KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Towards a Higher Level of Abstraction for Knowledge Representation Languages

Matthias Van der Hallen

Supervisors:
Prof. dr. ir. G. Janssens
Prof. dr. M. Denecker

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

June 2020

Towards a Higher Level of Abstraction for Knowledge Representation Languages

Matthias VAN DER HALLEN

Examination committee:
Prof. dr. ir. P. Wollants, chair
Prof. dr. ir. G. Janssens, supervisor
Prof. dr. M. Denecker, co-supervisor
Prof. dr. ir. T. Schrijvers
Prof. dr. B. Jacobs
Prof. dr. B. Bogaerts
(Vrije Universiteit Brussels)
Prof. dr. M. Leuschel
(Heinrich Heine University Düsseldorf)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

June 2020

© 2020 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, Matthias Van der Hallen, Celestijnenlaan 200A box 2402, B-3001 Leuven (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Preface

The end of a chapter in my life, fittingly, starts with a preface. This allows me to reflect on my experiences these last few years and to take a moment to thank the people that provided indispensable helping hands in my journey.

A first word of gratitude is owed to my supervisors. **Gerda**, you were a great promotor: tirelessly reading my texts, always ready with advice and endless in patience. It has been a pleasure working with you, towards my PhD, but also in educational councils and in the *declarative languages* course. This course made me contact you about PhD opportunities and I was lucky enough to be a teaching assistant for it for more than 4 yours. **Marc**, throughout the years you have consistently amazed me with the boundless enthusiasm and dedication you showed to your field of research. While intimidating at times, those qualities have helped me greatly and provide a never-ending source of ideas for the students that you supervise as a (in my case, co-) promotor. **Maurice**, I want to thank you too for finding taking the time to proof-read many of my texts.

Next, I want to thank my **jury**, for finding the time to read this PhD thesis. Your remarks and suggestions were greatly appreciated, as is your flexibility towards an online or face-to-face defense.

I thank also the Research Foundation - Flanders for granting me a four year PhD fellowship for strategic basic research.

In all these years, I have also had the pleasure to work with fantastic colleagues; in no specific order I want to thank them as well: **Bart**, **Pieter**, **Joachim**, **Jo**, **Ingmar**, **Ruben**, **Laurent**, **Tim**, **Simon**, **Pierre**, and **Đorđe**. We have had many interesting discussions (work and non-work related) and you are in large part to blame for my collection of board games.

Whenever I had to blow off steam, my friends were there for me. As trying to name them all would make this preface prohibitively long, I will simply name some (sometimes overlapping) groups: **Fintro**, **VTK 13-14 (Fuse)** and

Revue, **KU Leuven Triathlon 16-18** and **Team Ploeg**, thank you for being such wonderful friends.

Lastly, yet foremost, I want to thank my **family**. **Mom** and **dad**, thank you for your unconditional love, for being there every step along the way for the last 28 years. Thank you, **Sara and Wim**, and **Peter**. As my older siblings(-in-law), I have always looked up to you and you have always been there[1]; be it advice, horrible tales from my youth or simple encouragement, standing in your footsteps has helped me stand firm. Thank you **Mare** and **Noor**, my nieces, we all spoil you way too much. Thank you, **Elke**, for being there for me and enduring me, even in lockdown. I love you.

[1] Yes, Wim, you too.

Abstract

Every day, many problems are solved using computers. However, before a computer can solve a problem for us, we must transfer the required knowledge to the computer, using a language both the computer and we understand. While imperative languages follow an approach that instructs the computer step-by-step how to solve the problem, specification languages such as the FO($\cdot$) language and Answer Set Programming (ASP) specify only the knowledge involved in a problem, leaving it up to the computer to find out how to solve it.

The distinction between instructing step-by-step and representing knowledge is fundamental in the field of "Knowledge Representation and Reasoning". It is of great importance for this field that we can specify the knowledge involved in as many interesting problems as possible, in a way that is both clear and maintainable.

In this thesis, we explore how to improve the level of abstraction of knowledge representation languages based on logic so as to make them more expressive and reduce duplication, so that they are more readable and maintainable.

First, we analyze the *graph mining* problem and show that it features second-order and higher-order logic aspects. We show how these aspects can be encoded or simulated using solvers limited to first-order logic, and discuss the limitations and disadvantages of such encoding techniques.

The presence of second-order logic aspects in the graph mining, and the incompatibility of these aspects with solvers developed for first-order logic, such as the IDP system, inspire us to assemble a more extensive collection of problems with second-order aspects.

Motivated by the problems collected, we present a typed second-order specification language. We discuss the implementation of SOGrounder, a system that can translate these typed second-order specifications to Quantified Boolean Formulas (QBF), which in turn can be solved by existing QBF solvers.

We show that the approach shows promise with respect to an approach based on *encoding* second-order constraints for Answer Set Programming using the *saturation* technique.

Finally, we extend our typed second-order language with language constructs serving to reduce the duplication involved in the knowledge specification of many problems. We specifically support a recurrent pattern that we discern in second-order logic specification.

Summarized, this text explores how second-order logic can be used to make specification languages more expressive, both computationally as well as practically.

Beknopte samenvatting

Elke dag lossen computers veel problemen voor ons op. Maar vooraleer een computer een probleem kan oplossen, moeten mensen de vereiste kennis overbrengen op de computer. Hiervoor maken we gebruik van een taal die zowel de computer als wij begrijpen. Imperatieve talen baseren hun aanpak op het stap-voor-stap vastleggen hoe de computer het probleem moet aanpakken. Specificatie talen zoals FO($\cdot$) en Answer Set Programming (ASP), beschrijven dan weer alleen de kennis aanwezig in het probleem zelf, en laten het over aan de computer om uit te zoeken hoe het probleem opgelost kan worden.

Het onderscheid tussen het stap-voor-stap instrueren en het beschrijven van kennis is fundamenteel voor het onderzoeksveld "Kennisrepresentatie en redeneren". Voor dit onderzoeksdomein is het van groot belang dat we de kennis onderliggend aan zo veel mogelijk interessante problemen kunnen beschrijven, en dat op een manier die zowel helder is als gemakkelijk om te onderhouden.

In deze thesis onderzoeken we hoe we het abstractie niveau van kennisrepresentatie talen gebaseerd op logica kunnen verbeteren. We beogen hierbij om ze zowel expressiever te maken als om de hoeveelheid herhaling te verminderen. Dit moet leiden tot talen die enerzijds gemakkelijker leesbaar zijn en beter te begrijpen, en anderzijds ook specificaties opleveren die beter te onderhouden zijn.

Allereerst bestuderen we het *grafe ontginnings*probleem en tonen we aan dat dit probleem inherent aspecten bevat van tweede-orde en hogere-orde logica. We tonen hoe deze aspecten geëncodeerd of gesimuleerd kunnen worden met oplossers voor eerste-orde logica, en belichten de beperkingen en nadelen van zulke technieken.

De aanwezigheid van tweede-orde logica aspecten in het *grafe ontginnings*probleem, en de incompatibiliteit van dit soort aspecten met oplossers gemaakt voor eerste-orde logica zoals het IDP systeem, inspireert ons om een uitgebreidere verzameling aan problemen met tweede-orde logica aspecten samen

te stellen.

Gemotiveerd door deze verzamelde problemen, presenteren we een getypeerde tweede-orde specificatie taal. Vervolgens bespreken we de implementatie van SOGrounder, een systeem dat deze getypeerde tweede-orde specificaties kan vertalen naar Quantified Boolean Formulas (QBF). Deze kunnen op hun beurt opgelost worden door bestaande QBF oplossers. We tonen aan dat deze aanpak beloftevolle resultaten vertoont in vergelijking met een aanpak gebaseerd op het encoderen van tweede-orde aspecten in Answer Set Programming, waar we gebruik maken van een techniek met de naam *saturation*.

Tot slot breiden we onze tweede-orde specificatie taal uit met taalconstructies die er toe dienen om de herhaling in veel kennis specificaties weg te werken. Specifiek bieden we ook ondersteuning voor een veel voorkomend patroon dat we onderscheiden in specificaties gebaseerd op tweede-orde logica.

Samengevat onderzoekt deze tekst hoe tweede-orde logica gebruikt kan worden om specificatie talen expressiever te maken, zowel op theoretisch als op praktisch vlak.

List of Abbreviations

AFT Approximation Fixpoint Theory. 81, 84

AI Artificial Intelligence. 1

ASP Answer Set Programming. 3, 7, 15–18, 28, 29, 40, 41, 47, 49, 52, 56, 59, 63, 70, 80, 81, 127–129

CDCL Conflict-Driven Clause Learning. 18

CLP Constraint Logic Programming. 66

coNP co-Non-deterministic Polynomial Class. 33, 35, 120

CP Constraint Programming. 16, 66

CSP Constraint Satisfaction Problem. 109

CTL Computation Tree Logic. 2, 93

DMN Decision Model and Notation. 2

DRY "Do not Repeat Yourself", a software development principle. 7

EXPTIME The Exponential Time Complexity class. 86

FO First-Order. 3, 23, 29, 30, 66, 72

IDP . 4, 15, 17, 18, 23, 24, 28–31, 33, 40, 47–51, 53, 54, 56, 58, 59, 63, 86, 88, 115

KBS Knowledge Base System. 4

KR Knowledge Representation. 29, 59

KRR Knowledge Representation and Reasoning. 1

LTL Linear Temporal Logic. 2, 92

NP Non-deterministic Polynomial Class. 13, 33, 35, 65, 71, 115, 120, 122, 127

PH The Polynomial Hierarchy. 7, 73, 115, 117

QBF Quantified Boolean Formula. 7, 50–52, 62, 64, 75, 89, 117, 128

QCIR Quantified CIRcuit format. 75

SAT SATisfiable Formulas. 50, 52, 62, 66

SO Second-Order. 7, 65

UMLS Unified Medical Language System, an ontology for medical terminology. 2

List of Symbols

Δ	A definition Δ.
Γ	A typing context.
$\leq_p$	The precision order.
$\leq_t$	The truth order.
$\mathbb{G}$	A set of graphs 'G'.
$\mathcal{G}$	A graph 'G', see also $\mathcal{P}$.
$\mathcal{I}$	An interpretation or structure.
$\mathcal{P}$	A graph 'P', short for 'pattern'.
$\mathcal{T}_n$	A theory with name 'n'.
$\models$	The satisfaction relation.
$\overline{d}$	A tuple of d.
ϕ, ψ	Formulas.
$\sigma(\mathcal{T})$	The signature of a theory.
$\vdash$	The typing relation.
d	A domain element.
f	A function symbol f.
P	A predicate symbol P.
t	A term t.
V	A vocabulary V.
x	A variable x.

Contents

Abstract iii

Beknopte samenvatting v

List of Abbreviations viii

List of Symbols ix

Contents xi

List of Figures xv

List of Tables xvii

1 Introduction 1
- 1.1 Overview . . . 2
  - 1.1.1 Languages . . . 2
  - 1.1.2 Inferences . . . 3
  - 1.1.3 Systems . . . 4
- 1.2 Contributions . . . 4

2 Preliminaries 9
- 2.1 First-Order Logic: Syntax and Semantics . . . 9
  - 2.1.1 Syntax . . . 9
  - 2.1.2 Semantics . . . 10
- 2.2 Expressivity of Logics . . . 13

3 Analysis of Graph Mining 14
- 3.1 Introduction . . . 14
- 3.2 Preliminaries . . . 16
- 3.3 Formalization of graph mining . . . 18

3.3.1 Patterns . . . 18
3.3.2 Canonical patterns . . . 20
3.4 A higher-order specification of Graph Mining . . . 21
3.4.1 Representation of graphs . . . 22
3.4.2 A higher-order specification . . . 23
3.4.3 Desired properties of graph mining specifications . . . 27
3.5 First-order encodings of Graph Mining . . . 29
3.5.1 IDP . . . 29
3.5.2 ASP . . . 40
3.5.3 Comparative Summary . . . 47
3.5.4 Performance experiments . . . 47
3.6 Solver Techniques . . . 50
3.6.1 Nested Solvers . . . 51
3.6.2 Lazy Grounding . . . 56
3.7 Discussion and Future Work . . . 59
3.7.1 Grounding to QBF . . . 62
3.7.2 Lazy Grounding . . . 63
3.8 Conclusion . . . 63

4 A Second-Order Language and its Grounder 65
4.1 Introduction and Related Work . . . 65
4.2 Second-Order Logic . . . 66
4.2.1 Syntax . . . 67
4.2.2 Semantics . . . 68
4.2.3 SO Logic as a Modeling Language . . . 69
4.3 QBF . . . 70
4.4 Implementation . . . 71
4.4.1 Advanced grounding techniques . . . 74
4.4.2 Grounding to QCIR . . . 75
4.5 Experiments . . . 76
4.6 Conclusion and Future Work . . . 78

5 Semantics of Templates 79
5.1 Introduction . . . 80
5.2 Related Work . . . 81
5.3 Preliminaries: Rules and definitions . . . 82
5.3.1 Semantics of definitions . . . 83
5.4 Templates and Template Libraries . . . 84
5.4.1 The Complexity of Templates . . . 85
5.4.2 Template libraries for Existential Second-Order Logic . 86
5.5 Conclusion . . . 88

6 A Second-Order Pattern: Integrating inferences in logic 89

6.1 Introduction 89
6.1.1 Examples 91
6.2 Theoretical foundation 97
6.2.1 Semantics 102
6.2.2 Expressivity 103
6.3 Implementation 106
6.3.1 Parametrized Theories and their Applications 106
6.3.2 Quantifying over Variant Worlds 107
6.4 Use case: Zebra Puzzle 109
6.5 Conclusion 115

7 An Overview of Problems with Second-Order Constraints 117
7.1 Minimal Inconsistent Cores 118
7.2 Optimal Stable Matching Problem 120
7.3 Determining Path Vapnik-Chervonenkis Dimension 123
7.4 Secure Sets 125
7.5 Conclusion 126

8 Conclusion 127
8.1 Future Work 129

Bibliography 131

Curriculum Vitae 145

List of Figures

1.1 Structure of the text. 6

3.1 An illustrative graph mining instance 19
3.2 Pattern candidates for the graph mining instance shown in **Figure** 3.1. 20
3.3 A mapping of candidate 3.2b to the negative example 3.1b. . . 20
3.4 Possible patterns. 21
3.5 (Repeat) A graph mining instance with $(N_+ = 1, N_- = 0)$. . . 38
3.6 A subset of the pattern space for **Figure** 3.5. 39
3.7 Cumulative run times for the Yoshida dataset. 49
3.8 Cumulative ASP run time for the mutagenesis dataset. 49
3.9 The three different strategies proposed for subsolvers. 54
3.10 Cumulative runtimes and time spent per pattern by IDP for the three datasets with only positive examples. 57
3.11 Histograms of time needed to mine the next pattern by IDP. Only strictly positive datasets were used. 60
3.12 Grounding size (#lits) and memory usage (kilobytes) of Ground & Solve and Lazy Grounding approaches in Mutagenesis dataset. 61
3.13 Boxplot: # of example graphs inspected before accepting or refuting a pattern candidate. 61

5.1 The FO(ID^*), ESO(ID^*) and ASO(ID^*) fragments of SO(ID^*). 87

6.1 Visual representation of the second-order pattern. 90
6.2 Example transition system for temporal logics. 93

7.1 An example graph illustrating secure sets. 125

List of Tables

2.1 Kleene's truth table. 12

3.1 Summary of the desirable properties in IDP and ASP. 47

4.1 Overview of grounding times and sizes for strategic companies of size n. 77
4.2 Overview of solving times for SOGrounder and IS (QDimacs), Clingo (ASP), and GhostQ (QCIR). 77
4.3 Overview of SOGrounder's grounding times (ms) with and without binary quantification, 5 min time limit. 78

6.1 Substitutions for the application $*\mathcal{T}_{color}$`(Graph, Homomorphism, Used)`. 107
6.2 Substitutions for the variant world quantification $\Box\mathcal{T}_{color}$`(Graph :{(n1,n2)|t }, Homomorphism:{}, Used:{}):`ϕ. 108
6.3 Solution of the simple Pasta puzzle. 110

7.1 The $*$ operation on labels, modeling the relationship between a directed edge and its start and end vertex. 118

Chapter 1

Introduction

Mankind has a tendency to take things and use them in such a way as to make life easier for itself. That is why people systematically try to automate things using computers. However, getting a computer to do what you want it to is not always as straightforward. That is why *Artificial Intelligence* (AI) has devoted research into making this easier.

In particular, the field of *Knowledge Representation and Reasoning* (KRR) studies languages, which we will call *modeling languages*, that can specify the knowledge in a domain and devises methodologies and tools that allow reasoning over the knowledge specified in these languages. Using such languages, we make knowledge available to the computer and, instead of telling it exactly what to do, allow the computer to apply the reasoning methodologies and tools to perform automated reasoning and to solve our problems. We call these automated reasoning procedures *inferences*.

A very common example of knowledge that is often specified in a modeling language is the knowledge behind *scheduling*. For scheduling, a number of different resources such as classrooms, lectures, teachers or hours must be *matched*. This matching must conform to certain constraints that express practical limitations ("Only one concurrent lecture per classroom"), labour laws ("No teacher teaches more than three consecutive lectures"), etc. One scenario in which we want to automate reasoning is when we are given a certain matching and are tasked with verifying whether all constraints are indeed satisfied. This automated reasoning task is called the *model checking* inference.

Many *modeling* languages have been proposed, some even before we could rely on computers to do the heavy lifting of reasoning over them, for example *First-*

Order Logic (FOL). Although it predates computers, it still serves as a powerful foundation for modeling languages such as the FO(·) language [36], and logic in general is closely linked to the study of complexity and computability through the field of *descriptive complexity* [79]. Likewise, many different reasoning tasks, beyond the aforementioned model checking, have been defined and implemented in tooling. In the next section, we give a non-exhaustive overview of different languages, inferences and systems.

1.1 Overview

1.1.1 Languages

When studying how to express human knowledge, it rapidly becomes evident that knowledge comes in many different forms. For example, some knowledge is objective and universal (e.g. "All men are mortal"), other knowledge is dependent on time (e.g. "I am currently not hungry") or on other people's knowledge. Consider, as an example thereof, the joke where two mathematicians are asked whether they all want beer; only when the first mathematician answers "I do not know" can the second mathematician truthfully answer "yes". The form of reasoning the second mathematician performs is *epistemic*, and his underlying knowledge can be expressed as

$$Beer(M2).$$
$$K_{M1,w}(Beer(M1)).$$
$$Beer(M1) \Leftrightarrow K_{M1}(Beer(M1)).$$

where $K_{M1,w}$ and K_{M1} are epistemic operators that refer to the knowledge of the first mathematician[1]. The first mathematician's answer can be formalized as $K_{M1,w}(Beer(M1) \wedge Beer(M2))$. With this additional knowledge, the second mathematician can deduce $Beer(M1) \wedge Beer(M2)$.

It is only natural that different types of knowledge are reflected in different languages being proposed. Some of these serve to express knowledge in only a specific domain: *description logics* [6] such as Unified Medical Language System (UMLS) [15] for medical knowledge or Decision Model and Notation (DMN) [11] for business knowledge, and *temporal logics* such as Computational Tree Logic (CTL) [30] or Linear Temporal Logic (LTL) [118] are some prime examples.

[1]The operator K_{M1} is to be read as "M1 knows that [...]", while the operator $K_{M1,w}$ expresses "M1 knows whether [...]".

Other languages try to remain general purpose so that a vast range of problem domains can be expressed. Nevertheless, even these languages come in many variations. Some build upon the concept of constraints such as **alldifferent**, e.g. MiniZinc [112], others base themselves on logic. Examples of the latter include ASP-Core-2 [25], the communal language for Answer Set Programming (ASP), based on Horn clauses and rules, and FO(·) [36] which is based on First-Order (FO) logic.

Another explanation for the variety in modeling languages is the intended *computational expressivity* of the language. The FO(·) language, for example, extends traditional first-order logic with (amongst others) types, aggregates and inductive definitions. While the addition of types is purely syntactical and, from a theoretical view at least, does not raise expressive power, it is known that inductive definitions expand on the expressive power of first-order logic. Another example can be found in the dialects of ASP-Core-2, some of which include the more expressive *disjunctive heads* [33] while others do not.

While these languages are very high in expressive power, other languages purposefully limit their expressive power to ensure certain tractability results, e.g., the S-FEEL DMN language [109].

The design of modeling languages is thus often seen as a balancing act between generic and domain-specific knowledge, expressive power and tractability. As a consequence, highly expressive languages such as second-order or higher-order logic are often considered too expressive.

1.1.2 Inferences

Verifying whether a certain matching in a scheduling problem satisfies all constraints was previously introduced as the *model checking* inference. Model checking is far from the only form of inference.

For example, given an incomplete matching, we might want to complete this matching into one that satisfies all constraints. This task is called the *model expansion* [107] inference. When we additionally require that e.g., the number of students in each class is as close to a certain number as possible, we have turned the problem into a task implemented by the *model optimization* inference [60, 124]. When a certain matching does not satisfy the rules, we can ask ourselves which constraints are in fact unsatisfied, leading to an inference task called *unsat core extraction* [103]. If instead we want to reduce the size of the matching to a maximally imprecise matching that is unsatisfiable, we are looking at the *unsat structure* [137] inference. Finally, if we want to compute from a matching

how many hours every teacher teaches in a week, we will use the *querying* [139] inference.

Other inferences include *abduction* [19, 85] (finding the best explanation), *deduction* [75] (deriving necessary truths), *default reasoning* [120, 50, 51] (reasoning with assumptions), and *propagation* [142] (deriving necessary truths).

The inferences listed above are generic inference tasks, however, some applications built with KRR systems require problem-specific inferences, either entirely new inferences (for example, *progression* [100, 18], which computes the result of actions on a state-of-affairs), or specific variations of the inferences listed above, e.g., unsat core extraction with preferences for certain rules.

1.1.3 Systems

The variety of modeling languages and inferences have led to the development of many different systems that can automatically and efficiently perform these inferences on specifications in these languages. Some systems accept multiple languages, others are built in support of a single language.

Furthermore, while some of these systems are also tailored towards a specific inference, e.g., theorem provers and deduction, database systems and querying, the *Knowledge Base System* (KBS) paradigm takes the complete opposite view and tries to support a wide range of inferences so that a *single* specification can be the subject of many different reasoning tasks.

This paradigm, implemented by the IDP system [36], can automate the earlier described tasks for our scheduling problem without burdening us with expressing the same knowledge in many different specifications. To control these many reasoning tasks, the IDP system integrates a LUA-based procedural language. This procedural control language not only allows us to perform specific inferences, but also to store and even manipulate its results, and to tie multiple subsequent inferences together in a procedural style, if necessary.

1.2 Contributions

As mentioned above, an important factor in KR languages and KR language design is *expressivity*. A more expressive language often allows for *shorter* specifications, which tend to be more readable and, as a result, easier to maintain. On the other hand, some knowledge simply *cannot be expressed* at all in certain specification languages, e.g., the transitive closure in traditional

first-order logic. Here, we encounter a *trade-off* between *expressivity* on the one hand, and the feasibility of a solver efficiently performing inferences on our specification on the other hand. In general second-order and higher-order features are considered by many as too expressive, because they allow expression of knowledge for which too many inferences are intractable.

An important aspect to note is that in fact we are dealing with *two* kinds of expressivity: *computational expressivity* and *practical expressivity*. While the first focusses on the knowledge that is *expressible* in a certain language, the second focusses on the ease with which the knowledge can be modeled. Both kinds of expressivity are important, e.g., language extensions can be useful due to the impact they have on *computational* expressivity as well as their impact on *practical* expressivity. Consider, for example, a language based on logic without support for *functions*; while these can be modeled by predicates without loss in *computational expressivity*, their absence greatly impacts the *practical expressivity* [53].

As we noted earlier, second-order and higher-order features are often considered as too expressive, *computationally*. However, we take a stance that while many problems are, from a theoretically point of view, too complex for inferences to be feasible for solvers, many interesting *instances* of these problems indeed *are small enough* to be expressible. Furthermore, problems that were already expressible, can be written more succinctly and readable while at the same time better capturing the knowledge underlying the problem, leading to benefits in *development*, *maintenance* as well as *performance* due to the additional available knowledge. Consider, as an example, scheduling golfers in the Social Golfer Problem [72]. Here, one is asked to divide a group golfers in equally sized, disjoint subgroups (i.e., a partition) in w different ways such that no two golfers are ever grouped together twice or more. The partitions usually correspond to different 'weeks' of a tournament. Traditional first-order modelings introduce identifiers for each partition, leading to symmetries not present in the original problem, which higher-order modelings can prevent using sets of partitions [56].

Therefore, this thesis contributes to the investigation of second-order and higher-order features: their inherent prevalence in certain problems, the techniques or strategies used to encode them partially, the implementation of second-order features in a grounder, and how a language with second-order features can be used to also further advance the *practical expressivity* of languages through forms of *templating*.

The structure of the thesis is visualized in **Figure** 1.1 through its chapters and their dependencies. Specifically, the content of each chapter is as follows:

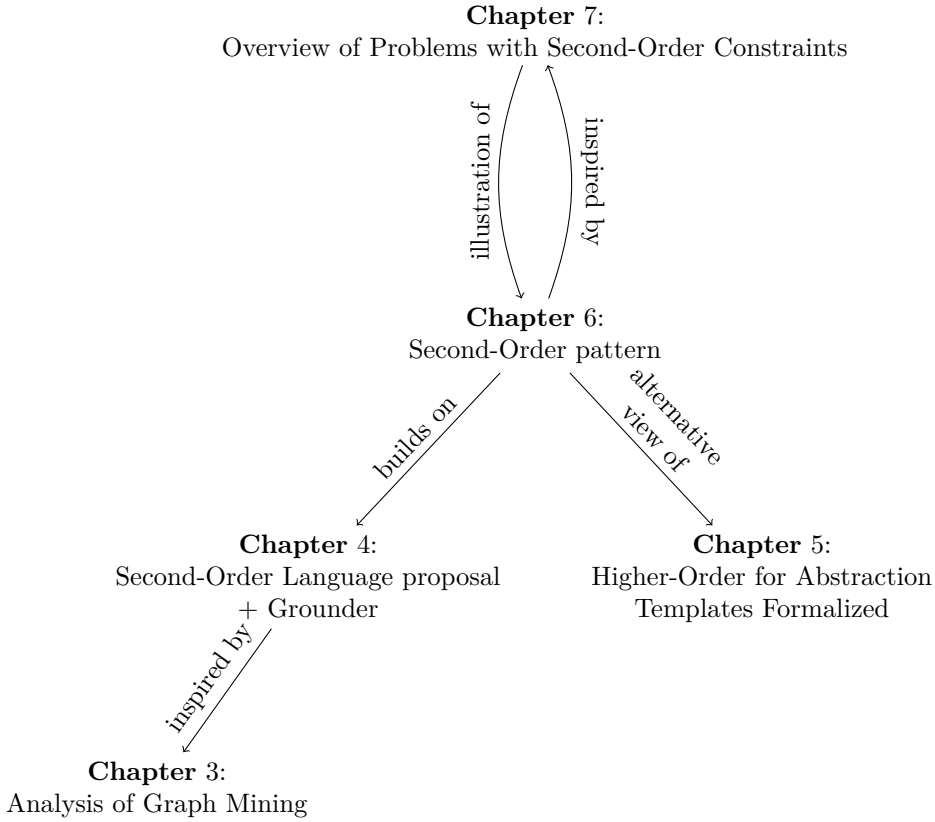


Figure 1.1: Structure of the text.

- **Chapter 2** introduces general and well-established concepts that are considered relevant for the remainder of this thesis.
- **Chapter 3** introduces the *graph mining* problem and analyses it from a Knowledge Representation perspective, focusing on its second-order and higher-order aspects. The graph mining problem is a specific variant of *frequent pattern mining* where *patterns* correspond to a *graph*, i.e., objects with a more complex structure than single items or sets of items. The study indicates that the complex nature of graphs complicates certain concepts essential to *pattern mining* problems, such as the generation of candidate patterns, the matching of candidates in (positive or negative) examples, and canonicity of patterns. It concludes that the *graph mining* problem, and others like it, can benefit greatly from a more expressive language,

both *computationally*, by allowing constraints that require solvers for the second level of the Polynomial Hierarchy, as well as *practically*, by allowing more succinct specification using a "Do not Repeat Yourself" (DRY) approach based on templates. Lastly, it postulates that the added computational expressivity, which allows more intractable problems, can even be of benefit to the solving performance, setting up an ad-hoc experiment to test the hypothesis.

The chapter is a reworked version of the following publication: van der Hallen, M., Paramonov, S., Janssens, G., and Denecker, M. "Knowledge Representation Analysis of Graph Mining", Annals of Mathematics and Artificial Intelligence 86, 1-3 (2019), pp. 21 – 60. [134].

- **Chapter 4** introduces a typed second-order language and discusses SOGrounder, a system that can ground specifications written in this language. Firstly, the chapter proposes our typed *Second-Order* (SO) language as a modeling language and shows how to model the *Strategic Companies* problem, a problem at the second level of the Polynomial Hierarchy (PH). Subsequently, the chapter details the implementation of the grounding procedure. The target of the grounding is *Quantified Boolean Formulas* (QBF) and the system can ground to QDimacs, a *Conjunctive Normal Form*-based representation, as well as to QCIR, which uses a *circuit*-based representation for formulas, the two most commonly accepted input formats for *QBF* solvers. Lastly, the chapter sets up an experiment comparing the performance of grounding and solving the *Strategic Companies* problem, with an ASP system (Clingo [59]) on the one hand and SOGrounder with two different QBF solvers on the other (The QDimacs-based depQBF and the QCIR-based GhostQ.).

  The chapter is a reworked version of the publication: van der Hallen, M. and Janssens, G. "SOGrounder; Modelling and Solving Second-Order Logic" in the Proceedings of the sixteenth international conference on Principles of Knowledge Representation and Reasoning, 2018, pp. 72 – 77. [133]

- **Chapter 5** discusses the concept of *templates* from a theoretical, semantic point of view. Templates are linked to second-order inductive definitions. As a result of this link, templates, previously known in, e.g., ASP as a purely syntactical constructs with an associated rewriting procedure, are given a proper semantic meaning by extending the *satisfaction relation*.

  The chapter is a summary of the publication: Dasseville, I., van der Hallen, M., Janssens, G. and Denecker, M. "Semantics of Templates in a Compositional Framework for Building Logics" in Theory and Practice of Logic Programming, 15, 4-5 (2015), pp. 681 – 695 [35], of which I am

a co-author, and is added to provide a background into the theoretical foundations of templates.

- **Chapter 6** introduces a commonly encountered pattern in *Second-Order* specifications. It introduces the concepts of *parametrized theories* and *variant world quantifiers* as a way of expressing this pattern. These concepts effectively introduce an alternative form of *templates*, and allow us to express many *inferences* in the specification language itself, opening up possibilities of continued reasoning on inference results, as opposed to using procedural control.

  The chapter represents new, unpublished work.

- **Chapter 7** presents a select overview of interesting problems that feature *second-order* logic aspects. As such, it serves as an inspiration and motivation for the development of a specification language covering *second-order* logic, but also as an illustration of the *parametrized theories* and *variant world quantifiers* introduced in **Chapter** 6.

Chapter 2

Preliminaries

In this section, we introduce some general and well-established concepts that make an appearance throughout the main text. Specifically, we introduce the syntax and semantics of first-order logic and discuss the expressivity of logics through *descriptive complexity*.

2.1 First-Order Logic: Syntax and Semantics

2.1.1 Syntax

Definition 1 (Vocabulary). *A vocabulary V consists of an infinite supply of variable symbols and a finite set of predicate symbols P/n and function symbols f/n, where $n \geq 0$ and n is known as the* arity *of the symbol. Predicates and functions of arity 0 are called* propositions *and* constants*, respectively.*

Alternatively, the *vocabulary* is also referred to as a *signature*, leading to the use of the symbol Σ. Without loss of generality, we can refer to a predicate symbol $P/n \in V$ as simply P, specifically when the arity is either irrelevant or clear from context.

Definition 2 (Term). *A term t is either a* variable $x \in V$ *or a function symbol $f/n \in V$ applied to n terms $t_1 \dots t_n$, written as $f(t_1, \dots, t_n)$.*[1]

[1] For constants $f/0$, we usally allow the shorthand f.

Definition 3 (Atom). *An atom is a predicate $P/n \in V$ applied to n terms $t_1, \ldots, t_n$, written as $P(t_1, \ldots, t_n)$.*[2]

Definition 4 (Formula). *Formulas ϕ over vocabulary V are defined inductively:*

- *All atoms are formulas .*
- *if ϕ is a formula, then its negation $\neg\phi$ is a formula.*
- *If ϕ and ψ are formulas, then $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \Rightarrow \psi$, and $\phi \Leftrightarrow \psi$ are formulas.*
- *If ϕ is a formula and x a variable $\in V$, then $\forall x : \phi$ and $\exists x : \phi$ are formulas. We say the variable x is bounded within ϕ by the quantifier $\forall$ or $\exists$.*
- *If t_1 and t_2 are terms, then $t_1 = t_2$ and $t_1 < t_2$ are formula.*

Any occurrence of a symbol s that is not bounded by a quantifier is called *free*. The *free symbols* of a formula ϕ, denoted by *free*(ϕ) are any symbols that have a *free occurence* in ϕ.

2.1.2 Semantics

The semantics of first-order logic are most commonly defined through model theory. First, we have to define the concept of a *structure* or *interpretation* over *vocabulary* V. Structures formalize the notion of a "*possible world*", sometimes also referred to as a "*state-of-affairs*".

Definition 5 (Structures). *A structure $\mathcal{I}$ over vocabulary V consists of a non-empty* domain *D (referred to as dom($\mathcal{I}$)), an assignment of an n-*ary relation *over the domain for every predicate symbol $P/n \in V$, an assignment of a* total n-ary function *over the domain for every function symbol $f/n \in V$, and an assignment of a domain element to every variable symbol.*

In this thesis, we will restrict ourselves to structures with a *finite* domain. Furthermore, *structures* are sometimes referred to as *interpretations*.

A domain D consists of domain elements d, and we assume an ordering exists on these domain elements. A predicate symbol $P/n \in V$ applied to an n-ary tuple of domain elements $(d_1, \ldots, d_n)$ (abbreviated to $\overline{d}$) is commonly called a domain atom of $\mathcal{I}$.

The assignment of relations and functions over the domain by a structure $\mathcal{I}$ to predicates P, respectively functions f, is called the *valuation* function of $\mathcal{I}$, and the value assigned to P (f) by this valuation function is written as $P^{\mathcal{I}}$ ($f^{\mathcal{I}}$).

[2] For propositions $P/0$, we usually allow the shorthand P.

Given a structure $\mathcal{I}$, we can extend or modify its valuation function by writing $\mathcal{I}[x : v]$, meaning the variable or symbol x is assigned the value v, where v, depending on x is either a *domain element of*, a *relation over* or a *function over* the domain $dom(\mathcal{I})$. Multiple modifications can be separated by commas, and are handled left-to-right, i.e., $\mathcal{I}[x : v, y : v', x : v']$ assigns both x and y the value v' in $\mathcal{I}$.

With minor abuse of notation, we write $\mathcal{I}[P(\overline{d}) : \mathbf{t}]$ and $\mathcal{I}[P(\overline{d}) : \mathbf{f}]$ to mean the relation assigned to P by $\mathcal{I}$ is modified to include (respectively exclude) the tuple $\overline{d}$ of domain elements.

Given a structure $\mathcal{I}$ over vocabulary V and a vocabulary $V' \subseteq V$, we can *project* $\mathcal{I}$ to V', writing $\mathcal{I}|_{V'}$. Informally, this restricts the valuation function of $\mathcal{I}$ to symbols in V'. Conversely, for vocabularies V and $V' \subseteq V$, we say a structure $\mathcal{I}$ over V *extends* a structure $\mathcal{I}'$ over V' iff $\mathcal{I}|_{V'} = \mathcal{I}'$.

Projecting structures onto vocabularies allows us to *compare* two structures $\mathcal{I}$ and $\mathcal{I}'$ over different vocabularies V and V' on a common sub-vocabulary V_c (i.e., $V_c \subseteq V$ and $V_c \subseteq V'$). Specifically, we say the structures $\mathcal{I}$ and $\mathcal{I}'$ are *vocabulary-equivalent* on V_c, written as $\mathcal{I} =_{V_c} \mathcal{I}'$ iff $\mathcal{I}|_{V_c} = \mathcal{I}'|_{V_c}$.

The *valuation* function $(\cdot)^{\mathcal{I}}$ is extended for terms t as follows:

- If t is a variable x, then $t^{\mathcal{I}} = x^{\mathcal{I}}$,
- If t is a function symbol f/n applied to n terms $t_1, \ldots, t_n$ then $t^{\mathcal{I}} = f^{\mathcal{I}}(t_1^{\mathcal{I}}, \ldots, t_n^{\mathcal{I}})$.

We also introduce the shorthand $(t_1, \ldots, t_n)^{\mathcal{I}}$ for tuples (sometimes written as $\overline{t}^{\mathcal{I}}$) to mean $({t_1}^{\mathcal{I}}, \ldots, {t_n}^{\mathcal{I}})$.

To define the semantics of formulas, we introduce the *satisfaction relation* $\models$ between structures $\mathcal{I}$ and formulas ϕ, usually written in infix notation as $\mathcal{I} \models \phi$.

Definition 6. *Given a formula ϕ and a structure $\mathcal{I}$ that assigns a value to all* free symbols *of ϕ, we define the satisfaction relation inductively based on the syntactical structure of ϕ:*

- $\mathcal{I} \models P(t_1, \ldots, t_n)$ *where P is an n-ary predicate symbol iff* $(t_1^{\mathcal{I}}, \ldots, t_n^{\mathcal{I}}) \in P^{\mathcal{I}}$. [3]
- $\mathcal{I} \models \neg\phi$ *iff* $\mathcal{I} \not\models \phi$.
- $\mathcal{I} \models \phi \wedge \psi$ *iff* $\mathcal{I} \models \phi$ *and* $\mathcal{I} \models \psi$.
- $\mathcal{I} \models \phi \vee \psi$ *iff* $\mathcal{I} \models \phi$ *or* $\mathcal{I} \models \psi$ *(possibly both).*
- $\mathcal{I} \models \exists x : \phi$ *iff there is a* $d \in dom(\mathcal{I})$ *s.t.* $\mathcal{I}[x : d] \models \phi$.
- $\mathcal{I} \models t_1 = t_2$ *iff* ${t_1}^{\mathcal{I}} = {t_2}^{\mathcal{I}}$.
- $\mathcal{I} \models t_1 < t_2$ *iff* ${t_1}^{\mathcal{I}} < {t_2}^{\mathcal{I}}$.

[3] Note that for propositions $P/0$, we say $\mathcal{I} \models P$ iff the empty tuple $() \in P^{\mathcal{I}}$.

		Value of p		
		t	**f**	**u**
Formula	$\neg p$	**f**	**t**	**u**
	$p \wedge q$	q	**f**	**u**
	$p \vee q$	**t**	q	q

Table 2.1: Kleene's truth table.

Formulas of the form $\phi \Leftrightarrow \psi$, $\phi \Rightarrow \psi$, $\forall x : \phi$ *are defined through their usual equivalences* $(\neg\phi \wedge \neg\psi) \vee (\phi \wedge \psi)$, $\neg\phi \vee \psi$ *and* $\neg\exists x : \neg\phi$.

We say a structure $\mathcal{I}$ *satisfies* ϕ iff $\mathcal{I} \models \phi$. Structures that satisfy ϕ are called *models* of ϕ.[4]

Three-valued semantics Sometimes, the semantics of first-order logic is extended to a *three-valued* semantics, specifically to define rule logics. In *three-valued* semantics, the concept of a *partial structure* is introduced. The valuation function associated with a partial structure assigns predicate symbols P/n a function from n-ary tuples of the domain to the set $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$ where $\mathbf{u}$ is the truth value *unknown*. Function symbols f/n are assigned a function from n-ary tuples of the domain to a set of domain elements; this set represents the possible values that the function can take for the given arguments.

Three-valued truth values admit two partial orderings, the *truth order* $\leq_t$ in which $\mathbf{f} <_t \mathbf{u} <_t \mathbf{t}$ and the *precision order* $\leq_p$ where $\mathbf{u} <_p \mathbf{f}$ and $\mathbf{u} <_p \mathbf{t}$. These orderings can be extended pointwise to structures. We say a partial structure is *exact* if $P^{\mathcal{I}}$ maps into $\{\mathbf{t}, \mathbf{f}\}$ for every P and $f^{\mathcal{I}}$ maps to a singleton for every f.

The most straightforward way to define three-valued semantics is to extend the valuation function to assign formulas a three-valued truth value $\mathbf{t}, \mathbf{f}, \mathbf{u}$ based on Kleene's truth tables (See **Table** 2.1) and define the *satisfaction* relation of partial structures as $\mathcal{I} \models \phi$ iff $\phi^{\mathcal{I}} = \mathbf{t}$.

Arithmetic Standard First-Order logic does not support arithmetic. However, support for arithmetic can be added. To do so, one must introduce the necessary (sometimes partial) functions for the common operations $+, -, *, /$ in the vocabulary and include the *integers* in the domain $dom(\mathcal{I})$ of every structure $\mathcal{I}$, as well as constants for every integer. Furthermore, one should

[4] Alternative definitions sometimes extend the valuation function to formulas, assigning formulas either the value $\mathbf{t}$ or $\mathbf{f}$, reducing the satisfaction relation to simply $\mathcal{I} \models \phi$ iff $\phi^{\mathcal{I}} = \mathbf{t}$.

fix the interpretation for the (partial) arithmetic functions to their standard interpretation.

2.2 Expressivity of Logics

There is a long history of characterizing complexity classes by the logics (over finite structures) that can express the languages within each class, starting from the first result in *Descriptive Complexity* by Fagin [52] that the complexity class NP is *captured* by *existential second-order* logic, i.e., a set of *existential* second-order quantifications followed by a first-order formula.

Definition 7 (Capturing). *A logic L captures a complexity class C if for every problem P, P is in C if and only if there exists a formula ϕ over a vocabulary V in L such that every input string of P corresponds to a finite structure $\mathcal{I}$ over V and $P = \{\mathcal{I} \mid \mathcal{I} \models \phi\}$.*

Continuing on this work, Immerman [78] showed *second-order logic*, with an arbitrary number of alternations, captures the Polynomial Hierarchy (PH) [129]. Specifically, the polynomial hierarchy PH is defined inductively as follows:

Definition 8 (Polynomial Hierarchy PH [129]). *The polynomial hierarchy is the union $\bigcup_{k=0}^{\infty} \Sigma_k^P = \bigcup_{k=0}^{\infty} \Delta_k^P = \bigcup_{k=0}^{\infty} \Pi_k^P$ where*

- *for $k = 0$, the $\Delta_0^P = \Sigma_0^P = \Pi_0^P = P$,*
- *for $k>0$, the classes $\Delta_k^P = P^{\Sigma_{k-1}^P}$, $\Sigma_k^P = NP^{\Sigma_{k-1}^P}$, $\Pi_k^P = coNP^{\Sigma_{k-1}^P}$.*

For Turing machines, Σ_k^P corresponds to the problems *verifiable* in a polynomial number of steps given an oracle solving Σ_{k-1}^P problems in a single step.

The definition of the Polynomial Hierarchy also defines an infinite number of classes that, if the Polynomial Hierarchy does not collapse [129], each correspond to a different level of complexity. This gives rise to *problem families* such as k-QBF family, a variant of the regular QBF problem where the number of quantifier alternations is limited to $k - 1$. If we fix the outermost quantifier to be existential, deciding satisfiability over such formulas is Σ_k^P-complete; if instead the outermost quantifier is fixed to be universal deciding satisfiability is Π_k^P-complete.

Chapter 3

Analysis of Graph Mining

This chapter is a reworked version of a publication in "Annals of Mathematics and Artificial Intelligence" [134]. The work was also presented at the Ninth International Workshop on Answer Set Programming and Other Computing Paradigms – October 16th, 2016, New York City, USA [135]. Modifications have been made to the introduction and the conclusion. Furthermore, some small modifications in presentation were made and errors were corrected.

Personal contribution: 90%.

3.1 Introduction

This chapter is a case study of the graph mining problem, a type of 'frequent pattern mining' task, from a Knowledge Representation point-of-view. From this case study, we studied if and how declarative languages and their solvers support higher-order logics.

In general, the graph mining task involves finding a graph that occurs frequently in a network or a in a database of graphs [49]. We discuss the latter, i.e., finding frequent occurrences in a dataset of graphs, a setting generally referred to as *transactional* as single graph in the dataset can represent a *transaction* in, e.g., shopping data. This is a problem of interest in many fields, such as bioinformatics [88], chemoinformatics [81, 82] and computer vision [86].

Graph mining exists in many variations and the knowledge behind graph mining is easily expressible in logic. These properties make it seem like a prime

candidate for a declarative approach as studied by Knowledge Representation, as these adapt easily to changing requirements and variations. Specifically, we say knowledge representation offers a natural framework for declarative modeling satisfying 'The Principle of Elaboration Tolerance' [105, 65]. This principle, in short, states that declarative specifications should be easily adapted to new requirements or changed circumstances. Graph mining is a prototypical example of a large family of real-world problems that can be formalized as the combination of various smaller problems with minor adaptations.

Current state-of-the-art KR languages such as IDP and ASP aspire to be practical solvers for such problems [23]. While these languages are elaboration tolerant, we show that expressing the graph mining problem in these languages requires unexpectedly complicated and unintuitive encoding techniques. These techniques are in contrast to the ease with which one can transform the mathematical definition of graph mining to a higher-order logic specification, i.e., a specification that allows for both 1) quantification over predicates and functions (covered already by second-order), and 2) higher-order predicates, which accept other predicates and functions as arguments.

These unintuitive encoding techniques furthermore distract from the problem essentials, complicating possible future adaptation. Thus, it is an open challenge for KR systems to provide support for abstract, higher-order modeling while remaining elaboration tolerant.

In this chapter, we argue that efforts should be made towards supporting higher-order logic specifications in modern specification languages, without unintuitive and complicated encoding techniques. We argue that this not only makes representation clearer and more susceptible to future adaptation, but might also allow for faster, more competitive solver techniques to be implemented. We study the *Graph Mining* problem as an example of a problem that could benefit from a higher-order specification, with the aim of gathering insight in and deriving techniques for such problems in general. As a first step towards this goal, we will:

- Propose a higher-order encoding of graph mining that closely follows its mathematical model (**Section** 3.4).
- Explore how the current state-of-the-art KR systems model graph mining using modeling techniques (**Section** 3.5).
- Propose and experiment with additional solver techniques derived from these modeling techniques that can support higher-order encodings, without affecting elaboration tolerance (**Section** 3.6).

3.2 Preliminaries

Graph mining One of the most fundamental tasks in the realm of data mining is *frequent pattern mining*: the task of enumerating patterns which occur frequently in a dataset. *Graph mining* is a variant of *frequent pattern mining* in which the patterns take the form of (labeled) graphs. The dataset in which patterns must occur is either a single large-scale network or a vast set of separate, smaller graphs. The latter option is often referred to as the *transactional setting*. In the context of graph mining, for a pattern to 'occur' in a graph $\mathcal{G}$, it must be homomorphic to a subgraph of graph $\mathcal{G}$.

This work will only consider the transactional setting, as it is computationally more feasible. The transactional setting is relevant as it can be used for *knowledge discovery* from graph structured data in many domains, such as chemoinformatics, natural language processing and bioinformatics. For example, in bioinformatics, graph mining can be used to find molecular substructures (such as benzene rings) that possibly predict or cause certain properties such as lumocity or the mutagenicity of diseases such as Salmonella [82]. In natural language processing, graph mining can identify key concepts in a transcript from a graph representation of the natural language sentences [73].

As real-world problems are computationally challenging, numerous specialized imperative algorithms for graph mining have been developed. These different imperative algorithms correspond to the many variants of pattern mining tasks described in the literature, from various types of item set mining, where data is propositional, to tasks involving more structured data and patterns, such as trees and graphs. Well known examples of algorithms for frequent pattern mining in databases of graphs are gspan [144] and gaston [113].

However, the need for many different algorithms for only slightly different variants within pattern mining tasks has motivated the exploration of more declarative approaches. For example, it has been shown that Constraint Programming (CP) [39] and Answer Set Programming (ASP) [83] can express *item set mining*, which is a setting of frequent pattern mining where data is propositional and can be represented in a table, while the patterns can be described as a *set of items*. Their results demonstrate that such tasks can be accomplished in a declarative way with an acceptable performance penalty. Furthermore, different variations can be supported with only minimal changes.

When mining more complex and structured data than item sets, such as graphs or sequences, predicate logic has been used for representation, and inductive logic programming [110] has emerged as a way to mine such data. While we know of no earlier declarative approaches to graph mining, recent work on sequence mining [68] uses ASP to mine frequent sequences. Their solution

performs pattern generation, the frequency check, and uses an extension of ASP called asprin [21] to prefer patterns that satisfy more involved properties such as maximality (no larger patterns exist) or coverage (no other pattern occurs in the same examples). However, because graphs are more complex structures than sequences, extending their solution to graphs is non-trivial. In the context of graphs, for example, checking *occurrence* corresponds with a homomorphism check (beyond P), instead of a (polynomial) subset check.

Owing to its descriptive complexity, the homomorphism check could easily be expressed using higher-order logic, but this is not supported by state-of-the-art declarative languages such as IDP and ASP. We show that graph mining can nevertheless be supported using various encoding techniques, making it an ideal candidate for our case study into combining support for higher-order logic with elaboration tolerance.

Higher-order logic As mentioned earlier, the composite nature of the graph mining problem lends itself for a higher-order logic specification, in which 1) quantified variables can range not only over individual elements but also over sets (represented by predicates) and 2) predicates can accept other predicates or functions as arguments. For example, a high-level view of the graph mining problem consists of generating connected labeled graphs (patterns), checking whether they occur frequently in a dataset, and filtering out patterns that are too similar to others (e.g., *isomorphic*), leaving only those patterns that we will call *canonical*. In this view, it makes sense to describe the mechanisms behind checking for canonicity and occurrence separately, which, as we will show, translates nicely to higher-order specifications.

Specification languages offer varying levels of support for higher-order logic. On the one hand, meta-programming, as known from Logic Programming [1], has inspired the introduction of higher-order atoms in Hex [44] and the higher-order syntax in HiLog [27]. Predicate symbols can be either constants as in Prolog (first-order case) or variables (second-order case). The latter range over predicate names, and not the predicate space itself, essentially combining second-order syntax with first-order semantics. On the other hand, formal specification languages such as Z [20], B [2], Event-B [3] and TLA [97] extend predicate logic with set theory and offer higher-order datastructures. ProB [98] is a constraint solver, animator and model checker for such languages, implemented in SICStus Prolog.

While it is possible to express the graph mining problem in such languages directly using higher-order logic, earlier work [135] has shown that these systems are restricted to model generation and checking (with or without imperative interfaces), and that their performance does not rival that of systems based on

revolutionary techniques such as CDCL [128]. However, these systems using CDCL, examples of which are the ones for the IDP [36] and the ASP [46, 58] languages, currently do not allow higher-order syntax. Nevertheless, several techniques exist for these languages that allow the user to simulate higher-order logic to model problems such as graph mining. This observation leads us to inquire whether these techniques can be generalized and be used to provide these languages with built-in support for higher-order.

3.3 Formalization of graph mining

In this section we mathematically formalize the transactional graph mining problem. As we will only consider the transactional setting in this work, we will simply refer to it as 'graph mining'. First, we define graphs, graph homomorphism, and the concept of a pattern. We then express what it means for a pattern to be canonical, which is needed when we want to mine more than one pattern.

3.3.1 Patterns

We start with a comprehensive formal definition of the graph mining problem. Throughout this chapter we will assume the existence of two finite, sufficiently large sets: a set V consisting of vertices, and a set L of labels for those vertices.

Definition 9 (Labeled Graph). *A labeled graph $\mathcal{G}$ is a tuple $\langle N, E, l\rangle$ where N is a subset of the vertices V, called the nodes of the graph $\mathcal{G}$, E is a binary predicate on N that represents the set of (directed) edges and l is a unary function from N to L.*

Definition 10 (Connectedness). *A graph $\mathcal{G} = \langle N, E, l\rangle$ is* connected *iff for each pair of nodes v and v' in N, there exists an edge $(v, v') \in E$ or there exists a sequence $v, v_1 \dots v_n, v'$ such that there exist edges (v, v_1), (v_i, v_{i+1}) and $(v_n, v') \in E$, where $1 \leq i \leq n-1$.*

Definition 11 (Graph Homomorphism). *A* (injective) graph homomorphism *f from a labeled graph $\mathcal{G} = \langle N, E, l\rangle$ to a labeled graph $\mathcal{G}' = \langle N', E', l'\rangle$ is a (injective) mapping $f : N \to N'$ from nodes of $\mathcal{G}$ to nodes of $\mathcal{G}'$ such that:*

- $\forall u, v \in N : (u, v) \in E \Rightarrow (f(u), f(v)) \in E'$ *(the mapping preserves edges), and*
- $\forall v \in N : l(v) = l'(f(v))$ *(the mapping preserves labelings).*

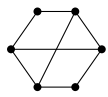
If a (injective) graph homomorphism from graph $\mathcal{G}$ to $\mathcal{G}'$ exists, we say $\mathcal{G}$ is (injectively) homomorphic[1] *to $\mathcal{G}'$.*

Definition 12 (Graph Mining). *Given a sufficiently large set of vertices V and labels L, two sets of graphs over V and L, $\mathbb{G}_+$ and $\mathbb{G}_-$, referred to as the positive respectively negative example graphs, two natural numbers N_- and N_+ (referred to as thresholds), and a graph $\mathcal{T}$ over V and L called the template, we look for a graph $\mathcal{P}$ such that:*

- *$\mathcal{P}$ is a vertex-induced subgraph of $\mathcal{T}$, meaning that the edges of $\mathcal{P}$ are exactly those edges of $\mathcal{T}$ for which both endpoints are also nodes of $\mathcal{P}$,*
- *$\mathcal{P}$ is* connected*,*
- *$\mathcal{P}$ is injectively homomorphic with at least N_+ positive examples $\mathcal{G}_+ \in \mathbb{G}_+$,*
- *$\mathcal{P}$ is injectively homomorphic with at most N_- negative examples $\mathcal{G}_- \in \mathbb{G}_-$.*

We call the example graphs to whom $\mathcal{P}$ is (injectively) homomorphic the positive (negative) homomorphisms, and the restriction on their number the positive (negative) homomorphic property, respectively.

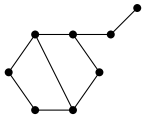
Note that we choose *injective* homomorphisms as the matching operator in this definition, and include the concept of a template graph to guide the search as well as to limit the search space. These choices are inspired by their appropriateness for many use cases in the realm of bioinformatics, chemoinformatics and social networks. However, both of these choices can be changed effortlessly, in the mathematical definition as well as in any specifications of the problem: we can easily drop the injectivity constraint in any logic specification, and can choose the fully connected graph as template without loss of generality. For the remainder of this chapter, we will use 'homomorphic' to mean 'injectively homomorph' unless specifically stated otherwise.



(a) Positive Example



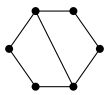
(b) Negative Example



(c) Template Graph

Figure 3.1: A graph mining instance ($N_+ = 1, N_- = 0$) with pattern candidates. Node labels are differentiated by the shape of their indicator (circle, diamond).

[1]Note that within the data mining community, injectively homomorphic is also commonly known as subgraph isomorphic.



(a) Candidate a



(b) Candidate b

Figure 3.2: Pattern candidates for the graph mining instance shown in **Figure** 3.1.

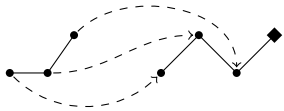


Figure 3.3: A mapping of candidate 3.2b to the negative example 3.1b.

As an example of a graph mining problem instance, take the problem set shown in **Figure** 3.1. Node labels are differentiated by the shape of their indicator: circle or diamond. All nodes have the same (circle) label, except for the rightmost node in the negative example 3.1b. Furthermore, when interpreting these graphs, all (visualized) edges are bidirectional. The angles and lengths of edges are irrelevant, only the connections are relevant. We take the positive and negative thresholds to be $N_+ = 1, N_- = 0$, meaning we require at least one homomorphism with a positive example and allow no homomorphisms with negative examples. There is one positive example (**Figure** 3.1a), and one negative example (**Figure** 3.1b), while **Figure** 3.1c shows the template graph.

Figures 3.2a–3.2b show a pattern and a non-pattern graph respectively: They are both connected subgraphs of the template. However, because we require at least one homomorphism with a positive example, and allow no homomorphisms with negative examples (i.e., problem parameters $N_+ = 1$ and $N_- = 0$), **Figure** 3.2a represents a pattern. It is clear that there exists a mapping from each node of the valid pattern to a node of the positive example, while no such mapping exists for the negative example. Looking at **Figure** 3.2b, this graph is clearly homomorphic with both the positive as well as the negative example: A possible mapping from 3.2b to the negative example 3.1b is shown in **Figure** 3.3. Therefore, 3.2b is a non-pattern graph.

3.3.2 Canonical patterns

To extend on the graph mining task described above, we can look for multiple patterns, instead of just one. In this case, we can impose restrictions on the

different patterns that are found. For example, it stands to reason that one wants only *canonical* solutions, meaning that no two patterns found are *isomorphic*.

Definition 13 (Graph Isomorphism). *A graph isomorphism f between two labeled graphs $\mathcal{G} = \langle N, E, l \rangle$ and $\mathcal{G}' = \langle N', E', l' \rangle$ is a* one-to-one *mapping $N \to N'$ such that f represents an injective homomorphism from $\mathcal{G}$ to $\mathcal{G}'$, and its inverse f^{-1} represents an injective homomorphism from $\mathcal{G}'$ to $\mathcal{G}$. If there exist graph isomorphisms between $\mathcal{G}$ and $\mathcal{G}'$ we say $\mathcal{G}$ and $\mathcal{G}'$ are* isomorphic.



(a) First candidate pattern



(b) Second candidate pattern

Figure 3.4: Possible patterns.

Given the graph mining problem instance specified in **Figure** 3.1, we have already established that **Figure** 3.4a is a pattern. When we try to mine a second pattern, we might suggest a pattern as shown in **Figure** 3.4b. A quick check, however, will show that there is a one-to-one mapping f such that both f as well as its inverse f^{-1} preserve edges. As a result, both patterns candidates are isomorphic, and thus only one should be accepted as a pattern.

Definition 14 (Canonical Patterns). *A set of* canonical patterns *is a set $\mathbb{P}$ of patterns $\mathcal{P}_1, ..., \mathcal{P}_n$, such that for each pair of different elements (of $\mathbb{P}$) $\mathcal{P}_i, \mathcal{P}_j$ holds that there does not exist an isomorphism between $\mathcal{P}_i$ and $\mathcal{P}_j$.*

When we mine multiple patterns, we will pose the additional requirement that the mined patterns must be canonical. Of course, with the above definition of canonicity, many solutions will exist: any pattern can be interchanged for any of its isomorphic counterparts to generate a new solution. If this is unwanted, this can be prevented by introducing an ordering on the isomorphic patterns, and requiring that each pattern in the solution is the minimal pattern among its isomorphic counterparts.

3.4 A higher-order specification of Graph Mining

In this section, we explore how the mathematical formalization of the graph mining problem can be translated towards a higher-order specification. In the first section, we will discuss how graphs introduce higher-order objects when modeled closely to the mathematical definition. Next, we will discuss a

higher-order specification of the complete graph mining problem. We conclude by identifying a set of desired properties for graph mining specifications and their solvers, which the proposed higher-order specification satisfies.

3.4.1 Representation of graphs

Graphs are the main concept in the graph mining problem, and, when represented using tuples $\langle N, E, l \rangle$, they take the form of *composite objects*: these graphs are a collection of first-order objects, namely two predicates and a function. Therefore, a set of graphs is equivalent to a set of tuples: the most straightforward representation of such a set would be a ternary predicate, with the node and edge predicate and the labeling function as arguments:

```
Patterns = {({1,2,3}, {(1,2),(2,3),(3,1)}, {(1↦a), (2↦b), (3↦a)}),
            ({1,2,3}, {(1,3),(3,2)},       {(1↦b), (2↦b), (3↦c)})}.
```

It is very natural to consider and represent each graph as a *coherent* grouping of its own components: all characteristics (edges, labeling ...) of a graph are represented by separate entities or concepts, which are grouped together for each graph $\mathcal{G}$ in the tuple that describes $\mathcal{G}$. We refer to this as the *local coherence* of the graph representation.

Alternative representations could, for example, introduce an edge predicate for each graph separately (e.g., called `edge_g1/2`[2], `edge_g2/2`), or, as we will be forced to do later on, they could introduce an edge predicate for *all* graphs at once. This obscures the relationship between the different characteristics of the same graph:

- In the first alternative this relationship is only present in the *name* of each predicate (which prohibits us to reason about it in a general way). Furthermore, without any additional constraints, it is possible to specify a graph only partially, e.g., only provide an edge relationship.
- In the second alternative the relationship can be expressed using an identifier. However, in contrast to the higher-order tupling approach, it is still possible to specify a graph only partially.

Representing graphs instead as a tuple of their components is not only a very natural choice, it also very explicitly shows that all example graphs are *independent*, and that the searches for homomorphisms between a pattern and

[2]We use `predicate_name/n` to mean the predicate with name `predicate_name` and arity `n`.

example graphs are independent too. This motivates us to reason about graphs as locally coherent objects in our logical models as well. The next section explores a higher-order specification that achieves this goal.

3.4.2 A higher-order specification

In **Listing** 3.1, we propose a specification for the graph mining problem, using features such as higher-order logic and inductive definitions. Regarding syntax and style, we devise a syntax for illustration purposes inspired by IDP [36], which closely corresponds to FO logic with $\leftarrow$ for inductive definitions, as opposed to $\Rightarrow$ for classical implication, and [*type*] for type annotations. We identify four major syntactical additions w.r.t. regular IDP syntax:

- We introduce the keyword `so-type`. The `so-type` keyword can be used to define a second-order type. As such, the type does not represent a set of domain elements, but instead represents a set of (tuples of) predicates or functions. These predicates and functions themselves must be typed using first-order types.
- When presented with an object of a second-order type consisting of a tuple, one will often want to access one specific part of the tuple. To this end, the different parts of a tuple are named. These names provide a way to project a second-order object to one of its parts using `.`-syntax familiar from object-oriented programming. For example, if a tuple `result` representing test scores contains two elements, the `name` and `score`, then `result.score` accesses the score.
- We introduce the possibility to quantify over predicates or functions, using the special quantifiers $\forall_{SO}$ and $\exists_{SO}$. These quantifications must be typed: $\exists_{SO}$ `F [I:O]` means that there exists a function `F` which takes elements of type `I` as input and returns elements of `O` as output. Likewise, $\exists_{SO}$ `P [I,O]` means that there exists a predicate `P`, taking elements from `I` and `O` as its first and second argument, respectively.
- We allow higher-order predicates with arguments of a second-order type. These predicates can be defined using an inductive definition. These are predicates that take (tuples of) predicates and functions as an argument. The semantics of these higher-order inductive definitions will be defined **Chapter** 5, where they are called *template definitions*. For an example, we refer to **Lines** 38–48 of **Listing** 3.1, where the predicate `is_pattern/1`, which takes a second-order argument `graph` as an argument, is defined using an inductive definition.

As with IDP, we first define a vocabulary `V`, and define a theory `T` over this vocabulary `V`. Then, when presented with a specific graph mining instance, we can encode this into a structure `S` and perform the *model expansion* inference to find a solution. We will further explore the contents of these three language blocks in the sections below.

Vocabulary

As mentioned above, the first thing we define is the vocabulary `V`. First, we introduce the types `vertex` and `label` to represent the set of vertices V and set of labels L from the mathematical formalization. Next, we define the second-order type `graph`, which is declared as a tuple of a predicate `node/1`, a predicate `edge/2`, and a partial function `labeling`. These predicates and functions represent the exact subset of vertices which are the nodes, the edges between these nodes and the labeling of these nodes. As such, we have defined all the necessary types for the graph mining problem.

Now, we introduce the necessary predicates symbols: We define the higher-order predicates `homomorph/2` and `isomorph/2`, which are binary relations between graphs. Next, we define some simple sets of graphs as unary higher-order predicates over graphs: the positive example set `positive/1` and its negative counterpart `negative/1`, the set of canonical patterns `canonical_pattern/1`, and the set of patterns `is_pattern/1`. Finally, we define:

- a ternary predicate `connected/3`, which should be true if the two nodes represented by the first two arguments are in fact connected in the graph given as a third argument,
- a higher-order function `template` to refer to the chosen template graph, and
- the two thresholds from the problem statement in **Definition** 12, N_- and N_+ as integers.

Theory

In the theory, we define a number of the higher-order predicates using the concept of template definitions, as described by Dasseville et al. [35]. Whenever a defined predicate accepts a second-order type as argument, it can be decomposed using matching (e.g., **Line** 20). Quantification over second-order objects uses annotated quantifiers ($\exists_{SO}$ and $\forall_{SO}$) and must be typed (any unary predicate symbol can be used as a type), e.g., **Line** 21. We will adhere to the convention

that variables referring to higher-order objects are upper case, whereas variables referring to first-order objects are lower case.

First, we define the concepts of `homomorph/2` and `isomorph/2`: We express the constraint that two graphs are only homomorphic if it is possible to find a function `F` from nodes of the first graph to nodes of the second graph, as can be derived from the existential second-order quantification $\exists_{SO}$ in combination with the typing statement `[N1:N2]` (**Line** 21). In line with **Definition** 11, we first express that this function must be injective (**Line** 21). Next, we specify that it must preserve edges (**Line** 22), and we conclude by specifying that it must preserve labels as well (**Line** 23). For the definition of `isomorph/2`, we follow the mathematical definition in the same way, except for the usage of f^{-1}: instead, we specifically state that the function `F` must be bijective (**Line** 27 and 28), and use that to express that $\texttt{F}^{-1}$ must preserve edges as well (**Line** 30).

Next, we define the concept of connectedness in a given graph: This can be defined rather straightforwardly using an inductive definition by noting that either the two nodes are connected directly, or there exists a third node connected with both argument nodes.

We continue by defining the concept of a pattern, following the requirements of **Definition** 12:

- A pattern is a vertex-induced subgraph of the template (using dot notation to access the separate components of a variable of second-order type, **Line** 41).
- A pattern must be connected.
- If we count the number of graphs from the positive (resp. negative) example set such that the proposed pattern graph is homomorphic with the chosen graph, the result should exceed (resp. should not exceed) the positive (resp. negative) threshold, as evidenced by the count aggregates (**Line** 43–44).

Furthermore, we include two constraints to enforce that all three components of a pattern graph are consistent, i.e. that edges only occur between nodes of the graph (**Line** 45) and that the labeling labels exactly the nodes of the graph (**Line** 46).

Lastly, we provide two constraints saying that for a graph `P` to be a canonical pattern, it must be a pattern, and no other canonical pattern `P2` can be isomorphic to it.

Listing 3.1: Higher-order encoding for the general graph mining problem.

```
vocabulary V {
  type vertex
  type label
  so-type graph of (node(vertex), edge(vertex,vertex), partial labeling(vertex):label)

  homomorph(graph, graph)
  isomorph(graph, graph)
  positive(graph) // a set of positive graphs
  negative(graph)
  canonical_pattern(graph)
  is_pattern(graph)
  connected(vertex,vertex, graph)
  template:graph // a given template
  N_-: int
  N_+: int
}

theory T {
{
  homomorph((N1, E1, L1), (N2, E2, L2)) ←
      (∃_SO F [N1:N2]: (∀ x [N1] y [N1]: x ≠ y ⇒ F(x) ≠ F(y)) ∧
       (∀ x [N1] y [N1]: E1(x, y) ⇒ E2(F(x), F(y))) ∧
       (∀ x [N1]: L1(x) = L2(F(x)))).
}
{
  isomorph((N1, E1, L1),(N2, E2, L2)) ←
      (∃_SO F [N1:N2]: (∀ y [N2]: ∃ x [N1]: F(x)=y) ∧
       (∀ x [N1] y [N1]: x ≠ y ⇒ F(x) ≠ F(y)) ∧
       (∀ x [N1] y [N1]: E1(x, y) ⇒ E2(F(x), F(y))) ∧
       (∀ x [N2] y [N2]: E2(x, y) ⇒ ∃ fx [N1] fy [N1]: E1(fx, fy) ∧ x = F(fx) ∧ y = F(
    fy))∧
       (∀ x [N1]: L1(x) = L2(F(x)))).
}
{
  connected(x, y, (N, E, L)) ← E(x, y) ∨ E(y, x).
  connected(x, y, (N, E, L)) ← ∃ z [N]: connected(x, z, (N, E, L)) ∧ connected(z, y, (N,
     E, L)).
}
{
  is_pattern((N, E, L)) ←
    (
       (∀ x [N]: (template.node(x) ∧ ∀ y [N] : (E(x,y) ⇔ template.edge(x,y))
            ∧ (L(x) = template.labeling(x))))  ∧
       (∀ x [N] y [N]: x ≠ y ⇒ connected(x, y, (N,E,L)))∧
       (#{ Pos : positive(Pos) ∧ homomorph((N,E,L), Pos) }≥ N_+) ∧
       (#{ Neg : negative(Neg) ∧ homomorph((N,E,L), Neg) }≤ N_-)  ∧
       (∀ x [vertex] y [Vertex] : E(x,y) ⇒ N(x) ∧ N(y)) ∧
       (∀ x [vertex] : (∃ l [label] : L(x)=y) ⇔ N(x))
    ).
}
∀P [graph] : canonical_pattern(P) ⇒ is_pattern(P).
∀P [graph] P2 [graph] : canonical_pattern(P)∧canonical_pattern(P2)∧P≠P2 ⇒ ¬isomorph(P,
     P2).
}
```

This encoding compactly specifies the graph mining problem, in a way that closely corresponds to its mathematical definition, providing several general graph properties as templates.

3.4.3 Desired properties of graph mining specifications

Using the graph mining problem as a case study, we derived a set of desirable properties that a good KR specification and its associated solver should satisfy. First, we discuss properties of the KR specification itself:

1. Labeled graphs are the main concept in the mathematical definition of the graph mining problem. In this definition, labeled graphs are seen as a mathematical object consisting of a vertex relation, an edge relation and a labeling function. Thus, a good KR specification should treat labeled graphs as (higher-order) objects.

   It is clear from the second-order type `graph` *in* ***Listing*** *3.1 that this higher-order specification satisfies this property.*

2. All example graphs are independent, so the search for a homomorphism between a pattern and a given example graph can be performed independently. A good KR specification should allow one to write the necessary quantifications locally, i.e., within a formula, as opposed to quantifying globally using the vocabulary. This is more natural, and has the added benefit of keeping the scope of these quantifications as small as possible and making the independence evident.

   The universal quantification over example graphs hidden in the count aggregates of ***Line*** *43 in* ***Listing*** *3.1, combined with the existential quantification of* `F` *on* ***Line*** *21, clearly identifies the independence: for every single example graph, a* ***separate*** *function* `F` *proving the homomorphism can be chosen.*

3. The definition of a homomorphism between pattern and example graph is always the same, regardless of the sign of the example graph (negative or positive). The only difference is the at most/at least constraint on the number of homomorphisms. A good KR specification preserves the similarity of these constraints.

   The great similarity between ***Lines*** *43 and 44 shows that our proposed higher-order specification satisfies this property.*

4. We want to be able to find multiple, non-isomorphic, patterns.

The definition of `canonical_pattern/1`*, and the definition of* `is_pattern/1` *as a set of pattern graphs allows us to express the problem independent of the number of patterns we want to mine.*

5. We want to express constraints such as connectedness of the different nodes in the pattern.

   The concept of inductive definitions, as used in ***Lines*** *34–35 of* ***Listing*** *3.1 shows that we can express constraints such as connectedness in an easy way.*

We also identify some desirable properties for the systems solving a good KR specification of the graph mining problem:

6. We want to perform multiple inferences on the problem, with only minimal changes to the model. In other words, the system should be elaboration tolerant with respect to other inferences, as well as new constraints.

   For example, we might not be interested in just any set of patterns, instead we might want a set of 5 patterns such that they share the highest number of nodes.

7. We prefer specification(s) that can (together) be solved in a single solver call. While specifications are preferably modular to make it easier to reuse them, ideally the composition of specifications would be solvable by a single solver call, requiring no procedural code to tie them together.

The higher-order encoding above satisfies the different properties we identify for a modeling; as such we view it as a *preferred way of encoding* the graph mining problem. Nevertheless, state-of-the-art specification systems either [135] do not accept such specifications, are restricted to model generation and checking (with or without an imperative interface to implement other inferences such as minimization) and/or miss the performance of techniques such Conflict Driven Clause Learning (CDCL) which can (up-to exponentially) reduce the search space by learning new clauses when encountering conflicts and backjumping.

In the next section, we explore which encoding techniques enable us to write a working specification for the graph mining problem in state-of-the-art specification systems: IDP and the systems supporting ASP such as Clingo [59] in particular.

3.5 First-order encodings of Graph Mining

In the previous section, we have shown how graph mining could be specified in a system that supports higher-order logic. In this section, we investigate how state-of-the-art KR systems without support for higher-order logic, such as IDP and systems supporting ASP, can model the graph mining problem, paying special attention to the use of encoding techniques, which might be used in the future to support higher-order logic in general.

3.5.1 IDP

First, we will explore how we can encode the graph mining problem in a state-of-the-art first-order solver such as IDP. We base ourselves on the mathematical specification of graph mining introduced in **Section** 3.3, as well as the higher-order specification explored in **Section** 3.4.

Existential Second-Order

The IDP language allows problem specifications written in *first-order* (FO) logic extended with types, arithmetic, aggregates, and inductive definitions. **Listing** 3.2 shows an example. The symbols in theories T of this logic can be quantified locally, or quantified implicitly in the vocabulary V. Symbols quantified locally can only be propositional, whereas the vocabulary can contain first-order symbols such as functions or predicates (making the vocabulary a *second-order object*).

In the graph mining problem, we are looking for an interpretation I of the symbols in vocabulary V such that I satisfies T, called a model. This corresponds to existential quantification of all (including FO) symbols in V. **Listing** 3.2 shows an example of how IDP extends a given interpretation `S` into a model `Result`. Due to the existential quantification of symbols in V, and the lack of locally quantifiable FO symbols, IDP is limited to model expansion for *existential second-order* problems, which does not include graph mining. We will expand on the underlying shortcomings, and how to sidestep them.

Inferences One of the main philosophies of IDP is its underlying *Knowledge Base* paradigm [36]. Essentially, this paradigm states that a modeller should model the knowledge in a problem domain, without thinking of how data will flow when solving specific queries, or wondering which inference will be performed. Instead, it should be possible to perform various inferences on

Listing 3.2: IDP example using inductive definitions

```
vocabulary V{
    type node
    edge(node, node)
    connected(node, node)
}

theory T : V {
    ∀n[node] : ∃n2[node] : edge(n, n2) ∨ edge(n2, n).
    {
        connected(x, y) ← edge(x, y) ∨ edge(y, x).
        connected(x, y) ← ∃z [node] : connected(x, z) ∧ connected(z,y).
    }
}

structure S : V{ node = {1;2;3} }

structure Result : V{
    node = {1; 2; 3}, edge = {1,1; 1,2; 2,3}
    connected = {1,1; 1,2; 1,3; 2,1; 2,2; 2,3; 3,1; 3,2; 3,3}
}
```

a single specification of the graph mining problem. For example, the most straightforward inference in the case of graph mining would likely be *model expansion*. **Listing** 3.2 shows how model expansion would expand the structure `S` into the structure `Result`. Other inferences of interest for the graph mining problem are, for example, *optimization*. Optimization would allow us to, e.g., minimize or maximize over the number of nodes in the pattern graph, or the number of nodes in the pattern with a certain label, with only minimal changes to the specification of what constitutes a valid pattern.

Definitions versus Constraints A main feature of the IDP language is that it supports FO formulas as well as a rule-based definition constructs (between curly braces). The FO formulas express open world knowledge while definitions express closed world knowledge, representing inductive or recursive definitions such as the definition of `connected/2` in **Listing** 3.2.

This follows from the use of the well-founded semantics underlying the definition construct [42]. In IDP, the theory of two atomic FO axioms `e1(1,2). e1 (2,1).` expresses the open world knowledge that `(1,2)` and `(2,1)` belong to the predicate `e1/2`, while the definition `{e1(1,2). e2(2,1).}`, written with brackets, expresses a definition by exhaustive enumeration, hence the closed world knowledge that `e1` *is* the set `(1,2)`, `(2,1)`. E.g., the first does not entail that `(1,1)` does not belong to `e1/2`, while the definition does. The theory in **Listing** 3.2 expresses that connected is the transitive closure of the edge relation, and that the connected relation is the total relation. The combination

of definition and axiom induces a strong constraint on the value of the edge relation.

Modeling the graph mining problem in IDP

In this subsection we identify three main issues encountered when modeling the graph mining problem:

- the representation of graphs,
- local existential (∃) quantification over functions, and
- local universal (∀) quantification over functions.

The paragraphs below discuss each of the issues and provide an overview of the ways we can currently solve them.

Issue 1: representing graphs First, we must represent the sets of graphs, as specified in **Definition** 12. **Listing** 3.3 shows how this was done in higher-order logic, defining a higher-order predicate `positive/3` with the node predicate, edge predicate and labeling function as arguments[3]. The first graph consists of nodes 1 (labeled a) and 2 (labeled b) and is fully connected. This locally coherent representation preserves a graph as an independent tuple of predicates and functions. However, IDP's vocabulary `V` cannot contain such a second-order symbol.

One possible solution is to replicate for each graph the different characteristic predicates and functions, as shown in **Listing** 3.4, which uses different predicate names for every part of every graph. Using this solution, encoding a property such as "In every graph, all nodes have at least two outgoing edges" must be stated for every graph and its edge predicate explicitly, as no relation exists between the different edge predicates and label functions:

```
∀ n[node] : ∃ n1 [node] n2[node] : e1(n, n1) ∧ e1(n, n2) ∧ n1 ≠ n2.
∀ n[node] : ∃ n1 [node] n2[node] : e2(n, n1) ∧ e2(n, n2) ∧ n1 ≠ n2.
```

It is clear that this solution is not a good KR approach. Furthermore, it is undesirable due to the way it scales with larger problem instances: it prohibits the abstraction (generalization) of knowledge in the theory.

[3]Note the use of inductive definitions, in contrast to constraints, as this allows the derivation of negative knowledge, i.e., `positive/3` *only* contains these two graphs and no others.

Listing 3.3: Higher-order predicate modeling the set $\mathbb{G}_+$ of **Definition** 12.

```
{
  positive({1,2}, {1,2; 2,1}, {1↦a; 2↦b}).
  positive({1,2,3}, {1,3; 2,1}, {1↦c; 2↦b; 3↦a}).
}
```

Listing 3.4: Multiple individual global relations.

```
{
  e1(1, 2). lb1(1)=a.
  e1(2, 1). lb1(2)=b.
  e2(1, 3). lb2(1)=c.
  e2(2, 1). lb2(2)=b.
               lb2(3)=a.
}
```

Listing 3.5: Disjoint union using indexed global relations.

```
{
  e(g1, 1, 2). lb(g1, 1)=a.
  e(g1, 2, 1). lb(g1, 2)=b.
  e(g2, 1, 3). lb(g2, 1)=c.
  e(g2, 2, 1). lb(g2, 2)=b.
                  lb(g2, 3)=a.
}
```

A more workable solution is to represent each characteristic property, such as the edge relation, by a single global relation for all graphs, as shown in **Listing** 3.5. This relation behaves the way it should for a specific graph instance based on an additional argument serving as an identifier for the graph of interest. This global edge relation now corresponds to the *disjoint* or *tagged union* of the graphs' edge relations, with tags drawn from a set `g` of graph identifiers. Generalizing over the different graphs, we can now encode the property stated above as:

$\forall$ `gid[g]` : $\forall$ `n[node]` : $\exists$ `n1 [node] n2[node]` : `e(gid, n, n1)` $\wedge$ `e(gid, n, n2)` $\wedge$ `n1` $\neq$ `n2.`

Although this representation based on reification is the de facto standard way of representing complex objects such as graphs, it is clear that this representation forces us to give up the local coherence of graph characteristics that was present in **Definition** 12. For example, without additional constraints, it is still possible to specify a graph $\mathcal{G}$ only partially, e.g., by providing only an entry in the global `edge/3` relation. Note that the higher-order model from **Section** 3.4 allows elegant expression, as it introduces specific terms (tuples and sets) which could elegantly express these graph characteristics in a way that preserves local coherence.

Issue 2: local $\exists$ quantification over functions The positive homomorphic property can be expressed using a count aggregate, as shown in **Listing** 3.6. First we quantify over all example graphs `G`, or per *issue 1*, their identifiers, and subsequently express that there must exist a function `F` that represents a homomorphism from our pattern graph $\mathcal{P}$ to `G`.

Listing 3.6: Quantifying over functions locally.

```
1 #{G | G ∈ 𝔾+ ∧ ∃ F : F is a homomorphism from 𝒫 to G} ≥ N+.
```

However, in IDP we cannot quantify locally over first-order symbols such as the function `F` from **Listing** 3.6, as it only allows first-order quantifications. We must promote the homomorphic functions to a symbol in the vocabulary, even though we are only interested in the existence of a mapping, not its identity. Reusing the disjoint union technique proposed above avoids the need to introduce a homomorphic function for each example graph separately. Note, we introduce a function `f/2` representing all homomorphisms, and make explicit its dependency on a specific example graph using an additional argument `gId`. In second-order logic, this dependency would follow directly from the syntactic order of the quantifications.

Listing 3.7: Globalized existential functions

```
1 vocabulary V {
2 ...
3 partial f(graphid, vertex):vertex
4 ...
5 }
6 ...
7 #{gId | gId ∈ 𝔾+ : f(gId) is a homomorphism from 𝒫 to gId} ≥ N+.
```

While all example graphs have an `edge`, `label`, ... relation, not all example graphs have a homomorphic function. Therefore, `f` is not defined for graph identifiers that correspond to such graphs, meaning `f` must become a partial function.

By adopting this proposed solution, we can now write an IDP specification for the graph mining problem handling only the positive constraint, as shown in **Listing** 3.8. Note that without the negative constraint, the problem is of a simpler nature (The decision problem is in NP). The next issue discusses how we can add the negative constraint into our IDP model.

Issue 3: local ∀ quantification over functions It is possible to restate the negative homomorphic constraint to *deciding that no homomorphism exists* for enough of the negative examples. However, deciding that no homomorphism from one graph to another exists is in coNP. As an NP (or Σ_1^P) solver, IDP cannot solve this problem directly. One might be tempted to simply specify the negative homomorphic property simply as:

```
#{g | g ∈ 𝔾−: f(g) is a homomorphism from 𝒫 to g} ≤ N−.
```

Listing 3.8: IDP specification handling the positive constraint of the graph mining problem.

```
vocabulary V_pos{
  type vertex isa nat
  type label
  type graphid

  //Predicates determining the template graph.
  template_node(vertex)
  template_edge(vertex, vertex)
  template_label(vertex):label

  //Predicates describing the pattern graph
  pattern_node(vertex)
  pattern_edge(vertex, vertex)
  pattern_label(vertex):label

  //Predicates describing the positive example graphs
  example_edge(graphid, vertex, vertex)
  example_label(graphid, vertex):label
  N+: int

  partial f(graphid, vertex):vertex //Represents the homomorphisms with the example graphs

  homo_with(graphid) //True for graphs for which f represents a correct homomorphism
  connected(vertex, vertex) //connected(a, b) is true if there exists a path
  //from a to b in the pattern
}

theory Positive:V_pos{
   //The pattern is a vertex-induced subgraph of the template:
   ∀x [vertex] : (pattern_node(x) ⇒ template_node(x))
        ∧ (∀ y [vertex] : pattern_edge(x, y) ⇔ (template_edge(x, y) ∧ pattern_node(x) ∧
      pattern_node(y))) ∧
        ∧ pattern_label(x) = template_label(x).
   //The pattern is a connected subgraph of the template: From every node in the pattern,
   //There exists a path to every other node in the pattern.
   ∀x [vertex] y[vertex] : x ≠ y ∧ pattern_node(x) ∧ pattern_node(y) ⇒ connected(x, y).
   {
      connected(x, y) ← pattern_edge(x, y) ∨ pattern_edge(y, x).
      connected(x, y) ← ∃z[vertex] : connected(x, z) ∧ connected(z, y).
   }

   //Existence of a homomorphic f from the pattern to example graph with graphid gid.
   ∀gid[graphid] : ∀x[vertex]   : homo_with(gid) ∧ pattern_node(x) ⇔ ∃ y[vertex] : y=f(
     gid,x).
   ∀gid[graphid] : ∀x [vertex] y[vertex] : homo_with(gid) ∧ pattern_node(x) ∧
     pattern_node(y) ∧ x≠y ⇒ f(gid, x) ≠ f(gid,y).
   ∀gid[graphid] : ∀x [vertex] y[vertex] : homo_with(gid) ∧ pattern_node(x) ∧
     pattern_node(y) ∧ pattern_edge(x,y) ⇒ example_edge(gid, f(gid,x), f(gid,y)).
   ∀gid[graphid] : ∀x[vertex] : homo_with(gid) ∧ pattern_node(x) ⇒
       pattern_label(x) = example_label(gid, f(gid,x)).

   //At least N homomorphisms must be found
   #{ gid [graphid] : homo_with(gid) } >= N+.
}
```

However, the IDP solver has no obligation to maximize the number of homomorphisms it finds for `f`, only to satisfy the constraints. Thus, it can choose `f` such that it does not represent a homomorphism for a graph `g`$\in \mathbb{G}_-$. As our constraints are satisfied, we are led to believe that our pattern candidate is a valid pattern.

It follows from results by Immerman [79] that this is inherently linked to IDPs limit to Existential Second-Order. Indeed, checking that our pattern $\mathcal{P}$ is homomorphic with no more than N_- negative graphs is equivalent with checking that enough negative examples `G` exist for which no homomorphism exists (**Listing** 3.9). This clearly leads to a universal quantification over a function variable, which IDP cannot express.

Listing 3.9: Quantifying over functions locally.

```
1 #{g | g ∈ 𝔾₋ ∧ ∀ f : f is not a homomorphism from 𝒫 to g}
```

A way to work around this is by encoding the dual (i.e., negated) problem, and conclude that the problem is satisfied if and only if no model exists for the dual problem. This can be checked using an NP solver. However, this technique can only be implemented in IDP by writing two theories:

- one (positive) theory $\mathcal{T}^+$ (see **Listing** 3.8), which expresses the positive homomorphic property and generates pattern candidates, and
- one negative theory $\mathcal{T}^-$, shown in **Listing** 3.10, which expresses the (dual of) negative homomorphic property and rejects pattern candidates that do not satisfy this constraint.

In IDP, one must provide procedural code that ties these two theories and their inferences together, allowing pattern candidates to be communicated between them.

Canonicity As graph isomorphism is known to be in NP (recent research suggests it is in the Quasi-Polynomial complexity class QP [7]), the isomorphism restriction when looking for multiple patterns is no more complex than coNP. Therefore, we can use the same technique of encoding the dual and performing a satisfiability check that must fail for the canonicity requirement.

However, it is at this point we take into account the evaluation strategy. As mentioned above, having two separate theories means that we must tie the inferences together using procedural code. This can be done using the Lua interface made available by IDP. However, using this interface, we cannot prevent having to reground the theory every time an inference is performed.

Listing 3.10: IDP specification handling the negative constraint of the graph mining problem.

```
vocabulary V_neg{
  type vertex isa nat
  type label
  type graphid

  //Predicates describing the pattern graph
  pattern_node(vertex)
  pattern_edge(vertex, vertex)
  pattern_label(vertex):label

  //Predicates describing the negative example graphs
  example_edge(graphid, vertex, vertex)
  example_label(graphid, vertex):label
  N_: int

  partial f(graphid, vertex):vertex //Represents the homomorphisms with the example graphs
  homo_with(graphid) //True for graphs for which f represents a correct homomorphism
}

theory Negative:V_neg{
   //Existence of a homomorphic f from the pattern to example graph with graphid gid.
   ∀gid[graphid] : ∀x[vertex]   : homo_with(gid) ∧ pattern_node(x) ⇔ ∃ y[vertex] : y=f(
     gid,x).
   ∀gid[graphid] : ∀x[vertex] y[vertex] : homo_with(gid) ∧ pattern_node(x) ∧
     pattern_node(y) ∧ x≠y ⇒ f(gid, x) ≠ f(gid,y).
   ∀gid[graphid] : ∀x[vertex] y[vertex] : homo_with(gid) ∧ pattern_node(x) ∧
     pattern_node(y) ∧ pattern_edge(x,y) ⇒ example_edge(gid, f(gid,x), f(gid,y)).
   ∀gid[graphid] : ∀x[vertex] : homo_with(gid) ∧ pattern_node(x) ⇒
       pattern_label(x) = example_label(gid, f(gid,x)).

   //Can we find more than N_ homomorphisms
   #{ gid [graphid] : homo_with(gid) } >= N_.
}
```

Consequently, to minimize the number of times that we must reground the theories, we choose to introduce a separate theory $\mathcal{T}^{iso}$ (shown in **Listing** 3.11) for the canonicity constraint, which *generates* all isomorphic patterns by finding values for a unary predicate `pattern/1` representing a pattern isomorphic with `pattern_node/1`, such that two functions `f/1` and `g/1` can be found that are each others inverse and that satisfy the conditions of homomorphisms; i.e., they preserve edges and labels.

This way, after finding a pattern candidate and checking the positive and negative homomorphism restriction, we generate all isomorphic patterns and subsequently introduce additional clauses in the candidate generation process which prohibit these patterns from becoming candidates.

Listing 3.11: IDP specification handling the canonicity constraint of the graph mining problem.

```
vocabulary V{
  type vertex isa nat
  type label

  //Predicates describing the pattern graph
  pattern_node(vertex)
  pattern_edge(vertex, vertex)
  pattern_label(vertex):label

  //Predicates describing the template
  template_node(vertex)
  template_edge(vertex, vertex)
  template_label(vertex):label

  //Predicate describing an isomorphic pattern
  pattern(vertex)

  partial f(vertex):vertex //The homomorphism from pattern_node to pattern.
  partial g(vertex):vertex //The homomorphism from pattern to pattern_node.
}

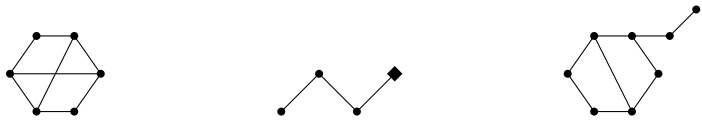
theory iso:V{
  //f and g only have an image for vertices in pattern_node / pattern respectively.
  ∀x [vertex] pattern_node(x) ⇔ ∃ y: y = g(x).
  ∀x [vertex] pattern(x) ⇔ ∃ y: y = f(x).

  ∀x [vertex]: pattern(x) ⇒ pattern_node(f(x)).
  ∀x [vertex]: pattern_node(x) ⇒ pattern(g(x)).

  //f and g preserve edges
  ∀x [vertex] y [vertex]: pattern_node(x) ∧ pattern_node(y) ∧ template_edge(x, y)    ⇒
     template_edge(g(x), g(y)).
  ∀x [vertex] y [vertex]: pattern(x) ∧ pattern(y) ∧ template_edge(x,y) ⇒ template_edge(
     f(x), f(y)).
  //f and g preserve labels
  ∀x [vertex]: pattern_node(x) ⇒ template_label(x) = template_label(g(x)).
  ∀x [vertex]: pattern(x) ⇒ template_label(x) = template_label(f(x)).
  //f and g are injective
  ∀x [vertex] y [vertex]: x < y ∧ pattern(x) ∧ pattern(y) ⇒ f(x) ≠ f(y).
  ∀x [vertex] y [vertex]: x < y ∧ pattern_node(x) ∧ pattern_node(y) ⇒ g(x) ≠ g(y).
  //f and g are each others inverse
  ∀x [vertex] : pattern(x) ⇒ g(f(x)) = x.
}
```

Visualising the Constraints

Retaking the example graph mining instance from **Section** 3.3 (See **Figure** 3.5), which consisted of 1 positive and 1 negative example, together with a template graph, and setting the graph mining parameters $N_+ = 1, N_- = 0$, we illustrate how the different constraints affect the possible patterns. We consider a subset of the candidate pattern space in **Figure** 3.6.



(a) Positive Example (b) Negative Example (c) Template Graph

Figure 3.5: (Repeat) A graph mining instance with $(N_+ = 1, N_- = 0)$.

First, **Lines** 28–31 of $\mathcal{T}^+$ (**Listing** 3.8) ensure that patterns are vertex-induced subgraphs of the template. This consists of three subconditions: that (**Line** 29) the nodes of a candidate pattern are a subset of the nodes the template graph, that (**Line** 30) if the template features an edge between two selected nodes, the candidates must feature this edge as well, and that (**Line** 31) a node's label remains unchanged.

With the pattern candidates visualized in **Figure** 3.6, this constraint prunes candidate 3.6a, as it misses the diagonal edge in the hexagon present in template graph 3.5c, failing the second subcondition. It also prunes candidate 3.6d, because the rightmost node's label has changed into a diamond, and candidate 3.6e as it has too many diagonal edges, again failing the second subcondition.

Lines 34–38 prune any candidates that are not connected, such as 3.6c.

Lines 41–48 prune candidates that do not occur often enough in the positive examples (for this toy instance, at least once). The constraints of **Lines** 41–45 ensure that in case `homo_with(gid)` is true, the following holds: a mapping (1) must exist, (2) it must preserve inequality, (3) it must preserve the patterns edges, and (4) it must preserve labels, respectively. **Line** 48 specifies that enough homomorphisms must exist (at least one). As a result, candidates 3.6d and 3.6e are pruned, as no mapping exists for 3.6e to positive example 3.5a that preserves the diamond label and no mapping for 3.6d preserves all diagonal edges.

Lastly, the negative theory $\mathcal{T}^-$ (**Listing** 3.10) uses the same constraints as $\mathcal{T}^+$ **Lines** 41–48 to find patterns that are homomorphic with too many negative

examples. These constraints prune 3.6f, as a possible mapping was shown earlier, in **Section** 3.3, **Figure** 3.3.

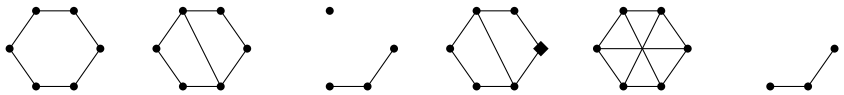


Figure 3.6: A subset of the pattern space for **Figure** 3.5.

Solving the graph mining problem using IDP

Now that we have modelled the graph mining problem in IDP, we also want to *use* this model to solve graph mining problems. As our model consists of multiple theories, we must use procedural code to tie together different inferences on the different theories, to eventually produce the correct answers. From the discussion above, we identify three main theories:

- $\mathcal{T}^+$ that generates a pattern, satisfying the positive homomorphic constraint,
- $\mathcal{T}^-$ that checks the negative homomorphic constraint, and
- $\mathcal{T}^{iso}$, that generates all isomorphic patterns.

The entire procedural loop can then be described as follows:

1. We first ask IDP for a model of $\mathcal{T}^+$.
2. We extract from this model the pattern candidate (i.e., the value of `pattern_node/1`, `pattern_edge/2` and `pattern_label/1`) and, using the satisfiability inference, check whether it satisfies $\mathcal{T}^-$; note that as $\mathcal{T}^-$ encodes the dual of the negative homomorphic constraint, failing the satisfiability check means the negative homomorphic constraint is satisfied and vice versa.
3. Regardless of whether the generated candidate satisfied $\mathcal{T}^-$, we let $\mathcal{T}^{iso}$ generate all its isomorphic patterns. These can be transformed to clauses that, when added to $\mathcal{T}^+$, prevent generation of isomorphic pattern candidates: such clauses state that it must not be true that a future generated pattern consists of exactly those nodes.
4. We repeat this process until the necessary number of patterns was found or the search space was exhausted.

3.5.2 ASP

The **ASP** language is closely related to IDP. An ASP encoding consists of a set of rules, which allow us to derive the head of a rule whenever its body is true. The head and body of a rule can contain variables, as long as every variable is *safe*, meaning it occurs positively in the body. In this case the head can be derived for any assignment to the variables that makes the body true.

One of the main differences between ASP and IDP is the choice of semantics: ASP looks for the answer set models, whereas IDP looks for well-founded models. Leveraging the minimality property of answer sets, ASP can prevent the invalid models of the example discussed in *Issue 3: local* $\forall$ *quantification over functions*, without creating two separate theories or writing procedural code. Instead, it relies on an encoding technique called the *saturation* technique [46], which we will discuss in **Section** 3.5.2 when we discuss how to encode the negative homomorphic property.

Another difference between ASP and IDP is that the former generally only allows *uninterpreted* functions, which can be viewed as constructors that bring structure in data. However, our specification of the graph mining problem features many *interpreted* functions, i.e., those representing a homomorphism between two graphs. Luckily, we can represent n-ary functions such as `pattern_label/1` using an $n+1$-ary predicate, and express functionality constraints explicitly.

Modeling the graph mining problem in ASP

Listing 3.13 shows the ASP specification of the graph mining problem. Note that we use the same naming scheme `pattern_node/1`, `pattern_edge/2` and `pattern_label/2`, and introduce the constants `np` and `nm` to correspond to the problem parameters N_+ and N_-, respectively.

We identify the same three issues for ASP as we had for IDP namely *representing graphs*, *local* $\exists$ *quantification over functions* and *local* $\forall$ *quantification over functions*. Due to the close relation between IDP and ASP, it is not surprising that the first two issues are once again solved using the same global disjoint union technique as explained in the sections discussing the *representation of graphs* and the *local* $\exists$ *quantification over functions* in IDP.

However, as ASP does not allow functions, we represent n-ary functions such as `pattern_label/1` using an $n+1$-ary predicate, and express functionality constraints explicitly.

Generating pattern candidates As in **Listing** 3.8, we will first specify that the pattern is a vertex-induced subgraph of the template (**Lines** 5–7):

- First, we open up the `pattern_node/1` predicate using a choice rule. (**Line** 5)
- we state that every edge in the template between two pattern nodes implies a corresponding edge in the pattern.
- we specify that every pattern node preserves the unique label it had in the template.

As patterns must be connected, we include a set of rules and constraints expressing that every node of the pattern must be connected to every other node of the pattern (**Lines** 10–14). As with IDP, for the earlier introduced toy example from **Figure** 3.5, these constraints filter candidates 3.6a, 3.6d, and 3.6e because they do not represent vertex-induced subgraphs, and 3.6c as it violates connectedness.

Positive homomorphisms First, we will look at the positive homomorphic constraint, specifying the necessary number of homomorphisms with positive examples. First, we guess for every positive example graph whether a homomorphism exists using a choice rule (**Line** 17), and represent these graphs using `homo_with(G)`. For every positive graph `G` with a homomorphism, we create a mapping `f(G,X,V)` relating a graph id `G` and pattern node `X` with *exactly one* example node `V` (**Line** 18). We introduce constraints such that, for each positive example graph with a homomorphism, the mapping must be injective and must preserve edges as well as labels (**Lines** 20–22). We conclude the positive constraint by specifying that the number of mappings that correspond with a homomorphism should be higher or equal to our threshold N_+ (**Line** 24). Again, this constraint filters candidates 3.6d and 3.6e.

Negative homomorphisms We now look at how to encode the negative homomorphic constraint, which specifies that the number of homomorphisms with negative graphs does not exceed N_-. To encode this in the same model, we use the *saturation* technique [46]: a powerful encoding technique that can check whether a certain property $\mathcal{P}$ holds for **all** possible interpretations. This corresponds to formulas of the form $\exists X \forall Y$, where the quantifications are second-order, which captures complexity class Σ_2^P-complete. The technique relies on the fact that every answer set is a minimal model.

Specifically, the technique extends the traditional guess-and-check paradigm of Answer Set modeling; the guess-and-check paradigm identifies two parts in every ASP model, i.e., $\mathtt{P}_{guess}$ and $\mathtt{P}_{check}$, that respectively add rules to *guess* a

solution and to *check* the guessed solution. Encodings using saturation add a third part, usually referred to as $\mathtt{P}_{sat}$. First, the modeler must devise an answer set M_{sat} such that all other possible answer sets candidates M are a subset of M_{sat}. Then, in $\mathtt{P}_{sat}$ the modeler must write the rules necessary to extend or *saturate* any answer set candidate in which the property $\mathcal{P}$ is found to hold. Minimality of answer sets now ensures that the answer set candidate M_{sat} is an answer set if and only if the property holds for *all* guesses.

We first illustrate this technique on the well-known example of three-colorability of a graph given by a `node/1` and an `edge/2` predicate. Answer set candidates will provide an interpretation of the predicate `color/2` assigning colors to nodes; if the graph is three-colorable, every node will be assigned exactly one color. If, however, every assignment of colors to nodes breaks the rules of a coloring - this is our universal property $\mathcal{P}$ - we want to derive answer set M_{sat}, which assigns every node all three colors at once. Note that every candidate coloring is indeed a subset of M_{sat}.

Looking at the model of **Listing** 3.12, we see that $\mathtt{P}_{guess}$ (**Line** 2) guesses a possible coloring using disjunction. Next, $\mathtt{P}_{check}$ (**Lines** 5–6) checks whether the coloring is valid, i.e., neighbors have different colors and no two colors for a single node. It derives `saturate` if the guessed coloring is *not* valid. Finally, $\mathtt{P}_{sat}$ (**Lines** 9–11) ensures that any invalid guesses derive the unique model M_{sat}, ensuring that it is the *only* answer set iff all possible guesses are invalid.

In summary, given any interpretation of the predicates `node/1` and `edge/2`, if a valid coloring exists every answer set will correspond to a valid covering and vice versa. If, on the other hand, no valid coloring exists, M_{sat} will be the unique answer set of **Listing** 3.12.

Listing 3.12: Saturation encoding for three-colorability, as taken from [46].

```
1 %P_guess
2 color(X,red) | color(X,green) | color(X,blue) :- node(X).

4 %P_check
5 saturate :- edge(X,Y), color(X,C), color(Y,C).
6 saturate :- node(X), color(X,C1), color(X, C2), C1 != C2.

8 %P_sat
9 color(X,red) :- node(X), saturate.
10 color(X,blue) :- node(X), saturate.
11 color(X,green) :- node(X), saturate.
```

To apply the saturation technique in the negative homomorphisms check, we guess an assignment for `f` such that in every negative example graph `G` each pattern node `X` is mapped to at least one example node `V` (**Line** 27). It is important to note that this formulation does not prevent that `f` maps `X` to *more* than one example node; as we will see shortly, this is essential for the saturation

technique.

The next rule (**Line** 28) corresponds to $\mathtt{P}_{sat}$. In this rule we express that if we can derive `saturated(G)` for a specific graph `G`, we will saturate `f` for `G` by mapping every template node `X` to every example node `V`.

We finish the saturation encoding by providing $\mathtt{P}_{check}$. $\mathtt{P}_{check}$ derives `saturated(G)` whenever our guess for `f` does not represent a homomorphism (**Lines** 32–36). Possible reasons are that the mapping is not injective, does not preserve edges, or does not preserve labels.

To conclude the encoding of the negative homomorphic constraint, we specify that the pattern is allowed to be homomorphic with at most N_- negative examples (**Lines** 38–39). In our toy example, this prunes candidate 3.6f.

Canonicity The same saturation technique can be applied to the isomorphism restriction, making it possible to model the entire graph mining problem in a single model. We also introduce the notion of a *lexicographical ordering* of graphs, based on the natural order of the nodes: we presume the `#max` and `#min` aggregates on nodes are defined, and a successor predicate `succ/2` is available that holds for any two nodes `a`, `b` s.t. `b` immediately follows `a` in this natural order. A graph $\mathcal{G}$ is lexicographically smaller than a graph $\mathcal{G}$' if the smallest node not shared between $\mathcal{G}$ and $\mathcal{G}$' is a node of $\mathcal{G}$. We can now say that a pattern $\mathcal{P}$ is canonical if it is the *lexicographically smallest* graph among all its isomorphic graphs. The main idea is that for every choice for `pattern_node/1`, we want ASP to find an isomorphism with another subset of template nodes that is lexicographically *smaller* (i.e., a counterexample for the statement that `pattern_node/1` is canonical). If ASP cannot find such an isomorphism, we saturate the answer set. Thus, saturated answer sets correspond to choices for `pattern_node/1` s.t. *no lexicographically smaller* isomorphic graph exists, which are exactly the canonical patterns.

Looking at our model (**Lines** 42–45), to enforce canonicity we again guess a relation, in this case `iso/2`. Semantically, `iso/2` is the predicate representation of a function between nodes of the pattern and nodes of a hypothetical, different and *canonical* form of that same pattern. Such a function does not exist if the pattern itself is, in fact, canonical: In that case `iso/2` will be saturated.

Because we only want answer sets that correspond to canonical patterns, after including the saturation rule $\mathtt{P}_{sat}$ (28) for `iso/2`, we add a constraint saying that all answer sets must be saturated. Furthermore, we create two helper predicates:

- `isoNode/1` which is true for those nodes in the image of `iso/2`, and

- `compl/1` which is true for those nodes *not* in the image of `iso/2`.

Because in some situations we will saturate `iso/2`, we cannot define `compl/1` as `compl(X):-not isoNode(X)`, as this would make the resulting saturated answer set unstable (The rules deriving `compl` would disappear from the Gelfond-Lifschitz reduct).

Therefore, we define by induction a helper predicate `codCT(X,Y)` which is true iff `X` is *not* the *unique image* of `Y` or any smaller node. Note that we do this *without negation of any saturated symbols* (`iso/2`, `isoNode/1`).

1. This trivially holds for the smallest node `F` if it is not a pattern node, as then `F` is not in the domain of `iso/2` (**Line** 51).
2. This holds for the smallest node `F` if `iso/2` maps `F` to a node differing from `X` (**Line** 52).
3. This holds by induction for the tuple (`X`,`Y`) if it holds for the node preceding `Y` and either `Y` is not a pattern node (**Line** 53) or `Y` is mapped to a node differing from `X` (**Line** 54).

We can now define `compl` as the nodes `X` s.t. `codCT` holds for the largest node, i.e., it is not the image of the highest node or any below it.

Next, we must specify when we saturate ($\mathtt{P}_{check}$). This occurs whenever `iso/2` does not represent an isomorphism (because it does not preserve edges, labels, or is not injective), or when `iso/2` represents an isomorphism with a graph that is *not lexicographically smaller*. To encode this last condition, we define by induction (**Lines** 71–75) a predicate `identity_below/1` which indicates that up to, but not including a certain node, `pattern_node/1` and `isoNode/1` are identical. Now, whenever there exists a node such that `pattern_node/1` and `isoNode/1` are identical up to that node, and that node itself is part of `pattern_node/1` but not of `isoNode/1` (expressed by the complement `compl/1`), we must saturate. Likewise, we must saturate whenever `pattern_node/1` and `isoNode/1` are identical, which we handle in **Lines** 66 and 74–75.

Saturation technique Saturation as a technique is a powerful way of including constraints that are expressed using formulas with second-order universal quantification (i.e., corresponding to Σ_2^P decision problems). Examples of such constraints are the introduction of negative example graphs or the canonicity of patterns as discussed above, but also other constraints that impose a preference order on patterns, e.g., maximality (prevents patterns that are subsets of other patterns) or coverage (orders patterns by comparing the set of matched positive patterns using subset ordering).

While the *saturation technique* successfully prevents the need of a procedural loop for such constraints, it is clear that this technique is not derived from a natural KR translation of the Graph Mining definition. For instance, one of the pitfalls when using saturation encodings is the use of negations in P_{guess} and P_{check}, which can easily break the encoding by making the saturated model unstable [119]. As a result, one must take care when using aggregates, for example, as these advanced language constructs introduce negations in their translation. To alleviate some of these concerns, efforts have been made to automate saturation [48].

Solving the graph mining problem using ASP

As ASP is able to represent the graph mining problem using a single model, solving the graph mining problem using ASP is pretty straight-forward. However, it is important to note that by default, finding a different homomorphism between a canonical pattern and an example would lead to a different answer set. However, as the pattern itself is the same, this is in fact not a new solution. As such, we must limit the answer sets to those that differ for their choice of pattern nodes. Using a solver such as clingo, this is possible by enabling *answer set projection* [61], which limits the different answer sets to those that differ on a specific set of facts. By specifically projecting to the facts representing the pattern nodes, we obtain the desired behavior.

Listing 3.13: ASP using the saturation technique.

```
#const nm = N_.
#const np = N+.

% Patterns are vertex-induced subgraphs of the template
0 { pattern_node(X) } 1 :- template_node(X).
pattern_edge(X, Y) :- pattern_node(X), pattern_node(Y), template_edge(X, Y).
pattern_label(X, V) :- pattern_node(X), template_label(X, V).

% Patterns are connected
connected(X) :- #min{Y : pattern_node(Y)}=X.
connected(Y) :- connected(X), pattern_edge(X, Y), X != Y.
connected(Y) :- connected(X), pattern_edge(Y, X), X != Y.

:- pattern_node(X), not connected(X).

% Positive homomorphic constraint:
{ homo_with(G) } :- positive(G).
1 { f(G, X, V) : example_node(G, V) } 1 :- homo_with(G), pattern_node(X).

:- homo_with(G), pattern_node(X), pattern_node(Y), X != Y, example_node(G, V), f(G, X, V),
      f(G, Y, V).
:- homo_with(G), f(G, X, V1), f(G, Y, V2), template_edge(X, Y), not example_edge(G, V1, V2
    ), pattern_node(X), pattern_node(Y).
:- homo_with(G), pattern_node(X), f(G, X, V), pattern_label(X, L), example_label(G, V, L2)
     , L != L2.

:- #count{G:homo_with(G)} < np.
```

```
% Negative homomorphic constraint:
f(G, X, V) : example_node(G, V) :- pattern_node(X), negative(G). % P_guess
f(G, X, V) :- saturated(G), pattern_node(X), example_node(G, V). % P_sat

% The following lines describe the reasons for a graph to be saturated (P_check):
% We cannot map two different pattern nodes to the same example node.
saturated(G) :- negative(G), f(G, X, V), f(G, Y, V), X != Y, pattern_node(X), pattern_node
      (Y).
% The mapping must preserve edges.
saturated(G) :- negative(G), template_edge(X, Y), f(G, X, V1), f(G, Y, V2), not
      example_edge(G, V1, V2), pattern_node(X), pattern_node(Y).
% The mapping must preserve labels.
saturated(G) :- negative(G), template_node(X), f(G, X, V), template_label(X, L),
      example_label(G, V, L2), L != L2.

neg_homo_with(G) :- not saturated(G), negative(G).
:- #count{G:neg_homo_with(G)} > nm.

% Canonicity constraint:
iso(X, V) : template_node(V) :- pattern_node(X). % P_guess
iso(X, V) :- pattern_node(X), template_node(V), sat. %P_sat
:- not sat.


isoNode(V) :- iso(X, V).
compl(X) :- template_node(X), codCT(X, M), M=#max{Z:template_node(Z)}.

%codCT(X,Y): X is not the image of iso for Y or any node below it
codCT(X, F) :- template_node(X), not pattern_node(F), F=#min{Z:template_node(Z)}.
codCT(X, F) :- template_node(X), iso(F, Y), Y!=X, F=#min{Z:template_node(Z)}.
codCT(X, B) :- template_node(B), succ(A, B), codCT(X, A), not pattern_node(B).
codCT(X, B) :- template_node(B), succ(A, B), codCT(X, A), iso(B, Y), Y!= X.

% iso must preserve edges
sat :- iso(X, W), iso(Y, Z), template_edge(X, Y), not template_edge(W, Z).
sat :- iso(X, W), iso(Y, Z), not template_edge(X, Y), template_edge(W, Z).
% iso must preserve labels
sat :- iso(X, Y), template_label(X, L1), template_label(Y, L2), L1!=L2.
% iso must be injective
sat :- iso(X, Y), iso(X, Z), Y!=Z.
% The inverse of iso must be injective
sat :- iso(X1, Y), iso(X2, Y), X2!=X1.
sat :- identity_below(X), pattern_node(X), compl(X).
sat :- identity_below(sup).

% identity_below(X) iff every node below (not including) X is either
% - in the iso candidate and the pattern, or
% - not in the iso candidate nor in the pattern.
identity_below(M) :- #min{X:template_node(X)}=M.
identity_below(X) :- template_node(X), succ(Y, X), identity_below(Y), pattern_node(Y),
      isoNode(Y).
identity_below(X) :- template_node(X), succ(Y, X), identity_below(Y), not pattern_node(Y),
       compl(Y).
identity_below(sup) :- identity_below(M), #max{X:template_node(X)}=M, pattern_node(M),
      isoNode(M).
identity_below(sup) :- identity_below(M), #max{X:template_node(X)}=M, not pattern_node(M),
       compl(M).

#show pattern_node/1.
```

3.5.3 Comparative Summary

In **Section** 3.4.3, we have identified a set of desirable properties that a good KR specification should satisfy. **Table** 3.1 summarizes how IDP and ASP score with respect to these properties.

Property	IDP	ASP
1. Graph as a single object	No: Global disjoint union technique	No: Global disjoint union technique
2. Independence of homomorphisms	No: Global disj. union & partial function	No: Global disj. union & partial function
3. Similarity of $\geq$ and $\leq$ constraint	Partial: Similar but theory splitting required	No: Requires saturation technique
4. Multiple patterns (isomorphism)	No: theory splitting required	Yes: Using saturation technique
5. Connectedness	Yes: Using inductive definitions	Yes: Using ASP rules
6. Multiple inferences	Yes: Model checking, expansion, minimization	Yes: Model checking, expansion, minimization
7. Single solver call	No: Two calls, one model per pattern	Partial: One answer set per pattern

Table 3.1: Summary of the desirable properties in IDP and ASP.

3.5.4 Performance experiments

In this section, we will investigate the performance of both the IDP as well as the ASP model. These experiments were performed on a Ubuntu 16.04 LTS system with an Intel i7-4770 CPU @ 3.40GHz, with 8GB RAM, on which IDP (version 3.7.0) and Clingo (version 5.2.2) were installed. Every experiment was run with an 8 GB memory limit and a 20 hour time limit.

We created graph mining problem instances from two well-known machine learning datasets [123]: mutagenesis and yoshida. These datasets consist of a set of labeled graphs representing labeled molecules;

- in *yoshida* 265 molecules are ranked according to their bioavailability, and
- in *mutagenesis*, 230 molecules are trialed for their mutagenicity on Salmonella.

Most discussions of state-of-the-art specialized algorithms do not use any negative examples. However, the mutagenesis dataset allows us to characterize

92 molecules as 'negative', specifically those who inhibit the mutability of Salmonella (i.e., a mutagenicity of ≤ 1). As our specifications easily accommodate a dataset with negative examples, we create graph mining instances from the *yoshida* and *mutagenesis* datasets by randomly selecting one positively labeled graph from the dataset to serve as the *template* required for our solution. Next, we chose values for N_+ and N_-: For *yoshida*, which only has positive examples, we chose an N_+ value of 26 (10% of the dataset). For *mutagenesis* we chose an N_+ of 90 (66% of the positive examples) and 30 (33% of the negative examples). We used both the ASP and the IDP solution to compute 120 *canonical* (i.e., non-isomorphic) patterns.

Results

Yoshida: The results for the *yoshida* dataset are visualized in **Figure** 3.7 (Note the need for different scales). Since IDP as a byproduct, can list all isomorphic solutions for each canonical pattern, we have cross-validated the results of both solvers.

It is clear that the ASP solution, which utilizes a single solver process, outperforms the IDP system, which must repeatedly ground the same problem due to the interaction of generating pattern candidates satisfying the positive homomorphic constraint and checking canonicity of pattern candidates. As the *yoshida* dataset only contains positive examples, we can also illustrate the performance of state-of-the-art specialized algorithms. However, specialized algorithms such as gSpan generally mine *all* patterns, and do this without requiring a specific template graph: it can use any graph in the database as a template. As such, we have mined all patterns for our *yoshida* instance; this takes about 1.39 seconds for gSpan [115]. Comparatively, when Clingo (ASP) and IDP are asked to mine all patterns, ASP takes about 282.5 seconds, whereas IDP does not finish within the time limit of 20 hours.

Mutagenesis: When mining the mutagenesis dataset for patterns with $N_+ = 90$ and $N_- = 30$, IDP does not find any patterns before the time limit of 20 hours has passed. The results for ASP are shown in **Figure** 3.8, which mines 120 patterns in a little under four hours.

Discussion

It is clear from these experiments that state-of-the-art specialized algorithms outperform our declarative solutions by several orders of magnitude. However,

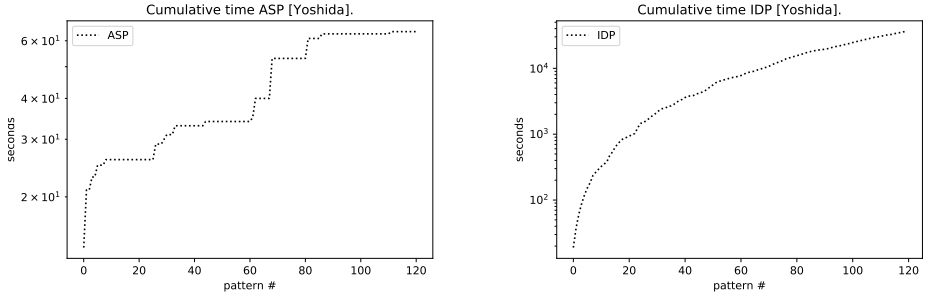


Figure 3.7: Cumulative run times for the Yoshida dataset.

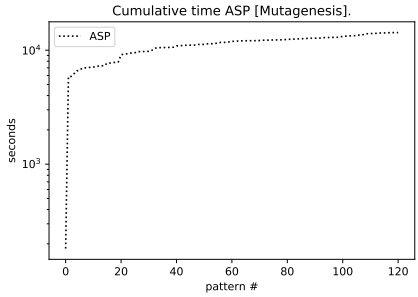


Figure 3.8: Cumulative ASP run time for the mutagenesis dataset.

our declarative solutions can easily be extended to support negative examples, such as those in the mutagenesis dataset, whereas specialized algorithms require an extensive overhaul. This is an example of the level of Elaboration Tolerance [105] that declarative languages exhibit. This gives declarative approaches a great benefit in use cases where performance is not the primordial factor, for example while prototyping or when requirements frequently change.

These experiments also show a clear divide between the ASP language and the IDP language. While ASP solvers can encode Σ_2^P-complete constraints using the *saturation* technique, IDP must resort to multiple solver instances tied together using procedural code. As a result, IDP must ground the problem repeatedly and, as communication between the solver instances is limited, is not able to learn valuable information about when a candidate will or will not pass the checks performed by different solver instances. On the other hand, constraints featuring second-order universal quantification are only supported by ASP using the advanced *saturation* technique, reducing the 'naturality' of the encoding.

In the next section, we reconsider the higher-order model of **Section** 3.4, and look how declarative systems, in particular IDP, can introduce support for such higher-order models.

3.6 Solver Techniques

In the previous sections, we have established that (1) an intuitive encoding of the graph mining problem exists using higher-order logic, and that (2) encoding techniques are required to express the problem in a specification language based on first-order logic. Regrettably, these encoding techniques decrease the graph mining model's intuitiveness and can be a significant hurdle for the modeller. This section investigates the encoding techniques encountered in **Section** 3.5 so as to identify ways to generalize these techniques and integrate them in the solver, effectively shifting the burden of these techniques from modeller to solver by supporting higher-order specifications. Specifically, integration techniques are suggested for the state-of-the-art IDP-system, which currently uses a typical ground-and-solve technique [87]. In a ground-and-solve system, two distinct phases can be identified: in the first phase, all quantifications are instantiated such that the encoding does not contain any variables, while in the second a SAT-solver finds a model for the resulting ground instance. In general, techniques that support higher-order logic will interleave these two phases in various degrees.

One concern typically raised in the context of solver for higher-order logic is that rising language expressivity will go hand in hand with decreasing performance. From a theoretical point of view, this is clearly a valid concern, however our hypothesis is that, in practice, expressing real-world problems using higher-order logic does not necessarily include a performance loss with respect to their previous first-order encodings. Furthermore, the additional structure expressed in higher-order encodings might even allow for a performance gain. Specifically, an important aspect through which we think a higher-order encoding, when supported with the right solver techniques, can in increase performance is *independence analysis*, i.e., the discovery of independent subproblems. Some support for the claim that better independence analysis will lead to better performance can be found on the propositional level, in recent work [101, 116] from the Quantified Boolean Formulas (QBF) research community. This work shows the benefit of estimating or learning the dependencies between quantifications of propositional variables. Writing down knowledge in a more expressive language such as higher-order logic leads to the availability of additional structure and latent constraints in the knowledge specification. Often, using the additional structure available, we already express many

Listing 3.14: Excerpt of the HO specification of graph mining.

```
1 homomorphism((N1, E1, L1), (N2, E2, L2)) ←
2     (∃SO F [N1:N2]: (∀ x, y [N1]: x ≠ y ⇒ F(x) ≠ F(y)) ∧
3      (∀ x [N1] y [N1]: E1(x, y) ⇒ E2(F(x), F(y))) ∧
4      (∀ x [N1]: L1(x) = L2(F(x)))).
5  ...
6 (#{ Pos : positive(Pos) ∧ homomorphism((N,E,L), Pos) } ≥ N+)
7  ...
```

interesting (in)dependencies. Consider the following two examples: First, when we existentially quantify over a (set of) higher-order object(s) satisfying a constraint, we can look for this (set of) object(s) independently from the larger problem. Second, in the case of a universal quantification of a higher-order object, we can check the relevant constraints for every possible instantiation of this higher-order quantification separately.

Returning to the higher-order modeling of graph mining, for example, it is clear that the question of whether two specific graphs match is a subproblem which can be solved *independently* of other matchings. This independence is signalled by the quantification over graphs `(N,E,L)` (**Line** 6 of **Listing** 3.14), even though it is hidden in the aggregate expression counting `homomorphisms`. Further evidence of the independence can be found in the definition of `homomorphism/2`, as it uses only two types of symbols: locally quantified symbols and predicate arguments. A smart solver should analyse the higher-order specification to detect and exploit these (in)dependencies, and, when discussing solver techniques, we will pay specific attention to how this can be achieved.

3.6.1 Nested Solvers

As was pointed out in **Section** 3.5, two of the main issues with higher-order encodings for systems such as IDP are (1) the ∀ quantification over higher-order objects such as functions and predicates, and (2) the occurrence of local quantification, both existential as well as universal. The technique of *nested solvers* addresses both these issues: when presented with a universal quantification or any local quantification over a function or predicate, the solver can spawn another, secondary instance of itself which is identical except for the specification being solved. Indeed, this second instance, also called an *oracle* is initialised with that part of the original specification where the quantified symbol is in scope, possibly transformed to an existential quantification.

At the propositional level, Bogaerts et al. [17] recently explored the idea of solving QBF instances using nested SAT solvers supporting CDCL, with favourable

results. Their underlying idea is to use the identity $\forall \overline{x} : \phi_1 \Leftrightarrow \neg\exists \overline{x} : \neg\phi_1$ to transform arbitrary formulas to the form $\exists \overline{x} : (\phi \ \wedge \neg\exists \overline{y} : \psi)$. They show how the top solver can perform standard SAT-solver propagation on ϕ, and how the second solver (the oracle) can check whether there exists an assignment $\overline{y}$ satisfying ψ. It is important to note that as the transformation above introduces negations in front of the quantification $\exists \overline{y}$, the oracle call is performed in a negative context: models $\mathcal{M}$ of $\mathcal{O}_\psi$ are transformed into conflicts and learned clauses $\mathcal{C}$ for the top solver. This approach effectively construes a QBF solver, and as predicted, this technique can exploit (in)depencies. While QCDCL [102] traditionally limits the order in which variables can be decided to the order in which they are specified in the quantifier prefix, this restriction is not necessary when working with nested solvers, opening up research tracks on the effects of eager versus lazy calling of the nested solver, or the effect of splitting up variables in a single quantifier prefix level if they are independent.

This technique, taken from the propositional level, can be recreated at the predicate level by rewriting and splitting theories and handing them off to separate, nested solvers functioning as oracles. As on the propositional level, we are faced with the same trade off between eager and lazy calling of nested solvers. Another research challenge is the transformation of models $\mathcal{M}$ to learned clauses $\mathcal{C}$: as multiple models for the oracle's theory can exist, one can follow different approaches for transforming one or more models of the theory into a learned clause.

On the level of predicates, the nested solver technique is closely related to "modulo-theory" frameworks such as *SAT-modulo-theory* or *ASP-modulo-theory* [59]. These frameworks offer a way of injecting *procedural code* for complex problems using *global constraints*. One example is the injection of prefix-projection [89] in recent declarative approaches to sequential pattern mining [4]. By contrast, the nested solvers technique does not inject *procedural code*, instead it injects another solver instance (an oracle). Thus, in the nested solver approach, even the injected knowledge is specified declaratively, and, in a full implementation, the split points between the levels of solvers are introduced without involvement of the user.

Nested solvers in Graph Mining: Turning back to graph mining, and looking only at the positive homomorphism requirement and the non isomorphism requirement, we identify three different strategies for introducing oracle calls to the graph mining problem. These three strategies correspond to different options for splitting the graph mining specification, and we will call these strategies the **monolithic**, the **semi decomposed** and the **fully decomposed** strategies. All three are visualized in **Figure** 3.9, where every (sub)solver or oracle call

is represented as an IDP block, and the different positive example graphs are labeled as Ex1, Ex2 and Ex3. In each case, the main or top-level solver is the leftmost IDP block, responsible for generating candidates.

- The **monolithic** strategy is the default strategy as explained in **Section** 3.5.2. As such, this strategy could only gain from a tighter integration between solver instances, which would allow reuse of grounding and efficient communication of learned clauses.

  This strategy splits off the *generation* of a pattern candidate and *checking the positive homomorphism constraint*, from the *non isomorphism check*. It thus consists of only two solver instances: one that takes the template and all example graphs, subsequently producing a pattern candidate satisfying the positive constraint; and one that, using the other patterns, checks whether a pattern candidate is isomorphic to an already discovered pattern. This second solver then reports back to the first, and the necessary clauses are generated to prevent generating the same pattern candidate again.

- The **semi decomposed** strategy further splits off the *generation* of a pattern candidate from the *test* phase where the solver checks whether the pattern candidate is homomorphic with sufficiently many positive example graphs. Based on the outcome of this check, the solver either reports back to the first solver, which can register that the pattern candidate was not a valid pattern and generates the necessary clauses to prevent regeneration, or it passes the pattern candidate on to a third solver performing the non isomorphism check as in the case of the **monolithic** strategy.

- the **fully decomposed** strategy exploits the independence between the different positive example graphs; it introduces a separate oracle call for each example graph, reporting the results of the checks to an aggregation unit. This aggregation unit then reports back to either the first solver or to another solver performing the non isomorphism check as in the case of the **monolithic** strategy.

Note that all three strategies split the theory on points where quantifications are used in the theory; in fact, they each correspond to a *splitting strategy*:

- The **monolithic** approach splits *only* when encountering a *second-order universal* quantification, which would put the problem outside of the expressive power of a conventional SAT solver.
- The **fully decomposed** approach splits the theory when encountering *any* second-order quantification; this includes the existential second-order quantification present in the definition of `homomorphism/2` in **Listing** 3.14.

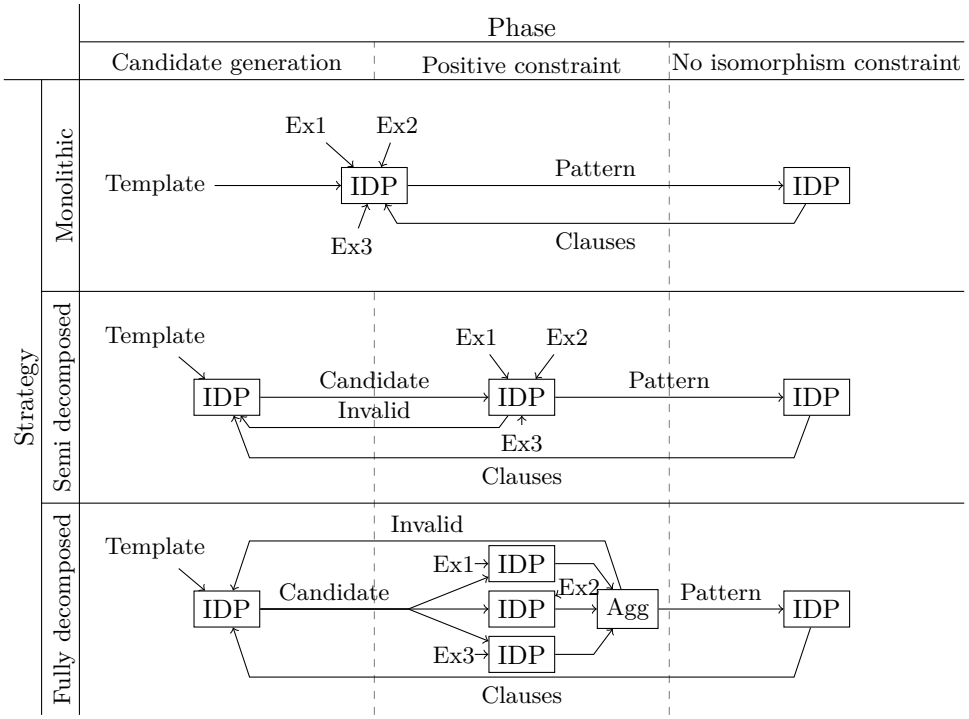


Figure 3.9: The three different strategies proposed for subsolvers.

- The **semi decomposed** approach splits the theory on the outermost quantification for any rule containing *any* second-order quantification.

As it is possible that after splitting, the theory being split off still contains formulas with second-order quantifications, the splitting rules must be performed recursively.

Experiments: To get an idea of the performance of the nested solvers technique, and whether it might lead to some performance gains, we mimicked the implementation of a nested solver, without implementing a fully functional nested solver. By introducing a 'pipeline' of separate IDP3 call instances, our implementation replicates the different oracle calls in the nested solver approach, while being specifically tailored to graph mining. It uses an imperative language to manage the different calls and modify the models $\mathcal{M}_\psi$ of oracle calls to new clauses $\mathcal{C}_\psi$ added to subsequent calls.

For each strategy proposed above, we have introduced a corresponding 'pipeline'

in the experiments. The experiment is set up such that every pipeline mines a certain amount of patterns from a dataset. Like most imperative solutions, they do this in a fixed order: first patterns of length n are mined, starting with $n = 2$, raising the length of the mined patterns to $n+1$ whenever all patterns of length n are mined. This fixed order allows exploitation of an *anti-monotonicity* property often used by imperative solutions: *Whenever a pattern candidate fails the positive homomorphism check, every extension of this pattern candidate will also fail the positive homomorphism constraint.*

This property can easily be encoded as additional knowledge in a higher-order specification of the graph mining problem, as it defines a predicate `isPattern/1` representing whether or not a graph $\mathcal{G}$ is a pattern. When we look at the proposed strategies, both the **semi** and **fully decomposed** pipeline capture the necessary information for exploiting the *anti-monotonicity* property: they signal both the valid as well as the invalid pattern candidates through the imperative interface managing the different oracle calls. However, the **monolithic** pipeline does not; invalid pattern candidates are discarded internally in the solver instance handling candidate generation and the positive constraint. As a result, the invalid pattern candidates of length n cannot be used by the **monolithic** pipeline to additionally filter the candidate generation when it starts searching for patterns of length $n + 1$. As a result, the **monolithic** pipeline does not exploit the *anti-monotonicity* property, but also does not impose a search direction, which can become an advantage for certain datasets.

Dataset generation & specifications: To test the performance of all three pipelines, we have reused the *yoshida* dataset from **Section** 3.5.4, and have modified the *mutagenesis* dataset by labeling all molecules as *positive*. This modification is motivated by two key insights:

1. Specialized algorithms do not feature negative examples,
2. When splitting the model into multiple theories solved by separate oracles, the theory for negative examples is the same as that for the positive examples. The only difference is how the satisfiability results are handled: for negative examples, an UNSAT is handled as a SAT for the positive examples and vice versa.

Lastly, we have also created a graph mining problem from the well known *bloodbarr* dataset [99], where 413 molecules are ranked according to the degree to which the molecule can cross the blood-barrier stream.

We have reused the approach described in **Section** 3.5.4, and ran experiments using the same machine and time/memory limits.

Results: **Figure** 3.10 shows the resulting cumulative runtimes for each of the pipelines on a log-scale y-axis, and a boxplot of the time spent mining each pattern for the **fully decomposed** and the **semi decomposed** pipelines. Our boxes cover the data from the first quartile (Q1) to the third quartile (Q3), while the whiskers extend to the last datum less than $Q3$ plus 1.5 times the interquartile range (IQR). All other data points are considered outliers, and are plotted as individual dots. A horizontal dotted line indicates the median. For the Bloodbarr datasets, no results for the **monolithic** pipeline could be given, as it exceeded the memory limit of 8GB.

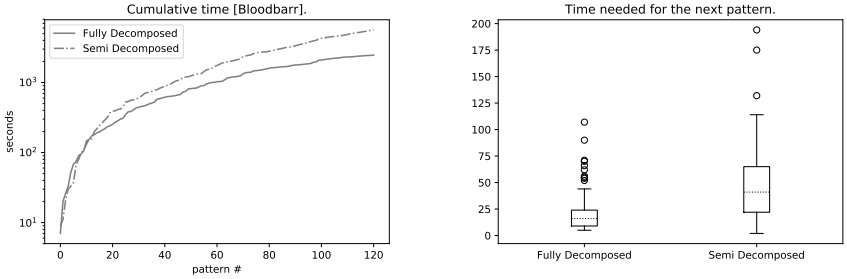
Focussing on the difference between the **semi** and **fully decomposed** pipelines, all three datasets (**Figure** 3.10b) show a similar factor of two difference in favor of the **fully decomposed** pipeline. The difference between the pipelines that use oracles for the positive constraint on the one hand (the **semi** and **fully decomposed** pipelines), and the **monolithic** pipeline on the other hand, suggests that a large benefit can be achieved from using a separate oracle for the checking phase.

Furthermore, the difference between the **semi** and **fully decomposed** pipelines shows that the benefit of introducing oracles, at least in graph mining, increases when we further introduce an oracle call for each independent graph. Recall that the possibilities for decomposition in the graph mining case are found by syntactical analysis; they correspond to second-order quantifications, and their position in the hierarchy relative to each other and other, first-order quantifications. In fact, this is why we advocate the use of local quantifications, as opposed to having to quantify all second-order symbols in the vocabulary. This syntactical argument suggests that finding good decompositions for other problems based on the presence of second-order quantifications is feasible.

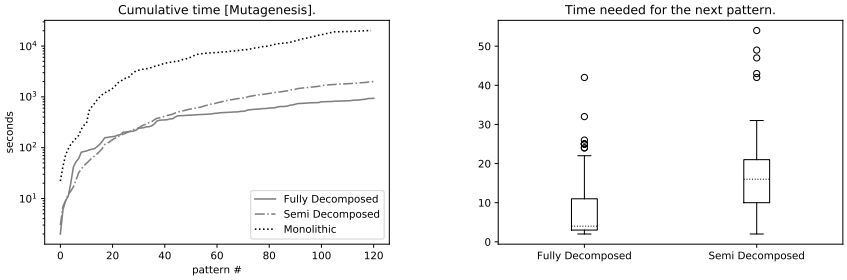
In fact, the **semi** and **fully decomposed** strategies can mine 120 patterns from the yoshida dataset in 1633 and 1255 seconds respectively. Likewise, for the mutagenesis dataset, these strategies mine the 120 requested patterns in 1996 and 939 seconds respectively. While this is still an order of magnitude larger than ASP, we note that in these experiments we focussed on how many oracles should be introduced and where, and as a result, IDP must still repeatedly ground each theory.

3.6.2 Lazy Grounding

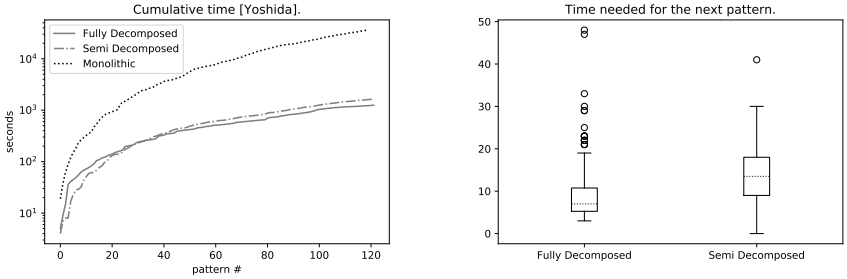
The third issue with higher-order encodings, as pointed out in **Section** 3.5, concerns the data representation of sets of higher-order objects, such as graphs in the *graph mining* problem. While the *disjoint union* technique proposed in **Section** 3.5 can be used, even automating the rewrite so it is no hinderance



(a) IDP results for the bloodbarr dataset, with N_+=41.



(b) IDP results for the mutagenesis dataset, with N_+=23.



(c) IDP results for the yoshida dataset, with N_+=26.

Figure 3.10: Cumulative runtimes and time spent per pattern by IDP for the three datasets with only positive examples.

for modellers, one problem is that it tends to produce very large groundings. Furthermore, the earlier introduced nested solvers technique results in a system that has to ground not just the main theory, but also has to ground theories for every nested solver, while retaining as many grounding optimization techniques as possible. This problem could be mitigated by using the *lazy grounding* technique.

Lazy grounding is a technique where theories are only grounded partially: only those parts of the theory that are relevant to finding a satisfying assignment are grounded. The framework of de Cat et al. [38], for example, uses the concept of justifications to denote a way of generating a complete assignment for non-grounded parts of the theory. It then suffices to ground parts only if it is not possible to construct a justification for them anymore. An experimental algorithm for model expansion with lazy grounding based on this framework has been implemented in IDP.

Experiments: To get an indication of the impact of lazy grounding, we repeated the experiments explained above while enabling lazy grounding for every IDP call covering the 'positive constraint checking' phase. Note that in this case, the disjoint union technique has already been applied manually. This would allow the solver to defer the grounding of example graphs until they are necessary to satisfy the positive constraint. In the 'Ground and solve' setup, the **fully decomposed** pipeline is able to prevent grounding and solving some example graphs by eagerly evaluating the aggregation and stopping as soon as the threshold is reached, giving it a clear advantage over the **semi decomposed** pipeline, which has to ground the entire problem first. When using lazy grounding, we expect the **semi decomposed** pipeline to behave more like the **fully decomposed** pipeline, as it should be able to bypass any grounding for unnecessary graphs.

Results: **Figure** 3.11 shows for each dataset a histogram of the time needed to mine the next pattern for the **semi decomposed** and **fully decomposed** pipelines with ground and search, and the **semi decomposed** pipeline with lazy grounding. These figures show that the lazy grounding option actually causes a slowdown for the **semi decomposed** pipeline, quite frequently needing significantly more time to check a pattern, as evidenced by the long tail of the **semi decomposed** pipeline with lazy grounding. **Figure** 3.12 shows, for the mutagenesis dataset, the size of grounding as the number of literals (3.12a) and the memory usage of the **semi decomposed** pipeline in kilobytes with and without lazy grounding (3.12b). While **Figure** 3.12a shows that lazy grounding produces smaller groundings, **Figure** 3.12b shows that the effective memory

usage using lazy grounding, while in general smaller, sometimes exceeds that of the default ground & solve option.

One possible cause for the apparent slowdown caused by the lazy grounding option is the setup cost of lazy grounding, which can be high: When using lazy grounding, additional data structures are required. Another possible factor is the 'penalty' incurred in the experimental implementation when lazy grounding has to ground an additional graph, w.r.t eagerly grounding all patterns at the same time. Further evidence for this factor can be found by noting that the slowdown with lazy grounding is less dramatic for the mutagenesis and yoshida datasets. From **Figure** 3.13, which shows a boxplot of the number of example graphs that had to be inspected before accepting or refuting a pattern candidate for each dataset[4], we can conclude that these datasets are in some sense 'easier', as the **fully decomposed** pipeline on average has to inspect fewer graphs per pattern candidate for these datasets than for the bloodbarr dataset.

3.7 Discussion and Future Work

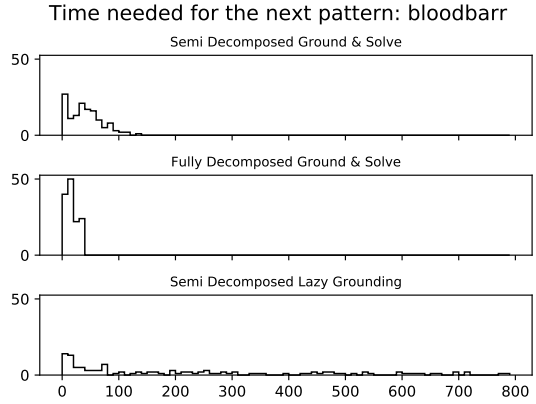
Looking at the limitations and shortcomings of the typical techniques to encode higher-order logic in KR systems such as IDP and ASP, e.g., *disjoint union* technique and *saturation*, we see opportunities for adding support for higher-order logic to state-of-the-art KR systems.

While the expressiveness of higher-order logic in general raises concerns about the performance of systems supporting higher-order logic, we suggested that in real-world applications, higher-order logic might open up new ways for solvers to benefit from structure in problems, for example, through independence analysis. To investigate this trade-off, in **Section** 3.6, we experimented with two techniques for adding higher-order logic support to KR systems, while paying attention to how they might aid such independence analysis: The first technique, *nested solvers*, was concerned with supporting universal quantification and local quantifications of higher-order objects, while the second, *lazy grounding*, was mainly concerned with issues surrounding data representation.

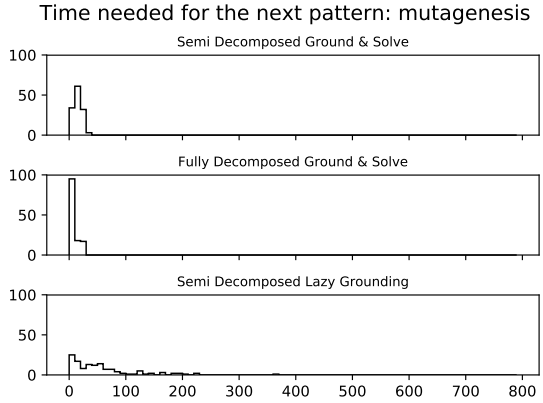
For *nested solvers* we experimented with an implementation in imperative code that should mimic nested solvers in two different settings (**fully decomposed** and **semi decomposed**) and have found that both settings hold a clear advantage over the first-order **monolithic** setting.

For *lazy grounding* we experimented with its existing experimental implementation in IDP, turning it on for the **semi decomposed** setting. While

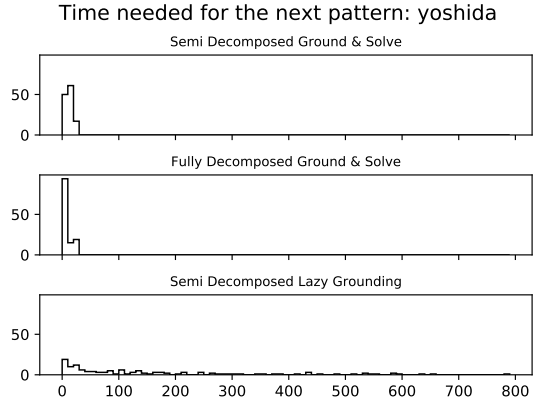
[4]Numbers taken from runs of the **fully decomposed** pipeline.



(a) Lazy grounding effects for IDP on the bloodbarr dataset, with N_+=41.

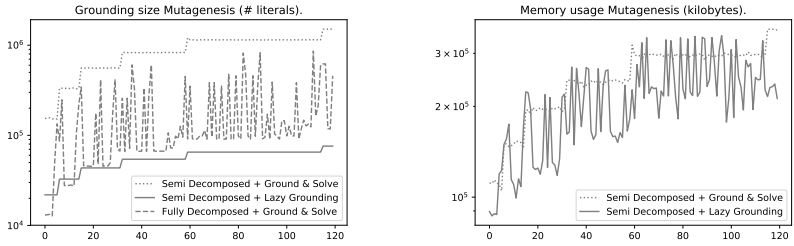


(b) Lazy grounding effects for IDP on the mutagenesis dataset, made strictly positive, with N_+=23.



(c) Lazy grounding effects for IDP on the yoshida dataset, with N_+=26.

Figure 3.11: Histograms of time needed to mine the next pattern by IDP. Only strictly positive datasets were used.



(a) Grounding size for pattern check in Mutagenesis dataset, as number of literals.

(b) Memory usage for pattern check in Mutagenesis dataset, in kilobytes.

Figure 3.12: Grounding size (#lits) and memory usage (kilobytes) of Ground & Solve and Lazy Grounding approaches in Mutagenesis dataset.

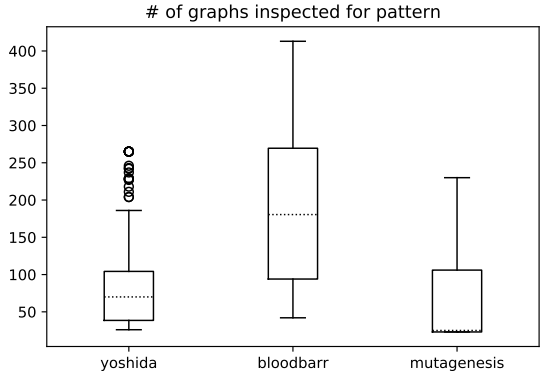


Figure 3.13: Boxplot: # of example graphs inspected before accepting or refuting a pattern candidate.

we had thought that *lazy grounding* would bring the **semi decomposed** setting closer to the **fully decomposed** setting, we actually found that, in its current implementation, *lazy grounding* actually causes a slowdown.

It is clear that more research with respect to existing techniques and systems is needed. Thus, we identify three potential paths for going forward: (1) we build upon the idea of nested solvers, implementing syntax based decomposition techniques introducing stacks of subsolvers automatically, not only for graph mining but for other higher-order logic specifications as well, (2) we ground to *quantified Boolean formulas* (QBF) as an alternative approach to nested solvers, and/or (3) we further explore *lazy grounding*. These last two paths are discussed in more detail.

3.7.1 Grounding to QBF

Existing state-of-the-art KR systems are commonly based on SAT solvers. One other option to introduce higher-order support is by using Quantified Boolean Formula or QBF solvers instead.

These solvers accept formulas of the form

$$\forall x_1 \exists x_2 \forall x_3 \ldots Q_n x_n : \phi(x_1, x_2, x_3, \ldots, x_n)$$

where $\forall x_1 \exists x_2 \forall x_3 \ldots Q_n x_n$ is called the *quantifier prefix* and $\phi(x_1, x_2, x_3, \ldots, x_n)$ represents an unquantified Boolean formula. To test the performance of a KR-system based on a QBF solver, we would first implement a grounding system from higher-order expressions to QBF formulas, making use of the fact that we can ground quantification over a predicate by quantifying over the ground atoms representing the predicate.

We can then use existing QBF solvers to solve the resulting ground formulas. By employing a solver which uses a dependency learning technique [116], it is possible to solve these formulas by assigning atoms a value without taking into account their order in the quantifier prefix.

Such a system can derive a set of independencies using the same syntactical analysis of the higher-order specification proposed earlier. Then, instead of starting with the empty set, these derived dependencies can be used to bootstrap a QBF solver supporting dependency learning [116] (e.g., DepQBF).This way, we can again leverage additional independencies evident in the higher-order specification while possibly deriving more, perhaps otherwise unidentified dependencies on the propositional level.

3.7.2 Lazy Grounding

As mentioned in **Section** 3.6.2, we expected that enabling lazy grounding for the **semi decomposed** pipeline would prevent the grounding and solving of some of the example graphs, and that, as a result, the **semi decomposed** pipeline would achieve results very close to the **fully decomposed** pipeline. Instead, our experiments showed an overall slowdown of the **semi decomposed** pipeline when using lazy grounding. Possible explanations are a high setup cost for the general method of lazy grounding or a high penalty being incurred every time an additional graph has to be grounded.

Metrics such as the 'hardness' of each dataset, which expresses how many example graphs on average are needed to accept or refute a pattern candidate, can give some indication towards the cause of the overall slowdown. However, for a detailed analysis changes to both the experiments as well as the lazy grounding implementation are needed.

It is important to note that other approaches exist to implement lazy grounding [32, 141] in ASP, for example lazy constraints, where a set of constraints $\mathcal{C}$ is identified which causes large grounding. This set $\mathcal{C}$ is *not grounded*, instead, a solution candidate is generated without it and this solution candidate is subsequently checked w.r.t. the constraints in $\mathcal{C}$. If any of the constraints in $\mathcal{C}$ is violated, a (ground) conflict is learned. We remark that this is similar to the way that the **semi** and **fully decomposed** pipelines split candidate generation from checking the positive (and negative) constraints. As such, we would expect that a good implementation of this technique would indeed show similar performance gains as the introduction of these pipelines, without the need of procedural code, and this can be the target of future research.

3.8 Conclusion

In this chapter, we introduced the Graph Mining, and showed that there is a straightforward transformation from its mathematical definition (**Section** 3.3) into a higher-order specification (**Section** 3.4).

We subsequently looked at how IDP and ASP, two state-of-the-art KR Languages allow one to model the graph mining problem, employing techniques such as *disjoint union technique* to encode *higher-order* predicates, and, in the case of ASP, *saturation* technique to encode *second-order* quantifications. Most notably, the fact that second-order quantification cannot be encoded in IDP stands at odds with its underlying *knowledge base paradigm*, as we must write procedural

code to perform multiple solver calls, which fixes the flow of information and the inference(s) used.

We then set up an experiment (**Section** 3.6) to investigate the hypothesis that the expressiveness of higher-order logic can actually be of benefit to solvers by allowing independence analysis, looking at different ways in which an approach based on nested solvers could decompose the search problem.

Lastly, we suggest two future lines of research, one based on *Lazy Grounding* as a way to minimize the grounding for quantifications over elements of a higher-order predicate, encoded through the disjoint union technique, the other focussed on grounding to QBF as an alternative for nested solvers, to support second-order quantifications. It is the latter line of research that we will investigate further in **Chapter** 4.

Chapter 4

A Second-Order Language and its Grounder

This chapter is a reworked version of a publication in the proceedings of the "Sixteenth International Conference on Principles of Knowledge Representation and Reasoning" [133]. Some modifications in spelling and presentation were made, some terminology was adjusted to be in line with the remainder of this thesis, and small errors were corrected.

Personal contribution: 100%.

4.1 Introduction and Related Work

In the spirit of Kowalski's seminal paper *Predicate Logic as a Programming Language* [93, 136], many KR languages are based on logic: some are propositional, some First-Order. In recent years, ASP has clearly shown the applicability of modeling problems in logic and handing them to a solver on real-world business use cases, for example, planning robot tasks in warehouses [62].

Yet, there is a demand for more powerful and richer KR languages. To this end, we propose Second-Order (SO) Logic as a modeling language. SO Logic allows many more problems to be expressed than ASP, FOL or propositional logic; it has a descriptive complexity of PH [80]. *SAT* and *ground (disjunctive) ASP* however have a descriptive complexity of NP and Σ_2^P respectively. As such, it is infeasible to ground SO to SAT or ground (disjunctive) ASP. Instead,

we propose to ground to *Quantified Boolean Formulas* (QBF), which have a descriptive complexity of PSPACE.

To our knowledge, the only other ground-and-solve tool with SO as a modeling language is SAT-TO-SAT [17] and its grounder SO2GROUNDER. Our system improves on SAT-TO-SAT by:

- Lifting a syntactical restriction: after the first FO quantification $\mathcal{Q}x$, SO2GROUNDER no longer allows SO quantifications of a different quantifier.
- Interfacing with arbitrary QBF solvers using the widely accepted QDimacs format.

In Constraint Programming (CP) languages, we find the ESSENCE [56] language, which features an expressive type system that allows for arbitrary nestings of sets and supports quantifications. Analysis by Mitchell and Ternovska [108] suggests that the ESSENCE language effectively has a descriptive complexity of at least ELEMENTARY, although they remark that no complete formal semantics for the language have been published. Contrary to our solver, however, ESSENCE modelings distinctively specify the direction in which reasoning is performed, e.g., when expressing reachability in a graph, the model depends on whether one wants to find the reachability relation or whether one wants to *generate* so-called *connected graphs*.

Lastly, there are formal specification languages such as TLA [97] and Event-B [3]. These languages extend predicate logic with set theory, allowing for higher-order quantifications. The ProB [98] system, implemented in SICSTus Prolog is a constraint solver and model checker for the Event-B language. Generally, these higher-order quantifications and features are handled by ProB's default Constraint Logic Programming (CLP) backend, with only limited involvement of its SAT backend Kodkod [132], potentially missing out on recent performance improvements of SAT technology. However, in recent work, Krings et al. [95] have reported promising results in a first attempt towards integrating the SAT and CLP backends.

4.2 Second-Order Logic

In this section we introduce SO logic over a vocabulary V, and show how we intend to use it as a modeling language.

4.2.1 Syntax

Variables are symbols representing either an element, a predicate P/n, or a function f/n. We call n the *arity* of the predicate or function.

When a variable represents an element, we call it a *first-order (FO)* variable. Variables representing either a predicate or a function are called *second-order (SO)* variables.

Terms are either a variable, the application of an n-ary function symbol f or a variable representing a function to n terms, or a predicate symbol P/n or function symbol f/n.

Atoms are n-ary predicate symbols P or variables representing a predicate applied to n terms.

Formulas are defined inductively using the following rules:

- All atoms are formulas.
- The negation ($\neg$) of a formula is a formula.
- If ϕ and ψ are formulas, the conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$) and equivalence ($\Leftrightarrow$) of ϕ and ψ are formulas, with $\phi \Rightarrow \psi$ and $\phi \Leftrightarrow \psi$ shorthand for $\neg\phi \vee \psi$ and $(\neg\phi \vee \psi) \wedge (\phi \vee \neg\psi)$ respectively.
- If x is a variable, and ϕ is a formula, then $\exists x : \phi$ and $\forall x : \phi$ are formulas, with $\forall x : \phi$ shorthand for $\neg\exists x : \neg\phi$. Note that no restrictions are imposed on whether x is a first-order or second-order variable.
- The pre-interpreted symbols $=$,$\neq$,$<$, and $\leq$ applied to two terms are formulas.

An example of a second-order formula is $\exists f : \forall x : \forall y : x \neq y \Rightarrow f(x) \neq f(y)$, where $\neq$ (and $=$) is a pre-interpreted predicate.

For modeling convenience, and to distinguish first-order from second-order variables, we introduce *types* to second-order logic. Predicates and functions are associated with types $(T_1, \ldots, T_n)$ and $T_1, \ldots, T_n \to T$ respectively. Variables are typed corresponding to the symbol they represent, and we write $x :: T$ to mean x is typed as T.

Note that a *first-order* quantification over an element x of type T is written as $\forall x :: T$, while a *second-order* quantification over a *predicate* P of elements from T is written as $\forall P :: (T)$, the difference being the parentheses surrounding T.

We say a formula is properly typed if all of its subformulas are properly typed. An application of a predicate p (function f) is properly typed if the

predicate (function) has type $(T_1, \ldots, T_n)$ (for functions, $T_1, \ldots, T_n \to T$) and its arguments x_1 to x_n have types T_1 to T_n. With typing, we could introduce the types *Color* and *Country*, and the above example formula becomes: $\exists f :: \mathit{Country} \to \mathit{Color} : \forall x :: \mathit{Country} : \forall y :: \mathit{Country} : x \neq y \Rightarrow f(x) \neq f(y)$.

4.2.2 Semantics

To define the semantics of second-order logic, we extend interpretations $\mathcal{I}$.

Specifically, a structure $\mathcal{I}$ for second-order logic must be extended with:

- an assignment of an n-ary relation over the domain for every n-ary predicate (and thus, second-order) variable, and
- an assignment of a total n-ary function over the domain for every n-ary function variable.

Furthermore, now that types are introduced, the assignments by $\mathcal{I}$ must respect the type of every symbol and variable.

The valuation function $(\cdot)^{\mathcal{I}}$ for terms t is extended for function *variables* applied to terms:

- If t is a function variable f of arity n applied to n terms $t_1, \ldots, t_n$ then $t^{\mathcal{I}} = f^{\mathcal{I}}(t_1^{\mathcal{I}}, \ldots, t_n^{\mathcal{I}})$.

The satisfaction relation $\models$ between structures $\mathcal{I}$ and formulas ϕ is modified as well.

Definition 15. *Given a formula ϕ and a structure $\mathcal{I}$, we define the satisfaction relation inductively based on the syntactical structure of ϕ:*

- $\mathcal{I} \models P(t_1, \ldots, t_n)$ *where P is an n-ary predicate symbol or predicate variable iff* $(t_1^{\mathcal{I}}, \ldots, t_n^{\mathcal{I}}) \in P^{\mathcal{I}}$.
- $\mathcal{I} \models \neg\phi$ *iff* $\mathcal{I} \not\models \phi$.
- $\mathcal{I} \models \phi \wedge \psi$ *iff* $\mathcal{I} \models \phi$ *and* $\mathcal{I} \models \psi$.
- $\mathcal{I} \models \phi \vee \psi$ *iff* $\mathcal{I} \models \phi$ *or* $\mathcal{I} \models \psi$ *(possibly both).*
- $\mathcal{I} \models \exists x :: T : \phi$ *iff*
  1. *T is a basic type and there is a $d \in dom(\mathcal{I})$ of type T s.t. $\mathcal{I}[x : d] \models \phi$, or*
  2. *T is a predicate type with arity n and there is an n-ary relation r over $dom(\mathcal{I})$ of type T s.t. $\mathcal{I}[x : r] \models \phi$, or*
  3. *T is a function type with arity n and there is an n-ary function f over $dom(\mathcal{I})$ of type T s.t. $\mathcal{I}[x : f] \models \phi$.*
- $\mathcal{I} \models t_1 = t_2$ *iff* ${t_1}^{\mathcal{I}} = {t_2}^{\mathcal{I}}$.

- $\mathcal{I} \models t_1 < t_2$ *iff* ${t_1}^{\mathcal{I}} < {t_2}^{\mathcal{I}}$ *where the ordering* $<$ *on the domain* $dom(\mathcal{I})$ *is extended pointwise to functions over* $dom(\mathcal{I})$ *and expresses the subset relationship for relations over* $dom(\mathcal{I})$.

4.2.3 SO Logic as a Modeling Language

Every Second-Order Logic modeling consists of three main parts: The *vocabulary*, the *structure* and the *theory*.

Vocabulary The *vocabulary* declares the types used in the modeling, and can declare predicates and functions. If these predicates or functions are not subsequently defined in the structure, they are understood to be implicitly quantified existentially on the outermost level, being in scope for the entire *theory*. Types can be declared to be a subset of the integers using the syntax "`as int`". The type `Bool` is added automatically to represent propositions.

Structure The *structure* assigns to each type T a domain $\mathcal{D}$, and can interpret certain predicates and functions that were declared in the *vocabulary*. As such, it is the perfect place to put instance-specific information. Currently, only *two-valued* structures are allowed, i.e., a structure must always fully specify which tuples are true/false for the predicates and functions it interprets.

Theory The *theory* consists of a set of properly typed SO formulas with no *free* (unquantified) *variables*. We call these formulas *sentences*. A theory is satisfied iff every *sentence* in the theory is satisfied.

By including the type of *integers* in every *structure*, and extending the *vocabulary* with (in some cases, partial) functions representing $+$, $-$, $*$, and $/$, we include support for *arithmetic functions*. We furthermore extend the *vocabulary* with constants for every integer, and fix the interpretations of these constants and the earlier introduced partial functions to their standard interpretations.

Example 1 (Strategic Companies). *The* Strategic companies *problem is a well-known example of a* Σ_2^P*-hard [24] problem. It models the conundrum of a large holding owning multiple companies, forced to downsize by selling off a company. Of course, the company wants to minimize the impact of selling this company. Specifically, two conditions have to be met:*

1. *The sale should not impact the holdings portfolio, i.e., it should produce the same set of goods.*

2. *Some sets of companies in the holding together control another company, this is called a* controlling set*. As such, it is possible to sell off a company while retaining control of it through a controlling set. In this case, the holding is not allowed to 'downsize' by selling this company.*

Definition 16 (Strategic set). *We call a set of companies SC controlled by the holding a* strategic set *if (1) the companies in SC together produce all goods, (2) SC is closed under ownership through controlling sets, and (3) none of its subsets $Y \subset SC$ satisfies both condition (1) and condition (2).*

The Strategic Companies problem was featured in the ASP Competition 2013 [77], where the challenge was to determine for two distinct companies c, c' whether a strategic set SS exists containing both c and c'. A pair of companies satisfying this requirement was called a strategic pair*.*

Some aspects of higher-order logic arise when modeling the strategic companies problem, specifically when choosing how to represent controlling sets*. While this could be encoded using a tagged union approach, in the ASP competition the additional restriction that "every controlling set contains at most 4 companies" was imposed instead.*

We model the *Strategic Companies* in **Listing** 4.1. We impose the same restriction as the ASP competition to allow a more direct comparison with ASP specifications. In **Lines** 1–6 we specify the *vocabulary*, containing the `Company` and `Good` types together with predicates to represent the *controlling sets* (`cont`), who produces what (`prod`), the strategic set (`ss`) and the strategic pair (`sp`). **Lines** 8–12 continue by specifying the *structure*, which contains instance specific data. We conclude with **Lines** 14–17, specifying the *theory*: The strategic set must contain the strategic pair (**Line** 14), it must produce all goods (**Line** 15), be closed under ownership through controlling sets (**Line** 16) and it must be subset-minimal (**Line** 17): no *strict subset* of `ss` must exist for which conditions (1) and (2) hold. Note that this model contains two second-order quantifications: ∃`ss` (implicitly) in the vocabulary and ¬∃`ss'` on **Line** 17.

4.3 QBF

A *quantified boolean formula* (QBF) is a formula in quantified propositional logic where variables can be quantified either existentially or universally, e.g.

$$\forall x : \exists y : (x \vee y \Rightarrow \exists z : y \vee z).$$

Listing 4.1: SO model for Strategic Companies.

```
1 type Company.                                                    // All companies
2 type Good.                                                       // All products
3 cont         :: (Company, Company, Company, Company, Company). // Controls: last is
      controlled
4 prod         :: (Company, Good).                                 // Produces
5 ss           :: (Company).                                       // Strategic Set
6 sp           :: (Company, Company).                              // Strategic pairs

8 Company := {Barilla;Dececco;Callippo;Star}.
9 Good    := {Pasta;Tonno}.
10 prod    := {Barilla,Pasta; Dececco,Pasta; Callippo,Tonno; Star,Tonno}.
11 cont    := {Star,Star,Star,Star,Barilla; Barilla,Barilla,Barilla,Barilla,Dececco}.
12 sp      := {Barilla,Callippo}.

14 ∀c :: Company : ∀c1 :: Company : sp(c,c1) ⇒ (ss(c) ∧ ss(c1)).
15 ∀g :: Good : ∃p :: Company : ss(p) ∧ prod(p, g).
16 ∀c :: Company : (∃o1 :: Company : ∃o2 :: Company : ∃o3 :: Company : ∃o4 :: Company : ss(
      o1) ∧ ss(o2) ∧ ss(o3) ∧ ss(o4) ∧ cont(o1,o2,o3,o4,c)) ⇒ ss(c).
17 ¬(∃ss' :: (Company) : (ss' ≠ ss) ∧ (∀c :: Company : ss'(c) ⇒ ss(c)) ∧ (∀g :: Good : ∃p
       :: Company : ss'(p) ∧ prod(p, g)) ∧ (∀c :: Company : (∃o1 :: Company : ∃o2 ::
      Company : ∃o3 :: Company : ∃o4 :: Company : ss'(o1) ∧ ss'(o2) ∧ ss'(o3) ∧ ss'(o4)
      ∧ cont(o1,o2,o3,o4,c)) ⇒ ss'(c))).
```

Quantifiers can alternate indefinitely and it is this property that makes deciding satisfiability for QBF PSPACE-complete, whereas deciding satisfiability for (unquantified) SAT formulas is NP-complete.

When each variable is quantified at the *beginning* of the formula, we say that it is in *prenex* form. Such formulas can be written as $\mathcal{Q}_1\overline{x}_1 \ldots \mathcal{Q}_n\overline{x}_n.\phi$ with $\mathcal{Q}_1, \mathcal{Q}_n$ quantifiers. We generally call $\mathcal{Q}_1\overline{x}_1 \ldots \mathcal{Q}_n\overline{x}_n$ the *quantifier prefix*, also written $\hat{\mathcal{Q}}$. We group subsequent quantifications of the same quantifier into quantifier *blocks* and define the *level($\mathcal{Q}_i$)* of a block $\mathcal{Q}_i$ to be the number of quantifier blocks preceding it. As such, the *highest* level or *innermost* quantification block is the one most to the right in the formula.

When the formula ϕ in $\hat{\mathcal{Q}}.\phi$ is in *Conjunctive Normal Form* (CNF) we say the QBF formula is in *Prenex Conjunctive Normal Form* (PCNF). QDimacs, the input encoding that most QBF solvers accept, is in fact a textual representation of a QBF formula in PCNF.

4.4 Implementation

We now discuss the implementation of a grounder from SO to QBF. We now describe our initial approach by listing, in order, the transformations $\phi \rightsquigarrow \psi$, implemented in the system as rewritings:

Push Negations Using the rules $\neg\exists x.\phi \rightsquigarrow \forall x.\neg\phi$, $\neg(\phi \wedge \psi) \rightsquigarrow \neg\phi \vee \neg\psi$, etc. we ensure that negations only appear in front of atoms.

Unnesting Using the rule $f(g(\overline{x})) = z \rightsquigarrow \exists y.f(y) = z \wedge g(\overline{x}) = y$, we remove function applications from positions where terms are expected.

Second-order Comparisons and (In)equalities We rewrite comparisons ($<$, $\leq$) and (in)equalities ($=$, $\neq$) over second-order variables to equivalent first-order formulas, e.g., $P = Q \rightsquigarrow \forall\overline{x}P(\overline{x}) \Leftrightarrow Q(\overline{x})$. Specifically, comparisons $<$, $\leq$ are extended pointwise for function symbols and express (strict) subset for predicate symbols.

Graphing Using the following three rules:

$$f(\overline{x}) = y \rightsquigarrow f'(\overline{x}, y) \tag{4.1}$$

$$\exists f :: T_1, \ldots, T_n \to T : \phi \rightsquigarrow \exists f' :: (T_1, \ldots, T_n) : F(f') \wedge \phi \tag{4.2}$$

$$\forall f :: T_1, \ldots, T_n \to T : \phi \rightsquigarrow \forall f' :: (T_1, \ldots, T_n, T) : F(f') \Rightarrow \phi \tag{4.3}$$

where $F(f') \equiv \forall\overline{x} : \exists y : f'(\overline{x}, y) \wedge \forall y' : (f'(\overline{x}, y') \Rightarrow y = y')$, we transform functions f/n of type $T_1, \ldots, T_n \to T$ into predicates $f'/n + 1$ with existence and uniqueness constraints.

Normalization Using the rules $\phi \Rightarrow \psi \rightsquigarrow \neg\phi \vee \psi$, $\phi \Leftrightarrow \psi \rightsquigarrow (\neg\psi \vee \phi) \wedge (\neg\psi \vee \phi)$ and $\phi \Leftrightarrow \psi \rightsquigarrow (\phi \wedge \psi) \vee (\neg\phi \wedge \neg\psi)$, we eliminate $\Rightarrow$,$\Leftrightarrow$. We choose between the two rules for equivalences based on the type of the enclosing connective.

FO Grounding Using the rule $\exists(\forall)x :: T : \phi \rightsquigarrow \bigvee_{t\in\mathcal{D}}(\bigwedge_{t\in\mathcal{D}}) : \phi[x/t]$ with x an FO variable with type T where T has domain $\mathcal{D}$, we instantiate all *first order* quantifications.

First-Order (In)equalities and Arithmetic Following the FO grounding step, all variables have been instantiated. Therefore, all arithmetic functions, (in)equalities and comparisons can be evaluated as usual.

Unique Names We introduce unique names for every remaining (SO) quantification, e.g., $\forall f :: (T) : \phi \wedge \exists f :: (T) : \forall g :: (T) : f(x) \vee \psi \rightsquigarrow \forall f :: (T) : \phi \wedge \exists f' :: (T) : \forall g :: (T) : f'(x) \vee \psi[f/f']$[1].

Prenex Form We pull all SO quantifiers to the front by applying the following two rules. Note that these rules only hold when variable α does not appear in ψ, a condition we have preemptively satisfied due to the **Unique**

[1] We use the notation $\phi[x/x']$ to mean ϕ where all occurrences of x are substituted by x'.

Names transformation and the restriction of sentences to properly typed SO formulas with *no free variables*.

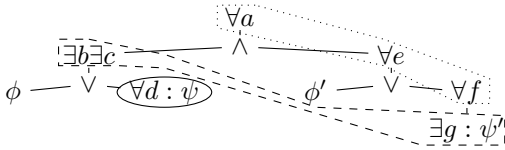
$$(\exists\alpha :: T : \phi) \wedge (\vee)\psi \rightsquigarrow \exists\alpha :: T : \phi \wedge (\vee)\psi \tag{1$\exists$}$$

$$(\forall\alpha :: T : \phi) \wedge (\vee)\psi \rightsquigarrow \forall\alpha :: T : \phi \wedge (\vee)\psi \tag{2$\forall$}$$

To minimize the number of quantifier alternations, we switch between applying rule (1$\exists$) and rule (2$\forall$) only when we cannot further apply the active rule.

Consider the formula $\forall a :: (T) : (\exists b :: (T) : \exists c :: (T) : (\phi \vee \forall d :: (T) : \psi)) \wedge (\forall e :: (T) : \phi' \vee \forall f :: (T) : \exists g :: (T) : \psi')$. This formula becomes $\forall a, e, f :: (T) : \exists b, c, g :: (T) : \forall d :: (T) : (\phi \vee \psi) \wedge (\phi' \vee \psi')$, as visualized below (types dropped for readability).

Note how $\forall e :: (T)$ and $\forall f :: (T)$ are pulled to the level of $\forall a :: (T)$ by rule (2$\forall$), whereas $\forall d :: (T)$ is *blocked* from being pulled to that level by the quantifications $\exists b :: (T)$ and $\exists c :: (T)$, as no rule allows switching the order of $\forall$ and $\exists$ quantifications.



Tseitinize The resulting prenex formula must be flattened to *Prenex Conjunctive Normal Form* (PCNF). We can do this by introducing a so called Tseitin variable for every nested subformula, e.g.: $Q_1 x : Q_2 y : \phi \wedge (\psi \vee (\chi \wedge \rho)) \rightsquigarrow Q_1 x : Q_2 y : \phi \wedge \exists T_i :: \mathit{Bool} : (\psi \vee T_i) \wedge (T_i \Leftrightarrow (\chi \wedge \rho))$. After pulling the existential quantification of the Tseitin variables to the front and rewriting the equivalences using the **Normalization** rule, we obtain PCNF.

Grounding SO We replace every predicate atom $p(\overline{t})$ with a proposition $p_{\overline{t}}$ and quantifications $\forall(\exists)p$ with quantifications $\forall(\exists)p_{\overline{t_0}}, \ldots, \forall(\exists)p_{\overline{t_n}}$ with $\overline{t_i}$ in the domain $\mathcal{D}_p$ of p, e.g., $\forall p : \phi \wedge p(1)$ with the domain $\mathcal{D}_p{=}\{1, 2\}$ would become $\forall p_1 : \forall p_2 : \phi[p(1)/p_1, p(2)/p_2] \wedge p_1$.

Complexity of Grounding

Recalling the result that second-order logic captures the *Polynomial Hierarchy* PH, we know that every second-order logic formula can be placed on a specific level of the hierarchy. We now discuss how the grounding procedure sketched

above influences the number of quantifier alternations in the resulting k-QBF. For any modeling M in second-order logic, we can compute the *maximum* number of *second-order* quantifier alternations $\#\text{alt}(M)$, keeping in mind that all second-order symbols in the vocabulary are quantified existentially. From the grounding procedure sketched above, we can conclude that any modeling M is transformed to a k-QBF where k is $\#\text{alt}(M)$ if $\#\text{alt}(M)$ is even, and $\#\text{alt}(M)+1$ if it is odd. Specifically, the Tseitin transformation from arbitrary second-order formulas to prenex-form can introduce new existentially quantified variables at the innermost level, potentially incrementing the number of alternations by one.

4.4.1 Advanced grounding techniques

In this section we will detail some improvements on the grounding process sketched above.

In the grounding process above, our **FO Grounding** transformation grounds every formula. However, when we know the truth value of certain (sub)formulas such as predicate atoms $p(\overline{x})$, we can replace the (sub)formula by the truth value and propagate this, instead of grounding the (known) formula. This improvement is known as Reduced Grounding or RED [37]. Although this is a simple technique based on *substitution* and *simplification*, it can be powerful. This technique is implemented on top of the grounding process above by SOGrounder.

Grounding With Bounds [143], or GWB, further improves upon RED by maintaining a symbolic representation for every formula in the theory, e.g., using Binary Decision Diagrams or BDDs. These symbolic representations allow for querying the instances which are known to be true (false), known as the CT (CF) bound of the formula. With this knowledge, when grounding for example a universal quantifier, we only need to introduce ground formulas for instances which are not known to be true, i.e., instances not in CT. Although SOGrounder does not support this improvement, it allows users to specify a generating formula for (a set of) quantifications by hand. We call this language construct the *binary quantification*, as it introduces a quantification taking *two* formulas; one formula generating variable instantiations and one being instantiated. E.g., after extending the syntax so that the generating formula takes the place of the type information, $\forall$ `x, y ::[graph(x,y)]:`ϕ instantiates ϕ only for those `x`,`y` for which `graph` holds. Note that the types of `x`,`y` can be derived from the type of `graph`. For strategic companies, this can simplify the nested quantification of `o1`, `o2`, `o3`, and `o4` to a single binary quantification, as shown in **Listing** 4.2 on **Line** 16.

Listing 4.2: SO model for Strategic Companies using *binary quantifications*, for example **Line** 14.

```
 1 type Company.
 2 type Good.
 3 controlledBy   :: (Company, Company, Company, Company, Company).
 4 produces       :: (Company, Good).
 5 strategicset   :: (Company).
 6 non_strat_pair :: (Company, Company).

 8 Company        := {Barilla;Dececco;Callippo;Star}.
 9 Good           := {Pasta;Tonno}.
10 produces       := {Barilla,Pasta; Dececco,Pasta; Callippo,Tonno; Star,Tonno}.
11 controlledBy   := {Star, Star, Star, Star, Barilla; Barilla, Barilla, Barilla, Barilla,
       Dececco}.
12 non_strat_pair := {Barilla,Star}.

14 ∀ good :: Good : ∃ prod :: [produces(prod, good)] : strategicset(prod) ∧ produces(prod,
       good).
15 ∀ controlled :: Company : (∃ o1, o2, o3, o4 :: [controlledBy(o1,o2,o3,o4,controlled)] :
16   strategicset(o1) ∧ strategicset(o2) ∧ strategicset(o3) ∧ strategicset(o4)) ⇒
       strategicset(controlled).
17 ¬(∃ subset :: (Company) :
18   ((subset ≠ strategicset) ∧ ∀ c :: Company : subset(c) ⇒ strategicset(c)) ∧
19   (∀ good :: Good : ∃ prod :: [produces(prod,good)] : subset(prod)) ∧
20   ∀ controlled :: Company : (∃ o1, o2, o3, o4 :: [controlledBy(o1,o2,o3,o4,controlled)] :
21     subset(o1) ∧ subset(o2) ∧ subset(o3) ∧ subset(o4)) ⇒ subset(controlled)).
22 ∀ c :: Company : ∀ c1 :: Company : non_strat_pair(c,c1) ⇒ (strategicset(c) ∧
       strategicset(c1)).
```

One last improvement implemented by SOGrounder is the way Tseitin variables are introduced. The **Tseitinize** transformation described above will introduce every Tseitin literal at the innermost quantification level [66]. However, it is possible to quantify the Tseitin variable T at a lower quantification level, as long as every variable v_i in the Tseitinized formula of T is quantified in the same quantifier block or earlier as T. Recent QBF research suggests that this has a large impact on search efficiency [10]. We also use *polarity optimization* [117] which takes the polarity of the Tseitinized formula into account to reduce the number of clauses introduced.

4.4.2 Grounding to QCIR

The negative impact of the Tseitinization procedure on the performance of QBF solvers has been the target of much study. This has led to the development of alternative transformations [92] and the proposal of new encodings that do not enforce a conjunctive normal form, such as QCIR [84].

The QCIR format [67] instead accepts QBF formulas represented as quantified circuits consisting of *and*, *or*, *xor* and *ite*[2] gates. While the *and* and *or* gates

[2] if-then-else

have an arbitrary number of inputs, the *xor* and *ite* gates are limited to two, respectively three inputs; each gate has a single output.

Following the procedures sketched above up to and including 'Prenex Form' results in an internal representation consisting only of second-order quantifiers over predicates, followed by an arbitrary nesting of conjunctions and disjunctions. By skipping the second-to-last step of Tseitinization and performing grounding on second-order quantifications directly, the resulting prenex non-CNF internal representation can be translated to the QCIR format in a very straightforward way, traversing the internal representation and introducing *and* and *or* gates where applicable. Note that by skipping only the Tseitinization transformation, grounding to QCIR does not use any *xor* or *ite* gates.[3]

4.5 Experiments

To evaluate SOGrounder's performance, we use the model of the *Strategic Companies* problem, modified to benefit from *binary quantification* as discussed above (**Listing** 4.2)[4].

To generate problem instances parametrized by the number of companies, we used the method described in Maratea et al. [104]. The resulting instances are comparable in size and difficulty to the benchmark set present in previous QBF solver competitions.[5] Using our model and SOGrounder, we generated QBF encodings in QDimacs, for which the grounding times and sizes are reported in **Table** 4.1. We also used the instantiation scheme (IS) described by Maratea et al. [104] to generate QBF encodings. We compare the resulting grounding size with that from SOGrounder by reporting the number of literals and clauses.

Regarding solving, we compare the performance of DepQBF [101] on the QBF encodings in **Table** 4.2. Furthermore, using the QCIR output option of SOGrounder, we have also solved the same instances using GhostQ [91], to judge the impact of translating to a more high-level non-CNF QBF representation. Note that as our tool generates QDimacs and QCIR files, we can use many other solvers instead of DepQBF and GhostQ.

As a verification, and motivated to also compare with state-of-the-art solvers in other paradigms, we also report grounding and solving times for the ASP solver Clingo in the respective tables. For Clingo, we use an existing ASP encoding

[3]Nor does it produce non-prenex QCIR, which is supported by the general format but is generally unsupported by solvers.

[4]Binaries and experiments available at https://bit.ly/2rRckXi

[5]http://qbflib.org/suite_detail.php?suiteId=19 on 17/5/2018.

that employs *saturation* [45] to encode the Σ_2^P-hard parts of the problem in ASP.

#Companies	Grounding time (ms)		Lits		Clauses	
	SOGrounder	Clingo	SOGrounder	IS	SOGrounder	IS
14	220	8	282	436	1,054	1,319
29	266	16	582	901	2,179	2,729
38	322	22	762	1,180	2,854	3,575
54	409	38	1,082	1,676	4,054	5,079
77	626	65	1,542	2,389	5,779	7,241
82	684	67	1,642	2,544	6,154	7,711
94	952	83	1,882	2,916	7,054	8,839
100	867	92	2,002	3,102	7,504	9,403

Table 4.1: Overview of grounding times and sizes for strategic companies of size n.

Solver	Solving time (ms)							
	14	29	38	54	77	82	94	100
DepQBF (SOGrounder- QDimacs)	19	38	55	160	4,980	6,725	14,546	16,533
DepQBF (IS - QDimacs)	24	80	161	1,291	20,078	15,388	31,714	42,311
Clingo (ASP)	11	41	85	340	2,522	3,202	7,354	9,351
GhostQ (SOGrounder- QCIR)	120	177	205	252	565	863	2,110	2,355

Table 4.2: Overview of solving times for SOGrounder and IS (QDimacs), Clingo (ASP), and GhostQ (QCIR).

It is clear from the results in **Table** 4.1 that SOGrounder produces smaller groundings than those produced by IS. One contributing factor is the advanced Tseitinization process described above. Other contributing factors are the tools used by IS, and their choice to model the dual problem such that only two quantifier blocks are introduced.

It is this smaller grounding that we identify as the main contributor for SOGrounder's better solving times w.r.t. IS, shown in **Table** 4.2. For Clingo, we expect its optimized *bottom-up* grounding and powerful lookback heuristics to account for its better ground and solving time. Finally, the highly positive impact of using a more high-level QBF representation, e.g., QCIR, is noteworthy in this use case. However, only further experiments will show whether this holds in general.

To illustrate the effects of employing binary quantification, **Table** 4.3 reports grounding times (in ms) for SOGrounder with and without using binary quantifications (5 min. limit). Clearly, without such constructs or techniques

to derive them, grounding can scale very bad. Note that binary quantification does not affect the grounding size in this case.

Binary quantification	Grounding time (ms)						
	5	8	14	17	29	30	33
no	244	385	3329	8608	–	–	–
yes	204	228	220	230	266	270	285

Table 4.3: Overview of SOGrounder's grounding times (ms) with and without binary quantification, 5 min time limit.

4.6 Conclusion and Future Work

We introduced a typed second-order language based on second-order logic, and have shown how it can be used to model problems with second-order constraints, taking the *strategic companies* problem as an example. Furthermore, we present SOGrounder, a tool that accepts modelings in our typed second-order language and grounds them to QBF. It is clear from experiments that this yields a viable option for specifying and solving problems of a higher computational complexity than FOL or even ASP. Its performance can match or even beat existing hand-written QBF and state-of-the-art ASP encodings.

Nevertheless, to ensure performance on the large range of possible problems, of varying complexity, we must expand the benchmark set of **Section** 4.5 and investigate how advanced techniques such as GWB and Lazy Grounding interact with Second-Order and the underlying QBF solvers. We also want to further investigate the effects of Tseitinization in QBF, and build support for QBF encodings which do not require CNF, such as QCIR. Lastly, we want to compare with other, non ground-and-solve systems supporting a language that includes SO, such as ProB [98].

Chapter 5

Semantics of Templates

This chapter is based on the following publication of which I am second author: Dasseville, I., van der Hallen, M., Janssens, G. and Denecker M., "Semantics of templates in a compositional framework for building logics", in Theory and Practice of Logic Programming, 15, 4-5 (2015), pp. 681 – 695 [35]. Specifically, that paper achieves two main contributions:

1. It introduces a framework by which logics can be constructed in a compositional way, and
2. It sketches the need for templates, discusses other approaches and semantically identifies templates as second-order definitions, superseding the earlier rewriting based semantics.

It is this second contribution that is of interest in this work, as **Chapter** 6 will provide an alternative viewpoint on templates. To provide the necessary background, this chapter contains the relevant parts of the paper cited above.

Specifically, the introduction and related work section are taken in large part from this paper; the introduction has been modified to clarify the drawbacks of rewriting-based semantics and to indicate additional benefits arising from the use of templates.

Personal contribution: 33%.

5.1 Introduction

Declarative specification languages have proven to be useful in a variety of applications, however sometimes parts of specifications contain duplicate information. This commonly occurs when different instantiations of the same abstract concept are needed. For example, in an application, we may have to constrain multiple relations such that each of them is an equivalence relation, or we may have multiple relations of which we want to define the transitive closure. In most current logics, the constraints (e.g., reflexivity, symmetry and transitivity) have to be written separately for each relation.

In the early days of programming, imperative programming languages suffered from a similar situation where code duplication was identified as a problem. The first solution proposed involved the use of macros, supported by a system that syntactically rewrites every occurrence of a macro.

For specification languages (e.g., ASP), an analog for macros was introduced and commonly called *templates*. Templates allow us to define a concept and instantiate it multiple times, without affecting the language's computational complexity. Asserting that the two relations `P` and `Q` are equivalence relations can be done using a template `isEqRelation` as in **Listing** 5.1.

Listing 5.1: This example defines an equivalence relation. (Reproduced from [35])

```
1 {
2   isEqRelation(F) ←
3     (∀a : F(a,a)) ∧
4     (∀a,b : F(a,b) ⇔ F(b,a)) ∧
5     (∀a,b,c : (F(a,b) ∧ F(b,c)) ⇒ F(a,c)).
6 }

8 isEqRelation(P) ∧ isEqRelation(Q).
```

Existing treatments of templates mostly proceed in the same vein as imperative programming macros, defining semantics by a source-to-source transformation. This approach comes with some drawbacks:

- It breaks with the traditional approach of declarative, and knowledge representation languages specifically, that language constructs, operators and features are defined as (existing or new) logic concepts, and
- Assigning meaning through a source-to-source transformation often impedes a general treatment of the concept itself: when templates are defined as a source-to-source transformation, studying templates that

are self recursive (**Example** 5.3) or mutually recursive (**Example** 5.4) becomes prohibitively hard.[1]

Apart from simply reducing duplication, the introduction of templates into specification languages offers an additional benefit from which one can already profit even when it is used only once: by grouping a set of constraints in a single template with a descriptive name, one is already achieving a higher level of abstraction in the specification, making the entire specification more readable.

In this chapter, we define the semantics of templates as *second-order inductive definitions*. Subsequently it follows that macro-like source-to-source transformations are simply a *valid* potential implementation when imposing certain restrictions on the templates used.

5.2 Related Work

Abstraction techniques have been an important area of research since the dawn of programming [126]. Popular programming languages such as C++ consider templates as a keystone for abstractions [111]. In the ASP community, work by Ianni et al. [76] and Baral et al. [8] introduced concepts to support composability, called templates and macros respectively. The key idea is to abstract away common constructs through the definition of generic 'template' predicates. These templates can then be resolved using a rewriting algorithm.

More formal attempts at introducing more abstractions in ASP were made. Dao-Tran et al. [34] introduced modules that can be used in similar ways as templates but has the disadvantage that his template system introduces additional computational complexity, so the user has to be very careful when trying to write a specification that systems will support efficiently.

Previously, meta-programming [1] has also been used to introduce abstractions, for example in systems such as HiLog [27]. One of HiLogs most notable features is that it combines a higher-order syntax with a first-order semantics. HiLogs main motivation for this is to introduce a useful degree of higher-order yet remain decidable. While decidability is undeniably an interesting property, the problem of decidability already arises in logic programs under well-founded or stable semantics with the inclusion of inductive definitions: the issue of undecidability is

[1] For another example, consider the Gelfond-Lifschitz reduct [64] from ASP, which is notoriously tricky-to-understand. We believe this is in large part because it is often seen just as a source-to-source transformation process that reduces a logic program by removing some rules in its entirety and dropping any negative atoms from others, as opposed to the result of an operator in Approximation Fixpoint Theory (AFT) [41, 140]

not inherent to the addition of template behavior. Furthermore, in recent times deduction inference has been replaced by various other, more practical inference methods such as model checking, model expansion, or querying. Furthermore, for practical applications, we impose the restriction of stratified templates for which an equivalent first-order semantics exists.

An alternative approach is to see a template instance as a call to another theory, using another solver as an oracle. An implementation of this approach exists in HEX [47]. This implementation however suffers from the fact that the different calls occur in different processes. As a consequence, not enough information is shared which hurts the search. This is analog to the approach presented by Tasharrofi and Ternovska [131], where a general approach to modules is presented. A template would be an instance of a module in this framework, however the associated algebra lacks the possibility to quantify over modules.

5.3 Preliminaries: Rules and definitions

In this section, we will define the interrelated concepts of *rules* and *definitions*. Specifically, we will make few assumptions on the vocabulary V and logic $\mathcal{L}$ for which we introduce rules as long as there is an associated three-valued valuation function. This valuation function maps formulas $\phi \in \mathcal{L}$ and three-valued structures $\mathcal{I}$ over V to the set $\{\mathbf{t},\mathbf{f},\mathbf{u}\}$.

Definition 17 (Rules). *For a given vocabulary V and logic $\mathcal{L}$, a rule is any expression $P(\overline{x}) \leftarrow \phi$, where P is a predicate symbol from V, $\overline{x}$ is a tuple of variables and ϕ is a formula in the logic $\mathcal{L}$ such that the free variables of ϕ are a subset of $\overline{x}$.*

The symbol P is often called the *head* of the rule, while the formula $\phi \in \mathcal{L}$ is referred to as the rule's *body*.

Definition 18 (Definitions). *For a given vocabulary V and logic $\mathcal{L}$, a definition Δ is a set of rules over V for $\mathcal{L}$.*

We divide the symbols occurring in a definition Δ in the set of *defined predicate symbols* $Def(\Delta)$, consisting of all predicate symbols occurring at least once in the *head* of a rule $\in \Delta$, and the set of opens $Open(\Delta)$ containing all other symbols. A *defined domain atom* of Δ is any domain atom $P(\overline{d})$ such that $P \in Def(\Delta)$.

To recover first-order inductive definitions using the definitions above, we restrict the vocabulary V to a set of (typed) first-order symbols, allow only first-order variables in $\overline{x}$ and take as logic $\mathcal{L}$ traditional first-order logic.

5.3.1 Semantics of definitions

We can derive the common two-valued *well-founded* semantics for definitions over V for $\mathcal{L}$ from the three-valued valuation for $\mathcal{L}$.

Definition 19 (t-set, u-set and f-sets). *We call a set of domain atoms a* $\mathbf{t}$*-set, respectively* $\mathbf{u}$*-set,* $\mathbf{f}$*-set of partial interpretation* $\mathcal{I}$ *if its elements have truth value* $\mathbf{t}$*, respectively* $\mathbf{u}$*,* $\mathbf{f}$ *in* $\mathcal{I}$*.*

Definition 20 (Closed partial interpretations). *We call a partial interpretation* $\mathcal{I}$ closed *under* Δ *iff for all domain atoms* $P(\overline{d})$ *and rules* $P(\overline{d}) \leftarrow \phi \in \Delta$ *it holds that* $\phi^{\mathcal{I}[\overline{x}:\overline{d}]} = \boldsymbol{t}$ *implies* $P(\overline{d}) = \boldsymbol{t}$*.*

Definition 21 (Unfounded sets). *An unfounded set* U *of* Δ *in* $\mathcal{I}$ *is a* non-empty $\mathbf{u}$*-set of defined domain atoms s.t. for all* $P(\overline{d}) \in U$ *and every rule* $r \in \Delta$ *with* P *as its* head *and* ϕ *as its body,* $\phi^{\mathcal{I}[\overline{x}:\overline{d},U:\mathbf{f}]} = \boldsymbol{f}$*.*

Informally, an unfounded set is any set of defined domain atoms such that if one would assign them false instead, all their associated (ground) rule bodies would evaluate to false as well.

Definition 22 (Partial stable definitions). *A partial interpretation* $\mathcal{I}$ *is a* partial stable interpretation *of* Δ *iff*

- *for each domain atom* $P(\overline{d})$ *where* $P \in Def(\Delta)$*,* $P(\overline{d})^{\mathcal{I}} = Max_{\leq_t}\{\phi^{\mathcal{I}[\overline{d}:\overline{x}]} \mid \forall \overline{x} \ . \ P(\overline{x}) \leftarrow \phi \in \Delta\}$*;*
- *there exists no non-empty* $\mathbf{t}$*-set* T *and no (possibly empty)* $\mathbf{u}$*-set* U *of* $\mathcal{I}$ *s.t.* $\mathcal{I}[T : \mathbf{u}][U : \mathbf{t}]$ *is closed under* Δ*; and*
- *there are no unfounded sets of* Δ *in* $\mathcal{I}$*.*

Definition 23 (Well-founded Interpretations). *We call an interpretation* $\mathcal{I}$ *a well-founded interpretation of* Δ *if*

1. $\mathcal{I}$ *is the* $\leq_p$*-least partial stable model* $\mathcal{I}'$ *of* Δ *such that* $\mathcal{I}'|_{Open(\Delta)} = \mathcal{I}|_{Open(\Delta)}$*, and*
2. $\mathcal{I}$ *is exact.*

Paradox-free definitions According to Denecker and Vennekens [43], sensible definitions formalize the (informal) inductive definitions found in mathematical texts, and are *paradox-free.*

Definition 24 (Paradox-free definition). *A definition* Δ *is* paradox-free *if for any* exact *interpretation* I *for* $Open(\Delta)$*, there is a (unique) well-founded interpretation of* Δ *extending* I*.*

Note that, by **Definition** 23, this is equivalent to "For any *exact* interpretation I for $Open(\Delta)$, the $\leq_p$-least partial stable model $\mathcal{I}$ s.t. $\mathcal{I}$ extends I, is exact.".

Nested definitions Note that the framework above, discussed in detail in [35], in fact allows for *nested* definitions, by taking as logic $\mathcal{L}$ a logic already admitting definitions, as long as we can provide a three-valued valuation for $\mathcal{L}$.

Specifically for definitions, this can be challenging. Recently, Charalambidis et al. [26] suggested extensions to Approximation Fixpoint Theory (AFT) to allow for a three-valued extensional semantics for sets of rules that generalizes the well-founded semantics. Alternatively, we can define such a three-valued valuation for $\mathcal{I}$ for definitions based on the two-valued well-founded semantics using *Ultimate approximation*, i.e., by evaluating under the two-valued well-founded semantics *all* two-valued extensions I of $\mathcal{I}$. The three-valued valuation is then **t** (respectively **f**) iff all two-valued extensions evaluate to **t** (respectively **f**), and **u** otherwise. For definitions Δ, we say the two-valued well-founded semantics evaluates I to **t** if I is the well-founded interpretation of Δ and evaluates to **f** if it is not.

5.4 Templates and Template Libraries

To achieve the stated goal of reducing duplication in specifications, we introduced [35] not just *templates*, but defined them within the context of a *library of templates*.

To construct a library of templates TL, we first define a single common *vocabulary* V_{TL} for the entire template library.

Definition 25 (Template Library Vocabulary). *A template library vocabulary V_{TL} is a vocabulary that consists of all pre-interpreted symbols (addition, equivalence, etc.) and a set of second-order symbols, called the* template symbols.

Definition 26 (Template). *A template over V_{TL} is a* paradox-free *second-order definition Δ over V_{TL} s.t. its defined symbols $Def(\Delta)$ are a subset of the* template symbols *of V_{TL}.*

Definition 27 (Template Library). *A template library TL is a set T of templates over a given* template library vocabulary *V_{TL} such that:*

1. *for every template symbol $TS \in V_{TL}$ there is* exactly one *template $\Delta \in T$ that defines TS (i.e., $TS \in Def(\Delta)$), and*

2. *The template library defines a strict ordering $<_{TL}$ on template symbols TS s.t. for every $\Delta \in TL$, if template symbol $P \in Def(\Delta)$, and template symbol $Q \in Open(\Delta)$ then $Q < P$.*

Note that by restricting the vocabulary of templates to the vocabulary V_{TL}, template symbols can only depend on:

- pre-interpreted symbols such as $+$, $-$, $=$, $\ldots$,
- other template symbols, and

However, any symbol $T \notin V_{TL}$ on which a template should depend, can simply be passed to the template as an additional argument.

The introduction already contained an example template in **Listing** 5.1. The meaning of this template is now clear: it is a second-order predicate that is true for any binary predicate F that is reflexive, symmetric and satisfies transitivity. We give two additional examples of templates, using more of the full power of templates.

The template `tc` in **Listing** 5.2 expresses for two binary predicates `P` and `Q` of type `(T, T)` that `Q` is the *transitive closure* of `P`.

Listing 5.2: The template `tc` expresses that Q is the transitive closure of P.

```
1 {
2   tc(P,Q) ←
3       { Q(x,y) ← P(x,y) ∨ (∃ z :: T : Q(x,z) ∧ Q(z,y)). }.
4 }
```

The template `adj` (**Listing** 5.3) uses both a nested definitions and self-recursion to express for two binary predicates `P` and `Q` of type `(T,T)` that `Q` is the n-step adjacency relation of `P`, e.g., `Q(x,y)` iff one can travel *exactly* n edges in `P` starting at `x` to arrive at `y`.

Listing 5.3: The template `adj` expresses reachability in exactly n steps.

```
1 {
2   adj(P, P, 1).
3   adj(P, Q, n) ← {Q(x,y) ← ∃Q' :: (T,T) : adj(P,Q',n-1) ∧ ∃z :: T : Q'(x,z) ∧ P(z,y)}.
4 }
```

5.4.1 The Complexity of Templates

In **Definition** 18, templates are defined as a *paradox-free second-order* definition Δ. Just as for first-order definitions, paradox-free second-order definitions are at

least as powerful as extending second-order logic with a *least fixed point* operator. It is a well-known result by Immerman [80] (p. 167) that second-order logic extended with a least fixed point operator[2] *captures* the EXPTIME complexity class.

Recall that generalized games that may last for a number of moves exponential in the size of the board are often EXPTIME-complete [130, 40], e.g., $n \times n$ chess [55] or go [121].

We show that such games can be expressed using templates, as shown in **Listing** 5.4; here, chessboards are represented as functions `Board`, `Board'` from pieces to positions. A `Board` is winning if there is a `Board'` s.t. it is a valid successor in chess and it is losing. A `Board` is losing if all its possible successors are winning; it follows that `lose` holds for all values for `Board` for which no successor exists (i.e., because the player is checkmate).

Note that $\texttt{move}_{chess}$ is a second-order template itself (not shown), defined to be true for boards B, B' if a valid chessmove leads from board B to board B'.

Listing 5.4: A template describing winning positions in chess.

```
1 {
2   win(Board)  ← ∃Board' :: Piece → Position : move_chess(Board, Board') ∧ lose(Board').
3   lose(Board) ← ∀Board' :: Piece → Position : move_chess(Board, Board') ⇒ win(Board').
4 }
```

5.4.2 Template libraries for Existential Second-Order Logic

We have seen in the previous section that general templates are very expressive. However, when we integrate templates into a specific language, e.g., the FO($\cdot$) language of the IDP system, we are interested in expanding the practical expressive power, rather than increasing the theoretical complexity.

Considering the IDP system specifically, it is possible to limit template libraries and their templates such that we obtain an expressive power of existential second-order logic. Model expansion over such templates is equally powerful as model expansion over first-order logic with (possibly nested) first order definitions (FO(ID^*)). Unsurprisingly, any occurrence of such a template can be *translated away* to an FO(ID^*) formula.

To achieve this, we identify the languages FO(ID^*) ϕ, ESO(ID^*) ϵ and ASO(ID^*) α defining, through mutual recursion, fragments of second-order logic with nested inductive definitions (SO(ID^*)) in **Figure** 5.1, where $\overline{x}$ are

[2] At least, one that accepts second-order variables as an argument.

$$\phi ::= s(\overline{x}) \mid \neg\phi \mid \phi \wedge \phi \mid \exists x : \phi \mid \{s(\overline{x}) \leftarrow \phi\} \mid \text{let } \{s(\overline{x}) \leftarrow \phi\} \text{ in } \phi$$

$$\epsilon ::= s(\overline{x}) \mid S(\overline{X}) \mid \neg\alpha \mid \epsilon \wedge \epsilon \mid \exists \overline{x} : \epsilon \mid \{s(\overline{x}) \leftarrow \phi\} \mid \exists X : \epsilon \mid \text{let } \{s(\overline{x}) \leftarrow \phi\} \text{ in } \epsilon$$

$$\alpha ::= s(\overline{x}) \mid S(\overline{X}) \mid \neg\epsilon \mid \alpha\wedge\alpha \mid \exists \overline{x} : \alpha \mid \{s(\overline{x}) \leftarrow \phi\} \mid \forall X : \alpha \mid \text{let } \{s(\overline{x}) \leftarrow \phi\} \text{ in } \alpha$$

Figure 5.1: The FO(ID^*), ESO(ID^*) and ASO(ID^*) fragments of SO(ID^*).

first-order variables, $\overline{X}$ are second-order variables while s are first-order symbols and S are second-order (template) symbols.

Specifically, all three languages feature (nested) first-order definitions and a let-construct which locally introduces predicate(s) for a subformula FO(ID^*), with the restriction that the quantified symbols must be defined by an accompanying (paradox-free) definition Δ. Effectively, this corresponds to a second-order quantification. However, as a paradox-free definition Δ has a unique well-founded interpretation given an exact interpretation of its opens, we are free to choose whether that quantification is *universal* or *existential*.

Beyond the two extensions above, FO(ID^*) corresponds to traditional first-order logic with no explicit second-order quantifications, while ESO(ID^*) and ASO(ID^*) allow strictly *existential* or *universal* quantification, respectively, taking into account occurrences under negation(s).

By limiting the template library to *non-recursive* definitions consisting of a *single rule* with a body $\in$ FO(ID^*) extended with template applications, we achieve the goal of gaining expressive power without increasing expressive power.

Proposition 1. *Any formula ϕ in* ESO(ID^*) *over vocabulary $V \cup V_{TL}$ s.t. V and V_{TL} are disjoint and ϕ does not contain definitions over template symbols $TS \in V_{TL}$, can be translated to a polynomially larger* FO(ID^*) *formula ϕ' over a vocabulary V' extending V s.t. it is V-equivalent to the* SO(ID^*) *formula $\phi \wedge TL$.*

We refer to Dasseville et al. [35] for a complete proof. Informally, because we restrict ourselves to *non-recursive* definitions, the well-founded semantics is equivalent to Clark's completion semantics [29], and completion semantics admit substitution of template symbol applications by its definition's body. Subsequently, because template libraries must be stratified, recursive substitution is guaranteed to terminate.

5.5 Conclusion

In this preliminary background, we showed how to construct *second-order inductive definitions*, whose potential to *abstract* over relations and whose *expressivity* allow us to easily characterize many powerful *templates*.

In general, however, when working from the context of extending a specific system such as IDP with support for templates the goal is to increase *practical* expressivity, and not computational complexity. Therefore, we show that further syntactical restrictions on templates can limit the expressivity of definitions such that a) many useful templates are still covered, and b) given a library of such templates, formula's applying these templates can be rewritten to first-order logic with (nested) inductive definitions.

Chapter 6

A Second-Order Pattern: Integrating inferences in logic

This chapter consists of new, unpublished work.

Personal contribution: 100%.

6.1 Introduction

In the previous chapters, we have shown how our typed second-order language can be grounded to QBF and have discussed the concept of 'templates' and their semantics. In this chapter, we identify an often reoccurring pattern in second-order logic specifications, and discuss how the reasoning that it represents can be supported by introducing the concepts of *parametrized theories* and *variant world quantifiers*, providing an alternative point-of-view on *templates*.

Essentially, the pattern corresponds to this: while the knowledge base paradigm traditionally offers a declarative modeling of knowledge and a procedural or imperative language for performing inferences on this knowledge, some knowledge concepts themselves correspond to the result of an inference over a smaller, simpler knowledge base. Parametrized theories and variant world quantifiers allow expression of these inferences in logic itself in a structured and declarative way.

First, we recall that an interpretation of a set of (first and second-order) symbols can be seen as a *possible world*. The pattern we then discern, visualized in

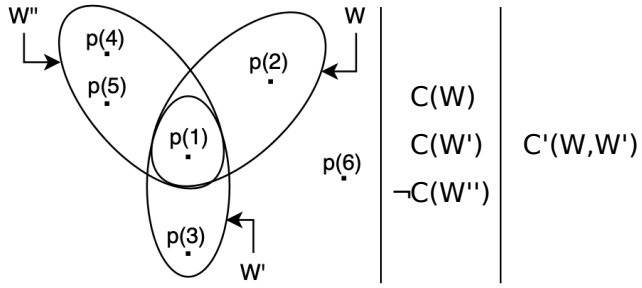


Figure 6.1: Visual representation of the second-order pattern, with the world W, its variants W' and W'', the conditions C and supplementary condition C' between W and variant worlds satisfying C.

Figure 6.1, expresses the following: (1) that the possible world W should satisfy a set of conditions C, and furthermore, (2) that either *some* or *all variant* worlds W' of the possible world W that satisfy the same conditions C, satisfy a supplementary condition C', such as being related in a specific way to W.

The second part of the pattern can be considered as a form of *'binary quantification'* over variant worlds: a condition C' is asserted for *some* or *all* members of a certain set. Thus, expressing this condition requires to express the relevant class of variant worlds W' of W. This class of variant worlds consists of those worlds W' that:

1. satisfy the specific conditions of C, just as W, and
2. coincide with W on a specific set of terms and atoms, called the 'common core'.

The definitions of language constructs supporting the pattern above are given in **Section** 6.2. However, we to allow a more detailed and comprehensive discussion of examples illustrating our envisioned pattern, we already introduce the language constructs informally as follows:

- *Parametrized theories* $\mathcal{T}$ and their application $*\mathcal{T}(S_1, \ldots, S_n)$ where S_1, $\ldots$, S_n are arguments of the theory $\mathcal{T}$.
- Two variant world quantifiers $\Box$ (*all*) and $\Diamond$ (*some*) that allow reinterpretation of (specific subsets of) symbols by a *parametrized theory*. For example, $\Box\mathcal{T}(S : \{x \mid \phi\})$ quantifies over *all* reinterpretations of S by $\mathcal{T}$ that do *not* change the value of S for any x for which ϕ holds.

- An unshadowing operator $^{\uparrow}$ that allows referral to symbols before reinterpretation.

6.1.1 Examples

To illustrate the pattern, this section explores some examples. These examples show how we can think about certain problems as wanting to find a *specific* model of a (simpler) theory, e.g., the minimal model, and how we consequently can express these problems using parametrized theories and variant world quantifiers.

While in some of the examples, the same properties can be expressed in specific languages or using inferences built into solvers, allowing expression of these properties and inferences in the language itself leads to a more general approach.

At the end of each example, to fully illustrate the problem and its modeling, we already show the specification of the problem using the language constructs that were introduced informally above. We encourage the reader to return to these specifications after the language constructs have been formally introduced in **Section** 6.2.

Chromatic Coloring Problem

As a first example, consider the well-known *Graph Coloring* problem and its variant *Chromatic Coloring*. To solve the *Graph Coloring* problem, we must find a coloring for a given graph $\mathcal{G}$.

Definition 28 (Coloring). *Given a graph $\mathcal{G}$ with vertices V and a set of colors C, a* coloring *is an assignment (function) f of vertices to colors such that for every two vertices v_1, v_2 that share an edge $(v_1, v_2) \in \mathcal{G}$, $f(v_1) \neq f(v_2)$.*

In the *Chromatic Coloring* problem, we must find an *optimal* coloring, i.e., one that uses the smalles number of colors possible.

Definition 29 (Optimal Coloring). *A function f from vertices V to colors C is an optimal coloring iff 1) f is a valid* coloring *and 2) for every other valid coloring f', the number of colors used by f is smaller than or equal to the number of colors used by f'.*

We now view the chromatic coloring problem as an instance of the second-order pattern; note that it corresponds to a minimal model of a (simpler) theory $\mathcal{T}$, expressing what it means to be coloring.

Listing 6.1: A theory representing the coloring constraints

```
theory T_color{
  type N.
  type C.

  G :: (N,N).
  f :: N → C.
  U :: (C).

  ∀ x, y :: N : G(x,y) ⇒ f(x) ≠ f(y).
  ∀ c :: C : U(c) ⇔ (∃ x :: N : f(x) = c).
}
```

First, we identify the world W from our pattern: for the chromatic coloring problem, this world W consists of the graph $\mathcal{G}$, a function f that represents a mapping from nodes to colors, and a predicate U that represents used colors.

Next, we specify the conditions C that a world W must satisfy: In this instance, the conditions C are simply those of a coloring with colors from U, i.e., two nodes sharing an edge are not assigned the same color c, and the set U is exactly the set of colors assigned to a node. These conditions are captured by the theory $\mathcal{T}_{color}$ of **Listing** 6.1.

In the chromatic coloring problem, the variant worlds W' to be considered are simply all worlds that:

1. satisfy the theory $\mathcal{T}_{color}$, and
2. coincide with W on the interpretation of $\mathcal{G}$.

For the world W and its variant worlds W', we want the following condition C' to hold: For every variant world W', it's interpretation of the set of used colors U is *not a subset* of the interpretation of U by W.

```
1 *T_color(G,f,U) ∧ □T_color(G:{(n1,n2) | True}, f:{}, U:{}) : ¬(U < U↑).
```

Temporal Logics

*Temporal logic*s are logic systems designed to represent and reason about properties in the context of time; for example, the property '*whenever a process requests a resource, it will eventually be granted access to this resource*'. The rules of processing requests and allocating resources can subsequently be described as a (labeled) transition system. We can then express the aforementioned property in a temporal logic such as Linear Temporal Logic [118] (LTL) as G (req => F

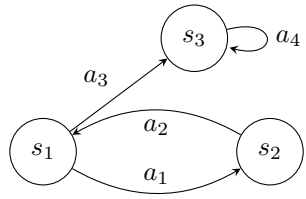


Figure 6.2: Example transition system for temporal logics.

alloc), or in Computation Tree Logic [30] (CTL) as AG (req ⇒ AF alloc). We give an example on how to model these same properties using our pattern.

Consider the transition system from **Figure** 6.2. We now want to express that 'there is a sequence of actions such that at each time t it is *possible* to reach the state s_3 in the future'. It is not necessary that we actually ever pass the state s_3. This property corresponds to the CTL expression AG EF s_3.

By making time explicit, we will model this property using our pattern.

Note that this corresponds at every time point to the computation of a model of a (simpler) theory T which expresses that a sequence of actions exists such that we reach state s_3.

First, we identify the world W: for our example, this consists of a sequence of actions represented by a function `act` from `Time` to `Action` and a state function `state` from `Time` to `State`.

Next, we specify the conditions C that a world W must satisfy: In general, these conditions enforce the preconditions and the effects of actions a at every time t. For our example, these conditions are modelled by the theory $\mathcal{T}_{actions}$ from **Listing** 6.2.

For this example, the variant worlds W' to be considered are all worlds that:

1. satisfy the theory $\mathcal{T}_{actions}$, and
2. coincide with W on the interpretation of `act` and `state` up until a given timepoint t'.

For every time point t', we want the following condition C' to hold for world W and its variant worlds W' for timepoint t': there is a variant world W' such that at some point $t_f > t'$, `state`(t_f)=$s3$ holds.

```
1 ∀ t' :: Time : ∗T_actions(act, state) ∧ ◇T_actions(act:{t | t < t'}, state:{t | t < t'}) : ∃ t
      :: Time : t > t' ∧ state(t)=s3.
```

Listing 6.2: A theory representing the transition system of **Figure** 6.2.

```
1 theory T_actions{
2   type Action.
3   type State.
4   type Time.

6   [...]

8   act   :: Time → Action.
9   state :: Time → State.
10  transitions :: (State, Action, State).

12  Action := {a1;a2;a3;a4}.
13  State  := {s1;s2;s3}.
14  transitions := {(s1,a1,s2); (s1,a3,s3); (s2,a2,s1); (s3,a4,s3)}.

16  ∀t :: Time : transitions(state(t),act(t),state(t+1)).
17 }
```

Actual Causation

Causality is an often studied subject in science, and 'actual causality' is the study of causality not in general (e.g. "smoking causes lung cancer"), but "*focus[ing] on particular events* [such as] *the fact that David smoked like a chimney for 30 years caused him to get cancer last year*" [70]. One popular way of defining when a certain fact is an actual cause of an event, is the use of counterfactual reasoning.

Such counterfactual reasoning can be modeled using this second-order pattern. Specifically, we consider the *structural equations*, as introduced by Halpern and Pearl [71]. Structural equations describe the mechanisms by which variables (causually) influence each other. For this purpose, the variables in structural equations are often split into the *exogenous* and *endogenous* variables. In short, the endogenous variables are described by the structural equations, while the exogenous variables are factors determined by the outside world (and consequently are never set by the equations).

To illustrate how the SO pattern can be applied here, we consider the following well-known example:

> "*Suzy and Billy both pick up rocks and throw them at a bottle. Suzy's rock gets there first, shattering the bottle. Since both throws are perfectly accurate, Billy's would have shattered the bottle had it not been preempted by Suzy's throw.*" (Paul and Hall [69])

We now use the SO Pattern to conclude that, by Halpern's modification [70] of

Listing 6.3: A theory representing the structural equations of the rock-throwing example.

```
theory T_causal{
    type Variable.

    trues   :: (Variable).
    shatter :: Bool.

    Variable := {suzy_throws;suzy_hits;billy_throws;billy_hits}.

    trues(suzy_throws) ⇔ trues(suzy_hits).
    (trues(billy_throws) ∧ ¬trues(suzy_hits)) ⇔ trues(billy_hits).
    (trues(suzy_hits) ∨ trues(billy_hits)) ⇔ shatter.
}
```

the Halpern and Pearl definition of actual causation, Suzy throwing her rock is an actual cause of the bottle shattering, while Bill's throw is not.

First, we will take at how to model the structural equations. Here, we choose to model the variables from the structural equations as Herbrand elements, and let their evaluation be modeled by a predicate `trues`. **Listing** 6.3 shows the specification for the structural equations of the rock-throwing example, where the 'hits' variables are introduced to model the preemption of Suzy's throw with respect to Billy's.

To apply the pattern, the world W consists of the predicate `trues`, which represents the evaluation of the rock-throwing problem's variables, and the proposition `shatters` indicating whether the bottle shatters.

Next, the conditions C that a world W must satisfy are exactly those given by the specification of the structural equations of **Listing** 6.3.

Lastly, for actual causation, we must consider variant worlds W' that satisfy the theory $\mathcal{T}_{causal}$, but, the 'common core' between W and W' can vary. Specifically, an *actual cause* is a subset-minimal set AC of variables such that reassigning the variables in AC while leaving all other variables the same, means the bottle **doesn't shatter**.

This corresponds to the following expression, which uses variant world quantifications twice: once to express that changing the variables in AC means the bottle doesn't shatter, and once to express that this set AC is minimal.

```
∃AC :: (Variable) : *T_causal(trues, shatter) ∧ shatter ∧
    (◇T_causal(trues:{v | ¬AC(v)},shatter:{}): ¬shatter) ∧
    ∀AC' :: (Variable) : (AC' < AC ⇒ □T_causal(trues:{v | ¬AC'(v)},shatter:{}) : shatter).
```

Note that in actual causation, we find an example where the 'common core' is

Listing 6.4: Example theory for brave (and cautious) reasoning.

```
1 theory T_birds {
2     type Animal.

4     bird :: (Animal).
5     pinguin :: (Animal).
6     fly :: (Animal).

8     ∀ animal :: Animal : bird(animal) ∧ ¬pinguin(animal) ⇒ fly(animal).
9 }
```

not known beforehand. Instead, the common core is in fact searched, as they relate to actual causes.

Brave Reasoning

Brave (and cautious) reasoning are two well-known forms of default reasoning [50]. In situations where different assumptions can be used to derive conclusions, that are potentially mutually inconsistent, brave and cautious reasoning represent two modes of reasoning: in the case of brave reasoning, anything for which a set of assumptions exists can be derived, while cautious reasoning only allows derivation of those conclusions that can be derived regardless of the set of assumptions.

To illustrate brave reasoning, we need a model where one or more assumptions can be made. Consider, as a toy example, the following situation: "All bird can fly, except for pinguins. Tweety is a bird. Can Tweety fly?". We now look at how we can solve the question "Can Tweety fly" under brave reasoning (in which case the answer is yes), using our SO pattern.

First, we identify the world W. It simply consists of a single predicate `fly`.

Next, we specify the conditions C that a world W must satisfy. This is a straightforward modeling of the sentences in our example, as can be found in **Listing** 6.4.

The variant worlds W' to be considered here are simply all worlds that satisfy the theory $\mathcal{T}_{birds}$, i.e., in this example, the common core is empty.

We say `fly(tweety)` is a brave consequence if in at least one of the variant worlds W', `fly(tweety)` is true. This corresponds to the following expression:

```
1 consequence ⇔ ◊T_birds(bird:{tweety}, pinguin:{}, fly:{}) : fly(tweety).
```

6.2 Theoretical foundation

We look at how the theoretical foundations of our typed second-order language can be extended with parametrized theories and their applications, and with variant worlds quantifications. First, we will recapitulate the typed second-order language without these extensions.

Consider a simple type system consisting of basic types T, and their combinations $(T_1, \ldots, T_n)$ and $T_1, \ldots, T_n \to T$, representing n-ary predicate and n-ary function types respectively. Remark that $T_1, \ldots, T_n$ represent only basic types; higher-order predicates or functions are not considered.

Definition 30 (Vocabulary). *A vocabulary V is a tuple consisting of 1) a set Θ of basic types T, each with a natural ordering $\leq_T$, 2) a set of typed predicate symbols P/n, and 3) a set of typed function symbols f/n. The types of these predicate and function symbols combine only T from Θ.*

Definition 31 (Terms). *A term t over a vocabulary V is either*

- *a* variable*,*
- *a* predicate *symbol P or* function *symbol f from V, or*
- *a function application $f(t_1, \ldots, t_n)$, where f is either a symbol from V or a variable, representing an n-ary function, and $t_1, \ldots, t_n$ are terms over V.*

Note that in second-order logic, predicate and function symbols can occur as terms, e.g., in the equality constraint $P_1 = P_2$, P_1 and P_2 are terms.

To type arbitrary terms t, we introduce a typing context Γ which is a sequence of symbols and variables with their types; we call the combination of a symbol or variable with its type an *assumption*. The element relation $\in$ expresses that the typing context Γ contains a symbol or variable with a certain type, e.g., $x :: T \in \Gamma$. The empty typing context can be written as $\emptyset$, and typing contexts can be extended using a comma, e.g., $\Gamma, x :: T$. With every vocabulary V is associated a typing context Γ_V, i.e., the sequence of all typed predicate symbols P/n and typed function symbols f/n declared in V.

Using the *typing context* Γ, we can define the typing relation on terms t: $\Gamma \vdash t :: T$ expresses that the term t is *well-typed* and has type T when given the assumptions in Γ.

Definition 32 (Well-Typed Terms). *A term t is* well-typed *under the assumptions of Γ with type T, i.e., $\Gamma \vdash t :: T$ iff*

- *t is a variable v s.t. $v :: T \in \Gamma$,*

- *t is a (*predicate *or* function*) symbol s s.t. $s :: T \in \Gamma$, or*
- *t is a function term $f(t_1, \ldots, t_n)$ with f a* function *variable or symbol s.t. $\Gamma \vdash f :: T_1, \ldots, T_n \to T$ and $\Gamma \vdash t_1 :: T_1, \ldots, \Gamma \vdash t_n :: T_n$.*

Definition 33 (Second-Order Formula). *Second-order formulas over a vocabulary V are inductively defined as follows:*

- *$P(t_1, \ldots, t_n)$ is a formula if $t_1, \ldots, t_n$ are terms over vocabulary V and P is either 1) an n-ary predicate symbol from V or 2) an n-ary predicate variable;*
- *$\neg\phi$, $\phi \wedge \psi$, and $\phi \vee \psi$ are formulas if ϕ and ψ are formulas;*
- *$\exists x :: T : \phi$ and $\forall x :: T : \phi$ are formulas if ϕ is a formula, and x is a first-order or second-order (predicate or function) variable of type T; and*
- *$t_1 = t_2$, $t_1 < t_2$ are formulas if t_1 and t_2 are terms over V.*

The shorthands $\Leftarrow$, $\Rightarrow$ and $\Leftrightarrow$ are defined as usual, as are $>$, $\leq$, and $\geq$.

We extend well-typedness to formulas ϕ, writing $\Gamma \vdash \phi$ to mean that ϕ is well-typed under the assumptions of Γ.

Definition 34 (Well-Typed Formulas). *A formula ϕ is* well-typed *under the assumptions of Γ, iff ϕ is either:*

- *a predicate application $P(t, \ldots, t_n)$ where $P :: (T_1, \ldots, T_n) \in \Gamma$ and $\Gamma \vdash t_1 :: T_1, \ldots, \Gamma \vdash t_n :: T_n$ respectively,*
- *$\neg\psi$, $\psi \wedge \psi'$, $\psi \vee \psi'$ where $\Gamma \vdash \psi$ and $\Gamma \vdash \psi'$,*
- *$\exists x :: T : \psi$, $\forall x :: T : \psi$ where $\Gamma, x :: T \vdash \psi$, or*
- *$t_1 = t_2$ or $t_1 < t_2$, where there is a type T s.t. $\Gamma \vdash t_1 :: T$ and $\Gamma \vdash t_2 :: T$.*

Parametrized Theories

We can now introduce parametrized theories as follows.

Definition 35 (Parametrized Theories). *A* parametrized theory *$\mathcal{T}$ consists of a* vocabulary *$V_{\mathcal{T}}$ and a second-order formula $\phi_{\mathcal{T}}$ over $V_{\mathcal{T}}$, called the* body *and (optionally) an interpretation for some of the types T in $V_{\mathcal{T}}$.*

Example 2. *Consider the example parametrized theory $\mathcal{T}_{color}$ of **Listing** 6.1. Its vocabulary $V_{\mathcal{T}_{color}}$ (**Lines** 2–7) defines two types, `N` and `C`. It also declares three symbols: a binary predicate `G` of type (`N`, `N`), a function `f` of type `N` → `C`, and a set `U` of type (`N`).*

*The formula $\phi_{\mathcal{T}}$ is given by the conjunction of the **Lines** 9–10.*

The symbols declared by vocabulary $V_{\mathcal{T}}$ are considered the arguments of the parametrized theory. The vocabulary $V_{\mathcal{T}}$ imposes a fixed order for the arguments.

Any auxiliary symbols that should not be arguments can be introduced explicitly by quantifying them in the second-order formula $\phi_{\mathcal{T}}$, and can be assigned a specific value using an equality constraint.

Definition 36 (Signatures). *The* signature $\sigma(\mathcal{T})$ *of parametrized theory* $\mathcal{T}$ *is defined as the tuple containing the type of every symbol defined in the vocabulary* $V_{\mathcal{T}}$*, in order of occurrence.*

The signature $\sigma(\mathcal{T}_{color})$ for parametrized theory $\mathcal{T}_{color}$ defined in **Listing** 6.1 is $\langle(\mathtt{N}, \mathtt{N}), \mathtt{N} \to \mathtt{C}, (\mathtt{C})\rangle$.

Definition 37 (Well-Typed Theories). *A parametrized theory* $\mathcal{T}$ *is* well-typed *if:*

- *The formula* $\phi_{\mathcal{T}}$ *is well-typed under the assumptions of* $\Gamma_{V_{\mathcal{T}}}$*, i.e.,* $\Gamma_{V_{\mathcal{T}}} \vdash \phi_{\mathcal{T}}$*, and*
- *the types* T *used in the signature* $\sigma(\mathcal{T})$ *of* $\mathcal{T}$ *are* not *interpreted by* $\mathcal{T}$*. All other types* T *declared in* $V_{\mathcal{T}}$ must *be interpreted by* $\mathcal{T}$*.*

The second requirement enforces a level of genericness on parametrized theories $\mathcal{T}$. On the one hand, types not present in the signature $\sigma(\mathcal{T})$ are interpreted by $\mathcal{T}$, but no argument symbol takes arguments of such a type. We sketch a use case for such a type: consider a theory $\mathcal{T}$ that declares a type *Node*, declares a type *Color* and interprets it as $\{R, G, B\}$, and declares a predicate *edge* of type $(Node, Node)$. The body of $\mathcal{T}$, $\phi_{\mathcal{T}}$, can quantify over functions *labeling* of type $Node \to Color$ to check that the graph represented by *edge* can be colored using only three colors. As *Color* is interpreted by $\mathcal{T}$ we *know* that only three colors exist, and that they are R, G and B.

On the other hand, types present in the signature $\sigma(\mathcal{T})$ cannot be interpreted by $\mathcal{T}$, thus $\mathcal{T}$ cannot make any assumptions about how many elements or which elements are in type T. When a specific element of type T is needed, the theory $\mathcal{T}$ can declare a constant of type T such that this specific element can be passed as an argument. Likewise, the set of elements of type T can be passed as a unary predicate.

We only impose this restriction to simplify the theoretical discussion; the implementation (**Section** 6.3) instead checks that when a parametrized theory $\mathcal{T}$ is applied, types are *consistent*: if $\mathcal{T}$ interprets a type T that occurs in $\sigma(\mathcal{T})$, then T has the same interpretation by $\mathcal{T}$ as by its caller. We can justify this more permissive condition by noting that this corresponds to adding the

Listing 6.5: An example of a parametrized theory application

```
type Node.
type Col.

Node := {n1; n2; n3; n4}.
Col := {Red; Blue}.

∃G :: (Node, Node) : ∃f :: Node→Col : ∃U :: (Col) : *T_color(G,f,U).
```

necessary additional arguments (a unary predicate and a constant for every element in the interpretation of T) and modifying $\phi_{\mathcal{T}}$ to enforce the consistency. Such an encoding would, however, be very cumbersome and counterproductive.

A parametrized theory $\mathcal{T}$ can be applied to variables and symbols. We extend **Definition** 33 (Formulas) and **Definition** 34 (well-typed formulas) in the following definitions:

Definition 38 (Second-Order Formulas ext.). *$*\mathcal{T}(S_1, \ldots, S_n)$ is a formula if $\mathcal{T}$ is a parametrized theory whose vocabulary $V_{\mathcal{T}}$ contains n symbols.*

We consider all parametrized theories that are defined to be available globally.

Definition 39 (Well-Typed Formulas ext.). *The application $*\mathcal{T}(S_1, \ldots, S_n)$ of a parametrized theory $\mathcal{T}$ is* well-typed *iff $\sigma(\mathcal{T}) = \langle T_1, \ldots, T_n \rangle$, $\Gamma \vdash S_1 :: T_1', \ldots, \Gamma \vdash S_n :: T_n'$, and there exists a substitution s of types T_i (from $V_{\mathcal{T}}$) to T_i' such that $s(\sigma(\mathcal{T}))$ is $\langle T_1', \ldots, T_n' \rangle$.*

Example 3. ***Listing*** *6.5 shows an application of parametrized theory $\mathcal{T}_{color}$ defined in* ***Listing*** *6.1. This application of $\mathcal{T}_{color}$ is well-typed with the substitution `N` $\mapsto$ `Node`, `C` $\mapsto$ `Col`.*

We point out that by extending the definition of second-order formulas with theory applications, these constructs can occur (recursively) in theory definitions. We introduce the notion of *stratification*, and require that the set of all defined parametrized theories be *stratified*.

Definition 40 (Stratification). *A set of parametrized theories $\{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ is* stratified *if there exists a partial ordering such that for every theory $\mathcal{T}_i$ whose formula $\phi_{\mathcal{T}_i}$ applies a theory $\mathcal{T}_j$, it holds that $\mathcal{T}_i$ precedes $\mathcal{T}_j$ in the ordering.*

Variant World quantifications

Now that we have introduced parametrized theories, we can introduce variant world quantifications. Recall that the common core of a class of variant worlds

was specified using sets of tuples of terms, either given by enumeration or by a formula. We formalize these common core sets:

Definition 41 (common core sets). *Given a predicate or function P with arity n from vocabulary V:*

- $P : \{\bar{t}_1, \ldots, \bar{t}_k\}$ *is a* common core set for P *iff every* $\bar{t}_i$ *is an* n*-ary tuple* $(t_{i,1}, \ldots, t_{i,n})$*,*
- $P : \{(x_1, \ldots, x_n) \mid \phi\}$ *is a* common core set for P *(by formula) iff* ϕ *is a formula whose free variables are a subset of* $\{x_1, \ldots, x_n\}$*.*

Definition 42 (Well-Typed Common Cores). *A common core set $P : s$ is well-typed ($\Gamma \vdash P : s$) iff $P :: (T_1, \ldots, T_n) \in \Gamma$ or $P :: T_1, \ldots, T_n \to T \in \Gamma$ (i.e., P is an n-ary predicate or function symbol in Γ) and $P : s$ is of the form:*

- $P : \{\bar{t}_1, \ldots, \bar{t}_k\}$ *where each* $\bar{t}_i$ *is an* n*-ary tuple of terms* $(t_{i,1}, \ldots, t_{i,n})$ *s.t.* $\Gamma \vdash t_{i,1} :: T_1, \ \ldots, \ \Gamma \vdash t_{i,n} :: T_n$*; or*
- $P : \{(x_1, \ldots, x_n) \mid \phi\}$ *and* $\Gamma, x_1 :: T_1, \ldots, x_n :: T_n \vdash \phi$*.*

We extend the definition of formulas (**Definition** 38) and terms (**Definition** 31) to include quantifications over variant worlds and the unshadowing operator $^{\uparrow}$.

Definition 43 (Second-Order Formulas ext.).

- $\Box\mathcal{T}(S_1 : s_1, \ldots, S_n : s_n) : \phi$ *and* $\Diamond\mathcal{T}(S_1 : s_1, \ldots, S_n : s_n) : \phi$ *are formulas if $\mathcal{T}$ is a parametrized theory with n arguments, $S_1, \ldots, S_n$ are symbols, every s_i is a common core set for symbol S_i and ϕ is a formula.*
- $P^{\uparrow}(t_1, \ldots, t_n)$ *is a formula if P is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms.*

Definition 44 (Terms ext.).

- $t^{\uparrow}$ *is a term iff t is a variable, predicate symbol P or function symbol f.*
- $f^{\uparrow}(t_1, \ldots, t_n)$ *is a term iff f is a function and $t_1, \ldots, t_n$ are terms.*

Note that this definition limits the $^{\uparrow}$ operator: it cannot be applied iteratively.

To check whether a formula or term is well-typed, we must keep track of whether we can dereference a variable or symbol using $^{\uparrow}$. We extend the typing context Γ s.t. it contains records $S^{\uparrow}$ for symbols that can be dereferenced.

Definition 45 (Well-typed Formulas ext.).

- $\Box\mathcal{T}(S_1 : s_1, \ldots, S_n : s_n) : \phi$ *and* $\Diamond\mathcal{T}(S_1 : s_1, \ldots, S_n : s_n) : \phi$ *are well-typed iff*

   1. *for every common core set* $S_i : s_i$, $\Gamma \vdash S_i : s_i$,
   2. *there exists a substitution* s *such that* $s(\sigma(\mathcal{T})) = \langle T_1, \ldots, T_n \rangle$, *where* $\Gamma \vdash S_1 :: T_1$, *...,* *and*
   3. $\Gamma, S_1^{\uparrow} :: T_1, \ldots, S_n^{\uparrow} :: T_n \vdash \phi$.
- $P^{\uparrow}(t_1, \ldots, t_n)$ *is well-typed iff* $P^{\uparrow} :: (T_1, \ldots, T_n) \in \Gamma$ *and* $\Gamma \vdash t_1 :: T_1$, *...*, $\Gamma \vdash t_n :: T_n$.

Definition 46 (Well-typed Terms ext.).

- $t^{\uparrow}$ *is well-typed with type* T *iff* $t^{\uparrow} :: T \in \Gamma$.
- $f^{\uparrow}(t_1, \ldots, t_n)$ *is well-typed with type* T *iff* $f^{\uparrow} :: T_1, \ldots, T_n \to T \in \Gamma$ *and* $\Gamma \vdash t_1 :: T_1$, *...*, $\Gamma \vdash t_n :: T_n$.

6.2.1 Semantics

This section formalizes the semantics of our constructs.

Definition 47 (Structure). *A structure* $\mathcal{I}$ *for a vocabulary* V *assigns 1) for every type* T_i *a finite domain* D_i, *2) for every predicate* P *of type* $(T_1, \ldots, T_n)$ *in* V *a relation* $P^{\mathcal{I}} \subseteq D_1 \times \ldots \times D_n$, *and 3) for every function* f *of type* $T_1, \ldots, T_n \to T_0$ *in* V *a function* $f^{\mathcal{I}} :: D_1, \ldots, D_n \to D_0$.

Definition 48 (Skeleton Structure). *Every parametrized theory* $\mathcal{T}$ *has a (unique)* skeleton structure $\mathcal{I}_{\mathcal{T}}$ *associated with it that consists only of domains for the types in* $V_{\mathcal{T}}$ *that are interpreted by* $\mathcal{T}$.

Recall that structures can be modified and extended using the notation $\mathcal{I}[x : v]$: if x already has a value in $\mathcal{I}$, its value is modified, otherwise $\mathcal{I}$ is extended with a value for x. The same holds for *skeleton structures*.

We extend the valuation function $(\cdot)^{\mathcal{I}}$ to common core sets $P : s$:

Definition 49. $(P : s)^{\mathcal{I}}$ *is defined as follows:*

- $P : \{\bar{t}_1, \ldots, \bar{t}_k\}^{\mathcal{I}} = \{\bar{t}_1{}^{\mathcal{I}}, \ldots, \bar{t}_k{}^{\mathcal{I}}\}$
- $P : \{(x_1, \ldots, x_n) | \phi\}$ *where the arguments of* P *have types* T_i *and* D_i *is the domain for* T_i *in* $\mathcal{I}$, *evaluates under* $\mathcal{I}$ *to* $\bigcup_{d_1 \in D_1, \ldots} \{(d_1, \ldots, d_n)\}$ *s.t.* $\mathcal{I}[x_1 : d_1, \ldots, x_n : d_n] \models \phi$,

Definition 50 (Coinciding Structures). *We say structure* $\mathcal{I}$ coincides *with structure* $\mathcal{I}'$ *on symbol* S *with common core set* s *and substitution of* S *by* S' *iff for all* $(d_1, \ldots, d_n) \in (S : s)^{\mathcal{I}}$, $S^{\mathcal{I}}(d_1, \ldots, d_n) = S'^{\mathcal{I}'}(d_1, \ldots, d_n)$.

The satisfaction relation $\mathcal{I} \models \phi$ is defined using structural induction on the formula ϕ, and is defined as usual for $\wedge, \vee, \neg, \forall$ and $\exists$. For conciseness, we only consider theories with a single argument S':

Definition 51 (Satisfaction Relation).

- $\mathcal{I} \models *\mathcal{T}(S)$ *where* $\mathcal{T}$ *defines* S' *in its vocabulary* $V_\mathcal{T}$ *iff the skeleton structure* $\mathcal{I}_\mathcal{T}$ *extended with the value* $S^\mathcal{I}$ *for symbol* S'*, and its domains, satisfies* $\phi_\mathcal{T}$*, i.e.,* $\mathcal{I}_\mathcal{T}[S' : S^\mathcal{I}] \models \phi_\mathcal{T}$*.*
- $\mathcal{I} \models \Box\mathcal{T}(S : s) : \phi$ *(resp.* $\Diamond$*) where* $\mathcal{T}$ *defines* S' *in its vocabulary* $V_\mathcal{T}$*, iff for every (resp. some) interpretation* $\mathcal{I}'$ *s.t.*
  1. $\mathcal{I}' \models \phi_\mathcal{T}$
  2. $\mathcal{I}$ *coincides with* $\mathcal{I}'$ *on* s *with substitution* $S = S'$*,*

  it holds that $\mathcal{I}[S : S'^{\mathcal{I}'}, S^\uparrow : S^\mathcal{I}] \models \phi$*.*
- $\mathcal{I} \models P^\uparrow(t_1, \ldots, t_n)$ *iff* $(t_1^\mathcal{I}, \ldots, t_n^\mathcal{I}) \in P^{\uparrow\mathcal{I}}$*.*

Note that we introduce a new *symbol* $S^\uparrow$ for any symbols shadowed by our variant world quantifications. We extend the valuation function of terms under $\mathcal{I}$ with rules such that terms constructed using the $^\uparrow$ operator refer to these new symbols.

Definition 52 (Valuation of terms - extended).

- $(S)^{\uparrow\mathcal{I}} = (S^\uparrow)^\mathcal{I}$*, where* S *is a variable, predicate symbol* P *or function symbol* f*.*

Note that in this definition, on the left-hand side of the equality, $^\uparrow$ is the unshadowing operator, whereas on the right-hand side $^\uparrow$ is part of the name of $S^\uparrow$.

6.2.2 Expressivity

In this section we show that the proposed language constructs are, in fact, equally expressive as second-order quantification. To this end, we will first show how to transform second-order logic to first-order formulas extended with $\Box$ and $\Diamond$ expressions while preserving semantics. Subsequently, we will show that any theory application and any $\Box$ (resp. $\Diamond$) expression can likewise be translated to a second-order logic expression.

Transforming Second-Order Logic to FOL with $\Box$ and $\Diamond$

We will define the transformation $[\![\cdot]\!]$ from a second-order formula ϕ to a formula $[\![\phi]\!]$ in first-order logic with $\Box$ and $\Diamond$.

Listing 6.6: Theory introduced for a second-order quantification over a symbol P.

```
1 theory T{
2     type T_1.
3     ...
4     type T_n.

6     P :: (T_1, ..., T_n).

8     True.
9 }
```

We first note that the transformation of the predicate or function application, conjunction, disjunction, negation and the *first-order* quantifiers $\forall$ and $\exists$ is trivial.

Any universal second-order quantification $\forall P :: (T_1, \ldots, T_n) : \phi$ can be transformed in the following way: First, we extend the vocabulary with a symbol of type $(T_1, \ldots, T_n)$. Furthermore, we introduce the parametrized theory $\mathcal{T}$ from **Listing** 6.6. Now we can substitute the quantification $\forall P :: (T_1, \ldots, T_n) : \phi$ by $\Box\mathcal{T}(P : \{\}) : [\![\phi]\!]$.

This transformation sketched above can trivially be extended to cover both quantifications over functions and existential quantifications.

Proposition 2. *Let ϕ be a formula over vocabulary V. For any interpretation $\mathcal{I}$, $\mathcal{I} \models_{SO} \phi$ iff there is an interpretation $\mathcal{I}'$ s.t. $\mathcal{I}$ and $\mathcal{I}$' are vocabulary-equivalent on V ($\mathcal{I} =_V \mathcal{I}'$) and $\mathcal{I}' \models_{\Box,\Diamond} [\![\phi]\!]$.*

Proof. The proof proceeds by structural induction on the formula ϕ. We focus specifically on the case of second-order quantifications in ϕ.

First, we extend the transformation relation to structures $\mathcal{I}$; this transformation maps $\mathcal{I}$ to a structure $[\![\mathcal{I}]\!]_\phi$ over vocabulary V extended with any second-order variables P/n that are bound by second-order quantifications in ϕ. The *valuation* function of $[\![\mathcal{I}]\!]_\phi$ assigns these variables P/n any arbitrary value in its type.

By the satisfaction relation $\models_{SO}$, $\mathcal{I} \models_{SO} \forall P :: (T_1, \ldots, T_n) : \phi$ iff for all values v of type $(T_1, \ldots, T_n)$, $\mathcal{I}[P : v] \models_{SO} \phi$.

Considering the transformation of $\forall P :: (T_1, \ldots, T_n) : \phi$, by the satisfaction relation $\models_{\Box,\Diamond}$, $[\![\mathcal{I}]\!]_\phi \models_{\Box,\Diamond} \Box\mathcal{T}(P : \{\}) : [\![\phi]\!]$ iff for all interpretations $\mathcal{I}'$ s.t. $\mathcal{I}' \models_{\Box,\Diamond} \mathcal{T}$, it holds that $[\![\mathcal{I}]\!]_\phi[P : P^{\mathcal{I}'}] \models [\![\phi]\!]$.

It is easy to see that any value v of type $(T_1, \ldots, T_n)$ corresponds to a structure $\mathcal{I}'$ satisfying $\mathcal{T}$ and vice versa. Thus, $\mathcal{I} \models_{SO} \forall P :: (T_1, \ldots, T_n) : \phi$ iff $[\![\mathcal{I}]\!]_\phi \models_{\Box,\Diamond} \Box\mathcal{T}(P : \{\}) : [\![\phi]\!]$. □

Transforming FOL with □ and ◇ to Second-Order Logic

We define the translation $[\![\cdot]\!]$ from a formula ϕ in first-order logic extended with $\Box$ and $\Diamond$ to a second-order formula $[\![\phi]\!]$.

Consider the variant world quantification $\Box\mathcal{T}(P : c) : \phi$ where

- theory $\mathcal{T}$ has a single vocabulary symbol P' with type $(T'_1, \ldots, T'_n)$. Furthermore, we assume that its body $\phi_\mathcal{T}$ does not mention any parametrized theories $\mathcal{T}$'.
- P is a predicate symbol or variable with type $(T_1, \ldots, T_n)$, and
- c is a *common core set* for P: $\{(x_1, \ldots, x_n) \mid \phi_{cc}\}$

Such a variant world quantification can be transformed as follows: First, a new symbol P_{quant} with type $(T_1, \ldots, T_n)$ is introduced by universal quantification. The common core set is translated into the formula ψ_{cc}, defined as $\forall x_1 :: T_1, \ldots, x_n :: T_n : \phi_{cc} \Rightarrow (P(x_1, \ldots, x_n) \Leftrightarrow P_{quant}(x_1, \ldots, x_n))$. Lastly, we instantiate the body $\phi_\mathcal{T}$ of theory $\mathcal{T}$, by substituting all occurrences of P' by the earlier introduced P_{quant} and substituting any occurrences of the types $T'_1 \ldots T'_n$ by the types $T_1 \ldots T_n$. In summary, we transform the variant world quantification to $\forall P_{quant} :: (T_1, \ldots, T_n) : (\psi_{cc} \wedge \phi_\mathcal{T}[P'/P_{quant}, T'_1/T_1, \ldots, T'_n/T_n]) \Rightarrow [\![\phi]\!]$.

Variant world quantifications of the form $\Diamond\mathcal{T}(P : c) : \phi$ can be transformed to $\exists P_{quant} :: (T_1, \ldots, T_n)$.

This transformation can trivially be adapted to cover both variant world quantifications with multiple arguments as well as function symbol arguments. Furthermore, the stratification of theories justifies the assumption that the body of $\phi_\mathcal{T}$ does not mention any parametrized theories.

Proposition 3. *Let ϕ be a formula over vocabulary V. For any interpretation $\mathcal{I}$, $\mathcal{I} \models_{\Box,\Diamond} \phi$ iff the interpretation $\mathcal{I} \models_{SO} [\![\phi]\!]$.*

It is easy to see that this proposition holds, as quantification over models of a theory $\mathcal{T}$ corresponds directly to quantification over the symbols in its vocabulary.

6.3 Implementation

In this section, we discuss the implementation of parametrized theories, variant world quantifiers and the unshadowing operator $^{\uparrow}$. It extends the grounder discussed in **Chapter** 4.

We will illustrate the rewriting with our specification for the *chromatic coloring* problem as given in **Listing** 6.1 (see also **Listing** 6.7).

Listing 6.7: A theory representing the coloring constraints

```
1  theory T_color{
2    type N.
3    type C.

5    G :: (N,N).
6    f :: N → C.
7    U :: (C).

9    ∀ x, y :: N : G(x,y) ⇒ f(x) ≠ f(y).
10   ∀ c :: C : U(c) ⇔ (∃ x :: N : f(x) = c).
11 }

13 *T_color(G,f,U) ∧ □T_color(G:{(n1,n2) | True}, f:{}, U:{}) : ¬(U < U^↑).
```

6.3.1 Parametrized Theories and their Applications

First, we detail how parametrized theories and their applications are processed. The rewriting starts by performing a simple renaming such that every type and symbol has a *unique* name.

Consider the following application $*\mathcal{T}_{color}$`(Graph, Homomorphism, Used)` of the theory $\mathcal{T}_{color}$ of **Listing** 6.1. In this application, the two predicates `Graph` and `Used` have types `(Node, Node)` and `(Color)` respectively, while the function `Homomorphism` has type `Node` $\rightarrow$ `Color`.

To translate this theory application, the implementation instantiates the body of $\mathcal{T}_{color}$ as appropriate. From the signature $\sigma(\mathcal{T}_{color})$ of theory $\mathcal{T}_{color}$ we derive the substitutions of **Table** 6.1. Applying these to the body $\phi_{\mathcal{T}_{color}}$, we get the following instantiation for the symbols and types of theory $\mathcal{T}_{color}$:

```
∀ x, y :: Node : Graph(x,y) ⇒ Homomorphism(x) ≠ Homomorphism(y)
∧ ∀ c :: Color : Used(c) ⇔ (∃ x :: Node : Homomorphism(x) = c).
```

We can now replace the occurrence $*\mathcal{T}_{color}$`(Graph, Homomorphism, Used)` by the instantiated body.

Table 6.1: Substitutions for the application $*\mathcal{T}_{color}$(Graph, Homomorphism, Used).

Original	Substitution
G	Graph
f	Homomorphism
U	Used
N	Node
C	Color

6.3.2 Quantifying over Variant Worlds

We now show how the implementation processes quantifications over variant worlds and the unshadowing operator.

When encountering a quantification over variant worlds $\Box\mathcal{T}(S : s) : \phi$, we introduce new variables for the arguments S, quantifying them universally (existentially, for $\Diamond$ expressions).

For the variant world quantification from the *chromatic coloring* problem, $\Box\mathcal{T}_{color}$ (Graph:{(n1,n2) | t}, Homomorphism:{}, Used:{}) : $\neg$(Used < Used$^{\uparrow}$), the variables introduced are Graph', Homomorphism' and Used'.

From the signature of the theory $\mathcal{T}$, we derive a set of substitutions for the arguments, for the types in the signature $\sigma(\mathcal{T})$, and for any arguments with the unshadowing operator $^{\uparrow}$ applied. Note that no further implementation support is needed for the unshadowing operator.

For the signature of $\mathcal{T}_{color}$, these substitutions are given by **Table** 6.2.

By applying these substitutions to the body $\phi_{\mathcal{T}}$ and the formula ϕ, we get new formula $\phi_{\mathcal{T}_{sub}}$ and ϕ_{sub} respectively. For our example, this would give the following formulas:

```
(∀ x,y :: Node : Graph'(x,y) ⇒ Homomorphism'(x) ≠ Homomorphism'(y))∧
(∀ c :: Color : Used'(c) ⇔ (∃ x :: Node : Homomorphism'(x) = c)).

¬(Used' < Used).
```

Now, we must enforce the common core between the variant worlds considered by the quantification. To this end, we introduce a formula ψ_{link}, which encodes the information in the common core sets as formulas that we call 'linking formula'. A common core of the form $f : \{(x_1, \ldots, x_n) \mid \psi\}$, with f a function of type $T_1, \ldots, T_n \to T$, ψ_{link} has linking formula $\forall x_1 :: T_1 : \ldots : \forall x_n :: T_n :$

Table 6.2: Substitutions for the variant world quantification $\Box\mathcal{T}_{color}$(Graph:{(n1 ,n2)|t }, Homomorphism:{}, Used:{}):ϕ.

Original	Substitution
G	Graph'
G$^{\uparrow}$	Graph
f	Homomorphism'
f$^{\uparrow}$	Homomorphism
U	Used'
U$^{\uparrow}$	Used
N	Node
C	Color

$\psi \Rightarrow f(x_1, \ldots, x_n) = f'(x_1, \ldots, x_n)$. For the common core of a predicate P, the same linking formula can be used, but with an equivalence ($\Leftrightarrow$) replacing the equality ($=$).

For our example, the common coreset Graph:{(n1,n2)|t} translates to the linking formula $\forall$n1, n2 :: Node : Graph(n1,n2) $\Leftrightarrow$ Graph'(n1,n2).The empty common core sets $\{\}$ for Homomorphism and Used simply translate to **t**.

The full transformation can then be summarized as follows:

$$\Box\mathcal{T}(S:s):\phi \rightsquigarrow \forall S' :: T : (\phi_{\mathcal{T}_{sub}} \wedge \psi_{link}) \Rightarrow \phi_{sub}.$$

$$\Diamond\mathcal{T}(S:s):\phi \rightsquigarrow \exists S' :: T : \phi_{\mathcal{T}_{sub}} \wedge \psi_{link} \wedge \phi_{sub}.$$

For the example variant world quantification from the *chromatic coloring problem* this becomes:

```
∀ Graph' :: (Node, Node) : ∀ Homomorphism' :: Node → Color :
∀ Used' :: (Color) :
((∀n1, n2 :: Node : Graph(n1,n2) ⇔ Graph'(n1,n2)) ∧
 (∀ x,y :: Node : Graph'(x,y) ⇒ Homomorphism'(x) ≠ Homomorphism'(y))∧
 (∀ c :: Color : Used'(c) ⇔ (∃ x :: Node : Homomorphism'(x) = c)))
⇒ ¬(Used' < Used).
```

The resulting formula(s) are strictly second-order formulas and can be handled by the grounder that was introduced in **Chapter** 4.

6.4 Use case: Zebra Puzzle

As previously stated, theory applications and variant world quantifications can express certain inferences in the language itself. One important advantage of this is the possibility to continue reasoning on the result of these inferences. The *Zebra puzzle* use case illustrates this advantage.

The *Zebra puzzle*, also known as the *Einstein* puzzle, is a well-known puzzle where one is asked to infer from a number of statements, or *clues*, for a set of persons of different nationality in what color house they live, what pet they have, which brand of cigarettes they smoke etc., for example "The man who smokes Chesterfields lives in the house next to the man with the fox". The *Zebra Puzzle*, and its variants are called called *logic grid* puzzles, and can be modeled as a type of *Constraint Satisfaction* Problem (CSP) using the following approach:

- Every property, e.g., nationality or cigarette brand, corresponds to a type.
- Every combination of two properties corresponds to a binary predicate, e.g., `NameBrand`, over the two types that the properties represent.
- The clues are modeled as first-order sentences (constraints) over these binary predicates.
- For every binary predicate, a sentence is added that expresses bijectivity: every value of one property is related with exactly *one* value of the other property.
- For every combination of three different binary predicates, a first-order sentence expressing *transitivity* is added. Transitivity says, for example, that "if the Englishman has a snail and lives in the red house, then the snail is related to the red house as well".

In recent years, Claes et al. [28, 16] have investigated how a system can produce not only a solution of a logic grid puzzle, but also an easy-to-understand proof, i.e., 'a sequence of *simple* explanations' [16]. Claes et al.'s system ZebraTutor proceeds by first separating the different constraints in the logic grid puzzle into different theories. ZebraTutor subsequently performs *Minimal Unsatisfiable Core* extraction inference on combinations of these partial theories, guided by a heuristic based on a cost function.

One disadvantage of this approach is that the heuristic does not guarantee *optimality* with respect to the cost function. However, by considering simple explanations as *propagations* from a *subset* of the theory, we can express the

Table 6.3: Solution of the simple Pasta puzzle.

	Arrabiata	**Bolognese**	**Carbonara**	**Penne**	**Farfalle**	**Spirelli**
Andrea	-	-	✓	✓	-	-
Bart	✓	-	-	-	✓	-
Casper	-	✓	-	-	-	✓
Penne	-	-	✓			
Farfalle	✓	-	-			
Spirelli	-	✓	-			

search for a simple explanation as a *parametrized theory* application, cfr. the earlier discussed brave and cautious reasoning.

Example 4 (Pasta). *Consider the (simple) example logic grid puzzle 'Pasta' with only three properties: person names (*Andrea*,* Bart *and* Casper*), choice of Pasta (*Penne*,* Farfalle *and* Spirelli*), and choice of Sauce (*Arrabiata*,* Bolognese *and* Carbonara*).*

Furthermore, we are given the following four clues:

1. *Andrea ate penne.*
2. *The person who ate penne, had carbonara sauce.*
3. *Bart ate penne or farfalle.*
4. *Spirelli was eaten with bolognese.*

The solution of this puzzle is visualized in ***Table*** *6.3.*

First, we define a theory representing the logic grid puzzle, for example $\mathcal{T}_{Pasta}$ of **Listing** 6.8. The theory includes reification of the different constraints representing the *clues*, the *transitivity*, and the *bijectivity* of the relations, by introducting before each sentence ϕ an implication `T(n)` $\Rightarrow \phi$. The reification of these constraints allows manipulation of the theory used to derive *propagations*, i.e., a constraint ϕ, expressed by `T(n)` $\Rightarrow \phi$, is only *active* when the reification literal `T(n)` is true.

It is easy to see that theory $\mathcal{T}_{main}$ of **Listing** 6.9 simply expresses model expansion over the full $\mathcal{T}_{pasta}$ theory, by activating every constraint, due to **Line** 17.

Note that the output predicates of the theory application are always two-valued. In the theory above, there is obviously only one correct solution for the entire

Listing 6.8: Theory representing the Pasta logic grid puzzle

```
1 theory T_Pasta{
2  type Person.
3  type Pasta.
4  type Sauce.
5  type Tseitins.

7  PersonPasta :: (Person, Pasta).
8  PersonSauce :: (Person, Sauce).
9  PastaSauce  :: (Pasta, Sauce).
10 T           :: (Tseitins).

12 Person   := {Andrea; Bart; Casper}.
13 Pasta    := {Penne; Farfalle; Spirelli}.
14 Sauce    := {Arrabiata; Bolognese; Carbonara}.
15 Tseitins := {1;2;3;4;5;6;7;8;9;10}.

17 // clues
18 T(1) ⇒ PersonPasta(Andrea, Penne).
19 T(2) ⇒ ∃ person :: Person : PersonPasta(person, Penne) ∧ PersonSauce(person, Carbonara).

20 T(3) ⇒ ∃ pasta :: Pasta : PersonPasta(Bart, pasta) ∧ (pasta = Penne ∨ pasta = Farfalle)
      .
21 T(4) ⇒ PastaSauce(Spirelli, Bolognese).

23 // Transitivity
24 T(5) ⇒ ∀person :: Person : ∀pasta :: Pasta : ∀sauce :: Sauce : (PersonPasta(person,
       pasta) ∧  PastaSauce(pasta, sauce)) ⇒ PersonSauce(person, sauce).
25 T(6) ⇒ ∀person :: Person : ∀pasta :: Pasta : ∀sauce :: Sauce : (PersonPasta(person,
       pasta) ∧  PersonSauce(person, sauce)) ⇒ PastaSauce(pasta, sauce).
26 T(7) ⇒ ∀person :: Person : ∀pasta :: Pasta : ∀sauce :: Sauce : (PersonSauce(person,
       sauce) ∧  PastaSauce(pasta, sauce)) ⇒ PersonPasta(person, pasta).

28 // Bijectivity and existence
29 T(8) ⇒ (∀p1, p2 :: Person : ∀ s1, s2 :: Sauce : (PersonSauce(p1,s1) ∧ PersonSauce(p2,s2
       ) ⇒ ((p1 = p2 ∧ s1 = s2) ∨ (p1 ≠ p2 ∧ s1 ≠ s2)))).
30 T(8) ⇒ ∀person :: Person : ∃ sauce :: Sauce : PersonSauce(person,sauce).
31 [...] //bijectivity and existence for the PersonPasta and PastaSauce predicates.
32 }
```

puzzle, but even when we deactivate certain constraints, we simply find *multiple* two-valued results.

To go from *model expansion* to *propagation*, we need to:

- represent *three-valued* structures, as both for the 'input' of the propagation and the 'output' of the propagation, and
- compare the *three-valued* and *two-valued* representations according to the precision order $\leq_p$.

A single predicate P in the structure can be represented in a three-valued way by two predicates P_{ct} and P_{cf} representing the known *true* and *false* subsets of P.

Listing 6.9: Theory $\mathcal{T}_{modelexpansion}$ modeling model expansion over the pasta puzzle

```
theory T_modelexpansion {
 type Person.
 type Pasta.
 type Sauce.
 type Tseitins.

 PersonPasta :: (Person, Pasta).
 PersonSauce :: (Person, Sauce).
 PastaSauce  :: (Pasta, Sauce).
 T           :: (Tseitins).

 Person   := {Andrea; Bart; Casper}.
 Pasta    := {Penne; Farfalle; Spirelli}.
 Sauce    := {Arrabiata; Bolognese; Carbonara}.
 Tseitins := {1;2;3;4;5;6;7;8;9;10}.

 ∀ n :: Tseitins : T(n).
 *T_Pasta(PersonPasta, PersonSauce, PastaSauce, T).
}
```

To compare two three-valued representations under the precision order $\leq_p$, we can simply compare the the true and false subsets separately using $=<$, i.e., $P_{ct} =< P'_{ct} \wedge P_{cf} =< P'_{cf}$[1].

To compare a three-valued representation P_{ct}, P_{cf} to a two-valued representation of P', we introduce a theory $\mathcal{T}_{\leq_{p,3V}}$ (see **Listing** 6.10). This theory $\mathcal{T}_{\leq_{p,3V}}$ expresses that the two-valued predicate P' (with a two-valued representation `P'`) is $\geq_p$ than a three-valued predicate P (represented by `P_ct` and `P_cf`). We can equivalently state that P' (represented by `P'`) is *consistent* with P (represented by `P_ct` and `P_cf`).

Listing 6.10: Theory that expresses that P' is $\geq_p$ than a three-valued predicate P (represented by `P_ct` and `P_cf`.

```
theory T_≤p,3V {
 type T1.
 type T2.

 P_ct :: (T1, T2).
 P_cf :: (T1, T2).
 P'   :: (T1, T2).

 ∀ x :: T1 : ∀ y :: T2 : (P_ct(x,y) ⇒ P'(x,y)) ∧ (P_cf(x,y) ⇒ ¬P'(x,y)).
}
```

Now we can model *propagation* in the Pasta puzzle by the parametrized theory $\mathcal{T}_{propagation}$, **Listing** 6.11. It takes as arguments 6 binary predicates forming

[1] Recall that $=<$ on predicates corresponds to the subset relation $\subseteq$

a three-valued representation of the *input* structure for *propagation*, 6 binary predicates forming a three-valued representation of the *output* of the propagation, and a single predicate indicating which constraints were active. In **Lines** 30–38, the theory expresses that the three-valued representation of the output is strictly more precise than the three-valued representation of the input. **Lines** 40–49 subsequently express that, for the set `T` determining the active constraints, every solution of the Pasta puzzle that extends the input structure must also extend the output structure, i.e., the predicates representing the input structure and the output structure respectively differ only in *cautious* consequences.

As the *propagation* inference is expressed in the language itself, we can now continue reasoning on its results, e.g., we can express that there is no `T'` $<$ `T` of tseitins that allows propagations[2]:

$*\mathcal{T}_{propagation}$`(..., T)` $\wedge$ $\Box\mathcal{T}_{propagation}$`(..., T:{})` : $\neg$ `(T <` $\mathtt{T}^{\uparrow}$`)`

When our proposed language is extended with aggregates (`COUNT`[3], `SUM`[4]), and with only minimal changes to the specification, we can instead compare the set of active tseitins by cardinality or even using a weight function, e.g.

$*\mathcal{T}_{propagation}$`(..., T)` $\wedge$ $\Box\mathcal{T}_{propagation}$`(..., T:{}) : COUNT{t :: Tseitins : T(t)}`
$\geq$ `COUNT{t :: Tseitins :` $\mathtt{T}^{\uparrow}$`(t)}.`

This shows the flexibility offered by being able to express these inferences in the language itself.

With the theory $\mathcal{T}_{propagation}$, given a three-valued representation of the knowledge we already have, we can use parametrized theory application to find consequences and a representation of their *explanation* (given by the true tseitins `T`). By combining this with variant world quantifications, we can restrict ourselves to finding consequences with explanations that are *minimal* w.r.t some ordering such as subset minimality or cardinality, e.g., '*simple explainable consequences*' for the ZebraTutor.

[2]For readability, we have omitted the passing of the arguments that represent the input and result of propagation.

[3]`COUNT` should be a built-in higher-order function on a set of domain elements; the elements of this set are either specified by enumeration or as all possible bindings to a set of variables that satisfy a given formula

[4]`SUM` should be a built-in higher-order function on a multiset; the elements of this multiset are either specified by enumeration or by a term t with free variables. In the second case, the multiset consists of all terms obtained by instantiating the free variables with those bindings that satisfy the given formula.

Listing 6.11: Theory $\mathcal{T}_{propagation}$ modeling propagation over the Pasta puzzle

```
1 theory T_propagation {
2  type Person.
3  type Pasta.
4  type Sauce.
5  type Tseitins.

7  // Representation of the three-valued input structure of propagation.
8  PersonPasta_ct :: (Person, Pasta).
9  PersonPasta_cf :: (Person, Pasta).
10 PersonSauce_ct :: (Person, Sauce).
11 PersonSauce_cf :: (Person, Sauce).
12 PastaSauce_ct  :: (Pasta, Sauce).
13 PastaSauce_cf  :: (Pasta, Sauce).

15 // Representation of the three-valued output structure of propagation.
16 PersonPasta_ct' :: (Person, Pasta).
17 PersonPasta_cf' :: (Person, Pasta).
18 PersonSauce_ct' :: (Person, Sauce).
19 PersonSauce_cf' :: (Person, Sauce).
20 PastaSauce_ct'  :: (Pasta, Sauce).
21 PastaSauce_cf'  :: (Pasta, Sauce).

23 T           :: (Tseitins).

25 Person   := {Andrea; Bart; Casper}.
26 Pasta    := {Penne; Farfalle; Spirelli}.
27 Sauce    := {Arrabiata; Bolognese; Carbonara}.
28 Tseitins := {1;2;3;4;5;6;7;8;9;10}.

30
31 PersonPasta_ct =< PersonPasta_ct' ∧ PersonPasta_cf =< PersonPasta_cf'.
32 PersonSauce_ct =< PersonSauce_ct' ∧ PersonSauce_cf =< PersonSauce_cf'.
33 PastaSauce_ct  =< PastaSauce_ct'  ∧ PastaSauce_cf  =< PastaSauce_cf'.

35 PersonPasta_ct < PersonPasta_ct' ∨ PersonPasta_cf < PersonPasta_cf' ∨
36 PersonSauce_ct < PersonSauce_ct' ∨ PersonSauce_cf < PersonSauce_cf' ∨
37 PastaSauce_ct  < PastaSauce_ct'  ∨ PastaSauce_cf  < PastaSauce_cf'.
38


40
41 ∀ PersonPasta :: (Person, Pasta) : ∀ PersonSauce :: (Person, Sauce) : ∀ PastaSauce :: (
       Pasta, Sauce) :
42    (*T_Pasta(PersonPasta, PersonSauce, PastaSauce, T) ∧
43     *T_≤p,3V(PersonPasta_ct, PersonPasta_cf, PersonPasta) ∧
44     *T_≤p,3V(PersonSauce_ct, PersonSauce_cf, PersonSauce) ∧
45     *T_≤p,3V(PastaSauce_ct, PastaSauce_cf, PastaSauce)) ⇒
46        (*T_≤p,3V(PersonPasta_ct', PersonPasta_cf', PersonPasta) ∧
47         *T_≤p,3V(PersonSauce_ct', PersonSauce_cf', PersonSauce) ∧
48         *T_≤p,3V(PastaSauce_ct', PastaSauce_cf', PastaSauce)).
49
50 }
```

6.5 Conclusion

In **Chapter** 5, we looked at the practical expressivity of specification languages, starting from the assertion that a lot of information was subject to *duplication*. We preferred instead to express that same information in a more generic way, leading to the introduction of templates and reducing duplication. We identified second-order definitions as a very expressive language construct, capable of abstracting over relations in much the same way as we expected from templates, and thus defined the semantics of templates through second-order definitions. However, the expressivity of second-order definitions also posed a challenge, as the computational expressivity of specification languages such as the FO(·) language of the IDP system is limited to existential second-order or NP. Thus, we limited templates to specific second-order definitions that increased the practical expressivity and not the computational expressivity.

In this chapter, we started from the SOGrounder, a grounder with second-order logic as its specification language, a much more expressive language in itself, however, still lacking the abstraction capabilities that we were looking for in templates. However, in this chapter, we continued on the observation that, as far as we could identify, the natural problems of a level higher than NP in the polynomial hierarchy PH in fact were on that level because they (repeatedly, or under negation) solved a problem one level lower. Often, though not necessarily always, there is a certain level of self-similarity in play; knowledge about a certain problem or concept can be expressed on a given level, but when we in fact want the solution that is *minimal* by some criterium (e.g., chromatic coloring), *critical* (e.g., actual causation or inconsistent cores) or *unique*,the concepts raise in descriptive complexity [114]. The observation above is not surprising if one considers the *oracle*-based description of PH.

Likewise, it seems closely linked to the secondary advantage of templates quoted in **Chapter** 5, that even if a template is used only once, it can benefit readability by grouping and naming constraints. This observed *pattern* in second-order logic led us to the idea that *theories*, the logical object by which we would gather the knowledge of any problem domain, were a prime candidate to semantically define *templates*. In related work, Eiter et al. [47] already considered *theories* as a form of abstraction and reuse when working on the DLVHEX system. However, in the work on DLVHEX, such theories are handled by its the external computation system, leading to separate solver calls.

We therefore started the chapter by providing some examples of problems that to us, exhibit the pattern of self-similarity, together with the language constructs that help us express them: *parametrized theories*, *variant world quantifications* and *common core* expressions.

We proceeded by the formalization of second-order logic extended with these new language constructs. We defined their syntax and semantics, and show that, as we would expect of templates, their introduction does not raise computational expressivity. Subsequently, we detailed our implementation, the rewriting mechanism to translate our new language constructs to standard second-order logic as can be processed by the SOGrounder. Furthermore we detail a use case that, to us, shows how in fact the language constructs can be used to integrate well-known inferences in the language, possibly in slight, application-specific variations, in a way that allows continued reasoning on the results.

Chapter 7

An Overview of Problems with Second-Order Constraints

In this chapter, we give a select overview of some problems in whose modelings second-order constraints naturally arise. Making such a selection is, in its very nature, a subjective task.

Thinking back to Immerman [78] result that *second-order logic* captures the Polynomial Hierarchy PH, we know that every problem $\mathcal{P}$ expressible in second-order logic can be polynomially transformed to deciding satisfiability of a k-QBF formula. Thus, in some sense, the only problem one needs to consider is that of deciding satisfiability of k-QBF. However, it is important to keep in mind that the polynomial transformation does not necessarily provide a nice translation of the domain of discourse and the terminology of $\mathcal{P}$ to k-QBF.

As our interest in supporting second-order logic stems from a desire to allow for better knowledge representation, we place great importance on the domain of discourse and the terminology associated with every problem. Therefore, we focus in this overview on problems that are of interest because of their simple and elegant natural language problem statements and/or because of their relevance in literature from other fields, originating from theoretical study or practical applications.

The problems in this overview can (and should be) considered supplementary examples (in addition to the earlier discussed *graph mining*, *strategic companies*

Table 7.1: The $*$ operation on labels, modeling the relationship between a directed edge and its start and end vertex.

$l1$	$l2$	$l1 * l2$
$+$	$+$	$+$
$+$	$-$	$-$
$-$	$-$	$+$
$-$	$+$	$-$

and the examples of **Chapter** 6) motivating the choice to support second-order logic; they can also be read as further examples of second-order modelings and the pattern discussed in **Chapter** 6. Although these problems are intended to provide (additional) motivation for adding support of second-order logic, we only discuss them now because representing them properly depends heavily on the additional constructs introduced in **Chapter** 6, or in some cases even the addition of aggregates to the language.

The chapter consists of new, unpublished work.

Personal contribution: 100%.

7.1 Minimal Inconsistent Cores

The concept of Minimal Inconsistent Cores has been introduced by Gebser et al. [63] to detect inconsistencies in biological networks that model polarity of biochemical and genetic reactions.

Specifically, the networks are modeled by *influence graphs* [9], which are directed graphs in which the *directed edges* are labeled with a sign $(+/-)$. Semantically, positive edges model activations, while negative edges model inhibitions. Based on observations in experiments, one can subsequently label the nodes of a network with a sign $(+/-)$ as well, signifying increased respectively decreased presence of the reactant represented by the vertex. We introduce a binary, associative operation $*$ over labels that models the relationship between a directed edge and its start and end vertices, whose definition is given by **Table** 7.1.

Definition 53 (consistency). *Consider a graph $\mathcal{G}$ with vertices V and directed edges E, and a pair $\langle l_E, l_V \rangle$ that consists of an edge-labeling function $l_E :: E \to \{+/-\}$ and a vertex-labeling function $l_V :: V \to \{+/-\}$. The pair $\langle l_E, l_V \rangle$ is* consistent with $\mathcal{G}$ on a vertex v *if there is an edge $(u, v) \in E$ s.t. $l_V(v) = l_V(u) * l_E(u, v)$ (See **Table** 7.1).*

A pair $\langle l_E, l_V \rangle$ *is* consistent with $\mathcal{G}$ *if it is consistent with* $\langle V, E \rangle$ *on* all *vertices* $v \in V$.

It is possible to extend the notion of consistency to:

- disregard *input* vertices, i.e., vertices without incoming edges, representing controlled variables in the experiments,
- extend the labeling to include orthogonal or non-significant influences.

A function that labels a set $V' \subset V$ is called a partial vertex-labeling function, referred to as $l_{V,p}$. We say a function l_V is a *total extension* of $l_{V,p}$ if l_V labels V and for all $v \in V'$ $l_V(v) = l_{V,p}(v)$. We use the same terminology for labelings of edges.

Definition 54 (Minimal Inconsistent Core). *Given an graph* $\mathcal{G}$ *and a pair* $\langle l_{E,p}, l_{V,p} \rangle$ *of* partial *edge- and vertex-labeling functions, a set of vertices* IC *is an* inconsistent core *of* $\mathcal{G}$ *iff every pair of total extending edge- and vertex-labelings* $\langle l_E, l_V \rangle$ *is* inconsistent *with* $\mathcal{G}$ *on* some *vertex* $i \in IC$.

An inconsistent core IC *is a* minimal inconsistent core *of* $\mathcal{G}$ *if none of its subsets* $Y \subset IC$ *is an inconsistent core of* $\mathcal{G}$.

In **Listing** 7.1, we model the concept of Minimal Inconsistent Cores, where the minimal inconsistent core is represented by a set of vertices `IC`. First, on **Lines** 15–26, we model the concept of total extension, representing partial labelings as predicates. Defining the parametrized theory `Consistent` (**Lines** 28–40), we model what it means for an edge- and vertex-labeling (`edgelabel`, `vertlabel` respectively) to be consistent with a graph on a set `IC`.

On **Lines** 42–43, we impose the restriction that no extension of `partialedge` and `partialvert` is consistent with the graph on the set of vertices `IC`.

Finally, we express the minimality condition on **Lines** 45–48; for every vertex `x` in `IC`, the partial labelings can be extended such that the extensions are consistent on all vertices in `IC` $\setminus$ {`x`}, i.e., the set `IC` $\setminus$ {`x`} is *not* an inconsistent core. Note that this implies that no set $S \subset$ `IC` $\setminus$ {`x`} is an inconsistent core.

From **Chapter** 6, we know that the variant world quantifications of **Lines** 42 and 46 correspond to second order quantifications.

Note that the minimality condition could be expressed by quantifying over the subsets of `IC` directly, and expressing that it is not an inconsistent core. Formulated this way, however, it is clear that when `IC` is given, there is no alternation of second-order quantifiers, only the conjunction of a second-order

universal and a second-order existential quantifier. This corroborates the assertion by Gebser et al. [63] that determining whether a given set is a minimal inconsistent core is in D^P, i.e., the second level of the Boolean Hierarchy [114], characterizing "languages that are the intersection of a language in NP and a language in coNP"[1] [114].

7.2 Optimal Stable Matching Problem

In their 1962 paper, Gale and Shapley [57] set out to solve the problem of matching applicants to colleges, or bachelor men and women, formalizing what became known as the *Stable Matching Problem*[2]. Given a number of men and women, where each man has ranked the women according to his preference and vice-versa, they wanted to match these men and women in such a way that the resulting matching was *stable*, i.e., no pair of a man and a woman exists that (1) are not matched but, (2) would prefer each other over their actual partners. In other words, no two individuals can be paired such that both would feel that they would "trade up".

Definition 55 (Stable Matching). *Given sets M and W, as well as total preference orderings $P_m :: (W, W)$, respectively $P_w :: (M, M)$ for every $m \in M$ and $w \in W$, we call a bijection $f :: M \to W$ a* stable matching *if there exist no m_u, w_u s.t. $P_{m_u}(w_u, f(m_u))$ and $P_{w_u}(m_u, f^{-1}(w_u))$.*

The problem is of specific interest to a knowledge representation approach (e.g., the study by [31]) as there exist many different variations; some variations introduce ties in the ordering, others introduce the possibility to prefer remaining unmatched above some potential partners. Faced with these variations, an elaboration tolerant approach based on declarative modeling languages has significant advantages over specialized algorithms.

Another aspect that can vary is when a stable matching is considered *optimal*. As many different *stable matchings* can exist, one can associate with each matching a *cost* and find the stable matching that minimizes this cost.

Definition 56 (Matching cost). *Consider the total preference orderings P_m and P_w. The cost $c_m(f)$ of an individual pairing $(m, f(m))$ is equal to the number of partners m would prefer to $f(m)$. The cost $c_w(f)$ of a pairing $(f^{-1}(w), w)$ is equal to the number of partners w would prefer to $f^{-1}(w)$.*

[1] Note that this implies NP, coNP $\subseteq D^P \subseteq \Delta_2^P$.
[2] Or sometimes "Stable Marriage Problem".

Listing 7.1: Specification of the *Minimal Inconsistent Cores* problem.

```
type Vertices.
type Label.

edge        :: (Vertices, Vertices).
IC          :: (Vertices).
partialedge :: (Vertices, Vertices, Label).
partialvert :: (Vertices, Label).
combine     :: Label, Label → Label.
ledge       :: Vertices, Vertices → Label.
lvert       :: Vertices → Label.

Label   := {pos;neg}.
combine := {pos,pos ↦ pos; pos,neg ↦ neg; neg,pos ↦ neg; neg,neg ↦ pos}.

theory Extends {
  type Vertices.
  type Label.

  partialedge :: (Vertices, Vertices, Label).
  partialvert :: (Vertices, Label).
  edgelabel   :: Vertices, Vertices → Label.
  vertlabel   :: Vertices → Label.

  ∀ x, y :: Vertices : ∀ l :: Label : (partialedge(x,y,l) ⇒ edgelabel(x,y)=l).
  ∀ x :: Vertices : ∀ l :: Label : (partialvert(x,l) ⇒ vertlabel(x)=l).
}

theory Consistent {
  type Vertices.
  type Label.

  IC              :: (Vertices).
  edge            :: (Vertices, Vertices).
  edgelabel       :: Vertices, Vertices → Label.
  vertlabel       :: Vertices → Label.
  combine         :: Label, Label → Label.

  ∀ v :: Vertices : IC(v) ⇒
     ∃ n :: Vertices : (edge(n,v)) ∧ vertlabel(v) = combine(vertlabel(n), edgelabel(n,v)).

}

□Extends(partialedge:{x,y,z|true}, partialvert:{x,y|true}, ledge:{}, lvert:{}) :
    ¬ *Consistent(IC,edge,ledge,lvert,combine).

∀ x :: Vertices : IC(x) ⇒
    (◊Extends(partialedge:{a,b,c | True}, partialvert:{a,b | True}, ledge:{}, lvert:{}) :
        ∀ v :: Vertices : IC(v) ∧ (v ≠ x) ⇒
            ∃ n :: Vertices : (edge(n,v)) ∧ lvert(v) = combine(ledge(n,v), lvert(n))).
```

The total cost $c_M(f)$ *of a matching* f *is given by* $\sum_{m \in M} c_m(f)$*, while the total cost* $c_W(f)$ *is given by* $\sum_{w \in W} c_w(f)$*.*

Traditionally, the stable matching f with the least cost $c_M(f)$ is called the *optimal stable matching*. However, one can optimize stable matchings with respect to other cost metrics such as the *egalitarian cost* $c_M(f) + c_W(f)$, the *sex-equal cost* $|c_M(f) - c_W(f)|$, or even the number of unmatched m's.

Over the years, these different variations have been shown to model many real world problems, from college admissions [57] over economic applications such as auctioning [12] to the medical problem of kidney exchange [122].

The traditional, simple problem without ties or incomplete matchings can be solved efficiently. Likewise, finding the optimal matching w.r.t. the penalty for M (or, by symmetry, W) is polynomial. However, optimizing the *sex-equal* cost ($|c_M(f) - c_W(f)|$) is NP-hard [106].

In **Listing** 7.2, we model the optimal stable matching problem for sex-equal cost. Note that this specification assumes the addition of a `SUM` and `COUNT` aggregate to the language. Total preference orders are represented ternary predicates `Pm` and `Pw`. The atom `P(m,w1,w2)` should be read as '`m` prefers `w1` to `w2`'.

Lines 11–27 define a parametrized theory expressing that `match` and `imatch` represent a stable matching with sex-equal cost `cost` for preference orders `Pm` and `Pw`.

On **Line** 29, we express that `match` and `imatch` must be a stable matching for `Pm` and `Pw` with cost `cost`. Furthermore, we express that every alternative matching for the same preference orderings has a cost equal or higher than `cost` using a variant world quantification. In **Chapter** 6, we have shown that such variant world quantification corresponds to a set of universal second-order quantifications.

Listing 7.2: Specification of the *sex-equal optimal stable matching* problem.

```
 1 type Men.
 2 type Women.
 3 type Cost as int.

 5 Pm     :: (M,W,W).
 6 Pw     :: (W,M,M).
 7 match  :: M → W.
 8 imatch :: W → M.
 9 cost   :: Cost.

11 theory Stable {
12   type M.
13   type W.
```

```
14  type Cost as int.

16  Pm     :: (M,W,W).
17  Pw     :: (W,M,M).
18  match  :: M → W.
19  imatch :: W → M.
20  cost   :: Cost.

22  ∀ m :: M : ∀ w :: W : match(m)=w ⇔ imatch(w)=m. //match and imatch are eachothers
       inverse.

24  ∀ m :: M : ∀ w :: W : (¬Pm(m,w,match(m)) ∨ ¬Pw(w,m, imatch(w))).

26  ∃ c :: Cost : c = SUM{m :: M : t : COUNT{w :: W : Pm(m,w,match(m))}} - SUM{w :: W : t :
       COUNT{m :: M : Pw(w,m,imatch(w))}} ∧ (cost = c ∨ (cost = -c ∧ c < 0)).
27 }

29 *Stable(Pm,Pw,match,imatch,cost) ∧ □Stable(Pm:{(m,w,w2) | t}, Pw:{(w,m,m2) | t}, match:{},
       imatch:{}, cost:{}) : cost >= cost↑.
```

7.3 Determining Path Vapnik-Chervonenkis Dimension

The Vapnik-Chervonenkis Dimension [138, 94] (VC-dimension) is a complexity measure originating from analysis of statistical functions. It is of use in, for example, Learning theory [14] and, when specialized to graphs, computational geometry and network theory, where it is linked to the number of nodes needed to be monitored to detect failures such as network splits [90, 127].

Definition 57 (Path Vapnik-Chervonenkis Dimension (for graphs [94])). *Given an undirected graph $\mathcal{G}$, a subset X of the vertices of $\mathcal{G}$ is* path-shattered *iff for every subset $Y \subseteq X$ of at least size 2, it holds that there is a sequence of edges from $\mathcal{G}$ joining distinct vertices (called a* path*) that contains every vertex $z \in Y$ but contains* no *$z' \in X \setminus Y$.*

The cardinality of the largest path-shattered *subset X is called the path VC-dimension of $\mathcal{G}$.*

The above definition of Path VC-Dimension is generalized into P VC-dimensions by considering P-shattering where P can be *tree*s, *clique*s, etc. We focus on the VC-Dimension induced by paths as deciding whether the path VC-dimension of a graph $\mathcal{G}$ is higher than k is Σ_3^P-complete [125]. Other inducing structures such as trees or neighbourhoods allow for simplifications, thereby lowering the complexity.

As an example of path VC-dimension, we discuss the path VC-dimension when the graph $\mathcal{G}$ is a tree of size > 2: for these graphs, the path VC-dimension is known to be two [94].

Example 5. *Given a tree G of size > 2, consider any subset X consisting of at least three vertices V_1, V_2 and V_3. We will show that X cannot be path-shattered.*

*Suppose that X is path-shattered, then by **Definition** 57, there must exist a path containing V_1, V_2, and V_3. Without loss of generality (by renaming and trimming), we obtain a path from V_1 to V_3 going through V_2. As paths in a tree are unique, this is the* only *path from V_1 to V_3.*

*By **Definition** 57, for $Y = \{V_1, V_3\}$, there must exist a path that contains V_1 and V_3 but does not contain any vertex from $X \setminus \{V_1, V_3\}$. However, every path containing V_1 and V_3 must extend the unique path from V_1 to V_3. This unique path contains V_2, and $V_2 \in X \setminus \{V_1, V_3\}$, leading to contradiction. Therefore, X cannot be path-shattered.*

It is trivial to see that there exists a set X of size two that is path-shattered, thus the path VC-dimension of any tree is two.

Listing 7.3 shows a specification of path-shattered sets. The specification defines a parametrized theory `PathIn` (**Lines** 6–21) with an (undirected) graph `graph`, a path `subpath` and a unary predicate `onPath` as its arguments. The predicate `onPath` provides a convenient way to refer to the set of all vertices contained in the path.

Using the parametrized theorie `PathIn`, **Lines** 23–30 specify that for every subset `Y` of `VC_Set` that has a size of at least two, there exists a path that contains all vertices of `Y` but none of `VC_Set` $\setminus$ `Y`.

Listing 7.3: Specification of path-shattered sets..

```
1  type Vertices.

3  G_Edge :: (Vertices, Vertices).
4  VC_Set :: (Vertices).

6  theory PathIn {
7    type Vertices.

9    graph   :: (Vertices, Vertices).
10   subpath :: (Vertices, Vertices).
11   onPath  :: (Vertices).

13   ∀ x :: Vertices : onPath(x) ⇔ ∃ y :: Vertices : subpath(x,y) ∨ subpath(y,x).
14   ∀ x, y :: Vertices : subpath(x,y) ⇒ graph(x,y) ∨ graph(y,x). // Subpath is a subset of
         the edges of graph
15   ∀ x, y :: Vertices : subpath(x,y) ⇒ ¬subpath(y,x). // No edge reuse.
16   ∀ x, y, z :: Vertices : subpath(x,y) ∧ subpath(x,z) ⇒ y=z. // All vertices have at
       most one predecessor
17   ∀ x, y, z :: Vertices : subpath(y,x) ∧ subpath(z,x) ⇒ y=z. // All vertices have at
       most one successor
```

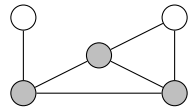


Figure 7.1: An example of a graph in which the shaded vertices form a *secure set*, while the unshaded vertices do not.

```
18  ∀ x, y  :: Vertices : (onPath(x) ∧ onPath(y) ∧ ∀ z :: Vertices : ¬subpath(z,x) ∧ ¬
       subpath(z,y)) ⇒ x=y. // Only one vertex in the path has no predecessor (i.e., the
       start vertex).
19  ∀ x, y  :: Vertices : (onPath(x) ∧ onPath(y) ∧ ∀ z :: Vertices : ¬subpath(x,z) ∧ ¬
       subpath(y,z)) ⇒ x=y. // Only one vertex in the path has no successor (i.e., the end
        vertex).
20  ∃ earlier :: (Vertices, Vertices) : ∀ x, y :: Vertices : (subpath(x,y) ⇒ (earlier(x,y)
       ∧ ∀ z :: Vertices : earlier(y,z) ⇒ earlier(x,z))) ∧ (¬earlier(x,y) ∨ ¬earlier(y,
       x)). // There exists an ordering respected by the edge sequences; no cycles can
       exist.
21 }

23 ∀ Y :: (Vertices) :
24   ((∀x :: Vertices : Y(x)⇒VC_Set(x)) ∧ (∃ x, y :: Vertices : Y(x) ∧ Y(y) ∧ x≠y)) ⇒
25       ∃ subpath :: (Vertices, Vertices) :
26       ∃ onPath  :: (Vertices) :
27         *PathIn(G_Edge, subpath, onPath) ∧
28         (∀ x :: Vertices :
29           (Y(x) ⇒ onPath(x)) ∧
30           ((VC_Set(x) ∧ ¬Y(x)) ⇒ ¬onPath(x))).
```

7.4 Secure Sets

The concept of *secure sets* in undirected graphs, introduced by Brigham et al. [22], builds upon the earlier concept of *defensive alliances* [96], i.e., any set of vertices such that for every vertex x in the alliance, the majority of its neighbours are also in the alliance. Applications include clustering in data mining [74], specific examples of which include modeling of political or economical landscapes, or finding (online) communities [54].

We follow a characterization of *secure sets* of Bliem and Woltran [13] in terms of *closed neighbourhoods*, i.e., the closed neighbourhood $N[Y]$ of a set Y consists of Y together with all vertices *adjacent* to Y.

Definition 58 (Secure Sets [13]). *Given an undirected graph $\mathcal{G}$, a subset SS of its vertices is a* secure set *if for every subset $Y \subseteq SS$, the comparison $|N[Y] \cap SS| \geq |N[Y] \setminus SS|$ holds.*

A careful reader will note the similarity between *secure sets* and the *Vapnik-Chervonenkis* dimension for *neighbourhoods*.

Example 6. *It is easy to see that in* ***Figure*** *7.1, the shaded vertices form a secure set. The unshaded vertices, however, do not.*

The problem of determining whether a *secure set* exists of size $|SS| \leq k$ is Σ_2^P-complete [13].

In **Listing** 7.4, we have modeled secure sets. Note that this specification assumes the addition of a `COUNT` aggregate to the language. **Line** 6 contains a second-order universal quantification.

Listing 7.4: Specification of the *Secure Sets* problem.

```
1 type Vertices.

3 Edge :: (Vertices, Vertices).
4 SS   :: (Vertices).

6 ∀ Y :: (Vertices) : (∀x :: Vertices : Y(x) ⇒ SS(x)) ⇒
7     COUNT{x :: Vertices : SS(x) ∧  (Y(x) ∨ (∃ y :: Vertices : Y(y) ∧ (Edge(y,x) ∨ Edge(x
       ,y))))} >=
8     COUNT{x :: Vertices : ¬SS(x) ∧ (∃ y :: Vertices : Y(y) ∧ (Edge(y,x) ∨ Edge(x,y)))}.

10 /*
11 Note that (Y(x) ∨ (∃ y :: Vertices : Y(y) ∧ (Edge(y,x) ∨ Edge(x,y)))) describes the
12 closed neighborhood of Y. In the second aggregate, we can drop Y(x) from this expression
13 as ∀ x : Y(x) ⇒ SS(x).
14 */
```

Note that a naive encoding could also include an existential quantifier over *neighbourhoods*. Instead, our encoding expresses the constraint involving neighbourhoods in terms of the subset X (cfr. to the earlier statement that the additional structure of neighbourhoods reduces the complexity of determining neighbourhood VC-dimension compared to that of path VC-dimension).

7.5 Conclusion

In this chapter, we provided a selection of interesting problems in whose modelings second-order constraints naturally arise. These problems and their models serve as a motivation to support second-order logic.

While these modelings can be written without using the language constructs introduced in **Chapter** 6, we believe that they are easier to represent when making use of language constructs such as parametrized theories and variant world quantifications. As a result, these motivational examples are included only after these language constructs have been introduced, allowing us to greatly simplify their representation.

Chapter 8

Conclusion

In this thesis, we set out to improve the level of abstraction possible in knowledge specification languages. Specifically, we wanted to advance both *computational expressivity* as well as *practical expressivity* of specification languages; the first focusses on breadth, complexity-wise, of problems that are expressible in a specification language while the other considers the ease of expression.

Both are important; for *practical expressivity* simply consider how tiresome it can be to specify knowledge without using functions, even though functionality can easily be encoded. For computational expressivity, one can consider the set of example problems of **Chapter** 7, which shows a selection of interesting problems with complexities starting on the second level of the Polynomial Hierarchy and higher.

We started in **Chapter** 3 with a Knowledge Representation based analysis of the *graph mining* problem, wanting to make clear the merits of *second-order* and *higher-order logic*. We argued that the problem, as a natural variant of *frequent pattern mining*, inherently requires expressivity on a level above NP as *graphs* represent structures more complex than single items or sets of items. Specifically, expressing the (lack of) matchings with examples requires a language capable of expressing Σ_2^P constraints, such as *second-order* logic or Answer Set Programming (ASP), albeit through the very opaque *saturation* technique.

Furthermore, the chapter discussed how the problem would benefit from a support for *higher-order*, representation wise. It detailed how these higher-order elements of the problem can instead be encoded by introducing an identity through a set of *identifiers* and resorting to *tagged unions*, reminiscent to

the technique of *defunctionalization* in systems based on lambda calculus. A takeaway observation can be that higher-order concepts, when they occur in limited sized sets, can be translated away. Finally, an ad-hoc experiment was set up to test the hypothesis that through its added expressivity, even though it can be translated away, a specification that defers such a translation to the solving system, can be beneficial to solving performance.

In **Chapter** 4, we introduced a *typed second-order* language and the SOGrounder system that grounds this language to *Quantified Boolean Formulas* (QBF). We detailed the underlying implementation, translating second-order logic into both common input languages for such solvers, QDimacs and QCIR. Experiments show that the approach of grounding to QBF promises *performance* that can compete with Answer Set Programming (ASP) solvers, without the need for complex and nontransparent encoding techniques such as *saturation*. Continuing research could investigate how to port additional techniques such as ground with bounds [143] to a system supporting *second-order* logic.

By allowing output in both QDimacs format, which requires a conjunctive normal form, and QCIR format from the same specification, we also contribute to QBF solver research with a tool to easily specify problems and study the impact of *tseitinization*.

In **Chapter** 5, we summarized work performed earlier in collaboration with I. Dasseville [35], introducing a formal semantics for templates in specification languages and logic programming by linking them to *second-order inductive definitions*. Furthermore it discussed a restriction to templates which do not increase expressivity with respect to *existential second-order*. **Chapter** 6 built on this work by starting from the more expressive *second-order* logic as supported by the SOGrounder from **Chapter** 4 and the observation of a recurrent *pattern* in second-order logic specifications: a dependency on well-known concepts with their own complexity, often with some self-similarity.

To support this pattern, we proposed new language constructs, essentially suggesting an alternative view on templates. Specifically, we proposed *parametrized theories*, *variant world quantifications* and *common core expressions*. Having defined their syntax and semantics, we discussed their expressivity and detailed our implementation translating them to standard *second-order* logic specifications fit for the SOGrounder of **Chapter** 4. We note that templates for second-order logic allow the integration of many well-known inferences in the language, in a way that allows continued reasoning on their results.

In **Chapter** 7 we offered a much-needed overview of naturally occurring problems on higher levels of the Polynomial Hierarchy, which hopefully convinces

anyone skeptical of increasing the expressivity of specification languages that plenty of problems are located on higher levels of the hierarchy that are of interest to users and designers of specification languages. These problems served as a motivation and inspiration to build a specification language supporting second-order logic, and now serve as an illustration of the concepts introduced in **Chapter** 6.

8.1 Future Work

We identify some open lines of future research on the ideas presented in this text.

Looking at the problems introduced in **Chapter** 3 and **Chapter** 4, we saw two problems, *graph mining* and *strategic companies*, where a good problem specification used not only *second-order* logic but also higher-order logic aspects, specifically in the representation of example graphs and controlling sets. While the *tagged union* can serve to *encode* the problem, it remains only an encoding and, as argumented in **Chapter** 3, it is prone to mistakes. A general approach for such specifications by performing the necessary rewrites is needed. However, such an approach is far from trivial. Open challenges with a system based on rewriting using the tagged union approach are:

- Introducing the correct number of tags when higher-order predicates are *searched* instead of given as input. Extending the domain with tags for every possible argument predicate leads to very large domains.
- Distinguishing between higher-order specifications that use *intentional* semantics (while meta-programming, for example) versus *extensional* semantics.

The SOGrounder introduced in **Chapter** 4 showed promising results on the *strategic companies* problem, w.r.t. ASP solvers. However, as mentioned in **Chapter** 4, extending the benchmark set is an important next step towards investigating the competitiveness of the approach taken; furthermore, only by starting from an expanded benchmark set can one accurately judge the effects of porting existing optimizations such as *Ground-With-Bounds* or *Lazy Grounding* in grounding techniques from first-order logic systems to systems supporting second-order logic.

Furthermore, as short-term future work, an implementation of aggregates is in order so that the SOGrounder introduced in **Chapter** 4 can be tested on the *graph mining* specification of **Chapter** 3.

In **Chapter** 6, a study of the performance overhead of *parametrized theories* and possible optimizations is undoubtedly of further interest.

Furthermore, we imposed the restriction of a stratification on *parametrized theories*, prohibiting self- and mutual-recursion. Without any restrictions, it is clear that the rewriting procedure underlying the implementation would not necessarily terminate. It would, however, definitely be interesting to allow at least some form of recursion; parametrized theories that support this would allow the modeling of problems where the number of second-order quantifier alternations is dependent on the input. Specifically, many specifications of general games require such features, as explored in a recent workshop paper [5].

A final possible extension on the work of *parametrized theories* lies in writing generally applicable templates. Consider, for example, the parametrized theory $\mathcal{T}_{\leq_{p,3V}}$ introduced in the Pasta Puzzle of **Chapter** 6. While such a template can be useful regardless of the arity of the predicates involved (as long as they have the same arity), this specific template is fixed to arity 2. It is an open question whether many *parametrized theories* exist that share the same problem, but the existence of *variadic* templates in imperative languages such as C++ suggests that the question at least warrants further research.

Bibliography

[1] Abramson, H., and Rogers, H. *Meta-programming in Logic Programming*. MIT Press, 1989.

[2] Abrial, J.-R. *The B-Book*. Cambridge University Press, 1996.

[3] Abrial, J.-R. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[4] Aoga, J. O. R., Guns, T., and Schaus, P. An efficient algorithm for mining frequent sequence with constraint programming. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2016, Riva del Garda, Italy, September 19-23, 2016, Proceedings, Part II* (2016), P. Frasconi, N. Landwehr, G. Manco, and J. Vreeken, Eds., vol. 9852 of *Lecture Notes in Computer Science*, Springer, pp. 315–330.

[5] Arteche, N., and van der Hallen, M. A formal language for QBF family definitions. In *Proceedings of the International Workshop on Quantified Boolean Formulas and Beyond* (2020 (accepted)).

[6] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., Eds. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[7] Babai, L. Graph isomorphism in quasipolynomial time. *CoRR abs/1512.03547* (2015).

[8] Baral, C., Dzifcak, J., and Takahashi, H. Macros, macro calls and use of ensembles in modular answer set programming. In *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings* (2006), pp. 376–390.

[9] Baudier, A., Fages, F., and Soliman, S. Graphical requirements for multistationarity in reaction networks and their verification in biomodels. *CoRR abs/1809.08891* (2018).

[10] Beyersdorff, O., Chew, L., and Janota, M. Extension variables in QBF resolution. In *Beyond NP, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 12, 2016* (2016).

[11] Biard, T., Mauff, A. L., Bigand, M., and Bourey, J. P. Separation of decision modeling from business process modeling using new "decision model and notation" (DMN) for automating operational decision-making. In *Risks and Resilience of Collaborative Networks - 16th IFIP WG 5.5 Working Conference on Virtual Enterprises, PRO-VE 2015, Albi, France, October 5-7, 2015, Proceedings* (2015), pp. 489–496.

[12] Bichler, M. *Market Design.* Cambridge University Press, 2017.

[13] Bliem, B., and Woltran, S. Complexity of secure sets. *Algorithmica 80*, 10 (2018), 2909–2940.

[14] Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. K. Learnability and the vapnik-chervonenkis dimension. *J. ACM 36*, 4 (1989), 929–965.

[15] Bodenreider, O. The unified medical language system (umls): integrating biomedical terminology. *Nucleic Acids Research 32*, 1 (01 2004), D267–D270.

[16] Bogaerts, B., Gamba, E., Claes, J., and Guns, T. Step-wise explanations of constraint satisfaction problems. In *Proceedings of the Twenty-fourth European Conference on Artificial Intelligence, European Conference on Artificial Intelligence, 2020 (accepted for publication)* (2020).

[17] Bogaerts, B., Janhunen, T., and Tasharrofi, S. Solving QBF instances with nested SAT solvers. In *Beyond NP, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 12, 2016.* (2016), A. Darwiche, Ed., vol. WS-16-05 of *AAAI Workshops*, AAAI Press.

[18] Bogaerts, B., Jansen, J., Bruynooghe, M., De Cat, B., Vennekens, J., and Denecker, M. Simulating dynamic systems using linear time calculus theories. *Theory Pract. Log. Program. 14*, 4-5 (2014), 477–492.

[19] Bonatti, P. A. Abduction, ASP and open logic programs. In *9th International Workshop on Non-Monotonic Reasoning (NMR 2002), April 19-21, Toulouse, France, Proceedings* (2002), pp. 184–190.

[20] Bowen, J. P. *Formal Specification and Documentation using Z.* International Thomson Computer Press, 1996.

[21] Brewka, G., Delgrande, J. P., Romero, J., and Schaub, T. asprin: Customizing answer set preferences without a headache. In *AAAI* (2015), AAAI Press, pp. 1467–1474.

[22] Brigham, R. C., Dutton, R. D., and Hedetniemi, S. T. Security in graphs. *Discret. Appl. Math. 155*, 13 (2007), 1708–1714.

[23] Bruynooghe, M., Blockeel, H., Bogaerts, B., De Cat, B., De Pooter, S., Jansen, J., Labarre, A., Ramon, J., Denecker, M., and Verwer, S. Predicate logic as a modeling language: modeling and solving some machine learning and data mining problems with *IDP3*. *Theory and Practice of Logic Programming (TPLP) 15*, 6 (2015), 783–817.

[24] Cadoli, M., Eiter, T., and Gottlob, G. Default logic as a query language. *IEEE Trans. Knowl. Data Eng. 9*, 3 (1997), 448–463.

[25] Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F., and Schaub, T. Asp-core-2 input language format. *Theory Pract. Log. Program. 20*, 2 (2020), 294–309.

[26] Charalambidis, A., Rondogiannis, P., and Symeonidou, I. Approximation fixpoint theory and the well-founded semantics of higher-order logic programs. *Theory Pract. Log. Program. 18*, 3-4 (2018), 421–437.

[27] Chen, W., Kifer, M., and Warren, D. S. Hilog: A foundation for higher-order logic programming. *The Journal of Logic Programming 15*, 3 (1993), 187–230.

[28] Claes, J., Bogaerts, B., Canoy, R., Gamba, E., and Guns, T. Zebratutor: Explaining how to solve logic grid puzzles. In *Proceedings of the 31st Benelux Conference on Artificial Intelligence (BNAIC 2019) and the 28th Belgian Dutch Conference on Machine Learning (Benelearn 2019), Brussels, Belgium, November 6-8, 2019* (2019).

[29] Clark, K. L. Negation as failure. In *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977* (1977), pp. 293–322.

[30] Clarke, E. M., Emerson, E. A., and Sistla, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst. 8*, 2 (1986), 244–263.

[31] Clercq, S. D., Schockaert, S., Cock, M. D., and Nowé, A. Solving stable matching problems using answer set programming. *Theory Pract. Log. Program. 16*, 3 (2016), 247–268.

[32] Cuteri, B., Dodaro, C., Ricca, F., and Schüller, P. Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis. *Theory and Practice of Logic Programming (TPLP) 17*, 5-6 (2017), 780–799.

[33] Dantsin, E., Eiter, T., Gottlob, G., and Voronkov, A. Complexity and expressive power of logic programming. *ACM Comput. Surv. 33*, 3 (2001), 374–425.

[34] Dao-Tran, M., Eiter, T., Fink, M., and Krennwallner, T. Modular nonmonotonic logic programming revisited. In *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings* (2009), pp. 145–159.

[35] Dasseville, I., van der Hallen, M., Janssens, G., and Denecker, M. Semantics of templates in a compositional framework for building logics. *Theory and Practice of Logic Programming (TPLP) 15*, 4-5 (2015), 681–695.

[36] De Cat, B., Bogaerts, B., Bruynooghe, M., Janssens, G., and Denecker, M. Predicate logic as a modeling language: the IDP system. In *Declarative Logic Programming: Theory, Systems, and Applications*. 2018, pp. 279–323.

[37] De Cat, B., Bogaerts, B., Devriendt, J., and Denecker, M. Model expansion in the presence of function symbols using constraint programming. In *25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2013, Herndon, VA, USA, November 4-6, 2013* (2013), pp. 1068–1075.

[38] De Cat, B., Denecker, M., Bruynooghe, M., and Stuckey, P. J. Lazy model expansion: Interleaving grounding with search. *J. Artif. Intell. Res. 52* (2015), 235–286.

[39] De Raedt, L., Guns, T., and Nijssen, S. Constraint programming for itemset mining. In *ACM SIGKDD* (2008), pp. 204–212.

[40] Demaine, E. D. Playing games with algorithms: Algorithmic combinatorial game theory. In *Mathematical Foundations of Computer Science 2001, 26th International Symposium, MFCS 2001 Marianske Lazne, Czech Republic, August 27-31, 2001, Proceedings* (2001), pp. 18–32.

[41] Denecker, M., Bruynooghe, M., and Vennekens, J. Approximation fixpoint theory and the semantics of logic and answers set programs. In *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz* (2012), pp. 178–194.

[42] Denecker, M., and Vennekens, J. The well-founded semantics is the principle of inductive definition, revisited. Chitta, Baral, pp. 1–10.

[43] Denecker, M., and Vennekens, J. The well-founded semantics is the principle of inductive definition, revisited. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014* (2014).

[44] Eiter, T., Fink, M., Ianni, G., Krennwallner, T., Redl, C., and Schüller, P. A model building framework for answer set programming with external computations. *Theory and Practice of Logic Programming (TPLP) 16*, 4 (2016), 418–464.

[45] Eiter, T., and Gottlob, G. On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell. 15*, 3-4 (1995), 289–323.

[46] Eiter, T., Ianni, G., and Krennwallner, T. Answer set programming: A primer. In *Reasoning Web* (2009), vol. 5689 of *Lecture Notes in Computer Science*, Springer, pp. 40–110.

[47] Eiter, T., Krennwallner, T., and Redl, C. Hex-programs with nested program calls. In *Applications of Declarative Programming and Knowledge Management - 19th International Conference, INAP 2011, and 25th Workshop on Logic Programming, WLP 2011, Vienna, Austria, September 28-30, 2011, Revised Selected Papers* (2011), pp. 269–278.

[48] Eiter, T., and Polleres, A. Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. *Theory Pract. Log. Program. 6*, 1-2 (2006), 23–60.

[49] Elseidy, M., Abdelhamid, E., Skiadopoulos, S., and Kalnis, P. Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow. 7*, 7 (Mar. 2014), 517–528.

[50] Etherington, D. W. Relating default logic and circumscription. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence. Milan, Italy, August 23-28, 1987* (1987), J. P. McDermott, Ed., Morgan Kaufmann, pp. 489–494.

[51] Etherington, D. W., and Crawford, J. M. Toward efficient default reasoning. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 1* (1996), pp. 627–632.

[52] Fagin, R. Generalized first-order spectra, and polynomial-time recognizable sets. 43–73.

[53] Farmer, W. M. The seven virtues of simple type theory. *J. Appl. Log. 6*, 3 (2008), 267–286.

[54] Flake, G. W., Lawrence, S., Giles, C. L., and Coetzee, F. Self-organization and identification of web communities. *IEEE Computer 35*, 3 (2002), 66–71.

[55] Fraenkel, A. S., and Lichtenstein, D. Computing a perfect strategy for n*n chess requires time exponential in n.

[56] Frisch, A. M., Harvey, W., Jefferson, C., Hernández, B. M., and Miguel, I. Essence : A constraint language for specifying combinatorial problems. *Constraints An Int. J. 13*, 3 (2008), 268–306.

[57] Gale, D., and Shapley, L. S. College admissions and the stability of marriage. *The American Mathematical Monthly 69*, 1 (1962), 9–15.

[58] Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012.

[59] Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. Clingo = ASP + control: Preliminary report. *CoRR abs/1405.3694* (2014).

[60] Gebser, M., Kaminski, R., and Schaub, T. Complex optimization in answer set programming. *Theory Pract. Log. Program. 11*, 4-5 (2011), 821–839.

[61] Gebser, M., Kaufmann, B., and Schaub, T. Solution enumeration for projected boolean search problems. In *Constraint Programming, Artificial Intelligence and Operations Research (CPAIOR)* (2009), vol. 5547 of *Lecture Notes in Computer Science*, Springer, pp. 71–86.

[62] Gebser, M., Obermeier, P., Otto, T., Schaub, T., Sabuncu, O., Nguyen, V., and Son, T. C. Experimenting with robotic intra-logistics domains. *Theory Pract. Log. Program. 18*, 3-4 (2018), 502–519.

[63] Gebser, M., Schaub, T., Thiele, S., and Veber, P. Detecting inconsistencies in large biological networks with answer set programming. *Theory Pract. Log. Program. 11*, 2-3 (2011), 323–360.

[64] Gelfond, M., and Lifschitz, V. The stable model semantics for logic programming. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)* (1988), pp. 1070–1080.

[65] Gelfond, M., and Przymusinska, H. Towards a theory of elaboration tolerance: Logic programming approach. *International Journal of Software Engineering and Knowledge Engineering 6*, 1 (1996), 89–112.

[66] Giunchiglia, E., Marin, P., and Narizzano, M. Reasoning with quantified boolean formulas. In *Handbook of Satisfiability.* 2009, pp. 761–780.

[67] Giunchiglia, E., Narizzano, M., Pulina, L., and Tacchella, A. Qcir-g14: A non-prenex non-cnf format for quantified boolean formulas. Accessed: 2020-08-03.

[68] Guyet, T., Moinard, Y., Quiniou, R., and Schaub, T. Efficiency analysis of ASP encodings for sequential pattern mining tasks. In *Advances in Knowledge Discovery and Management - Volume 7 [Best of EGC 2016,Reims, France]* (2016), B. Pinaud, F. Guillet, B. Crémilleux, and C. de Runz, Eds., vol. 732 of *Studies in Computational Intelligence*, Springer, pp. 41–81.

[69] Hall, N., and Paul, L. *Causation: A User's Guide.* Oxford University Press, Oxford, 2013.

[70] Halpern, J. Y. A modification of the halpern-pearl definition of causality. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence* (2015), AAAI Press, pp. 3022–3033.

[71] Halpern, J. Y., and Pearl, J. Causes and explanations: A structural-model approach: Part 1: Causes. In *UAI '01: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence, University of Washington, Seattle, Washington, USA, August 2-5, 2001* (2001), J. S. Breese and D. Koller, Eds., Morgan Kaufmann, pp. 194–202.

[72] Harvey, W. CSPLib problem 010: Social golfers problem. `http://www.csplib.org/Problems/prob010`.

[73] Hassan, M., Coulet, A., and Toussaint, Y. Learning subgraph patterns from text for extracting disease - symptom relationships. In

Proceedings of the 1st International Workshop on Interactions between Data Mining and Natural Language Processing co-located with The European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases, DMNLP@PKDD/ECML 2014, Nancy, France, September 15, 2014 (2014), P. Cellier, T. Charnois, A. Hotho, S. Matwin, M. Moens, and Y. Toussaint, Eds., vol. 1202 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 81–96.

[74] Hassan-Shafique, K. *Partitioning A Graph In Alliances And Its Application To Data Clustering.* PhD thesis, University of Central Florida, Orlando, USA, 2004.

[75] Hou, P., Wittocx, J., and Denecker, M. A deductive system for PC(ID). In *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings* (2007), pp. 162–174.

[76] Ianni, G., Ielpa, G., Pietramala, A., Santoro, M. C., and Calimeri, F. Enhancing answer set programming with templates. In *10th International Workshop on Non-Monotonic Reasoning (NMR 2004), Whistler, Canada, June 6-8, 2004, Proceedings* (2004), pp. 233–239.

[77] Ianni, G., Krennwallner, T., and Calimeri, F. Asp competition 2013.

[78] Immerman, N. Languages that capture complexity classes. *SIAM J. Comput. 16*, 4 (1987), 760–778.

[79] Immerman, N. Descriptive complexity and model checking. In *Foundations of Software Technology and Theoretical Computer Science, 18th Conference, Chennai, India, December 17-19, 1998, Proceedings* (1998), V. Arvind and R. Ramanujam, Eds., vol. 1530 of *Lecture Notes in Computer Science*, Springer, pp. 1–5.

[80] Immerman, N. *Descriptive complexity.* Graduate texts in computer science. Springer, 1999.

[81] Inokuchi, A., Washio, T., and Motoda, H. An apriori-based algorithm for mining frequent substructures from graph data. vol. 1910, pp. 13–23.

[82] Inokuchi, A., Washio, T., Okada, T., and Motoda, H. Applying the apriori-based graph mining method to mutagenesis data analysis. *Journal of Computer Aided Chemistry 2* (01 2001).

[83] Järvisalo, M. Itemset mining as a challenge application for answer set enumeration. Logic Programming and Nonmonotonic Reasoning (LPNMR), pp. 304–310.

[84] Jordan, C., Klieber, W., and Seidl, M. Non-cnf QBF solving with QCIR. In *Beyond NP, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 12, 2016* (2016).

[85] Kakas, A. C., Van Nuffelen, B., and Denecker, M. A-system: Problem solving through abduction. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001* (2001), pp. 591–596.

[86] Kalviainen, H., and Oja, E. Comparisons of attributed graph matching algorithms for computer vision. In *In Proc. of STEP-90, Finnish Artificial Intelligence Symposium* (1990), pp. 354–368.

[87] Kaufmann, B., Leone, N., Perri, S., and Schaub, T. Grounding and solving in answer set programming. *AI Magazine 37*, 3 (2016), 25–32.

[88] Keller, S., Miettinen, P., and Kalinina, O. V. Frequent subgraph mining for biologically meaningful structural motifs. *bioRxiv* (2020).

[89] Kemmar, A., Lebbah, Y., Loudni, S., Boizumault, P., and Charnois, T. Prefix-projection global constraint and top-k approach for sequential pattern mining. *Constraints 22*, 2 (2017), 265–306.

[90] Kleinberg, J. Detecting a network failure. In *Proceedings 41st Annual Symposium on Foundations of Computer Science* (2000), pp. 231–239.

[91] Klieber, W. Ghostq. *J. Satisf. Boolean Model. Comput. 11*, 1 (2019), 65–72.

[92] Klieber, W., Janota, M., Marques-Silva, J., and Clarke, E. M. Solving QBF with free variables. In *CP* (2013), vol. 8124 of *Lecture Notes in Computer Science*, Springer, pp. 415–431.

[93] Kowalski, R. A. Predicate logic as programming language. In *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974* (1974), pp. 569–574.

[94] Kranakis, E., Krizanc, D., Ruf, B., Urrutia, J., and Woeginger, G. The vc-dimension of set systems defined by graphs. *Discrete Applied Mathematics 77*, 3 (1997), 237 – 257.

[95] Krings, S., Leuschel, M., Körner, P., Hallerstede, S., and Hasanagic, M. Three is a crowd: Sat, SMT and CLP on a chessboard. In *Practical Aspects of Declarative Languages - 20th International Symposium, PADL 2018, Los Angeles, CA, USA, January 8-9, 2018, Proceedings* (2018), pp. 63–79.

[96] Kristiansen, P., Hedetniemi, S. M., and Hedetniemi, S. T. Alliances in graphs. *Journal of Combinatorial Mathematics and Combinatorial Computing 48* (2004), 157–177.

[97] Lamport, L. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[98] Leuschel, M., and Butler, M. J. ProB: An automated analysis toolset for the B method. *STTT 10*, 2 (2008), 185–203.

[99] Li, H., Yap, C. W., Ung, C. Y., Xue, Y., Cao, Z. W., and Chen, Y. Z. Effect of selection of molecular descriptors on the prediction of bloodbrain barrier penetrating and nonpenetrating agents by statistical learning methods. *Journal of Chemical Information and Modeling 45*, 5 (2005), 1376–1384. PMID: 16180914.

[100] Lin, F., and Reiter, R. How to progress a database. *Artif. Intell. 92*, 1-2 (1997), 131–167.

[101] Lonsing, F., and Biere, A. Depqbf: A dependency-aware QBF solver. *JSAT 7*, 2-3 (2010), 71–76.

[102] Lonsing, F., Egly, U., and Van Gelder, A. Efficient clause learning for quantified boolean formulas via QBF pseudo unit propagation. In *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings* (2013), M. Järvisalo and A. Van Gelder, Eds., vol. 7962 of *Lecture Notes in Computer Science*, Springer, pp. 100–115.

[103] Lynce, I., and Silva, J. P. M. On computing minimum unsatisfiable cores. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings* (2004).

[104] Maratea, M., Ricca, F., Faber, W., and Leone, N. Look-back techniques and heuristics in DLV: implementation, evaluation, and comparison to QBF solvers. *J. Algorithms 63*, 1-3 (2008), 70–89.

[105] McCarthy, J. Elaboration tolerance. In *Working Papers of the Fourth International Symposium on Logical formalizations of Commonsense Reasoning, Commonsense-1998* (1998).

[106] McDermid, E., and Irving, R. W. Sex-equal stable matchings: Complexity and exact algorithms. *Algorithmica 68*, 3 (2014), 545–570.

[107] Mitchell, D. G., and Ternovska, E. A framework for representing and solving NP search problems. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA* (2005), pp. 430–435.

[108] Mitchell, D. G., and Ternovska, E. Expressive power and abstraction in essence. *Constraints An Int. J. 13*, 3 (2008), 343–384.

[109] Montali, M. Putting decisions in perspective. In *Business Process Management Workshops* (Cham, 2019), C. Di Francescomarino, R. Dijkman, and U. Zdun, Eds., Springer International Publishing, pp. 355–361.

[110] Muggleton, S., and De Raedt, L. Inductive logic programming: Theory and methods. *J. Log. Program. 19/20* (1994), 629–679.

[111] Musser, D. R., and Saini, A. *STL tutorial and reference guide - C++ programming with the standard template library.* Addison-Wesley professional computing series. Addison-Wesley, 1996.

[112] Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., and Tack, G. Minizinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings* (2007), pp. 529–543.

[113] Nijssen, S., and Kok, J. N. Frequent graph mining and its application to molecular databases. In *Proceedings of the IEEE International Conference on Systems, Man & Cybernetics: The Hague, Netherlands, 10-13 October 2004* (2004), IEEE, pp. 4571–4577.

[114] Papadimitriou, C. H., and Yannakakis, M. The complexity of facets (and some facets of complexity). *J. Comput. Syst. Sci. 28*, 2 (1984), 244–259.

[115] Paramonov, S., Chen, T., and Guns, T. Generic mining of condensed pattern representations under constraints. In *CEUR: Young Scientist's Second International Workshop on Trends in Information Processing Proceedings (YSIP)* (2017), vol. 1837, pp. 138–177.

[116] Peitl, T., Slivovsky, F., and Szeider, S. Dependency learning for QBF. In *Theory and Applications of Satisfiability Testing - SAT 2017 -*

20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings (2017), S. Gaspers and T. Walsh, Eds., vol. 10491 of *Lecture Notes in Computer Science*, Springer, pp. 298–313.

[117] Plaisted, D. A., and Greenbaum, S. A structure-preserving clause form translation. *J. Symb. Comput. 2*, 3 (1986), 293–304.

[118] Pnueli, A. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977* (1977), IEEE Computer Society, pp. 46–57.

[119] Redl, C. Explaining inconsistency in answer set programs and extensions. In *Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings* (2017), M. Balduccini and T. Janhunen, Eds., vol. 10377 of *Lecture Notes in Computer Science*, Springer, pp. 176–190.

[120] Reiter, R. A logic for default reasoning. *Artif. Intell. 13*, 1-2 (1980), 81–132.

[121] Robson, J. M. Combinatorial games with exponential space complete decision problems. In *Mathematical Foundations of Computer Science 1984, Praha, Czechoslovakia, September 3-7, 1984, Proceedings* (1984), pp. 498–506.

[122] Roth, A., Sönmez, T., and Unver, U. Pairwise kidney exchange. *Journal of Economic Theory 125*, 2 (2005), 151–188.

[123] Rückert, U., and Kramer, S. Optimizing feature sets for structured data. In *Machine Learning: ECML 2007, 18th European Conference on Machine Learning, Warsaw, Poland, September 17-21, 2007, Proceedings* (2007), J. N. Kok, J. Koronacki, R. L. de Mántaras, S. Matwin, D. Mladenic, and A. Skowron, Eds., vol. 4701 of *Lecture Notes in Computer Science*, Springer, pp. 716–723.

[124] Saikko, P., Dodaro, C., Alviano, M., and Järvisalo, M. A hybrid approach to optimization in answer set programming. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018* (2018), pp. 32–41.

[125] Schäfer, M. Deciding the vapnik-cervonenkis dimension is Σ_3^{p}-complete. In *Proceedings of the Eleventh Annual IEEE Conference on Computational Complexity, Philadelphia, Pennsylvania, USA, May 24-27, 1996* (1996), pp. 77–80.

[126] Shaw, M. Abstraction techniques in modern programming languages. *IEEE Software 1*, 4 (1984), 10–26.

[127] Shrivastava, N., Suri, S., and Tóth, C. D. Detecting cuts in sensor networks. *ACM Trans. Sen. Netw. 4*, 2 (Apr. 2008).

[128] Silva, J. P. M., and Sakallah, K. A. GRASP - a new search algorithm for satisfiability. In *International Conference on Computer-Aided Design (ICCAD), San Jose, California, USA, November 10-14 1996* (1996), pp. 220–227.

[129] Stockmeyer, L. J. The polynomial-time hierarchy. *Theor. Comput. Sci. 3*, 1 (1976), 1–22.

[130] Stockmeyer, L. J., and Chandra, A. K. Provably difficult combinatorial games. *SIAM J. Comput. 8*, 2 (1979), 151–174.

[131] Tasharrofi, S., and Ternovska, E. A semantic account for modularity in multi-language modelling of search problems. In *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings* (2011), pp. 259–274.

[132] Torlak, E., and Jackson, D. Kodkod: A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings* (2007), pp. 632–647.

[133] van der Hallen, M., and Janssens, G. SOGrounder: Modelling and solving second-order logic. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018* (2018), pp. 72–77.

[134] van der Hallen, M., Paramonov, S., Janssens, G., and Denecker, M. Knowledge representation analysis of graph mining. *Ann. Math. Artif. Intell. 86*, 1-3 (2019), 21–60.

[135] van der Hallen, M., Paramonov, S., Leuschel, M., and Janssens, G. Knowledge representation analysis of graph mining. *CoRR abs/1608.08956* (2016).

[136] van Emden, M. H., and Kowalski, R. A. The semantics of predicate logic as a programming language. *J. ACM 23*, 4 (1976), 733–742.

[137] Van Hertum, P., Dasseville, I., Janssens, G., and Denecker, M. The KB paradigm and its application to interactive configuration. *Theory Pract. Log. Program. 17*, 1 (2017), 91–117.

[138] Vapnik, V. N., and Chervonenkis, A. Y. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications 16*, 2 (1971), 264–280.

[139] Vardi, M. Y. Querying logical databases. *J. Comput. Syst. Sci. 33*, 2 (1986), 142–160.

[140] Vennekens, J., Wittocx, J., Mariën, M., and Denecker, M. Predicate introduction for logics with a fixpoint semantics. part I: logic programming. *Fundam. Inform. 79*, 1-2 (2007), 187–208.

[141] Weinzierl, A. Blending lazy-grounding and CDNL search for answer-set solving. In *Logic Programming and Nonmonotonic Reasoning (LPNMR)* (2017), vol. 10377 of *Lecture Notes in Computer Science*, Springer, pp. 191–204.

[142] Wittocx, J., Denecker, M., and Bruynooghe, M. Constraint propagation for first-order logic and inductive definitions. *ACM Trans. Comput. Log. 14*, 3 (2013), 17:1–17:45.

[143] Wittocx, J., Mariën, M., and Denecker, M. Grounding FO and FO(ID) with bounds. *J. Artif. Intell. Res. 38* (2010), 223–269.

[144] Yan, X., and Han, J. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9-12 December 2002, Maebashi City, Japan* (2002), pp. 721–724.

Curriculum Vitae

Matthias Van der Hallen (Deurne, 28 september 1991), graduated secondary school, the College van het Eucharistisch Hart Essen (Greek-Latin), in 2009 and subsequently started a bachelor's in Engineering Science at KU Leuven, with major in Computer Science and a minor in Electrical Engineering. He went on to obtain a master's degree in Engineering Science: Computer Science, magna cum laude, in September 2014, with a master thesis titled "Secure Compilation of ML Modules" supervised by prof. dr. ir. Frank Piessens.

After obtaining his master's degree, Matthias joined the DTAI (Declarative Languages and Artificial Intelligence) research group at KU Leuven to persue his PhD under supervision of prof. dr. ir. Gerda Janssens. Starting January 1, 2015 he was granted a four year PhD fellowship strategic basic research by the Research Foundation - Flanders.

List of publications

Journal Articles

- Dasseville, I., van der Hallen, M., Janssens, G., and Denecker M. "Semantics of Templates in a Compositional Framework for Building Logics." in Theory and Practice of Logic Programming, vol 15, 4-5 (2015), pp 681–695.
- van der Hallen, M., Paramonov, S., Janssens, G., and Denecker, M. "Knowledge Representation Analysis of Graph Mining." in Annals of Mathematics and Artificial Intelligence, vol 86, 1-3 (2019), pp 21 –60.

Peer-reviewed Articles at Conferences

- Dasseville, I., van der Hallen, M., Bogaerts, B., Janssens, G., and Denecker, M. "A Compositional Typed Higher-Order Logic with Definitions." in Technical Communications of the 32nd International Conference on Logic programming, New York City, USA, 16 Oct. - 21 Oct. 2016.
- van der Hallen, M., and Janssens, G. "SOGrounder: Modelling and Solving Second-Order Logic." in Principles of Knowledge Representation and Reasoning: Proceedings of the sixteenth International Conference, 30 Oct. - 02 Nov. 2018, The AAAI Press, pp 72–77.

Peer-reviewed Articles at Workshops

- van der Hallen, M., Paramonov, S., Leuschel, M., and Janssens, G. "Knowledge Representation Analysis of Graph Mining." in Proceedings

of the Workshop on Answer Set Programming and Other Computing Paradigms, New York City, USA, 16 Oct. 2016, pp 55–76

- van der Hallen, M., and Janssens, G. "A Grounder From Second-Order Logic To QBF" in Proceedings of the International Workshop on Quantified Boolean Formulas and Beyond, Oxford, England, 8 July 2018.
- Arteche, N., and van der Hallen, M. "A Formal Language for QBF Family Definitions" in Proceedings of the International Workshop on Quantified Boolean Formulas and Beyond, 2020 (accepted).



FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DTAI
Celestijnenlaan 200A box 2402
B-3001 Leuven
matthias.vanderhallen@kuleuven.be
http://www.dtai.cs.kuleuven.be