1\}, M_2/\emptyset := \{p_2\})$.

While $\mathbf{M}_1$ is an answer set of $\mathbf{P}$, this does not hold for $\mathbf{M}_2$, since $\mathbf{M}_1 < \mathbf{M}_2$.

Fages (1994) demonstrated for normal logic programs that supported models have the intriguing property that whenever the rules in a program P are acyclic, i.e., no atom depends recursively on itself (also called tight logic programs by Erdem and Lifschitz, 2003), then P has a single supported model, which gives rise to an answer set of P.

Based on Lemma 7.1 the following can be shown for MLPs.

Lemma 7.3
The models of $\gamma(\mathbf{P}) \wedge \sigma(\mathbf{P})$ correspond one-to-one to the supported models of $\mathbf{P}$.

Proof ($\Rightarrow$) Let M be a model of $\gamma(\mathbf{P}) \wedge \sigma(\mathbf{P})$ and let $\mathbf{M}$ be the interpretation defined in Lemma 7.1, item 1. Since $M \models \gamma(\mathbf{P}) \wedge \sigma(\mathbf{P})$, we have $M \models \gamma(\mathbf{P})$. By Lemma 7.1, $M \models \gamma(\mathbf{P})$ implies $\mathbf{M} \models \mathbf{P}$. We show now that $\mathbf{M}$ is a supported model, i.e., we show that for every atom $\alpha \in M_i/S$, there exists an $r \in \mathrm{SR}(\alpha, I_{\mathbf{P}}(P_i[S]))$ such that $\mathbf{M}, P_i[S] \models B(r)$.

If $\alpha \in S$, i.e., α is of form $q_{i,j}(\mathbf{c})$ from the formal input parameters of m_i, then $M \models \gamma(\mathbf{P}, P_i[S])$ implies that $\mathbf{M}, P_i[S] \models S$, and therefore there exists a rule $r \in S$ with $H(r) = \{\alpha\}$, hence $r \in \mathrm{SR}(\alpha, S)$ and thus $r \in \mathrm{SR}(\alpha, I_{\mathbf{P}}(P_i[S]))$. As this r is a fact, we have $B(r) = \varnothing$ and thus $\mathbf{M}, P_i[S] \models B(r)$ holds vacuously.

If $\alpha \in M_i/S \setminus S$, we have $\alpha^S \in M$, and thus we can conclude from the formulas in $\sigma(\mathbf{P}, P_i[S])$ that the antecedent α^S requires that the consequent $\bigvee \beta(P_i[S], r)$ is true in M for $r \in \mathrm{SR}(\alpha, I_{\mathbf{P}}(P_i[S]))$. It follows that there must be at least one of the formulas $\beta(P_i[S], r)$ true in M. Let r be the rule from $\mathrm{SR}(\alpha, R(m_i))$ such that $M \models \beta(P_i[S], r)$. Since $M \models B_o^+(r)^S$ and $M \not\models B_o^-(r)^S$, we get that $\mathbf{M}, P_i[S] \models B_o^+(r)$ and $\mathbf{M}, P_i[S] \not\models B_o^-(r)$, thus ordinary atoms from $B(r)$ are satisfied by $\mathbf{M}$ at $P_i[S]$. Additionally, we have $M \models \mu(P_i[S], \beta(\mathbf{c}))$ and $M \models \bar{\mu}(P_i[S], \beta(\mathbf{c}))$ for each $\beta(\mathbf{c}) \in B_m^+(r)$ and $\beta(\mathbf{c}) \in B_m^-(r)$, respectively. Thus, at least one of the disjuncts $\big(\epsilon(P_i[S], P_k[T]) \wedge o^T(\mathbf{c})\big)$ or one of the disjuncts $\big(\epsilon(P_i[S], P_k[T]) \wedge \neg o^T(\mathbf{c})\big)$ is true in M for a $\beta(\mathbf{c}) \in B_m^+(r) \cup B_m^-(r)$. From that we can conclude that $M \models o^T(\mathbf{c})$ implies $\mathbf{M}, P_k[T] \models \beta(\mathbf{c}) = P_k[\mathbf{p}].o(\mathbf{c})$, where $T \subseteq \mathrm{HB}_{\mathbf{P}}|_{\mathbf{q}_k}$ is the unique set T such that $M \models \bigwedge_j (p_j^S(\mathbf{c}) \equiv q_{k,j}^T(\mathbf{c}))$. That is, $M \models \mu(P_i[S], \beta(\mathbf{c}))$ implies $\mathbf{M}, P_i[S] \models \beta(\mathbf{c})$ for $\beta(\mathbf{c}) \in B_m^+(r)$, and $M \models \bar{\mu}(P_i[S], \beta(\mathbf{c}))$ implies $\mathbf{M}, P_i[S] \not\models \beta(\mathbf{c})$ for $\beta(\mathbf{c}) \in B_m^-(r)$. Hence, it follows that $\mathbf{M}, P_i[S] \models B(r)$, and therefore $\mathbf{M}$ is a supported model of $\mathbf{P}$.

($\Leftarrow$) Let $\mathbf{M}$ be a supported model of $\mathbf{P}$. Let M be an interpretation of $\gamma(\mathbf{P}) \wedge \sigma(\mathbf{P})$ as defined in Lemma 7.1, item 2. By Lemma 7.1 we can conclude that $M \models \gamma(\mathbf{P})$. We show now that $M \models \sigma(\mathbf{P})$. From the construction of M, we have $\alpha \in M_i/S$ iff $\alpha^S \in M$. Since $\mathbf{M}$ is supported there is a rule $r \in \mathrm{SR}(\alpha, I_{\mathbf{P}}(P_i[S]))$ such that $\mathbf{M}, P_i[S] \models B(r)$. This satisfies the formulas from $\sigma(\mathbf{P}, P_i[S])$: since $\alpha^S \in M$, M must satisfy the consequent $\bigvee \beta(P_i[S], r)$ for $r \in \mathrm{SR}(\alpha, I_{\mathbf{P}}(P_i[S]))$, i.e., at least one of the disjuncts $\beta(P_i[S], r)$ must hold in M, which follows from $\mathbf{M}, P_i[S] \models B(r)$. Thus, M is a model for $\gamma(\mathbf{P}) \wedge \sigma(\mathbf{P})$. □

Example 7.4 (cont'd) We continue with Example 7.1. Formula $\gamma(\mathbf{P})$ admits the following models:

- $M_1 = \{r^{S_2^1}, q^{S_2^1}\}$,
- $M_2 = \{p^{S_1}, r^{S_2^0}, r^{S_2^1}, q^{S_2^1}\}$,
- $M_3 = \{p^{S_1}, r^{S_2^1}, q^{S_2^1}\}$, and
- $M_4 = \{p^{S_1}, r^{S_2^0}, r^{S_2^1}, q^{S_2^0}, q^{S_2^1}\}$.

They correspond to the classical models

- $\mathbf{M}_1 = (M_1/\emptyset \coloneqq \emptyset, M_2/\emptyset \coloneqq \emptyset, M_2/\{q\} \coloneqq \{r, q\})$,
- $\mathbf{M}_2 = (M_1/\emptyset \coloneqq \{p\}, M_2/\emptyset \coloneqq \{r\}, M_2/\{q\} \coloneqq \{r, q\})$,
- $\mathbf{M}_3 = (M_1/\emptyset \coloneqq \{p\}, M_2/\emptyset \coloneqq \emptyset, M_2/\{q\} \coloneqq \{r, q\})$, and
- $\mathbf{M}_4 = (M_1/\emptyset \coloneqq \{p\}, M_2/\emptyset \coloneqq \{r, q\}, M_2/\{q\} \coloneqq \{r, q\})$

for $\mathbf{P}$. The formula $\gamma(\mathbf{P}) \wedge \sigma(\mathbf{P})$ permits only M_1, M_3, and M_4 as models, which give us the supported models $\mathbf{M}_1$, $\mathbf{M}_3$, and $\mathbf{M}_4$ of $\mathbf{P}$.

Example 7.5 (cont'd) In Example 7.2, the models of $\gamma(\mathbf{P})$ are

- $M_1 = \emptyset$ and
- $M_2 = \{p_1^S, p_2^S\}$,

which are also the models of $\gamma(\mathbf{P}) \wedge \sigma(\mathbf{P})$. Both of them correspond to the classical as well as supported models of $\mathbf{P}$, namely

- $\mathbf{M}_1 = (M_1/\emptyset \coloneqq \emptyset, M_2/\emptyset \coloneqq \emptyset)$ and
- $\mathbf{M}_2 = (M_1/\emptyset \coloneqq \{p_1\}, M_2/\emptyset \coloneqq \{p_2\})$.

7.2 Loop Formulas for MLPs

In this section, we develop *modular loop formulas* that instantiate each program module with possible input to create the classical theory of the program, and then add loop formulas similar to the approach of Lee and Lifschitz (2003) and Lin and Zhao (2004). However, we have to respect loops not only inside a module, but also across modules due to module atoms. The latter will be captured by a *modular dependency graph*, which records positive dependencies that relate module instantiations with the atoms in a module. The instantiation of the modules makes it necessary to create fresh propositional atoms very similarly to grounding of logic programs, and our complexity results in Chapter 5 suggest that there is no way to circumvent this; see Tables 5.1–5.3 for a synopsis of the complexity results. In the rest of this section, we assume that MLP $\mathbf{P}$ is ground and normal.

We define now the modular dependency graph to keep track of dependencies between modules and rules. It is a ground dependency graph with two additional types of edges.

Figure 7.2: Modular dependency graphs

Definition 7.5 (Modular dependency graph).
Let $\mathbf{P} = (m_1, \ldots, m_n)$ be a normal MLP. The *modular dependency graph* of $\mathbf{P}$ is the digraph $\mathrm{MG}_\mathbf{P} = (V, E)$ with vertex set $V = \mathrm{HB}_\mathbf{P}$ and edge set E containing the following edges:

- $p(\mathbf{c}_1) \to q(\mathbf{c}_2)$, for each $r \in R(m_i)$ with $H(r) = \{p(\mathbf{c}_1)\}$ and $q(\mathbf{c}_2) \in B^+(r)$.
- $a \to b$, if one of (1)–(2) holds, where α is of the form $P_j[\mathbf{p}].o(\mathbf{c})$ in $R(m_i)$ and P_j has the associated input list $\mathbf{q}_j$:
  1. $a = \alpha$ and $b = o(\mathbf{c}) \in \mathrm{HB}_\mathbf{P}$;
  2. $a = q_\ell(\mathbf{c}) \in \mathrm{HB}_\mathbf{P}|_{\mathbf{q}_j}$ and $b = p_\ell(\mathbf{c}) \in \mathrm{HB}_\mathbf{P}|_\mathbf{p}$ for $1 \leq \ell \leq |\mathbf{q}_j|$.

Intuitively, the modular dependency graph does not cater for module instantiations, i.e., all module atoms are purely syntactic and the dependencies between atoms are coarse. This also means that cycles that show up in the modular dependency graph must be instantiated in the formulas.

Example 7.6 (cont'd) The modular dependency graphs of the MLPs defined in Examples 7.1 and 3.2 are shown in Figures 7.2a and 7.2b, respectively. In both figures, the two upper nodes are from m_2, while the nodes below stem from m_1. Note that the dashed edges stem from condition (1) in Definition 7.5, while dotted edges are from condition (2). Straight edges are standard head-body dependencies.

Next, we define modular loops, which are based on modular dependency graphs.

Definition 7.6 (Modular loops).
A set of atoms $\mathcal{L} \subseteq V(\mathrm{MG}_\mathbf{P})$ is called a *modular loop for* $\mathbf{P}$ iff the subgraph of $\mathrm{MG}_\mathbf{P}$ induced by $\mathcal{L}$ is strongly connected.

Note that $\mathcal{L}$ may contain module atoms, and single-atom loops are allowed.

Modular loop formulas have then the same shape as standard loop formulas (Lee and Lifschitz, 2003; Lin and Zhao, 2004), with the important distinction that external support formulas may take the input S from the value call $P_i[S]$. For that, we first define external support rules.

Definition 7.7 (External support rules).
Let R be a set of ground normal rules. The *external support rules* of R with respect to a set of atoms $\mathcal{L} \subseteq \mathrm{HB}_{\mathbf{P}}$ is

$$\mathrm{ER}(\mathcal{L}, R) = \{r \in R \mid H(r) \cap \mathcal{L} \neq \emptyset \wedge B^+(r) \cap \mathcal{L} = \emptyset\} \quad .$$

Note that $\mathcal{L}$ may contain module atoms.

Modular loops may go through the atoms of multiple modules, but do not take care of cycles over module instantiations that stem from the input. Given a modular loop, the instantiated loop may be exponentially longer in the propositional case, whereas it could have double-exponential length in the nonground case. To keep record of these loops, we next define cyclic instantiation signatures that are used to instantiate modular loops. In the following, let $\mathbf{P} = (m_1, \ldots, m_n)$ be a normal MLP such that the set of predicate symbols $\mathcal{P} = \mathcal{P}_1 \cup \cdots \cup \mathcal{P}_n$ such that $\mathcal{P}_i$ consists only of predicate symbols from m_i.

Definition 7.8 (Cyclic instantiation signature).
Let $\mathcal{L}$ be a modular loop for the normal MLP $\mathbf{P} = (P_1[\mathbf{q}_1], \ldots, P_n[\mathbf{q}_n])$. A *cyclic instantiation signature for* $\mathcal{L}$ is a tuple $\mathcal{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$, where each $\mathcal{S}_i \subseteq 2^{\mathrm{HB}_{\mathbf{P}}|_{\mathbf{q}_i}}$ is a family of sets over $\mathrm{HB}_{\mathbf{P}}|_{\mathbf{q}_i}$ such that

1. $\mathcal{S}_i \neq \emptyset$ and all $S \in \mathcal{S}_i$ satisfy $S \cap \mathcal{L} = \emptyset$, whenever $\mathcal{L}$ contains ground atoms with predicates from $\mathcal{P}_i$, and
2. $\mathcal{S}_i = \emptyset$ otherwise.

Intuitively, we use a modular loop as template to create loops that go over module instantiations, as illustrated in the next examples.

Example 7.7 (cont'd) The MLP $\mathbf{P}$ in Example 7.1 has the loop $\mathcal{L} = \{p, P_2[p].r, r, q\}$, for which we get one cyclic instantiation signature $\mathcal{S}_1 = (\{\emptyset\}, \{\emptyset\})$. Note that the tuples $(\{\emptyset\}, \{\{q\}\})$ and $(\{\emptyset\}, \{\emptyset, \{q\}\})$ are not cyclic instantiation signatures as they share atoms with $\mathcal{L}$, thus always get support from input S. Intuitively, this captures those module instantiations that cycle over module input, but have no support from the formal input, viz., $P_1[\emptyset] \leftrightarrow P_2[\emptyset]$.

Example 7.8 (cont'd) In Example 3.2, we have a loop $\mathcal{L} = \{p_1, P_2.p_2, p_2, P_1.p_1\}$. We get one cyclic instantiation signatures: $\mathcal{S}_1 = (\{\emptyset\}, \{\emptyset\})$. Here, $\mathcal{S}_1$ builds a cycle over module instantiations from the mutual calls in m_1 and m_2.

Based on modular loops, cyclic instantiation signatures, and external support rule, we are now in the position to define modular loop formulas for normal MLPs.

Definition 7.9 (Modular loop formulas).
Let $\mathcal{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$ be an instantiation signature for the modular loop $\mathcal{L}$ in MLP $\mathbf{P}$. The *loop formula* $\lambda(\mathcal{S}, \mathcal{L}, \mathbf{P})$ *for* $\mathcal{L}$ *with respect to* $\mathcal{S}$ *in* $\mathbf{P}$ is

$$\bigvee_{i=1}^{n} \bigvee_{T \in \mathcal{S}_i} \Big(\bigvee (\mathcal{L}|_{\mathcal{P}_i})^T \Big) \quad \supset \quad \bigvee_{i=1}^{n} \bigvee_{S \in \mathcal{S}_i} \bigvee_{r \in \mathrm{ER}(\mathcal{L}, I_{\mathbf{P}}(P_i[S]))} \Big(H(r)^S \supset \beta(P_i[S], r) \Big) \ . \tag{7.1}$$

Given $\mathbf{P}$, the *loop formula for a modular loop* $\mathcal{L}$ *in* $\mathbf{P}$ is the conjunction

$$\lambda(\mathcal{L}, \mathbf{P}) = \bigwedge_{\mathcal{S}} \lambda(\mathcal{S}, \mathcal{L}, \mathbf{P})$$

for all cyclic instantiation signatures $\mathcal{S}$ of $\mathcal{L}$, and the *loop formula for* $\mathbf{P}$ is the conjunction

$$\lambda(\mathbf{P}) = \bigwedge_{\mathcal{L}} \lambda(\mathcal{L}, \mathbf{P})$$

for all modular loops $\mathcal{L}$ in $\mathbf{P}$. The *modular loop formula for the MLP* $\mathbf{P}$ is then

$$\Lambda(\mathbf{P}) = \gamma(\mathbf{P}) \wedge \sigma(\mathbf{P}) \wedge \lambda(\mathbf{P}) \ .$$

Intuitively, the formal input in a value call $P_i[S]$ always adds external support for the input atoms in S as we add S to the instantiation $I_{\mathbf{P}}(P_i[S])$. Since we obtain all supported models with $\gamma(\mathbf{P}) \wedge \sigma(\mathbf{P})$, thus also have S there, we can restrict to those instantiation signatures $\mathcal{S}$ for a modular loop $\mathcal{L}$ that have no support from formal input.

Example 7.9 (cont'd) Continuing with Example 7.1, we get the following modular loop formulas based on the loop $\mathcal{L}$ and instantiation signature $\mathcal{S}_1$ for $\mathcal{L}$ shown in Example 7.7 (here, $S_1 = \varnothing$, $S_2^0 = \varnothing$): $\lambda(\mathcal{S}_1, \mathcal{L}, \mathbf{P}) = (p^{S_1} \vee r^{S_2^0} \vee q^{S_2^0}) \supset \bot$. This formula and $\gamma(\mathbf{P}) \wedge \sigma(\mathbf{P})$ yield $\Lambda(\mathbf{P})$, whose model is $M_1 = \{r^{S_2^1}, q^{S_2^1}\}$, which coincides with the answer set $\mathbf{M}_1 = (\varnothing, \varnothing, \{r, q\})$ of $\mathbf{P}$.

Example 7.10 (cont'd) Based on Examples 7.2 and 7.8 we get the following modular loop formulas using the loop $\mathcal{L}$ and instantiation signature $\mathcal{S}_1$ ($S = \varnothing$): $\lambda(\mathcal{S}_1, \mathcal{L}, \mathbf{P}) = p_1^S \vee p_2^S \supset \bot \vee \bot$. The conjunction of $\gamma(\mathbf{P}) \wedge \sigma(\mathbf{P})$ and $\lambda(\mathcal{S}_1, \mathcal{L}, \mathbf{P})$ yields formula $\Lambda(\mathbf{P})$, and its model is thus $M_1 = \varnothing$, which coincides with the single answer set $\mathbf{M}_1 = (\varnothing, \varnothing)$ of $\mathbf{P}$.

We can now state our main result and show that $\Lambda(\mathbf{P})$ captures the answer sets of normal MLPs $\mathbf{P}$.

Theorem 7.4 (MLP loop formulas)
Given a normal MLP $\mathbf{P}$ the answer sets of $\mathbf{P}$ and the models of $\Lambda(\mathbf{P})$ correspond, such that

1. if $M \models \Lambda(P)$, then there is some answer set $\mathbf{M}$ of $\mathbf{P}$ such that $M_i/S = \{p(\mathbf{c}) \in \mathrm{HB}_\mathbf{P} \mid p^S(\mathbf{c}) \in M \wedge p \in \mathcal{P}_i\}$ for all $P_i[S] \in \mathrm{VC}(\mathbf{P})$, and
2. if $\mathbf{M}$ is an answer set of $\mathbf{P}$, then $M \models \Lambda(\mathbf{P})$, where

$$M = \bigcup_{P_i[S] \in \mathrm{VC}(\mathbf{P})} (M_i/S)^S \ .$$

Proof Since a modular loop $\mathcal{L}$ might span over multiple modules, we need to look at the rules from multiple module instantiations. Let $m(\mathcal{L}) = \{m_{j_1}, \dots, m_{j_k}\}$ be the set of all modules whose module atoms appear in $\mathcal{L}$.

We begin with proving item 1. Let $M \models \Lambda(\mathbf{P})$. Since $M \models \gamma(\mathbf{P})$ and $M \models \sigma(\mathbf{P})$, Lemma 7.3 implies that $\mathbf{M}$ is a supported model of $\mathbf{P}$. We show now that $\mathbf{M}$ is the model of $\mathbf{P}$ whose atoms in each M_i/S can be derived from $f\,\mathbf{P}(P_i[S])^\mathbf{M}$ for all $P_i[S] \in \mathrm{VC}(\mathbf{P})$. Thus, $\mathbf{M}$ must be the minimal model for $f\,\mathbf{P}^\mathbf{M}$. Since $M \models \lambda(\mathbf{P})$, for all cyclic instantiation signature $\mathcal{S}$ and all modular loops $\mathcal{L}$ the formulas $\lambda(\mathcal{S}, \mathcal{L}, \mathcal{P})$ are satisfied by M. Observe that for a modular loop $\mathcal{L}$, there always exists a subset-maximal loop $\mathcal{L}'$ such that $\mathcal{L} \subseteq \mathcal{L}'$ and there is no modular loop $\mathcal{L}''$ such that $\mathcal{L}' \subset \mathcal{L}''$.

Let $\mathcal{R}_0$ be the rule base inheriting rules from $f\,\mathbf{P}^\mathbf{M}$ such that both the body and the head is true in $\mathbf{M}$, i.e.,

$$\mathcal{R}_0 = \big(F_{P_i[S]} \mid P_i[S] \in \mathrm{VC}(\mathbf{P})\big)$$

where

$$F_{P_i[S]} = \big\{r \in f\,\mathbf{P}(P_i[S])^\mathbf{M} \mid \mathbf{M}, P_i[S] \models H(r) \wedge \mathbf{M}, P_i[S] \models B(r)\big\} \ .$$

For $k \geq 0$ we inductively define $\mathcal{R}_{k+1}$ as follows:

$$\mathcal{R}_{k+1} = \begin{cases} \mathcal{R}_k & \text{if there is no modular loop in } \mathcal{R}_k \\ \mathrm{LR}(\mathcal{S}, \mathcal{L}_k, \mathcal{R}_k) & \text{for the subset-maximal loop } \mathcal{L}_k \text{ in } \mathcal{R}_k \text{ with respect to } \mathcal{S} \end{cases}$$

where for the rule base $\mathcal{R}_k = \big(R_{P_i[S]}\big)$, modular loop $\mathcal{L}_k$, and cyclic instantiation signature $\mathcal{S}$ we let

$$\mathrm{LR}(\mathcal{S}, \mathcal{L}_k, \mathcal{R}_k) = \big(R_{P_i[S]}(\mathcal{L}_k) \setminus E_{P_i[S]}(\mathcal{L}_k) \mid m_i \in m(\mathcal{L}_k) \wedge S \in \mathcal{S}_i\big)$$

such that

$$R_{P_i[S]}(\mathcal{L}_k) = \left\{r \in R_{P_i[S]} \mid \begin{array}{c} H(r) \cap \mathcal{L}_k \neq \emptyset \wedge B^+(r) \cap \mathcal{L}_k \neq \emptyset \wedge \\ \mathbf{M}, P_i[S] \models H(r) \wedge \mathbf{M}, P_i[S] \models B(r) \end{array}\right\}$$

and

$$E_{P_i[S]}(\mathcal{L}_k) = \left\{ r \in R_{P_i[S]}(\mathcal{L}_k) \mid \begin{array}{c} \exists r' \in \mathrm{ER}(\mathcal{L}_k, I_{\mathbf{P}}(P_i[S])) \text{ such that} \\ H(r) = H(r') \wedge \mathbf{M}, P_i[S] \vDash B(r') \end{array} \right\}.$$

Note that there always exists an $E_{P_i[S]}(\mathcal{L}_k)$ which is nonempty: $\mathcal{L}_k$ must be a modular loop in $\mathbf{P}$ as well, and since both $M \vDash \lambda(\mathcal{S}, \mathcal{L}_k, I_{\mathbf{P}}(P_i[S]))$ and $M \vDash p^S$ for some $p \in \mathcal{L}_k$, we must have a rule $r' \in \mathrm{ER}(\mathcal{L}_k, I_{\mathbf{P}}(P_i[S]))$ whose body $B(r')$ is satisfied by $\mathbf{M}$ at $P_i[S]$.

Intuitively, starting from a maximal loop $\mathcal{L}_k$ the rule base $\mathcal{R}_{k+1}$ removes those rules from $\mathcal{R}_k$ that "break" the cycle $\mathcal{L}_k$ in $\mathbf{P}$—i.e., from the rules $R_{P_i[S]}(\mathcal{L}_k)$—using externally supported rules from $E_{P_i[S]}(\mathcal{L}_k)$. The result is $\mathcal{R}_{k+1}$, which might contain a different maximal modular loop $\mathcal{L}_{k+1}$ that is set for removal in $\mathcal{R}_{k+2}$. Eventually, as $M \vDash \lambda(\mathbf{P})$, no loop will be left, as argued next.

There exists a finite ℓ such that for all $m > \ell$, $\mathcal{R}_m = \mathcal{R}_\ell$ and $\mathcal{R}_\ell$ does not have any modular loops since we started with $\mathcal{R}_0$ having only a finite number of modular loops and every $\mathcal{R}_k$, $k \geq 0$, is a sub-rule base of $\mathcal{R}_0$.

Next, we show that for each atom $a \in M_i/S$, there is a rule r from $\mathcal{R}_\ell$ such that $H(r) = \{a\}$. We proceed by induction on $k \geq 0$. For the base case $k = 0$, from Lemma 7.3 we deduce that there is a rule $r \in I_{\mathbf{P}}(P_i[S])$ such that $H(r) = \{a\}$ since $\mathbf{M}, P_i[S] \vDash B(r)$, hence $r \in f\,\mathbf{P}(P_i[S])^{\mathbf{M}}$ and by construction $r \in F_{P_i[S]}$ from $\mathcal{R}_0$. For the inductive step, assume that our claim holds for $\mathcal{R}_k$ such that $k \geq 0$. We show now that the statement holds for $\mathcal{R}_{k+1}$ as well. We obtain the following cases:

- If $\mathcal{R}_k$ has no modular loops, then $\mathcal{R}_{k+1} = \mathcal{R}_k$ and our claim follows immediately.
- If $\mathcal{R}_k$ has a modular loop, then $\mathcal{R}_{k+1}$ is the result of removing $E_{P_i[S]}(\mathcal{L}_k)$ from $\mathcal{R}_k$ for the maximal modular loop $\mathcal{L}_k$ in $\mathcal{R}_k$.

  Therefore, if $H(r)$ does not appear in the heads of the rules $E_{P_i[S]}(\mathcal{L}_k)$, $r \in R_{P_i[S]}(\mathcal{L}_k) \setminus E_{P_i[S]}(\mathcal{L}_k)$ and hence r will be in $\mathcal{R}_{k+1}$ as well.

  Otherwise, if $H(r)$ appears in the heads of some rules in $E_{P_i[S]}(\mathcal{L}_k)$ that have been removed from $\mathcal{R}_k$ in the construction of $\mathcal{R}_{k+1}$, we show now that r exists in $\mathcal{R}_{k+1}$ such that $r \in \mathrm{ER}(\mathcal{L}_k, I_{\mathbf{P}}(P_i[S]))$. We prove this by induction on j such that $0 \leq j < k$. For the base case $j = 0$, we have that $\mathcal{L}_k$ is a loop in $\mathcal{R}_k$, thus it is also a loop in $\mathcal{R}_0$ and hence $r \in F_{P_i[S]}$ from $\mathcal{R}_0$. To carry out the inductive step, assume that our claim of r appearing in $\mathcal{R}_j$ holds for $0 \leq j < k$. We show now that our r appears in $\mathcal{R}_{j+1}$ as well. Towards a contradiction, assume that r has been removed in $\mathcal{R}_{j+1}$. Then, there exists a maximal modular loop $\mathcal{L}_j$ such that $r \in R_{P_i[S]}(\mathcal{L}_j)$ and $r \in E_{P_i[S]}(\mathcal{L}_j)$. Now since $H(r)$ is in both $\mathcal{L}_j$ and in $\mathcal{L}_k$, their union $\mathcal{L}_j \cup \mathcal{L}_k$ is a loop in $\mathcal{R}_j$ since $j < k$. From $\mathcal{L}_j$ being maximal, we derive $\mathcal{L}_k \subseteq \mathcal{L}_j$. As r has been removed in $\mathcal{R}_{j+1}$, $r \notin R_{P_i[S]}(\mathcal{L}_j) \setminus E_{P_i[S]}(\mathcal{L}_j)$ and from the construction of $\mathcal{R}_{j+1}$ there is no rule $r' \in R_{P_i[S]}(\mathcal{L}_j) \setminus E_{P_i[S]}(\mathcal{L}_j)$ such that

$H(r') = \{a\}$. Hence, there is no rule $r \in R_{P_i[S]}(\mathcal{L}_k)$ such that $H(r) = \{a\}$, which contradicts that $H(r)$ appears in the heads of some rules in $E_{P_i[S]}(\mathcal{L}_k)$ that have been removed from $\mathcal{R}_k$ in the construction of $\mathcal{R}_{k+1}$. Hence, r is present in $\mathcal{R}_{j+1}$, and thus r exists in $\mathcal{R}_{k+1}$.

We have shown now that for all $k \geq 0$, if $a \in M_i/S$ then there exists a rule r from $\mathcal{R}_k$ such that $H(r) = \{a\}$. Hence, there is a rule r from $\mathcal{R}_\ell$ such that $H(r) = \{a\}$.

Next, we prove that for all $a \in M_i/S$, a is derived from a rule r from $\mathcal{R}_\ell$ such that $H(r) = \{a\}$ and $\mathbf{M}, P_i[S] \models B(r)$ such that $B(r)$ depends on a finite sequence of rule applications starting from an atom a_0 such that there exists a fact $a_0 \leftarrow$ in $\mathcal{R}_\ell$. This is without loss of generality: whenever several facts $a_0 \leftarrow$ to $a_k \leftarrow$ for $k > 0$ are needed to establish $a \in M_i/S$, we can replace, for $1 \leq j \leq k$, the fact $a_i \leftarrow$ by rule $a_i \leftarrow a_0$ as intermediary in $\mathcal{R}_\ell$. From above, we know that there exists a rule r such that $H(r) = \{a\}$. From the construction of $\mathcal{R}_\ell$, all rules satisfy $\mathbf{M}, P_i[S] \models B(r)$ and $\mathbf{M}, P_i[S] \models H(r)$. Moreover, $\mathcal{R}_\ell$ has no modular loop. Thus, there must exist a predecessor a' for a in $\mathrm{MG}_\mathbf{P}$ using a rule r from $\mathcal{R}_\ell$ and since $\mathbf{M}, P_i[S] \models B(r)$, a' must be true in $\mathbf{M}$. Continuing with a', and since $\mathcal{R}_\ell$ has no modular loop, we arrive using a finite number of rule applications at a rule $a_0 \leftarrow$ in $\mathcal{R}_\ell$ such that a_0 is true in $\mathbf{M}$. Thus, $\mathcal{R}_\ell$ gives us $\mathbf{M}$ as derived consequences. Since $\mathcal{R}_\ell$ is a sub-rule base of $f\,\mathbf{P}^\mathbf{M}$, all atoms of $\mathbf{M}$ can be derived from $f\,\mathbf{P}(P_i[S])^\mathbf{M}$ for all $P_i[S] \in \mathrm{VC}(\mathbf{P})$. Thus, $\mathbf{M}$ must be a minimal model for $f\,\mathbf{P}^\mathbf{M}$.

Next, we show item 2. Let $\mathbf{M}$ be an answer set of $\mathbf{P}$. From $\mathbf{M} \models \mathbf{P}$ we can conclude by Lemma 7.1 that $M \models \gamma(\mathbf{P})$. Proposition 7.2 implies that $\mathbf{M}$ is a supported model for $\mathbf{P}$, thus Lemma 7.3 gives us $M \models \sigma(\mathbf{P})$. We show now that $M \models \lambda(\mathbf{P})$. Assume to the contrary that $M \not\models \lambda(\mathbf{P})$. There is a modular loop $\mathcal{L}$ for $\mathbf{P}$ and a cyclic instantiation signature $\mathcal{S}$ for $\mathcal{L}$ such that $M \not\models \lambda(\mathcal{S}, \mathcal{L}, \mathcal{P})$. This means that the antecedent of formula (7.1) is true in M, while the consequent is false. Thus,

$$M \not\models \bigvee_{i=1}^{n} \bigvee_{S \in \mathcal{S}_i} \bigvee_{r \in \mathrm{ER}(\mathcal{L}, I_\mathbf{P}(P_i[S]))} \left(H(r)^S \supset \beta(P_i[S], r) \right) \quad , \tag{7.2}$$

hence at least one ordinary atom $a^T \in (\mathcal{L}|_{\mathcal{P}_i})^T$ for $T \in \mathcal{S}_i$ must be true in M, which means that $a \in M_i/T$. Clearly, one module $m_{j_l} \in m(\mathcal{L})$ must be m_i. Let

$$E_\mathcal{L} = \left(\mathrm{ER}(\mathcal{L}, I_\mathbf{P}(P_j[U])) \mid m_j \in m(\mathcal{L}) \wedge U \in \mathcal{S}_j \right)$$

be the rule base

$$\left(\begin{array}{ll} R_{P_{j_1}[U]} = \left\{\begin{array}{ll} p_{j_1,1} & \leftarrow B_{j_1,1} \\ & \vdots \\ p_{j_1,n_1} & \leftarrow B_{j_1,n_1} \end{array}\right\} \\ \vdots \\ R_{P_{j_k}[U]} = \left\{\begin{array}{ll} p_{j_k,1} & \leftarrow B_{j_k,1} \\ & \vdots \\ p_{j_k,n_k} & \leftarrow B_{j_k,n_k} \end{array}\right\} \end{array}\right) ,$$

where U ranges over all possible $U \in \mathcal{S}_{j_l}$ and each $r \in R_{P_j[U]}$ is a rule from a module $m_j \in m(\mathcal{L})$ and $B_{j,l}$ are lists of literals. The atoms $p_{j,l}$, where $j_1 \leq j \leq j_k$ and $1 \leq l \leq n_j$, account for the rules in $I_{\mathbf{P}}(P_{j_l}[U])$ that share their heads with an ordinary atom from the loop $\mathcal{L}$, but none of $B_{j,l}$ is contained in $\mathcal{L}$. Let $E^{\mathbf{M}}_{\mathcal{L}}$ be the FLP-reduct of $E_{\mathcal{L}}$ with respect to $\mathbf{M}$, i.e.,

$$E^{\mathbf{M}}_{\mathcal{L}} = \left(f\,\mathbf{P}(P_j[U])^{\mathbf{M}} \cap \mathrm{ER}(\mathcal{L}, I_{\mathbf{P}}(P_j[U])) \mid m_j \in m(\mathcal{L}) \wedge P_j[U] \in \mathcal{S}_j\right) \ ,$$

since $f\,\mathbf{P}(P_j[U])^{\mathbf{M}} \subseteq I_{\mathbf{P}}(P_j[U])$ and $\mathrm{ER}(\mathcal{L}, I_{\mathbf{P}}(P_j[U])) \subseteq I_{\mathbf{P}}(P_j[U])$. Then, (7.2) implies $M \not\models \bigwedge_{r\in E} \beta(P_j[U], r)$ for each set of rules E from rule base $E^{\mathbf{M}}_{\mathcal{L}}$.

From $\mathbf{M}, P_i[T] \models a$, we must have the following sequence of rules appearing in the rule base $\left(f\,\mathbf{P}(P_j[U])^{\mathbf{M}} \mid m_j \in m(\mathcal{L}) \wedge P_j[U] \in \mathrm{VC}(\mathbf{P})\right)$:

$$\begin{array}{rrl} r_0\colon & q_0 & \leftarrow \\ r_1\colon & q_1 & \leftarrow B_1 \\ & & \vdots \\ r_{\ell-1}\colon & q_{\ell-1} & \leftarrow B_{\ell-1} \\ r_\ell\colon & q_\ell & \leftarrow B_\ell \end{array}$$

where $q_\ell = a$ at $P_i[T]$, and for each $j = 1, \ldots, \ell$, $B^+_o(r_j) \subseteq \{q_0, \ldots, q_{j-1}\}$ and $B^+_m(r_j) \subseteq \{P_{j_1}[\mathbf{p}].o, \ldots, P_j[\mathbf{p}].o\}$. There must be an x for $x = 0, \ldots, \ell$ such that $\{q_0, \ldots, q_{x-1}\} \cap \mathcal{L} = \emptyset$ and $q_x \in \mathcal{L}$. From $B^+_o(r_x) \subseteq \{q_0, \ldots, q_{x-1}\}$ follows that $B^+_o(r_x) \cap \mathcal{L} = \emptyset$, thus r_x must appear in $E^{\mathbf{M}}_{\mathcal{L}}$ for a value call $P_j[U]$. Thus, $r_x \in f\,\mathbf{P}(P_j[U])^{\mathbf{M}} \cap \mathrm{ER}(\mathcal{L}, I_{\mathbf{P}}(P_j[U]))$ and so we deduce that $\mathbf{M}, P_j[U] \models B(r_x)$. Since $M \not\models \bigwedge_{r\in E} \beta(P_j[U], r)$ for each set of rules E from rule base $E^{\mathbf{M}}_{\mathcal{L}}$, we now arrive at $M \not\models \beta(P_j[U], r_x)$, which is a contradiction to $\mathbf{M}, P_j[U] \models B(r_x)$. Therefore, $M \models \lambda(\mathbf{P})$. □

7.3 Ordered Completion and Translational Semantics for MLPs on Finite Structures

In this section, we develop *ordered completion* for nonground normal MLPs based on the approach by Asuncion et al. (2012). We consider MLPs in the Datalog setting, i.e., a

normal MLP $\mathbf{P}$ can be viewed as a *modular nonmonotonic Datalog program* that has an infinite set of constants $\mathcal{C}$ and is domain-independent (this is ensured by forcing safety conditions to rules in $\mathbf{P}$). Grounding of $\mathbf{P}$ is done with respect to a *finite relational structure* $\mathfrak{M}$ (extended to MLPs), having a finite universe $U_{\mathfrak{M}}$ accessible by constants; it is the active domain we are restricted to. We also need to adapt the notion of answer set for this setting, which we develop in §7.3.1 by defining a *translational semantics* for MLPs in second-order logic. This semantics will also be used to prove the correctness of ordered completion, whose definitions and results are going to be presented in §7.3.2.

As in §7.2, we assume that MLP $\mathbf{P}$ is normal. Without loss of generality, we assume that MLPs do not contain facts, i.e., rules of form (3.2) have nonempty body, since we can remove facts from an MLP and map them to extensional relations in a finite relational structure.

7.3.1 Finite Structures and Translational Semantics for MLPs

We start with defining finite structures for MLPs, which require the notion of intensional and extensional predicates, as customary in the Datalog setting.

Definition 7.10 (Intensional and extensional predicates).
Given an MLP $\mathbf{P}$ and a predicate symbol $p \in \mathcal{P}$ of $\mathbf{P}$, we call p *intensional* if it occurs in the head of a rule in a module of $\mathbf{P}$ or in the formal input parameters $\mathbf{q}_i$ of a module $m_i = (P_i[\mathbf{q}_i], R_i)$, and *extensional* otherwise.

Intuitively, intensional predicates are defined by the rules in $\mathbf{P}$ and the input given to a module instantiation, whereas extensional predicates stem from the extension given by a relational structure, which will be defined next.

Definition 7.11 (Relational structure).
A *finite (Herbrand) relational structure for* $\mathbf{P}$ (H-structure) is defined as a pair $\mathfrak{M} = \left(U_{\mathfrak{M}}, \cdot^{\mathfrak{M}}\right)$, where the finite universe $U_{\mathfrak{M}}$ consists of constants in $\mathbf{P}$ and $\cdot^{\mathfrak{M}}$ is a mapping associating

- each constant c in $\mathbf{P}$ with itself, i.e., $c^{\mathfrak{M}} = c$,
- each extensional predicate q in $\mathbf{P}$ with a relation $q^{\mathfrak{M}}$ over $\mathfrak{M}$, where $q^{\mathfrak{M}}$ has the same arity as q,
- each intensional predicate p in a module $m_i = (P_i[\mathbf{q}_i], R_i)$, together with each input S from the value calls $P_i[S] \in \mathrm{VC}(\mathbf{P})$, with a relation $p^{\mathfrak{M},S}$ whose arity is the same as p.

Note that extensional predicates q only appear once in $\mathfrak{M}$, while intensional predicates p are labelled with the set S to distinguish different instantiations for value calls $P_i[S] \in \mathrm{VC}(\mathbf{P})$ of MLP $\mathbf{P}$.

Next, we define signatures and grounding of an MLP $\mathbf{P}$.

Definition 7.12 (Signature).
The *signature* of an MLP $\mathbf{P}$ contains all intensional predicates, extensional predicates, and constants occurring in $\mathbf{P}$. The set of intensional (respectively, extensional) predicates in a module m is denoted by $Int(m)$ (respectively, $Ext(m)$).

The grounding process is gradually defined as follows.

Definition 7.13 (Grounding with relational structures).
The grounding of a rule r under $\mathfrak{M}$ is the set $gr(r, \mathfrak{M})$ of all ground instances of r by replacing all variables occurring in r by domain objects in $\mathfrak{M}$. The *grounding of a module* $m = (P[\mathbf{q}], R)$ *with respect to* $\mathfrak{M}$, denoted by $gr(m, \mathfrak{M})$, is defined as

$$gr(m, \mathfrak{M}) = (P[\mathbf{q}], gr(m, R, \mathfrak{M})) \ ,$$

where

$$gr(m, R, \mathfrak{M}) = \bigcup_{r \in R} gr(r, \mathfrak{M}) \cup \left\{ q(\mathbf{c}) \mid q \in Ext(m) \wedge \mathbf{c} \in q^{\mathfrak{M}} \right\} \ .$$

Finally, the *grounding of* $\mathbf{P}$ *with respect to* $\mathfrak{M}$ is

$$gr(\mathbf{P}, \mathfrak{M}) = (gr(m_1, \mathfrak{M}), \dots, gr(m_n, \mathfrak{M})) \ .$$

Intuitively, $gr(\cdot, \mathfrak{M})$ means that rules are grounded with respect to $\mathfrak{M}$ and facts are taken from the finite structure as a database.

Definition 7.14 (Relational structures as answer sets).
A relational structure $\mathfrak{M}$ is an answer set of the normal MLP $\mathbf{P}$ if and only if the MLP interpretation

$$\mathbf{M} = (\mathcal{E}(\mathfrak{M}, P_i[S]) \cup \mathcal{I}(\mathfrak{M}, P_i[S]) \mid P_i[S] \in \mathrm{VC}(\mathbf{P}))$$

is an answer set of the MLP $gr(\mathbf{P}, \mathfrak{M})$ according to Definition 3.10, where

$$\mathcal{E}(\mathfrak{M}, P_i[S]) = \left\{ p(\mathbf{c}) \mid p \in Ext(m_i) \wedge \mathbf{c} \in p^{\mathfrak{M}} \right\}$$

and

$$\mathcal{I}(\mathfrak{M}, P_i[S]) = \left\{ p(\mathbf{c}) \mid p \in Int(m_i) \wedge \mathbf{c} \in p^{\mathfrak{M}, S} \right\} \ .$$

Next, we define the translational semantics for MLPs, which recasts an MLP into an equivalent second-order sentence $\Phi(\mathbf{P})$ akin to the proof of Proposition 3.1. For ordinary answer set programs P, the translational semantics is based on the stable model operator $\mathrm{SM}(P)$ (see Asuncion et al., 2012; Ferraris et al., 2011; Lin and Zhou, 2011) and generalizes to MLPs with $\Phi(\mathbf{P})$. In the following section, the translational semantics will be used to prove ordered completion for MLPs.

Let π be a predicate symbol from the signature of MLP $\mathbf{P}$. If π is an intensional predicate p then $\widetilde{\pi}$ is defined as the fresh predicate symbol $\widetilde{p}$ not occurring in the signature of $\mathbf{P}$, and if p is extensional then $\widetilde{\pi} = p$. For a list $\mathbf{p} = p_1, \ldots, p_\ell$ of predicate symbols from $\mathbf{P}$, we define $\widetilde{\mathbf{p}} = \widetilde{p_1}, \ldots, \widetilde{p_\ell}$. For an ordinary atom $p(\mathbf{x})$, we let $\widetilde{p(\mathbf{x})} = \widetilde{p}(\mathbf{x})$, and for a module atom $\beta(\mathbf{x}) = P_k[\mathbf{p}].o(\mathbf{x})$ we let $\widetilde{\beta(\mathbf{x})} = P_k[\widetilde{\mathbf{p}}].\tilde{o}(\mathbf{x})$. For a set of atoms $A = \{\alpha_1, \ldots, \alpha_k\}$ we define $\widetilde{A} = \{\widetilde{\alpha_1}, \ldots, \widetilde{\alpha_k}\}$. Recall that $\gamma(\mathbf{P})$ is defined in Definition 7.3 for modular completion.

Definition 7.15 (Translational semantics for MLPs).
Let $\mathbf{P}$ be a nonground and normal MLP, let $I = \{p_1, \ldots, p_k\}$ be the intensional predicates of $\mathbf{P}$, and let $J = \{\widetilde{p_1}, \ldots, \widetilde{p_k}\}$ be a set a fresh predicates not occurring in $\mathbf{P}$.

We define $\Phi(\mathbf{P})$ to be the second order logic sentence

$$\gamma(\mathbf{P}) \wedge \neg\exists J\,[(J < I) \wedge \tilde{\gamma}(\mathbf{P})] \quad ,$$

where

- for the variables $\mathbf{x} = \{x_1, \ldots, x_y\}$ occurring in $\mathbf{P}$,

$$\tilde{\gamma}(\mathbf{P}) = \forall \mathbf{x} \bigwedge_{P_i[S] \in \mathrm{VC}(\mathbf{P})} \bigwedge_{r \in R(m_i)} \tilde{\gamma}(P_i[S], r) \quad ,$$

  where

$$\tilde{\gamma}(P_i[S], r) = \bigwedge \widetilde{B_o^+(r)}^S \wedge \bigwedge_{\beta(\mathbf{t}) \in \widetilde{B_m^+(r)}} \mu(P_i[S], \beta(\mathbf{t})) \quad \wedge$$

$$\bigwedge \neg . B_o^-(r)^S \wedge \bigwedge_{\beta(\mathbf{t}) \in B_m^-(r)} \bar{\mu}(P_i[S], \beta(\mathbf{t})) \supset \widetilde{H(r)}^S$$

- $J < I$ is short for $J \leq I \wedge \neg(I \leq J)$ such that

$$J \leq I = \bigwedge_{P_i[S] \in \mathrm{VC}(\mathbf{P})} \bigwedge_{j=1}^{k} \forall \mathbf{x} \left(\widetilde{p_j}^S(\mathbf{x}_j) \supset {p_j}^S(\mathbf{x}_j) \right)$$

  and

$$I \leq J = \bigwedge_{P_i[S] \in \mathrm{VC}(\mathbf{P})} \bigwedge_{j=1}^{k} \forall \mathbf{x} \left({p_j}^S(\mathbf{x}_j) \supset \widetilde{p_j}^S(\mathbf{x}_j) \right) \quad .$$

Note that $\tilde{\gamma}(P_i[S], r)$ replaces $B_o^+(r)$, $B_m^+(r)$, and $H(r)$ in $\gamma(P_i[S], r)$ by $\widetilde{B_o^+(r)}$, $\widetilde{B_m^+(r)}$ and $\widetilde{H(r)}$, respectively.

The next result shows that the translational semantics captures the MLP semantics for relational structures.

Theorem 7.5 (MLP translational semantics)
Let $\mathbf{P}$ be a normal MLP. The answer sets of $\mathbf{P}$ correspond one-to-one to the models of $\Phi(\mathbf{P})$.

Proof Let $\mathfrak{A} = (U_{\mathfrak{A}}, \cdot^{\mathfrak{A}})$ be a finite H-structure for $\mathbf{P}$, let

$$\mathbf{M} = (\mathcal{E}(\mathfrak{A}, P_i[S]) \cup \mathcal{I}(\mathfrak{A}, P_i[S]) \mid P_i[S] \in \mathrm{VC}(\mathbf{P})) \quad ,$$

be an interpretation for $\mathbf{P}$, and let

$$M = \bigcup_{P_i[S] \in \mathrm{VC}(\mathbf{P})} (M_i/S)^S \quad .$$

Observe that $\mathfrak{M}$ is a structure for $\Phi(\mathbf{P})$ that is constructed from $\mathfrak{A}$ by setting

- $U_{\mathfrak{M}} = U_{\mathfrak{A}}$;
- $c^{\mathfrak{M}} = c^{\mathfrak{A}}$ for each constant symbol c of $\mathbf{P}$;
- $(q^S)^{\mathfrak{M}} = q^{\mathfrak{A}}$ for extensional n-ary predicates q and any value call $P_i[S] \in \mathrm{VC}(\mathbf{P})$;
- $(p^S)^{\mathfrak{M}} = p^{\mathfrak{A},S}$ for intensional n-ary predicates p and a value call $P_i[S] \in \mathrm{VC}(\mathbf{P})$; and
- $(\widetilde{p}^S)^{\mathfrak{M}} \subset p^{\mathfrak{A},S}$ for intensional n-ary predicates p and a value call $P_i[S] \in \mathrm{VC}(\mathbf{P})$.

Note that $\widetilde{p}^S$ encodes a proper subset of p^S, which we use in subformula $J < I$.

We first show that $\mathfrak{M} \models \gamma(\mathbf{P})$ iff $\mathbf{M} \models gr(\mathbf{P}, \mathfrak{A})$. To wit,

$\mathbf{M} \models gr(\mathbf{P}, \mathfrak{A})$
$\iff$ $\mathbf{M}$ satisfies all ground rules at all $P_i[S] \in \mathrm{VC}(\mathbf{P})$ by Definition 7.14
$\iff$ $\mathbf{M}, P_i[S] \models r\theta$, for an $r \in R(m_i)$ and ground substitution θ by Definition 3.6
$\iff$ $M \models \gamma(P_i[S], a(\mathbf{x}))\theta$ for $H(r) = \{a(\mathbf{x})\}$ by Lemma 7.1
$\iff$ $\mathfrak{M} \models \gamma(P_i[S], a(\mathbf{x}))\theta$ for $H(r) = \{a(\mathbf{x})\}$ by reconstruction from $\mathfrak{A}$
$\iff$ $\mathfrak{M} \models \gamma(P_i[S], a(\mathbf{x}))$ by $\mathfrak{M}$ being finite
$\iff$ $\mathfrak{M} \models \gamma(\mathbf{P})$ by Definition 7.3 .

Next we show that $\mathfrak{M} \models \neg\exists J\,[(J < I) \wedge \tilde{\gamma}(\mathbf{P})]$ iff there exists no $\mathbf{N}$ such that $\mathbf{N} < \mathbf{M}$ and $\mathbf{N}$ satisfies $f\,gr(\mathbf{P}, \mathfrak{A})^{\mathbf{M}}$.

We obtain

$$\mathfrak{M} \models \neg\exists J\,[(J < I) \wedge \tilde{\gamma}(\mathbf{P})]$$

if and only if there are no H-structures $\mathfrak{B}$ for $\mathbf{P}$ such that $\mathfrak{B} = (U_{\mathfrak{B}}, \cdot^{\mathfrak{B}})$ satisfies

- $U_{\mathfrak{B}} = U_{\mathfrak{A}}$;
- $c^{\mathfrak{B}} = c^{\mathfrak{A}}$ for all constant symbols c;
- $q^{\mathfrak{B}} = q^{\mathfrak{A}}$ for all extensional predicates q;
- $p^{\mathfrak{B},S} \subseteq p^{\mathfrak{A},S}$ for all intensional predicates p and all value calls $P_i[S]$ such that for some intensional predicate $\check{p}$, $\check{p}^{\mathfrak{B},S} \subset \check{p}^{\mathfrak{A},S}$; and
- for all ground substitutions θ such that for all rules r in $R(m_i)$, and all value calls $P_i[S]$, if $\mathbf{N}, P_i[S] \models B^+(r\theta)$ and $\mathbf{M}, P_i[S] \not\models B^-(r\theta)$ then $\mathbf{N}, P_i[S] \models H(r\theta)$, where

$$\mathbf{N} = (\mathcal{E}(\mathfrak{B}, P_i[S]) \cup \mathcal{I}(\mathfrak{B}, P_i[S]) \mid P_i[S] \in \mathrm{VC}(\mathbf{P})) \quad ,$$

if and only if there exists no $\mathbf{N}$ such that $\mathbf{N} < \mathbf{M}$, and for all value calls $P_i[S]$ such that for all rules $r\theta \in fgr(\mathbf{P}, \mathfrak{A})(P_i[S])^{\mathbf{M}}$ for all ground substitutions θ such that $\mathbf{N}, P_i[S] \models r\theta$, if and only if there exists no $\mathbf{N}$ such that $\mathbf{N} < \mathbf{M}$ and $\mathbf{N} \models fgr(\mathbf{P}, \mathfrak{A})^{\mathbf{M}}$.

Now, we have that for all $\mathbf{M}$ such that $\mathbf{M} \models gr(\mathbf{P}, \mathfrak{A})$, there does not exist an $\mathbf{N}$ such that $\mathbf{N} < \mathbf{M}$ and $\mathbf{N} \models fgr(\mathbf{P}, \mathfrak{A})^{\mathbf{M}}$ if and only if $\mathfrak{M} \models \gamma(\mathbf{P}) \wedge \neg\exists J\,[(J < I) \wedge \tilde{\gamma}(\mathbf{P})]$. Thus, $\mathfrak{A}$ is an answer set of $\mathbf{P}$ if and only if $\mathfrak{M} \models \Phi(\mathbf{P})$, and so we have shown that the answer sets of $\mathbf{P}$ correspond one-to-one to the models of $\Phi(\mathbf{P})$. □

7.3.2 Ordered Completion for MLPs

In the following, we develop ordered completion for MLPs. Given an MLP $\mathbf{P}$, our goal is to give a translation of $\mathbf{P}$ to a first-order formula such that the models of the latter correspond to the answer sets of the former. The basic intuition of ordered completion is to recast program completion as defined in §7.1 to the nonground setting with a modification concerning the order between predicates.

Following Asuncion et al. (2012), we use labeled predicates D to keep track of the derivation/dependency order of predicates occurring in an MLP $\mathbf{P}$. Essentially, predicate D is labeled with super- and subscripts describing the two related predicates with their inputs from value calls (the former is used in deriving the latter, in a transitive way). For example, the atom $D^{q,P_k[T]}_{p,P_i[S]}(\mathbf{y}, \mathbf{x})$ means that $q(\mathbf{y})$ in a value call $P_k[T]$ is used to derive $p(\mathbf{x})$ in $P_i[S]$. Hence, the formula

$$D^{q,P_k[T]}_{p,P_i[S]}(\mathbf{y}, \mathbf{x}) \wedge \neg D^{p,P_i[S]}_{q,P_k[T]}(\mathbf{x}, \mathbf{y})$$

expresses that there is no cyclic dependency between $q(\mathbf{y})$ and $p(\mathbf{x})$, which is essential in ordered completion.

Definition 7.16 (Ordered modular derivation).
Let $P_i[S]$ and $P_k[T]$ be value calls from MLP $\mathbf{P}$, and let $p(\mathbf{x})$ and $q(\mathbf{y})$ be two ordinary atoms appearing in m_i and m_k, respectively. The ordered modular derivation is the formula

$$\delta(p(\mathbf{x}), P_i[S], q(\mathbf{y}), P_k[T]) = D^{q,P_k[T]}_{p,P_i[S]}(\mathbf{y}, \mathbf{x}) \wedge \neg D^{p,P_i[S]}_{q,P_k[T]}(\mathbf{x}, \mathbf{y}) \ .$$

We can now use δ to treat module atoms using an ordering on the predicates. Given a value call $P_i[S]$ of the module $m_i = (P_i[\mathbf{q}_i], R_i)$, let $\beta(\mathbf{y}) = P_k[\mathbf{p}].o(\mathbf{y})$ be a module atom for accessing a module $m_k = (P_k[\mathbf{q}_k], R_k)$. In the following definition, we will reconfigure the translation for module atoms μ as defined in Definition 7.1 to $\widehat{\mu}$ with atom $a(\mathbf{x})$ as an additional argument, which not only takes care of matching labels but also prevents loops between the output atom of $\beta(\mathbf{y})$ and $a(\mathbf{x})$, as well as loops between input predicates and formal arguments of the respective module call. To this end, we reuse formula $\epsilon(P_i[S], P_k[T])$ from Definition 7.1 of §7.1 for matching the interpretation of $\mathbf{p}$ to a value call $P_k[T]$ of $\beta(\mathbf{y})$ in our nonground setting. We obtain the accurate $P_k[T]$ by ranging over all possible subsets $T \subseteq \mathrm{HB}_{\mathbf{P}}|_{\mathbf{q}_k}$ of the called module m_k with formal input parameters $\mathbf{q}_k$. For each of the subsets, we translate the module atom $\beta(\mathbf{y})$ to its output predicate labeled with the corresponding value call $P_k[T]$.

In the following definition, p_j and $q_{k,j}$ stem from the input predicate list $\mathbf{p} = p_1, \ldots, p_{n_k}$ of module atom $\beta(\mathbf{y})$ and the formal arguments $\mathbf{q}_k = q_{k,1}, \ldots, q_{k,n_k}$ of module m_k.

Definition 7.17 (Ordered module atom completion).
Let $\mathbf{P}$ be a normal MLP, let $P_i[S] \in \mathrm{VC}(\mathbf{P})$ be a value call from $\mathbf{P}$, let $\beta(\mathbf{y}) = P_k[\mathbf{p}].o(\mathbf{y})$ be a module atom from module m_i, and let $a(\mathbf{x})$ be an ordinary atom from m_i. The *ordered module atom completion* is defined as

$$\begin{aligned}\widehat{\mu}(P_i[S], \beta(\mathbf{y}), a(\mathbf{x})) = \\ \bigvee_{P_k[T] \in \mathrm{VC}(\mathbf{P})} \big(\epsilon(P_i[S], P_k[T]) \wedge o^T(\mathbf{y}) \wedge \delta(a(\mathbf{x}), P_i[S], o(\mathbf{y}), P_k[T]) \quad \wedge \\ \bigwedge_{j=1}^{n_k} \bigwedge_{\chi^T(q_{i,j}(\mathbf{c}))=1} \big(\delta(q_{k,j}(\mathbf{c}), P_k[T], p_j(\mathbf{c}), P_i[S])\big)\big) \ .\end{aligned}$$

We are now in the position to define ordered completion. While Clark's completion formula (Clark, 1978) is based on logical biconditional $\equiv$ for defining the completion of relations, we split the completion into its logical equivalent form using the conjunction of two material conditionals. The "only-if-part" of the ordered completion is $\hat{\gamma}(P_i[S], a(\mathbf{x}))$, which lifts $\gamma(P_i[S], r)$ to the nonground case by merging all supporting rules for $a(\mathbf{x})$. The "if-part" is $\hat{\sigma}(P_i[S], a(\mathbf{x}))$, which is based on $\sigma(\mathbf{P}, P_i[S])$ and applies to every intensional predicate a. Intuitively, $\hat{\sigma}$ makes sure that whenever a head is true,

then there must be some rule with the body satisfied, plus there is no loop involving the head and any atom in the body (both ordinary and module atoms), or between the input predicates and the corresponding formal input parameters of the called module; this is encoded in δ and $\widehat{\mu}$, respectively. We assume that rules with intensional predicate a use the same tuple $\mathbf{x}$ of distinct variables, i.e., a predicate a appearing in the head of a rule always has the form $a(\mathbf{x})$.

Definition 7.18 (Ordered modular completion).
Let $m_i = (P_i[\mathbf{q}_i], R_i)$ be a module from MLP $\mathbf{P}$, let $P_i[S] \in \mathrm{VC}(\mathbf{P})$ be a value call, let $r \in R(m_i)$ be a rule, let $a(\mathbf{x})$ be an atom with intensional predicate $a \in \mathit{Int}(m_i)$ such that $H(r) = \{a(\mathbf{x})\}$, and let $\mathbf{y}$ be the free variables in the body of r. We define

$$\hat{\beta}(P_i[S], r) = \exists \mathbf{y} \bigwedge B_o^+(r)^S \wedge \bigwedge_{\beta(\mathbf{z}) \in B_m^+(r)} \widehat{\mu}(P_i[S], \beta(\mathbf{z}), a(\mathbf{x})) \quad \wedge$$
$$\bigwedge_{b \in \mathit{Int}(m_i)} \bigwedge_{b(\mathbf{z}) \in B_o^+(r)} \delta(a(\mathbf{x}), P_i[S], b(\mathbf{z}), P_i[S]) \wedge$$
$$\bigwedge \neg . B_o^-(r)^S \wedge \bigwedge_{\beta(\mathbf{z}) \in B_m^-(r)} \bar{\mu}(P_i[S], \beta(\mathbf{z})) \ .$$

and

$$\hat{\sigma}(P_i[S], a(\mathbf{x})) = \forall \mathbf{x} \left(a^S(\mathbf{x}) \supset \bigvee_{r \in \mathrm{SR}(a(\mathbf{x}), R_i)} \hat{\beta}(P_i[S], r) \right) \ .$$

For an MLP $\mathbf{P}$ we define

$$\hat{\gamma}(\mathbf{P}) = \bigwedge_{P_i[S] \in \mathrm{VC}(\mathbf{P})} \bigwedge_{a \in \mathit{Int}(m_i)} \gamma(P_i[S], a(\mathbf{x}))$$

and

$$\hat{\sigma}(\mathbf{P}) = \bigwedge_{P_i[S] \in \mathrm{VC}(\mathbf{P})} \bigwedge_{a \in \mathit{Int}(m_i)} \hat{\sigma}(P_i[S], a(\mathbf{x})) \ .$$

Example 7.11 (cont'd) Take $\mathbf{P}$ from Example 7.1 and the labels S_1, S_2^0, and S_2^1 from Example 7.9. We have

$$\hat{\gamma}(P_1[\varnothing], p) = \left(\neg p^{S_1} \wedge r^{S_2^0} \right) \vee \left(p^{S_1} \wedge r^{S_2^1} \right) \supset p^{S_1} \ .$$

Moreover, we get

$$\hat{\sigma}(P_1[\varnothing], p) = p^{S_1} \supset \left(\neg p^{S_1} \wedge r^{S_2^0} \wedge D_{p,S_1}^{r,S_2^0} \wedge \neg D_{r,S_2^0}^{p,S_1} \wedge D_{q,S_2^0}^{p,S_1} \wedge \neg D_{p,S_1}^{q,S_2^0} \right) \vee$$
$$\left(p^{S_1} \wedge r^{S_2^1} \wedge D_{p,S_1}^{r,S_2^1} \wedge \neg D_{r,S_2^1}^{p,S_1} \wedge D_{q,S_2^1}^{p,S_1} \wedge \neg D_{p,S_1}^{q,S_2^1} \right) \ .$$

To capture the closure condition of the dependencies $D^{p,P_i[S]}_{q,P_k[T]}(\mathbf{x},\mathbf{y})$ not only inside but also across module instances, we consider triples of value calls $P_i[S]$, $P_j[T]$, and $P_k[U]$ (not necessarily distinct) coming from the call graph $\mathrm{CG}_\mathbf{P}$.

Definition 7.19 (Ordered modular transitive derivation).
Let $\mathbf{P}$ be a normal MLP. The *ordered modular transitive derivation* $\tau(\mathbf{P})$ is the conjunction of formulas

$$\forall \mathbf{xyz}\left(D^{p,P_i[S]}_{q,P_j[T]}(\mathbf{x},\mathbf{y}) \wedge D^{q,P_j[T]}_{r,P_k[U]}(\mathbf{y},\mathbf{z}) \supset D^{p,P_i[S]}_{r,P_k[U]}(\mathbf{x},\mathbf{z})\right)$$

for all modules m_i, m_j, m_k from $\mathbf{P}$ such that $p \in \mathit{Int}(m_i)$, $q \in \mathit{Int}(m_j)$, $r \in \mathit{Int}(m_k)$, and $P_i[S], P_j[T], P_k[U] \in \mathrm{VC}(\mathbf{P})$.

Then, the ordered completion for an intensional predicate a is simply the conjunction of $\hat{\gamma}$, $\hat{\sigma}$, and τ. The ordered completion thus collects the completions for all value calls in the call graph $\mathrm{CG}_\mathbf{P}$ and the closure axiom of the dependency order between labeled predicates.

Definition 7.20 (Ordered completion for MLPs).
Let $\mathbf{P}$ be a normal MLP. The *ordered completion of* $\mathbf{P}$ is defined as the conjunction of the ordered derivation and the ordered intensional completion of $\mathbf{P}$,

$$\Omega(\mathbf{P}) = \hat{\gamma}(\mathbf{P}) \wedge \hat{\sigma}(\mathbf{P}) \wedge \tau(\mathbf{P}) \ .$$

Comparing ordered completion to the characterization of MLPs with loop formulas $\Lambda(\mathbf{P}) = \gamma(\mathbf{P}) \wedge \sigma(\mathbf{P}) \wedge \lambda(\mathbf{P})$ defined in the previous section yields a striking resemblance of the shape of formulae. While $\Lambda(\mathbf{P})$ gives us the characterization via program completion $\gamma(\mathbf{P}) \wedge \sigma(\mathbf{P})$ and additional loop formulae $\lambda(\mathbf{P})$ that are based on cyclic dependencies in the MLP, ordered completion $\Omega(\mathbf{P})$ uses an adapted notion of program completion $\hat{\gamma}(\mathbf{P}) \wedge \hat{\sigma}(\mathbf{P})$ that includes expressions about noncyclic dependencies in the MLP and adds transitive derivation formulae $\tau(\mathbf{P})$.

Example 7.12 (cont'd) The conjunction of the formulas in Example 7.11 gives us

$$\hat{\gamma}(P_1[\varnothing], p) \wedge \hat{\sigma}(P_1[\varnothing], p)$$

for module m_1 of the MLP $\mathbf{P}$ in Example 7.1. For module m_2 we obtain the formulas

$$\hat{\gamma}(P_2[\varnothing], r) \wedge \hat{\sigma}(P_2[\varnothing], r) = \left(q^{S^0_2} \supset r^{S^0_2}\right) \wedge \left(r^{S^0_2} \supset q^{S^0_2} \wedge D^{q,S^0_2}_{r,S^0_2} \wedge \neg D^{r,S^0_2}_{q,S^0_2}\right)$$

and

$$\hat{\gamma}(P_2[\{q\}], r) \wedge \hat{\sigma}(P_2[\{q\}], r) = q^{S^1_2} \wedge \left(q^{S^1_2} \supset r^{S^1_2}\right) \wedge \left(r^{S^1_2} \supset q^{S^1_2} \wedge D^{q,S^1_2}_{r,S^1_2} \wedge \neg D^{r,S^1_2}_{q,S^1_2}\right) \ .$$

The transitive derivation is

$$\tau(\mathbf{P}) = \bigwedge D^{\alpha_1,\nu_1}_{\alpha_2,\nu_2} \wedge D^{\alpha_2,\nu_2}_{\alpha_3,\nu_3} \supset D^{\alpha_1,\nu_1}_{\alpha_3,\nu_3}$$

where $\alpha_i \in \{p, q, r\}$, $\nu_i = P_1[S_1]$ if $\alpha_i = p$ and $\nu_i \in \{P_2[S_2^0], P_2[S_2^1]\}$ otherwise. The ordered completion $\Omega(\mathbf{P})$ is then the conjunction of all four formulas above, i.e., $\hat{\gamma}(P_1[\varnothing], p) \wedge \hat{\sigma}(P_1[\varnothing], p) \wedge \hat{\gamma}(P_2[\varnothing], r) \wedge \hat{\sigma}(P_2[\varnothing], r) \wedge \hat{\gamma}(P_2[\{q\}], r) \wedge \hat{\sigma}(P_2[\{q\}], r) \wedge \tau(\mathbf{P})$, which has a single model whose projection to labeled atoms is $\{r^{S_2^1}, q^{S_2^1}\}$. This model corresponds to the answer set mentioned in Example 7.1.

In order to prove that the models of the ordered completion $\Omega(\mathbf{P})$ capture the answer sets of an MLP $\mathbf{P}$, we define the derivation order of an MLP $\mathbf{P}$ with respect to an H-structure $\mathfrak{M}$. Recall that $\mathcal{I}(\mathfrak{M}, P_i[S])$ and $\mathcal{E}(\mathfrak{M}, P_i[S])$ denote the intensional and extensional atoms contained in $\mathfrak{M}$ at $P_i[S]$, respectively (confer Definition 7.14).

Definition 7.21 (Derivation order).
The *derivation order* of an H-structure $\mathfrak{M}$ on a normal MLP $\mathbf{P}$ is a sequence $D^{\mathfrak{M}}(\mathbf{P}) = (p_1(\mathbf{c}_1), \nu_1), \ldots, (p_k(\mathbf{c}_k), \nu_k)$ of pairs of ground atoms and value calls such that

- $\{\nu_1, \ldots, \nu_k\} \subseteq \mathrm{VC}(\mathbf{P})$,
- for all $P_i[S] \in \mathrm{VC}(\mathbf{P})$,

$$\mathcal{I}(\mathfrak{M}, P_i[S]) = \left\{ p_j(\mathbf{c}_j) \mid \begin{array}{c} \nu_j = P_i[S] \wedge (p_j(\mathbf{c}_j), \nu_j) \in \left\{D^{\mathfrak{M}}(\mathbf{P})\right\} \wedge \\ p_j \in \mathit{Int}(m_i) \wedge \mathbf{c}_j \in p_j^{\mathfrak{M},S} \end{array} \right\} ,$$

- all $\mathcal{I}(\mathfrak{M}, P_i[S])$ cover $D^{\mathfrak{M}}(\mathbf{P})$, i.e.,

$$\bigcup_{P_i[S] \in \mathrm{VC}(\mathbf{P})} \{(p(\mathbf{c}), P_i[S]) \mid p(\mathbf{c}) \in \mathcal{I}(\mathfrak{M}, P_i[S])\} = \left\{D^{\mathfrak{M}}(\mathbf{P})\right\} ,$$

- for all $j = 1, \ldots, k$ and for all $P_i[S] \in \mathrm{VC}(\mathbf{P})$ there exists a rule $r \in R(m_i)$ and a ground substitution θ for the variables appearing in r such that
  - $(H(r)\theta, P_i[S]) = \big(p_j(\mathbf{c}_j), \nu_j\big)$,
  - for all intensional $q(\mathbf{t}) \in B_o^+(r)$,

$$(q(\mathbf{t})\theta, P_i[S]) \in \left\{(p_1(\mathbf{c}_1), \nu_1), \ldots, \big(p_{j-1}(\mathbf{c_{j-1}}), \nu_{j-1}\big)\right\} ,$$

  - for all intensional $q(\mathbf{t}) \in B_o^-(r)$,

$$q(\mathbf{t})\theta \notin \mathcal{I}(\mathfrak{M}, P_i[S]) ,$$

– for all $P_k[\mathbf{p}].o(\mathbf{t}) \in B_m^+(r)$ such that for $T = (\mathcal{J}(\mathfrak{M}, P_i[S]) \cup S)|_{\mathbf{p}}^{\mathbf{q}_k}$,

$$(o(\mathbf{t})\theta, P_k[T]) \in \left\{(p_1(\mathbf{c}_1), v_1), \ldots, \left(p_{j-1}(\mathbf{c_{j-1}}), v_{j-1}\right)\right\} \ ,$$

– for all $P_k[\mathbf{p}].o(\mathbf{t}) \in B_m^-(r)$ such that for $T = (\mathcal{J}(\mathfrak{M}, P_i[S]) \cup S)|_{\mathbf{p}}^{\mathbf{q}_k}$,

$$o(\mathbf{t})\theta \notin \mathcal{J}(\mathfrak{M}, P_k[T]) \ , \text{ and}$$

– for all extensional $q(\mathbf{t}) \in B_o^+(r)$ and $q(\mathbf{t}) \in B_o^-(r)$, $q(\mathbf{t})\theta \in \mathcal{E}(\mathfrak{M}, P_i[S])$ and $q(\mathbf{t})\theta \notin \mathcal{E}(\mathfrak{M}, P_i[S])$, respectively.

We can now show that derivation orders capture answer sets of normal MLPs.

Lemma 7.6
Let $\mathbf{P}$ be a normal MLP and $\mathfrak{M}$ be a finite H-structure. Then, $\mathfrak{M}$ is an answer set of $\mathbf{P}$ iff $\mathfrak{M} \models \gamma(\mathbf{P})$ and there exists at least one derivation order $D^{\mathfrak{M}}(\mathbf{P})$.

Proof Without loss of generality, we assume that $\mathbf{P}$ is a ground normal MLP.
($\Rightarrow$) Let $\mathfrak{M}$ be an answer set of $\mathbf{P}$. By Theorem 7.5, $\mathfrak{M} \models \gamma(\mathbf{P})$. We construct a sequence of pairs $D = (p_1(\mathbf{c}_1), v_1), \ldots, (p_k(\mathbf{c}_k), v_k)$ from $\mathfrak{M}$ as follows:

- for $j = 1, \ldots, k$ where $v_j = P_i[S] \in \mathrm{VC}(\mathbf{P})$, there exists a rule $r \in R(m_i)$ such that

  1. $(H(r), P_i[S]) = \left(p_j(\mathbf{c}_j), v_j\right)$,
  2. for all intensional $q(\mathbf{c}) \in B_o^+(r)$,

  $$(q(\mathbf{c}), P_i[S]) \in \left\{(p_1(\mathbf{c}_1), v_1), \ldots, \left(p_{j-1}(\mathbf{c_{j-1}}), v_{j-1}\right)\right\} \ ,$$

  3. for all intensional $q(\mathbf{c}) \in B_o^-(r)$,

  $$q(\mathbf{c}) \notin \mathcal{J}(\mathfrak{M}, P_i[S]) \ ,$$

  4. for all $P_k[\mathbf{p}].o(\mathbf{c}) \in B_m^+(r)$ such that for $T = (\mathcal{J}(\mathfrak{M}, P_i[S]) \cup S)|_{\mathbf{p}}^{\mathbf{q}_k}$,

  $$(o(\mathbf{c}), P_k[T]) \in \left\{(p_1(\mathbf{c}_1), v_1), \ldots, \left(p_{j-1}(\mathbf{c_{j-1}}), v_{j-1}\right)\right\} \ ,$$

  5. for all $P_k[\mathbf{p}].o(\mathbf{c}) \in B_m^-(r)$ such that for $T = (\mathcal{J}(\mathfrak{M}, P_i[S]) \cup S)|_{\mathbf{p}}^{\mathbf{q}_k}$,

  $$o(\mathbf{c}) \notin \mathcal{J}(\mathfrak{M}, P_k[T]) \ , \text{ and}$$

  6. for all extensional $q(\mathbf{c}) \in B_o^+(r)$ and $q(\mathbf{c}) \in B_o^-(r)$, $q(\mathbf{c}) \in \mathcal{E}(\mathfrak{M}, P_i[S])$ and $q(\mathbf{c}) \notin \mathcal{E}(\mathfrak{M}, P_i[S])$, respectively;

- for all $P_i[S] \in \mathrm{VC}(\mathbf{P})$ there is no rule $r \in R(m_i)$ such that $(H(r), P_i[S]) \notin \{D\}$ and for $j = k$, (2)–(6) holds.

Let

$$\mathbf{D} = \left(\left\{p_j(\mathbf{c}_j) \mid \left(p_j(\mathbf{c}_j), P_i[S]\right) \in \{D\}\right\} \mid P_i[S] \in \mathrm{VC}(\mathbf{P})\right) \quad .$$

Then, $\mathbf{D}$ is an interpretation for $gr(\mathbf{P}, \mathfrak{M})$. Now consider $\mathbf{M}$ obtained from $\mathfrak{M}$ as in Definition 7.14. From the construction of $\mathbf{D}$ from $\mathfrak{M}$, we conclude that $\mathbf{D} \leq \mathbf{M}$. Since $\mathfrak{M}$ is an answer set of $\mathbf{P}$, thus $\mathbf{M}$ is an answer set for $gr(\mathbf{P}, \mathfrak{M})$, therefore $\mathbf{M} \vDash f\, gr(\mathbf{P}, \mathfrak{M})^{\mathbf{M}}$. By the construction of $\mathbf{D}$, we infer that $\mathbf{D} \vDash f\, gr(\mathbf{P}, \mathfrak{M})^{\mathbf{M}}$. Towards a contradiction, assume that $\mathbf{D} \neq \mathbf{M}$, i.e., $\mathbf{D} < \mathbf{M}$. This would mean that $\mathbf{M}$ is not a minimal model of $f\, gr(\mathbf{P}, \mathfrak{M})^{\mathbf{M}}$, a contradiction. Thus, $\mathbf{D} = \mathbf{M}$. This implies that all $\mathcal{J}(\mathfrak{M}, P_i[S])$ cover D since $\mathbf{D} = \mathbf{M}$. Hence, D is a derivation order of $\mathfrak{M}$ on $\mathbf{P}$.

($\Leftarrow$) Let $\mathfrak{M} \vDash \gamma(\mathbf{P})$ and let $D^{\mathfrak{M}}(\mathbf{P}) = (p_1(\mathbf{c}_1), \nu_1), \ldots, (p_k(\mathbf{c}_k), \nu_k)$ be a derivation order of $\mathfrak{M}$ on $\mathbf{P}$. Let $\mathbf{M}$ be obtained from $\mathfrak{M}$ as in Definition 7.14. Since $\mathfrak{M} \vDash \gamma(\mathbf{P})$, we can infer from Theorem 7.5 that $\mathbf{M}$ is a model of $gr(\mathbf{P}, \mathfrak{M})$ and a model of $f\, gr(\mathbf{P}, \mathfrak{M})^{\mathbf{M}}$. Now we show that $\mathbf{M}$ is a minimal model of $f\, gr(\mathbf{P}, \mathfrak{M})^{\mathbf{M}}$. Assuming the contrary, there exists an interpretation $\mathbf{N} < \mathbf{M}$ such that $\mathbf{N} \vDash f\, gr(\mathbf{P}, \mathfrak{M})^{\mathbf{M}}$. For a value call $P_i[S]$ there exists an atom $p(\mathbf{c}) \in M_i/S$ such that $p(\mathbf{c}) \notin N_i/S$. Since $D^{\mathfrak{M}}(\mathbf{P})$ is a derivation order, we have that $(p(\mathbf{c}), P_i[S])$ appears in the sequence $D^{\mathfrak{M}}(\mathbf{P})$. Without loss of generality, let $(p(\mathbf{c}), P_i[S])$ be the element obtained from $\mathbf{M}$ such that $p(\mathbf{c}) \notin N_i/S$ and $(p(\mathbf{c}), P_i[S])$ appears in $D^{\mathfrak{M}}(\mathbf{P})$ with the least ordinal $j \leq k$. Thus, there exists a rule $r \in R(m_i)$ such that $(H(r), P_i[S]) = (p(\mathbf{c}), P_i[S])$. Following Definition 7.21 for derivation orders, we must have that $\mathbf{M} \vDash B(r)$, and since j is the least ordinal in $D^{\mathfrak{M}}(\mathbf{P})$ such that $p(\mathbf{c}) \notin N_i/S$, we get that also $\mathbf{N} \vDash B(r)$. But $\mathbf{N}, P_i[S] \nvDash H(r)$, which contradicts $\mathbf{N} \vDash f\, gr(\mathbf{P}, \mathfrak{M})^{\mathbf{M}}$. Hence, $\mathbf{M}$ is a minimal model of $f\, gr(\mathbf{P}, \mathfrak{M})^{\mathbf{M}}$, and from this follows that $\mathfrak{M}$ is an answer set for $\mathbf{P}$. □

Based on the translational semantics, Theorem 7.5, and Lemma 7.6, we can now show the correctness of ordered completion.

Theorem 7.7 (MLP ordered completion)
Let $\mathbf{P} = (m_1, \ldots, m_n)$ be a normal MLP. Then,

1. if H-structure $\mathfrak{M}$ is an answer set of $\mathbf{P}$, then $\mathfrak{O}$ is a model of $\Omega(\mathbf{P})$, where $\mathfrak{O}$ satisfies, for all $P_i[S] \in \mathrm{VC}(\mathbf{P})$,
   - $U_{\mathfrak{O}} = U_{\mathfrak{M}}$,
   - for each constant symbol c from $\mathbf{P}$, $c^{\mathfrak{O}} = c^{\mathfrak{M}}$,
   - for each extensional predicate q of $\mathbf{P}$, $\left(q^S\right)^{\mathfrak{O}} = q^{\mathfrak{M}}$, and
   - for each intensional predicate p at $P_i[S]$, $\left(p^S\right)^{\mathfrak{O}} = p^{\mathfrak{M},S}$.

- for the derivation order $D^{\mathfrak{M}}(\mathbf{P}) = (p_1(\mathbf{c}_1), v_1), \ldots, (p_k(\mathbf{c}_k), v_k)$ and ordinals $j_1, j_2 \in \{1, \ldots, k\}$ such that $j_1 \leq j_2$, we set for each predicate $D^{p_{j_1}, v_{j_1}}_{p_{j_2}, v_{j_2}}$,

$$\left(D^{p_{j_1}, v_{j_1}}_{p_{j_2}, v_{j_2}}\right)^{\mathfrak{D}} = \left\{(\mathbf{c_{j_1}}, \mathbf{c_{j_2}}) \mid \left(p_{j_1}(\mathbf{c_{j_1}}), v_{j_1}\right), \left(p_{j_2}(\mathbf{c_{j_2}}), v_{j_1}\right) \in \{D^{\mathfrak{M}}(\mathbf{P})\}\right\} \ .$$

2. if $\mathfrak{D} = (U_{\mathfrak{D}}, \cdot^{\mathfrak{D}})$ is a model of $\Omega(\mathbf{P})$, then the $\mathfrak{M} = (U_{\mathfrak{M}}, \cdot^{\mathfrak{M}})$ is an answer set of $\mathbf{P}$, where $\mathfrak{M}$ satisfies, for all $P_i[S] \in \mathrm{VC}(\mathbf{P})$,

   - $U_{\mathfrak{M}} = U_{\mathfrak{D}}$,
   - for each constant symbol c from $\mathbf{P}$, $c^{\mathfrak{M}} = c^{\mathfrak{D}}$,
   - for each extensional predicate q of $\mathbf{P}$, $q^{\mathfrak{M}} = \left(q^S\right)^{\mathfrak{D}}$, and
   - for each intensional predicate p at $P_i[S]$, $p^{\mathfrak{M},S} = \left(p^S\right)^{\mathfrak{D}}$.

Proof (item 1) Let $\mathfrak{M}$ be an answer set of $\mathbf{P}$. By Lemma 7.6 there exists a derivation order $D^{\mathfrak{M}}(\mathbf{P})$. We show now that $\mathfrak{D} \models \hat{\gamma}(\mathbf{P}) \wedge \hat{\sigma}(\mathbf{P}) \wedge \tau(\mathbf{P})$. By construction of $\mathfrak{D}$ from $D^{\mathfrak{M}}(\mathbf{P})$, we obtain that $\mathfrak{D} \models \tau(\mathbf{P})$, as $D^{\mathfrak{M}}(\mathbf{P})$ satisfies transitivity and does not contain loops. By Theorem 7.5, we get that $\mathfrak{D} \models \hat{\gamma}(\mathbf{P})$ by construction of $\mathfrak{D}$ from $\mathfrak{M}$ agreeing on all predicates and constant symbols, since $\gamma(\mathbf{P}) = \hat{\gamma}(\mathbf{P})$.

We show now that $\mathfrak{D} \models \hat{\sigma}(\mathbf{P})$ by contradiction. Assume that $\mathfrak{D} \not\models \hat{\sigma}(\mathbf{P})$. There exists an $a \in Int(m_i)$ and $P_i[S]$ such that $\mathfrak{D} \not\models \hat{\sigma}(P_i[S], a(\mathbf{x}))$. Thus there exists a ground substitution θ such that $\mathfrak{D} \models a^S(\mathbf{x})\theta$ implies

$$\mathfrak{D} \not\models \bigvee_{r \in \mathrm{SR}(a(\mathbf{x}), R_i)} \hat{\beta}(P_i[S], r)\theta \ .$$

Hence, there is no rule $r \in \mathrm{SR}(a(\mathbf{x}), R_i)$ such that $\hat{\beta}(P_i[S], r)\theta$ is satisfied by $\mathfrak{D}$. As $\mathfrak{D} \models \gamma(\mathbf{P})$ and $\mathfrak{M} \models gr(\mathbf{P}, \mathfrak{M})$, i.e., all rules in $gr(\mathbf{P}, \mathfrak{M})$ must be satisfied. Furthermore, from the existence of $D^{\mathfrak{M}}(\mathbf{P})$ we obtain that there is a rule $r \in \mathrm{SR}(a(\mathbf{x}), R_i)$ such that the body of r is satisfied by $\mathfrak{M}$. We argue now based on $D^{\mathfrak{M}}(\mathbf{P})$ that our initial assumption $\mathfrak{D} \not\models \hat{\sigma}(\mathbf{P})$ leads to a contradiction. Thus, since the body of r is true in $\mathfrak{M}$, all the parts of $\hat{\beta}(P_i[S], r)$ must be satisfied by $\mathfrak{D}$ except for the cases

(a) $\delta(a(\mathbf{x}), P_i[S], b(\mathbf{z}), P_i[S])\theta$ from

$$\bigwedge_{b \in Int(m_i)} \bigwedge_{b(\mathbf{z}) \in B^+_O(r)} \delta(a(\mathbf{x}), P_i[S], b(\mathbf{z}), P_i[S]) \ , \text{ and}$$

(b) $\delta(a(\mathbf{x}), P_i[S], o(\mathbf{y}), P_k[T])\theta$ from $\bigwedge_{\beta(\mathbf{z}) \in B^+_m(r)} \widehat{\mu}(P_i[S], \beta(\mathbf{z}), a(\mathbf{x}))$,

which are not satisfied by $\mathfrak{O}$ by our assumption.

In case (a), where

$$\mathfrak{O} \not\models \delta(a(\mathbf{x}), P_i[S], b(\mathbf{y}), P_i[S])\theta \ ,$$

we have

$$\mathfrak{O} \not\models D^{b,P_i[S]}_{a,P_i[S]}(\mathbf{z},\mathbf{x})\theta \wedge \neg D^{a,P_i[S]}_{b,P_i[S]}(\mathbf{x},\mathbf{z})\theta \ ,$$

thus infer that $\mathfrak{O} \models \neg D^{b,P_i[S]}_{a,P_i[S]}(\mathbf{z},\mathbf{x})\theta$ or $\mathfrak{O} \models D^{a,P_i[S]}_{b,P_i[S]}(\mathbf{x},\mathbf{z})\theta$. In case (b), where

$$\mathfrak{O} \not\models \delta(a(\mathbf{x}), P_i[S], o(\mathbf{y}), P_k[T])\theta \ ,$$

we have assumed that

$$\mathfrak{O} \not\models D^{o,P_k[T]}_{a,P_i[S]}(\mathbf{y},\mathbf{x})\theta \wedge \neg D^{a,P_i[S]}_{o,P_k[T]}(\mathbf{x},\mathbf{y})\theta \ ,$$

and obtain that $\mathfrak{O} \models \neg D^{o,P_k[T]}_{a,P_i[S]}(\mathbf{y},\mathbf{x})\theta$ or $\mathfrak{O} \models D^{a,P_i[S]}_{o,P_k[T]}(\mathbf{x},\mathbf{y})\theta$.

Consider a pair $(a(\mathbf{x})\theta, P_i[S])$ from $D^{\mathfrak{M}}(\mathbf{P})$. According to Definition 7.21, there is a rule whose head is $a(\mathbf{x})$. Without loss of generality, let r from above be the rule such that $H(r)\theta = \{a(\mathbf{x})\theta\}$. We consider now the different types of body atoms of r. For case (a), we consider ordinary atoms, and by Definition 7.21 for all intensional $q(\mathbf{z})\theta$ from $B^+_o(r\theta)$, we obtain that the ordinal of pair $(q(\mathbf{z})\theta, P_i[S])$ is less than the ordinal for $(a(\mathbf{x})\theta, P_i[S])$. Thus, we conclude $\mathfrak{O} \models D^{b,P_i[S]}_{a,P_i[S]}(\mathbf{z},\mathbf{x})\theta$ and $\mathfrak{O} \models \neg D^{a,P_i[S]}_{b,P_i[S]}(\mathbf{x},\mathbf{z})\theta$, which contradicts

$$\mathfrak{O} \not\models \delta(a(\mathbf{x}), P_i[S], b(\mathbf{y}), P_i[S])\theta$$

in case (a). For case (b), we get from Definition 7.21 that all module atoms $P_k[\mathbf{p}].o(\mathbf{y})\theta$ are contained in $B^-_m(r\theta)$ such that $T = (\mathcal{J}(\mathfrak{M}, P_i[S]) \cup S)|^{\mathbf{q}_k}_{\mathbf{p}}$, hence the ordinal of pair $(o(\mathbf{y})\theta, P_k[T])$ is less than the ordinal for $(a(\mathbf{x})\theta, P_i[S])$. Thus, we obtain $\mathfrak{O} \models D^{o,P_k[T]}_{a,P_i[S]}(\mathbf{y},\mathbf{x})\theta$ and $\mathfrak{O} \models \neg D^{a,P_i[S]}_{o,P_k[T]}(\mathbf{x},\mathbf{y})\theta$, which contradicts

$$\mathfrak{O} \not\models \delta(a(\mathbf{x}), P_i[S], o(\mathbf{y}), P_k[T])\theta$$

in case (b). Therefore, both (a) and (b) are satisfied in $\hat{\beta}(P_i[S], r)$, and so we can infer that $\mathfrak{O} \models \hat{\sigma}(P_i[S], a(\mathbf{x}))$. We have shown that $\mathfrak{O} \models \hat{\sigma}(\mathbf{P})$, and can infer that $\mathfrak{O} \models \Omega(\mathbf{P})$.

(item 2) Let $\mathfrak{O} \models \Omega(\mathbf{P})$ and let $\mathfrak{M}$ be as defined. We let

$$\mathbf{M} = (\mathcal{E}(\mathfrak{M}, P_i[S]) \cup \mathcal{J}(\mathfrak{M}, P_i[S]) \mid P_i[S] \in \mathrm{VC}(\mathbf{P})) \ ,$$

and construct from $\mathbf{M}$ the interpretation

$$M = \bigcup_{P_i[S] \in \mathrm{VC}(\mathbf{P})} (M_i/S)^S \ ,$$

and show now that $\mathbf{M}$ is a model of $gr(\mathbf{P}, \mathfrak{M})$ and that $\mathbf{M}$ is a minimal model of $f\,gr(\mathbf{P}, \mathfrak{M})^{\mathbf{M}}$. Theorem 7.5 implies that from $\mathfrak{O} \models \hat{\gamma}(\mathbf{P})$ follows $\mathbf{M} \models gr(\mathbf{P}, \mathfrak{M})$.

We base our proof on the propositional modular loop formula $\Lambda(gr(\mathbf{P}, \mathfrak{M}))$. It suffices to show that for all modular loops $\mathcal{L}$ and cyclic instantiation signatures $\mathcal{S}$ for the ground program $f\,gr(\mathbf{P}, \mathfrak{M})^{\mathbf{M}}$ the set M of ground atoms satisfies

$$M \models \lambda(\mathcal{S}, \mathcal{L}, f\,gr(\mathbf{P}, \mathfrak{M})^{\mathbf{M}}) \ . \tag{7.3}$$

Towards a contradiction, assume that there exists a modular loop $\mathcal{L}$ and cyclic instantiation signatures $\mathcal{S}$ for $f\,gr(\mathbf{P}, \mathfrak{M})^{\mathbf{M}}$ such that $M \not\models \lambda(\mathcal{S}, \mathcal{L}, f\,gr(\mathbf{P}, \mathfrak{M})^{\mathbf{M}})$. There exists an atom $p_0(\mathbf{c}_0) \in \mathcal{L}$ from $P_i[S]$ such that for $p_0 \in Int(m_i)$ formula (7.1) is false in M, i.e., for $p_0(\mathbf{c}_0) \in \mathcal{L}$ in the antecedent of (7.1) $M \models p_0^S(\mathbf{c}_0)$ but the consequent of (7.1) is false in M. This implies for all rules r such that $H(r) = \{p_0(\mathbf{c}_0)\}$ and $B^+(r) \cap \mathcal{L} = \emptyset$, $M \not\models H(r)^S \supset \beta(P_i[S], r)$. Hence, $p_0(\mathbf{c}_0)$ has no external support with respect to $\mathcal{L}$ in $f\,gr(\mathbf{P}, \mathfrak{M})^{\mathbf{M}}$.

By $\mathfrak{O} \models \hat{\gamma}(\mathbf{P}) \wedge \hat{\sigma}(\mathbf{P})$ we can infer that there exists an $r \in \mathrm{SR}(p_0(\mathbf{x}), R_i)$ such that for the ground substitution θ such that $p_0(\mathbf{x})\theta = p_0(\mathbf{c}_0)$, $\mathfrak{O} \models \hat{\beta}(P_i[S], r)\theta$. Hence,

(a) all $\delta(p_0(\mathbf{x}), P_i[S], b(\mathbf{z}), P_i[S])\theta$ from

$$\bigwedge_{b \in Int(m_i)} \bigwedge_{b(\mathbf{z}) \in B_o^+(r)} \delta(p_0(\mathbf{x}), P_i[S], b(\mathbf{z}), P_i[S]) \ , \text{ and}$$

(b) all $\delta(p_0(\mathbf{x}), P_i[S], o(\mathbf{y}), P_k[T])\theta$ from

$$\bigwedge_{\beta(\mathbf{z}) \in B_m^+(r)} \widehat{\mu}(P_i[S], \beta(\mathbf{z}), p_0(\mathbf{x}))$$

are satisfied by $\mathfrak{O}$.

Thus, for the case (a), both $\mathfrak{O} \models D^{b,P_i[S]}_{a,P_i[S]}(\mathbf{z}, \mathbf{x})\theta$ and $\mathfrak{O} \models \neg D^{a,P_i[S]}_{b,P_i[S]}(\mathbf{x}, \mathbf{z})\theta$ hold, and for case (b), both $\mathfrak{O} \models D^{o,P_k[T]}_{a,P_i[S]}(\mathbf{y}, \mathbf{x})\theta$ and $\mathfrak{O} \models \neg D^{a,P_i[S]}_{o,P_k[T]}(\mathbf{x}, \mathbf{y})\theta$ hold. If all $b(\mathbf{z})\theta$ and $\beta(\mathbf{z})\theta$ were not contained in $\mathcal{L}$ then $p_0(\mathbf{c}_0)$ would have external support, contradicting our assumption.

Thus, there exists a $p_1(\mathbf{c}_1) \in \mathcal{L}$ in case (a), or $\beta_1(\mathbf{c}_1) = P_k[\mathbf{p}].p_1(\mathbf{c}_1) \in \mathcal{L}$ and $p_1(\mathbf{c}_1) \in \mathcal{L}$ in case (b), where p_1 is an intensional predicate, such that for the ground substitution θ, we have $p_1(\mathbf{c}_1) \in B_o^+(r)\theta$ and $\beta_1(\mathbf{c}_1) \in B_m^+(r)\theta$, respectively. Hence, in case (a), $\mathfrak{O} \models D^{p_1,P_i[S]}_{p_0,P_i[S]}(\mathbf{c}_1, \mathbf{c}_0)$ and $\mathfrak{O} \models \neg D^{p_0,P_i[S]}_{p_1,P_i[S]}(\mathbf{c}_0, \mathbf{c}_1)$ hold, and in case (b), $\mathfrak{O} \models D^{p_1,P_k[T]}_{p_0,P_i[S]}(\mathbf{c}_1, \mathbf{c}_0)$ and $\mathfrak{O} \models \neg D^{p_0,P_i[S]}_{p_1,P_k[T]}(\mathbf{c}_0, \mathbf{c}_1)$ hold.

We can now apply above procedure starting from $p_1(\mathbf{c}_1) \in \mathcal{L}$ and a rule r such that for the ground substitution θ, $H(r)\theta = \{p_1(\mathbf{c}_1)\}$, and infer that there exists a $p_2(\mathbf{c}_2) \in \mathcal{L}$ in case (a), or $\beta_2(\mathbf{c}_2) = P_k[\mathbf{p}].p_2(\mathbf{c}_2) \in \mathcal{L}$ and $p_2(\mathbf{c}_2) \in \mathcal{L}$ in case (b), such

that for the intensional predicate p_2 we have $p_2(\mathbf{c}_2) \in B_o^+(r)\theta$ and $\beta_2(\mathbf{c}_2) \in B_m^+(r)\theta$, respectively. Hence, in case (a), $\mathfrak{O} \models D_{p_1,P_i[S]}^{p_2,P_i[S]}(\mathbf{c}_2, \mathbf{c}_1)$ and $\mathfrak{O} \models \neg D_{p_2,P_i[S]}^{p_1,P_i[S]}(\mathbf{c}_1, \mathbf{c}_2)$ hold, and $\mathfrak{O} \models D_{p_1,P_i[S]}^{p_2,P_k[T]}(\mathbf{c}_2, \mathbf{c}_1)$ and $\mathfrak{O} \models \neg D_{p_2,P_k[T]}^{p_1,P_i[S]}(\mathbf{c}_1, \mathbf{c}_2)$ in case (b).

Applying this procedure iteratively, we obtain a sequence of ground atoms

$$p_0(\mathbf{c}_0), p_1(\mathbf{c}_1), p_2(\mathbf{c}_2), ... \tag{7.4}$$

such that for all $j \geq 0$, p_j is an intensional predicate, $p_j(\mathbf{c}_j) \in \mathcal{L}$, and in case (a),

$$\mathfrak{O} \models D_{p_j,P_i[S]}^{p_{j+1},P_i[S]}(\mathbf{c_{j+1}}, \mathbf{c}_j) \text{ and } \mathfrak{O} \models \neg D_{p_{j+1},P_i[S]}^{p_j,P_i[S]}(\mathbf{c}_j, \mathbf{c_{j+1}})$$

hold, and in case (b),

$$\mathfrak{O} \models D_{p_j,P_i[S]}^{p_{j+1},P_k[T]}(\mathbf{c_{j+1}}, \mathbf{c}_j) \text{ and } \mathfrak{O} \models \neg D_{p_{j+1},P_k[T]}^{p_j,P_i[S]}(\mathbf{c}_j, \mathbf{c_{j+1}}) \ ,$$

hold.

Since $\mathfrak{O}$ is finite and $\mathfrak{O} \models \tau(\mathbf{P})$ implies the transitivity of derivation sequence (7.4), there must exist an $m \geq 0$ with $m < n$ for an $n \geq 0$ such that $p_m(\mathbf{c}_m) = p_n(\mathbf{c}_n)$ in (7.4), hence (7.4) is finite. We obtain $\mathfrak{O} \models D_{p_j,P_i[S]}^{p_{j+1},P_i[S]}(\mathbf{c_{j+1}}, \mathbf{c}_j)$ in case (a), respectively $\mathfrak{O} \models D_{p_j,P_i[S]}^{p_{j+1},P_k[T]}(\mathbf{c_{j+1}}, \mathbf{c}_j)$ in case (b), for all $j \geq 0$, hence by transitivity, we get $\mathfrak{O} \models D_{p_{m+1},P_i[S]}^{p_n,P_i[S]}(\mathbf{c}_n, \mathbf{c_{m+1}})$ in case (a), respectively $\mathfrak{O} \models D_{p_{m+1},P_i[S]}^{p_n,P_k[T]}(\mathbf{c}_n, \mathbf{c_{m+1}})$ in case (b).

Therefore, in case (a), $\mathfrak{O} \models D_{p_{m+1},P_i[S]}^{p_m,P_i[S]}(\mathbf{c}_m, \mathbf{c_{m+1}})$, respectively in case (b), $\mathfrak{O} \models D_{p_{m+1},P_i[S]}^{p_m,P_k[T]}(\mathbf{c}_m, \mathbf{c_{m+1}})$ holds by $p_m(\mathbf{c}_m) = p_n(\mathbf{c}_n)$. But, in case (a), we infer that $\mathfrak{O} \models \neg D_{p_{j+1},P_i[S]}^{p_j,P_i[S]}(\mathbf{c}_j, \mathbf{c_{j+1}})$, respectively in case (b), $\mathfrak{O} \models \neg D_{p_{j+1},P_k[T]}^{p_j,P_i[S]}(\mathbf{c}_j, \mathbf{c_{j+1}})$ holds for all $j \geq 0$, each leading to a contradiction.

Thus our assumption that M does not satisfy formula (7.3) has been contradicted, which shows that $\mathbf{M}$ is a minimal model of $fgr(\mathbf{P}, \mathfrak{M})^{\mathbf{M}}$. Thus, $\mathfrak{M}$ is an answer set of $\mathbf{P}$. □

7.4 Discussion

The translations $\Lambda(\mathbf{P})$, $\Phi(\mathbf{P})$, and $\Omega(\mathbf{P})$ from §7.2, §7.3.1, and §7.3.2, respectively, allow us to express the existence of answer sets of an MLP $\mathbf{P}$ as a satisfiability problem in propositional logic, first-order and second-order predicate logic that is decidable. However, for arbitrary calls by value, the resulting formulas are huge in general, given that there are double exponential many value calls $P_i[S]$ for a list of input predicates $\mathbf{q}$ in general.

Loop formulas Loops in an MLP can be very long; in the general case, they can have double exponential length. Ordinary normal logic programs already require to use exponentially many loop formulas to recast programs to propositional formulas (Lifschitz and Razborov, 2006). However, the intrinsic complexity of MLPs already mentioned in §7.2 suggests that even in the propositional case (where the number of different inputs S is at most exponential) we cannot expect a polynomially computable transformation of brave inference $\mathbf{P} \models a$ into a propositional SAT instance, as the problem is EXP-complete for propositional Horn MLPs and NEXP-complete for propositional normal MLPs.

There are multiple sources of additional complexity in MLPs that require to create propositional formulas of double exponential size: first, we need to ground rules with variables, module input requires to inspect double exponential many module instances, and therefore loop formulas may have double exponential size.

Ben-Eliyahu and Dechter (1994) studied the class of head-cycle-free disjunctive logic programs and provide a characterization of answer sets by a polynomial-time translation to propositional theories. Reinterpreted into our setting, the complexity of MLPs without inputs restricted to this class should drop to that of head-cycle-free disjunctive logic programs using an appropriate translation. But allowing (relational) input to the modules will give us the additional blowup observed in this chapter.

Lin and Zhao (2004) developed an algorithm that starts with the completion of a logic program, tries to find a model, and then iteratively adds selected loop formulas to the completion such that not all—in the worst case exponentially many—loop formulas have to be added upfront before we arrive at an answer set. Their approach is based on finding loops such that the corresponding loop formula is not satisfied by the classical model of the current stage of the incremental loop formula translation. A similar technique could work in the MLP setting as well, which requires experiments to provide empirical evidence.

Another aspect would be to use first order loop formulas (Y. Chen et al., 2006; Lee and Meng, 2011), but as shown by Y. Chen et al. (2006) the resulting first-order theory may be infinite in size due to potential infinite number of loop formulas, thus a translation for MLP would require to handle infinite first-order theories as well.

Ordered completion Regarding ordered completion for MLPs, our results hold only for finite structures as ordered completion does not capture the answer set semantics of normal logic programs over arbitrary structures, i.e., considering infinite structures as well. Asuncion et al. (2012) showed this by using a normal logic program P encoding the transitive closure of a graph whose ordered completion $OC(P)$ over finite structures encodes the answer sets of P. But over infinite structures there is no (infinite) first-order theory ϕ_P such that the models of ϕ_P capture the answer sets of P using Fagin's Theorem (see, e.g., Libkin, 2004).

As Example 7.12 shows, the transitive derivation formula $\tau(\mathbf{P})$ can grow quickly if the number of value calls is large in an MLP $\mathbf{P}$. In practice, it could be that not all conjuncts of $\tau(\mathbf{P})$ are relevant and one might develop strategies to introduce them lazily on demand.

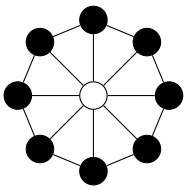
Extending ordered completion over finite structures to disjunctive logic programs was considered by Asuncion et al. (2012), but the result was negative: they have shown that based on a widely believed assumption from computational complexity theory that NP is not closed under complement (i.e., NP $\neq$ co-NP) implies that the answer set semantics for disjunctive logic programs cannot be expressed by sentences in first-order logic. This has also the immediate negative result that there is no translation into first-order predicate logic for disjunctive MLPs.

Further extensions employing aggregates in MLPs may use ordered completion with aggregates as a basis (Asuncion et al., 2015), which shows that normal logic programs with convex aggregates—covering monotone and anti-monotone aggregate functions—can be expressed by a first-order formula over convex aggregate contexts. But there is a negative result with respect to nonconvex aggregates similar to the negative result for expressing ordered completion for disjunctive logic programs (Asuncion et al., 2012). Translations such as the one presented by Alviano et al. (2015b) could prove to be useful to bring programs with convex aggregates into a simpler form that is amenable to standard ASP solvers.

In general, the ordered completion formula $\Omega(\mathbf{P})$ for a normal MLP $\mathbf{P}$, which can be seen as a Σ_1^1 formula—i.e., a formula in existential second order logic—, is thus evaluable in nondeterministic exponential time over finite structures (Vardi, 1982). Here, in the propositional case the input values S and T may be encoded using (polynomially many) predicate arguments (e.g., $o^T(\mathbf{y})$ becomes $o(\mathbf{x}, \mathbf{y})$, where $\mathbf{x} = x_1, \ldots, x_k$ encodes T) and disjunction/conjunction over S and T expressed by (first-order) quantification. In this way, it is possible to obtain a Σ_1^1 formula of polynomial size over a finite structure, such that this modified transformation is worst-case optimal with respect to the complexity of propositional normal MLPs. Similar encoding techniques can be applied for nonground MLPs if the predicate arities of formal input predicates are bounded by a constant.

In the general nonground case, such polynomial encoding techniques are not evident; already in the Horn case deciding $\mathbf{P} \models a$ is 2EXP-complete, and for normal MLPs brave inference is 2NEXP-complete. One may resort to predicate variables for encoding S and T, and naturally arrive at a formula in higher-order logic (e.g., $o^T(\mathbf{y})$ becomes $o(\mathbf{T}, \mathbf{y})$, where $\mathbf{T} = T_1, \ldots, T_k$ is a list of predicate variables for the formal input predicates $\mathbf{q} = q_1, \ldots, q_k$). It remains to be seen, however, whether the structure of the resulting (polynomial-size) formula would readily permit worst-case optimal evaluation with respect to the complexity of MLPs. The translational semantics $\Phi(\mathbf{P})$ may provide insights for a characterization.

Noticeably, however, we do not get a blowup if no call by value is made, i.e., if all inputs lists are empty (which means all S and T have the single value $\varnothing$). This setting is still useful for structured programming, and amounts in the propositional case to the DLP-functions of Janhunen et al. (2009b), and permits unlimited recursion through modules, in particular positive recursion. Our results thus also provide ordered completion formulas for DLP-functions over normal programs. See also Chapter 9 for the relationship between MLPs and DLP-functions.



8

Relevance-driven Evaluation of Modular Nonmonotonic Logic Programs

Based on results shown by Dao-Tran et al. (2009b), we will discuss in this chapter the essential ideas for an efficient top-down evaluation strategy for the fragment of input-call stratified MLPs.

In this chapter, we focus on reviewing efficient top-down evaluation techniques for MLPs, which consider only calls to relevant module instances. To this end, Dao-Tran et al. (2009b) generalize the well-known Splitting Theorem to the MLP setting and present notions of call stratification. Call-stratified MLPs allow to split module instantiations into two parts, one for computing input of module calls, and one for evaluating the calls themselves with subsequent computations. Based on these results, Dao-Tran et al. (2009b) have developed a top-down evaluation procedure that expands only relevant module instantiations.

As the semantics of MLPs is based on module instantiations (which takes possible input values into account), a naive evaluation following the definition is—similar to grounding of ordinary ASP programs—infeasible in practice. In general, a particular module may have double exponentially many instances (see Chapter 5). Towards implementation, efficient evaluation strategies are thus essential, which are sensitive to program classes that do not require a simple guess-and-check procedure on the instantiation, but allow for a guided model building process. Starting from the main module, instances of modules may be created on demand as needed by module calls, focusing on relevant module instances.

Restrictions on programs, like stratification for normal MLPs as shown in §4.3, may be helpful in this regard. However, the notion of stratification is very strict. It requires that *all* module instances are stratified. Moreover, the fix-point semantics for stratified

programs given there is inherently bottom-up and only applies to normal programs, excluding a large class of programs that exploit recursion in a common and natural way and are evaluable top-down, even if they are not normal or unstratified in the sense of Definition 4.10.

For illustration, let us reconsider the Even MLP $\mathbf{P} = (m_1, m_2, m_3)$ from Example 1.2 in §1.2.1. This example exploits a mutually recursive call pattern between two library modules m_2 and m_3 to determine whether a set has even cardinality. Intuitively, m_1 calls m_2 to check if the number of facts for predicate q is even. The call to m_2 "returns" *even*, if either the input q_2 to m_2 is empty (as then $skip_2$ is false), or the call of m_3 with q_2' resulting from q_2 by arbitrarily removing one element (then $skip_2$ is true) returns *odd*. Module m_3 returns *odd* for input q_3, if a call to R_2 with q_3' analogously constructed from q_3 returns *even*.

According to Definition 4.10, this program is a normal unstratified program, since *even* negatively depends on *odd*, which in turn depends on *even* through the module call $P_2[q_3'].even$.

However, along the mutual recursive chain of calls $P_3[q_2'].odd$, $P_2[q_3'].even$ the inputs q_2' and q_3' gradually decrease until the base, i.e., the empty input, is reached. Taking such decreasing inputs of the relevant module calls into account, we can evaluate MLPs efficiently along the relevant call graph using a finer grained notion of stratification, tolerating also disjunctive or unstratified rules in modules.

We briefly report on appropriate notions of *call stratification* and *input stratification* for MLPs as a generalization of the Splitting Theorem (see §8.1 and Lifschitz and Turner, 1994). Based on this, module instances calling other modules can be locally split into an input preparation part and a calling part. A top-down evaluation procedure exploiting this local splitting will be reviewed, which only expands the relevant module instances in §8.2. Wijaya (2011) has implemented the TD-MLP system for evaluating input-call stratified MLPs, and based on this system we have developed a benchmark based on the LUBM ontology (Guo et al., 2005). We report experimental results in §8.3 for the dl-programs rewriting techniques shown in §6.5 above.

8.1 Splitting for Modular Nonmonotonic Logic Programs

We investigate splitting for MLPs at two different levels: the global (module instantiation) level along the relevant call graph, and the local level (inside module instantiations) with respect to the (instance) dependency graph. These two notions reveal a class of MLPs, for which an efficient top-down algorithm can be developed for answer set computation.

8.1.1 Global splitting for call-stratified MLPs

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

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

Example 8.1 Recalling the program from Example 1.2, let $S^i_\emptyset = \emptyset$, $S^i_a = \{q_i(a)\}$, $S^i_b = \{q_i(b)\}$, and $S^i_{ab} = \{q_i(a), q_i(b)\}$. Then $\mathrm{VC}(\mathbf{P}) = \{P_1[\emptyset], P_2[S^2_v], P_3[S^3_w]\}$, where $v, w \in \{\emptyset, a, b, ab\}$, and $CG_\mathbf{P}$ has edges $P_1[\emptyset] \xrightarrow{q} P_2[S^2_v]$, $P_2[S^2_v] \xrightarrow{q'_2} P_3[S^3_w]$, and $P_3[S^3_w] \xrightarrow{q'_3} P_2[S^2_v]$. For the interpretation $\mathbf{M}$ such that

- $M_1/\emptyset = \{q(a), q(b), ok\}$,
- $M_2/S^2_{ab} = \{q_2(a), q_2(b), q'_2(a), skip_2, even\}$,
- $M_2/\emptyset = \{even\}$, and
- $M_3/S^3_a = \{q_3(a), skip_3, odd\}$,

the nodes of $CG_\mathbf{P}(\mathbf{M})$ are $P_1[\emptyset]$, $P_2[S^2_{ab}]$, $P_2[\emptyset]$, and $P_3[S^3_a]$.

Example 8.2 Consider the MLP $\mathbf{P}$ and the interpretation $\mathbf{M}$ from Example 8.1. It is easily verified that $\mathbf{P}$ is c-stratified with respect to $\mathbf{M}$. One possible call chain for evaluation is

$$P_1[\emptyset] \xrightarrow{q} P_2[\{q(a), q(b)\}] \xrightarrow{q'_2} P_3[\{q'_2(a)\}] \xrightarrow{q'_3} P_2[\emptyset] \ .$$

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

Example 8.3 Consider $\mathbf{P}$ from Example 1.2 and $\mathbf{M}$ from Example 8.1, then M_2/S^2_{ab} is an answer set of $I_\mathbf{P}(P_2[S^2_{ab}])$ relative to $\mathbf{M}$.

For a concrete procedure, we need a notion of local splitting inside module instances, to be introduced in the next section.

8.1.2 Local splitting for input and call stratified MLPs

Dao-Tran et al. (2009b) extended the notion of Splitting Sets (Lifschitz and Turner, 1994) to MLPs. For practical purposes, we are interested in splitting a module instance with respect to module calls. To this end, Dao-Tran et al. (2009b) introduce a general and another specific notion of *input splitting sets*.

Example 8.4 Consider **P** from Example 1.2 and $P_2[S^2_{ab}]$ from Example 8.1. Let R be the instantiation $gr(I_{\mathbf{P}}(P_2[S^2_{ab}]))$. A possible splitting set for R is

$$U = \{q_2(a), q_2(b), q'_2(a), q'_2(b), skip_2\} \ .$$

Then the bottom $b_U(R)$ is

$$\begin{aligned} q_2(\alpha) &\leftarrow \\ skip_2 &\leftarrow q_2(\alpha), \text{not } q'_2(\alpha) \\ q'_2(\alpha) &\leftarrow \text{not } q'_2(\beta), q_2(\alpha), q_2(\beta) \end{aligned}$$

where $\alpha, \beta \in \{a, b\}$ such that $\alpha \neq \beta$.

Example 8.5 Consider **P** from Example 1.2, **M** from Example 8.1, and R from Example 8.4. An answer set of R is $N = \{q_2(a), q_2(b), q'_2(a), skip_2\}$. By updating R to $\{R \setminus b_U(R)\} \cup N$, we obtain R':

$$\begin{aligned} q'_2(a) &\leftarrow \\ q_2(a) &\leftarrow \\ q_2(b) &\leftarrow \\ skip_2 &\leftarrow \\ even &\leftarrow skip_2, P_3[q'_2].odd \\ even &\leftarrow \text{not } skip_2 \end{aligned}$$

Then M_2/S^2_{ab} is an answer set of R' relative to **M**.

Dao-Tran et al. (2009b) single out a subclass of c-stratified MLPs, namely *input and call stratified (ic-stratified)* MLPs, which guarantee that input splitting sets exist for their local splitting. The property of *input stratification* is defined at two different levels of the dependency graph: the schematic level and the instance level. Comparing these two options, checking the property at the schematic level is easier, but is often too strong and misses input stratification at the instance level.

This notion of dependency graph refines the one in §4.3 concerning the labels of arcs (types of dependencies) and allows us to capture input stratification. As an example, the MLP in Example 1.2 is si-stratified.

8.1.3 Instance stratification

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

Example 8.6 In Example 8.4, U is an input splitting set for $P_3[S^3_a].odd$. As $\mathbf{P}$ is c-stratified with respect to $\mathbf{M}$ (cf. Example 8.2) and si-stratified, $\mathbf{P}$ is ic-stratified with respect to $\mathbf{M}$.

Since ic-stratification (of an MLP $\mathbf{P}$ with respect to $\mathbf{M}$) ensures that no cycle in $G^{\mathbf{M}}_{\mathbf{P}}$ has in-edges, it yields intended local splits, where the input for any module atom is *fully prepared* before this module atom is called.

8.2 Top-Down Evaluation Algorithm

Dao-Tran et al. (2009b) present the *comp* algorithm, which is a top-down evaluation procedure *comp* for building the answer sets of ic-stratified MLPs along the call graph. Intuitively, *comp* traverses the relevant call graph from top to the base and back. In forward direction, it gradually prepares input to each module call in a set R of rules, in the order given by the instance local labeling function for R. When all calls are solved, R is rewritable to a set of ordinary rules, and standard methods can be used to find the answer sets, which are fed back to a calling instance, or returned as the result if we are at the top level.

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

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

If R is ordinary, meaning that all module atoms (if any) are solved, *ans* can be applied to find answer sets of R. Now, if *path* is empty, then a main module is reached and $\mathbf{M}$ can be completed by the answer sets of R and put into $\mathcal{AS}$ Otherwise, i.e., *path* is nonempty, *comp* marks all instances in C as finished, and goes back to the tail of *path* where a call to C was issued. In both cases, the algorithm uses an operator $\uplus$ for

combining two partial interpretations as follows: $\mathbf{M} \uplus \mathbf{N} = \{M_i/S \uplus N_i/S \mid P_i[S] \in VC(\mathbf{P})\}$, where $x \uplus y = x \cup y$ if $x, y \neq nil$ and $x \uplus nil = x$, $nil \uplus x = x$.

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

Example 8.7 Consider Algorithm *comp* on $\mathbf{P}$ from Examples 1.2 and 8.1. The call chain

$$P_1[\varnothing] \xrightarrow{q} P_2[\{q(a), q(b)\}] \xrightarrow{q'_2} P_3[\{q'_2(a)\}] \xrightarrow{q'_3} P_2[\varnothing] \xrightarrow{q'_2} P_3[\varnothing] \xrightarrow{q'_3} P_2[\varnothing]$$

will be reflected by the list $\{P_1[S^1_\varnothing]\}, \{P_2[S^2_{a,b}\}, \{P_3[S^3_a]\}, \{P_2[S^2_\varnothing]\}, \{P_3[S^3_\varnothing]\}$ in *path*, and a current set of value calls $C = \{P_2[S^2_\varnothing]\}$. Here, rather than prefixes, we use superscripts and subscripts like for instances (cf. Example 8.1). At this point, the last two elements of the path will be removed and joined with C yielding $C = \{P_2[S^2_\varnothing], P_3[S^3_\varnothing]\}$. The rewriting R with respect to C is

$$\begin{aligned}
{q'}^1_\varnothing(X) \vee {q'}^1_\varnothing(Y) &\leftarrow q^1_\varnothing(X), q^1_\varnothing(Y), X \neq Y \\
skip^1_\varnothing &\leftarrow q^1_\varnothing(X), \text{not } {q'}^1_\varnothing(X) \\
{q'}^2_\varnothing(X) \vee {q'}^2_\varnothing(Y) &\leftarrow q^2_\varnothing(X), q^2_\varnothing(Y), X \neq Y \\
skip^2_\varnothing &\leftarrow q^2_\varnothing(X), \text{not } {q'}^2_\varnothing(X) \\
even^2_\varnothing &\leftarrow skip^2_\varnothing, odd^3_\varnothing \\
odd^3_\varnothing &\leftarrow skip^3_\varnothing, even^2_\varnothing \\
even^2_\varnothing &\leftarrow \text{not } skip^2_\varnothing
\end{aligned}$$

The only answer set of R is $\{even^2_\varnothing\}$. On the way back, $even^2_v$ is toggled with odd^3_w, and at P_1 the answer set $\{q^1_\varnothing(a), q^1_\varnothing(b), ok^1_\varnothing\}$ is built; *comp* adds a respective (partial) interpretation $\mathbf{M}$ to $\mathcal{AS}$, i.e., where $M_2/\varnothing = \{even\}$, $M_3/\varnothing = \varnothing$, etc., and $M_1/\varnothing = \{q(a), q(b), ok\}$. Following the chain

$$P_1[\varnothing] \xrightarrow{q} P_2[\{q(a), q(b)\}] \xrightarrow{q'_2} P_3[\{q'_2(b)\}] \xrightarrow{q'_3} P_2[\varnothing] \to \cdots,$$

comp finds another answer set of $\mathbf{P}$.

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

Table 8.1: Benchmark dl-programs DReW vs. TD-MLP (Runtime in secs)

(a) Ontology U_1

Program	DReW		TD-MLP	
	[clingo]	[DLV]	[clingo]	[DLV]
P_1	0.31	0.45	1.98	2.88
P_2	0.32	0.44	1.69	2.47
P_3	0.32	0.44	2.63	3.82
P_4	0.31	0.43	1.66	2.42
P_5	0.32	0.45	2.45	3.63
P_6	0.61	0.86	1.66	2.46
P_7	1.79	2.76	5.65	8.41
P_8	2.70	4.30	8.04	11.60
P_9	2.76	4.26	9.70	14.12

(b) Ontology U_{15}

Program	DReW		TD-MLP	
	[clingo]	[DLV]	[clingo]	[DLV]
P_1	6.49	10.27	30.43	42.53
P_2	4.00	6.27	21.22	30.12
P_3	3.95	6.08	32.65	45.24
P_4	3.98	6.13	20.94	30.33
P_5	4.15	6.43	28.19	39.93
P_6	7.97	12.66	21.54	30.87
P_7	23.52	40.56	72.86	103.76
P_8	36.33	64.05	108.38	145.04
P_9	36.58	61.71	128.01	181.41

8.3 Implementation and Experimental Results

In this section, we review experimental results for an MLP benchmark scenario (see Eiter et al., 2012b, for the report including further benchmarks scenarios unrelated to MLPs). The algorithm for solving input-call stratified MLPs (Dao-Tran et al., 2009b) has been implemented in the TD-MLP solver (Wijaya, 2011), which is based on the DLVHEX system (Eiter et al., 2018, 2017).[1] Using this solver, we could perform initial experiments with the dl-program rewriting, which is the modular Datalog encoding $\Delta(KB)$ from §6.5. We compare our MLP encoding with DReW (Xiao et al., 2013), a sys-

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

Table 8.2: Benchmark programs P_1–P_5

Program	Rules
P_1	$q(X,Y) \leftarrow \mathrm{DL}[Faculty](X), \mathrm{DL}[Faculty](Y),$ $\mathrm{DL}[doctoralDegreeFrom](X,U_1), \mathrm{DL}[worksFor](X,D_1),$ $\mathrm{DL}[doctoralDegreeFrom](Y,U_2), \mathrm{DL}[worksFor](Y,D_2),$ $U_1 \neq U_2, D_1 = D_2$
P_2	$q(X,Y) \leftarrow \mathrm{DL}[GraduateStudent](X),$ $\mathrm{DL}[takesCourse](X,Y), Y = graduateCourse0$
P_3	$q(X,Y,Z) \leftarrow \mathrm{DL}[GraduateStudent](X), \mathrm{DL}[University](Y),$ $\mathrm{DL}[Department](Z), \mathrm{DL}[memberof](X,Z),$ $\mathrm{DL}[subOrganizationOf](Z,Y),$ $\mathrm{DL}[undergraduateDegreeFrom](X,Y)$
P_4	$q(X,Y) \leftarrow \mathrm{DL}[Publication](X),$ $\mathrm{DL}[publicationAuthor](X,Y), Y = assistantProfessor0$
P_5	$q(X,Y_1,Y_2,Y_2) \leftarrow \mathrm{DL}[Professor](X), \mathrm{DL}[worksFor](X,Z),$ $Z = department0@University0,$ $\mathrm{DL}[name](X,Y_1), \mathrm{DL}[emailAddress](X,Y_2),$ $\mathrm{DL}[telephone](X,Y_3)$

tem specifically tailored to evaluate dl-programs over Datalog-rewritable Description Logics using Datalog rewriting techniques.

The experiments have been run on an Ubuntu Linux 11.10 system on an AMD Opteron Magny-Cours 6176 SE 2.3GHz system with 24 cores and 128GB RAM. Further details are given on the benchmark webpage,[2] with detailed implementation information, all benchmark instances and benchmark details, as well as test run log files.[3]

For our experiments, we used dl-programs of the form $KB_{i,j} = (U_i, P_j)$, where U_i ($i \in \{1, 15\}$) are simplified $\mathcal{EL}$ versions of the Lehigh University Benchmark (LUBM) ontology (Guo et al., 2005) and programs P_j ($j \in \{1, \ldots, 9\}$) encode variants of the LUBM queries;[4] all dl-programs were acyclic.

We denote with U_i the LUBM ontology instance that incorporates i universities in the ABox. The original LUBM is not fully in $\mathcal{EL}$ (inverse roles and data types are not part of $\mathcal{EL}$): there are 2 violating axioms in the TBox, and 2857 (respectively, 33154) ABox axioms with data types are violated in U_1 (respectively, U_{15}). The resulting $\mathcal{EL}$

[2] http://www.kr.tuwien.ac.at/research/systems/drew/experiments.html

[3] http://www.kr.tuwien.ac.at/research/systems/drew/downloads/foiks2012-elp+mlp.tar.7z

[4] http://swat.cse.lehigh.edu/projects/lubm/query.htm

version of LUBM then contains 86 TBox axioms using 43 concepts and 25 roles, and 5738 (respectively, 67691) ABox axioms with 1555 (respectively, 17174) individuals in instance U_1 (respectively, U_{15}). The rules are from the DReW LUBM benchmark queries and consist of nine programs P_1–P_9. They can be split into two categories:

(C1) P_1–P_5 have between 2 and 5 dl-atoms but no input list, while

(C2) P_6–P_9 have between 2 and 9 dl-atoms, each with distinct input list.

The rules for the benchmark programs P_1–P_5 and P_6–P_9 are shown in Table 8.2 and Tables 8.3–8.5, respectively.

We used two Datalog engines to compute the models of the native and the modular encodings: clingo 3.0.3 (Gebser et al., 2011) and DLV 2010-10-14 (Leone et al., 2006). We compared TD-MLP to DReW using four benchmark run settings (the systems in square brackets denote the model builders used to calculate the model):

- DReW[clingo],
- DReW[DLV],
- TD-MLP[clingo], and
- TD-MLP[DLV].

The test results are shown in Table 8.1a for $KB_{1,j} = (U_1, P_j)$ and in Table 8.1b for $KB_{15,j} = (U_{15}, P_j)$.

DReW outperforms TD-MLP in all tests. But DReW's lead is shrinking if we increase the number of dl-atoms in the dl-programs from category (C2), i.e., for dl-programs (U_i, P_6)–(U_i, P_9). The reason is that with DReW we create copies of the ontology as Datalog rules for every dl-atom upfront, thus creating a large single Datalog program. In TD-MLP, we can always use a single copy of the rewritten ontology and let the MLP semantics create the copies for us. As the current TD-MLP implementation is not sophisticated enough, the overhead for instantiating modules during evaluation is prevalent.

Table 8.3: Benchmark programs P_6–P_7

Program	Rules
P_6	$pub(a) \leftarrow$
	$pubAuth(a, assistProf0) \leftarrow$
	$q(X,Y) \leftarrow \mathrm{DL}[Publication \uplus pub; Publication](X),$
	$\mathrm{DL}[publicationAuthor \uplus pubAuth; publicationAuthor](X,Y),$
	$Y = assistProf0$
P_7	$gradStud(a) \leftarrow$
	$uni(b) \leftarrow$
	$dept(c) \leftarrow$
	$memb(a,c) \leftarrow$
	$subOrga(c,b) \leftarrow$
	$ugd(a,b) \leftarrow$
	$q(X,Y,Z) \leftarrow \mathrm{DL}[GraduateStudent \uplus gradStud; GraduateStudent](X),$
	$\mathrm{DL}[University \uplus uni; University](Y),$
	$\mathrm{DL}[Department \uplus dept; Department](Z),$
	$\mathrm{DL}[memberOf \uplus memb; memberOf](X,Z),$
	$\mathrm{DL}[subOrganizationOf \uplus subOrga; subOrganizationOf](Z,Y),$
	$\mathrm{DL}[underGraduateDegreeFrom \uplus ugd; underGraduateDegreeFrom](X,Y)$

Table 8.4: Benchmark program P_8

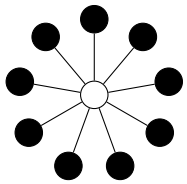
Program	Rules
P_8	$gradStud(a) \leftarrow$
	$uni(b) \leftarrow$
	$memb(a, c) \leftarrow$
	$subOrga(c, b) \leftarrow$
	$ugd(a, b) \leftarrow$
	$q(X, Y, Z) \leftarrow \mathrm{DL}[GraduateStudent \uplus gradStud; GraduateStudent](X),$
	$\mathrm{DL}[University \uplus uni; University](Y),$
	$\text{not}\ \mathrm{DL}[Department \uplus dept; Department](Z),$
	$\mathrm{DL}[memberOf \uplus memb; memberOf](X, Z),$
	$\mathrm{DL}[subOrganizationOf \uplus subOrga; subOrganizationOf](Z, Y),$
	$\mathrm{DL}[underGraduateDegreeFrom \uplus ugd; underGraduateDegreeFrom](X, Y)$
	$q(X, Y, Z) \leftarrow \mathrm{DL}[GraduateStudent \uplus gradStud; GraduateStudent](X),$
	$\mathrm{DL}[University \uplus uni, Department \uplus dept; University](Y),$
	$\mathrm{DL}[Department \uplus dept; Department](Z),$
	$\text{not}\ \mathrm{DL}[memberOf \uplus memb; memberOf](X, Z),$
	$\mathrm{DL}[subOrganizationOf \uplus subOrga; subOrganizationOf](Z, Y),$
	$\mathrm{DL}[underGraduateDegreeFrom \uplus ugd, Department \uplus dept; underGraduateDegreeFrom](X, Y)$
	$q(X, Y, Z) \leftarrow \mathrm{DL}[GraduateStudent \uplus gradStud; GraduateStudent](X),$
	$\mathrm{DL}[University \uplus uni, Department \uplus dept; University](Y),$
	$\mathrm{DL}[Department \uplus dept; Department](Z),$
	$\text{not}\ \mathrm{DL}[memberOf \uplus memb; memberOf](X, Z),$
	$\mathrm{DL}[GraduateStudent \uplus gradStud, subOrganizationOf \uplus subOrga; subOrganizationOf](Z, Y),$
	$\mathrm{DL}[underGraduateDegreeFrom \uplus ugd, Department \uplus dept; underGraduateDegreeFrom](X, Y)$

Table 8.5: Benchmark program P_9

Program	Rules
P_9	$gradStud(a) \leftarrow$
	$uni(b) \leftarrow$
	$dept(c) \leftarrow$
	$memb(a, c) \leftarrow$
	$subOrga(c, b) \leftarrow$
	$ugd(a, b) \leftarrow$
	$q(X, Y, Z) \leftarrow \mathrm{DL}[GraduateStudent \uplus gradStud; GraduateStudent](X),$
	$\mathrm{DL}[University \uplus uni; University](Y),$
	$\mathrm{DL}[Department \uplus dept; Department](Z),$
	$\mathrm{DL}[memberOf \uplus memb; memberOf](X, Z),$
	$\mathrm{DL}[subOrganizationOf \uplus subOrga; subOrganizationOf](Z, Y),$
	$\mathrm{DL}[underGraduateDegreeFrom \uplus ugd; underGraduateDegreeFrom](X, Y)$
	$q(X, Y, Z) \leftarrow \mathrm{DL}[GraduateStudent \uplus gradStud; GraduateStudent](X),$
	$\mathrm{DL}[University \uplus uni, Department \uplus dept; University](Y),$
	$\mathrm{DL}[Department \uplus dept; Department](Z),$
	$\mathrm{DL}[memberOf \uplus memb; memberOf](X, Z),$
	$\mathrm{DL}[subOrganizationOf \uplus subOrga; subOrganizationOf](Z, Y),$
	$\mathrm{DL}[underGraduateDegreeFrom \uplus ugd, Department \uplus dept; underGraduateDegreeFrom](X, Y)$
	$q(X, Y, Z) \leftarrow \mathrm{DL}[GraduateStudent \uplus gradStud; GraduateStudent](X),$
	$\mathrm{DL}[University \uplus uni, Department \uplus dept; University](Y),$
	$\mathrm{DL}[Department \uplus dept; Department](Z),$
	$\mathrm{DL}[memberOf \uplus memb; memberOf](X, Z),$
	$\mathrm{DL}[GraduateStudent \uplus gradStud, subOrganizationOf \uplus subOrga; subOrganizationOf](Z, Y),$
	$\mathrm{DL}[underGraduateDegreeFrom \uplus ugd, Department \uplus dept; underGraduateDegreeFrom](X, Y)$

IV

Related Approaches and Conclusion



9

Relationship to DLP-Functions

This chapter examines the relationship between Modular Nonmonotonic Logic Programs and DLP-functions (Janhunen et al., 2009b), a prominent formalism for modular logic programs under the stable model semantics that conforms to the Programming-in-the-large paradigm. Here, a modular program is a sequence of independent modules with well-defined input-output interface of propositional atoms. Whenever each pair of distinct modules in this sequence satisfies syntactic dependency criterions that make them able to combine them, a join operator is defined and the stable models of the combined modules agree with the stable models of each individual module.

More specifically, a DLP-function Π is a tuple $\langle R, I, O, H\rangle$, where R is a set of propositional disjunctive rules and I, O, H are sets of propositional atoms defining input, output, and hidden atoms, respectively. The operator $\oplus$ sends a pair of DLP-functions to a new DLP-function that respect hidden atoms of each input DLP-function. Then, if two such DLP-functions Π_1 and Π_2 are not mutually (positive) dependent, then their join $\Pi_1 \sqcup \Pi_2$ is defined and $\Pi_1 \oplus \Pi_2$ is the outcome. Note that joinability allows negative loops between DLP-functions, but prohibits positive ones. On top of joinable DLP-functions, the Module Theorem is at the heart for computing the answer sets of a sequence of DLP-functions by taking the union of mutually compatible answer sets of each member; hence joinable DLP-functions qualify for having a compositional semantics. The Module Theorem is fruitfully applied to build incremental computations in Clingo (Gebser et al., 2017), an answer set solver that allows to deal with continuously changing logic programs.

In this chapter, we will first review the syntax and semantics of DLP-functions, and the Module Theorem in §9.1. Then, we will investigate on the relationship between DLP-functions and MLPs by defining two translations: ∇ for rewriting a sequence of DLP-functions to an MLP in §9.2, and Δ for rewriting an MLP to a sequence of DLP-functions in §9.3, provided that the MLP adheres to syntactic restrictions that match the joinability conditions of standard DLP-functions. As results we obtain that we

can capture the stable models of joined DLP-functions using an MLP, and that there is a one-to-one correspondence between the stable models of DLP-functions and the answer sets of a restricted class of MLPs.

9.1 DLP-Functions

We will now recapitulate the basic definitions for DLP-functions from Janhunen et al. (2009b). We start with the syntax of DLP-functions, which is based on propositional disjunctive logic programs.

9.1.1 Syntax of DLP-Functions

We begin with defining DLP-functions, which are entities to define modules in answer set programs.

Definition 9.1 (DLP-function).
A *DLP-function* is a quadruple $\Pi = \langle R, I, O, H \rangle$, where I, O, and H are pairwise distinct sets of *input atoms*, *output atoms*, and *hidden atoms*, respectively, and R is a disjunctive logic program such that for each rule $r \in R$ of form (2.1),

1. $H(r) = \{a_1, \ldots, a_k\}$ and $B(r) = B^+(r) \cup B^-(r) = \{b_1, \ldots, b_n\}$ are positive atoms,
2. $H(r) \cup B(r) \subseteq I \cup O \cup H$, and
3. if $H(r) \neq \emptyset$, then $H(r) \cap (O \cup H) \neq \emptyset$.

Based on DLP-functions, we can define when two DLP-functions can be composed into a combined DLP-function. The key concept for this is to respect input/output interfaces.

For the following definitions, we let $\Pi = \langle R, I, O, H \rangle$, $\Pi_1 = \langle R_1, I_1, O_1, H_1 \rangle$, and $\Pi_2 = \langle R_2, I_2, O_2, H_2 \rangle$ be DLP-functions.

Definition 9.2 (Input/output interfaces).
Let S be a set of atoms and R be a disjunctive logic program, we define

$$\mathrm{Def}_R(S) = \{r \in R \mid H(r) \cap S \neq \emptyset\} \quad .$$

Two DLP-functions Π_1 and Π_2 *respect the input/output interfaces of each other* if and only if

1. $(I_1 \cup O_1 \cup H_1) \cap H_2 = \emptyset$,
2. $(I_2 \cup O_2 \cup H_2) \cap H_1 = \emptyset$,

3. $O_1 \cap O_2 = \emptyset$,
4. $\mathrm{Def}_{R_1}(O_1) = \mathrm{Def}_{R_1 \cup R_2}(O_1)$, and
5. $\mathrm{Def}_{R_2}(O_2) = \mathrm{Def}_{R_1 \cup R_2}(O_2)$.

Definition 9.3 (Composition).
Let Π_1 and Π_2 be two DLP-functions that respect the input/output interfaces of each other. Then, the *composition of* Π_1 *and* Π_2 is defined and determined by

$$\Pi_1 \oplus \Pi_2 = \langle R_1 \cup R_2, (I_1 \setminus O_2) \cup (I_2 \setminus O_1), O_1 \cup O_2, H_1 \cup H_2 \rangle \ .$$

The composition of two DLP-functions does not rule out the possibility of mutually dependent DLP-functions. We therefore define a join operator that operates on a restricted fragment of DLP-functions, which requires the following definition.

Definition 9.4 (Signature).
We define the *signature* $\mathrm{At}(\Pi)$ of Π as $I \cup O \cup H$. Relative to signature $\mathrm{At}(\Pi)$, we let

- $\mathrm{At}_\mathrm{v}(\Pi) = I \cup O$ be the *visible* part,
- $\mathrm{At}_\mathrm{h}(\Pi) = H = \mathrm{At}(\Pi) \setminus \mathrm{At}_\mathrm{v}(\Pi)$ be the *hidden* part,
- $\mathrm{At}_\mathrm{i}(\Pi) = I$ are the input atoms of Π, and
- $\mathrm{At}_\mathrm{o}(\Pi) = O$ are the output atoms of Π, respectively.

For any set $S \subseteq \mathrm{At}(\Pi)$ of atoms, we denote the projections of S on $\mathrm{At}_\mathrm{i}(\Pi)$, $\mathrm{At}_\mathrm{o}(\Pi)$, $\mathrm{At}_\mathrm{v}(\Pi)$, and $\mathrm{At}_\mathrm{h}(\Pi)$ by

- $\mathrm{At}_\mathrm{i}(\Pi, S) = S \cap I$,
- $\mathrm{At}_\mathrm{o}(\Pi, S) = S \cap O$,
- $\mathrm{At}_\mathrm{v}(\Pi, S) = S \cap (I \cup O)$, and
- $\mathrm{At}_\mathrm{h}(\Pi, S) = S \cap H$, respectively.

Definition 9.5 (Dependency graph).
For a DLP-function Π, we define $DG^+(\Pi) = \langle O \cup H, \leq_1 \rangle$ as the positive dependency graph of Π, where $b \leq_1 a$ holds for a pair of atoms $a, b \in O \cup H$ if and only if there is a rule $r \in R$ such that $a \in H(r)$ and $b \in B^+(r)$. The reflexive and transitive closure of $\leq_1$ gives rise to the dependency relation $\leq$ over $O \cup H$.

A strongly connected component (SCC) C of the graph $DG^+(\Pi)$ is a maximal set $C \subseteq O \cup H$ such that $b \leq a$ for every pair $a, b \in C$ of atoms.

Given that $\Pi_1 \oplus \Pi_2$ is defined, we say that Π_1 and Π_2 are *mutually dependent* if and only if $DG^+(\Pi_1 \oplus \Pi_2)$ has an SCC C such that $C \cap O_1 \neq \emptyset$ and $C \cap O_2 \neq \emptyset$, i.e., the component C is shared by the DLP-functions Π_1 and Π_2 in this way. If Π_1 and Π_2 are not mutually dependent, we also call them *mutually independent*.

Mutually independent DLP-functions can be *joined*.

Definition 9.6 (Joins).
If the composition $\Pi_1 \oplus \Pi_2$ of DLP-functions Π_1 and Π_2 is defined and Π_1 and Π_2 are mutually independent, then the *join* $\Pi_1 \sqcup \Pi_2$ *of* Π_1 *and* Π_2 is defined and we let $\Pi_1 \sqcup \Pi_2$ to be determined by the composition $\Pi_1 \oplus \Pi_2$.

9.1.2 Semantics of DLP-Functions

We define now the semantics of DLP-functions. For the following definitions, we let $\Pi = \langle R, I, O, H \rangle$.

Definition 9.7 (Models).
An interpretation $M \subseteq \mathrm{At}(\Pi)$ is a (classical) model of Π, denoted $M \models \Pi$, iff $M \models R$, i.e., for every rule $r \in R$, $B^+(r) \subseteq M$ and $B^-(r) \cap M = \emptyset$ imply $M \cap H(r) \neq \emptyset$.

We can now define instantiations of DLP-functions relative to a subset of input I.

Definition 9.8 (Instantiation).
For an actual input $J \subseteq I$ for Π such that I are the input atoms of Π, the instantiation of Π with respect to J, denoted by Π/J, is the quadruple $\langle R', \emptyset, O, H \rangle$, where R' is the set of rules

$$(H(r) \setminus I) \leftarrow (B^+(r) \setminus I), \mathrm{not}(B^-(r) \setminus I)$$

for each $r \in R$ such that

- $J \cap \mathrm{At_i}(\Pi, H(r)) = \emptyset$,
- $\mathrm{At_i}(\Pi, B^+(r)) \subseteq J$, and
- $J \cap \mathrm{At_i}(\Pi, B^-(r)) = \emptyset$.

Minimal models are defined with respect to input I.

Definition 9.9 (Minimal models).
A model $M \subseteq \mathrm{At}(\Pi)$ of Π is I-minimal if and only if there is no model N of Π such that $\mathrm{At_i}(\Pi, N) = \mathrm{At_i}(\Pi, M)$ and $N \subset M$. The set of I-minimal models of Π is denoted by $\mathrm{MM}(\Pi)$.

The reduct of DLP-functions is defined akin to the GL-reduct.

Definition 9.10 (Reduct).
Given an interpretation $M \subseteq \mathrm{At}(\Pi)$ for Π, the reduct of Π with respect to M is the positive DLP-function $\Pi^M = \langle R^M, I, O, H \rangle$, where

$$R^M = \{H(r) \leftarrow B^+(r) \mid r \in R \wedge M \models B^+(r) \wedge M \not\models B^-(r)\} \quad .$$

The stable models of a DLP-function Π gives us the semantics of Π.

Definition 9.11 (Stable models).
An interpretation $M \subseteq \mathrm{At}(\Pi)$ is a stable model of a DLP-function Π with an input signature $\mathrm{At_i}(\Pi)$ iff $M \in \mathrm{MM}\left(\Pi^M\right)$, i.e., M is an $\mathrm{At_i}(\Pi)$-minimal model of Π^M. We let

$$\mathrm{SM}(\Pi) = \left\{M \subseteq \mathrm{At}(\Pi) \mid M \in \mathrm{MM}\left(\Pi^M\right)\right\}$$

denote the set of stable models of Π.

9.1.3 Module Theorem

Next, we will define mutually compatible interpretations and the natural join of mutually compatible interpretations, which form the basis for the Module Theorem.

Definition 9.12 (Mutually compatible interpretations).
Let Π_1 and Π_2 be DLP-functions, we say that the interpretations $M_1 \subseteq \mathrm{At}(\Pi_1)$ and $M_2 \subseteq \mathrm{At}(\Pi_2)$ are *mutually compatible with respect to* Π_1 *and* Π_2, denoted $M_1 \triangleq_{\mathrm{v}} M_2$, if $M_1 \cap \mathrm{At_v}(\Pi_2) = M_2 \cap \mathrm{At_v}(\Pi_1)$.

Definition 9.13 (Natural join).
Let Π_1 and Π_2 be two DLP-functions such that the join $\Pi_1 \sqcup \Pi_2$ is defined. Given any sets of interpretations $\mathcal{A}_1 \subseteq 2^{\mathrm{At}(\Pi_1)}$ and $\mathcal{A}_2 \subseteq 2^{\mathrm{At}(\Pi_2)}$, we define the set of interpretations

$$\mathcal{A}_1 \bowtie \mathcal{A}_2 = \left\{M_1 \cup M_2 \mid (M_1, M_2) \in \mathcal{A}_1 \times \mathcal{A}_2 \text{ such that } M_1 \triangleq_{\mathrm{v}} M_2\right\}$$

as the *natural join of* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *with respect to* $\mathrm{At_v}(\Pi_1) \cap \mathrm{At_v}(\Pi_2)$.

Janhunen et al. (2009b) have shown the Module Theorem, which will be subsequently used in the proofs for the translations in §9.2 and §9.3.

Module Theorem (Janhunen et al., 2009b)If Π_1 and Π_2 are DLP-functions such that the join $\Pi_1 \sqcup \Pi_2$ is defined, then $\mathrm{SM}(\Pi_1 \sqcup \Pi_2) = \mathrm{SM}(\Pi_1) \bowtie \mathrm{SM}(\Pi_2)$.

In the rest of this chapter, we will define two translations: one for turning DLP-functions into MLP modules, and another translation that is defined on a fragment of MLPs without input and generates an equivalent sequence of DLP-functions. To be in line with (Janhunen et al., 2009b), we consider only the propositional case.

9.2 Translation from DLP-Functions to MLPs

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

Definition 9.14 (MLP Translation).
Let $(\Pi_1, \dots, \Pi_n)$ be a sequence of DLP-functions such that the join $\Pi = \Pi_1 \sqcup \cdots \sqcup \Pi_n$ is defined. We define for each $a \in \mathrm{At}(\Pi_j)$, $1 \leq j \leq n$, the mapping

$$\nabla(a) = \begin{cases} P_k.a & a \in \mathrm{At_i}(\Pi_j) \text{ and there exists } \Pi_k \text{ such that } a \in \mathrm{At_o}(\Pi_k) \\ a & \text{otherwise} \end{cases} .$$

Let r be a rule of form (2.1) that appears in a DLP-function Π_j. Whenever r is normal or disjunctive, we let

$$\nabla(r) = \nabla(a_1) \vee \cdots \vee \nabla(a_k) \leftarrow \nabla(b_1), \dots, \nabla(b_m), \mathrm{not}\, \nabla(b_{m+1}), \dots, \mathrm{not}\, \nabla(b_n) \ ,$$

and if r is a constraint, we define

$$\nabla(r) = \mathit{fail} \leftarrow \nabla(b_1), \dots, \nabla(b_m), \mathrm{not}\, \nabla(b_{m+1}), \dots, \mathrm{not}\, \nabla(b_n), \mathrm{not}\, \mathit{fail} \ ,$$

where *fail* is a fresh atom not occurring in any Π_j and hidden from others. Given $\Pi_j = \langle R_j, I_j, O_j, H_j \rangle$, we let $\nabla(\Pi_j) = (P_j[], \nabla(R_j))$ be an MLP main module, where P_j is a module name and

$$\nabla(R_j) = \{\nabla(r) \mid r \in R_j\} \cup \left\{ a \vee \overline{a} \leftarrow \;\middle|\; a \in I_j \setminus \bigcup_{j \neq k} \mathrm{At_o}(\Pi_k) \right\}$$

such that $\overline{a}$ are fresh atoms not occurring in any Π_j. For the sequence $(\Pi_1, \dots, \Pi_n)$, we denote by $\nabla(\Pi)$ the MLP formed by

$$(\nabla(\Pi_1), \dots, \nabla(\Pi_n)) \ .$$

Note that Π_k in the definition of $\nabla(a)$ is unique since each distinct pair Π_j, Π_k from $(\Pi_1, \dots, \Pi_n)$ respect the input/output interfaces of each other, i.e., they satisfy condition $\mathrm{At_o}(\Pi_j) \cap \mathrm{At_o}(\Pi_k) = \emptyset$ for every $j \neq k$.

In the following, we exemplify the MLP translation using illustrative examples.

Example 9.1 Given a sequence (Π_1, Π_2) of DLP-functions, where

$$\Pi_1 = \langle \{a \leftarrow \text{not } b\}, \{b\}, \{a\}, \varnothing \rangle$$

and

$$\Pi_2 = \langle \{b \leftarrow \text{not } a\}, \{a\}, \{b\}, \varnothing \rangle \ ,$$

the MLP translation of the join $\Pi = \Pi_1 \sqcup \Pi_2$, which is given by

$$\Pi = \left\langle \left\{ \begin{array}{ll} a & \leftarrow \text{not } b \\ b & \leftarrow \text{not } a \end{array} \right\}, \varnothing, \{a, b\}, \varnothing \right\rangle \ ,$$

is $\nabla(\Pi) = (\nabla(\Pi_1), \nabla(\Pi_2))$, where $\nabla(\Pi_1)$ and $\nabla(\Pi_2)$ are the main modules whose associated sets of rules are $\{a \leftarrow \text{not } P_2.b\}$ and $\{b \leftarrow \text{not } P_1.a\}$, respectively. Here, both Π and $\nabla(\Pi)$ possess two answer sets: Π has $\{a\}$ and $\{b\}$, while $\nabla(\Pi)$ has $(\{a\}, \varnothing)$ and $(\varnothing, \{b\})$.

Now, let Π_1 be from above. In this case, we obtain the MLP $(\nabla(\Pi_1))$, where $\nabla(\Pi_1) = (P_1, \nabla(R_1))$ and $\nabla(R_1) = \{a \leftarrow \text{not } b;\ b \vee \overline{b} \leftarrow\}$. We have that

- Π_1 has the stable models $\{a\}$ and $\{b\}$, and
- $(\nabla(\Pi_1))$ has the answer sets $(\{a, \overline{b}\})$ and $(\{b\})$.

We show now that ∇ captures the stable models of DLP-functions.

Proposition 9.1 (Capturing stable models of DLP-functions)
Let $(\Pi_1, \ldots, \Pi_n)$ be a sequence of DLP-functions such that the join $\Pi = \Pi_1 \sqcup \cdots \sqcup \Pi_n$ is defined. Then, the stable models of Π correspond one-to-one to the answer sets of MLP $\nabla(\Pi)$.

Proof Let $(\Pi_1, \ldots, \Pi_n)$ be a sequence of DLP-functions such that each DLP-function $\Pi_i = \langle R_i, I_i, O_i, H_i \rangle$. We define for a set $A \subseteq \text{At}(\Pi)$ of atoms the set $\overline{A} = \{\overline{a} \mid a \in A\}$ of fresh atoms not occurring in Π.

($\Rightarrow$) Let N be a stable model of Π. Since $\Pi_1 \sqcup \cdots \sqcup \Pi_n$ is defined, we can apply the Module Theorem and get that $\text{SM}(\Pi) = \text{SM}(\Pi_1) \bowtie \cdots \bowtie \text{SM}(\Pi_n)$, hence $N = \bigcup_{i=1}^{n} N_i$ such that each $N_i \in \text{SM}(\Pi_i)$ is a stable model of Π_i and $N_i \triangleq_{\text{v}} N_j$ for all distinct pairs $i, j \in \{1, \ldots, n\}$. Let

$$F_i = I_i \setminus \bigcup_{i \neq j} O_j \ , 1 \leq i \leq n,$$

and let $\mathbf{M} = (M_1/\varnothing, \ldots, M_n/\varnothing)$ be an interpretation for MLP $\nabla(\Pi)$, where for each $P_i[\varnothing] \in \text{VC}(\nabla(\Pi))$ we set

$$M_i/\varnothing = N_i \setminus I_i \cup (N_i \cap F_i) \cup \overline{(F_i \setminus N_i)} \ . \tag{9.1}$$

We show now that $\mathbf{M}$ is an answer set of MLP $\nabla(\Pi)$ by showing that $\mathbf{M}$ is a model of $\nabla(\Pi)$ and that $\mathbf{M}$ is a minimal model of $f\,\nabla(\Pi)^{\mathbf{M}}$.

We first show that $\mathbf{M} \models \nabla(\Pi)$. Since N is a stable model of Π, we have that $\mathrm{At_o}(\Pi, N) \cup \mathrm{At_h}(\Pi, N)$ is a stable model of $\Pi/\mathrm{At_i}(\Pi, N)$. Since $N \models \Pi$, we have $N_k \models \Pi_k/\mathrm{At_i}(\Pi_k, N_k)$ for all $1 \leq k \leq n$. Let r be a rule from R_k, and let r' be the rule $(H(r) \setminus I_k) \leftarrow (B^+(r) \setminus I_k), \mathrm{not}(B^-(r) \setminus I_k)$. If $\mathrm{At_i}(\Pi_k, N_k) \not\models \mathrm{At_i}(\Pi_k, B^+(r))$ or $\mathrm{At_i}(\Pi_k, N_k) \models \mathrm{At_i}(\Pi_k, B^-(r))$ then $N_k \not\models B(r)$ and $r' \notin \Pi_k/\mathrm{At_i}(\Pi_k, N_k)$. Otherwise, $r' \in \Pi_k/\mathrm{At_i}(\Pi_k, N_k)$, thus $N_k \models \Pi_k/\mathrm{At_i}(\Pi, N_k)$ means either $N_k \not\models B(r')$ (then $N_k \not\models B(r)$) or $N_k \models H(r)$. Finally, we need to consider either

(1) $N_k \not\models B(r)$, or

(2) $N_k \models H(r)$.

Assuming (1), then either (a) there exists $a \in B^+(r)$ such that $N_k \not\models a$, or (b) there exists $a \in B^-(r)$ such that $N_k \models a$. We distinguish

- $a \in \mathrm{At_i}(\Pi_k) \setminus \bigcup \mathrm{At_o}(\Pi_\ell)$, or $a \notin \mathrm{At_i}(\Pi_k)$, then $\nabla(a) = a$, and by (9.1), we have in case (a) that $a \notin M_k/\varnothing$, hence $\mathbf{M}, P_k[\varnothing] \not\models B^+(\nabla(r))$, and therefore $\mathbf{M}, P_k[\varnothing] \not\models B(\nabla(r))$. In case (b) we have $a \in M_k/\varnothing$, hence $\mathbf{M}, P_k[\varnothing] \models B^-(\nabla(r))$, and thus $\mathbf{M}, P_k[\varnothing] \not\models B(\nabla(r))$;

- $a \in \mathrm{At_i}(\Pi_k)$ and there exists Π_ℓ, $1 \leq \ell \leq n$, such that $a \in \mathrm{At_o}(\Pi_\ell)$, then $\nabla(a) = P_\ell.a$. Furthermore, since $N_k \triangleq_{\mathrm{v}} N_\ell$, we have $N_k \cap \mathrm{At_v}(\Pi_\ell) = N_\ell \cap \mathrm{At_v}(\Pi_k)$. In case (a), from $a \in \mathrm{At_v}(\Pi_k)$, $a \in \mathrm{At_v}(\Pi_\ell)$, and $a \notin N_k$ follows $a \notin N_\ell$. Conversely, in case (b) from $a \in \mathrm{At_v}(\Pi_k)$, $a \in \mathrm{At_v}(\Pi_\ell)$, and $a \in N_k$, it holds that $a \in N_\ell$. Hence, by (9.1), we can deduce that $a \notin M_\ell/\varnothing$ and $\mathbf{M}, P_k[\varnothing] \not\models \nabla(a)$ in case (a), and $a \in M_\ell/\varnothing$ and $\mathbf{M}, P_k[\varnothing] \models \nabla(a)$ in case (b). Eventually, $\mathbf{M}, P_k[\varnothing] \not\models B(\nabla(r))$.

Now assume (2), then there exists $a \in H(r)$ such that $N_k \models a$. In this case, $a \notin \mathrm{At_i}(\Pi_k)$, hence $a \in M_k/\varnothing$ and we get that $\mathbf{M}, P_k[\varnothing] \models H(r)$.

We have now that $\mathbf{M}, P_k[\varnothing] \models \nabla(r)$ for each $r \in R_k$ and each $P_k[\varnothing] \in \mathrm{VC}(\nabla(\Pi))$. Now consider a rule r of form $a \vee \overline{a} \leftarrow \text{from } \nabla(\Pi_k)$. If $a \in N_k \cap F_k$, then $a \in M_k/\varnothing$ and thus $\mathbf{M}, P_k[\varnothing] \models r$. Otherwise, $\overline{a} \in \overline{F_k \setminus N_k}$, thus $\overline{a} \in M_k/\varnothing$ and $\mathbf{M}, P_k[\varnothing] \models r$. Therefore, all rules in $\nabla(\Pi_k)$ are satisfied by $\mathbf{M}$ at $P_k[\varnothing]$ for all $P_k[\varnothing] \in \mathrm{VC}(\mathbf{P})$, consequently $\mathbf{M}$ is a model of $\nabla(\Pi)$.

Next, we show that $\mathbf{M}$ is a minimal model for $f\,\nabla(\Pi)^{\mathbf{M}}$. Towards a contradiction, assume that there is an interpretation $\mathbf{M}' < \mathbf{M}$ such that $\mathbf{M}' \models f\,\nabla(\Pi)^{\mathbf{M}}$, i.e., there is an $\alpha \in M_k/\varnothing$ such that $\alpha \notin M'_k/\varnothing$. By (9.1), $\alpha \in N_k$ or α is of form $\overline{a}$, thus we distinguish

(i) $\alpha \in \mathrm{At}(\Pi_k)$ such that $\alpha \notin \mathrm{At_i}(\Pi_k)$,

(ii) $\alpha \in N_k \cap F_k$, or

(iii) $\alpha \in F_k \setminus N_k$ and $\alpha = \overline{a}$.

In case (i), we have $N_k \in \mathrm{MM}\left(\Pi_k^{N_k}\right)$, there must exist a rule $r \in R_k$ such that $\alpha \in H(r)$, $N_k \models B^+(r)$ and $N_k \not\models B^-(r)$, hence there exists a rule $r' = H(r) \leftarrow B^+(r) \in R_k^{N_k}$, and by minimality $N_k \setminus \{\alpha\} \not\models H(r)$. By our translation, $\nabla(\alpha) = \alpha$ for $\alpha \in H(r)$, thus $H(\nabla(r)) = \{\alpha\}$. Since $r' \in R_k^{N_k}$, we must have $\nabla(r) \in f\,\nabla(\Pi)^{\mathbf{M}}$. We therefore obtain $\mathbf{M}, P_k[\varnothing] \models B(\nabla(r))$ and $\mathbf{M}, P_k[\varnothing] \models H(\nabla(r))$, hence $\mathbf{M}', P_k[\varnothing] \models B(\nabla(r))$ and $\mathbf{M}', P_k[\varnothing] \not\models H(\nabla(r))$, hence $\mathbf{M}', P_k[\varnothing] \not\models \nabla(r)$, which is a contradiction to $\mathbf{M}' \models f\,\nabla(\Pi)^{\mathbf{M}}$. Considering case (ii), we obtain from our translation that there is a rule $r = a \vee \overline{a} \leftarrow$ in $f\,\nabla(\Pi)^{\mathbf{M}}$. Since $a \in N_k$, we have $a \in M_k/\varnothing$ but $\overline{a} \notin M_k/\varnothing$ as $a \notin F_k \setminus N_k$, hence $\mathbf{M}', P_k[\varnothing] \not\models r$, which is a contradiction for $\mathbf{M}'$ being a model of $f\,\nabla(\Pi)^{\mathbf{M}}$. In case (iii), we obtain from our translation that there is a rule $r = a \vee \overline{a} \leftarrow$ in $f\,\nabla(\Pi)^{\mathbf{M}}$. Since $\overline{a} \in \overline{F_k \setminus N_k}$, we have $a \notin N_k$ and thus $a \notin M_k/\varnothing$ and since $\overline{a} \notin M_k/\varnothing$ we get $\mathbf{M}', P_k[\varnothing] \not\models r$, which is a contradiction for $\mathbf{M}'$ being a model of $f\,\nabla(\Pi)^{\mathbf{M}}$. Thus, $\mathbf{M}$ is a minimal model of $f\,\nabla(\Pi)^{\mathbf{M}}$, and we therefore have established that $\mathbf{M}$ is an answer set for $\nabla(\Pi)$.

($\Leftarrow$) Let $\mathbf{M} = (M_1/\varnothing, \ldots, M_n/\varnothing)$ be an answer set of $\nabla(\Pi)$. Let $N = \bigcup_{k=1}^{n} N_k$, where

$$N_k = (M_k/\varnothing \cap \mathrm{At}(\Pi_k)) \cup \bigcup_{i \neq k} \{a \in \mathrm{At_o}(\Pi_i) \cap M_i/\varnothing \mid a \text{ appears in } R_k\} \tag{9.2}$$

We show that N is a stable model of Π by showing that for an N_k from N, N_k is a model of $\Pi_k/\,\mathrm{At_i}(\Pi_k, N_k)$ and N_k is a minimal model of $\Pi_k^{N_k}/\,\mathrm{At_i}(\Pi_k, N_k)$.

We show that N_k satisfies all rules in Π_k, then immediately it satisfies all rules in $\Pi_k/\,\mathrm{At_i}(\Pi_k, N_k)$. For a rule $r \in R_k$, if $N_k \not\models B(r)$ then $N_k \models r$. Otherwise, we need to show that $N_k \models H(r)$. For such a rule r, we have the corresponding rule $\nabla(r)$ in $\nabla(\Pi_k)$. Since $\mathbf{M}$ is a model of $\nabla(\Pi)$, either $\mathbf{M}, P_k[\varnothing] \not\models B(\nabla(r))$ or $\mathbf{M}, P_k[\varnothing] \models H(\nabla(r))$. We show that the first case cannot happen. Assume that $\mathbf{M}, P_k[\varnothing] \not\models B(\nabla(r))$ and $N_k \models B(r)$, then either (i) there exists $a \in B^+(r)$ such that $N_k \models a$ and $\mathbf{M}, P_k[\varnothing] \not\models \nabla(a)$, or (ii) there exists $a \in B^-(r)$ such that $N_k \not\models a$ and $\mathbf{M}, P_k[\varnothing] \models \nabla(a)$. We distinguish the following cases:

- $\nabla(a) = a$: We have $N_k \models a$ (respectively, $N_k \not\models a$) and $\mathbf{M}, P_k[\varnothing] \not\models a$ (respectively, $\mathbf{M}, P_k[\varnothing] \models a$) in case (i) (respectively, (ii)). By (9.2), there must exist a module atom $P_\ell.a$ in a rule in $R(m_k)$ (respectively, $a \in N_k$) in case (i) (respectively, (ii)). For case (i), this module atom must be translated from Π_ℓ, which means $\nabla(a) = P_\ell.a$. In both cases, we have now arrived at a contradiction.

- $\nabla(a) = P_\ell.a$: We have $N_k \models a$ (respectively, $N_k \not\models a$) and $\mathbf{M}, P_k[\varnothing] \not\models P_\ell.a$ (respectively, $\mathbf{M}, P_k[\varnothing] \models P_\ell.a$) in case (i) (respectively, (ii)). For case (i) we have $a \in M_k/\varnothing$ by (9.2), which we can only get from Π_k, hence $\nabla(a) = a$. In case (ii), $\mathbf{M}, P_\ell[\varnothing] \models a$, and from (9.2) we get $a \in N_k$. Again, both cases arrive at a contradiction.

We can conclude that $\mathbf{M}, P_k[\varnothing] \models H(r)$. Since $\nabla(a) = a$ for all $a \in H(r)$, there exists $a \in H(r)$ such that $a \in M_k/\varnothing$, therefore $a \in N_k$, hence $N_k \models H(r)$. We therefore conclude that $N_k \models r$, and thus $N_k \models \Pi_k/\operatorname{At_i}(\Pi_k, N_k)$, which leads to $N \models \Pi$.

Next we show that N_k is an I_k-minimal model of $\Phi_k = \Pi_k^{N_k}/\operatorname{At_i}(\Pi_k, N_k)$. Towards a contradiction, let us assume that there exists $N_k' \subset N_k$ such that $N_k' \models \Phi_k$, hence there exists $a \in N_k$ such that $a \notin N_k'$. By our translation, the following cases can arise:

- If $\nabla(a) = P_\ell.a$, then $a \in \operatorname{At_i}(\Pi_k)$, therefore $a \in \operatorname{At_i}(\Pi_k, N_k)$. Hence by the creation of Φ_k, there exists a fact $a \leftarrow$ in $\Pi_k/\operatorname{At_i}(\Pi_k, N_k)$, and also in Φ_k. Immediately, we get that $N_k' \not\models \Phi_k$, which is contradiction.

- Let $\nabla(a) = a$. If $a \in \operatorname{At_i}(\Pi_k)$, the above argument leads to a contradiction. In the following we consider the case in which $a \notin \operatorname{At_i}(\Pi_k)$. Since $a \in N_k$, it must be concluded from a rule r' from Φ_k. Hence there exists a rule r from Π_k such that $a \in H(r)$, $\operatorname{At_i}(\Pi_k, N_k) \models \operatorname{At_i}(\Pi_k, B^+(r)), \text{not } \operatorname{At_i}(\Pi_k, B^-(r))$, and $N_k \models \text{not } B^-(r)$. Due to our translation, a is also concluded in M_k from the rule $\nabla(r) \in P_k$. Since M_k is a minimal model of P_k, it holds that $M_k \models B(\nabla(r))$, $M_k \models H(\nabla(r))$, and $M_k \setminus \{a\} \not\models H(\nabla(r))$. By (9.2), atoms in M_k are copied into N_k, plus atoms from M_ℓ where a module atom exists. Since those atoms from M_ℓ do not appear in the head of a rule, we get that $N_k \models B(r)$, $N_k \models H(r)$, and $N_k \setminus \{a\} \not\models H(r)$. Therefore $N_k' \not\models r'$, and $N_k' \not\models \Phi_k$, a contradiction.

Eventually, we get that N_k is an I_k-minimal model of Φ_k, hence N is a stable model of Π. □

9.3 Translation from MLPs to DLP-Functions

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

1. m_ℓ retrieves input b from m_k explicitly in form of an additional fact $p \leftarrow$ whenever b holds in some instantiation of $P[q]$, which can be seen as call by value, and

2. m_k retrieves input from m_ℓ implicitly in form of a, which plays a similar role to call by reference input in DLP-functions.

Here, we restrict our attention to MLPs with input of type (2). By complexity arguments and the results of Chapter 5, translating MLPs with inputs of type (1) into sequences of DLP-functions is likely to cause an exponential blow-up in general.

Without loss of generality, we consider only propositional MLPs without input in which if a module atom $P_i.a$ appears in a rule $r \in R(m_j)$ of module m_j, then a also appears in heads of some rule $r' \in R(m_i)$. Roughly speaking, in such a situation, we can pre-process $R(m_j)$ as follows: if $P_i.a \in B^+(r)$ then remove r from $R(m_j)$; if $P_i.a \in B^-(r)$ then remove $P_i.a$ from $B^-(r)$. Furthermore, we assume that MLPs do not contain self-loops, i.e., in a module $m_i = (P_i[], Q_i)$ there is no rule in Q_i that contains a module atom of form $P_i.a$. Such module atoms can always be replaced simply by a in Q_i.

We proceed as follows: we first define a tagging function Δ_{P_i} which maps propositional ordinary and module atoms to propositional atoms. This mapping will be lifted to rules as usual. Finally, for a given module $m_i = (P_i[], R_i)$, the input-output interface of the corresponding DLP-function will be generated by the occurrences of module atoms in m_i by means of a mapping Δ, which will be lifted to whole MLPs as well.

Definition 9.15 (DLP-function translation).
Let $\mathbf{P} = (m_1, \dots, m_n)$ be a propositional MLP and $m_i = (P_i[], Q_i)$ be one of the modules of $\mathbf{P}$. For an atom $\alpha \in \mathrm{HB}_\mathbf{P}$, we define

$$\Delta_{P_i}(\alpha) = \begin{cases} a_{P_i} & \text{if } \alpha \text{ is an atom } a \\ b_{P_j} & \text{if } \alpha \text{ is a module atom of form } P_j.b \end{cases} .$$

For a rule $r \in Q_i$ of form (3.2), we denote by $\Delta_{P_i}(r)$ the DLP rule

$$\Delta_{P_i}(\alpha_1) \vee \dots \vee \Delta_{P_i}(\alpha_k) \leftarrow \Delta_{P_i}(\beta_1), \dots, \Delta_{P_i}(\beta_m), \text{not } \Delta_{P_i}(\beta_{m+1}), \dots, \text{not } \Delta_{P_i}(\beta_n) \ .$$

Given a module m_i from $\mathbf{P}$, we let $\Delta(m_i) = \langle R_i, I_i, O_i, H_i \rangle$ to be a DLP-function, where

- $R_i = \{\Delta_{P_i}(r) \mid r \in Q_i\}$,
- $I_i = \{\Delta_{P_i}(P_j.b) \mid P_j.b \text{ occurs in } Q_i\}$,
- $O_i = \{\Delta_{P_i}(a) \mid P_i.a \in \mathrm{HB}_\mathbf{P}\}$, and
- $H_i = \{\Delta_{P_i}(\alpha) \mid \alpha \in \mathrm{HB}_\mathbf{P} \text{ occurs in } Q_i\} \setminus (I_i \cup O_i)$.

Next, we provide some examples of Δ.

Example 9.2 Let $\mathbf{P} = (m_1, m_2)$ be a propositional MLP with modules $m_1 = (P_1[], Q_1)$ and $m_2 = (P_2[], Q_2)$, where $Q_1 = \{a \leftarrow \text{not } P_2.b\}$, and $Q_2 = \{b \leftarrow \text{not } P_1.a\}$. The translation of $\mathbf{P}$ to a sequence of DLP-functions is given by $(\Delta(m_1), \Delta(m_2))$, where

- $\Delta(m_1) = \langle \{a_{P_1} \leftarrow \text{not } b_{P_2}\}, \{b_{P_2}\}, \{a_{P_1}\}, \varnothing \rangle$ and
- $\Delta(m_2) = \langle \{b_{P_2} \leftarrow \text{not } a_{P_1}\}, \{a_{P_1}\}, \{b_{P_2}\}, \varnothing \rangle$.

Their join $\Pi = \Delta(m_1) \sqcup \Delta(m_2)$ is defined as

$$\Delta(m_1) \sqcup \Delta(m_2) = \left\langle \left\{ \begin{array}{ll} a_{P_1} & \leftarrow \text{not } b_{P_2} \\ b_{P_2} & \leftarrow \text{not } a_{P_1} \end{array} \right\}, \varnothing, \{a_{P_1}, b_{P_2}\}, \varnothing \right\rangle \ .$$

Both $\mathbf{P}$ and Π admit two answer sets respectively stable models:

- $(M_1/\varnothing \coloneqq \{a\}, M_2/\varnothing \coloneqq \varnothing)$ and $(M_1/\varnothing \coloneqq \varnothing, M_2/\varnothing \coloneqq \{b\})$ for $\mathbf{P}$, and
- $\{a_{P_1}\}$ and $\{b_{P_2}\}$ for Π.

Example 9.3 Consider an MLP $\mathbf{P} = (m_1)$ with single main module $m_1 = (P_1[], Q_1)$ such that $Q_1 = \{a \leftarrow \text{not } b \ ; \ b \vee \overline{b} \leftarrow\}$. We can translate $\mathbf{P}$ to a DLP-function $\Delta(m_1)$, where

$$\Delta(m_1) = \left\langle \left\{ \begin{array}{ll} a_{P_1} & \leftarrow \text{not } b_{P_1} \\ b_{P_1} \vee \overline{b}_{P_1} & \leftarrow \end{array} \right\}, \varnothing, \varnothing, \{a_{P_1}, b_{P_1}, \overline{b}_{P_1}\} \right\rangle \ .$$

Both $\mathbf{P}$ and $\Delta(m_1)$ have two answer sets respectively stable models:

- $(M_1/\varnothing \coloneqq \{a, \overline{b}\})$ and $(M_1/\varnothing \coloneqq \{b\})$ for $\mathbf{P}$, and
- $\{a_{P_1}, \overline{b}_{P_1}\}$ and $\{b_{P_1}\}$ for $\Delta(m_1)$.

Example 9.4 Consider an MLP $\mathbf{P} = (m_1, m_2)$ with modules $m_1 = (P_1[], Q_1)$ and $m_2 = (P_2[], Q_2)$ such that $Q_1 = \{a \leftarrow \text{not } b; \ b \vee \overline{b} \leftarrow\}$ and $Q_2 = \{d \leftarrow \text{not } P_1.a\}$. The sequence of DLP-functions $(\Delta(m_1), \Delta(m_2))$ is given by

$$\Delta(m_1) = \left\langle \left\{ \begin{array}{ll} a_{P_1} & \leftarrow \text{not } b_{P_1} \\ b_{P_1} \vee \overline{b}_{P_1} & \leftarrow \end{array} \right\}, \varnothing, \{a_{P_1}\}, \{b_{P_1}, \overline{b}_{P_1}\} \right\rangle$$

and

$$\Delta(m_2) = \langle \{d_{P_2} \leftarrow \text{not } a_{P_1}\}, \{a_{P_1}\}, \varnothing, \{d_{P_2}\} \rangle$$

such that their join $\Pi = \Delta(m_1) \sqcup \Delta(m_2)$ is

$$\left\langle \left\{ \begin{array}{ll} a_{P_1} & \leftarrow \text{not } b_{P_1} \\ b_{P_1} \vee \overline{b}_{P_1} & \leftarrow \\ d_{P_2} & \leftarrow \text{not } a_{P_1} \end{array} \right\}, \varnothing, \{a_{P_1}\}, \{b_{P_1}, \overline{b}_{P_1}, d_{P_2}\} \right\rangle \ .$$

We obtain two answer sets for $\mathbf{P}$ and two stable models for Π:

- $(M_1/\emptyset \coloneqq \{a, \overline{b}\}, M_2/\emptyset \coloneqq \emptyset)$ and $(M_1/\emptyset \coloneqq \{b\}, M_2/\emptyset \coloneqq \{d\})$ for $\mathbf{P}$, and
- $\{a_{P_1}, \overline{b}_{P_1}\}$ and $\{b_{P_1}, d_{P_2}\}$ for Π.

The mapping Δ translates MLPs to DLP-functions, which respect the hidden atoms due to the global renaming in Δ_{P_i}, hence the composition operator $\oplus$ is defined on them. DLP-functions $\Pi = \langle R, I, O, H\rangle$ require that the heads of the rules in R come from $O \cup H$. With Δ, this prerequisite comes for free, since input of translated DLP-functions comes from modular atoms which never appear in heads of rules.

However, the join operator $\sqcup$ is not always defined for two DLP-functions $\Delta(m_i)$ and $\Delta(m_j)$, where m_i and m_j are from $\mathbf{P}$. The join $\Delta(m_i) \sqcup \Delta(m_j)$ is only defined when $\Delta(m_i)$ and $\Delta(m_j)$ are mutually independent. We show this situation in the next example.

Example 9.5 When we translate the MLP

$$\mathbf{P} = (m_1 \coloneqq (P_1[], Q_1 \coloneqq \{a \leftarrow P_2.b\}), m_2 \coloneqq (P_2[], Q_2 \coloneqq \{b \leftarrow P_1.a\}))$$

into the sequence of DLP-functions $(\Delta(m_1), \Delta(m_2))$ such that

- $\Delta(m_1) = \langle\{a_{P_1} \leftarrow b_{P_2}\}, \{b_{P_2}\}, \{a_{P_1}\}, \emptyset\rangle$, and
- $\Delta(m_2) = \langle\{b_{P_2} \leftarrow a_{P_1}\}, \{a_{P_1}\}, \{b_{P_2}\}, \emptyset\rangle$,

we create the mutually dependent DLP-functions $\Delta(m_1)$ and $\Delta(m_2)$. The dependency graph of

$$\Delta(m_1) \oplus \Delta(m_2) = \left\langle \left\{ \begin{array}{ll} a_{P_1} & \leftarrow b_{P_2} \\ b_{P_2} & \leftarrow a_{P_1} \end{array} \right\}, \emptyset, \{a_{P_1}, b_{P_2}\}, \emptyset \right\rangle$$

has the SCC $C = \{a_{P_1}, b_{P_2}\}$ such that $C \cap \mathrm{At}_\mathrm{o}(\Delta(m_1)) \neq \emptyset$ and $C \cap \mathrm{At}_\mathrm{o}(\Delta(m_2)) \neq \emptyset$, hence the join $\Delta(m_1) \sqcup \Delta(m_2)$ is not defined.

Next, we provide a similar restriction on MLPs as the one given in Definition 9.5 on DLP-functions in order to create a faithful translation from MLPs to DLP-functions, such that the join $\sqcup$ is defined and the Module Theorem is applicable.

Definition 9.16 (Positive dependency graph).
Let $\mathbf{P}$ be an MLP, we define the *positive dependency graph* of $\mathbf{P}$ as the directed graph $G^+_\mathbf{P} = (V, E)$, where V contains all ordinary atoms from $\mathrm{HB}_\mathbf{P}$, and E is the set of edges $a \to^1 b$ such that for all modules m_i from $\mathbf{P}$ and rules $r \in R(m_i)$,

- $a \in H(r)$ and $b \in B^+(r)$, or
- $a \in H(r)$ and $P_j.b \in B^+(r)$.

The reflexive and transitive closure $\rightarrow$ of $\rightarrow^1$ is then a dependency relation over V.

A *strongly connected component* (SCC) C of $G^+_{\mathbf{P}}$ is a maximal set $C \subseteq V$ such that $a \rightarrow b$ for every pair $a, b \in C$.

We define $O(m_i) = \{a \mid P_i.a \in \mathrm{HB}_{\mathbf{P}}\}$. We say that two modules m_i and m_j from $\mathbf{P}$ are *mutually dependent*, if $G^+_{\mathbf{P}}$ has a strongly connected component C such that $C \cap O(m_i) \neq \emptyset$ and $C \cap O(m_j) \neq \emptyset$. If m_1 and m_2 are not mutually dependent, we call them *mutually independent*.

The next result shows that given two modules m_1 and m_2 that are mutually dependent, the translated DLP-functions $\Delta(m_1)$ and $\Delta(m_2)$ are also mutually dependent, and vice versa.

Lemma 9.2
Let $\mathbf{P} = (m_1, m_2)$ be an MLP and $\Pi_1 = \Delta(m_1)$ and $\Pi_2 = \Delta(m_2)$ be two DLP-functions. Then, m_1 and m_2 are mutually dependent iff Π_1 and Π_2 are mutually dependent.

Proof We have to show that there exists an SCC $C_{\mathbf{P}}$ in $G^+_{\mathbf{P}}$ such that $C_{\mathbf{P}} \cap O(m_1) \neq \emptyset$ and $C_{\mathbf{P}} \cap O(m_2) \neq \emptyset$ if and only if there exists and SCC C_Π in $DG^+(\Pi_1 \oplus \Pi_2)$ such that $C_\Pi \cap \mathrm{At_o}(\Pi_1) \neq \emptyset$ and $C_\Pi \cap \mathrm{At_o}(\Pi_2) \neq \emptyset$.

We obtain for the not necessarily distinct pair of modules $m_i, m_j \in \{m_1, m_2\}$ the following observation: the dependency $b_{P_j} \leq_1 a_{P_i}$ is in $DG^+(\Pi_1 \oplus \Pi_2)$ if and only if the dependency $a \rightarrow_1 b$ is in $G^+_{\mathbf{P}}$ such that a is from m_i and b is from m_j. This holds since every rule $r \in R(m_i) \cup R(m_j)$ appears as $\Delta(r)$ in $R_i \cup R_j$, thus there exists a bijection such that every element $(b_{P_j}, a_{P_i}) \in \leq_1$ can be mapped to an element $(a, b) \in \rightarrow_1$. Hence, there exists a bijection from the reflexive and transitive closure of $\leq_1$ to the reflexive and transitive closure of $\rightarrow_1$, which means that there is a bijection of the SCCs in $DG^+(\Pi_1 \oplus \Pi_2)$ to the SCCs in $G^+_{\mathbf{P}}$.

What remains to be shown is that for an SCC $C_{\mathbf{P}}$ in $G^+_{\mathbf{P}}$ such that $C_{\mathbf{P}} \cap O(m_1) \neq \emptyset$ and $C_{\mathbf{P}} \cap O(m_2) \neq \emptyset$ there is an SCC C_Π in $DG^+(\Pi_1 \oplus \Pi_2)$ such that $C_\Pi \cap \mathrm{At_o}(\Pi_1) \neq \emptyset$ and $C_\Pi \cap \mathrm{At_o}(\Pi_2) \neq \emptyset$, and vice versa.

($\Rightarrow$) Let $C_{\mathbf{P}}$ be an SCC in $G^+_{\mathbf{P}}$ such that $C_{\mathbf{P}} \cap O(m_1) \neq \emptyset$ and $C_{\mathbf{P}} \cap O(m_2) \neq \emptyset$. Let

$$C_\Pi = \{a_{P_i} \mid a \in C_{\mathbf{P}} \text{ and } a \text{ appears in } m_i \text{ for } i \in \{1, 2\}\}.$$

We need to show that $C_\Pi \cap \mathrm{At_o}(\Pi_1) \neq \emptyset$ and $C_\Pi \cap \mathrm{At_o}(\Pi_2) \neq \emptyset$. Let $i \in \{1, 2\}$ and let $a \in C_{\mathbf{P}} \cap O(m_i)$, thus there exists a module atom $P_i.a$ in $\mathbf{P}$ by $a \in O(m_i)$, hence $a_{P_i} \in \mathrm{At_o}(\Pi_i)$. And since $a \in O(m_i)$, we must have that a appears in m_i, and thus $a \in C_{\mathbf{P}}$ implies that $a_{P_i} \in C_\Pi$. Therefore, $a_{P_i} \in C_\Pi \cap \mathrm{At_o}(\Pi_i)$, and hence $C_\Pi \cap \mathrm{At_o}(\Pi_i) \neq \emptyset$ for $i \in \{1, 2\}$.

($\Leftarrow$) Let C_Π be an SCC in $DG^+(\Pi_1 \oplus \Pi_2)$ such that $C_\Pi \cap \mathrm{At_o}(\Pi_1) \neq \emptyset$ and $C_\Pi \cap \mathrm{At_o}(\Pi_2) \neq \emptyset$. Let

$$C_{\mathbf{P}} = \{a \mid a_{P_i} \in C_\Pi \text{ and } a_{P_i} \text{ appears in } \Pi_i \text{ for } i \in \{1, 2\}\}.$$

We need to show that $C_{\mathbf{P}} \cap O(m_1) \neq \emptyset$ and $C_{\mathbf{P}} \cap O(m_2) \neq \emptyset$. Let $i \in \{1,2\}$ and let $a_{P_i} \in C_\Pi \cap \mathrm{At_o}(\Pi_i)$, thus there exists a module atom $P_i.a$ in $\mathbf{P}$ by $a_{P_i} \in \mathrm{At_o}(\Pi_i)$, hence $P_i.a \in \mathrm{HB}_{\mathbf{P}}$ implies $a \in O(m_i)$. And since $a_{P_i} \in \mathrm{At_o}(\Pi_i)$, we must have that a_{P_i} appears in Π_i, and thus $a_{P_i} \in C_\Pi$ implies that $a \in C_{\mathbf{P}}$. Therefore, $a \in C_{\mathbf{P}} \cap O(m_i)$, and hence $C_{\mathbf{P}} \cap O(m_i) \neq \emptyset$ for $i \in \{1,2\}$. □

For the following results, we define for a set of atoms X the set $\Delta_{P_i}(X) = \{\Delta_{P_i}(a) \mid a \in X\}$. Let $\mathbf{P} = (m_1, m_2)$ be an MLP with two modules without input, and let $\mathbf{M}$ be an interpretation for $\mathbf{P}$. We define a permutation $\pi\colon \{1,2\} \to \{1,2\}$ such that $\pi = \left(\begin{smallmatrix}1 & 2\\ 2 & 1\end{smallmatrix}\right)$, and let

$$I_k^{\mathbf{P}}(\mathbf{M}) = \Delta_{P_k}(M_k/\emptyset) \cup \left(\mathrm{At_i}(\Delta(m_k)) \cap \Delta_{P_{\pi(k)}}\left(M_{\pi(k)}/\emptyset\right)\right)$$

for $k \in \{1,2\}$ be a DLP interpretation.

The following Lemma shows that $I_1^{\mathbf{P}}(\mathbf{M})$ and $I_2^{\mathbf{P}}(\mathbf{M})$ are mutually compatible.

Lemma 9.3
Let $\mathbf{P} = (m_1, m_2)$ be an MLP such that m_1 and m_2 are mutually independent, and $\mathbf{M} = (M_1/\emptyset, M_2/\emptyset)$ be an interpretation for $\mathbf{P}$, then $I_1^{\mathbf{P}}(\mathbf{M}) \triangleq_{\mathrm{v}} I_2^{\mathbf{P}}(\mathbf{M})$ with respect to DLP-functions $\Delta(m_1)$ and $\Delta(m_2)$.

Proof We need to show that $I_1^{\mathbf{P}}(\mathbf{M}) \cap \mathrm{At_v}(\Delta(m_2)) = I_2^{\mathbf{P}}(\mathbf{M}) \cap \mathrm{At_v}(\Delta(m_1))$. Since m_1 and m_2 are mutually independent, we get that $\Delta(m_1)$ and $\Delta(m_2)$ are mutually independent by Lemma 9.2, i.e., for all SCCs C in $DG^+(\Delta(m_1) \oplus \Delta(m_2))$ we have $C \cap \mathrm{At_o}(\Delta(m_2)) = \emptyset$ or $C \cap \mathrm{At_o}(\Delta(m_1)) = \emptyset$. By definition of Δ, we get that $\alpha \in \mathrm{At_o}(\Delta(m_2))$ iff $\alpha \in \mathrm{At_i}(\Delta(m_1))$, and $\alpha \in \mathrm{At_o}(\Delta(m_1))$ iff $\alpha \in \mathrm{At_i}(\Delta(m_2))$. Thus, both $\mathrm{At_o}(\Delta(m_2)) = \mathrm{At_i}(\Delta(m_1))$ and $\mathrm{At_o}(\Delta(m_1)) = \mathrm{At_i}(\Delta(m_2))$ hold, and therefore $\mathrm{At_v}(\Delta(m_1)) = \mathrm{At_v}(\Delta(m_2))$. Then, both $\Delta_{P_1}(M_1/\emptyset) \cap \mathrm{At_o}(\Delta(m_2)) = \emptyset$ and $\Delta_{P_2}(M_2/\emptyset) \cap \mathrm{At_o}(\Delta(m_1)) = \emptyset$ hold, as well as both $\Delta_{P_1}(M_1/\emptyset) \cap \mathrm{At_i}(\Delta(m_1)) = \emptyset$ and $\Delta_{P_2}(M_2/\emptyset) \cap \mathrm{At_i}(\Delta(m_2)) = \emptyset$.

Hence, by following the chain of equivalent transformations, we obtain:

$$\begin{aligned}
&I_1^{\mathbf{P}}(\mathbf{M}) \cap \mathrm{At_v}(\Delta(m_2)) = \\
&\qquad \left(\Delta_{P_1}(M_1/\emptyset) \cup \left(\mathrm{At_i}(\Delta(m_1)) \cap \Delta_{P_2}(M_2/\emptyset)\right)\right) \cap \mathrm{At_v}(\Delta(m_2)) = \\
&\qquad \left(\Delta_{P_1}(M_1/\emptyset) \cup \left(\mathrm{At_i}(\Delta(m_1)) \cap \Delta_{P_2}(M_2/\emptyset)\right)\right) \cap \left(\mathrm{At_i}(\Delta(m_2)) \cup \mathrm{At_o}(\Delta(m_2))\right) = \\
&\qquad \left(\Delta_{P_1}(M_1/\emptyset) \cap \mathrm{At_i}(\Delta(m_2))\right) \cup \left(\mathrm{At_i}(\Delta(m_1)) \cap \Delta_{P_2}(M_2/\emptyset) \cap \mathrm{At_o}(\Delta(m_2))\right) = \\
&\qquad \left(\Delta_{P_1}(M_1/\emptyset) \cap \mathrm{At_i}(\Delta(m_2))\right) \cup \left(\mathrm{At_o}(\Delta(m_2)) \cap \Delta_{P_2}(M_2/\emptyset)\right) =
\end{aligned}$$

$$\begin{gathered}\left(\Delta_{P_2}(M_2/\varnothing) \cap \mathrm{At_o}(\Delta(m_2))\right) \cup \left(\mathrm{At_i}(\Delta(m_2)) \cap \Delta_{P_1}(M_1/\varnothing)\right) = \\ \left(\Delta_{P_2}(M_2/\varnothing) \cap \mathrm{At_i}(\Delta(m_1))\right) \cup \left(\mathrm{At_i}(\Delta(m_2)) \cap \Delta_{P_1}(M_1/\varnothing) \cap \mathrm{At_o}(\Delta(m_1))\right) = \\ \left(\Delta_{P_2}(M_2/\varnothing) \cup \left(\mathrm{At_i}(\Delta(m_2)) \cap \Delta_{P_1}(M_1/\varnothing)\right)\right) \cap \left(\mathrm{At_i}(\Delta(m_1)) \cup \mathrm{At_o}(\Delta(m_1))\right) = \\ \left(\Delta_{P_2}(M_2/\varnothing) \cup \left(\mathrm{At_i}(\Delta(m_2)) \cap \Delta_{P_1}(M_1/\varnothing)\right)\right) \cap \mathrm{At_v}(\Delta(m_1)) = \\ I_2^{\mathbf{P}}(\mathbf{M}) \cap \mathrm{At_v}(\Delta(m_1)) \ . \quad \square\end{gathered}$$

Akin to the Module Theorem for DLP-functions, we can show now a variant of it applied to the MLPs, which we then use to proof that Δ captures the answer sets of MLPs.

Theorem 9.4 (MLP module theorem)
Let $\mathbf{P} = (m_1, m_2)$ be an MLP such that m_1 and m_2 are mutually independent. Then, $\mathrm{SM}(\Delta(m_1) \sqcup \Delta(m_2)) = \mathrm{SM}(\Delta(m_1)) \bowtie \mathrm{SM}(\Delta(m_2))$.

Proof This theorem follows from Lemma 9.2, Lemma 9.3, and the Module Theorem by showing that for an answer set $\mathbf{M}$ of $\mathbf{P}$, we have $I_1^{\mathbf{P}}(\mathbf{M}) \cup I_2^{\mathbf{P}}(\mathbf{M}) \in \mathrm{SM}(\Delta(m_1)) \bowtie \mathrm{SM}(\Delta(m_2))$ if and only if $I_1^{\mathbf{P}}(\mathbf{M}) \in \mathrm{SM}(\Delta(m_1))$ and $I_2^{\mathbf{P}}(\mathbf{M}) \in \mathrm{SM}(\Delta(m_2))$. $\square$

Now we can prove that answer sets of MLPs correspond to stable models of the DLP-function $\bigsqcup_i \Delta(m_i)$. We first do this for MLPs consisting of two mutually independent modules m_1 and m_2, the generalization to n modules then follows immediately.

Proposition 9.5 (Capturing mutually independent MLPs)
Let $\mathbf{P} = (m_1, m_2)$ be an MLP such that m_1 and m_2 are mutually independent. Then, the answer sets of $\mathbf{P}$ correspond one-to-one to the stable models of $\Delta(m_1) \sqcup \Delta(m_2)$.

Proof For readability, let $\Pi = \Pi_1 \sqcup \Pi_2$ stand for the DLP-function $\Delta(m_1) \sqcup \Delta(m_2)$ such that for $i \in \{1, 2\}$, $\Pi_i = \langle R_i, I_i, O_i, H_i \rangle$.

($\Rightarrow$) Let $\mathbf{M} = (M_1/\varnothing, M_2/\varnothing)$ be an answer set of $\mathbf{P}$. We create $N = N_1 \cup N_2$ as an interpretation for Π, where $N_1 = I_1^{\mathbf{P}}(\mathbf{M})$ and $N_2 = I_2^{\mathbf{P}}(\mathbf{M})$. Recall that

$$I_k^{\mathbf{P}}(\mathbf{M}) = \Delta_{P_k}(M_k/\varnothing) \cup \left(\mathrm{At_i}(\Delta(m_k)) \cap \Delta_{P_{\pi(k)}}\left(M_{\pi(k)}/\varnothing\right)\right) \ ,$$

we show now that N is a stable model of Π by showing that N is a model of Π and that N_k is a minimal model of $\Pi_k^{N_k} / \mathrm{At_i}(\Pi_k, N_k)$ for $k \in \{1, 2\}$.

Since $\mathbf{M}$ is an answer set of $\mathbf{P}$, we have that $\mathbf{M}, P_k[\varnothing] \models R(m_k)$ for all $k \in \{1, 2\}$. Let $r \in R(m_k)$, for which we have a corresponding rule $\Delta_{P_k}(r)$ in R_k. By $\mathbf{M}, P_k[\varnothing] \models R(m_k)$, either $\mathbf{M}, P_k[\varnothing] \not\models B(r)$ or $\mathbf{M}, P_k[\varnothing] \models H(r)$. We show that $N_k \not\models B(\Delta_{P_k}(r))$ or $N_k \models H(\Delta_{P_k}(r))$. We distinguish two cases for $\Delta_{P_k}(\alpha)$ for atoms α appearing in r:

- If $\alpha = a$ is an ordinary atom, then $\Delta_{P_k}(a) = a_{P_k}$. We have $\mathbf{M}, P_k[\varnothing] \not\models B(r)$ whenever $\alpha \in B(r)$ (respectively, $\mathbf{M}, P_k[\varnothing] \models H(r)$ whenever $\alpha \in H(r)$). Since $N_k = I_k^{\mathbf{P}}(\mathbf{M})$, we obtain $a_{P_k} \notin N_k$ for $\mathbf{M}, P_k[\varnothing] \not\models B(r)$ and $a_{P_k} \in N_k$ for $\mathbf{M}, P_k[\varnothing] \models H(r)$, respectively. Thus, $N_k \not\models B(\Delta_{P_k}(r))$ (respectively, $N_k \models H(\Delta_{P_k}(r))$).

- If α is a module atom of the form $P_\ell.b$, then $\Delta_{P_k}(a) = b_{P_\ell}$. We have $\mathbf{M}, P_k[\varnothing] \not\models B(r)$ whenever $\alpha \in B(r)$, thus $N_k = I_k^{\mathbf{P}}(\mathbf{M})$ implies that $b_{P_\ell} \notin N_k$ and so we obtain $N_k \not\models B(\Delta_{P_k}(r))$.

Therefore, $N_k \models \Delta_{P_k}(r)$, and from this we get that $N_k \models \Pi_k$ and $N \models \Pi$.

Now we show that N_k is a minimal model of $\Phi_k = \Pi_k^{N_k} / \operatorname{At_i}(\Pi_k, N_k)$ for $k \in \{1, 2\}$. Towards a contradiction, let us assume that there is an $N_k' \subset N_k$ such that $N_k' \models \Phi_k$, i.e., there is an $a_{P_\ell} \in N_k$ such that $a_{P_\ell} \notin N_k'$. There are two cases to consider:

- If $a_{P_\ell} = a_{P_{\pi(k)}}$, then $a_{P_{\pi(k)}} \in \operatorname{At_i}(\Pi_k, N_k)$. Now consider $\mathbf{M}'$ such that $M_k'/\varnothing = M_k/\varnothing$ and $M_{\pi(k)}'/\varnothing = M_{\pi(k)}/\varnothing \setminus \{a\}$. Then $\mathbf{M}' < \mathbf{M}$, hence $\mathbf{M}' \not\models f\,\mathbf{P}^{\mathbf{M}}$ as $\mathbf{M}$ is a minimal model of $f\,\mathbf{P}^{\mathbf{M}}$. There is a rule $r \in f\,\mathbf{P}(P_k[\varnothing])^{\mathbf{M}}$ such that $\mathbf{M}', P_k[\varnothing] \not\models r$, thus $\mathbf{M}', P_k[\varnothing] \models B(r)$ and $\mathbf{M}', P_k[\varnothing] \not\models H(r)$. There must be a corresponding rule $\Delta_{P_k}(r) \in R_k$ such that $N_k \models B(\Delta_{P_k}(r))$, thus the reduced rule $r' = H(\Delta_{P_k}(r)) \leftarrow B^+(\Delta_{P_k}(r)) \in \Pi_k^{N_k}$. Therefore, $N_k' \not\models r'$ as $H(r') = H(\Delta_{P_k}(r))$, $a_{P_{\pi(k)}} \in H(r')$ and $a_{P_{\pi(k)}} \notin N_k'$. We can conclude that $N_k' \not\models \Phi_k$, a contradiction.

- If $a_{P_\ell} = a_{P_k}$, then $a \in M_k/\varnothing$. Now consider $\mathbf{M}'$ such that $M_k'/\varnothing = M_k/\varnothing \setminus \{a\}$ and $M_{\pi(k)}'/\varnothing = M_{\pi(k)}/\varnothing$. Due to minimality of $\mathbf{M}$, there exists a rule $r \in f\,\mathbf{P}(P_k[\varnothing])^{\mathbf{M}}$ such that $\mathbf{M}, P_k[\varnothing] \models B(r)$, $\mathbf{M}, P_k[\varnothing] \models H(r)$, but $\mathbf{M}', P_k[\varnothing] \not\models H(r)$. There is a corresponding rule $\Delta_{P_k}(r) \in R_k$ such that $N_k \models B(\Delta_{P_k}(r))$, thus the reduced rule $r' = H(\Delta_{P_k}(r)) \leftarrow B^+(\Delta_{P_k}(r)) \in \Pi_k^{N_k}$. Therefore, $N_k' \not\models r'$ as $H(r') = H(\Delta_{P_k}(r))$, $a_{P_k} \in H(r')$ and $a_{P_k} \notin N_k'$. We can conclude that $N_k' \not\models \Phi_k$, a contradiction.

Therefore, N_k is a minimal model of Φ_k, and therefore N is an stable model of $\Pi_1 \sqcup \Pi_2$.

($\Leftarrow$) Let $N = N_1 \cup N_2$ be a stable model of Π. Let $\mathbf{M} = (M_1/\varnothing, M_2/\varnothing)$ such that

$$M_i/\varnothing = \{a \mid a_{P_i} \in N\} \ .$$

We show that $\mathbf{M}$ is an answer set of $\mathbf{P}$ by showing that $\mathbf{M} \models \mathbf{P}$ and that $\mathbf{M}$ is a minimal model of $f\,\mathbf{P}^{\mathbf{M}}$.

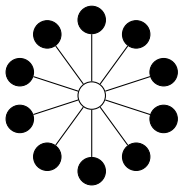
Since N is a stable model of Π, $N_k \models \Pi_k$ for all $k \in \{1, 2\}$. Then, a rule $r \in R(m_k)$ has a corresponding rule $\Delta_{P_k}(r) \in R_k$ such that $N_k \models \Delta_{P_k}(r)$. From the construction

of $\mathbf{M}$ we immediately get that $\mathbf{M}, P_k[\varnothing] \models r$ iff $N_k \models \Delta_{P_k}(r)$, hence $\mathbf{M}, P_k[\varnothing] \models R(m_k)$ for all $k \in \{1, 2\}$ and thus $\mathbf{M} \models \mathbf{P}$.

Now we show that $\mathbf{M}$ is a minimal model of $f\,\mathbf{P}^{\mathbf{M}}$. Towards a contradiction, let us assume that there is an $\mathbf{M}' < \mathbf{M}$ such that $\mathbf{M}' \models f\,\mathbf{P}^{\mathbf{M}}$, i.e., there is an $a \in M_k/\varnothing$ such that $a \notin M'_k/\varnothing$. Since $a \in M_k/\varnothing$ we get $a_{P_k} \in N$, and since $N_1 \triangleq_{\mathrm{v}} N_2$, we obtain $a_{P_k} \in N_k$. Since $N_k \in \mathrm{MM}\left(\Pi_k^{N_k}\right)$, there must exist a rule $\Delta_{P_k}(r) \in R_k$ and a the corresponding reduced rule r' in $\Pi_k^{N_k}$ such that the following statements hold: $a_{P_k} \in H(\Delta_{P_k}(r))$, $a_{P_k} \in H(r')$ as $H(r') = H(\Delta_{P_k}(r))$, $N_k \models B(\Delta_{P_k}(r))$, $N_k \models B(r')$ and therefore $N_k \models B^+(r')$, $N_k \models H(\Delta_{P_k}(r))$ and thus $N_k \models H(r')$, and $N_k \setminus \{a_{P_k}\} \not\models H(\Delta_{P_k}(r))$, which implies $N_k \setminus \{a_{P_k}\} \not\models H(r')$. Therefore, r must appear in $f\,\mathbf{P}(P_k[\varnothing])^{\mathbf{M}}$ as $\mathbf{M}, P_k[\varnothing] \models B(r)$. But then we obtain that $\mathbf{M}', P_k[\varnothing] \models B(r)$ and $\mathbf{M}', P_k[\varnothing] \not\models H(r)$, and so $\mathbf{M}', P_k[\varnothing] \not\models r$, a contradiction to our assumption that $\mathbf{M}' \models f\,\mathbf{P}^{\mathbf{M}}$. We can conclude that $\mathbf{M}$ is an answer set of $\mathbf{P}$. □

From Theorem 9.4 and Proposition 9.5 we can derive that Δ faithfully translates arbitrary mutually independent MLPs to DLP-functions.

Corollary 9.6 (Capturing answer sets of MLPs)
Let $\mathbf{P} = (m_1, \ldots, m_n)$ such that the join $\Delta(m_i) \sqcup \Delta(m_j)$ for all m_i and m_j are pairwise defined and are mutually independent. Then, there is a one-to-one correspondence between the answer sets of $\mathbf{P}$ and the stable models of $\Delta(m_1) \sqcup \cdots \sqcup \Delta(m_n)$.



10

Related Work

Several modular logic programming formalisms and frameworks have been proposed in the ASP context, which we are going to compare to MLPs in this chapter. We have already reviewed the history of modularity in logic programming in §1.2, modular logic programs based on generalized quantifiers by Eiter et al. (1997b) in §2.3, and provided an in-detail investigation of DLP-functions by Janhunen et al. (2009b) in Chapter 9. This chapter will now review further and related approaches to modularity in logic programming.

10.1 Compositional Approaches

Enumeration operators Fitting (1987) defines logic programs as recursion-theoretic *enumeration operators* (see Rogers, Jr., 1987) that are used in modular logic programs. Let ω be the set $\{0, 1, 2, \ldots\}$. Enumeration operators are functions of type $2^\omega \mapsto 2^\omega$, and Fitting uses them as the basis for defining a minimal model semantics for modular Horn clauses of the form $[\mathbf{P}^{\mathbf{I}}_{\mathbf{O}}]$, where $\mathbf{P}$ is a set of Horn clauses, $\mathbf{I}$ is an input predicate, and $\mathbf{O}$ is an output predicate, such that for an input set $S \subseteq \omega$ the application $[\mathbf{P}^{\mathbf{I}}_{\mathbf{O}}](S)$ is the set of n-tuples $\mathbf{x} \subseteq \omega \times \cdots \times \omega$ such that $\mathbf{O}(\mathbf{x})$ can be derived from the logic program $\mathbf{P} \cup \{\mathbf{I}(\mathbf{y}) \mid \mathbf{y} \in S\}$. Then, two modules $[\mathbf{P}^{\mathbf{I}}_{\mathbf{J}}]$ and $[\mathbf{Q}^{\mathbf{K}}_{\mathbf{L}}]$ can be combined to build a new module using closure operations like composition $[\mathbf{P}^{\mathbf{I}}_{\mathbf{J}}]([\mathbf{Q}^{\mathbf{K}}_{\mathbf{L}}](S))$ or Cartesian product $[\mathbf{P}^{\mathbf{I}}_{\mathbf{J}}](S) \times [\mathbf{Q}^{\mathbf{K}}_{\mathbf{L}}](S)$. This approach effectively brings a functional programming approach to modularity in logic programming, and bears some similarity to the MLP approach we have defined here. In contrast to enumeration operators, MLPs allow to define a full ensemble of modules as a program that is linked together using recursive module calls, while the combination of enumeration operators is strictly hierarchical. MLPs use disjunctive rules and answer set semantics, whereas enumeration operators are restricted to Horn clauses and minimal model semantics.

Compositional ASP Modules and DLP-Functions Another compositional approach is defined by Janhunen et al. (2009b) and Oikarinen (2008), which represent modular answer set programs by means of module composition operators as a way to compose individual modules. Each program module consists of an input/output interface which is used to combine the modules. In the most recent work, Janhunen et al. (2009b) extended the Gaifman-Shapiro-style module architecture (Gaifman and Shapiro, 1989) to the case of disjunctive logic programs as *DLP-functions*. The main differences and similarities between MLPs and DLP-functions have been studied in greater detail in Chapter 9.

Multi-Shot ASP Solving An interesting approach that semantically builds upon the Module Theorem as shown by Oikarinen (2008) is *Multi-Shot ASP solving* (Gebser et al., 2018, 2017), which uses Clingo as a backend for evaluating multi-shot programs. Here, multi-shot logic programs behave as modules that evolve during grounding and solving, using an imperative component for continuously selecting and linking of replaceable logic program modules. The Clingo system uses the `#program` directive to structure logic programs into subcomponents. Modules in the sense of multi-shot ASP solving are parameterizable subprograms that take input as a tuple of terms that are used to ground the subprogram, in contrast to the relational input to program modules in MLPs. Similar to the compositional approach by Oikarinen (2008), multi-shot solving programs are hierarchical and may not be defined when atoms are cyclic over module boundaries.

First-Order Modular Logic Programs Harrison and Lierler (2016) define *first-order modular logic programs* and their conservative extensions using a finite set of modules $\{\mathrm{SM}_{\mathbf{p}_1}[F_1], \ldots, \mathrm{SM}_{\mathbf{p}_n}[F_n]\}$, where each module $\mathrm{SM}_{\mathbf{p}_i}[F_i]$ is identified by the *stable model operator of first-order formula F_i with intensional predicates* $\mathbf{p}_i$ as defined by Ferraris et al. (2011). This approach classifies modular programs into *coherent* programs (no mutual positive recursion over module boundaries) and *incoherent* ones like the following: $\{\mathrm{SM}_p[q \supset p], \mathrm{SM}_q[p \supset q]\}$. Coherent modular programs can be identified as ordinary logic programs, as their answer sets coincide, whereas incoherent programs do not have this property. Clearly, the incoherent modular program shown before essentially amounts to Example 3.2 in the MLP setting. First-order modular logic programs build upon results for *propositional modular logic programs* (Lierler and Truszczyński, 2013), which used *input answer sets* (Lierler and Truszczyński, 2011) as the means to assign answer sets to a set of propositional modules. A set X of atoms is an *input answer set* of a normal logic program Π if X is an answer set of of the program $\Pi \cup (X \setminus hd(\Pi))$, where $hd(\Pi)$ denotes the set of rules head from Π. Then, for a set of programs $\mathcal{P} = \{\Pi_1, \ldots, \Pi_n\}$ (called propositional modular logic program), X is an answer set of $\mathcal{P}$ if X is an input answer set for each $\Pi_i \in \mathcal{P}$.

Compositional Semantics for Cyclic ASP Modules Moura (2016) worked on the framework for normal logic programs as defined by Oikarinen and Janhunen (2008) and—in contrast to the other methods reviewed here—allows positive recursion over modules using a translational approach, which modifies the original modules such that the natural join $\bowtie$ is applicable on them. To this end, a *conservative composition operator* $\otimes$ and an *output renaming* translation are defined such that the *conservative module theorem* provides a variation of the Module Theorem (see Chapter 9). This approach provides a compositional semantics with respect to the original signature before the translation. Different from the approaches above, every conservative composition introduces fresh atoms that do not appear in the original signature of the modules to be composed, and an additional module that was not present before. In general, this technique introduces additional stable models that would not exist on the original signature.

10.2 Modularity by Language Constructs

Macros and Ensembles Towards code reusability in ASP, Baral et al. (2006) introduced language constructs that allow one to specify reusable modules and call those via *macros*. A module is defined in a parameterized way, i.e., predicates/relations and variables appearing in the argument list of the module definition are at a schematic level and can be replaced by actual predicates and variables in the calls. A module may include call-macro statements to other modules; however, the calling relation must be acyclic. Furthermore, a module can specialize or generalize another module, which can be seen as a similarity to class-subclass definition used in object oriented programming.

For the semantics, the authors described an initial macro-expansion phase in which macro calls are appropriately replaced by AnsProlog code. The resulted program can then be evaluated by a usual ASP solver.

To allow easy management of modules, the authors proposed to group modules under "headings" called *ensembles*, which are pairs $E = (S, R)$ of a set of classes S and a set of relation schemas R. Associated with each ensemble is a set of modules about those classes and schemas. Exploiting the class-subclass relation as an inheritance mechanism, the notion of sub-ensembles was defined in which a sub-ensemble $E' = (S', R')$ of E can inherit or override original modules in E, have more modules, depending on the relations between S and S', as well as R and R'.

Templates Similar to macros is the DLP^T *template* approach, which was introduced in the DLP^T language (Calimeri and Ianni, 2006; Ianni et al., 2003) to rapidly develop new predefined constructs and to deal with compound data structures. Template predicates can be seen as a way to define intensional predicates by means of subprograms,

which are generic and reusable. The DLP^T language based on this notion was implemented on top of the DLV system (Leone et al., 2006).

The semantics of the DLP^T language is given through a suitable *Explode* algorithm, which rewrites a DLP^T program P into an ordinary disjunctive logic program P', by iteratively applying the unfold operation to P: append to P the rewritten code P^s for each template signature s in P, and replace other template atoms a having signature s by suitable atoms a^s. The resulting program P' can be evaluated by DLV and answer sets of P can be extracted from those of P' by looking into atoms belonging to the Herbrand base of P only.

The *Explode* algorithm only terminates if the *dependency graph* encoding dependencies between template atoms and template definition is acyclic. Determining this property can be done in polynomial time. Furthermore, P' is polynomially larger than P, hence DLP^T keeps the same expressive power as *DLP*, which guarantees that DLP^T program encodings are as efficient as plain *DLP* encodings.

Comparing the macro/ensemble (Baral et al., 2006) approach to the template approach (Calimeri and Ianni, 2006; Ianni et al., 2003), we can notice that both exploit the idea of wrapping a piece of reusable logic programming rules into a unit which can be referred to in another place. While macro calls to modules are separated to the rules in the programs, template predicates appear in the bodies of the rules. Both serve at a schematic level, i.e., arguments in their definitions can be replaced by actual predicates and terms in each call. For the semantics, both approaches provided translations to ordinary ASP encoding under the same condition that the calling relationship in their settings must be acyclic. The macro call technique by Baral et al. (2006) went one step further and provided the notion of ensembles for grouping similar modules together and allowing inheritance of modules.

Modular ASP Programs A different approach is used by Tari et al. (2005). Here, the modules allow to import answer sets from other modules to compute the overall solution using different types of reasoning modes resembling cautious and brave reasoning, and an ordering reasoning mode. However, this approach considers only modular ASP programs with acyclic dependency graph.

RSig Another system called RSig (Balduccini, 2007) allows to specify modules and provides an information hiding mechanism. Instead of substantial redesigning of an ASP language, RSig aims at extending A-Prolog, in particular the LPARSE syntax, to fulfill two requirements: modules exchange information with a global state via import/export declarations, signature declarations of predicates, and function symbols. Logic programs in RSig can be composed of regular rules at a global level, regular directives, signature declarations, and module definitions. Each module has a unique name and should consist of *#import*, *#export* directives, and regular rules. Predicate

and function symbols in a module are *local* by default. If they appear in an #*import* or #*export* directive then they are *global*; in this case, their truth values must coincide with the ones with the same name in the global rules.

The semantics of RSig programs is given by a mapping of them to the language of LPARSE in two steps: eliminating module definitions, and introducing explicit typing for arguments of functions and predicates with signature declarations. The former is of our interest. It is done based on module-elimination of function and predicate symbols in each module, by adding to the local ones the module name as a prefix, while keeping the global ones as they were. Module-elimination for rules is straightforward as replacing, in each rule, its components (function and predicate symbols) by the module-eliminated ones; and the whole module is replaced by the set of its module-eliminated rules. After this step, we get a mixed program where local symbols from module definitions are distinguished from global ones by the prefixes. After adding explicit typing predicates for the signature declarations, the result is a usual LPARSE program and can be evaluated to give the global answer sets of the program (if any).

Nested HEX-Programs Eiter et al. (2013) use *external atoms* to access the answer sets of a subprogram. There are three basic external atoms:

- &*callhexfile*[`prog`](P), which takes subprogram `prog` as input filename and assigns it a handle P;
- &*answersets*[P](A), which takes a subprogram handle P and returns a sequence counter A, one for each answer set of P; and
- &*arguments*[P, A, Q](I, N, V), which takes a subprogram handle P, an answer set counter A, and a predicate name Q as input and receives a literal identifier I, predicate argument position N, and the constant value V at the N^{th} position of $Q(t_1, \ldots, t_k)$.

Take, as an example, the following HEX-program consisting of three rules that have access to subprogram `paths`, which stores in *node* all the nodes of fixed graph G, while *path* keeps all paths of G.

$$
\begin{aligned}
h(P, A) &\leftarrow \&callhexfile[\texttt{paths}](P), \&answersets[P](A) \\
v(X) &\leftarrow h(P, A), \&arguments[P, A, node](I, 0, X) \\
t(X, Y) &\leftarrow h(P, A), \&arguments[P, A, path](I, 0, X), \&arguments[P, A, path](I, 1, Y)
\end{aligned}
$$

The first rule imports pairs $h(P, A)$, which uniquely identify each answer set of program `paths`. The second rule retrieves, based an answer set A of P, the extension of

node in A as v. Similarly, the third rule takes all paths $path(X, Y)$ from A and assign them to t.

This approach only allows for nested programs without module recursion, as this would quickly lead to an infinite chain of external function calls. In order to support module input, nested HEX uses external atoms of the form

$$\&simulator_{n,m}[\texttt{prog}, p_1, \dots, p_n](X_1, \dots, X_m) \ ,$$

which calls stratified subprograms `prog` with additional facts $in_i(a_1, \dots, a_{k_i})$ as input for each atom $p_i(a_1, \dots, a_{k_i})$ to be true in an answer set of the calling program, while the result of this called external atom stems from atoms $out(a_1, \dots, a_m)$ that can be derived in the unique answer set of `prog`.

10.3 Modules as Splitting Sets and Related Techniques

Alternative to the approaches mentioned in §10.1 are the techniques mentioned in this section, which focus on syntactic notions of modularity.

Splitting Sets and Modularity Properties For nonmonotonic logic programs, Lifschitz and Turner (1994) defined splitting sets as a means to partition a program into (a chain of) independent subprograms such that they can be evaluated in an ordered way. In the context of disjunctive Datalog, Eiter et al. (1994, 1997a) independently defined modularity properties for disjunctive programs under stable model semantics, which are equivalent to the splitting set technique. Both generalize stratified programs pioneered by Apt et al. (1988), upon which the results in §4.3 build upon. A *splitting set* U is a subset of the Herbrand base such that for all rules r of a program P, if the head of r contains literals in U then the literals of r must be contained in U. Such U then intuitively divides a logic program P into two parts: the *bottom* $b_U(P)$ containing all rules from P whose literals are contained in U, and the *top* $P \setminus b_U(P)$. The *Splitting Set Theorem* allows one to compute the answer sets of a given program P using the top and bottom parts independently, i.e., by evaluating the top part for each answer set of the bottom part. In the Datalog setting, Eiter et al. (1997a) define that π_1 is *independent* to π_2 (denoted $\pi_2 \rhd \pi_1$), where π_1 and π_2 are two programs, whenever all predicates of the heads of the rules in π_2 do no occur in any rule of π_1. Then, equivalently to the splitting set theorem, Eiter et al. (1997a, Lemma 5.1) allows to obtain the answer sets of the combined program $\pi = \pi_1 \cup \pi_2$ incrementally by adding, for each answer set M of π_1, the set M as facts to π_2 and by computing the answer sets of program $\pi_2 \cup M$. Dao-Tran et al. (2009b) extended the notion of splitting sets to MLPs, which splits module instantiations with respect to module calls. Here, splitting sets are used

to get a sequence of module instantiation such that we can compute the answer sets of an MLP hierarchically. Chapter 8 discusses the results and provides examples.

Vennekens et al. (2006) use the framework of approximation fixpoint theory to characterize stratification and splitting of fixed point operators similar to the T_P operator for ordinary Horn programs P (Emden and Kowalski, 1976; Lloyd, 1987). Our fixed-point characterization of MLPs defined in §4.2 may use the results by Vennekens et al. (2006) to obtain a splitting for operator $T_{\mathbf{P}}$ for MLPs $\mathbf{P}$.

Ji et al. (2015) define strong splitting as an extension to splitting sets, allowing to use non-trivial strong splitting sets to split a disjunctive logic program, where standard splitting sets would only allow to use trivial sets to split a program (i.e., the empty set and the Herbrand base). This technique comes at the cost of introducing new atoms to the program, but experimental results suggest that this technique may be valuable to efficiently evaluate a logic program by strong splitting.

Generalizations of the splitting sets technique for semantics related to the stable model semantics are investigated by Eiter et al. (1997c) for partial stable models, which approximate total stable models, and by Amendola et al. (2016) for semi-equilibrium semantics, a paracoherent semantics for answer set programs. Both approaches explore the modularity properties in their respective semantics, thus paving the way for modularity in alternative stable model semantics. As to be discussed in Chapter 11, semi-equilibrium semantics could prove to be useful to define a less strict semantics in the context of MLPs in non-relevant module instantiations.

Magic Sets, Independent Sets, and Modules In the context of data integration, Faber et al. (2007) developed an optimization technique based on magic sets (Bancilhon et al., 1986) for query answering over Datalog programs with unstratified negation. To this end, Faber et al. (2007) defined *dangerous predicates* that occur in an odd cycle in the dependency graph of a Datalog program or in the body of a rule with dangerous head. Rules that contain dangerous predicates are also called dangerous, and based on dangerous rules, they defined *independent sets*, which induce *modules*, i.e., subsets of the given program. Then, based on a module, one can partition a Datalog program P into a module-part T and the program-part without T, i.e., $P \setminus T$, and whenever P is consistent, then the stable models of T are in one-to-one correspondence to the stable models of P restricted to the signature of T. In general, each stable model of P restricted to the signature of T is a stable model of T. This property allows one to obtain that answering a cautious or brave query Q on P is equivalent to query answering Q on T, whenever Q is defined by the rules of T and P is consistent. Since T is usually smaller than P, this should give a performance boost for query answering algorithms.

Symmetric Splitting and Module Theorem for the General Theory of Stable Models Ferraris et al. (2009) build upon the idea of the Module Theorem and de-

fine symmetric splitting in the general theory of stable models, which gives a stable model semantics for first-order theories based on the $\mathrm{SM}_{\mathbf{p}}[F]$ stable model operator defined by Ferraris et al. (2011). The Symmetric Splitting Theorem provides the solid background to split a first-order theory $F \wedge G$ (consisting of two sentences F and G) and obtain that $\mathrm{SM}_{\mathbf{pq}}[F \wedge G]$ is equivalent to $\mathrm{SM}_{\mathbf{p}}[F] \wedge \mathrm{SM}_{\mathbf{q}}[G]$, whenever $\mathbf{p}$ and $\mathbf{q}$ obey syntactic restrictions on the predicate dependency graph for F and G.

Babb and Lee (2012) extend the symmetric splitting theorem and define suitable joinability conditions for *first-order modules* to show a Module Theorem for the general theory of stable models akin to the Module Theorem for DLP-functions.

10.4 Equivalence Notions for Modular Logic Programming

Several notions for checking equivalence of two programs or two modules have been studied in the past, which relates to modular logic programming as a tool to check whether two alternative modules behave in the same expected way. When two modules have shown to be equivalent, one module may be replaced by the other for, e.g., efficiency reasons or whether a supposedly innocent local change in a module has ill consequences.

Janhunen and Oikarinen (2007) are concerned with checking *visible equivalence* of two SMODELS logic programs P and Q that agree on visible atoms, i.e., atoms that are considered to be relevant to represent solutions, while hidden atoms are used for representing auxiliary relations. Visible and hidden atoms are disjoint, and their union determines the Herbrand base of P or Q. Given P and Q, P is said to be visible equivalent to Q if both have the same visible atoms and there is a bijective map f that translates every stable model M of P to a stable model $f(M)$ of Q such that the visible atoms contained in M are equal to the visible atoms of $f(M)$. The task of verifying visible equivalence of P and Q is identical to checking unsatisfiability of the faithful translations $\mathrm{EQT}(P, Q)$ and $\mathrm{EQT}(Q, P)$, which send P and Q to appropriately prepared SMODELS programs.

This approach has shown to be useful not only for ordinary logic programs, but also in the context of the compositional ASP module approach (Janhunen et al., 2009b; Oikarinen, 2008). Oikarinen and Janhunen (2009) study the problem of module equivalence, i.e., given two SMODELS modules $\mathbb{P}$ and $\mathbb{Q}$, they are called modular equivalent if they agree on their input atoms and $\mathbb{P}$ and $\mathbb{Q}$ are visible equivalent (where the definition for visible equivalence is adapted from ordinary logic programs).

10.5 Distributed and Heterogeneous Knowledge Bases

Related to modularity is research on hybrid and distributed knowledge bases, i.e., knowledge bases that consists of multiple potentially different logical frameworks. Here, instead of combining multiple subprograms to a coherent modular logic program, one is concerned with putting together independent knowledge bases that may not reside on the same machine, potentially using different logical frameworks.

MWeb Analyti et al. (2011) proposed a modular framework for web rule bases (`MWeb`), mainly concerning the support for hidden knowledge and the safe use of strong and weak negation in the Semantic Web. Such an `MWeb` modular rule base $\mathcal{S}$ is a set of `MWeb` rule bases. Each rule base $s \in \mathcal{S}$ can import knowledge for a predicate p from other rule bases in $\mathcal{S}$ that define p and are willing to export this knowledge to s. Each predicate p defined or imported by a rule base is associated with a reasoning mode (appearing in increasing order), *definite* (weak negation is not accepted at all), *open* (only open-world assumptions are accepted), *closed* (both closed-world and open-world assumptions are accepted), or *normal* (weak negation is fully accepted). Moreover, a rule base can assign to its predicates a scope, *global*, *local*, or *internal*. These scopes indicate, respectively, that p is either allowed to be redefined, allowed to be used but not redefined by, or is invisible to other rule bases. The import-dependencies of the rule bases must be acyclic.

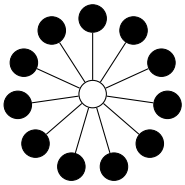
Heterogeneous Nonmonotonic Multi-Context Systems The seminal paper by Brewka and Eiter (2007) instantiated a line of research for *Heterogeneous Nonmonotonic Multi-Context Systems (MCS)*. Their approach is based on *equilibrium semantics*, which allows to combine heterogeneous and independent knowledge bases (the contexts) into a multi-context system using nonmonotonic bridge rules to model the information flow in such an MCS. Dao-Tran (2014) and Dao-Tran et al. (2015) developed algorithms and optimization techniques for the distributed evaluation of MCS knowledge bases, which includes model streaming techniques for concurrent evaluation of MCS (Dao-Tran et al., 2011). To this end, they defined partial equilibria at a certain context in an MCS that can be used to incrementally build up the (global) equilibrium semantics of an MCS in a distributed system setting. On the semantic level, Dao-Tran and Eiter (2017) extend the equilibrium semantics for asynchronous continuous computations by introducing window atoms that take a snapshots of input streams in a context (such as sensory data) and operate on a closed interval of time points with associated sets of atoms.

Modular Model Expansion and Modular Systems Tasharrofi (2013), Tasharrofi and Ternovska (2014), and Ternovska (2015) devise a modular framework based on

model expansion called *modular model expansion*, by viewing individual modules as model expansion tasks and therefore are able to give a semantics to various heterogeneous logics using their modular framework. Based on model expansion tasks, the modular system combines individual modules by suitable operators such as projection, composition, union, and feedback, to form a newly integrated knowledge base from individual constituents.

Abstract Modular Inference Systems Lierler and Truszczyński (2016) define a framework for *abstract modular inference systems (AMS)* that can model multi-logics system comprising logics with different semantics by abstracting their model generation algorithms into *transition graphs* pioneered by Nieuwenhuis et al. (2006). AMS are capable of expressing modular logic programs by Lierler and Truszczyński (2013), an approach to modularity that generalizes lp-modules by Oikarinen and Janhunen (2009).

Discussion The main difference of MLPs to the heterogeneous formalisms shown here is that MLPs only allow to modularize knowledge expressed with nonmonotonic disjunctive logic programs, whereas the latter allow to interlink a variety of logical knowledge bases, possibly with diverse semantics, into a combined and coherent knowledge base. The main distinction here is that MLPs enable the exchange of relations, i.e., we may supply an MLP module with relational input computed by a module.



11

Conclusion

Concluding the work, we have studied modular nonmonotonic logic programs (MLPs), a new approach to modular answer set programs under a call by value mechanism. We have defined a model theoretic semantics and studied its semantic properties. One of the outcomes is that MLPs allow to express the Even property using monadic relations, and we have shown that this is not possible with ordinary answer set programs by expressing answer set existence of Answer Set Programs with unary relations in Monadic Second Order Logic. We have defined syntactic fragments of MLPs that admit a unique model, namely Horn MLPs and stratified MLPs, and showed how to compute their answer sets through (iterated) fixed-point computation using appropriate consequence operators. We have investigated the computational costs for various classes of MLPs and provide completeness results for the computational complexity of deciding whether a ground atom is contained in the least-fixpoint of a Horn MLP as well as answer set existence for propositional and nonground MLPs for various classes of MLPs.

Towards translating MLPs with module input into equivalent MLPs without input we have explored rewriting techniques to reformulate MLPs into programs with simpler structure. Horn MLPs without input may be pruned module-by-module and turned into ordinary logic programs with macro rewriting techniques, and using this method paves the way to translate Datalog-rewritable Description Logic Programs into MLPs iteratively. We have discussed the essential ideas for relevance-driven evaluation of MLPs and provide experimental results for an MLP benchmark scenario based on the macro rewriting techniques applied to Datalog-rewritable Description Logic Programs.

Then, we have characterized the answer sets of MLPs in terms of classical models and developed propositional logic encodings using loop formulas for ground normal MLPs, and we have formulated first-order logic encodings over finite structures for nonground normal MLPs using ordered completion. For this purpose we have expressed the MLP semantics as second-order logic sentences over finite structures. Fur-

thermore, we have analyzed the relationship between MLPs and DLP-functions using two translations, one for rewriting a sequence of DLP-functions into an equivalent MLP without input, and one for rewriting an MLP without input into a sequence of DLP-functions, provided that syntactic restrictions on the MLP are in order similar to the ones for standard DLP-functions.

Next, we summarize the findings of this thesis in more detail in §11.1, report outlook and open issues and further research directions in §11.2.

11.1 Summary

The framework we have defined for writing modular logic programs is based on disjunctive logic programs that consist of modules of the form $(P[\mathbf{q}], R)$, whose module name P has a list of input predicates $\mathbf{q} = q_1, \ldots, q_k$ and an associated set of disjunctive rules R, as a way to structure logic programs. Modules admit input $\mathbf{q}$ as facts provided by other modules, which in spirit resembles a call by value regimen traditionally found in imperative programming. One module may access the truth value of atoms defined by another module by using module atoms α of the form $P[p_1, \ldots, p_k].o(c_1, \ldots, c_n)$ in the rule bodies, with the intuitive meaning that α is true whenever $o(c_1, \ldots, c_n)$ is true in an instantiation of the module P identified by *value call* $P[S]$ given the extension S of predicates $p_1, \ldots, p_k$ from the calling module as facts. In essence, this allows that one module may call other modules and additionally provide input. The rules given in modules have no essential restriction and modules may recursively call each other with additional input.

In the following, we will emphasize the results of this thesis in more detail.

11.1.1 Model Theoretic Semantics and Semantic Properties of MLPs

We have defined a model theoretic semantics for a system $P_1[\mathbf{q}_1], \ldots, P_n[\mathbf{q}_n]$ of program modules consisting of at least one main module P_i without input (i.e., $\mathbf{q}_i$ is void), and library modules that may have input (i.e., $\mathbf{q}_i$ can be void). The semantics assigns an answer set to each main module and module instance that is called by the program under a call by value mechanism (Eiter et al., 1997b). The answer set must be reproducible from the rules along its recursive computation.

The declarative semantics for MLPs provides a solution for the situation where modules may introduce cyclic input, i.e., where modules are allowed to call each other recursively. The framework uses module instantiations in form of a *call graph* to abstract from the computational view of module calls. Such call graphs are in the spirit of Kripke-style semantics, where we use *value calls* as input instead of Kripke worlds, and the call graph of an MLP can be thought as the accessibility relation in a Kripke

frame, which is defined by the MLP. The semantics can be contextualized, i.e., we defined a *context-based reduct* and *MLP answer sets* based on the FLP-reduct (Faber et al., 2011) relative to a set of value calls (the *context*).

We have studied the semantic properties of MLP semantics showed that many of the desired properties of ordinary logic programs generalize to MLPs: the answer sets of a positive MLP are minimal models, Horn MLPs have a least model computable by least fixpoint iteration, and stratified MLPs have a canonical model that coincides on relevant instances.

11.1.2 Computational Complexity of MLPs

We have characterized the computational complexity of the new formalism using alternating Turing machine simulations and domino tiling problems. The results are classified along various syntactic restrictions on the form of MLPs. Deciding whether a modular nonmonotonic logic program has an answer set has the same complexity as for ordinary answer set programs (the propositional case is complete for Σ_2^p and the nonground case is complete for NEXP^{NP}), provided that the modules have no input. MLPs with unrestricted input have computational costs that are exponentially higher: NEXP^{NP}-complete for propositional MLPs and 2NEXP^{NP}-complete for nonground MLPs. Modular logic programs by Eiter et al. (1997b) have EXPSPACE complexity, which is believed to be strictly contained in 2NEXP^{NP}, thus MLPs are more expressive than Modular Logic Programs as GQLPs. Analog to the unrestricted case, deciding whether an atom is contained in the least model of a Horn MLP without input has the same complexity as an ordinary Horn logic program (P-complete for propositional MLPs and EXP-complete for nonground MLPs, respectively). With unconstrained input, we have obtained tight bounds for propositional Horn MLPs (EXP-complete) and Horn MLPs in the Datalog setting (2EXP-complete). If module input of an MLP is bounded by a constant, then the computational complexity is unchanged compared to ordinary ASP and to MLPs whose input is void. We have defined acyclic MLPs as propositional normal MLPs whose call graph does not contain cycles and found that deciding whether acyclic MLPs have an answer set is a NEXP-complete problem, just like propositional normal MLPs.

In essence, whenever we have MLPs with bounded predicate input or disallow module input, we obtain the same computational costs as ordinary ASP. Unconstrained module input shifts the complexity by one level from the weak exponential hierarchy to the weak double exponential hierarchy in case of nonground MLPs, and from the polynomial hierarchy to the weak exponential hierarchy for propositional MLPs; this holds even for the case of acyclic propositional MLPs. The problem whether a propositional acyclic or a propositional normal MLP has an answer set is one the first level of the exponential hierarchy, whereas answer set existence for unrestricted propositional MLPs is on the second level, which is believed to contain harder computational

problems. The same is true for nonground MLPs, just one exponential higher on the double exponential hierarchy.

11.1.3 Rewriting MLPs to Datalog

We have developed two rewriting techniques for translating MLPs with module input into programs of simpler structure. Our results operate on MLPs in a certain canonical form, i.e., MLPs having at most one module input predicate and only one module with empty input, which by definition will be the main module. The first restriction can be accomplished by module reification, and the second one by introducing a fresh main module that collects empty input modules. Every MLP can be rewritten into its canonical form without much overhead, i.e., at most linear fresh rules need to be introduced for the reification, and the answer sets of the unrestricted MLP match one-to-one to the answer sets of the canonical MLP.

The first Datalog rewriting technique, where we translate arbitrary MLPs to ones without module input, aims at transforming MLPs into logic programs without modules at all. To this end, we have studied *instance rewriting* as a first step that creates copies of modules with input to modules without input by introducing fresh predicate arguments as a means to keep track of module input. The result will be a larger MLP with a *shadow* copy that preserves the call structure of the original part. Then, the next step introduces *call rewriting* to separate the cloned part from the original part such that the original modules will not be called anymore and can be readily removed. The final step will apply *module removal of connected closed call sets*, i.e., the original part will be removed, and what is left is an MLP without input. We have shown that the answer sets of the instance rewriting and the call rewriting are in one-to-one correspondence, and once we have taken off the shadow part introduced by instance rewriting by using module removal, we have a correspondence to the original MLP. This approach is costly in general and may generate exponentially larger programs, as we need to blow up the arity of the predicates in a program.

11.1.4 Macro Expansion for MLPs

The second approach to rewriting MLPs is called *macro rewriting* and it converts a restricted syntactic class of MLPs into MLPs of simpler structure, i.e., Horn MLPs whose call chain is acyclic. The purpose of macro rewriting is to avoid the exponential blow-up introduced by instance rewriting and convert a given MLP into one without modules. Macro rewriting works step by step along the *module connection graph* by copying the rules of a module callee into the module caller. Once all modules have been copied, we may apply module pruning and remove unneeded modules from the MLP. This technique works incrementally, i.e., we may rewrite only those parts of an MLP that are Horn and acyclic, or rewrite just a subset of the modules.

Based on macro rewriting, we have developed an application for MLPs: Datalog-rewritable Description Logic Programs (dl-programs), a hybrid knowledge representation formalism that combines description logics with logic programs. We have shown how to encode dl-programs over a Datalog-rewritable Description Logic as an MLP. Since the outcome is an acyclic Horn MLP for the modules that implement the Description Logic part, we may apply macro rewriting and simplify the structure. Eventually, we may use an off-the-shelf Datalog reasoner for evaluating the result.

Using the TD-MLP reasoner (Wijaya, 2011) we have created a dl-program benchmark based on LUBM (Guo et al., 2005) and compared our encoding for dl-programs into MLPs with the DReW rewriting scheme (Xiao et al., 2013). The outcome showed that DReW is superior to the MLP encoding with modules, but increasing the number of dl-atoms decreases the competitive edge of DReW. The results suggest to further the work on MLP implementations and optimize them for practical use.

11.1.5 Modular Loop Formulas

We have characterized the answer set semantics of normal MLPs by the classical models of propositional theories, i.e., by using *loop formulas* (Lin and Zhao, 2004) and *program completion* (Clark, 1978) for normal ground MLPs. We have shown how to translate a normal MLP with module input into a formula in classical propositional logic by recasting Clark's program completion for module atoms and MLP's module input mechanism. To address dependencies of module boundaries, we have defined the *modular dependency graph*, which is a positive dependency graph for MLPs that captures the dependencies between the rules within a particular module, and the (uninstantiated) coarse dependencies between the modules of an MLP. Using the notion of *cyclic instantiation signatures*, we can instantiate the dependencies of the modular dependency graph and based on them we devise the *modular loops* of an MLP for generating *modular loop formulas*, which lie at the heart of the characterization.

The size of the loop formula encoding for an MLP is double exponentially larger in general. There are two sources of additional complexity that need to be handled: one is the intrinsic complexity of the answer set semantics for ordinary logic programs (in general, one needs to administer exponentially many loop formulas), and the other is the MLP input mechanism, which requires to address exponentially many module instantiations.

11.1.6 Ordered Modular Completion

In order to shape the answer sets of nonground normal MLPs using classical first-order formulas, we further the work on *ordered completion* and investigate the approach of Asuncion et al. (2012) to the MLP setting. To this end, we have defined finite relational structures for MLPs, which in the MLP Datalog setting provide the means to

ground the rules. To prove our results, we have defined a *translational semantics* for MLP using second order logic and showed that the second order sentence captures the models of MLPs. The ordered completion formula makes use of *ordered modular derivation*, which plays the role of forcing a derivation order on the ground atoms in a model of an MLP. The ordered completion for MLPs is essentially the conjunction of an adapted ordered program completion and *ordered transitive derivation*. Using the result of the translational semantics and derivation orders, we have shown that ordered completion captures the answer sets of MLPs.

11.1.7 Relationship between DLP-Functions and MLPs

We have analyzed the interrelationship between MLPs and DLP-functions (Janhunen et al., 2009b), a formalism that proved to be appealing for modular ASP. As one result, we have shown that DLP-functions can be straightly embedded into an MLP with empty input list. The second result shows that the fragment of *mutually independent* MLPs can be converted into a sequence of DLP-functions. Hence, we can view MLPs as a generalization of DLP-functions, as MLPs permit positive loops over modules, possibly under a call by value scheme.

11.2 Open Issues and Further Research Directions

While we have presented here the basic approach for modular nonmonotonic logic programming with a call by value semantics, several issues remain open as prospective research questions and directions. In this section, we will outline them and provide further pointers we deem advisable as attractive topics of research in the context of modularity in logic programming.

11.2.1 Formal Semantics

An interesting issue is to further analyze contexts and, e.g., to determine conditions for contexts that are fully stable, which desirably should be small. Some (less effective) conditions may be determined by syntactic analysis. To keep the semantics simple, MLPs use minimal models as an approximation of answer sets in module instances that are outside of a *context* (i.e., a *scope*), in which stability of models is strictly required. This context contains always at least the module instances along the call graph of the program and optionally further instances to increase in a sense the degree of stability. The smaller the context, the more permissive is the semantics. An alternative to using minimal models for ensuring consistency would, e.g., be to use paracoherent answer set semantics (Amendola et al., 2016; Sakama and Inoue, 1995); however the latter has

higher computational complexity than ordinary answer set semantics. On the semantic side, we can imagine alternative ways of tolerating violations of stability outside the context. This could be done, e.g., by using partial FLP-reducts (where not all rules with false bodies are dropped, leading to a superset of the answer sets), or by genuine approximations. Variants of stratification and splitting sets would also be interesting. Another line of research is to improve the understanding of MLP semantics and give it a logical foundation using (generalized) equilibrium logic (Pearce, 1997) and applying results on FLP-semantics (Truszczyński, 2010). Furthermore, one may relax the restriction to minimal models in nonrelevant instantiations and use semi-equilibrium models (Amendola et al., 2016) in the context of MLPs instead.

11.2.2 Extensions and Fragments of MLPs

Another issue is extensions of MLP to richer classes of programs, including constructs like strong negation, constraints, external functions, etc. Going into a different direction, research on further useful fragments of MLPs and characterize their computational complexity in order to find MLPs whose complexity is lower than the standard semantics is worthwhile. On the computational side, a detailed complexity study of MLPs that considers various fragments is of interest, where in particular the interplay of major classes of ordinary logic programs with dependency information through module calls deserves attention; various notions similar to the ones mentioned by Eiter et al. (1997b) might be considered here. Furthermore, efficient methods and algorithms to compute answer sets of MLPs remain to be developed, as well as implementations.

The approach by Moura (2016) shows how to modify the framework of modular SMODELS programs (Oikarinen and Janhunen, 2008) such that positive recursion over modules is possible. Based upon the work in Chapter 9, one may find ways to extend this approach to disjunctive logic programs similar to DLP-functions. The resulting approach will be less expressive than general MLPs defined here, as DLP-functions and their cyclic companions do not admit input values.

11.2.3 Implementation

Algorithms that efficiently implement the semantics of MLPs are required to animate Modular Nonmonotonic Logic Programming and help answer set programmers build modular programs. The work by Dao-Tran et al. (2009b) was an early attempt towards a practical implementation for MLP, and Wijaya (2011) provided the first implementation. There are several further aspects that may be considered for implementation strategies.

Based on modular loop formulas and ordered modular completion one may develop new algorithms that interweave methods for conflict-driven model building (see Gebser et al., 2012; Marques Silva et al., 2009, for background) and module instantiation.

Related to this is to investigate first-order theorem proving techniques (Robinson and Voronkov, 2001) in the context of MLPs.

In direction of divide and conquer algorithms, Ferraris et al. (2009) present Symmetric Splitting as a generalization of the Module Theorem (Janhunen et al., 2009b; Oikarinen and Janhunen, 2008) allowing to decompose also nonground programs like MLPs do. This technique is only applicable to programs without positive cycles in the dependency graph. Studying the relationship between Symmetric Splitting and our notions of stratification is an interesting subject for future work.

Several further issues remain open, including extensions and refinements of the stratification approach (Dao-Tran et al., 2009b). The relevance-driven evaluation approach there has focused on decreasing inputs in terms of set inclusion, thus the extension of the method to other partial orderings of inputs having bounded decreasing chains is suggestive.

11.2.4 Loop Formulas and Ordered Completion

As for future work, refinement of the results and exploitation of the results for answer sets computation using SAT and QBF solvers, as well as theorem provers remains to be investigated; here, fragments of MLPs that allow for reasonable encodings might be considered, and the suitability of higher-order theorem provers evaluated.

For this thesis, we have fixed the context C in the context-based FLP-reduct to the set $\mathrm{VC}(\mathbf{P})$ of all value calls, which thus can be omitted. Intuitively, a given context C may be incorporated by ensuring that loop formulas are built only for relevant instantiation signatures $(\mathcal{S}_1, \ldots, \mathcal{S}_n)$; for modular loops, which are those that contain some value call $\mathcal{S}_i$ inside C; furthermore, either none or all value calls $\mathcal{S}_i$ must be in C. Relative to an interpretation $\mathbf{M}$, the minimal context $C = V(\mathrm{CG}_{\mathbf{P}}(\mathbf{M}))$ may be defined using suitable formulas. The technical elaboration of these ideas is beyond this thesis.

Unfounded sets for logic programs with arbitrary aggregates have been defined by Faber (2005). Given that for ordinary logic programs unfounded sets are a semantic counterpart of loop formulas, this may inspire a similar notion of unfounded set for MLPs and help developing a syntactic counterpart in terms of loop formulas. The work by Lee and Meng (2009) and Truszczyński (2010), which inspect the FLP-semantics on a more principled level, may also be useful in this respect. Different from answer semantics under the GL-reduct, not only positive atoms need to be considered for derivability, but also negated nonmonotonic module atoms.

A further issue are encodings for disjunctive MLPs, i.e., MLPs where the head of a rule may be a disjunction $\alpha_1 \vee \cdots \vee \alpha_k$ of atoms. Loop formulas have been developed for ordinary disjunctive logic programs (Lee and Lifschitz, 2003), and for general propositional theories under Answer Set Semantics (Ferraris et al., 2006). There is no principal obstacle to extend the loop formula encoding of this thesis to disjunctive MLPs. In contrast, ordered completion formulas for disjunctive MLPs and already ordinary

LPs needs further work; they may require a blowup given that ordinary disjunctive Datalog programs have NEXP^{NP} complexity. Janhunen (2004) provides an alternative to loop formulas, which is an encoding to propositional formulas whose size remains in the order of $n \log_2 m$ in the length n of the logic program and the number of atoms m. This could be used to tackle the potentially exponential number of loop formulas.

Finally, relationships to other semantics of logic programming is an interesting issue. X. Chen et al. (2013) showed that loops with at most one external support rule in the program have a close connection to (disjunctive) well-founded semantics. Studying MLPs under similar restrictions could provide similar results, yet well-founded semantics for MLPs remains to be formalized.

The work of Asuncion et al. (2015, 2012) also present an interesting approach to implement answer set semantics for normal logic programs using SMT solvers (Barrett et al., 2009; Nieuwenhuis et al., 2006), which is based on reductions of answer set programs to *difference logic* by Janhunen et al. (2009a) and Niemelä (2008). Here, the derivation order predicates from the ordered completion mimic the level ranking from Niemelä (2008), thus standard SMT solver are readily utilizable to compute the answer sets of normal logic programs. Similar techniques can be used to implement MLPs, although the ordered completion for MLPs have usually larger formulas. Related to the SMT approach is work on SAT modulo acyclicity (Gebser et al., 2014b), which allows to implement answer set semantics using a translation shown by Gebser et al. (2014a). Experimental evidence suggests that the translation into a SAT modulo acyclicity problem gives a performance gain over the SMT translation. One downside of this translation is that the one-to-one correspondence between answer sets of the logic program and the models of the SAT modulo acyclicity encoding may be lost. Using dedicated algorithms that enumerate projections of satisfying assignment can circumvent this issue.

11.2.5 Modular Patterns for Logic Programming

A different line of research that deserves attention is to establish a firm foundation for practical programming techniques and design principles in spirit of software design pattern languages. Some recurring patterns have been described by Eiter et al. (2009b), but no support for modularity patterns is shown there. An interesting aspect that may provide useful insight is to bring *Liskov Substitution Principle* (Liskov and Wing, 1994) closer to Logic Programming, which states replaceability conditions and requirements for object-oriented programming. In the realm of Logic Programming there is no hierarchy of types and classes, but the basic ideas live on in several equivalence notions for answer set programming (see Eiter et al., 2007b; Woltran, 2008). One consideration in this respect is to find simpler and potentially easier to evaluate code fragments or modules that can be replaced on-the-fly. In the context of modular logic programming, work by Janhunen and Oikarinen (2007), Oikarinen and Janhunen

(2009), and Truszczyński and Woltran (2009) on equivalence notion in the context of modular logic programming should be the starting point for further research.

Bibliography

Abiteboul, Serge, Richard Hull, and Victor Vianu (1995). *Foundations of Databases*. Addison-Wesley (cit. on p. 70).

Abiteboul, Serge and Victor Vianu (1991). "Generic Computation and Its Complexity". In: *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*. STOC '91. New Orleans, Louisiana, USA: ACM, pp. 209–219. ISBN: 0-89791-397-3. DOI: 10.1145/103418.103444 (cit. on p. 23).

Adrian, Weronika T., Mario Alviano, Francesco Calimeri, Bernardo Cuteri, Carmine Dodaro, Wolfgang Faber, Davide Fuscà, Nicola Leone, Marco Manna, Simona Perri, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari (2018). "The ASP System DLV: Advancements and Applications". In: *Künstliche Intelligenz*. First Online: 14 May 2018, p. 3. ISSN: 1610-1987. DOI: 10.1007/s13218-018-0533-0 (cit. on p. 35).

Alexandrescu, Andrei (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley (cit. on p. 16).

Alviano, Mario, Francesco Calimeri, Carmine Dodaro, Davide Fuscà, Nicola Leone, Simona Perri, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari (2017). "The ASP System DLV2". In: *Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings*, ed. by Marcello Balduccini et al. Vol. 10377. Lecture Notes in Computer Science. Springer, pp. 215–221. ISBN: 978-3-319-61659-9. DOI: 10.1007/978-3-319-61660-5_19 (cit. on p. 35).

Alviano, Mario, Carmine Dodaro, Nicola Leone, and Francesco Ricca (2015a). "Advances in WASP". In: *Logic Programming and Nonmonotonic Reasoning*, ed. by Francesco Calimeri et al. Cham: Springer International Publishing, pp. 40–54. ISBN: 978-3-319-23264-5. DOI: 10.1007/978-3-319-23264-5_5 (cit. on p. 35).

Alviano, Mario, Wolfgang Faber, and Martin Gebser (2015b). "Rewriting recursive aggregates in answer set programming: back to monotonicity". In: *Theory and Practice of Logic Programming* 15.4–5, pp. 559–573. DOI: 10.1017/S1471068415000228 (cit. on p. 216).

Amendola, Giovanni, Thomas Eiter, Michael Fink, Nicola Leone, and João Moura (2016). "Semi-equilibrium models for paracoherent answer set programs". In: *Artificial Intelligence* 234, pp. 219–271. ISSN: 0004-3702. DOI: 10.1016/j.artint.2016.01.011 (cit. on pp. 29, 257, 266 sq.).

Analyti, Anastasia, Grigoris Antoniou, and Carlos Viegas Damásio (Jan. 2011). "MWeb: a Principled Framework for Modular Web Rule Bases and its Semantics". In: *ACM Transactions on Computational Logic* 12.2, 17:1–17:46. DOI: 10.1145/1877714.1877723 (cit. on pp. 20, 259).

Apt, Krzysztof R. (1990). "Logic Programming". In: *Formal Models and Semantics*, ed. by Jan van Leeuwen. Handbook of Theoretical Computer Science. Elsevier, pp. 493–574. ISBN: 978-0-444-88074-1. DOI: 10.1016/B978-0-444-88074-1.50015-9 (cit. on p. 12).

Apt, Krzysztof R., Howard A. Blair, and Adrian Walker (1988). "Towards a Theory of Declarative Knowledge". In: *Foundations of Deductive Databases and Logic Programming*, ed. by J. Minker. Washington DC: Morgan Kaufman, pp. 89–148. DOI: 10.1016/B978-0-934613-40-8.50006-3 (cit. on pp. 19, 37, 73, 77, 190, 256).

Armstrong, Joe (2003). "Making reliable distributed systems in the presence of software errors". PhD thesis. The Royal Institute of Technology, Stockholm, Sweden. URL: http://erlang.org/download/armstrong_thesis_2003.pdf (cit. on p. 17).

— (2007). *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf (cit. on p. 17).

Arni, Faiz, KayLiang Ong, Shalom Tsur, Haixun Wang, and Carlo Zaniolo (2003). "The Deductive Database System $\mathcal{LDL}$++". In: *Theory and Practice of Logic Programming* 3.1, pp. 61–94. DOI: 10.1017/S1471068402001515 (cit. on p. 19).

Asuncion, Vernon, Yin Chen, Yan Zhang, and Yi Zhou (2015). "Ordered completion for logic programs with aggregates". In: *Artificial Intelligence* 224.Supplement C, pp. 72–102. ISSN: 0004-3702. DOI: 10.1016/j.artint.2015.03.007 (cit. on pp. 216, 269).

Asuncion, Vernon, Fangzhen Lin, Yan Zhang, and Yi Zhou (2012). "Ordered completion for first-order logic programs on finite structures". In: *Artificial Intelligence* 177–179, pp. 1–24. ISSN: 0004-3702. DOI: 10.1016/j.artint.2011.11.001 (cit. on pp. 30, 33, 184, 199, 201, 204, 215 sq., 265, 269).

Babb, Joseph and Joohyung Lee (2012). "Module theorem for the general theory of stable models". In: *Theory and Practice of Logic Programming* 12.4–5, pp. 719–735. DOI: 10.1017/S1471068412000269 (cit. on p. 258).

Balduccini, Marcello (2007). "Modules and Signature Declarations for A-Prolog: Progress Report". In: *Proceedings of the 1st International Workshop on Software Engineering for Answer Set Programming (SEA'07)*. URL: http://www.depts.ttu.edu/cs/research/documents/61.pdf (cit. on p. 254).

Bancilhon, François, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman (1986). "Magic Sets and Other Strange Ways to Implement Logic Programs (Extended Abstract)". In: *Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. New York, NY, USA: ACM, pp. 1–15. ISBN: 0-89791-179-2. DOI: 10.1145/6012.15399 (cit. on p. 257).

Baral, Chitta, Juraj Dzifcak, and Hiro Takahashi (2006). "Macros, Macro calls and Use of Ensembles in Modular Answer Set Programming". In: *Proceedings of the 22th International Conference on Logic Programming (ICLP 2006)*. LNCS 4079. Springer, pp. 376–390. DOI: 10.1007/11799573_28 (cit. on pp. 20, 253 sq.).

Barrett, Clark, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli (2009). "Satisfiability Modulo Theories". In: *Handbook of Satisfiability: Frontiers in Artificial Intelligence and Applications*. Vol. 185. IOS Press, pp. 825–885. DOI: 10.3233/978-1-58603-929-5-825 (cit. on pp. 3, 269).

Bass, Len, Paul Clements, and Rick Kazman (2013). *Software Architecture in Practice*. 3rd. Addison-Wesley Professional (cit. on p. 10).

Bauters, Kim, Steven Schockaert, Dirk Vermeir, and Martine De Cock (2011). "Communicating ASP and the Polynomial Hierarchy". In: *11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011), Vancouver, BC, Canada, May 16-19, 2011*, ed. by James Delgrande et al. Vol. 6645. LNCS. Springer, pp. 67–79. DOI: 10.1007/978-3-642-20895-9_8 (cit. on p. 20).

Ben-Eliyahu, Rachel and Rina Dechter (Mar. 1994). "Propositional Semantics for Disjunctive Logic Programs". In: *Annals of Mathematics and Artificial Intelligence* 12 (1–2), pp. 53–87. ISSN: 1573-7470. DOI: 10.1007/BF01530761. URL: https://www.ics.uci.edu/~dechter/publications/r25-prop-sem-disj-logic-prog.pdf (cit. on p. 215).

Bienvenu, Meghyn, Balder ten Cate, Carsten Lutz, and Frank Wolter (2013). "Ontology-based Data Access: A Study Through Disjunctive Datalog, CSP, and MMSNP". In: *Proceedings of the 32Nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '13. New York, New York, USA: ACM, pp. 213–224. ISBN: 978-1-4503-2066-5. DOI: 10.1145/2463664.2465223 (cit. on p. 175).

Biere, Armin, Marijn Heule, Hans van Maaren, and Toby Walsh (2009). *Handbook of Satisfiability: Frontiers in Artificial Intelligence and Applications*. Vol. 185. IOS Press (cit. on p. 3).

Blackburn, Patrick and Johan van Benthem (2007). "Modal Logic: A Semantic Perspective". In: *Handbook of Modal Logic*, ed. by Patrick Blackburn et al. Vol. 3. Studies in Logic and Practical Reasoning. Elsevier, pp. 1–84. DOI: 10.1016/S1570-2464(07)80004-8 (cit. on p. 53).

Blackburn, Patrick, Johan van Benthem, and Frank Wolter, eds. (2007). *Handbook of Modal Logic*. Vol. 3. Studies in Logic and Practical Reasoning. Elsevier.

Blass, Andreas, Yuri Gurevich, and Dexter Kozen (Oct. 1986). "A zero-one law for logic with a fixed-point operator". In: *Information and Control* 67.1–3, pp. 70–90. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(85)80027-9 (cit. on p. 23).

Bogaerts, Bart, Tomi Janhunen, and Shahab Tasharrofi (Apr. 2016). "Declarative Solver Development: Case Studies". In: *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town,*

South Africa, April 25-29, 2016, ed. by Chitta Baral et al. AAAI Press, pp. 74–83. URL: http://www.aaai.org/ocs/index.php/KR/KR16/paper/view/12822 (cit. on p. 58).

Börger, Egon, Erich Grädel, and Yuri Gurevich (1997). *The Classical Decision Problem*. Berlin Heidelberg: Springer (cit. on pp. 79, 81).

Brewer, Eric (Jan. 2012). "CAP Twelve Years Later: How the "Rules" Have Changed". In: *Computer* 45, pp. 23–29. ISSN: 0018-9162. DOI: 10.1109/MC.2012.37. URL: https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/ (cit. on p. 10).

Brewka, Gerd and Thomas Eiter (July 2007). "Equilibria in Heterogeneous Nonmonotonic Multi-Context Systems". In: *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-2007), July 22–26, 2007, Vancouver, British Columbia, Canada*. AAAI Press, pp. 385–390. URL: https://www.aaai.org/Papers/AAAI/2007/AAAI07-060.pdf (cit. on pp. 10, 259).

Brewka, Gerd, Thomas Eiter, and Miroslaw Truszczyński (2011). "Answer Set Programming at a Glance". In: *Communications of the ACM* 54.12, pp. 92–103. DOI: 10.1145/2043174.2043195 (cit. on p. 3).

— eds. (2016). *AI Magazine: Special Issue on Answer Set Programming*. Vol. 37. 3. Editorial pp. 5–6. AAAI Press. DOI: 10.1609/aimag.v37i3.2669 (cit. on p. 3).

Brogi, Antonio, Paolo Mancarella, Dino Pedreschi, and Franco Turini (1994). "Modular logic programming". In: *ACM Transactions on Programming Languages and Systems* 16.4, pp. 1361–1398. ISSN: 0164-0925. DOI: 10.1145/183432.183528 (cit. on pp. 18, 20).

Buccafurri, Francesco and Gianluca Caminiti (2008). "Logic programming with social features". In: *Theory and Practice of Logic Programming* 8.5-6, pp. 643–690. DOI: 10.1017/S1471068408003463 (cit. on p. 20).

Bugliesi, Michele, Evelina Lamma, and Paola Mello (1994). "Modularity in Logic Programming". In: *Journal of Logic Programming* 19–20 (Supplement 1), pp. 443–502. DOI: 10.1016/0743-1066(94)90032-9 (cit. on pp. 18, 20).

Cabeza, Daniel and Manuel Hermenegildo (2000). "The Ciao Module System: A New Module System for Prolog". In: *Electronic Notes in Theoretical Computer Science* 30.3, pp. 122–142. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(05)80105-7 (cit. on p. 19).

Calimeri, Francesco, Davide Fuscà, Simona Perri, and Jessica Zangari (2017). "I-DLV: The new intelligent grounder of DLV". In: *Intelligenza Artificiale* 11.1, pp. 5–20. DOI: 10.3233/IA-170104 (cit. on p. 36).

Calimeri, Francesco and Giovambattista Ianni (2006). "Template programs for Disjunctive Logic Programming: An operational semantics". In: *AI Communications* 19.3, pp. 193–206. URL: http://content.iospress.com/articles/ai-communications/aic373 (cit. on pp. 20, 253 sq.).

Carlsson, Mats and Per Mildner (Jan. 2012). "SICStus Prolog—The first 25 years". In: *Theory and Practice of Logic Programming* 12.1-2, pp. 35–66. DOI: 10.1017/S1471068411000482 (cit. on p. 19).

Carlsson, Richard (2003). "Parameterized modules in Erlang". In: *2nd ACM SIGPLAN Erlang Workshop 2003 (ERLANG'03), Uppsala, Sweden, 29 August, 2003*. ACM, pp. 29–35. DOI: 10.1145/940880.940885 (cit. on p. 17).

Chandra, Ashok K. and David Harel (1982). "Structure and complexity of relational queries". In: *Journal of Computer and System Sciences* 25.1, pp. 99–128. ISSN: 0022-0000. DOI: 10.1016/0022-0000(82)90012-5 (cit. on p. 23).

Chandra, Ashok K., Dexter C. Kozen, and Larry J. Stockmeyer (1981). "Alternation". In: *Journal of the ACM* 28.1, pp. 114–133. ISSN: 0004-5411. DOI: 10.1145/322234.322243 (cit. on pp. 81, 85, 104).

Chen, Xiaoping, Jianmin Ji, and Fangzhen Lin (Feb. 2013). "Computing Loops with at Most One External Support Rule". In: *ACM Transactions on Computational Logic* 14.1, 3:1–3:34. ISSN: 1529-3785. DOI: 10.1145/2422085.2422088 (cit. on p. 269).

Chen, Yin, Fangzhen Lin, Yisong Wang, and Mingyi Zhang (2006). "First-Order Loop Formulas for Normal Logic Programs". In: *Tenth International Conference on Principles of Knowledge Representation and Reasoning 2006*, pp. 298–307. URL: http://www.aaai.org/Papers/KR/2006/KR06-032.pdf (cit. on p. 215).

Clark, Keith L. (1978). "Negation as Failure". In: *Logic and Data Bases*, pp. 293–322 (cit. on pp. 30, 183, 185, 205, 265).

Dahl, Ole-Johan, Edsger W. Dijkstra, and C. A. R. Hoare, eds. (1972). *Structured Programming*. Academic Press Ltd. (cit. on p. 11).

Damásio, Carlos Viegas and João Moura (2011). "Modularity of P-Log Programs". In: *11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011), Vancouver, BC, Canada, May 16-19, 2011*, ed. by James Delgrande et al. Vol. 6645. LNCS. Springer, pp. 13–25. DOI: 10.1007/978-3-642-20895-9_4 (cit. on p. 20).

Dantsin, Evgeny, Thomas Eiter, Georg Gottlob, and Andrei Voronkov (2001). "Complexity and Expressive Power of Logic Programming". In: *ACM Computing Surveys* 33.3, pp. 374–425. DOI: 10.1145/502807.502810 (cit. on pp. 81, 86–88).

Dao-Tran, Minh (Feb. 2014). "Distributed Nonmonotonic Multi-Context Systems: Algorithms and Efficient Evaluation". PhD thesis. Karlsplatz 13, 1040 Vienna, Austria: Vienna University of Technology. URL: https://repositum.tuwien.ac.at/download/pdf/1634355 (cit. on p. 259).

Dao-Tran, Minh and Thomas Eiter (2017). "Streaming Multi-Context Systems". In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pp. 1000–1007. DOI: 10.24963/ijcai.2017/139 (cit. on pp. 10, 259).

Dao-Tran, Minh, Thomas Eiter, Michael Fink, and Thomas Krennwallner (July 2009a). "Modular Nonmonotonic Logic Programming Revisited". In: *25th International Con-*

ference on Logic Programming (ICLP 2009), Pasadena, California, USA, July 14–17, 2009, ed. by Patricia M. Hill et al. Vol. 5649. LNCS. Springer, pp. 145–159. ISBN: 978-3-642-02845-8. DOI: 10.1007/978-3-642-02846-5_16. URL: http://www.kr.tuwien.ac.at/staff/tkren/pub/2009/iclp2009-mlp.pdf (cit. on p. 32).

Dao-Tran, Minh, Thomas Eiter, Michael Fink, and Thomas Krennwallner (Aug. 2009b). "Relevance-driven Evaluation of Modular Nonmonotonic Logic Programs". In: *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009), Potsdam, Germany, September 14-18, 2009*, ed. by Esra Erdem et al. Vol. 5753. LNCS. Potsdam, Germany: Springer, pp. 87–100. ISBN: 978-3-642-04237-9. DOI: 10.1007/978-3-642-04238-6_10. URL: http://www.kr.tuwien.ac.at/staff/tkren/pub/2009/lpnmr2009-splitting.pdf (cit. on pp. 32 sq., 219, 222 sq., 225, 256, 267 sq.).

— (2011). "First-Order Encodings of Modular Nonmonotonic Logic Programs". In: *Datalog 2.0*, ed. by Georg Gottlob. LNCS. Springer. URL: http://www.kr.tuwien.ac.at/staff/tkren/pub/2011/datalog20-fomlp.pdf (cit. on p. 33).

Dao-Tran, Minh, Thomas Eiter, Michael Fink, and Thomas Krennwallner (2011). "Model Streaming for Distributed Multi-Context Systems". In: *2nd Workshop on Logic-based Interpretation of Context: Modelling and Applications, Vancouver, Canada, May 16, 2011*, ed. by Alessandra Mileo et al. Vol. 738. CEUR Workshop Proceedings, pp. 11–22. URL: http://ceur-ws.org/Vol-738/dao-tran-etal.pdf (cit. on pp. 10, 259).

— (2015). "Distributed Evaluation of Nonmonotonic Multi-context Systems". In: *Journal on Artificial Intelligence Research* 52, pp. 543–600. DOI: 10.1613/jair.4574 (cit. on pp. 10, 259).

Demiger, Matúš (2016). *Official Practice Test for the 11th World Sudoku Championship 2016 (Wacky Slovak Classics)*. URL: http://logicmastersindia.com/lmitests/?test=M201610S (cit. on p. 4).

DeRemer, Frank and Hans Kron (June 1975). "Programming-in-the Large Versus Programming-in-the-small". In: *ACM SIGPLAN Notices - International Conference on Reliable Software* 10.6, pp. 114–121. ISSN: 0362-1340. DOI: 10.1145/390016.808431 (cit. on pp. 12, 18).

Dix, Jürgen (1995). "Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview". In: *Logic, Action and Information. Proc. of the Konstanz Colloquium in Logic and Information (LogIn'92)*, ed. by Andre Fuhrmann et al. DeGruyter, pp. 241–329 (cit. on p. 35).

Egly, Uwe, Thomas Eiter, Hans Tompits, and Stefan Woltran (2000). "Solving Advanced Reasoning Tasks using Quantified Boolean Formulas". In: *Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA*, ed. by

Henry A. Kautz et al. AAAI Press, pp. 417–422. URL: http://www.aaai.org/Papers/AAAI/2000/AAAI00-064.pdf (cit. on p. 58).

Eiter, Thomas, Gerhard Brewka, Minh Dao-Tran, Michael Fink, Giovambattista Ianni, and Thomas Krennwallner (Sept. 2009a). "Combining Nonmonotonic Knowledge Bases with External Sources". In: *7th International Symposium on Frontiers of Combining Systems (FroCos 2009), Trento, Italy, September 16–18, 2009*, ed. by Silvio Ghilardi et al. Vol. 5749. LNAI. Springer, pp. 18–42. ISBN: 978-3-642-04221-8. DOI: 10.1007/978-3-642-04222-5_2 (cit. on p. 32).

Eiter, Thomas, Wolfgang Faber, Michael Fink, and Stefan Woltran (2007a). "Complexity results for answer set programming with bounded predicate arities and implications". In: *Annals of Mathematics and Artificial Intelligence* 51.2–4, pp. 123–165. DOI: 10.1007/s10472-008-9086-5 (cit. on p. 125).

Eiter, Thomas, Wolfgang Faber, and Mushthofa Mushthofa (2010). "Space Efficient Evaluation of ASP Programs with Bounded Predicate Arities". In: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*. AAAI'10. Atlanta, Georgia: AAAI Press, pp. 303–308 (cit. on p. 125).

Eiter, Thomas, Michael Fink, Thomas Krennwallner, and Christoph Redl (2012a). "Conflict-driven ASP solving with external sources". In: *Theory and Practice of Logic Programming* 12.4–5, pp. 659–679. DOI: 10.1017/S1471068412000233 (cit. on pp. 28, 42).

Eiter, Thomas, Michael Fink, and Stefan Woltran (July 2007b). "Semantical Characterizations and Complexity of Equivalences in Answer Set Programming". In: *ACM Transactions on Computational Logic* 8.3. ISSN: 1529-3785. DOI: 10.1145/1243996.1244000 (cit. on p. 269).

Eiter, Thomas, Stefano Germano, Giovambattista Ianni, Tobias Kaminski, Christoph Redl, Peter Schüller, and Antonius Weinzierl (2018). "The DLVHEX System". In: *Künstliche Intelligenz*. First Online: 15 May 2018, p. 3. ISSN: 1610-1987. DOI: 10.1007/s13218-018-0535-y (cit. on pp. 35, 225).

Eiter, Thomas and Georg Gottlob (Sept. 1995). "On the computational cost of disjunctive logic programming: Propositional case". In: *Annals of Mathematics and Artificial Intelligence* 15.3, pp. 289–323. DOI: 10.1007/BF01536399 (cit. on p. 87).

Eiter, Thomas, Georg Gottlob, and Heikki Mannila (1994). "Adding Disjunction to Datalog". In: *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 24-26, 1994, Minneapolis, Minnesota, USA*, pp. 267–278. DOI: 10.1145/182591.182639 (cit. on pp. 20, 256).

— (Sept. 1997a). "Disjunctive Datalog". In: *ACM Transactions on Database Systems* 22.3, pp. 364–417. DOI: 10.1145/261124.261126 (cit. on pp. 20, 256).

Eiter, Thomas, Georg Gottlob, and Helmuth Veith (1997b). "Modular Logic Programming and Generalized Quantifiers". In: *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-1997)*. Vol. 1265.

LNCS. Springer, pp. 290–309. URL: http://www.kr.tuwien.ac.at/staff/eiter/et-archive/cdtr97108.ps.gz (cit. on pp. 20, 22 sq., 28 sq., 42, 47, 53, 251, 262 sq., 267).

Eiter, Thomas, Georg Gottlob, and Helmuth Veith (2000). "Generalized Quantifiers in Logic Programs". In: *Generalized Quantifiers and Computation, 9th European Summer School in Logic, Language, and Information, ESSLLI'97 Workshop, Aix-en-Provence, France, August 11-22, 1997, Revised Lectures*, ed. by Jouko Väänänen. LNCS 1754. Springer, pp. 72–98. DOI: 10.1007/3-540-46583-9_4 (cit. on pp. 42, 47, 49).

Eiter, Thomas, Giovambattista Ianni, and Thomas Krennwallner (Sept. 2009b). "Answer Set Programming: A Primer". In: *5th International Reasoning Web Summer School (RW 2009), Brixen/Bressanone, Italy, August 30–September 4, 2009*, ed. by Sergio Tessaris et al. Vol. 5689. LNCS. Springer, pp. 40–110. ISBN: 978-3-642-03753-5. DOI: 10.1007/978-3-642-03754-2_2. URL: http://www.kr.tuwien.ac.at/staff/tkren/pub/2009/rw2009-asp.pdf (cit. on pp. 38, 269).

Eiter, Thomas, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits (Aug. 2008). "Combining answer set programming with description logics for the Semantic Web". In: *Artificial Intelligence* 172.12-13, pp. 1495–1539. DOI: 10.1016/j.artint.2008.04.002 (cit. on pp. 160, 174 sq.).

Eiter, Thomas, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits (2006a). "dlvhex: A System for Integrating Multiple Semantics in an Answer-Set Programming Framework". In: *Proceedings 20th Workshop on Logic Programming and Constraint Systems (WLP '06)*, ed. by M. Fink et al. TU Wien, Inst. f. Informationssysteme, TR 1843-06-02, pp. 206–210. URL: http://www.kr.tuwien.ac.at/events/wlp06/S02-final.ps.gz (cit. on p. 35).

— (2006b). "Effective Integration of Declarative Rules with external Evaluations for Semantic Web Reasoning". In: *Proceedings of the 3rd European Semantic Web Conference (ESWC 2006)*. Vol. 4011. LNCS. Springer, pp. 273–287. DOI: 10.1007/11762256_22 (cit. on pp. 28, 42).

Eiter, Thomas, Tobias Kaminski, Christoph Redl, Peter Schüller, and Antonius Weinzierl (2017). "Answer Set Programming with External Source Access". In: *Reasoning Web. Semantic Interoperability on the Web: 13th International Summer School 2017, London, UK, July 7-11, 2017, Tutorial Lectures*, ed. by Giovambattista Ianni et al. Springer International Publishing, pp. 204–275. ISBN: 978-3-319-61033-7. DOI: 10.1007/978-3-319-61033-7_7 (cit. on pp. 35, 42, 225).

Eiter, Thomas, Thomas Krennwallner, and Christoph Redl (2013). "HEX-Programs with Nested Program Calls". In: *Applications of Declarative Programming and Knowledge Management*, ed. by Hans Tompits et al. Berlin, Heidelberg: Springer, pp. 269–278. ISBN: 978-3-642-41524-1. DOI: 10.1007/978-3-642-41524-1_15 (cit. on pp. 49, 255).

Eiter, Thomas, Thomas Krennwallner, Patrik Schneider, and Guohui Xiao (Mar. 2012b). "Uniform Evaluation of Nonmonotonic DL-Programs". In: *7th International Sym-*

posium on Foundations of Information and Knowledge Systems (FoIKS 2012), ed. by Thomas Lukasiewicz et al. Vol. 7153. LNCS. Kiel, Germany: Springer, pp. 1–22. DOI: 10.1007/978-3-642-28472-4_1. URL: http://www.kr.tuwien.ac.at/staff/tkren/pub/2012/foiks2012-uniform.pdf (cit. on pp. 33, 225).

Eiter, Thomas, Nicola Leone, and Domenico Saccà (Mar. 1997c). "On the Partial Semantics for Disjunctive Deductive Databases". In: *Annals of Mathematics and Artificial Intelligence* 19.1–2, pp. 59–96. ISSN: 1573-7470. DOI: 10.1023/A:1018947420290 (cit. on pp. 20, 257).

Eiter, Thomas, Magdalena Ortiz, Mantas Šimkus, Trung-Kien Tran, and Guohui Xiao (2012c). "Query Rewriting for Horn-$\mathcal{SHIQ}$ Plus Rules". In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*. AAAI'12. Toronto, Ontario, Canada: AAAI Press, pp. 726–733. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/4931 (cit. on p. 175).

Emden, Maarten H. van and Robert A. Kowalski (1976). "The Semantics of Predicate Logic as a Programming Language". In: *Journal of the ACM* 23, pp. 733–742. DOI: 10.1145/321978.321991 (cit. on pp. 70, 257).

Erdem, Esra and Vladimir Lifschitz (July 2003). "Tight Logic Programs". In: *Theory and Practice of Logic Programming* 3.4, pp. 499–518. ISSN: 1471-0684. DOI: 10.1017/S1471068403001765 (cit. on p. 190).

Faber, Wolfgang (2005). "Unfounded Sets for Disjunctive Logic Programs with Arbitrary Aggregates". In: *8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, ed. by Chitta Baral et al. Vol. 3662. LNCS. Springer, pp. 40–52. DOI: 10.1007/11546207_4 (cit. on p. 268).

Faber, Wolfgang, Gianluigi Greco, and Nicola Leone (2007). "Magic Sets and their application to data integration". In: *Journal of Computer and System Sciences* 73.4. Special Issue: Database Theory 2005, pp. 584–609. ISSN: 0022-0000. DOI: 10.1016/j.jcss.2006.10.012 (cit. on p. 257).

Faber, Wolfgang, Nicola Leone, and Gerald Pfeifer (Jan. 2011). "Semantics and complexity of recursive aggregates in answer set programming". In: *Artificial Intelligence* 175.1, pp. 278–298. DOI: 10.1016/j.artint.2010.04.002 (cit. on pp. 27 sq., 54, 62 sq., 263).

Fages, François (1994). "Consistency of Clark's completion and existence of stable models". In: *Methods of Logic in Computer Science* 1.1, pp. 51–60 (cit. on p. 190).

Fagin, Ronald (1976). "Probabilities on Finite Models". In: *Journal of Symbolic Logic* 41.1, pp. 50–58. DOI: 10.1017/S0022481200051756 (cit. on p. 23).

Ferraris, Paolo, Joohyung Lee, and Vladimir Lifschitz (2006). "A generalization of the Lin-Zhao theorem". In: *Ann. Math. Artif. Intell.* 47.1-2, pp. 79–101. DOI: 10.1007/s10472-006-9025-2 (cit. on pp. 183, 268).

Ferraris, Paolo, Joohyung Lee, and Vladimir Lifschitz (2011). "Stable models and circumscription". In: *Artificial Intelligence* 175.1, pp. 236–263. ISSN: 0004-3702. DOI: 10.1016/j.artint.2010.04.011 (cit. on pp. 201, 252, 258).

Ferraris, Paolo, Joohyung Lee, Vladimir Lifschitz, and Ravi Palla (July 2009). "Symmetric Splitting in the General Theory of Stable Models". In: *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI-09), Pasadena, California, USA, July 11–17, 2009*, ed. by Craig Boutilier. AAAI Press, pp. 797–803 (cit. on pp. 257, 268).

Fitting, Melvin (Mar. 1987). "Enumeration operators and modular logic programming". In: *Journal of Logic Programming* 4.1, pp. 11–21. ISSN: 0743-1066. DOI: 10.1016/0743-1066(87)90019-7 (cit. on pp. 19, 251).

Fowler, Martin (2004). *Inversion of Control Containers and the Dependency Injection pattern*. URL: https://martinfowler.com/articles/injection.html (cit. on p. 10).

Gaifman, Haim and Ehud Shapiro (1989). "Fully abstract compositional semantics for logic programs". In: *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, pp. 134–142. DOI: 10.1145/75277.75289 (cit. on pp. 19 sq., 252).

Garey, Michael R. and David S. Johnson (1979). *Computers and Intractability – A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman (cit. on pp. 5, 81).

Garey, Michael R., David S. Johnson, and Larry J. Stockmeyer (1976). "Some simplified NP-complete graph problems". In: *Theoretical Computer Science* 1.3, pp. 237–267. ISSN: 0304-3975. DOI: 10.1016/0304-3975(76)90059-1 (cit. on p. 37).

Gebser, Martin, Tomi Janhunen, and Jussi Rintanen (2014a). "Answer Set Programming as SAT modulo Acyclicity". In: *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*. Vol. 263. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 351–356. ISBN: 978-1-61499-418-3. DOI: 10.3233/978-1-61499-419-0-351 (cit. on p. 269).

— (2014b). "SAT Modulo Graphs: Acyclicity". In: *Logics in Artificial Intelligence: 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014*. Springer International Publishing, pp. 137–151. ISBN: 978-3-319-11558-0. DOI: 10.1007/978-3-319-11558-0_10 (cit. on p. 269).

Gebser, Martin, Roland Kaminski, Benjamin Kaufmann, Patrick Lühne, Philipp Obermeier, Max Ostrowski, Javier Romero, Torsten Schaub, Sebastian Schellhorn, and Philipp Wanko (2018). "The Potsdam Answer Set Solving Collection 5.0". In: *Künstliche Intelligenz*. First Online: 14 May 2018, p. 2. ISSN: 1610-1987. DOI: 10.1007/s13218-018-0528-x (cit. on p. 252).

Gebser, Martin, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub (2013). *Answer Set Solving in Practice*. Morgan & Claypool Publishers (cit. on p. 3).

— (2014c). "*Clingo* = ASP + Control: Preliminary Report". In: *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*, ed. by M. Leuschel et al. Vol. arXiv:1405.3694v1. Theory and Practice of Logic Programming, Online Supplement (cit. on pp. 5, 36).
— (2017). "Multi-shot ASP solving with clingo". In: *CoRR* abs/1705.09811. URL: http://arxiv.org/abs/1705.09811 (cit. on pp. 35, 233, 252).
Gebser, Martin, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider (2011). "Potassco: The Potsdam Answer Set Solving Collection". In: *AI Communications* 24.2, pp. 107–124. ISSN: 0921-7126. DOI: 10.3233/AIC-2011-0491 (cit. on pp. 35, 227).
Gebser, Martin, Benjamin Kaufmann, and Torsten Schaub (2012). "Conflict-driven answer set solving: From theory to practice". In: *Artificial Intelligence* 187, pp. 52–89. DOI: 10.1016/j.artint.2012.04.001 (cit. on pp. 5, 35, 267).
Gelfond, Michael and Vladimir Lifschitz (1988). "The Stable Model Semantics for Logic Programming". In: *Logic Programming: Proceedings Fifth Intl Conference and Symposium*. Cambridge, Mass.: MIT Press, pp. 1070–1080 (cit. on pp. 35, 45).
— (1990). "Logic programs with classical negation". In: *Logic Programming: Proc. of the Seventh Int'l Conf.* Ed. by D. Warren et al. Cambridge, MA, USA: MIT Press, pp. 579–597 (cit. on p. 35).
— (1991). "Classical negation in logic programs and deductive databases". In: *New Generation Computing* 9, pp. 365–385 (cit. on pp. 3, 27, 35, 53).
Gierz, Gerhard, Karl Heinrich Hofmann, Klaus Keimel, Jimmie D. Lawson, Michael Mislove, and Dana S. Scott (2003). *Continuous Lattices and Domains*. Vol. 93. Encyclopedia of Mathematics and its Applications. Cambridge University Press, p. 628 (cit. on p. 69).
Gilbert, Seth and Nancy Lynch (June 2002). "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services". In: *SIGACT News* 33.2, pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: https://www.comp.nus.edu.sg/~gilbert/pubs/BrewersConjecture-SigAct.pdf (cit. on p. 10).
Giunchiglia, Enrico, Yuliya Lierler, and Marco Maratea (Sept. 2006). "Answer Set Programming Based on Propositional Satisfiability". In: *Journal of Automated Reasoning* 36.4, pp. 345–377. ISSN: 1573-0670. DOI: 10.1007/s10817-006-9033-2 (cit. on p. 35).
Glebskii, Y. V., D. I. Kogan, I. M. Liogonki, and V. A. Talanov (1969). "The extent and degree of satisfiability of a form of the restricted predicate calculus". In: *Kibernetika* 2, pp. 31–42 (cit. on p. 23).
Goranko, Valentin and Martin Otto (2007). "Model Theory of Modal Logic". In: *Handbook of Modal Logic*, ed. by Patrick Blackburn et al. Vol. 3. Studies in Logic and Practical Reasoning. Elsevier, pp. 249–329. DOI: 10.1016/S1570-2464(07)80008-5 (cit. on p. 53).

Gottlob, Georg, Simon Ceri, and Letizia Tanca (1989). "What You Always Wanted to Know About Datalog (And Never Dared to Ask)". In: *IEEE Transactions on Knowledge & Data Engineering* 1, pp. 146–166. ISSN: 1041-4347. DOI: 10.1109/69.43410 (cit. on pp. 19, 129).

Gottlob, Georg, Stanislav Kikot, Roman Kontchakov, Vladimir Podolskii, Thomas Schwentick, and Michael Zakharyaschev (2014). "The price of query rewriting in ontology-based data access". In: *Artificial Intelligence* 213, pp. 42–59. ISSN: 0004-3702. DOI: 10.1016/j.artint.2014.04.004 (cit. on p. 175).

Grädel, Erich (1989). "Dominoes and the complexity of subclasses of logical theories". In: *Annals of Pure and Applied Logic* 43.1, pp. 1–30. ISSN: 0168-0072. DOI: 10.1016/0168-0072(89)90023-7 (cit. on p. 109).

Grädel, Erich (2007). "Finite Model Theory and Descriptive Complexity". In: *Finite Model Theory and Its Applications*. Berlin, Heidelberg: Springer, pp. 125–230. ISBN: 978-3-540-68804-4. DOI: 10.1007/3-540-68804-8_3 (cit. on p. 85).

Guo, Yuanbo, Zhengxiang Pan, and Jeff Heflin (2005). "LUBM: A benchmark for OWL knowledge base systems". In: *Web Semantics* 3.2-3, pp. 158–182. ISSN: 1570-8268. DOI: 10.1016/j.websem.2005.06.005 (cit. on pp. 220, 226, 265).

Harrison, Amelia and Yuliya Lierler (2016). "First-Order Modular Logic Programs and their Conservative Extensions". In: *Theory and Practice of Logic Programming* 16.5-6, pp. 755–770. DOI: 10.1017/S1471068416000430 (cit. on p. 252).

Hemachandra, Lane A. (1989). "The strong exponential hierarchy collapses". In: *Journal of Computer and System Sciences* 39.3, pp. 299–322. ISSN: 0022-0000. DOI: 10.1016/0022-0000(89)90025-1 (cit. on p. 84).

Heymans, Stijn, Thomas Eiter, and Guohui Xiao (2010). "Tractable Reasoning with DL-Programs over Datalog-rewritable Description Logics". In: *19th European Conference on Artificial Intelligence, 16-20 August 2010 (ECAI 2010)*, ed. by Helder Coelho et al. IOS Press, pp. 35–40. DOI: 10.3233/978-1-60750-606-5-35 (cit. on pp. 30, 129, 175).

Hudak, Paul (Sept. 1989). "Conception, Evolution, and Application of Functional Programming Languages". In: *ACM Computing Surveys* 21.3, pp. 359–411. DOI: 10.1145/72551.72554 (cit. on p. 12).

Ianni, Giovambattista, Guiseppe Ielpa, Adriana Pietramala, and Maria Carmela Santoro (2003). "Answer Set Programming with Templates". In: *Proceedings of the 2nd International Answer Set Programming Workshop (ASP'03)*. CEUR Workshop Proceedings. CEUR WS. URL: http://www.ceur-ws.org/Vol-78/asp03-final-ianni.pdf (cit. on pp. 253 sq.).

ISO-Prolog (June 2000). *Programming Language Prolog part 2 – Modules*. Tech. rep. ISO/IEC 13211-2:2000(E). JTC1/SC22/WG17 (International Standardization Working Group for the Programming Language Prolog) (cit. on p. 19).

Janhunen, Tomi (2004). "Representing Normal Programs with Clauses". In: *16th Eureopean Conference on Artificial Intelligence, ECAI 2004, Valencia, Spain, August 22-27, 2004*, ed. by Ramón López de Mántaras et al., pp. 358–362. URL: http://www.tcs.hut.fi/~ttj/publications/ecai04.ps.gz (cit. on p. 269).

— (2018). "Cross-Translating Answer Set Programs Using the ASPTOOLS Collection". In: *Künstliche Intelligenz*. First Online: 14 May 2018, p. 3. ISSN: 1610-1987. DOI: 10.1007/s13218-018-0529-9 (cit. on p. 35).

Janhunen, Tomi and Ilkka Niemelä (2016). "The Answer Set Programming Paradigm". In: *AI Magazine: Special Issue on Answer Set Programming*, ed. by Gerd Brewka et al. Vol. 37. 3. Editorial pp. 5–6. AAAI Press, pp. 13–24. DOI: 10.1609/aimag.v37i3.2671 (cit. on pp. 8, 38).

Janhunen, Tomi, Ilkka Niemelä, Dietmar Seipel, Patrik Simons, and Jia-Huai You (Jan. 2006). "Unfolding Partiality and Disjunctions in Stable Model Semantics". In: *ACM Transactions on Computational Logic* 7.1, pp. 1–37. ISSN: 1529-3785. DOI: 10.1145/1119439.1119440 (cit. on p. 35).

Janhunen, Tomi, Ilkka Niemelä, and Mark Sevalnev (2009a). "Computing Stable Models via Reductions to Difference Logic". In: *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, ed. by Esra Erdem et al. Vol. 5753. Springer, pp. 142–154. DOI: 10.1007/978-3-642-04238-6_14 (cit. on p. 269).

Janhunen, Tomi and Emilia Oikarinen (2007). "Automated Verification of Weak Equivalence within the SMODELS System". In: *Theory and Practice of Logic Programming* 7.6, pp. 697–744. DOI: 10.1017/S1471068407003031 (cit. on pp. 258, 269).

Janhunen, Tomi, Emilia Oikarinen, Hans Tompits, and Stefan Woltran (2009b). "Modularity Aspects of Disjunctive Stable Models". In: *Journal of Artificial Intelligence Research* 35, pp. 813–857. DOI: 10.1613/jair.2810 (cit. on pp. 20, 22, 27, 30, 32, 183 sq., 217, 233 sq., 237, 251 sq., 258, 266, 268).

Järvisalo, Matti, Emilia Oikarinen, Tomi Janhunen, and Ilkka Niemelä (2009). "A Module-Based Framework for Multi-language Constraint Modeling". In: *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009), Potsdam, Germany, 14-18 September, 2009*, ed. by Esra Erdem et al. Springer, pp. 155–168. DOI: 978-3-642-04238-6_15 (cit. on p. 20).

Ji, Jianmin, Hai Wan, Ziwei Huo, and Zhenfeng Yuan (2015). "Splitting a Logic Program Revisited". In: *Twenty-Ninth AAAI Conference on Artificial Intelligence*. AAAI'15. Austin, Texas: AAAI Press, pp. 1511–1517. ISBN: 0-262-51129-0. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9538 (cit. on p. 257).

Johnson, Ralph E. and Brian Foote (June 1988). "Designing Reusable Classes". In: *Journal of Object-Oriented Programming* 1.2, pp. 22–35 (cit. on p. 10).

Kernighan, Brian W. and Dennis M. Ritchie (1996). *The C Programming Language*. 2nd. Prentice Hall (cit. on p. 13).

Kleene, Stephen Cole (1952). *Introduction to Metatmathematics*. North Holland (cit. on p. 72).

Knuth, Donald E. (2018). *The Art of Computer Programming, Volume 4B, Fascicle 5. Mathematical Preliminaries Redux; Backtracking; Dancing Links*. Addison-Wesley. URL: https://www-cs-faculty.stanford.edu/~knuth/fasc5c.ps.gz (cit. on p. 4).

Krennwallner, Thomas (July 6, 2011). "Promoting Modular Nonmonotonic Logic Programs". In: *Technical Communications of the 27th International Conference on Logic Programming (ICLP 2011), Lexington, Kentucky, U.S.A., July 6–10, 2011*, ed. by John Gallagher et al. Vol. 11. Leibniz International Proceedings in Informatics (LIPIcs). Lexington, Kentucky, U.S.A.: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 274–279. ISBN: 978-3-939897-31-6. DOI: 10.4230/LIPIcs.ICLP.2011.274. URL: http://www.kr.tuwien.ac.at/staff/tkren/pub/2011/iclp2011dc-mlp.pdf (cit. on p. 33).

Krötzsch, Markus (July 2011). "Efficient Rule-Based Inferencing for OWL EL". In: *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*. AAAI Press, pp. 2668–2673. DOI: 10.5591/978-1-57735-516-8/IJCAI11-444 (cit. on p. 175).

Krötzsch, Markus, Sebastian Rudolph, and Pascal Hitzler (2013). "Complexities of Horn Description Logics". In: *ACM Transactions on Computational Logic* 14.1, 2:1–2:36. DOI: 10.1145/2422085.2422087 (cit. on p. 175).

Krötzsch, Markus, Sebastian Rudolph, and Peter H. Schmitt (2015). "A closer look at the semantic relationship between Datalog and description logics". In: *Semantic Web* 6.1, pp. 63–79. DOI: 10.3233/SW-130126 (cit. on p. 175).

Lakos, John (2016). *Large-Scale C++ Software Design*. Addison-Wesley Professional (cit. on p. 10).

Lee, Joohyung and Vladimir Lifschitz (Dec. 2003). In: *Proceedings of the Nineteenth International Conference on Logic Programming (ICLP-03)*. Springer, pp. 451–465. DOI: 10.1007/978-3-540-24599-5_31 (cit. on pp. 183, 192, 194, 268).

Lee, Joohyung and Yunsong Meng (2009). "On Reductive Semantics of Aggregates in Answer Set Programming". In: *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009), Potsdam, Germany, 14-18 September, 2009*, ed. by Esra Erdem et al. Vol. 5753. LNCS. Springer, pp. 182–195. DOI: 10.1007/978-3-642-04238-6_17 (cit. on p. 268).

— (2011). "First-order stable model semantics and first-order loop formulas". In: *Journal of Artificial Intelligence Research* 42, pp. 125–180. DOI: 10.1613/jair.3337 (cit. on p. 215).

Leone, Nicola, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello (2006). "The DLV System for Knowledge Represen-

tation and Reasoning". In: *Transactions on Computationl Logic* 7.3, pp. 499–562. ISSN: 1529-3785. DOI: 10.1145/1149114.1149117 (cit. on pp. 35, 227, 254).

Leroy, Xavier, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon (2017). *The OCaml system release 4.06, Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique. URL: http://caml.inria.fr/pub/docs/manual-ocaml/ (cit. on p. 18).

Lewis, Harry R. (Oct. 1978). "Complexity of solvable cases of the decision problem for the predicate calculus". In: *19th Annual Symposium on Foundations of Computer Science (FOCS 1978)*, pp. 35–47. DOI: 10.1109/SFCS.1978.9 (cit. on p. 110).

Libkin, Leonid, ed. (2004). *Elements of Finite Model Theory*. Springer. DOI: 10.1007/978-3-662-07003-1 (cit. on pp. 56, 58, 215).

Lierler, Yuliya and Miroslaw Truszczyński (2011). "Transition systems for model generators—A unifying approach". In: *Theory and Practice of Logic Programming* 11.4–5, pp. 629–646. DOI: 10.1017/S1471068411000214 (cit. on p. 252).

— (2013). "Modular Answer Set Solving". In: *Late-Breaking Developments in the Field of Artificial Intelligence, Bellevue, Washington, USA, July 14-18, 2013*. Vol. WS-13-17. AAAI Workshops. AAAI. URL: http://www.aaai.org/ocs/index.php/WS/AAAIW13/paper/view/7077 (cit. on pp. 252, 260).

— (2016). "On abstract modular inference systems and solvers". In: *Artificial Intelligence* 236, pp. 65–89. ISSN: 0004-3702. DOI: 10.1016/j.artint.2016.03.004 (cit. on pp. 20, 260).

Lifschitz, Vladimir and Alexander Razborov (Apr. 2006). "Why Are There So Many Loop Formulas?" In: *ACM Transactions on Computational Logic* 7.2, pp. 261–268. ISSN: 1529-3785. DOI: 10.1145/1131313.1131316 (cit. on p. 215).

Lifschitz, Vladimir and Hudson Turner (June 1994). "Splitting a Logic Program". In: *Proceedings of the 11th International Conference on Logic Programming (ICLP 1994)*. MIT Press, pp. 23–37 (cit. on pp. 20, 30, 220, 222, 256).

Lifschitz, Vladimir and Thomas Y. C. Woo (1992). "Answer Sets in General Nonmonotonic Reasoning (Preliminary Report)". In: *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning* (*KR 1992*), ed. by B. Nebel et al. Morgan Kaufmann, pp. 603–614. URL: http://www.cs.utexas.edu/users/vl/papers/answersets.ps (cit. on p. 35).

Lin, Fangzhen and Yuting Zhao (2004). "ASSAT: computing answer sets of a logic program by SAT solvers". In: *Artificial Intelligence* 157.1–2. Preliminary version in AAAI'02, pp. 115–137. DOI: 10.1016/j.artint.2004.04.004 (cit. on pp. 30, 33, 35, 183, 192, 194, 215, 265).

Lin, Fangzhen and Yi Zhou (2011). "From answer set logic programming to circumscription via logic of GK". In: *Artificial Intelligence* 175.1, pp. 264–277. ISSN: 0004-3702. DOI: 10.1016/j.artint.2010.04.001 (cit. on p. 201).

Liskov, Barbara and Jeannette M. Wing (Nov. 1994). "A Behavioral Notion of Subtyping". In: *ACM Transactions on Programming Languages and Systems* 16.6, pp. 1811–1841. ISSN: 0164-0925. DOI: 10.1145/197320.197383. URL: http://reports-archive.adm.cs.cmu.edu/anon/1999/CMU-CS-99-156.ps (cit. on p. 269).

Liskov, Barbara and Stephen Zilles (Mar. 1974). "Programming with Abstract Data Types". In: *ACM SIGPLAN Notices* 9.4, pp. 50–59. ISSN: 0362-1340. DOI: 10.1145/942572.807045 (cit. on p. 11).

Lloyd, John W. (1987). *Foundations of Logic Programming*. Berlin: Springer (cit. on pp. 46, 70, 257).

Marek, Victor W. and V.S. Subrahmanian (1992). "The relationship between stable, supported, default and autoepistemic semantics for general logic programs". In: *Theoretical Computer Science* 103.2, pp. 365–386. ISSN: 0304-3975. DOI: 10.1016/0304-3975(92)90019-C (cit. on p. 190).

Marek, Victor W. and Miroslaw Truszczyński (1999). "Stable Models and an Alternative Logic Programming Paradigm". In: *The Logic Programming Paradigm: A 25-Year Perspective*. Springer, pp. 375–398. ISBN: 978-3-642-60085-2. DOI: 10.1007/978-3-642-60085-2_17 (cit. on p. 12).

Marques Silva, João P., Inês Lynce, and Sharad Malik (2009). "Conflict-Driven Clause Learning SAT Solvers". In: *Handbook of Satisfiability: Frontiers in Artificial Intelligence and Applications*. Vol. 185. IOS Press, pp. 131–153. DOI: 10.3233/978-1-58603-929-5-131 (cit. on p. 267).

Martin, Robert C. (2018). *Clean Architecture: A Craftman's Guide to Software Structure and Design*. Prentice Hall (cit. on p. 10).

Milner, Robin, Mads Tofte, Robert Harper, and David MacQueen (1997). *The Definition of Standard ML (Revised)*. MIT Press (cit. on p. 18).

Moura, João (Dec. 2016). "Modular Logic Programming: Full Compositionality and Conflict Handling for Practical Reasoning". PhD thesis. Campus de Campolide, 1099-085 Lisboa, Portugal: Universidade Nova de Lisboa. URL: http://hdl.handle.net/10362/30470 (cit. on pp. 253, 267).

Moura, João and Carlos Viegas Damásio (2014). "Generalizing Modular Logic Programs". In: *CoRR* abs/1404.7205. URL: http://arxiv.org/abs/1404.7205 (cit. on pp. 20, 22).

— (2015). "Allowing Cyclic Dependencies in Modular Logic Programming". In: *17th Portuguese Conference on Artificial Intelligence (EPIA 2015), Coimbra, Portugal, Sep. 8-11, 2015*, ed. by Francisco Pereira et al. Springer, pp. 363–375. DOI: 10.1007/978-3-319-23485-4_37 (cit. on pp. 20, 22).

Murphy, Gail C., Robert J. Walker, Elisa L. A. Baniassad, Martin P. Robillard, Albert Lai, and Mik A. Kersten (Oct. 2001). "Does aspect-oriented programming work?" In: *Communications of the ACM* 44.10, pp. 75–77. DOI: 10.1145/383845.383862 (cit. on p. 10).

Niemelä, Ilkka (1999). "Logic Programming with Stable Model Semantics as Constraint Programming Paradigm". In: *Annals of Mathematics and Artificial Intelligence* 25.3–4, pp. 241–273. DOI: 10.1023/A:1018930122475 (cit. on pp. 12, 35).

Niemelä, Ilkka (Aug. 2008). "Stable Models and Difference Logic". In: *Annals of Mathematics and Artificial Intelligence* 53.1–4, pp. 313–329. ISSN: 1012-2443. DOI: 10.1007/s10472-009-9118-9 (cit. on p. 269).

Nieuwenhuis, Robert, Albert Oliveras, and Cesare Tinelli (Nov. 2006). "Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T)". In: *Journal of the ACM* 53.6, pp. 937–977. ISSN: 0004-5411. DOI: 10.1145/1217856.1217859 (cit. on pp. 3, 260, 269).

Norvig, Peter (2006). *Solving Every Sudoku Puzzle*. URL: http://www.norvig.com/sudoku.html (visited on 01/08/2017) (cit. on p. 5).

Oikarinen, Emilia (Oct. 2008). "Modularity in Answer Set Programs". TKK Dissertations in Information and Computer Science. P.O.Box 5400, 02015 Espoo, Finland: Helsinki University of Technology. ISBN: 978-951-22-9582-1. URL: http://lib.tkk.fi/Diss/2008/isbn9789512295821/ (cit. on pp. 252, 258).

Oikarinen, Emilia and Tomi Janhunen (Nov. 2008). "Achieving compositionality of the stable model semantics for Smodels programs". In: *Theory and Practice of Logic Programming* 8.5–6, pp. 717–761. DOI: 10.1017/S147106840800358X (cit. on pp. 20, 22, 253, 267 sq.).

— (2009). "A Translation-based Approach to the Verification of Modular Equivalence". In: *Journal of Logic and Computation* 19.4, pp. 591–613. DOI: 10.1093/logcom/exn039 (cit. on pp. 258, 260, 269).

OSGi Alliance (June 2014). *OSGi Core Release 6 Specification*. Tech. rep. The OSGi Alliance. URL: https://osgi.org/download/r6/osgi.core-6.0.0.pdf (cit. on p. 13).

Papadimitriou, Christos H. (1994). *Computational Complexity*. Addison-Wesley (cit. on pp. 81, 85).

Papazoglou, Mike P. and Willem-Jan van den Heuvel (Mar. 2007). "Service oriented architectures: approaches, technologies and research issues". In: *The VLDB Journal* 16.3, pp. 389–415. ISSN: 0949-877X. DOI: 10.1007/s00778-007-0044-3 (cit. on p. 10).

Parlog, Nicolai (2018). *The Java 9 Module System*. Manning Early Access Program (MEAP). Manning (cit. on p. 12).

Parnas, David Lorge (Dec. 1972). "On the Criteria to Be Used in Decomposing Systems into Modules". In: *Communications of the ACM* 15.12, pp. 1053–1058. ISSN: 0001-0782. DOI: 10.1145/361598.361623 (cit. on p. 11).

Pautasso, Cesare, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai Josuttis (Jan. 2017a). "Microservices in Practice, Part 1: Reality Check and Service De-

sign". In: *IEEE Software* 34.1, pp. 91–98. ISSN: 0740-7459. DOI: 10.1109/MS.2017.24 (cit. on p. 10).

Pautasso, Cesare, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai Josuttis (Jan. 2017b). "Microservices in Practice, Part 2: Service Integration and Sustainability". In: *IEEE Software* 34.2, pp. 97–104. ISSN: 0740-7459. DOI: 10.1109/MS.2017.56 (cit. on p. 10).

Pearce, David (1997). "A new logical characterisation of stable models and answer sets". In: *Non-Monotonic Extensions of Logic Programming*, ed. by Jürgen Dix et al. Vol. 1216. LNCS. Springer, pp. 57–70. DOI: 10.1007/BFb0023801 (cit. on p. 267).

Poggi, Antonella, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati (2008). "Linking Data to Ontologies". In: *Journal on Data Semantics* 10, pp. 133–173. DOI: 10.1007/978-3-540-77688-8_5 (cit. on p. 175).

Reinhold, Mark (Apr. 2015). *Java Platform Module System: Requirements*. URL: http://openjdk.java.net/projects/jigsaw/spec/reqs/2015-04-01 (cit. on p. 12).

— (Mar. 2016). *The State of the Module System*. URL: http://openjdk.java.net/projects/jigsaw/spec/sotms/2016-03-08 (cit. on p. 12).

Reis, Gabriel Dos, Mark Hall, and Gor Nishanov (Feb. 2016). *A Module System for C++ (Revision 4)*. Tech. rep. P0142R0. JTC1/SC22/WG21 (The C++ Standards Committee ISOCPP). URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0142r0.pdf (cit. on pp. 12–15).

Reiter, Raymond (1980). "A Logic for Default Reasoning". In: *Artificial Intelligence* 13, pp. 81–132. DOI: 10.1016/0004-3702(80)90014-4 (cit. on p. 35).

Robinson, Alan and Andrei Voronkov, eds. (2001). *Handbook of Automated Reasoning*. North-Holland. DOI: 10.1016/B978-044450813-3/50000-X (cit. on pp. 3, 268).

Rogers, Jr., Hartley (1987). *Theory of Recursive Functions and Effective Computability*. Reprint of the 1967 edition. New York: MIT Press (cit. on p. 251).

Ross, Keneth A. (1994). "Modular Stratification and Magic Sets for Datalog Programs with Negation". In: *Journal of the ACM* 41.6, pp. 1216–1267 (cit. on p. 19).

Rossi, Francesca, Peter van Beek, and Toby Walsh, eds. (2006). *Handbook of Constraint Programming*. Elsevier. ISBN: 9780444527264 (cit. on p. 3).

Sakama, Chiaki and Katsumi Inoue (1995). "Paraconsistent Stable Semantics for Extended Disjunctive Programs". In: *Journal of Logic and Computation* 5.3, pp. 265–285. DOI: 10.1093/logcom/5.3.265 (cit. on pp. 29, 266).

Savelsbergh, Martin and Peter van Emde Boas (1984). "BOUNDED TILING, an alternative to SATISFIABILITY?" In: *2nd Frege Conference*, ed. by Gerd Wechsung. Akademie Verlag, pp. 354–363. URL: https://ir.cwi.nl/pub/6527/6527A.pdf (cit. on p. 109).

Shapiro, Ehud and Leon Sterling (1994). *The Art of Prolog*. 2nd. MIT Press (cit. on p. 12).

Shields, Mark and Simon Peyton Jones (2002). "First class modules for Haskell". In: *9th International Conference on Foundations of Object-Oriented Languages (FOOL 9), Portland, Oregon*. Springer, pp. 28–40. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/first_class_modules.pdf (cit. on p. 18).

Simons, Patrik, Ilkka Niemelä, and Timo Soininen (June 2002). "Extending and Implementing the Stable Model Semantics". In: *Artificial Intelligence* 138 (1–2), pp. 181–234. DOI: 10.1016/S0004-3702(02)00187-X (cit. on p. 35).

Stockmeyer, Larry J. (1976). "The polynomial-time hierarchy". In: *Theoretical Computer Science* 3.1, pp. 1–22. ISSN: 0304-3975. DOI: 10.1016/0304-3975(76)90061-X (cit. on p. 83).

Strachey, Christopher (2000). "Fundamental Concepts in Programming Languages". In: *Higher-Order and Symbolic Computation* 13.1, pp. 11–49. ISSN: 1573-0557. DOI: 10.1023/A:1010000313106 (cit. on p. 21).

Stroustrup, Bjarne (2013). *The C++ Programming Language*. 4th. Addison-Wesley (cit. on p. 15).

Swift, Terrance and David S. Warren (Jan. 2012). "XSB: Extending Prolog with Tabled Logic Programming". In: *Theory and Practice of Logic Programming* 12.1-2, pp. 157–187. DOI: 10.1017/S1471068411000500 (cit. on p. 19).

Syrjänen, Tommi (2001). "Omega-Restricted Logic Programs". In: *Logic Programming and Nonmotonic Reasoning (LPNMR 2011)*, ed. by Thomas Eiter et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 267–280. ISBN: 978-3-540-45402-1. DOI: 10.1007/3-540-45402-0_20 (cit. on p. 36).

Syrjänen, Tommi (Mar. 2009). "Logic Programs and Cardinality Constraints: Theory and Practice". TKK Dissertations in Information and Computer Science. P.O.Box 5400, 02015 Espoo, Finland: Helsinki University of Technology. ISBN: 978-951-22-9763-4. URL: http://lib.tkk.fi/Diss/2009/isbn9789512297634/ (cit. on p. 36).

Tari, Luis, Chitta Baral, and Saadat Anwar (July 2005). "A Language for Modular Answer Set Programming: Application to ACC Tournament Scheduling". In: *3rd Answer Set Programming Workshop (ASP'05), Bath, UK, July 27th–29th 2005*. Vol. 142. CEUR Workshop Proceedings. CEUR WS, pp. 277–293. URL: http://www.ceur-ws.org/Vol-142/page277.pdf (cit. on p. 254).

Tarski, Alfred (1955). "A lattice-theoretical fixpoint theorem and its applications." In: *Pacific Journal of Mathematics* 5.2, pp. 285–309. URL: http://projecteuclid.org/euclid.pjm/1103044538 (cit. on p. 71).

Tasharrofi, Shahab (Dec. 2013). "Arithmetic and Modularity in Declarative Languages for Knowledge Representation". PhD thesis. 8888 University Drive, Burnaby, BC, Canada V5A 1S6: Simon Fraser University. URL: http://summit.sfu.ca/item/13936 (cit. on p. 259).

Tasharrofi, Shahab and Eugenia Ternovska (2014). "Three Semantics for Modular Systems". In: *CoRR* abs/1405.1229.1405.1229. URL: http://arxiv.org/abs/1405.1229 (cit. on p. 259).

Ternovska, Eugenia (2015). "An Algebra of Combined Constraint Solving". In: *GCAI 2015. Global Conference on Artificial Intelligence*, ed. by Georg Gottlob et al. Vol. 36. EPiC Series in Computing. EasyChair, pp. 275–295. DOI: 10.29007/976n (cit. on p. 259).

Truszczyński, Miroslaw (Nov. 2010). "Reducts of propositional theories, satisfiability relations, and generalizations of semantics of logic programs". In: *Artificial Intelligence* 174.16–17, pp. 1285–1306. DOI: 10.1016/j.artint.2010.08.004 (cit. on pp. 267 sq.).

Truszczyński, Miroslaw and Stefan Woltran (Nov. 2009). "Relativized hyperequivalence of logic programs for modular programming". In: *Theory and Practice of Logic Programming* 9.6, pp. 781–819. DOI: 10.1017/S1471068409990159 (cit. on pp. 269 sq.).

Väänänen, Jouko (1999). "Generalized Quantifiers, an Introduction". In: *Generalized Quantifiers and Computation: 9th European Summer School in Logic, Language, and Information, ESSLLI'97 Workshop, Aix-en-Provence, France, August 1997*. Vol. 1754. LNCS. Springer, pp. 1–17. URL: http://www.math.helsinki.fi/logic/opetus/ylkvantII/beatcs.pdf (cit. on p. 42).

Van Roy, Peter and Seif Haridi (2004). *Concepts, Techniques, and Models of Computer Programming*. MIT Press (cit. on p. 11).

Vardi, Moshe Y. (1982). "The complexity of relational query languages (Extended Abstract)". In: *14th Annual ACM Symposium on Theory of Computing (STOC), May 5-7, 1982, San Francisco, California, USA*. San Francisco, pp. 137–146. DOI: 10.1145/800070.802186. URL: http://www.cs.rice.edu/~vardi/papers/stoc82.pdf.gz (cit. on p. 216).

Vennekens, Joost, David Gilis, and Marc Denecker (Oct. 2006). "Splitting an Operator: Algebraic Modularity Results for Logics with Fixpoint Semantics". In: *ACM Transactions on Computational Logic* 7.4, pp. 765–802. DOI: 10.1145/1183278.1183284 (cit. on pp. 20, 257).

Warnholz, Sebastian (July 22, 2017). *modules: Self Contained Units of Source Code*. CRAN. URL: https://CRAN.R-project.org/package=modules (cit. on p. 18).

Wielemaker, Jan, Tom Schrijvers, Markus Triska, and Torbjörn Lager (Jan. 2012). "SWI-Prolog". In: *Theory and Practice of Logic Programming* 12.1-2, pp. 67–96. DOI: 10.1017/S1471068411000494 (cit. on p. 19).

Wijaya, Tri Kurniawan (Aug. 2011). "Top-Down Evaluation Techniques for Modular Nonmonotonic Logic Programs". MSc Thesis. Karlsplatz 13, 1040 Vienna, Austria: Vienna University of Technology. URL: https://repositum.tuwien.ac.at/download/pdf/1614489 (cit. on pp. 220, 225, 265, 267).

Woltran, Stefan (2008). "A common view on strong, uniform, and other notions of equivalence in answer-set programming". In: *Theory and Practice of Logic Programming* 8.2, pp. 217–234. DOI: 10.1017/S1471068407003250 (cit. on p. 269).

Wrathall, Celia (1976). "Complete sets and the polynomial-time hierarchy". In: *Theoretical Computer Science* 3.1, pp. 23–33. ISSN: 0304-3975. DOI: 10.1016/0304-3975(76)90062-1 (cit. on p. 83).

Xiao, Guohui (Dec. 2013). "Inline Evaluation of Hybrid Knowledge Bases". PhD thesis. Karlsplatz 13, 1040 Vienna, Austria: Vienna University of Technology. URL: https://repositum.tuwien.ac.at/download/pdf/1633893 (cit. on pp. 175–178).

Xiao, Guohui, Thomas Eiter, and Stijn Heymans (2013). "The DReW System for Nonmonotonic DL-Programs". In: *Semantic Web and Web Science*, ed. by Juanzi Li et al. New York, NY: Springer, pp. 383–390. ISBN: 978-1-4614-6880-6. DOI: 10.1007/978-1-4614-6880-6_33 (cit. on pp. 225, 265).

Yato, Takayuki and Takahiro Seta (2003). "Complexity and Completeness of Finding Another Solution and Its Application to Puzzles". In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E86-A.5, pp. 1052–1060 (cit. on p. 5).

Yourdon, Edward and Larry L. Constantine (1979). *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall (cit. on p. 9).

Colophon

This thesis was written in the LaTeX 2_ε language using AMS-LaTeX, MATHTOOLS, NTHEOREM for mathematical macros and environments, TikZ / PGF 3.0.1a for drawing graphs and figures, LISTINGS for automatic source code pretty-printing, and BibLaTeX v3.11 for citation management. The document layout is based on an adapted *veelo* chapter style from the MEMOIR class version 3.7g, where the epigraphs in each chapter n show hypercube graphs Q_n in Part I, complete graphs K_n in Part II, wheel graphs W_n in Part III, and star graphs S_n in Part IV. The source code was written with the GNU Emacs 25.2.2 editor employing version 12.1.1 of the AUCTeX package. Revisions and changes of this thesis were managed using the git 2.18.0 version control system. The thesis was typeset with the LuaLaTeX 107.0 LaTeX-compiler and the biber v2.11 processor for BibTeX database files from the TeX Live 2018 distribution of the mainline development version of the Debian GNU/Linux distribution codenamed *Sid*. This document uses version 5.3.0 of the serif *Linux Libertine*, the sans-serif Linux Biolinum, the monospaced `Fira Code` v1.204 typefaces for program listings, and *STIX Two* version 2.0.0 with stylistic sets substitutions 2 and 14 for mathematical equations. Chapter lettrines were produced with Linux Libertine Initial (v5.3.0).

Biographical Sketch

Thomas Krennwallner received his BSc (Bakk. techn.) in software and information engineering (2005) and MSc (Dipl.-Ing.) in computational intelligence (2007) from the Vienna University of Technology, Austria. Since 2015 he works as senior software engineer for data-intensive applications at XIMES GmbH and Qmetrix GmbH, implementing data science workflow systems for quantitative analysis methods and business analytics, work shift scheduling, and real-time airport passenger flow optimization. From 2008 to 2014 he worked as research and university assistant in the Knowledge-Based Systems Group at the Institute of Logic and Computation (Vienna University of Technology), where he was doing research on modularity aspects and evaluation algorithms for answer set programs and distributed multi-context systems. He taught courses on introduction into artificial intelligence, logic programming and knowledge-based systems. In 2007 and 2008, he was employed as research intern at the Digital Enterprise Research Institute (DERI) of the National University of Galway, Ireland, where he was studying query language extensions over RDF(S) and XML data for semantic web reasoning tasks. Between 1999 and 2004 he was working as software developer in several companies in the context of internet telephony and public key infrastructure. He won the Content Award Vienna 2012 Smart City Price with the MyITS project. His research was honored with a best paper award of the Int'l Conference on Logic Programming and Nonmonotonic Reasoning 2011, and a best presentation award at the Doctoral Consortium of the Int'l Conference on Logic Programming 2011. He has been awarded the OCG Förderpreis 2009 (OCG advancement award 2009) of the Austrian Computer Society for his master thesis "Integration of Conjunctive Queries over Description Logics into HEX-Programs." Krennwallner is the competition chair of the Federated Logic Conference (FLoC) Olympic Games 2014, organizing cochair of the fourth Answer Set Programming Competition 2013, and local organization cochair of the Reasoning Web (RW) 2012 summer school and the Int'l Web Reasoning and Rules Conference (RR) 2012 in Vienna. He is a team member of knowledge representation systems such as DLVHEX, DMCS, GiaBATA, and XSPARQL, and a maintainer for several Debian Science packages. He is a coauthor of one edited book, four book chapters, ten journal and magazine articles, one W3C Member Submission, and more than 30 conference and workshop papers.