KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

# Separating Knowledge from Computation

## An FO($\cdot$) Knowledge Base System and its Model Expansion Inference

**Broes De Cat**

Supervisor:
Prof. dr. Marc Denecker

Dissertation presented in partial
fulfillment of the requirements for the
degree of Doctor in Engineering

May 2014

# Separating Knowledge from Computation

## An FO(·) Knowledge Base System and its Model Expansion Inference

**Broes DE CAT**

Examination committee:
Prof. em. dr. ir. Hugo Hens, chair
Prof. dr. Marc Denecker, supervisor
Prof. dr. ir. Maurice Bruynooghe
Prof. dr. Dave Clarke
Prof. dr. Patrick De Causmaecker
(KU Leuven Kulak, Belgium)
Prof. dr. Michael Leuschel
(HH Universität Düsseldorf, Germany)
Prof. dr. Peter Stuckey
(University of Melbourne, Australia)

Dissertation presented in partial
fulfillment of the requirements for
the degree of Doctor
in Engineering

May 2014

Uitgegeven in eigen beheer, Broes De Cat, Celestijnenlaan 200A box 2402, B-3001 Heverlee (Belgium)

ISBN 978-94-6018-835-0

D/2014/7515/59

# Preface

> The most exciting phrase to hear in science, the one that heralds new discoveries, is not “Eureka!” (I found it!) but rather, “That’s funny…”
>
> Isaac Asimov

Finally ready to write the preface of this dissertation. It has been a great four years of research, with moments of satisfying results, interesting discussions, great new ideas and fun trips. Woven between those were also moments of writer’s block, ideas that didn’t work out and frustrating debugging. But in the end, I look back happily, and for that there are many people I would like to thank here.

First and foremost, I have to thank **Marc Denecker** for willing to be my supervisor and to provide guidance through these years. I admire his ideas, which convinced me to join his group, and his courage, for pursuing the really broad goal his ideas entail. Over the years, we had quite some heavy discussions, but in the end, I usually saw the light of what he had already known from the start. I would also like to thank **Maurice**, who took it upon himself to relieve Marc a bit by revising my writing. My research was supported by the Agency for Innovation by Science and Technology (**IWT**) and the KU Leuven.

The group I couldn’t have gone without *at* work are my colleagues. It is often said that doing a phd is a solitary business. While this may often be the case, I would never have made it to the end if I had not been part of a great team, working together with a shared vision. Thank you **Maarten, Johan, Bart, Stef, Jo, Joachim, Pieter and Ingmar**! For the collaboration, the company at conferences and the fun while playing board and computer games. Special thanks go to **Bart**, who was my office mate the longest: I will miss our discussions and I think we made quite a team, building on each other's ideas time and again! I really enjoyed working with you! Thanks also for your patience and diplomacy; I guess you know what I mean ;)

During my phd, I was able to do quite a number of interesting research visits to far-away lands and see a bit of the world while doing so. Thank you **Peter**, **Yuliya**, **Illka** and **Torsten** for giving me the opportunity to stay with you, exchange ideas and work together.

The group I couldn't have gone without *outside* work: **Chiro**! For providing the necessary distraction and fun throughout those years: bedankt aan alle oudleiding (zeker **Fille**, **Boris** en **Marijn**), gewesters, verbonders en medebegeleiding doorheen de jaren!

Thanks also to **Jan**, for being a great friend and gaming buddy. I wish you all the best of luck with your own phd!

Thanks to my **family**, for supporting me up-to-this point! Hopefully you will soon have a bit of an idea of what I have been doing all these years! I'm also looking forward to both my siblings graduating this year as well!

As the last are absolutely the best, I want to thank **Ellen**, my girlfriend. Not just for being my love and the greatest person in the world, but also for standing by me all this time, listening to my (not always interesting) ideas, destressing me at many points in time, proofreading unintelligible text and letting me see the light in the darkest hours. Dank je Ellie!

Broes

# Abstract

In the area of Artificial Intelligence, the field of *Knowledge Representation* is devoted to the study of how knowledge can be represented and how it can be used for automated reasoning. At this moment, a popular approach are *declarative programming* paradigms, which consist of developing a formal language (a *logic*), to symbolically represent knowledge, and an associated form of inference, to solve a type of computational task. Recently, the Knowledge Base System (KBS) paradigm was proposed, based on the idea that knowledge is not inherently linked to a specific reasoning task. Instead, the paradigm proposes to express knowledge in a truly declarative language and different computational tasks can then be accomplished by applying the proper inference.

In this work, we develop IDP, a knowledge base system intended to be a laboratory for the study of software engineering in the context of the KBS paradigm. The system supports a rich, declarative logic and implements a range of inference engines, enabling it to solve a broad class of computational tasks. In addition, IDP offers an interface between the KBS language and an imperative language, leading to a novel kind of tight integration between imperative and declarative languages. The declarative language is the logic $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$, an extension of First-Order Logic (FO) with aggregates, inductive definitions, partial functions and types. The aim is to provide a language in which a user can *naturally model* his applications and provide *robust* inference engines that free the user from performance considerations. Both the language and the system are designed with extensibility in mind, to allow the addition of new language constructs and new forms of inference.

After having introduced the IDP system, we study one particular inference task, namely (optimal) *model expansion*, a core task in many applications. It is the task of finding an interpretation for a set of symbols, over a known domain, that (optimally) satisfies a given logical theory over those symbols. The standard approach to model expansion consists of two phases. First, the theory is *reduced* to an equivalent, propositional (quantifier- and function-free) theory, among others by exhaustively instantiating quantified variables with elements from the domain. Afterwards, an efficient *search algorithm* is applied to search for models of the propositional theory. A long-standing issue with this (and similar) approaches is the bottleneck caused by the first phase: it blows up the size of the theory and makes the approach infeasible for many applications. We develop three techniques to improve model expansion by addressing this blowup problem. First, we show that by allowing (nested) *function terms* to occur in the intermediate theory, the size of the reduced theory becomes smaller. We then develop a search algorithm for such more general reduced theories, which works by creating the propositional theory on-demand *during* search. Second, in view of our robustness aim, we develop an automated approach to *detect functional dependencies* implicit in the input. We show that such dependencies can be exploited automatically to introduce nested function terms, using *deduction*. In combination with the first technique, this results in improved performance. Hence, this frees the user of minding about performance considerations when choosing between a predicate or a function symbol during modeling. Last, we develop a general framework for *interleaving the instantiation of quantifications with the search process*, removing the distinction between both phases. The framework is based on deriving guarantees under which parts of the (uninstantiated) theory can still be satisfied and hence do not need to be considered during search just yet. Only when the guarantees no longer hold is further instantiation required. The first technique is in fact a highly optimized version of this general framework. The result is a theoretical framework on how to interleave quantifier instantiation and search and a state-of-the-art algorithm for optimal model expansion for the logic $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$.

# Samenvatting

Binnen het domein van artificiële intelligentie spitst het veld van *kennisrepresentatie* zich toe op onderzoek naar hoe kennis voorgesteld kan worden en hoe het gebruikt kan worden om automatisch redeneertaken op te lossen. Op dit moment zijn *declaratieve programmeerparadigma's* hier een belangrijke aanpak voor. Een dergelijk paradigma bestaat uit een formele taal (een *logica*), waarin kennis symbolisch kan worden voorgesteld, en een geassocieerde vorm van inferentie om een type van computationele taak op te lossen. Recent werd het *kennisbanksysteem*(KBS) paradigma voorgesteld, gebaseerd op de idee dat kennis niet inherent verbonden is met een specifieke redeneertaak. In plaats daarvan stelt het voor om kennis uit te drukken in één declaratieve taal, zodat het oplossen van computationele taken neerkomt door het toepassen van de gepaste inferentie.

In deze thesis wordt IDP ontwikkeld, een kennisbanksysteem bedoeld als laboratorium voor de studie van software engineering in de context van het KBS paradigma. IDP ondersteunt een rijke, declaratieve logica en biedt diverse redeneermogelijkheden aan, zodat een brede klasse van computationele taken opgelost kan worden. Daarnaast beschikt IDP over een koppeling tussen de kennisbanktaal en een imperatieve taal, een nieuw type van hechte integratie tussen declaratieve en imperatieve talen. De kennisbanktaal van IDP is de logica $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$, een uitbreiding van eerste-orde logica met inductieve definities, aggregaten, partiële functies en een type systeem. Een van de doelen is om een taal aan te bieden waarin een gebruiker op een *natuurlijke* manier toepassingen kan *modelleren* en om dan *robuuste* redeneermogelijkheden aan te bieden zodat

de gebruiker zich niet hoeft in te laten met efficiëntie-overwegingen. Zowel de taal als het systeem zijn ontworpen met uitbreidbaarheid in het achterhoofd, om de toevoeging van nieuwe taalconstructies of nieuwe vormen van inferentie toe te laten.

Na de introductie van IDP onderzoeken we de inferentietaak (optimale) *modelexpansie* in detail, een basistaak voor vele toepassingen. Deze taak bestaat uit het zoeken van een interpretatie voor een set van symbolen, over een gekend domein, die een (optimaal) model is voor een gegeven logische theorie over die symbolen. De standaard aanpak van modelexpansie bestaat uit 2 fasen. Eerst wordt de theorie *gereduceerd* tot een equivalente, propositionele theorie (dus zonder kwantificaties en functies), onder andere door exhaustief gekwantificeerde variabelen te instantiëren met elementen uit het domein. Daarna wordt een efficiënt *zoekalgoritme* toegepast om modellen te zoeken van de gereduceerde theorie. Een gekend probleem met deze (en vergelijkbare) aanpakken is het feit dat de eerste fase de grootte van de theorie opblaast. Voor vele toepassingen maakt dit de standaard aanpak onbruikbaar. We ontwikkelen drie technieken om modelexpansie te verbeteren door dit probleem aan te pakken. Eerst tonen we aan dat door het toelaten van (geneste) *functietermen* in de intermediaire theorie, de grootte van die theorie sterk daalt. We ontwikkelen dan een zoekalgoritme dat kan omgaan met dergelijke, meer algemene intermediaire theorieën, gebaseerd op het creëren van de propositionele theorie op een on-demand manier *tijdens* het zoekproces. Daarna, gedreven door de gewenste robuustheid, ontwikkelen we een automatische aanpak om dergelijke functietermen te introduceren, zodat ze de gebruiker niet beperken in zijn modelleervrijheid. Door *deductie* wordt de aanwezigheid van *functionele afhankelijkheden* bewezen, die dan gebruikt worden om functietermen te introduceren; door bovenstaande aanpak resulteert dit in hogere efficiëntie. Als laatste ontwikkelen we een algemeen framework dat beide fasen *verweeft*. Dit is gebaseerd op het afleiden van voorwaarden waaronder bepaalde delen van de originele (niet-geïnstantieerde) theorie nog voldaan kunnen worden en dus nog niet beschouwd moeten worden door het zoekproces. Enkel wanneer dergelijke voorwaarden niet meer voldaan zijn, is er extra instantiatie nodig. De eerste techniek is een sterk geoptimaliseerde versie van dit algemene framework. Het resultaat is een theoretisch framework over het verweven van de instantiatie van kwantificaties met een zoekproces en een grensverleggend algoritme voor optimale modelexpansie voor $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$.

# Abbreviations

**ASP** Answer Set Programming.

**CDCL** Conflict-Driven Clause-Learning.
**CNF** Conjunctive Normal Form.
**CP** Constraint Programming.

**DCA** Domain Closure Assumption.

**ECNF** Extended Conjunctive Normal Form.

**FO** First-Order Logic.

**KBS** Knowledge Base System.
**KR** Knowledge Representation.

**LCG** Lazy Clause Generation.
**LP** Logic Programming.

**MX** Model Expansion.

**NNF** Negation Normal Form.

**PC** Propositional Calculus.

**SMT** SAT Modulo Theories.

**UNA** Unique Names Axioms.

# List of Symbols

| | |
|---|---|
| $\mathrm{FO}(\cdot)$ | A class of languages that are extensions of first-order logic, page 3 |
| $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ | The language of the IDP system, an instantiation of $\mathrm{FO}(\cdot)$ , page 3 |
| $\Sigma$ | A vocabulary, page 9 |
| $P, Q, R$ | Predicate symbols, page 9 |
| $f, g, c$ | Function symbols, $c$ is a constant, page 9 |
| $\mathcal{T}$ | A $\Sigma$-theory, page 10 |
| $a$ | An atom, page 10 |
| $l$ | A literal, page 10 |
| $d$ | A domain element, page 10 |
| $x, y$ | Variables, page 10 |
| $G_f$ | The $n+1$-ary graph predicate symbol of a function symbol $f/n$, page 12 |
| $>_p$ | The precision order, page 13 |
| $>_t$ | The truth order, page 13 |
| $\varphi, \psi$ | Formulas, page 14 |

| | |
|---|---|
| $\mathcal{I}$ | A four-valued $\Sigma$-structure, page 14 |
| $D$ | A domain, page 14 |
| $T$ | A Tseitin symbol, page 16 |
| $\sim$ | Any of the comparison operators $=$, $\neq$, $<$, $\leq$, $>$ or $\geq$, page 25 |
| $agg$ | Any aggregate function (sum, product, cardinality, minimum or maximum), page 26 |
| $\Delta$ | An inductive definition, page 26 |
| $\mathrm{def}(\Delta)$ | The defined symbols of $\Delta$ , page 27 |
| $\mathrm{open}(\Delta)$ | The open symbols of $\Delta$ , page 27 |
| $comp(P,\Delta)$ | The completion of $P$ as defined by $\Delta$ , page 27 |
| SuppF | The set of function symbols allowed in the ground theory, page 75 |
| $\mathcal{T}_m$ | The mapping theory constructed during grounding, page 75 |
| $\mathcal{T}_g$, $c_g$ | Ground theory and optimization term, page 75 |
| $\mathcal{T}_{in}$, $c_{in}$ | Input theory and optimization term, page 75 |
| **undef** | Representation of a non-denoting term, page 80 |
| $\lceil c \sim v \rceil$ | A shorthand for the order-encoding representation of $c \sim v$, page 114 |
| $\mathcal{T}_s$ | The intermediate theory during search, page 110 |
| $d\,\langle s, S, i\rangle$ | A functional dependency, where the $i$-th argument of symbol $s$ depends on index set $S$, page 143 |
| $J$ | A justification (graph), page 168 |
| $J_L$ | The restriction of a justification graph to the nodes and edges of $J$ reachable from literals in $L$, page 169 |
| $\{P_{\mathcal{T}}, \Delta\}$ | Canonical theory consisting of the atomic sentence $P_{\mathcal{T}}$ and definition $\Delta$ , page 167 |
| $\Delta_{\mathrm{g}}$, $\Delta_{\mathrm{d}}$, $\Delta_{\mathrm{gd}}$ | Respectively the ground definition, delayed definition and the union of both, page 173 |
| $sent_{\mathrm{g}}$ | Set of ground sentences, page 182 |

# Contents

# List of Figures

# List of Tables

# 1

# Introduction

The ability to allow anyone to write up any task in a natural language like English and have a computer automatically solve it, is a very attractive idea. It is also one of the ideas behind the area of Artificial Intelligence, the study of how tasks can be solved intelligently in an automated way. Working towards a next step in this direction was the main motivation behind my research. It originated from observing that at this moment, many tasks can be solved automatically, but it typically requires an expert, e.g., a programmer, and considerable time to develop a solution. I'm convinced that, in time, such a capability will be available to all.

Such a capability requires us to be able to solve many smaller tasks which, by themselves, are currently studied in their own fields within Artificial Intelligence, such as Natural Language Processing, Knowledge Representation, Machine Learning, etc. In my work, I focus on the field of Knowledge Representation (KR): the study of how knowledge (any piece of information) can be represented and how it can be used for automated problem solving.

Originally, solving problems by computer came down to writing assembler code, which directly addresses the hardware. Over time, this has evolved into the use of high-level programming languages. The latter is made possible by a large number of automatic compilation steps before reaching that hardware level. With the advent of these higher level languages, increasingly complex tasks, with large numbers of various constraints, are being addressed. Moreover, to accommodate for frequent changes in requirements, the ease of understanding and maintenance of software gains importance. Finally, to automate the many tasks that are currently still solved in a laboriously manual way, it is essential that solving these tasks requires less programming expertise and can more easily be mastered by domain experts. There is evidence of this evolution in many fields, including *administration* [Green et al., 2012], *scheduling* [Balduccini, 2011], *data mining* [Blockeel et al., 2012], *verification* [Mendonça de Moura and Bjørner, 2011], *configuration* [Vlaeminck et al., 2009] and *robotics* [Thielscher, 2000].

The foundation of the field of Logic Programming (LP) in 1974 with Kowalski's seminal paper *Predicate Logic as a Programming Language* [Kowalski, 1974] was an important step in that direction, by giving the Horn-clause subset of predicate logic a procedural interpretation to use it for programming. Gradually, it emerged that a logic program is in fact a definition and that the well-founded semantics [Van Gelder et al., 1991] is the most natural semantics to capture the meaning of definitions [Denecker, 1998, Denecker et al., 2001, Denecker and Vennekens, 2014]. The XSB Prolog system [Chen and Warren, 1996] was the first to support the well-founded semantics.

Whereas Prolog uses deduction as inference method, other inference methods exist. Most prominent is model generation as used in propositional SAT solvers. Also the inference method of Constraint Programming can be considered as model generation; indeed, its solvers attempt to assign values to variables while satisfying a set of constraints. Since that time, tremendous progress has been made in automated reasoning technology, particularly in SAT solving and Constraint Programming (CP). This has allowed the field of LP to explore more pure forms of declarative programming, where control is handled by the solver and the user only has to care about the problem specification, and for which *declarative modeling* is a more appropriate term. The best-known exponent of this research is the field of Answer Set Programming (ASP) [Baral, 2003, Brewka et al., 2011, Gebser et al., 2012a], where logic programs are interpreted according to the stable semantics.

All this progress raises the question what is the status of predicate logic as a modeling language. SAT is restricted to propositional logic. It can be considered as the assembler language for modeling. Indeed, there are many examples of programs that generate SAT encodings to obtain state-of-the-art solvers for various classes of problems. One can find examples in the areas of planning, hardware verification and of generating deterministic finite automata, to name just a few. However, SAT is not suited as a language for developing models. For what concerns ASP, it is an expressive high level language but it is not based on predicate logic. Today, many intricacies of stable model semantics are hidden in high level ASP constructs such as constraints and choice rules; however, its two forms of negation (“not” and strong negation) [Brewka et al., 2011] clearly distinguish it from first-order logic; the deviation from first-order-logic semantics could be an obstacle for newcomers.

In this work, we use the logic $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$, a different approach which stays closer to the origins of logic programming. It integrates inductive definitions (a generalization of Prolog’s rules under the well-founded semantics) with First-Order Logic (FO) formulas to express general knowledge about the problem domain. Historically, predicate logic was always viewed as a very expressive modeling language. This is remarkable given that anyone who used it for modeling a practical domain will have experienced its inconvenience in expressing certain common propositions. A clear weakness is expressing inductively definable concepts such as the transitive closure of a binary relation. Another deficiency is in expressing bounds on the cardinality or the sum of sets. Practical modeling languages in CP or ASP therefore support some of these propositions. A more conservative solution that preserves FO’s foundations is to extend it with suitable language constructs. For instance, it was argued in several papers, for example in [Denecker and Ternovska, 2008], that a rule set formalism under an extension of the well-founded semantics [Van Gelder et al., 1991] is a natural formalism to express the most common forms of inductive definitions. Such a formalism can be integrated with FO in a conceptually clean way. The resulting logic was named $\mathrm{FO}(ID)$ by [Denecker and Ternovska, 2008]. The link between $\mathrm{FO}(ID)$ and ASP was recently studied in [Denecker et al., 2012]. In this text, we use the notation $\mathrm{FO}(\cdot)$ to denote the family of extensions of first-order logic.

The IDP system we develop, which supports the $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$ language, is conceived as a *knowledge base System* (KBS). A KBS essentially consists of two components. On one hand, a *language* which is both formal (and unambiguous) and as natural as possible (i.e., the intended meaning of a sentence should correspond to its semantics) and, on the other hand, as many *inference techniques* as necessary. The more inference techniques are available in the KBS, the less programming is required of its users. The paradigm is inspired by several observations. First, *imperative programming* languages allow programmers to directly encode specialized algorithms, but knowledge about the problem domain is hidden deep within those algorithms. This facilitates high-performance solutions, but makes debugging and maintenance very difficult. Second, a program is typically written to *perform one task* and perform it well, but cannot handle many related tasks based on the same knowledge. Third, *knowledge representation* languages excel at representing knowledge in a natural, human-understandable format. Programming language designers are starting to realize this and provide constructs to express generic knowledge, such as the Language-Integrated Queries[linq, ] in Microsofts .NET framework. Last, the above-mentioned progress in automated reasoning techniques allows language designers to move the control burden from user to inference engine ever more and to reduce the importance of clever modeling on performance. The Knowledge Base paradigm [Denecker and Vennekens, 2008] is an answer to these observations: application knowledge is modeled in a high-level KR language and state-of-the-art inferences techniques are applied to reason on the modeled knowledge. The paradigm makes it possible to truly separate the representation of knowledge from the computation(s) for which it is used.

The question is also whether predicate logic can be used to solve practical problems. For this, we study the inference tasks of model expansion and optimization in detail. Model expansion is the problem of finding models of a given theory that expand a given partial structure that interprets at least the domain. Optimization also requires that the found model is optimal according to some optimization function. Both tasks are core tasks in the field of A.I., used for example to solve tasks in planning, scheduling and configuration, and closely related to the tasks of answer set generation and constraint satisfaction and optimization. The standard approach to model expansion works in two phases. First, in a *grounding* phase, the input theory is reduced to a propositional one, among others by instantiating quantified variables with values in the domain. In the second, *solving*, phase, a *search algorithm*, e.g., a SAT-solver, is applied to look for models of the reduced theory. An important problem is the fact that the grounding phase blows up the size of the theory. For large domains or complex constraints, this can make the first phase intractable. In this work, we develop techniques to address this bottleneck. First, we design a model expansion algorithm that allows function symbols to occur in the intermediate theory. Second, we extend the algorithm to tightly interleave the grounding and solving phase, further reducing the size of the grounding. Last, we also investigate how the new technologies can be applied automatically to remove this burden from the user.

**Contributions.** The main contributions of the presented research are:

- The development of the *knowledge base system* IDP, based on the logic $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$. The system provides a range of inference tasks such as *deduction*, *querying* and *model expansion*, and integrates tightly with the procedural language Lua.
- A demonstration of the applicability of the knowledge base paradigm and discussion of its advantages such as natural modeling, reduced de-

velopment time and acceptable or even superior performance compared to existing procedural solutions.

- Inference engines for model expansion and optimization for $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$, specifically addressing the problem of the blow-up of the size of the theory in the standard approach to model expansion.[1] This is achieved by (**i**) grounding to a richer intermediate language (the full ground fragment $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$), (**ii**) developing the MINISAT(ID) search algorithm for the ground fragment of $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ and (**iii**) developing a general framework on how to interleave grounding and search.
- A deduction engine for $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$, which works by transforming $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ theories into weaker FO theories and applying existing FO theorem provers. We present how deduction can be used to make model expansion more robust, by automatically detecting functional dependencies between arguments of the same symbol and exploiting them to obtain increased performance by building on (**i**) and (**ii**).

## 1.1 Structure of the Text

The rest of the thesis is structured as follows.

- In Chapter 2, the technical concepts and notations are introduced that will be used throughout the rest of the text. Among these are an introduction to first-order logic and search algorithms.
- The knowledge base system IDP is presented in Chapter 3, where we go into details on its language, the supported inferences and its integration with the imperative language Lua. We discuss modeling patterns for IDP and give an overview of available tools and applications in which it is used. A case study in the field of machine learning is developed in detail to illustrate the KBS approach.
- An inference engine for optimization for $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ is developed in Chapters 4 and 5. In Chapter 4, the overall workflow is presented, including pre- and postprocessing techniques and the grounding algorithm itself. In Chapter 5, we develop a search algorithm for the ground fragment of $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$.

---

[1] As model expansion is straightforwardly captured by optimization (using a constant optimization function), we typically use optimization to refer to both inference tasks.

- In Chapter 6, we show how an FO theorem prover can be integrated into IDP to provide deduction inference. Afterwards, we apply deduction to detect functional dependencies and show how these can be exploited to eliminate quantifications.
- In Chapter 7, we go one step further and develop a theoretical framework and practical algorithms to interleave the algorithms presented in Chapters 4 and 5 to lazily instantiate quantifications during search.
- Finally, conclusions on the whole of the presented research are drawn in Chapter 8.

## 1.2 Implementation and Experimental Evaluation

In the context of this thesis, the software packages IDP and MINISAT(ID) were developed. Both software packages are freely available and have been released under the open-source LGPL-3.0 license. Their latest versions are available at `dtai.cs.kuleuven.be/krr/software`. The version at the time of writing is available at `people.cs.kuleuven.be/broes.decat`, together with all benchmarks, experimental data, results and application specifications described in this text.

# 2

# Background

In this introductory chapter, we present the concepts and notations used through the rest of the text. It consists of three parts. In Section 2.1, we review FO, the formal language used as base language throughout the thesis. In Section 2.2, we give an overview of declarative modeling paradigms.

To ease the understanding of new concepts, we make use of the following example.

**Example 2.0.1.** Sudoku is a well-known Japanese puzzle. A Sudoku consists of a grid of (typically) 9x9 squares, with squares either blank or containing a number from 1 to 9. An example puzzle is shown in Figure 2.1. The task is to fill each blank square with a number from 1 to 9 such that each number occurs only once in each row, in each column and in each of the indicated blocks of 3x3 squares. A correctly solved Sudoku is shown in Figure 2.2.

## 2.1 First-Order Logic

In this section, we give an overview of FO and the notations that will be used throughout this thesis. For more details, we refer the reader to [Enderton, 2001].

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 2 | | 5 | | 1 | | 9 | |
| 8 | | | 2 | | 3 | | | 6 |
| | 3 | | | 6 | | | 7 | |
| | | 1 | | | | 6 | | |
| 5 | 4 | | | | | | 1 | 9 |
| | | 2 | | | | 7 | | |
| | 9 | | | 3 | | | 8 | |
| 2 | | | 8 | | 4 | | | 7 |
| | 1 | | 9 | | 7 | | 6 | |

Figure 2.1: An unsolved Sudoku.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *4* | 2 | *6* | 5 | *7* | 1 | *3* | 9 | *8* |
| 8 | *5* | *7* | 2 | *9* | 3 | *1* | *4* | 6 |
| *1* | 3 | *9* | *4* | 6 | *8* | *2* | 7 | *5* |
| *9* | *7* | 1 | *3* | *8* | *5* | 6 | *2* | *4* |
| 5 | 4 | *3* | *7* | *2* | *6* | *8* | 1 | 9 |
| *6* | *8* | 2 | *1* | *4* | *9* | 7 | *5* | *3* |
| *7* | 9 | *4* | *6* | 3 | *2* | *5* | 8 | *1* |
| 2 | *6* | *5* | 8 | *1* | 4 | *9* | *3* | 7 |
| *3* | 1 | *8* | 9 | *5* | 7 | *4* | 6 | *2* |

Figure 2.2: The unique solution of the Sudoku in Figure 2.1.

### 2.1.1 Syntax

A formal language is typically defined by a *vocabulary* (the set of available symbols), a set of *syntax* rules (how those symbols can be combined) and their *semantics* (how statements in that language are interpreted). In first-order logic, a vocabulary $\Sigma$ consists of a set of *predicate* symbols $\Sigma_P$ and a set of *function* symbols $\Sigma_f$. The vocabulary $\Sigma$ is then sometimes denoted as $\left\langle \Sigma_P, \Sigma_f \right\rangle$. We say a vocabulary $\Sigma'$ is a *subvocabulary* of $\Sigma$, denoted $\Sigma' \subseteq \Sigma$, if $\Sigma'_P \subseteq \Sigma_P$ and $\Sigma'_f \subseteq \Sigma_f$.

With each predicate symbol $P$ and function symbol $f$, we associate a natural number, their *arity*, which indicates the number of arguments the symbol takes. For an $n$-ary predicate symbol $P$, we sometimes use $P/n$ to indicate that $P$ has arity $n$, and similarly for function symbols. Propositional symbols are 0-ary predicate symbols, constants are 0-ary function symbols. Propositional symbols include $\top$ and $\bot$, denoting true, respectively false.

Predicate symbols are usually denoted by $P$, $Q$ and $R$, function symbols by $f$ and $g$ and constants by $c$.

**Example 2.1.1.** To formalize the Sudoku example, we use a vocabulary $\Sigma_{Sud}$, consisting of the predicate symbols $Number/1$ and $Box/3$ and the function symbol $Value/2$. The predicate symbol $Number$ denotes the set of allowed values and $Box(id, r, c)$ indicates to which box $id$ a square belongs, where the square $\langle r, c \rangle$ is identified by its row $r$ and column $c$. The function symbol $Value(r, c)$ represents our target function, which maps a square $\langle r, c \rangle$ to the value it contains.

A $\Sigma$-*structure* associates values to the predicate and functions symbols in $\Sigma$. Such a structure $\mathcal{I}$ consists of

- a *domain* $D^{\mathcal{I}}$, the set of *domain elements*, the allowed values. This is also known as the *universe*.
- an *interpretation* $P^{\mathcal{I}}$ for each predicate symbol $P/n$, with $P^{\mathcal{I}} \subseteq (D^{\mathcal{I}})^n$.
- an *interpretation* $f^{\mathcal{I}}$ for each function symbol, a mapping $(D^{\mathcal{I}})^n \mapsto D^{\mathcal{I}}$.

We assume the predicate symbol $=/2$ (equality) is implicitly contained in every vocabulary; $t_1 \neq t_2$ is a shorthand for $\neg(t_1 = t_2)$. The propositional symbols $\top$ and $\bot$ are respectively interpreted as **t** and **f**.

**Example 2.1.2.** Given the vocabulary $\Sigma_{Sud}$, we get the following $\Sigma_{Sud}$-structure $\mathcal{I}_{Sud,sol}$ for the Sudoku solution depicted in Figure 2.2:

$$\begin{aligned}
D^{\mathcal{I}} &= \{1,2,3,4,5,6,7,8,9\} \\
Number^{\mathcal{I}} &= \{1,2,3,4,5,6,7,8,9\} \\
Box^{\mathcal{I}} &= \{1,1,1;1,1,2;1,1,3;1,2,1;1,2,2;1,2,3;1,3,1;1,3,2;1,3,3;\ldots\} \\
Value^{\mathcal{I}} &= \{1,1 \to 4;1,2 \to 2;1,3 \to 6;1,4 \to 5;1,5 \to 7;1,6 \to 1;\ldots\}
\end{aligned}$$

We are now ready to define how to construct logical expressions over a vocabulary $\Sigma$. Terms are defined inductively as

- a *variable* $v$ is a term,
- if $f$ is an $n$-ary function symbol in $\Sigma$ and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a (function) term.

We usually consider logical expressions in the context of a fixed domain $D$, in which case domain elements are also considered terms.

A *formula* is defined inductively as

- if $P$ is an $n$-ary predicate symbol in $\Sigma$ and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is a formula, also called an *atom*,
- if $\varphi$ is a formula and $x$ is a variable, then $\neg\varphi$ (*negation*), $\forall x : \varphi$ (*universal quantification*) and $\exists x : \varphi$ (*existential quantification*) are formulas.
- if $\varphi_1, \ldots, \varphi_n$ are formulas, with $n \geq 2$, then $\varphi_1 \vee \ldots \vee \varphi_n$ (*disjunction*) and $\varphi_1 \wedge \ldots \wedge \varphi_n$ (*conjunction*) are formulas.

Atoms are usually denoted by $a$ and *literals* (atoms or their negation) by $l$. Variables are denoted by $x$ and $y$, domain elements by $d$, an ordered set of elements $e_1, \ldots, e_n$ by $\bar{e}$. Unless specified otherwise, theories and structures range over the vocabulary $\Sigma$. The expressions $\varphi \Rightarrow \varphi'$, $\varphi \Leftarrow \varphi'$ and $\varphi \Leftrightarrow \varphi'$ are shorthands for $\neg\varphi \vee \varphi'$, respectively $\varphi \vee \neg\varphi'$ and $(\neg\varphi \vee \varphi') \wedge (\varphi \vee \neg\varphi')$. The expressions $\forall\overline{x} : \varphi$ and $\exists\overline{x} : \varphi$, with $\overline{x}$ a tuple of variables, are shorthands for $\forall x_1 : \ldots : \forall x_n : \varphi$ and $\exists x_1 : \ldots : \exists x_n : \varphi$, respectively.

In the context of a domain $D$, a *domain atom* is an atom of the form $P(\overline{d})$ with $\overline{d}$ an $n$-tuple of domain elements and $P$ a predicate symbol of arity $n$.

Likewise, we consider *domain literals* and *domain formulas*, formulas without free (non-quantified) variables but in which domain elements may occur.

A *sentence* is a formula without free variables. A *theory* $\mathcal{T}$ over $\Sigma$ consists of a set of sentences over $\Sigma$. We denote the vocabulary of a theory $\mathcal{T}$ by $voc(\mathcal{T})$.

**Example 2.1.3.** We can now provide a formal representation of the rules of the Sudoku puzzle. The $\Sigma_{Sud}$-theory $\mathcal{T}_{Sud}$ consists of the following sentences.

$$\forall r\ c_1\ c_2 : Value(r, c_1) = Value(r, c_2) \Rightarrow c_1 = c_2 \tag{2.1}$$

$$\forall r_1\ r_2\ c : Value(r_1, c) = Value(r_2, c) \Rightarrow r_1 = r_2 \tag{2.2}$$

$$\begin{aligned}\forall r_1\ r_2\ c_1\ c_2 : Value(r_1, c_1) = Value(r_2, c_2) \wedge Box(r_1, c_1) = Box(r_2, c_2) \\ \Rightarrow r_1 = r_2 \wedge c_1 = c_2\end{aligned} \tag{2.3}$$

The sentences express that a number can only occur once in every row (2.1), column (2.2) and box (2.3).

A formula/term $f$ containing occurrences of a formula/term $f'$ is denoted as $f[f']$; the replacement of $f'$ in $f$ by $f''$ is denoted as $f[f'/f'']$. Given two tuples of terms $\bar{t}$ and $\bar{t}'$ of equal length $n$, $\bar{t} = \bar{t}'$ denotes the conjunction $t_1 = t'_1 \wedge \cdots \wedge t_n = t'_n$. For a tuple $\bar{e}$ and an element $e'$, $\bar{e} :: e'$ represents the concatenation of $\bar{e}$ and $e'$. For a set $S$ and an element $e$, $S + e$ and $S - e$ are shorthands for set addition $S \cup \{e\}$, respectively set difference $S \setminus \{e\}$.

For a $\Sigma$-structure $\mathcal{I}$ and a subvocabulary $\Sigma'$ of $\Sigma$, $\mathcal{I}|_{\Sigma'}$ denotes the $\Sigma'$-structure that has the same domain as $\mathcal{I}$ and interprets all symbols in $\Sigma'$ identically to $\mathcal{I}$.

Different from standard FO, we also allow that quantification is over explicitly specified subsets of the domain $D$, denoted as $\exists x \in D' : \varphi$ and $\forall x \in D' : \varphi$. Given a formula $\varphi$, $\varphi[\bar{x}]$ indicates that $\bar{x}$ are the free variables of $\varphi$. Substitution of a variable $x$ in formula $\varphi$ by a domain element $d$ is denoted by $\varphi[x/d]$. Over a domain $D$, $\exists x : \varphi$ and $\forall x : \varphi$ are equivalent to $\exists x \in D :: \varphi$ and $\forall x \in D :: \varphi$, respectively. A *ground formula* (in domain $D$) is a formula without variables (hence without quantifiers) and thus a Boolean combination (using $\wedge, \vee, \neg$) of domain atoms.

### 2.1.2 Semantics

The *value* of a term $t$ in a structure $\mathcal{I}$, denoted as $t^{\mathcal{I}}$, is the domain element $d$ if $t$ is the domain term $d$ or if $t$ is of the form $f(\bar{t}')$ and $f^{\mathcal{I}}(\bar{t}'^{\mathcal{I}}) = d$ (hence, $f(\bar{t}')^{\mathcal{I}} = d$).

The truth assignment function is defined by structural induction for pairs of formulas $\varphi$ and structures $\mathcal{I}$ that interpret $\varphi$:

- $P(\bar{t})^{\mathcal{I}} = P^{\mathcal{I}}(\bar{t}^{\mathcal{I}})$,
- $\neg\psi^{\mathcal{I}} = (\psi^{\mathcal{I}})^{-1}$,
- $(\varphi_1 \wedge \ldots \wedge \varphi_n)^{\mathcal{I}}$ is true iff $\varphi_i^{\mathcal{I}}$ is true for all $i \in [1, n]$,
- $(\varphi_1 \vee \ldots \vee \varphi_n)^{\mathcal{I}}$ is true iff $\varphi_i^{\mathcal{I}}$ is true for at least one $i \in [1, n]$,
- $\forall x : \psi$ is true iff for each $d \in D^{\mathcal{I}}$, $\psi[x/d]^{\mathcal{I}}$ is true,
- $\exists x : \psi$ is true iff for at least one $d \in D^{\mathcal{I}}$, $\psi[x/d]^{\mathcal{I}}$ is true.

A $\Sigma$-structure $\mathcal{I}$ is a *model* of / *satisfies* a $\Sigma$-sentence $\varphi$ (notation $\mathcal{I} \models \varphi$) if $\varphi^{\mathcal{I}} = \mathbf{t}$. A structure is a model of a theory $\mathcal{T}$ if it is a model of sentences in $\mathcal{T}$. If $\mathcal{I} \models \varphi$, we say $\varphi$ is *true* in $\mathcal{I}$, otherwise $\varphi$ is *false* in $\mathcal{I}$.

All structures interpret $\top$ as **t**, $\bot$ as **f** and equality ($= /2$) as identity.

### 2.1.3 Three- and Four-valued Structures

Often, one would want to specify structures which only have partial information or even allow structures that specify inconsistent information. To that end, we extend the notion of $\Sigma$-structure as follows. The interpretation of an $n$-ary predicate symbol $P$ in $\Sigma$, denoted $P^{\mathcal{I}}$, consists of two sets $P^{\mathcal{I}}_{ct}$ and $P^{\mathcal{I}}_{cf}$, both subsets of $(D^{\mathcal{I}})^n$. The interpretation of an $n$-ary function symbol $f$ in $\Sigma$, denoted $f^{\mathcal{I}}$, consists of two sets $f^{\mathcal{I}}_{ct}$ and $f^{\mathcal{I}}_{cf}$, subsets of $(D^{\mathcal{I}})^{n+1}$.

Informally, the set $ct$ (for "certainly true") is the set of tuples that are certainly *in* the relationship and the set $cf$ (for "certainly false") is the set of tuples that are certainly *not in* the relation. For a function symbol $f/n$, this coincides with interpreting the *graph symbol* of $f$: a predicate symbol $G_f/n+1$ that interprets $G_f$ as all tuples $\overline{d} :: d'$ where $f(\overline{d}) \mapsto d'$. For a symbol $s$, we sometimes refer to the sets of *possibly true* $s^{\mathcal{I}}_{pt}$ and *possibly false* $s^{\mathcal{I}}_{pf}$ tuples as all tuples in the domain that are not in $s^{\mathcal{I}}_{cf}$, respectively not in $s^{\mathcal{I}}_{ct}$, and the set $s^{\mathcal{I}}_{u}$ of all *unknown* tuples, not in $s^{\mathcal{I}}_{ct}$ nor $s^{\mathcal{I}}_{cf}$.

We then say the interpretation of an $n$-ary predicate symbol $P$ in a structure $\mathcal{I}$ is *two-valued* if $P^{\mathcal{I}}_{ct}$ and $P^{\mathcal{I}}_{cf}$ are disjoint and their union coincides with $D^n$. For an $n$-ary function symbol $f$, the above conditions have to hold (on $D^{n+1}$)

and, in addition, for each tuple $\overline{d} \in D^n$, there exists exactly one tuple $\overline{d} :: d'$, $d' \in D$, in $f_{ct}^{\mathcal{I}}$ (the function has exactly one image). A predicate interpretation is *three-valued* (or *partial*) if $P_{ct}^{\mathcal{I}}$ and $P_{cf}^{\mathcal{I}}$ are disjoint. A function interpretation is *three-valued* if $f_{ct}^{\mathcal{I}}$ and $f_{cf}^{\mathcal{I}}$ are disjoint and for each tuple $\overline{d} \in D^n$, there exists at most one tuple $(\overline{d} :: d') \in f_{ct}^{\mathcal{I}}$, with $d' \in D$, and at least one tuple $(\overline{d} :: d'') \notin f_{cf}^{\mathcal{I}}$, with $d'' \in D$. Four-valued interpretations provide neither guarantee. A structure $\mathcal{I}$ is two-valued if all its interpretations are two-valued and three-valued or partial if all its interpretations are three-valued. Otherwise, a structure is four-valued. In this text, we assume structures are four-valued unless specified otherwise.

We say a $\Sigma$-structure $\mathcal{I}$ is *more precise* than a $\Sigma$-structure $\mathcal{I}'$ if they have the same domain and for each symbol $s$, $s_{ct}^{\mathcal{I}} \subseteq s_{ct}^{\mathcal{I}}$ and $s_{cf}^{\mathcal{I}'} \subseteq s_{cf}^{\mathcal{I}}$. In that case, we say $\mathcal{I}'$ is an *expansion* of $\mathcal{I}$. We extend the notion of model: a partial $\Sigma$-structure $\mathcal{I}$ is a model of a $\Sigma$-theory $\mathcal{T}$ if for all two-valued expansions $\mathcal{I}'$ of $\mathcal{I}$, $\mathcal{I}' \models \mathcal{T}$ holds.

**Example 2.1.4.** We can now represent the input of the Sudoku puzzle as the partial $\Sigma_{Sud}$-structure $\mathcal{I}_{Sud,in}$ consisting of

$$\begin{aligned}
D^{\mathcal{I}} &= \{1,2,3,4,5,6,7,8,9\} \\
Number^{\mathcal{I}} &= \{1,2,3,4,5,6,7,8,9\} \\
Box^{\mathcal{I}} &= \{1,1,1;1,1,2;1,1,3;1,2,1;1,2,2;1,2,3;1,3,1;1,3,2;1,3,3;\ldots\} \\
Value_{ct}^{\mathcal{I}} &= \{1,2 \to 2;1,4 \to 5;1,6 \to 1;1,8 \to 9;2,1 \to 8;2,4 \to 2;\ldots\}
\end{aligned}$$

It is then easy to see that $\mathcal{I}_{Sud,sol}$ is an expansion of $\mathcal{I}_{Sud,in}$ that is a model of $\mathcal{T}$.

Next to the truth values true and false, we distinguish two additional truth values, namely *unknown* (**u**) and *inconsistent* (**i**). For a domain atom $P(\overline{d})$, we then say that $P(\overline{d})^{\mathcal{I}}$ is inconsistent if $\overline{d} \in P_{ct}^{\mathcal{I}}$ and $\overline{d} \in P_{cf}^{\mathcal{I}}$, true if $\overline{d} \in P_{ct}^{\mathcal{I}}$, false if $\overline{d} \in P_{cf}^{\mathcal{I}}$ and unknown otherwise. We say a term $f(\overline{d})$ has value $d'$ if $(\overline{d} :: d') \in f_{ct}^{\mathcal{I}}$ and $f(\overline{d})$ does not have value $d'$ if $(\overline{d} :: d') \in f_{cf}^{\mathcal{I}}$.

The (partial) *truth* order $>_t$ on truth values is defined by $\mathbf{t} >_t \mathbf{u} >_t \mathbf{f}$ and $\mathbf{t} >_t \mathbf{i} >_t \mathbf{f}$. The (partial) *precision* order $>_p$ on truth values is defined by $\mathbf{i} >_p \mathbf{t} >_p \mathbf{u}$ and $\mathbf{i} >_p \mathbf{f} >_p \mathbf{u}$. For two structures $\mathcal{I}$ and $\mathcal{I}'$ with domain $D$ interpreting the same vocabulary, we say that $\mathcal{I}'$ is an *expansion* of $\mathcal{I}$ if $P^{\mathcal{I}'}(\overline{d}) \geq_p P^{\mathcal{I}}(\overline{d})$ for all predicate symbols $P$ and tuples of domain elements $\overline{d}$. Viewing structures as sets of domain literals, this corresponds to $\mathcal{I} \subseteq \mathcal{I}'$.

The satisfaction relation is monotone: if $\mathcal{I} \leq_p \mathcal{I}'$, then $\varphi^{\mathcal{I}} \leq_p \varphi^{\mathcal{I}'}$. Hence, if a formula is true in a partial structure, it is true in all two-valued expansions of it.

A structure is determined by its domain and the truth value of its domain atoms. If $D$ and $\Sigma$ are fixed, we can use the following alternative way to represent a four-valued structure with domain $D$, namely as a set of domain literals. Indeed, there is a one-to-one correspondence between such sets $S$ and four-valued structures: for a domain atom $a$, $a^S$ is $\mathbf{i}$ if both $a$ and $\neg a$ are in $S$, $\mathbf{t}$ if only $a$ is in $S$, $\mathbf{f}$ if only $\neg a$ is in $S$ and unknown otherwise. From now on, we may treat four-valued structures as sets of domain literals and vice versa.

Multiple well-known semantics exist to define the satisfaction of formulas and the value of terms in a partial structure $\mathcal{I}$, which balance computational cost against completeness. On the one hand, according to *supervaluation* [van Fraassen, 1966], a formula is true in $\mathcal{I}$ iff it is true in all two-valued expansions of $\mathcal{I}$, false if it is false in all those expansions and unknown otherwise. For example the formula $P \vee \neg P$ always evaluates to true under supervaluation. This is the most complete semantics, but has an intractable computation cost. On the other hand, the *Kleene* semantics [Kleene, 1952] extends the truth tables of all operators to handle unknown. E.g., $\varphi \vee \psi$ is true if either $\varphi$ or $\psi$ is true, false if both are false and unknown otherwise. The result is that formula $P \vee \neg P$, with $P$ unknown in $\mathcal{I}$, is unknown, while it will clearly be true in any two-valued expansion of $\mathcal{I}$.

We sometimes refer to the *three-valued vocabulary* $\Sigma_3$ of a vocabulary $\Sigma$, by which we mean the vocabulary which extends $\Sigma$ with the symbols $s_{ct}$ and $s_{cf}$ for each symbol $s \in \Sigma$, and their interpretation in $\Sigma$-structures is then as above.

Unless the context specifies it differently, $\varphi$ and $\psi$ denote formulas, $t$ denotes a term and $\mathcal{I}$ a four-valued $\Sigma$-structure with domain $D$. Slightly abusing notation, with $\overline{d}$ a tuple of domain elements, we say that $P(\overline{d}) \in \mathcal{I}$ iff the tuple $\overline{d} \in P^{\mathcal{I}}$, and similarly for $f(\overline{d}) = d \in \mathcal{I}$.

### 2.1.4 Herbrand Interpretations

Sometimes, for example in logic programming, the domain is restricted to the *Herbrand universe*, the set of all terms over the vocabulary without variables or domain elements. A Herbrand interpretation $\mathcal{I}$ has the Herbrand universe as its domain and interprets each constant and function symbol by itself. The models of a theory that contain the Unique Names Axioms (UNA) [Reiter, 1980] and the Domain Closure Assumption (DCA) [Reiter, 1982] are isomorphic to Herbrand interpretations.

The UNA of a vocabulary $\Sigma$, which expresses that any two different domain terms map to different domain elements, can be expressed in FO as follows. For each function symbols $f/n$ in $\Sigma$ with $n > 0$, the sentence $\forall \overline{x}\ \overline{y} : f(\overline{x}) = f(\overline{y}) \Rightarrow \overline{x} = \overline{y}$, with $\overline{x}$ and $\overline{y}$ of length $n$. For each two function symbols $f/n$ and $g/n'$ in $\Sigma$, the sentence $\forall \overline{x}\ \overline{y} : f(\overline{x}) \neq g(\overline{y})$, with $\overline{x}$ of length $n$ and $\overline{y}$ of length $n'$.

The DCA of $\Sigma$, which expresses that the domain only contains objects represented by variable- and domain-element-free terms over $\Sigma$, cannot be represented in FO in general, as the representation would be infinite. It can however be expressed if the vocabulary only contains constants $c_1, \ldots, c_n$, namely through the sentence $\forall x : c_1 = x \vee \ldots c_n = x$. It can also be expressed in FO($ID$), the language which extends FO with inductive definitions (see next chapter), as shown in Section 6.1.

### 2.1.5 Equivalence and Normal Forms

Several notions of equivalence can be considered between theories. Classically, two theories $\mathcal{T}_1$ and $\mathcal{T}_2$ are said to be *equivalent*, denoted $\mathcal{T}_1 \equiv \mathcal{T}_2$, if $\mathcal{T}_1 \models \mathcal{T}_2$ and $\mathcal{T}_2 \models \mathcal{T}_1$, or in other words, each model of $\mathcal{T}_1$ is a model of $\mathcal{T}_2$ and vice versa.

Often, we are interested in equivalence in the context of a fixed vocabulary, and possibly also a fixed structure interpreting that vocabulary. To that end, for a vocabulary $\Sigma$ and a $\Sigma$-structure $\mathcal{I}$, we also define the notion of $\Sigma$-equivalence and $\langle\Sigma, \mathcal{I}\rangle$-equivalence as follows. Two theories $\mathcal{T}_1$ and $\mathcal{T}_2$ are $\Sigma$-equivalent if for each model $\mathcal{M}$ of $\mathcal{T}_1$, a model $\mathcal{M}'$ of $\mathcal{T}_2$ exists such that $\mathcal{M}|_\Sigma = M'|_\Sigma$, and vice versa. Two theories $\mathcal{T}_1$ and $\mathcal{T}_2$ are $\langle\Sigma, \mathcal{I}\rangle$-equivalent if for each model $\mathcal{M}$ of $\mathcal{T}_1$ that expands $\mathcal{I}$, a model $\mathcal{M}'$ of $\mathcal{T}_2$ exists, also expanding $\mathcal{I}$, such that $\mathcal{M}|_\Sigma = M'|_\Sigma$, and vice versa. We call the equivalence *strong* if a one-to-one mapping exists between the models of both theories and *weak* otherwise.

**Example 2.1.5.** For example, consider an alternative Sudoku theory $\mathcal{T}_{Sud,2}$ over a vocabulary $\Sigma_{Sud,2}$ as follows. The vocabulary $\Sigma_{Sud,2}$ extends $\Sigma_{Sud}$ by adding the predicate symbols $SameRow/4$, $SameCol/4$ and $SameBlock/4$. Theory

$\mathcal{T}_{Sud,2}$ consists of the sentences:

$$\forall r_1\ c_1\ r_2\ c_2 : SameRow(r_1,c_1,r_2,c_2) \Leftrightarrow r_1 = r_2$$

$$\forall r_1\ c_1\ r_2\ c_2 : SameCol(r_1,c_1,r_2,c_2) \Leftrightarrow c_1 = c_2$$

$$\forall r_1\ c_1\ r_2\ c_2 : SameBlock(r_1,c_1,r_2,c_2) \Leftrightarrow (\exists id : Box(id,r_1,c_1) \land Box(id,r_2,c_2))$$

$$\begin{aligned}\forall r_1\ c_1\ r_2\ c_2 : (&SameRow(r_1,c_1,r_2,c_2) \lor SameCol(r_1,c_1,r_2,c_2)\\ &\lor SameRow(r_1,c_1,r_2,c_2)) \Rightarrow Value(r,c_1) \neq Value(r,c_2)\end{aligned}$$

The theories $\mathcal{T}_{Sud}$ and $\mathcal{T}_{Sud,2}$ are strongly $\Sigma_{Sud}$-equivalent.

#### Negation Normal Form (NNF)

A formula is in NNF if implications and equivalences are eliminated, $\neg$ only occurs directly in front of atoms and a logical operator never occurs as a direct subformula of the same operator (e.g., $a_1 \lor a_2 \lor a_3$ is in NNF, but $a_1 \lor (a_2 \lor a_3)$ is not).

### 2.1.6 CNF and Tseitin Transformation

The fragment of FO where the vocabulary only contains propositional symbols (0-ary predicate symbols) is referred to as Propositional Calculus (PC). A PC theory is in Conjunctive Normal Form (CNF) if it is a conjunction of disjunctions.

Any PC $\Sigma$-theory $\mathcal{T}$ in NNF can be transformed into CNF in polynomial time using *Tseitin introduction* [Tseitin, 1968]. This consists of (**i**) replacing an occurrence of a formula $\varphi$ by a new propositional "Tseitin" symbol $T_\varphi$ and (**ii**) adding the sentences $\neg T_\varphi \lor \varphi$ and $\neg\varphi \lor T_\varphi$ to the theory. Theory $\mathcal{T}$ can then be transformed into a strongly $\Sigma$-equivalent CNF theory as follows. First, apply Tseitin introduction recursively in a top-down fashion to conjunctions in $\mathcal{T}$ that occur as a direct subformulas of disjunctions. Second, push all negations down and replace sentences $\neg T_\varphi \lor (l_1 \land \ldots \land l_n)$ by the sentences $\neg T_\varphi \lor l_i$ for $i \in [1, n]$.

If the user is only interested in weak model equivalence (e.g., not in the exact number of unique $\Sigma$-models), a less strict approach is possible. Indeed, if $\varphi$ occurs in a monotone context (not under negation), it is sufficient to only add the sentence $\neg T_\varphi \lor \varphi$. If $\varphi$ occurs in an anti-monotone context, $\neg\varphi \lor T_\varphi$ suffices.

## 2.2 Declarative Paradigms

In this text, we develop a declarative modeling paradigm based on an extension of first-order logic. In this section, we give an brief overview of two other well-known modeling paradigms which are closely related to our work and with which we compare frequently, namely Constraint Programming (CP) and Answer Set Programming (ASP).

### 2.2.1 Constraint Programming

A Constraint Programming [Apt, 2003] *model* consists of a set of *variables* $V = \{var_1, \ldots, var_n\}$, each with an associated set of elements $dom_i$, and a set $C$ of constraints of the form $name(vars)$, with $vars \subseteq V$ and $name$ is the type of constraint, from a predefined set of constraint classes. The set $dom_i$ is also referred to as the *domain* of $var_i$. The set of allowed domains and constraint types depends on the specific CP language selected. Some generally accepted domains are Boolean domains ($\{true, false\}$), sets and ranges of integer or real numbers, sets of the previous types, etc. A type of constraint takes a specified number of variables as arguments, with restrictions in which domains they can take. Well-known types of constraints are binary comparisons ($a < b$, with $a$ and $b$ variables), arithmetic operations, alldifferent, permutation, etc.

For each type of constraint, it is defined for which assignment of its variables the constraint is *satisfied*. For example an alldifferent constraint is satisfied if all of its arguments are assigned different values.

A variable assignment $A$ over $V$ is called a *solution* with model $\langle V, D, C \rangle$ if all constraints are satisfied in $A$. The problem of finding solutions to a CP model is referred to as a *Constraint Satisfaction Problem(CSP).*

**Example 2.2.1.** A Sudoku puzzle can be specified in a generic CP language as

```
int var n = 9
int var value[1..n,1..n] in 1..n

forall(i in 1..n) (
    all_different([x[i,j] | j in 1..n])
)
forall(i in 1..n) (
    all_different([x[j,i] | j in 1..n])
)
// similar constraint on blocks
// The input:
equal(value[1,2],2)
...
equal(value[9,8],6)
```

A solution to this model is also a solution of the original Sudoku puzzle.

The *Mixed Integer Programming (MIP)* paradigm [Nemhauser and Wolsey, 1988] is closely related to modeling in CP, although the set of supported constraints is not as rich (basically variables over integers and linear constraints).

## 2.2.2 Answer Set Programming

The field of ASP [Baral, 2003, Brewka et al., 2011, Gebser et al., 2012a] originated from the field of LP [Kowalski, 1974] in a move towards truly declarative semantics for logic programs. Given a vocabulary $\Sigma$ as in FO, a *logic program*[1] $\mathcal{P}$ consists of a set of rules of the form

$$q_1(\bar{t}) \mid \ldots \mid q_l(\bar{t}) \coloneq p_1(\bar{t}_1),\ \ldots,\ p_m(\bar{t}_m),\ not\ p_{m+1}(\bar{t}_{m+1}),\ \ldots,\ not\ p_n(\bar{t}_n)$$

with $\bar{t}$ a tuple of $\Sigma$-terms as in FO and $p_i$ predicate symbols in $\Sigma$. The domain of a logic program is the Herbrand universe of $\Sigma$. The *Herbrand base* is the set of all atoms $P(\bar{t})$, with $P$ in $\Sigma$ and $\bar{t}$ terms in the Herbrand universe.

The *ground program* $grnd(\mathcal{P})$ of $\mathcal{P}$ is the program consisting of all possible instantiations of rules of $\mathcal{P}$, with variables instantiated to elements of the Herbrand universe. A ground rule in $grnd(\mathcal{P})$ is satisfied in a $\Sigma$-structure $\mathcal{I}$ (a subset of the Herbrand base) if all its body literals are true in $\mathcal{I}$ (interpreting $not\ p(\bar{t})$ as $\neg p(\bar{t})$) and at least one of its head literals is true in $\mathcal{I}$ or at least one body literal is false and all head literals are false. If $\mathcal{I}$ satisfies all rules in $grnd(\mathcal{P})$, it is a *model* of $grnd(\mathcal{P})$ and of $\mathcal{P}$.

A structure $\mathcal{I}$ that is a model of $\mathcal{P}$ is an *answer set* of the program $\mathcal{P}$ if it is a subset-minimal model of the *reduct* of the program, the set of all rules in $grnd(\mathcal{P})$ in which negative body literals are replaced by their truth value in $\mathcal{I}$.

A rule without a head is called a *constraint*.

**Example 2.2.2.** The knowledge of the Sudoku problem can be represented as the following logic program. Note that, by convention, predicate and function symbols start with a lower-case character, variables by an upper-case one.

---

[1] The term “logic program” originated from Kowalski’s 1974 paper [Kowalski, 1974].

```
size(9).
number(1).
...
number(9).
value(X,Y,1) | ... | value(X,Y,9) :- number(X), number(Y).
:- value(R,C,V1), value(R,C,V2), V1!=V2.
:- value(R1,C,V), value(R2,C,V), R1!=R2.
:- value(R,C1,V), value(R,C2,V), C1!=C2.
// Similar constraint on blocks

// The input:
value(1,2,2).
...
value(9,8,6).
```

Again, answer sets of the above program correspond to solutions of the given Sudoku.

### 2.2.3 Terminology Table

As a convenience for readers familiar with CP or ASP, we show in Table 2.1 how the concepts of FO, CP and ASP relate to each others.

In the rest of the thesis, we will adhere to the terminology of first-order logic. We use *specification* to refer to specified knowledge in general.

| **FO** | **CP** | **ASP** |
|---|---|---|
| sentence | constraint | rule/constraint |
| constant | variable | constant |
| variable | — | variable |
| function symbol | variable array | function symbol |
| codomain[2] | domain | codomain |
| predicate symbol | Boolean variable array | predicate symbol |
| atomic sentence | — | fact |
| theory | model | logic program |
| interpretation | assignment | set of literals |
| model | solution | answer set |

Table 2.1: Relation between concepts in First-Order Logic, Answer Set Programming and Constraint Programming.

## 2.3 Combinatorial Search

In general, a *search* problem is the task of finding an item among a collection of items that satisfies a set of conditions. The term *combinatorial* search problem refers to problems for which items are defined by the values they take for a given set of properties. For example in Sudoku solving, the items are all possible assignments of values (1 to $n$) to squares. Hence, the size of the search space is defined by the number of possible assignments (or *combinations*). It is easy to see that the number of possible combinations grows exponentially with the number of properties.

Depending on what type of items and conditions are imposed, different classes of combinatorial search problems are distinguished. Given a parameter $n$ that controls the size of the structure of interest, the two best known ones are the $P$ and $NP$ complexity classes. A combinatorial search problem is said to be *in* $P$ if an algorithm exists that can decide in time polynomial in $n$ whether a solution exists. The structure that "proves" the existence is then called a *witness*. A problem is *in* $NP$ if deciding whether a solution exists might require more time than polynomial in $n$, but checking whether a witness is correct is still polynomial in $n$. An interesting other class is the class of *optimization* problems, where next to specific properties the structure should satisfy, a value function is associated with structures and the aim is to find a structure with an optimal value among all structures satisfying the required properties. Typically, the optimization version of a problem is more complex than the decision version.

A well-known problem in $P$ is finding the shortest path between two nodes in a graph, for which we can for example use Dijkstra's algorithm. A well-known problem considered to be in $NP$ is the satisfiability (SAT) problem: deciding for a given PC theory whether it has any models. Up till now, no proofs have been delivered that establish that any problem is in $NP$ and not in $P$, and that, hence, no polynomial algorithm exists.

Problems in $NP$ are found in a range of application domains. As they might take exponential time to solve (or worse), algorithms for combinatorial search are studied extensively in various fields such as CP, ASP, MIP, Local Search, SAT, etc.

One class of algorithms traverse the search space by incrementally building the solution: states correspond to the set of values each property can still take and the set of conditions considered. Search starts from the state in which no values have been eliminated and proceeds by applying four general techniques:

**Choose** splits the remaining search space into several parts (*branches*) and one is selected into which search continues.

**Propagate** uses information from one of the conditions to make the current state more precise. Whenever it is detected that a condition can no longer be satisfied, this is referred to as a *conflict*.

**Learn** derives a condition that was entailed by some conditions and adds it explicitly to the set of conditions.

**Backtrack** returns to a previous state, for example when a conflict has been found. As conditions are entailed by the input, they are typically not erased on backtracking. If a choice was made in the state to which backtracking returned, choose might now select another of the branches.

**Example 2.3.1.** Consider the following instance of the SAT problem (recall, it takes a PC theory as input):

$$\begin{array}{l} A \lor B \lor C \\ \neg B \lor C \\ \neg C \lor \neg B \end{array}$$

Each clause is considered one of the conditions to be satisfied and a state is then a partial structure and set of clauses (initialized to the input theory). The four general techniques can then be instantiated as follows. **Choose** assigns on of the unassigned atoms. For example assign $B$ true. **Propagate** checks whether a clause has only one non-false literal left and assigns it true. This is referred to as *unit propagation*. In our case, the second clause results in propagating $C$ as $B$ is true, resulting in the third attempting to propagate $\neg B$, which results in a conflict. **Learn** applies resolution to a set of clauses. It could for example derive $\neg B$ as the resolvent of the last two clauses. **Backtrack** could return to the empty structure. With the added clause, $\neg B$ is now immediately assigned. The algorithm could then continue to make either $A$ or $C$ true, after which we have a structure of which all expansions are models of the theory and the algorithm can terminate.

Naturally, one does not want to explore the full search space if possible. For that purpose, when to apply which rule and how to apply it is governed by *heuristics*. Often, heuristics can make the difference between being able to solve the problem or not. The work in this thesis is based on the class of algorithms known as Conflict-Driven Clause-Learning (CDCL) SAT-solvers [Biere et al., 2009]. These algorithms are able to learn during search and use generic heuristics that performed well on a range of applications without requiring user intervention. Learning during search goes back to the work of [Doyle, 1979] on truth maintenance systems; one of the first proposals for its application in the context of solving constraint satisfaction problems, of which SAT-solvers are a special case, was in [Bruynooghe, 1981]. More details on CDCL solvers are provided in Chapter 5.

# 3

# The IDP Knowledge Base System

In this chapter, we present the IDP system, a state-of-the-art KBS. The system has already existed for several years, but only recently evolved into a KBS. Up until 2012, IDP was a model expansion system (the IDP$^2$ system)[1] capable of representing knowledge in a rich extension of FO and performing model expansion by applying its grounder GIDL and its solver MINISAT(ID). Recently, we have extended it into *the* IDP *knowledge base framework* for general knowledge representation and reasoning (referred to as IDP$^3$); the earlier technology is reused for its model expansion inference. The IDP system goes beyond the KBS paradigm and is in fact a *knowledge base Programming Environment* [De Pooter et al., 2011], which provides an imperative programming interface to make the KBS paradigm practically applicable in software engineering domains. Such an interface, in which logical components are first-class citizens, allows users to handle input and output (e.g., to a physical database), to modify logical objects in a procedural way and to combine multiple inferences to solve more involved tasks. Here, we use KBS to refer to this three-component architecture consisting of language, inferences and procedural integration. The IDP system provides such a procedural integration through the scripting language Lua [Ierusalimschy et al., 1996].

---

[1] Given a logical theory and a structure interpreting the domain of discourse, model expansion searches for a model of the theory that extends the structure.

The system's name "IDP", *Imperative-Declarative Programming*, also refers to this paradigm.

In the work revolving around IDP, we can distinguish the knowledge representation language $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ and the state-of-the-art inference engines. One can *naturally* model diverse application domains in $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$; this contrasts with many approaches that *encode* knowledge such that a specific inference task becomes efficient. Furthermore, *reuse* of knowledge is central. $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ is modular and provides fine-grained management of logic components. E.g., it supports *namespaces*: formulas and terms can be declared in one component and used in several other components. The implementation of the inference engines provided by IDP aims at the reuse of similar functionality (see Chapter 4). This has two important advantages: (i) improvement of one inference engine (e.g., due to progress in one field of research) immediately has a beneficial effect on other engines; (ii) once "generic" functionality is available, it becomes easy to add new inference engines. To lower the bar for modelers, we aim at reducing the importance of clever modeling on the performance of the inference engines. We discuss this in the context of the model expansion inference in Chapters 4 and 6.

At this moment, various applications have already been developed using IDP, in fields such as scheduling, configuration and machine learning. We demonstrate its applicability through a case study in the context of one of those applications, namely a study in the field of *stemmatology*, a field of philology that studies the relationship between surviving variants of a text. We also discuss relevant $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ modeling patterns and the ease with which different tasks can be solved while reusing parts of the specification.

The main contribution of this chapter is to demonstrate how the theoretical ideas underpinning an FO-based knowledge base system, such as the various extensions of FO for knowledge representation and the well-known inference tasks such as model expansion and deduction, can be put into practice. As such, we do not present novel theoretical concepts here, but demonstrate the value of separating knowledge from computation. As such, the building of the system itself, its architecture, practical language and vision form the contribution here.

The main results of this section have been published in [De Cat et al., 2014] and [Blockeel et al., 2013]. An initial version of this work was published as [De Pooter et al., 2011].

In the rest of the text, we use IDP to refer to the $\mathrm{IDP}^3$ knowledge base system. Whenever relevant, we will specify which version of $\mathrm{IDP}^3$ is referred to. The system as presented in this chapter will be available soon as version 3.4; it will also contain most of the features presented in the rest of this text.

The chapter is structured as follows. In Section 3.1, we formally review the extensions with which $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ extends FO.The general IDP framework as well as a running example about course administration are presented in Section 3.2. This is followed by the main KBS components; the knowledge representation language $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ is described in Section 3.3, also motivating the choice of language extensions, and the system's high-level architecture and main inference engines in Section 3.4. After presenting the IDP system, we go into the practical use of IDP. In Section 3.5.1, we give an overview of modeling best practices using $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$. In Section 3.5.2, an overview is given of the tools that are available to aid the user in her modeling efforts. In Section 3.5.3, we discuss in which (classes of) applications IDP is currently used and we present the stemmatology case study. A discussion on related work is presented in Section 3.6, followed by a conclusion.

## 3.1 Formal Base Language

Before defining the language of the IDP system, we give an overview of its formal basis, the logic $\mathrm{FO}(ID, Agg, PF)$, an extension of FO with inductive definitions, aggregates and partial functions.

We assume the domain of a structure is always totally ordered. Every vocabulary contains comparison operators: the predicates $= /2$, $\neq /2$, $< /2$, $> /2$, $\geq /2$ and $\leq /2$, which are interpreted as usual according to the total order. We use $\sim$ to denote any comparison operator.

To support arithmetic operations and aggregates, the domain of every structure is a superset of all integer and real numbers. They are ordered before every other domain element, and, internally, they follow the natural order.

### 3.1.1 Partial Functions

A function can be declared as *partial*, indicating that for some inputs, the output can be undefined; otherwise the function is *total*. If for a function $f$, a structure $\mathcal{I}$ and $\overline{d} \in D$, it holds that $f(\overline{d})^{\mathcal{I}} = \emptyset$, we say the image is *undefined*. The interpretation of a term with a direct subterm that is undefined is also undefined; that of an atom with a direct subterm that is undefined is *false*. Motivation for these choices is given in the next section.

An interpretation for a total $n$-ary function symbol is two-valued if, among others, each input tuple has exactly one image. For a partial function, the condition is relaxed to having zero or one images. The condition for being three-valued is relaxed to having at most one tuple $\overline{d} :: d'$ that is true for any input tuple $\overline{d}$.

### 3.1.2 Sets and Aggregates

Set expressions are expressions of the forms $\{\overline{x} : \varphi\}$ and $\{\overline{x} : \varphi : t\}$, with $\varphi$ any formula, the *set condition*, and $t$ any term, the *set term*. Given a domain $D$, an interpretation $\mathcal{I}$ and an assignment $\overline{d}$ to the free variables $\overline{y}$ of a set expression, the interpretation $\{\overline{x} : \varphi[\overline{y}/\overline{d}]\}^{\mathcal{I}}$ is the set $\{\overline{d}' \in \overline{D} \mid \varphi[\overline{x}/\overline{d}', \overline{y}/\overline{d}]^{\mathcal{I}} = \mathbf{t}\}$, the interpretation of $\{\overline{x} : \varphi[\overline{y}/\overline{d}] : t[\overline{y}/\overline{d}]\}^{\mathcal{I}}$ is the multiset $\{(t[\overline{x}/\overline{d}', \overline{y}/\overline{d}]^{\mathcal{I}}) \mid \overline{d}' \in \overline{D}$ and $(\varphi \land \exists y' : t = y')[\overline{x}/\overline{d}', \overline{y}/\overline{d}]^{\mathcal{I}} = \mathbf{t}\}$. Thus, in the context of a given assignment for the variables $\overline{y}$, the expression denotes the multiset of terms $t$ for which $\varphi$ holds. Note that variable instantiations that result in non-denoting set terms are excluded from the set (the $\exists y' : t = y'$ subformula). Unions of multisets are interpreted in a similar fashion.

We extend the notion of term to include *aggregate terms*. Aggregate terms are of the form $agg(S)$, with $S$ a set expression and $agg$ an aggregate function. Currently, the aggregate functions cardinality, sum, product, minimum and maximum are defined. The last four only take sets with tuples of arity 1. The cardinality function maps a set interpretation to the number of elements it contains. The aggregate functions sum, product, minimum and maximum map a set to respectively the sum, product, minimum and maximum (on the order of the domain elements) of the first (and only) element in each tuple in the set. Sum and product are defined to be 0, respectively 1, if the set is empty. Note that sum and product aggregate terms are undefined when the set interpretation contains non-numeric values; minimum and maximum are undefined for the empty set (so are partial functions).

Abusing notation, we generally use "function symbol" to refer to first-order symbols, thus not including aggregates.

### 3.1.3 Definitions

We extend the notion of formulas to include *definitions*. Definitions $\Delta$ are sets of rules of the forms $\forall \overline{x} : P(\overline{t}) \leftarrow \varphi$ or $\forall \overline{x} : f(\overline{t}) = t' \leftarrow \varphi$, with the free variables of $\varphi$ among the $\overline{x}$. We refer to $P(\overline{t})$ and $f(\overline{t}) = t'$ as the *head* of the rule and to $\varphi$ as the *body*. In the first form, $P$ is the defined symbol; in the second, $f$ is. The defined symbols of $\Delta$ are all symbols that are defined by at least one of its rules; all other symbols occurring in $\Delta$ are called *parameters* or *open* symbols of $\Delta$.

Intuitively, for each two-valued interpretation of the parameters, $\Delta$ determines the interpretation of the defined symbols in a unique way. The set of open symbols of $\Delta$ is denoted by $\mathrm{open}(\Delta)$, the set of defined symbols by $\mathrm{def}(\Delta)$.

The satisfaction relation of FO can be extended to handle definitions by means of the well-founded semantics [Van Gelder, 1993]. This semantics formalizes the informal semantics of rule sets as inductive definitions [Denecker, 1998, Denecker et al., 2001, Denecker and Vennekens, 2014]. First, consider definitions $\Delta$ that only define predicate symbols and let $\Delta'$ be the definition constructed from $\Delta$ by replacing each rule $\forall \overline{x} : P(\overline{t}) \leftarrow \varphi$ with $\forall \overline{y} : P(\overline{y}) \leftarrow \exists \overline{x} : \overline{t} = \overline{y} \wedge \varphi$. The interpretation $\mathcal{I}$ satisfies $\Delta$ ($\mathcal{I} \models \Delta$) if $\mathcal{I}$ is a parametrized well-founded model of $\Delta$, that means that $\mathcal{I}$ is the well-founded model of $\Delta'$ when the open symbols are interpreted as in $\mathcal{I}$.

When functions are involved, we transform them away so that we can use parametrized well-founded models as above. Let the *graph* predicate symbol of a function $f/n$ be the new predicate symbol $F/n+1$; given an interpretation $\mathcal{I}$, let $\mathcal{I}'$ be $\mathcal{I}$ with the interpretations of all function symbols $f/n$ replaced by an interpretation of the corresponding graph predicate symbols $F/n+1$ such that $F(\overline{d}, d')$ holds in $\mathcal{I}'$ if and only if $f(\overline{d}) = d'$ holds in I. Let $\Delta'$ be the definition constructed from $\Delta$ by replacing rules defining predicate symbols as above and replacing rules $\forall \overline{x} : f(\overline{t}) = t' \leftarrow \varphi$ as follows: first, replace them by $\forall \overline{y}, y' : F(y_1, \dots, y_n, y') \leftarrow \exists \overline{x} : \overline{t} = \overline{y} \wedge t' = y' \wedge \varphi$. Next, replace occurrences of terms $f(\overline{t})$ in the bodies. An atom $A[f(\overline{t})]$ is replaced by $\exists x : A[f(\overline{t})/x] \wedge F(\overline{t}, x)$; a set expression $\{\overline{x} : \varphi : f(\overline{t})\}$ is replaced by $\{\overline{x} : \varphi \wedge \exists x : F(\overline{t}, x) : x\}$; concerning priority of application, replacement is done from the leaves of the parse tree upwards. We say that $\mathcal{I}$ satisfies the definition $\Delta$ ($\mathcal{I} \models \Delta$) if $\mathcal{I}'$ is a parametrized well-founded model of definition $\Delta'$.

The *completion* of $\Delta$ for a symbol $P$, defined in $\Delta$ by the rules $\forall \overline{x}_i : P(\overline{t}_i) \leftarrow \varphi_i$ with $i \in [1, n]$, is the set consisting of the sentence $\forall \overline{x}_i : \varphi_i \Rightarrow P(\overline{t}_i)$ for each $i \in [1, n]$ and the sentence $\forall \overline{x} : P(\overline{x}) \Rightarrow \bigvee_{i \in [1,n]} (\overline{x} = \overline{t}_i \wedge \varphi_i)$; the completion for defined function symbols is defined similarly This set is denoted as $\mathrm{comp}(P, \Delta)$ , the union of all these sets for $\Delta$ as $\mathrm{comp}(\Delta)$

It is well-known that, if $I \models \Delta$, then $I \models \mathrm{comp}(\Delta)$ but not always vice-versa (e.g., the inductive definition expressing transitive closure is stronger than its completion).

A definition $\Delta$ is *total* if for each structure $\mathcal{I}$ that is two-valued on the $\mathrm{open}(\Delta)$ and is unknown on $\mathrm{def}(\Delta)$, a two-valued expansion of $\mathcal{I}$ exists which is a model of $\Delta$ [Denecker and Ternovska, 2008]. Equivalently, this is the case if the well-founded model of $\Delta$ in $\mathcal{I}$ is two-valued. Broad classes of definitions have been proven to be total [Denecker and Ternovska, 2008]. As explained in the same paper, it is a methodological guideline that the user write total definitions.

We can extend Tseitin introduction to allow the replacement of formulas in the body of rules. To apply Tseitin introduction to an occurrence of a formula $\varphi$ that occurs in the body of a rule in definition $\Delta$, replace $\varphi$ by a new symbol $T_\varphi$ and add the rule $T_\varphi \leftarrow \varphi$ to $\Delta$. As discussed in [Vennekens et al., 2007], care has to be taken when applying predicate introduction to definitions. A sufficient condition is that, in the body of rules, it is not applied under negation nor within aggregate expressions.

### 3.1.4 Types

The above language FO($ID, Agg, PF$) is already a rich KR language. However, in many real applications, the use of *types* might be desired. Here, we present a typed extension of this language in terms of a translation to FO($ID, Agg, PF$).

A vocabulary not only consists of predicate and function symbols, but also contains type symbols. Furthermore it assigns to each of its predicate and function symbols, a tuple of types (of the correct arity). Intuitively, it means that these predicates and functions are only defined on the (tuples of) domain elements within the correct type. Note that, for functions, totality means "defined over every well-typed tuple" instead of over the whole universe as in an untyped logic. Lastly, every quantified variable is also typed, meaning that the quantification does not range over the entire domain, but only over that specific type.

A typed vocabulary can be denoted as $\left\langle \Sigma_T, \Sigma_P, \Sigma_f \right\rangle$, consisting of the set of types $\Sigma_T$, the set of typed predicate symbols $\Sigma_P$ and the set of typed function symbols $\Sigma_f$. Consider, e.g., a vocabulary $\Sigma_T = \{human, country\}$, $\Sigma_P = \{visited[human, country]\}$ and $\Sigma_f = \{livesIn[human \mapsto country]\}$, which contains the types *human* and *country*, a predicate symbol expressing which countries someone has already visited and a function indicating where someone is living now.

Semantics of the typed language are defined in terms of the untyped language: types are treated as unary predicate symbols. In every structure, the interpretation of a typed predicate symbol can only contain tuples of domain elements within the correct type (for which the corresponding unary predicate symbol is true) and functions are only defined on those elements within the correct type. Note that this means that the translation of the typed language to the untyped language makes all function partial. Every quantification

$\forall x : \varphi$, $\exists x : \varphi$, $\{x \mid \varphi\}$ or $\forall x : P(t) \leftarrow \varphi$, where $x$ is typed $T$, is replaced by (respectively) $\forall x : T(x) \Rightarrow \varphi$, $\exists x : T(x) \wedge \varphi$, $\{x \mid T(x) \wedge \varphi\}$ or $\forall x : P(t) \leftarrow T(x) \wedge \varphi$.

Types are intended to mimic the classification of the domain of discourse. As such, it is natural to extend the notion of type to type *hierarchies*: a type can be a subtype or supertype of another type, indicating that an interpretation of the former will be a subset/superset of the latter. We could extend the above example with a type $male \subseteq human$, representing the subset of humans which are male.

## 3.2 IDP as a KBS

We start the section with a description of the architecture and a discussion of design decisions. We finish with sketching an application where the same knowledge is used for different tasks.

### 3.2.1 Architecture and Design Decisions

Here, we introduce the basic design decisions underlying the IDP system, the decisions that determine the look and feel of IDP as a KBS. While the implementation and the algorithms used in it may vary over time, these decisions are rather fixed. Let us recall the architecture of the KBS. Besides the two main components, the *language* and the *inferences*, there is also a *procedural integration* component. An overview is shown in Figure 3.1.

The first design decision, the one most visible to users, is about the language of the KBS. The language should be (i) *rich* enough so that users can express all their needs; (ii) *natural* enough so that theories stay close to the original (natural language) problem statement and are easy to read and to debug; and (iii) *modular* enough to allow for reuse and future extensions.

It is sometimes argued that the expressivity of a language should be limited, since this might “make the language” undecidable or intractable. We disagree. First, note that decidability and tractability depend on the task at hand. While deduction in first order logic is undecidable, other forms of inference, such as model expansion and querying in the context of a finite domain, are decidable. Second, while a more expressive language might allow users to express tasks high in the polynomial hierarchy, that does not imply that simple tasks become harder to solve. Rather to the contrary, stating the problem in a richer language

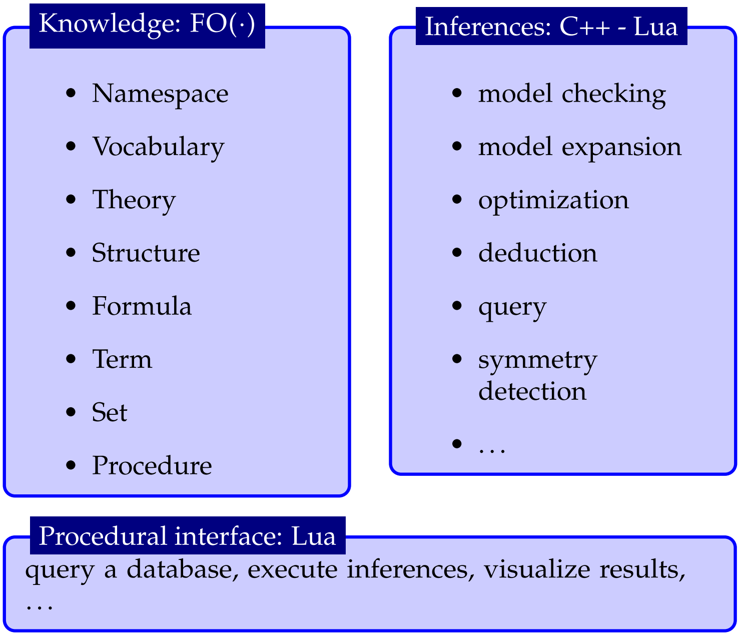


Figure 3.1: High-level representation of a knowledge base system

sometimes allow the KBS to exploit structural information that would be hidden in a more lower level problem statement.

To address the requirement of a rich and a natural language, we have opted for typed FO($ID, Agg, PF$), FO extended with definitions, aggregates, partial functions and types. First order logic because conjunction, disjunction, universal and existential quantification have a very natural meaning. Extensions because FO has various weaknesses. Inductive definitions overcome the weakness that FO cannot express inductively defined concepts. Also non-inductive definitions are very useful. They often eliminate the use of ambiguous if statements (in mathematical texts, it is common practice to use "if" when defining concepts; this "if" actually corresponds to an equivalence in first-order logic, or, as we would say, a definitional implication). Aggregates allow users to concisely express information requiring lengthy and complex FO formulas. Types are omnipresent in the context of natural language, where quantification typically refers to a specific set of objects (e.g., everyone is mortal). For the integration with a procedural language, we did not pin ourselves to one specific language, currently offering an interface to the languages Lua and C++.

The third requirement, modularity, is important both at the language and at the systems level. Our choice for first order logic as base language implies, as it is sentence-based, that we can add language extensions without much interference at the syntactical level. Consider for example the introduction of aggregates above; we only need to extend satisfaction to atoms in which an aggregate occurs in order to obtain a semantics for a language extended with aggregates. Opting for a purely declarative language for representing knowledge is also crucial to the modularity of the system. It paves the way for using the same knowledge for different tasks. While tasks do have a procedural component, they are organized from the interface where particular inference methods are invoked on specific, purely declarative, theories and structures. New inference engines can be invoked from that interface as they are added to the system. We have also attempted to organize inference engines in a modular way so that components can be reused in multiple engines. For example, the model expansion inference is currently implemented as ground-and-solve; the solver can be used separately from the grounder, and the grounding phase is composed of several smaller, reusable parts (such as, for example, evaluation of input$*$ definitions [Jansen et al., 2013]). Also various approaches to preprocess naive models are integrated in the system. Examples are symmetry detection and - breaking methods [Devriendt et al., 2012] and function detection methods [De Cat and Bruynooghe, 2013].

Such preprocessing techniques also work towards the aim of the system to provide *robust* inference engines. Indeed, to separate modeling as much as possible from performance considerations of specific inference engines, techniques to detect and exploit implicit knowledge are paramount.

### 3.2.2 Multiple Inferences Within One Application Domain

Given any knowledge base, there are often multiple applications that require different kinds of inference. By way of example, we explore the setting of a university course-management system. Its input is a database with information on students, professors, classrooms,. . . . One task of the system is to help students to choose their courses satisfying certain restrictions. Such an application is usually interactive; students make choices and, in between, the system checks the knowledge base. It removes choices when they become invalid, adds required prerequisites when a course is selected, . . . ; this is an example of *propagation* inference. Another task is to generate a schedule where every course is assigned a location and a starting time such that 1) no person has to be at two places at the same time, 2) no room is double-booked and 3) availability of professors is taken into account. Such an inference, searching for a valid solution, is called *model generation* or *model expansion*: one starts with partial information (availability of the professors, courses chosen by students,

…) and wants to extend it into a complete solution, namely a model of the scheduling theory. However, due to the large number of optional courses, such a solution (in which no student has overlapping courses) probably does not exist. In this case, we might want to find a solution in which the number of conflicts is minimal; this requires *minimization* inference. Now, one might want to mail students with schedules with overlaps to give them the opportunity to change their selection. Hence, the solution of the minimization inference should be *queried* to find the overlapping courses for every student. In the course of a semester, professors might have to cancel a lecture due to other urgent obligations. In that case, we want to find a *revision* of the current schedule, taking the changed restrictions into account and minimizing the number of changes with respect to the current schedule. In case such revisions are done manually, the *model checking* inference can be used so that no new conflicts are introduced. If some conflict does occur, an *explanation* should be provided. Finally, if a valid schedule is found, a *visualization* inference can be used to create an easy-to-understand, visual representation of the schedule, personalized by the viewer's status (student, professor, administrative personnel, …).

## 3.3 The Knowledge Base Language $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$

The language $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$, the knowledge representation language of the IDP system, is a modular and extendable language that provides a computer-readable notation for the concepts in $\mathrm{FO}(ID, Agg, PF)$, the formal base language.

In this section, we present the full $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ language. The first part follows the structure of the previous section; it defines syntax of language components and links them to the formal concepts previously introduced. Next, we present extensions to the language that aim at increasing its capability to act as a knowledge representation language and/or its capability for use as a general programming paradigm. We then extend the language with features that provide modularity, such as namespaces and reuse of specifications. Finally, we show how to integrate it with a procedural language. A discussion on related languages is delayed until Section 3.6.

## 3.3.1 Language Basics

### Tokens and Their Role.

Statements are terminated by “`;`”. A tuple is specified by an enumeration of elements separated by commas and surrounded by brackets (e.g., “`(1,2)`”), a (multi)-set by an enumeration of elements separated by “`;`” and surrounded by curly brackets (e.g., “`{(1,2) ;(2,3) ;}`”). The separator (“`;`”) at the end of the enumeration is optional, as are brackets around tuples whenever clear from the context.

By *character*, we mean Latin letters, digits and most common special symbols (“,”, “.”, “;”, . . . ); or formally, any ASCII-character in the range 32 – 127 (note that this excludes, e.g., escape characters). A *string* is any sequence of characters except double quotes “`"`”. A *name* is a string that starts with a Latin letter and may contain Latin letters, digits and the special characters “`'`” and “`_`”. An *identifier* is either an integer, a floating point number (“`n.m`” with $n$ an integer and $m$ a natural number), a name or a string surrounded by quotes (e.g., “`"200 A"`”). Upper- and lowercase names are different identifiers[2]. Everything after the characters “//” and on the same line is considered a comment, as is everything between “/*” and the nearest next “*/”.

### Symbol Declarations

FO(·)$^{\mathrm{IDP}}$ is a typed logic; the domain of discourse is divided (by the modeler) into sets of domain elements that conceptually belong to different types (or classes). As such, it has all the benefits of strongly-typed programming languages (Milner's “well-typed programs can't go wrong”) at the expense of some verbosity. In the course management example, among the types considered are `courses`, `persons` and `locations`. Arguments of predicate and function symbols, as well as the the image of function symbols have to be typed by means of declarations. For example, the argument of a function `age` can be restricted to domain elements of type `person`, and the image to the natural numbers.

A vocabulary introduces a set of type, predicate and function symbols, as in the following example:

```
vocabulary database is {
  type course;
  type person;
```

---

[2] The language does not use cases to distinguish between different kinds of identifiers.

```
   type student subtype of person;
   pred takes[student, course];
   func age[person -> nat];
    ...
}
```

The vocabulary is named *database* and introduces three types; the third type, `student`, is declared as a subtype of `person` (all students are persons). It also declares a (binary) predicate symbol `takes`, a relation between students and courses, and a (unary) function symbol `age` mapping persons to their age (a natural number). There is no restriction on the order of the elements in a vocabulary; however, cyclic dependencies are forbidden; e.g., "`type student subtype of person; type person supertype of student;`" is illegal as `student` depends on `person` and vice versa.

A type is declared when a set of domain elements is judged to be of sufficient interest for the application at hand. As explained in Section 3.1.4, types can be compiled away; unary predicates are introduced and this results in an untyped logic. So, a type declaration `type T` introduces in fact a unary predicate `T`. This translation is exploited in the language and one can use `T` both as type (e.g., in the declaration of functions and predicates, and as the type of a variable) and as predicate (e.g., to check that a domain element belongs to a subtype). In the other direction, a unary predicate declared as `pred` does not introduce a type. A type hierarchy can be declared by adding subtype expressions (of the form "`subtype of T2`") and supertype expressions (of the form "`supertype of T2`") following the type declaration.

#### Well-typedness.

A vocabulary is *well-typed* if the type hierarchy is acyclic. A theory is well-typed if all variables are typed and a term $t$ of type $T$ only occurs in positions that are typed as $T$ or an ancestor of $T$. A structure is well-typed if no domain element $d$ occurs in a position typed as $T$ where $d \notin T$ and if for any two types $T$ and $T'$ with $T$ declared as supertype of $T'$ (or $T'$ as subtype of $T$), $T' \subseteq T$ holds.

One advantage of guaranteeing well-typedness is that types with a common ancestor can safely share domain elements. Consider for example a type `car` that is interpreted by a set of identifiers for the different cars. Accidental use of a term typed `car` in an arithmetic operation is not well-typed, as `car` is not a subtype of `int`. However, a term typed car can be used in an argument position declared as `vehicle` (with `vehicle` a supertype of `car`).

In Section 3.3.5, we show how overloading is supported.

Several shorthands are supported to reduce the verbosity of the language. For symbols of arity 0, one can write "`pred P`" instead of "`pred P[]`" and "`func C ->T`" instead of "`func C[->T]`".

### Structure.

Structures are described by a list of equalities of symbols to sets of tuples, and can be partial. Below is an example of a structure specification in FO(·)$^{\text{IDP}}$.

```
structure data1 over database is {
   course = {Logic; Chinese};
   student = {1..3};
   takes.ct = {1,Logic};
   takes.cf = {2,Chinese};
   age.ct = {1->25; 3->30};
}
```

It declares a structure named "data1"; it interprets the type `course` by the domain elements "Logic" and "Chinese" and the type student by the identifiers 1, 2, and 3. It also states that the atom `takes(1,Logic)` is true (the "ct", or certainly true, table) and the atom `takes(2,Chinese)` false (the "cf" table). Note that `takes(1,Chinese)` is in none of the sets, so the structure only partially interprets "takes". Finally, the structure states that students `1` and `3` are respectively 25 and 30 years old, while the age of student `2` is not known in `data1`.

As the example shows, the general form of an interpretation statement has the form of `pX = S`, with `p` the symbol, `X` either empty (the interpretation is total), `.ct` (the true tuples are given) or `.cf` (the false tuples are given) and `S` a set of tuples. The set `S` can take different forms. First, it can be a plain enumeration of tuples of domain elements. Second, it can be an expression of the form `{n..m}`, with `n` and `m` either both integer numbers, floating-point numbers or chars and with $n \leq m$, expressing that $S = \{\langle x \rangle \mid x \in [n..m]\}$. Lastly, one can also express a procedural interpretation; this is discussed in Section 3.4.3.

Some symbols are built-in; by default they are part of every vocabulary and their interpretation is included in every structure. First, the binary comparison operators $=$, $\neq$ (written $\sim=$), $<$, $>$, $\leq$ (written $=<$), $\geq$ (written $>=$) can always be used. The interpretation of $=$ and $\neq$ is the standard one. For the last four, the total order over the domain elements determines the interpretation. Second, the 0-ary predicate symbols `true` and `false`, which refer to $\top$, respectively $\bot$.

Third, a number of pre-interpreted types are implicit in every vocabulary and interpreted in every structure: The set of all strings (`string`), the set of all Latin letters (`char`, subtype of `string`), and the sets of all natural, integer and real numbers (`nat`, `int` and `real` with `nat` subtype of `int` and `int` subtype of `real`). Fourth, a number of binary arithmetic functions is provided, namely addition $+$, subtraction $-$, multiplication $*$, division / and modulo %. All of these are declared over `real` and the last two are partial. Lastly, the functions *min*, *max*, *pred* and *succ* are defined for each (root) type in a vocabulary. In any structure, for type $T$, `min[->T]` and `max[->T]` map to the smallest, respectively largest, domain element in $T$ (according to the total order on domain elements). The partial functions `pred[T->T]` and `succ[T->T]` map domain elements of $T$ to their predecessor, respectively successor, in $T$.

### Theory.

The following table shows how the basic operators of $\mathrm{FO}(\mathit{ID}, \mathit{Agg}, \mathit{PF})$ are denoted in $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$.[3]

| $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ | $\mathrm{FO}(\mathit{ID}, \mathit{Agg}, \mathit{PF})$ | $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ | $\mathrm{FO}(\mathit{ID}, \mathit{Agg}, \mathit{PF})$ |
|---|---|---|---|
| `true` | $\top$ | `false` | $\bot$ |
| `&` | $\wedge$ | `\|` | $\vee$ |
| `=>` | $\Rightarrow$ | `<=` | $\Leftarrow$ |
| `<=>` | $\Leftrightarrow$ | `<-` | $\leftarrow$ |
| `!` | $\forall$ | `?` | $\exists$ |
| `~` | $\neg$ | | |

Quantified formulas are written as "`!x[T]: f`" or "`?x[T]: f`" with `x` a variable, `T` a type and `f` a formula; a formula "`!x[T1], ..., xn[Tn]: f`" is a shorthand for "`!x[T1]: ... : !xn[Tn]: f`".

Below is a small scheduling theory using function symbols `courseOf[session ->course]`, `planned[session->timeslot]`, and `teaches[course->person]` and predicate symbols `available[person,timeslot]`, `takes[student,course]`, and `attends[student,session]`. The first sentence states that a teacher is available for all sessions of the courses he teaches; the second one that each student attends each session of all courses he takes.

```
theory sessionAssignment over scheduling is {
  ! sess [ session ]:  available (teaches(courseOf(sess)),planned(sess));
  !stud[student],  sess [ session ]:  takes(stud,courseOf(sess))
     => attends(stud,sess);
```

---

[3] To distinguish inequality and implication operators, note that the latter form arrows.

```
}
```

Whereas set notation was used in structures for enumerating interpretations of symbols, in theories it is used for multisets. An expression `{x1[T1], ..., xn [Tn]: f : (t1 ,..., tn) }` represents the multiset $\{\langle t_1(\overline{x}), \ldots, t_n(\overline{x})\rangle \,|x_1 \in T_1 \wedge \ldots \wedge x_n \in T_n \wedge f(\overline{x})\}$, i.e., the multiset of tuples `(t1 ,..., tn)` (of the specified types) for which the formula `f` holds. The last argument (tuple of terms) is optional; if absent, the multiset of empty tuples is taken (typically as input for the cardinality aggregate function).

Multisets can be used in different contexts. The aggregate functions minimum, maximum, sum, product and cardinality, denoted as `min`, respectively `max` , `sum`, `prod` and `#` or `count`, take a multiset as their only argument. An alternative for constraints on the cardinality of sets are *extended* existential quantifications, e.g., instead of writing `#{stud[student]: attends(stud,sess )} >= 5`, one can write `?>=5 stud[student]: attends(stud,sess)`. Formally, $\exists_{\sim n}\, \overline{x} : \varphi$ is equivalent with $\#(\{\overline{x} : \varphi\}) \sim n$.

Anywhere a multiset can be used, a predicate symbol $P$ can also be used, which then acts as a shorthand for $\{\overline{x} \mid P(\overline{x})\}$. E.g., the number of courses can be represented as `#(Course)`.

## 3.3.2 Partial Functions

In standard logic, function symbols denote total functions. In practice, partial functions are unavoidable, e.g., a function $spouse[person \rightarrow person]$ is naturally undefined for singles and the arithmetic operation division is undefined for zero. Partial functions are declared as `partial func` instead of plain `func`. Assigning a proper semantics to non-denoting terms, however, may give rise to ambiguity. For instance, $White(Unicorn)$ can be interpreted as "if the unicorn exists it is white" or as "the unicorn exists and is white".

In [Wittocx, 2010] and [Frisch and Stuckey, 2009], it has already been pointed out that many approaches exist to avoid ambiguity. The simplest solution is to restrict the syntax of formulas. One could, e.g., only allow terms of the form $f(t)$ in contexts where it is certain that $f(t)$ is defined. This option is often taken in mathematics, where terms like, e.g., $\frac{1}{0}$ are considered meaningless, but quantifications of the form $\forall x : x \neq 0 \Rightarrow \frac{1}{x} \neq 42$ are allowed as it is clear that the division $\frac{1}{x}$ will be defined for all relevant $x$. This idea has been implemented for example in the Rodin toolset for Event-B [Abrial et al., 2010], where for every occurrence of a partial function, it should be provable that the function will only by applied to values in its domain. It was also shown in [Wittocx, 2010] that, in the context of a KBS, such an approach is too restrictive; it can, e.g., not be used when it is not known in advance for which input arguments a function will be undefined. Another approach, proposed by Kleene [Kleene, 1952], falls back to a three-valued logic, in which undefinedness is made explicit. From a practical point of view, however, the semantics are counter-intuitive on some types of formulas, which is undesirable in a KBS. The approach taken in [Wittocx, 2010] is one that was first proposed in [Russell, 1905]. The value of an atom $P(t)$, where $t$ is a term with undefined subterms, is either **t** or **f** depending on the context in which the atom appears. In a positive (negative) context, it is interpreted as **t** (**f**). Intuitively, this choice maximizes the truth-value of the formula it occurs in. For example, $White(Unicorn)$ is false in a positive context if $Unicorn$ is undefined, but true in a negative context in that case.

After some years of experimenting, we selected a still different approach, which turned out to be flexible, intuitive and to allow for elegant modeling. The semantics is based on the conversion of the function to its graph predicate and boils down to replacing the atom $A[f(\bar{t})]$, that provides the context for the term, by the formula $\exists x : A[x] \land F(\bar{t}, x)$. Or, intuitively, for an atom to be true, nested function applications have to be defined. Note that this corresponds to the semantics of $A[f(\bar{t})] \land \exists x : f(\bar{t}) = x$. E.g., $White(Unicorn)$ is always false if $Unicorn$ is undefined. After this translation, the requirements of the Rodin toolset (provability that functions are only applied to values in their domain) are satisfied. We made the choice that atoms with undefined terms are considered false. The result is a semantics that is close to the *relational semantics* proposed in [Frisch and Stuckey, 2009]. For cases where the default choice does not result in the intended meaning, we provide the abbreviation `denotes(f(t))` for $\exists x : f(t) = x$, which the user can insert at a different location to suit her needs. For example, the atom `cost/number=<100`, with `cost` and `number` both function symbols, is false when `number` equals zero. If the user wants the atom to be true, she can use `~denotes(cost/number) | cost/number =<100`. Similarly, by default, an aggregate term is undefined if any of the term instantiations of its set is undefined; however, one can use `denotes(t) & f` instead of `f` in the formula of the aggregate to deviate from this default.

### 3.3.3 Definitions

The logic FO$(\cdot)^{\mathtt{IDP}}$ contains a definition construct to express different kinds of definitions. This construct is one of the most original aspects of FO$(\cdot)^{\mathtt{IDP}}$ and we explain it in somewhat more detail. For even more details we refer to [Denecker and Ternovska, 2008].

Definitions are one of the important blocks that science is built with. Much knowledge of a human expert consists of definitions. It is well-known also that, in general, inductive/recursive definitions cannot be expressed in FO. Definitions belong to informal language; no formal rules exist on how to write a definition but a frequently used linguistic convention is to express them as a set of informal rules. This convention is formalized in FO(·)$^{\mathrm{IDP}}$ where a formal definition is expressed using the optional keyword `define` followed by a set of rules of the form “`! x1 ... xm: P(t1 ,..., tn) <- f`” or “`! x1 ... xm: f(t1 ,..., tn)=t0 <- f`”, with the `ti` terms, `f` a formula and `P` a predicate symbol. Observe that the second rule defines a function symbol $f$.

Informal definitions have some extraordinary properties. Certainly those used in formal mathematical text strike us for the precision of their meaning. The formal semantics of FO(·)$^{\mathrm{IDP}}$ definitions carefully formalizes this meaning. Several types of informal definitions can be distinguished. Below, the three most common ones are illustrated with two formal and one informal definition: Definition 3.3.1 is a non-recursive one, Definition 3.3.2 is a monotone one, while the informal one, Definition 3.3.3 is by induction over a well-founded order or semi-order. The latter definition is by induction over the subformula order. Definitions over a well-founded order frequently contain non-monotone rules. For instance the rule defining $I \models \neg\alpha$ has a non-monotone condition $I \not\models \alpha$.

**Definition 3.3.1.** The relation of non-overlapping sessions is defined as:

```
definition {
   !s[session]: noOverlap(s,s);
   !s1[session], s2[session]:
      noOverlap(s1,s2) <- planned(s1)+length(s1)=<planned(s2)
                        | planned(s2)+length(s2)=<planned(s1);
}
```

**Definition 3.3.2.** The relation *canTake*, expressing which courses can be taken according to the course-dependency relationship *depends*, is defined as:

```
definition {
    !c1[course]: canTake(c1) <- !c2[course]: depends(c1,c2)
                                   => canTake(c2);
}
```

**Definition 3.3.3.** The satisfaction relation $\models$ of propositional logic is defined by induction over the structure of formulas:

- $I \models P$ if $P \in I$.

- $I \models \alpha \land \beta$ if $I \models \alpha$ and $I \models \beta$.
- $I \models \alpha \lor \beta$ if $I \models \alpha$ or $I \models \beta$ (or both).
- $I \models \neg\alpha$ if $I \not\models \alpha$.

The different sorts of definitions have different semantic properties. It is commonly assumed that the defined set is the least set that satisfies the rules of the definition. However, this is only true for monotone definitions, but not for non-monotone definitions such as Definition 3.3.3: there is no least relation that satisfies its rules. Still, there is an explanation that applies to both [Buchholz et al., 1981]: the set defined by an inductive definition is the result of a construction process. The construction starts with the empty set, and proceeds by iteratively applying non-satisfied rules, till the set is saturated. In the case of monotone definitions, rules can be applied in any order; but in the case of definitions over a well-founded order, rule application must follow the specified order. This condition is necessary for the non-monotone rules. If they would be applied too early, later rule applications may invalidate their condition. E.g., in the initial step of the construction of $\models$, when the relation is still empty, we could derive $I \models \neg\varphi$ for each $\varphi$, but the condition $I \not\models \varphi$ will in many cases later become invalidated. The role of the induction order is exactly to prevent such an untimely rule application. E.g., in Definition 3.3.3, one is not allowed to apply a rule to derive $\mathcal{I} \models \varphi$ as long as rules can still be applied for deriving the satisfaction of subformulas of $\varphi$.

The problem we face in formalizing this idea for the semantics of $\text{FO}(\cdot)^{\text{IDP}}$ definitions, is that the syntax of $\text{FO}(\cdot)^{\text{IDP}}$ does not specify an explicit induction order for non-monotone $\text{FO}(\cdot)^{\text{IDP}}$ definitions. Thus, the question is whether one can somehow "guess" the induction order. Indeed, if we look back at Definition 3.3.3, we see that the order is implicit in the structure of the rules: formulas in the head of rules are always higher in the induction order than those in the body. This holds true in general. It should be possible then to design a mathematical procedure that somehow is capable to exploit this implicit structure.

In [Denecker and Vennekens, 2007], this idea was elaborated. The induction process of an $\text{FO}(\cdot)^{\text{IDP}}$ definition is formalized as a sequence of three-valued structures of increasing precision. Such a structure records what elements have been derived to be in the set, what elements have been derived to be out of the set, and which have not been derived yet. Using three-valued truth evaluation, one can then establish whether it is safe to apply a rule or not. All induction sequences can be proven to converge. In case the definition has the form of a logic program and the underlying structure is a Herbrand interpretation, the resulting process can be proven to converge to the well-known well-founded model of the program [Van Gelder et al., 1991]. As such, the semantics of FO$(\cdot)^{\mathrm{IDP}}$ definitions is a generalization of the well-founded semantics, to arbitrary bodies, arbitrary structures and with parameters. This (extended) well-founded semantics provides a uniform formalization for the two most common forms of induction (monotone and over a well-founded order) and even for the less common form of iterated induction. Compared to other logics of iterated inductive definitions, e.g. IID [Buchholz et al., 1981], the contribution is that the order does not have to be expressed, as this can be very tedious.

### 3.3.4 Constructed Types

In LP and ASP, the universe is traditionally assumed to be closed and functions have Herbrand interpretations, where each ground function term maps to a unique domain element. However, for various applications, it is more natural to treat functions as open. Allowing for this is currently an important topic in, e.g., ASP research [Bartholomew and Lee, 2012, Lifschitz, 2012]. In our setting, the inverse problem presents itself. In FO, functions are considered open and can take any interpretation. However, sometimes it is more natural to fix the interpretation of ground terms over a function $f$ to unique and different domain elements of a specific type. Consider for example the notion of "direction" in a spatial-related application (with one distinct constant for each direction), or the different days of the week in our scheduling application. One way to impose this is to explicitly express the unique names and domain closure axioms.

As the UNA and DCA are quite cumbersome to encode, the language provides a more natural way to express this, constructing a typed, local Herbrand interpretation given a set of new *constructor* function symbols. The statement `type days constructed from Monday; Tuesday; ...; Sunday` declares a type (and predicate) `days` and 7 new constants (`Monday` etc.) that all map to the type days. Additionally, in any structure, each of those constants is interpreted by a new, anonymous domain element that is different from all other domain elements (UNA), and `days` is interpreted as those 7 domain elements (DCA). Without such a constraint, the alternative is to define a type `days` and constants `func Monday[->days]`, `func Tuesday[->days]`, ... in the vocabulary and provide a manual interpretation of both the type (`days = Monday, Tuesday, ...`) and all constants (`Monday=Monday, ...`) in the structure.

Non-constant function symbols are also supported in this construct, as well as recursion. Hence one can for example declare a (Prolog-like) list of integers as `type list constructed from nil ; cons[int, list ]` or a data type of binary trees of integers as `type tree constructed from nil; t[tree,int,tree]`. For the former, an anonymous domain element is constructed (lazily) for `nil` and for each term `cons(i,d)` with `i` an `int` and `d` an (anonymous) domain element of type `list`. Note that there are an infinite number of domain elements of type `list`. Having declared lists, one can write definitions such as `head(cons(h,t))=h` and `tail(cons(h,t))=t`.

A reader might note that various language expressions, such as recursively constructed types and types such as `int` and `real`, can give rise to infinite domains. Currently, most inference engines in IDP only provide preliminary support to generally handle such domains. In the context of model expansion, Section 4.3.5 discusses several approaches employed to that end.

### 3.3.5 Improve Usability

In the previous subsections, we already introduced various shorthands to facilitate the comfort of the user; here we introduce two additional features: namespaces —which increase the modularity of the language— and overloading. We end this subsection with showing how these two features pave the road for inclusion and reuse of various pieces of $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ code.

#### Namespaces.

Define a *logic component* as either a vocabulary, a theory, a structure or a term. Each logic component $o$ is associated with a *type* (`vocabulary`, respectively `theory`, `structure` and `term`) and a vocabulary (unless it is a vocabulary itself), referred to as $type(o)$, respectively $voc(o)$.

Logic components are associated with names in the context of a *namespace*, a new type of language construct that maps names to logic components and (other) namespaces. A namespace acts as a context for logic components, to allow overloading and increase modularity. For example, the following specification declares two namespaces, `A` and `B`; each one contains a (different) vocabulary `V`; moreover, `B` contains a term `t` over the latter vocabulary.

```
namespace A is {
   vocabulary V is { ... }
}
namespace B is {
   vocabulary V is { ... }
   term t over V is sum({x: P(x): t(x)});
}
```

Logic components and namespaces not declared in an explicit namespace are considered part of the implicitly defined namespace `global`.

As is clear already in the above example, names themselves do not have to be unique and we need to allow the user to uniquely identify logic components and namespaces. For each name in a namespace, we define the *Fully Qualified Name*(FQN) by induction as follows. The global namespace has `global` as FQN. Any name `name` in a namespace `n` has as FQN `fn.name`, with `fn` the FQN of namespace $n$. For symbols, the type information is included in the FQN, as to allow overloading within one vocabulary: the FQN of a predicate symbol `s[T1 ,..., Tn]` in a vocabulary $V$, with FQN `fv`, is `fv.s[T1...Tn]`. It is defined similarly for function symbols. In any well-typed specification, all FQNs have to be unique, except for the FQN of namespaces. All namespaces with the same FQN are considered to be partial declarations of one combined namespace.

Using the FQN, we could declare a theory `T` in `B` but over the vocabulary `V` in `A` by `theory T over global.A.V is { ... }`.

### Overloading and Disambiguation.

Always having to use the full FQN for any name would result in lengthy specifications, as does the requirement to explicitly type every variable. To resolve this, IDP has a disambiguation component in its preprocessing that associates names to logical components. When the ambiguity cannot be resolved, the user is forced to provide more details (the context and/or the type signature). Users need a good understanding of the disambiguation strategy as they often rely upon it to be less verbose. Not only can users omit parts of the FQN, they can also omit the type of quantified variables in sentences and definitions; they can even omit the outer universal quantifications, in which case all unknown names, in positions where a term is expected, are considered universally quantified over the whole sentence or definition. All this omitted information is recovered by the disambiguation strategy as follows.

A *disambiguation* is an assignment of types to variables and of logical components to (occurrences of) names that satisfies the following constraints.

- The type of the logic component has to be consistent with where it occurs in the specification. E.g., for `theory T over V is ...`, `V` has to be a vocabulary.

- The type of a variable is the most specific type in the type hierarchy that is a supertype of the types of all occurrences of the variable.
- If a logic component $o$ is assigned to a name $name$, there has to be an FQN $fqn$ that refers to $o$ such that $name$ equals $fqn$ or $name$ is a suffix of $fqn$. For symbols, $name$ can also be a suffix of $fqn$ without its type specification or $name/n$, with $n$ the arity of the symbol.
- For atoms $P(t_1, \ldots, t_n)$, the arity of the predicate symbol $s$ assigned to $P$ has to be $n$ and the output type of each term and the type of the associated argument position of $s$ have to have a common supertype in the type hierarchy. For a term $f(t'_1, \ldots, t'_m)$, an analogous property holds, namely that the arity of the function symbol $s$ assigned to $f$ is $n$ and that the type of the terms and their associated arguments have a common supertype.

A specification is well-typed if only one disambiguation exists. If no disambiguations exist, an error is reported; if multiple exist, the user is asked to provide more detail.

For example, consider the predicate symbols `P[T1]`, `P[T2]` and the function symbol `P[T1->T2]`, all declared in a vocabulary `V`. Their FQN is respectively `global.V.P[T1]`, `global.V.P[T2]` and `global.V.P[T1->T2]`. In that context, the sentence `P(P(x))` has only one disambiguation, so is well-typed. Indeed, the inner `P` can only refer to a function symbol named $P$, so there is only one possible value. In that case, `x` has to take type `T1` and the outer `P` has to be a predicate symbol of arity one with an argument of type `T2`, which also uniquely determines the logic component. The full specification of that sentence would then be `!x[T1]: global.V.P[T2](global.V.P[T1->T2](x))`.

### Inclusion of Logic Components.

To facilitate reuse, a vocabulary `V` can contain `includes name` statements, where `name` refers to a vocabulary. The (conceptual) effect is as if vocabulary `V` also contained all declarations in the latter vocabulary. A simple example:

```
vocabulary database is {
   type course;    type location;    type session;
   func nameOf[course->string];
}
vocabulary schedule is {
   includes database;
   func assigned[session -> course];
   func assigned[session -> location];
}
```

As vocabulary `schedule` includes vocabulary `database`, it also contains all symbols in `database`, such as `session`, and can use them in its own declarations.

Next to vocabularies, include statements are also allowed in theories, structures and namespaces, with analogous effect. Inclusion can be more refined, e.g., `includes database.nameOf` includes only the function symbol `nameOf [course−>string]` and the type `course` from `database` (`string` is present by default); similarly, a statement `includes data1.age;` in a structure component includes only the interpretation of `age` from structure `data1`. Note that include statements have to be stratified.

## 3.4 Inferences and System Architecture

In this section, we discuss the architecture and the inference methods of the IDP system.

### 3.4.1 Main Inferences

The IDP system supports a range of functionalities to solve various inference problems and other logic-related tasks. Below, we specify a task through its input arguments $i_1, \ldots, i_n$, a precondition $Pre(i_1, \ldots, i_n)$, output arguments $o_1, \ldots, o_m$, and a postcondition $Post(i_1, \ldots, i_n, o_1, \ldots, o_m)$. This specifies a task as a partial (potentially non-deterministic) function from a space of input values to a space of output values. An *inference* task is a task that has logic objects (logic components, symbols, domain elements, ...) as input and output and where the pre- and postcondition depend on semantic properties of these (logically equivalent input has logically equivalent output). For example, the normalization of a formula is considered a task, but not an inference task: an input formula $\varphi$ is transformed into a logically equivalent formula $\psi$. In the following, we discuss the inference tasks currently supported by IDP.

**Query inference** takes as input a structure $\mathcal{I}$ and a set expression $\{x \mid \varphi\}$ over a vocabulary $V$ and returns the set $\{x \mid \varphi\}^{\mathcal{I}}$. The formula $\varphi$ can only contain symbols that are two-valued in $\mathcal{I}$. However, for any symbol $P(\overline{T})$ in a vocabulary $V$, $V$ also contains two “derived” symbols $P_{ct}(\overline{T})$ and $P_{cf}(\overline{T})$. In any structure $\mathcal{I}$ interpreting $V$, $P_{ct}$ and $P_{cf}$ are interpreted in $\mathcal{I}$ as the set of all tuples of domain elements for which $P$ is true, respectively false, in $\mathcal{I}$ (and similarly for functions, which are interpreted over their graph). Query inference is implemented by transforming the set expression into an extended First-Order Binary Decision Diagram (FOBDD) and afterwards querying the FOBDD over $\mathcal{I}$, as described in [Wittocx et al., 2010].

**Model expansion** was originally defined in [Mitchell et al., 2006] as the inference task that takes as input a theory $\mathcal{T}$ over vocabulary $\Sigma$ and a two-valued structure $\mathcal{I}$ over a subvocabulary of $\Sigma$, and returns an expansion $\mathcal{M}$ of $\mathcal{I}$ that is a model of $\mathcal{T}$. Here, it will be the more general inference problem as defined in [Wittocx et al., 2013] that takes as input a (potentially partial) structure $\mathcal{I}$ over $\Sigma$ and returns an expansion $\mathcal{M}$ of $\mathcal{I}$ that is a model of $\mathcal{T}$. **Optimization inference** takes an additional term $c$ as input and produces models that have a minimal value for $c$. Both inference tasks are discussed in more detail in Chapters 4 and 5. As an extra argument, we allow an *output* vocabulary $\Sigma_{out} \subseteq \Sigma$, which expresses in which part of the model the user is interested. If an output vocabulary $\Sigma_{out}$ is provided, the inferences return $\Sigma_{out}$-structures for which a $\Sigma$-structure expansion exists that expands $\mathcal{I}$, is a model of $\mathcal{T}$ and optimal for $c$. **Model checking** is a special case of model expansion with $\mathcal{I}$ a two-valued structure interpreting $voc(\mathcal{T})$; it is implemented through model expansion.

**Propagation**$\langle \mathcal{T}, \mathcal{I} \rangle$ takes as input a theory $\mathcal{T}$ and a partial structure $\mathcal{I}$, returning a more precise partial structure $\mathcal{I}_{out}$ that approximates all models of $\mathcal{T}$ that are expansions of $\mathcal{I}$ [Wittocx et al., 2013]. Equivalently, propagation deduces ground literals that hold in all models of $\mathcal{T}$ that expand $\mathcal{I}$. The system provides a complete propagation system (co-NP complete) and a sound but incomplete polynomial algorithm based on a lifted form of unit propagation. The latter proved very useful for building interactive knowledge based configuration systems [Vlaeminck et al., 2009].

**Deduction** inference **entails**$\langle \mathcal{T}_1, \mathcal{T}_2 \rangle$ takes as input two $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$-theories $\mathcal{T}_1$ and $\mathcal{T}_2$ and returns true if $\mathcal{T}_1 \models \mathcal{T}_2$. In Chapter 6, we go into detail on how deduction is implemented in IDP, through the use of off-the-shelf theorem provers for FO, and show how deduction can be used to reduce the variable quantification depth and to replace predicate by function symbols, improving performance of the model expansion engine.

**$\Delta$-model expansion**$\langle \Delta, \mathcal{I}_{in} \rangle$ takes as input a definition $\Delta$ and a structure $\mathcal{I}_{in}$, interpreting all parameters of $\Delta$, and returns the unique model $\mathcal{I}$ that expands $\mathcal{I}_{in}$. This task is an instance of model expansion, but is solved in IDP using different technology. The close relationship between definitions and logic programs under the well-founded semantics is exploited to translate $\Delta$ and $\mathcal{I}_{in}$ into a tabled Prolog program, after which XSB is used to compute $\mathcal{I}$. Taking an extra formula $\varphi$ as input, with free variables $\overline{x}$, the same approach is used to solve the query $\varphi$ with respect to $\Delta$ and $\mathcal{I}_{in}$ in a goal-oriented way [Jansen et al., 2013].

Next to inferences, the system also provides a large number of smaller-scale functionalities. An example are (model-preserving) normalization procedures, that take a theory $\mathcal{T}$ and (optionally) a structure $\mathcal{I}$ and transform $\mathcal{T}$ into a theory satisfying certain properties. Below, an overview is given of a well-known subset of the normalization procedures available in IDP.

**flatten($\mathcal{T}$)** puts disjunction and conjunctions in left-associative form.

**simplify($\mathcal{T}$,$\mathcal{I}$)** replaces terms and formulas known in $\mathcal{I}$ by their interpretation.

**push-negations($\mathcal{T}$)** puts a theory in negation normal form.

**push-quantifications($\mathcal{T}$)** moves quantifications down if the associated variable does not occur in all subformulas.

**nest-variables($\mathcal{T}$)** uses atoms $x = t$, with $x$ a variable and $t$ a term, to replace $x$ by $t$ wherever allowed, possibly eliminating $x = t$ itself.

**ground($\mathcal{T}$,$\mathcal{I}$)** transform $\mathcal{T}$ into a ground theory $\mathcal{T}_g$ such that models of $\mathcal{T}_g$ can be mapped one-to-one to models of $\mathcal{T}$ expanding $\mathcal{I}$. The ground theory can be transformed into different language formats, through the option `stdoptions.language`, such as CNF (Clausal Normal Form), ECNF (the native format of MINISAT(ID)) or ground ASP.

Next to FO$(\cdot)^{\text{IDP}}$ inferences, more domain-specific inferences are provided. One example is **unsat-debugging**: given a theory $\mathcal{T}$ and structure $\mathcal{I}, \mathcal{I} \not\models \mathcal{T}$, find a minimal subset of $\mathcal{T}$ that is unsatisfiable in $\mathcal{I}$. Unsat-debugging is a great aid for users to help pinpoint modeling errors.

### 3.4.2 Internal Representation.

The system offers the language FO$(\cdot)^{\text{IDP}}$ to the user, but internally, a more manageable representation is necessary. The main representation closely follows the structure of the FO$(ID, Agg, PF)$ language: each component of the parse-tree is stored as a separate object with pointers to its children. E.g., a *theory*-object consists of a set of sentences, which can be retrieved and modified; a namespace provides a mapping from names to its subcomponents and a name resolution mechanism. The only exception is for interpretations which come in many different flavours (enumerated, symbolic, procedurally interpreted) and can be very large, so a naive enumeration would consume too much memory while access would be inefficient.

### 3.4.3 Procedural Integration

The integration of logic with a procedural language works in two directions. Within a procedural program, logical components and inferences are available to obtain more complex functionality. Within a declarative specification, symbols can be interpreted as the result of procedural calls, useful, e.g., for integration with external sources.

In the current implementation, we selected the Lua [Ierusalimschy et al., 1996] programming language as procedural language. We chose Lua because it is a well-known scripting language that aims at being easy to embed in other languages. Hence, it would allow us to easily develop and evaluate our ideas of integrating both paradigms before considering the move to other programming languages. Currently, we are exploring the integration of $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$ with two other languages, namely Scala and Haskell.

In the former direction, logical objects are made available to a procedural language. For that purpose, IDP allows users to write "procedure" components within namespaces that define Lua functions. For example the code `procedure append(left, right){ return left .. right; };` defines a function "append" that takes two arguments and returns their concatenation (".." is the string concatenation operator in Lua). Each logic object and namespace declared in the global namespace is associated with a corresponding object in the Lua run-time environment, with the same name. Hence, logic object and namespaces are first-class citizens in the Lua stack. Additionally, IDP inferences and functionalities and "procedure" components are available as native Lua functions. For example, the code `procedure onemodel() { return modelexpand(T,S)[1]; };` passes the variables `T` and `S` to the model expansion inference. If these refer to a theory, respectively structure, in the global namespace, model expansion is executed and the first model is returned[4].

In the latter direction (interpreting logical symbols through procedural calls), in a structure, symbols can be interpreted by procedures as follows. In the context of a structure $\mathcal{I}$, one can specify the interpretation of a predicate symbol `P[T1…Tn)]` by a Lua Boolean function `f` with the same type of arguments as `P = procedure f`. In that case, `P(d1…dn)` is true (false) iff `f(d_1…d_n)` returns "true", respectively "false". For a function symbol `f`, a procedure should, given `d1 … dn` as input, return the interpretation of `f(d1 … dn)`, or `nil` if the term is non-denoting. For example, interpreting a function by string concatenation, an operation that already exists in Lua, can be modeled as:

```
vocabulary V is { func add[string,string->string];
                  func c[->string]; }
```

---

[4] If no models exist, `nil` is returned, the Lua equivalent of an empty reference.

```
theory T over V is { c = add("Hello, ", "world.") }
structure S over V is { add = procedure append }
procedure append(left,right) is { return left .. right }
```

Here, ".." is the Lua operator for string concatenation.

Two restrictions on using a Lua function `f` as interpretation of a symbol $s$ are (**i**) that `f` is a *pure* function (i.e., the return value should only depend on the input arguments and there should be no side-effects) and (**ii**) that it results in a consistent interpretation for $s$ that also obeys its type signature.

FO(·) **Datamodel for Lua.** An IDP specification organizes its logic components in a hierarchy of namespaces with the namespace *global* at the root. Referencing a concrete element in this hierarchy is done through `n[s]`, with `n` the name of a variable that refers to a namespace $N$ and `s` the name of a logical object, namespace or procedure that belongs to $N$. In a similar fashion, an access `V[n]`, retrieves the symbol with name `n` from vocabulary `V`, and `S[s]` retrieves the interpretation of the symbol `s` from structure `S`.

For example, calling the procedure `main()` first passes the values of the variables `T` and `S` to the function `onemodel`. Next, the symbol $c$ with name `n` is retrieved from the vocabulary `V` and is used to retrieve the interpretation of $c$ from the model found. Finally, this interpretation is printed.

```
procedure main() { print(onemodel(T,S)[V[c]]); }
```

As is often the case, many inferences can be customized in some way. For example, how structures should be printed (interpretations of sets or lists of facts), how many models a call to model expansion should (try to) return or how much output should be printed while solving a query, . . . . In IDP, most of these options are set globally through the `stdoptions` Lua table. For example, `stdoptions.nbmodels=n;` indicates that subsequent model generation tasks should try to find `n` models. For more details, we refer the reader to the online manual and tutorial examples at `http://dtai.cs.kuleuven.be/krr/software/idp`.

**Dynamics.** Execution of the procedural program requires semantics for the state of logic components at different points in time. For a component $C$, we refer to its "version" at time $t$ as $C_t$, with $C_1$ that state after parsing. Datamodel operations change the underlying logic components (not just the contents of Lua variables). Dependencies in the declarative component of the language, such as includes, are *dynamic*. E.g., if a vocabulary $V$ includes vocabulary $V'$ and at $t$, a new symbol $P$ is added to $V'$, then $V_{t+1}$ also contains $P$. A similar principle holds for structures, theories, . . .

Such a behaviour is required to support modularity during execution. For example, consider a robot adding knowledge to his `world` theory that contains basic knowledge about the world around him. One would expect any theory including `world` to automatically refer to the newest knowledge instead to a copy of the old information.

**Example 3.4.1.** Consider again the Sudoku puzzle, with the vocabulary $\Sigma_{Sud}$ and theory $\mathcal{T}_{Sud}$ as presented in Chapter 2 (the full FO$(\cdot)^{\texttt{IDP}}$ specification of this example can be found in Appendix B.2). This time, we would like to generate Sudoku puzzles ourselves. The main property our puzzles should have is that they have a unique solution, so that a player never needs to make guesses.

Assume the procedures `creategrid(n)` and `solve( partialgrid )` are available. The former takes `n` as input and generates a $\Sigma_{Sud}$-structure representing an empty grid; the latter takes a $\Sigma_{Sud}$-structure and tries to return 2 different solutions to the corresponding puzzle (if they exist, otherwise less are returned). The latter is achieved by applying model expansion to the given structure and a theory expressing the Sudoku constraints (not shown).

Using our Lua interface, the Sudoku generation problem can then be solved as follows. We create a procedure `create`, shown below, which takes as input the desired size `n` of the Sudoku to be generated and returns a Sudoku puzzle as a $\Sigma_{Sud}$-structure. The procedure proceeds by first initializing $\Sigma_{Sud}$-structure `grid` to the empty grid. Afterwards, it iterates over attempting to generate 2 solutions for `grid` and randomly fixing a value in `grid` on which both solutions differ, until there is only one solution to `grid` anymore. Last, it loops over all squares with a fixed value and clears the square if clearing it still results in only one solution, until fixpoint. Which square to clear is selected randomly, through the procedure `randomly_permute` which takes a table as input and returns a random permutation of the table.

```
procedure create(n) {
    grid = creategrid(n) // create an empty grid
    values = grid[voc_Sud.Value].graph // Value graph interpretation

    // iterate until one solution is left
    stdoptions.nbmodels = 2
    currsols = solve(grid)
    while #currsols > 1 do
        repeat
            col = math.random(1,size)
            row = math.random(1,size)
            val = currsols [1][ voc_Sud.Value](row,col)
        until val ~= currsols[2][voc_Sud.Value](row,col)
        maketrue(values,{row,col,val})
        currsols = solve(grid)
    end

    // try to remove elements
    change = true
    while change do
        change = false
        permvalues = randomly_permute(values.ct)
        for i,v in ipairs(permvalues) do
            makeunknown(values,v)
            currsols = solve(grid)
            if #currsols == 1 then
                change = true;
                break
            else
                maketrue(values,v)
            end
        end
    end
    return grid
}
```

**File Inclusion.** The main method of creating logic objects is through providing a list of files, which are then parsed into the relevant logic objects. To allow distributing knowledge over multiple files, the *require* statement is provided: a file that contains a statement `require file ;` instructs IDP to also load the file `file`. For example, one can construct a file with a structure for each instance of a problem and this file can require to load the file(s) containing the vocabularies, the theories, the task, etc., needed to solve the problem of which the structure is an instance. It then suffices to load a single file to solve an instance. The system provides built-in packages with general-purpose components. These include predefined vocabularies and theories as well as extra inference methods, and can be included through `require−std name`, with `name` the name of the package. For example, `require−std mx` includes useful procedures related to model expansion.

**Interactive Shell.** Both Lua and $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$ are interpreted languages (or, at least, compiled Just-In-Time). This allows IDP to provide an interactive interpreter for the language, similar to interpreters available in, e.g., many Prolog systems, but now by executing Lua code on the fly. The interpreter provides *autocompletion*, suggesting available IDP procedures that match a partial input; the autocompletion also specifies the expected argument types.

## 3.5 IDP in Practice

In this section, we explore how IDP can be used for practical applications. We first discuss relevant modeling patterns and tools available to complement IDP itself. Afterwards, we give an overview of applications in which IDP is used and develop the stemmatology case study.

### 3.5.1 Modeling in $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$

It is well-known that there is a significant difference between the approach to develop procedural code and declarative models. In this section, we provide some useful insights on how common declarative modeling tasks can be accomplished in $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$, by developing an application related to course administration: a GUI-workflow manager for students to select the courses they will take. Additional modeling examples can be found in [Blockeel et al., 2013] and at `dtai.cs.kuleuven.be/krr/software/idp-examples`. As a number of KR languages, such as ASP, Zinc and ProB, share modeling patterns, additional modeling patterns can be found in the references mentioned in Section 3.6.

Below, we model part of the flow of a Graphical User Interface (GUI) for an application where students select their courses for the coming year. Such a general application allows users for example to simulate a specific flow or even apply model checking to verify temporal properties (the former is available as **progression** inference in IDP, model checking isn't available in IDP yet).

The simplified use-case we consider presents a user with a list of courses he can take, with associated weight (expected amount of work), where the user can select courses he wants to take (and deselect them again) and the system disables selection of courses for which selection would exceed the maximum allowed weight. A form of Linear Time Calculus (LTC) [Shanahan, 1997] is used as the basis of the specification, which is well-suited to model dynamic systems. We now give a overview of the specification and some important modeling practices.

The first components are a generic LTC vocabulary and theory expressing concepts such as actions and fluents and specifying the frame axioms. Modularity often increases the ease of comprehension of a specification, if combined with readable names. It also increases reusability. An important modeling principle is to create symbols for relevant, reusable concepts (*reification*). Types are sets of objects relevant to the domain at hand; properties of those objects are modeled as (non-type) predicates and functions. Types that depend on other types can be modeled through definitions or constructors.

```
namespace ltc is {
vocabulary V is {
   type time subtype of nat;
   type action;                    type fluent ;
   pred do[action,time];
   pred holds[fluent ,time];       pred initHolds[fluent ];
   pred causes[action, fluent ];   pred ends[action, fluent ];
}
theory T over V is {
    definition {
       !f: holds(f,MIN[time]) <− initHolds(f);
       !f t: holds(f,t+1)     <− cause(f,t);
       !f t: holds(f,t+1)     <− holds(f,t) & ∼ends(f,t);
   }
}
}
```

The above components are then used as building blocks for the concrete dynamic system modeled below. It defines the concrete actions (select/deselect) and fluents (selected courses) and course-related concepts. Large arities are an artifact of representing knowledge as a database. $FO(\cdot)^{IDP}$ is not limited to such representations and large arities often force the modeler to use deep nesting of quantifiers, reducing performance.

Defined concepts such as the selected courses and whether a course can still be taken are naturally modeled as definitions. On the other hand, sentences are used to model constraints, such as the constraint that only courses can be selected that are still available. Functional relations are modeled as function symbols, allowing them to be nested, with aforementioned advantages.

```
namespace course−selection is {
vocabulary cs−V is {
   includes ltc.V;
   type course;
   type fluent constructed from {selected(course)};
   type action constructed from {select(course); deselect(course)};
   func weightOf[course->nat]; func maxweight[->nat];
   pred takes[course,time];      pred canTake[course,time];
};
theory cs-T over cs-V is {
   includes ltc.T;
    definition {
      !c: initHolds(selected(c))  <- false;
      !c t: causes(selected(c),t) <- do(select(c),t);
      !c t: ends(selected(c),t)    <- do(deselect(c),t);
   }
    definition {
      !f t: takes(courseOf(f),t)  <- holds(selected(f),t).
      !c t: canTake(c,t)          <- sum({c':takes(c',t):weightOf(c')})
                                      +weightOf(c)=<maxweight;
   }
   !c t: do(select(c),t) => canTake(c,t) & ~takes(c,t);
    ...
}
}
```

### 3.5.2 Supporting Software Engineering

Next to the core systems IDP and its search algorithm MINISAT(ID), several tools are available that support the user in his or her modeling efforts.

**Declarative Visualization.** The tool IDP DRAW [IDPDraw, 2012] offers declarative visualization. IDP-DRAW takes as input a structure interpreting the fixed vocabulary $\Sigma_{\text{IDP-DRAW}}$, which contains symbols that are interpreted in the context of drawing objects on a screen (they are prefixed by *idpd_*).

**Example 3.5.1.** Consider a $\Sigma_{\text{IDP-DRAW}}$-structure consisting of the facts

$$idpd\_polygon(3, tr, 0, 0, 0, 1, 1, 0),$$

$$idpd\_color(tr, 0, 0, 0),$$

$$idpd\_xpos(tr, 10), \text{ and}$$

$$idpd\_ypos(tr, 10).$$

The facts express that a polygon *tr* should be drawn that

- is a triangle, with corners at coordinates $\langle 0,0\rangle$, $\langle 0,1\rangle$ and $\langle 1,0\rangle$,
- is black (RGB value $\langle 0,0,0\rangle$), and
- should be translated to coordinate $\langle 10,10\rangle$.

A common use case is to visualize structures $\mathcal{I}$ interpreting some application domain $\Sigma$. This is achieved by expressing the visualization as a definition with open symbols in $\Sigma$ and defined symbols in $\Sigma_{\text{IDP-DRAW}}$. Evaluating that definition over $\mathcal{I}$ results in a set of facts which IDP-DRAW can use to generate the visualization.

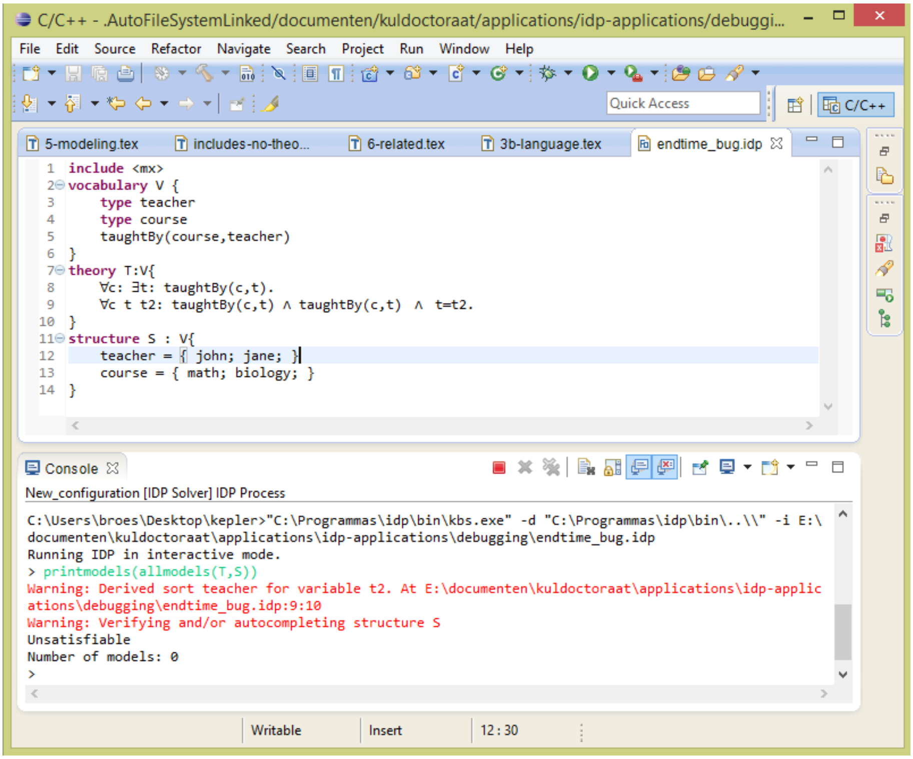
The system also supports time-dependent behaviour, useful in the analysis of dynamic systems, by adding a time parameter to all facts. The parameter then indicates at which point in time they take effect.

**Integrated Development Environment.** The plugin (IDP-IDE) is available for Eclipse, a well-known IDE Platform. IDP-IDE is still quite basic, but already supports auto-completion, syntax highlighting and automatically installs and calls IDP. Logical operators in ASCII syntax are visualized as their proper respective logical symbols. In Figure 3.2, a screenshot of IDP-IDE is shown. The plugin was developed by Philippe Rivière, as his master thesis.

**Debugging.** One issue users often face are errors in their specifications, because theories have either unexpected models or a structure turns out not to be a model while the user expected it. For the former, the user typically requires less assistance than for the latter, as she is provided with a solution (although unexpected) and use it to determine which of the constraints is not strong enough. For the latter, IDP is able to help the user through the **printcore** procedure, which takes a theory and (partial) structure as input and looks for a minimal subset of the theory which is already unsatisfiable in the structure. As a result, the user often gets a small (and instantiated) part of her theory which is unsatisfiable, making it significantly easier to find bugs.

**Example 3.5.2.** Consider the following $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$ specification also in the course scheduling domain.

```
vocabulary V is {
   type teacher;
   type course;
```


C/C++ - .AutoFileSystemLinked/documenten/kuldoctoraat/applications/idp-applications/debuggi...
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
C/C++
5-modeling.tex
includes-no-theo...
6-related.tex
3b-language.tex
endtime_bug.idp
Console
New_configuration [IDP Solver] IDP Process
Writable
Insert
12 : 30

```
include <mx>
vocabulary V {
    type teacher
    type course
    taughtBy(course,teacher)
}
theory T:V{
    ∀c: ∃t: taughtBy(c,t).
    ∀c t t2: taughtBy(c,t) ∧ taughtBy(c,t)  ∧  t=t2.
}
structure S : V{
    teacher = { john; jane; }
    course = { math; biology; }
}
```

```
C:\Users\broes\Desktop\kepler>"C:\Programmas\idp\bin\kbs.exe" -d "C:\Programmas\idp\bin\..\\" -i E:\
documenten\kuldoctoraat\applications\idp-applications\debugging\endtime_bug.idp
Running IDP in interactive mode.
> printmodels(allmodels(T,S))
Warning: Derived sort teacher for variable t2. At E:\documenten\kuldoctoraat\applications\idp-applic
ations\debugging\endtime_bug.idp:9:10
Warning: Verifying and/or autocompleting structure S
Unsatisfiable
Number of models: 0
>
```

Figure 3.2: A screenshot of the IDE, showing part of a scheduling application and the interactive IDP interface. The syntax is in a slightly older version of the language.

```
    pred taughtBy(course,teacher);
    ...
}
theory T over V is {
    !c: ?t: taughtBy(c,t);
    !c t t2: taughtBy(c,t) & taughtBy(c,t) & t=t2;
    ...
}
structure S over V is {
    teacher = { john; jane;  ...  };
    course = { math; biology;  ...  };
    ...
}
```

We only show the sentences in the theory that should express that each course is taught by exactly one teacher. However, the user made a common error: in the second sentence, he put a second conjunction instead of an implication. The result is a theory that has no models that expand any structure that contains more than one teacher (as can be seen from Figure 3.2). Using the **printcore** command, we get the following output.

```
Running IDP in interactive mode.
> printcore(T,S)
Warning: Verifying and/or autocompleting structure S
>>> Generating an unsatisfiable subset of the given theory.
>>> Unsatisfiable subset found, trying to reduce its size
    (might take some time, can be interrupted with ctrl-c.
The following is an unsatisfiable subset ...:
   ("jane" = "john")
instantiated from line 9 with c="biology", t="jane", t2="john".
>
```

The output indicates that the sentence at line 9 (the second sentence shown above) can be instantiated with `"jane"` and `"john"` such that the sentence `"jane"="john"` has to be true.

### 3.5.3 Applications

Currently, IDP is most often used for its model expansion and optimization inference. Recall that these are closely related to answer set generation and to solving CSPs. As such, IDP naturally shares applications with those domains, general examples of which are scheduling, planning, verification and configuration problems. Developing an algorithm in a procedural language is time consuming, error-prone and challenging. The use of a declarative modeling language liberates the programmer from the task and allows him to devote more time to the proper formalization. Moreover, the default heuristics of the underlying solvers are often sufficient to obtain adequate solutions. More concretely, model expansion and optimization using IDP is being applied in the following contexts.

- In the field of machine learning and data mining, IDP is being used in several applications, such as stemmatology (discussed below), modeling evolution [Labarre and Verwer, 2014] and automata learning [Blockeel et al., 2013]. In those applications, it was shown that IDP is able to replace procedural approaches (still common in that field) with minimal performance loss and greatly reduced development time. Specifically the latter makes it very suitable as a rapid prototyping approach for new ideas. Also in [Blockeel et al., 2013] (Section 5), it is demonstrated that a KR system like IDP can be practically used as a front-end for lower-level solvers such as SAT-solvers, instead of manually encoding problems in SAT or through custom scripting. Experimental results show that IDP is able to relieve this burden with minimal performance loss and a significantly smaller specification.

- In the context of software security, IDP is used in several contexts where an emphasis is placed on formal approaches that allow intuitive modeling of the involved knowledge. Examples are privacy analysis [Decroix et al., 2013], secure software design [Heyman, 2013] and ongoing work on distributed access control management.

- Configuration systems are another type of application well-suited for the KBS paradigm [Vlaeminck et al., 2009]. Here, one has some complex knowledge concerning possible configurations of a product (e.g., consumer goods such as bikes or cars, the setup of IT-networks) and one wants to solve different tasks with it, such as given a partial configuration, which choices are still available, or to automatically complete a partial configuration.

- It is used as part of a software benchmarking tool, where the experimental setup can be defined declaratively using IDP. Instead of tediously scripting which experiments to use in which setup, a logical vocabulary and structure are created that reflect the file system at hand and declarative constraints can be used to express which files should be combined in which way. These expressions are then solved using model expansion and experiments are run based on the resulting solution.

Another application is *bootstrapping*: building part of IDP through logical inferences supplied by IDP itself. As will be demonstrated in the following chapters, algorithms for state-of-the-art inference engines can become quite complex to design and to implement. An example discussed earlier is type derivation in IDP, another one is the well-known problem of query-optimization. For such applications, it can be tedious to develop a procedural algorithm or it might require the development of heuristics or exhaustive search if some cost function is involved. Another approach, which we are actively investigating, is to use a meta-representation of logical components (a vocabulary over terms, formulas, theories, . . . ). Such a representation allows us to represent formulas in a logical structure and to write logical expressions on the structure of such formulas. For example, the conditions under which a well-typed theory is a valid derivation for a partially typed theory are concise and clear cut. Using such a meta-level approach, model expansion inference can then be used to find logical theories that satisfy these conditions, instead of the need for a procedural algorithm. At this moment, only a few preprocessing steps are solved declaratively, namely definition processing (see Chapter 4) and the global lazy grounding approach (see Chapter 7), but the aim is to exploit the idea further to reduce development time and make the system more robust.

Last, IDP is used as a didactic tool in various logic-oriented courses, among others at KU Leuven. There, also the theorem proving turns out to be a useful feature.

### 3.5.4 Case Study: Stemmatology

The rest of the section is devoted to an application. The material is taken from [Blockeel et al., 2013]. The application is in the area of *stemmatology*, concerned with the study of ancient manuscripts. A *stemma* is a kind of "family tree" of a *tradition*, a set of related manuscripts. It indicates which manuscripts have been copied from which other manuscripts ("parents"), and which manuscript is the original source. It may include both extant (currently existing and available) and non-extant ("lost") manuscripts. A stemma is not necessarily a tree: sometimes a manuscript has been copied partially from one manuscript, and partially from another, in which case the manuscript has multiple parents.

More formally, a stemma can be defined as a CRDAG, a connected directed acyclic graph with a single root [Andrews and Macé, 2012]. A dataset contains the manuscripts from one tradition. Each manuscript is described by a fixed set of features $F_1, \ldots, F_n$, each of which has a nominal domain $Dom(F_i)$ (variant readings of feature $F_i$). Typically, a feature refers to a particular location or section in a text, though it can also be the spelling of a particular word, e.g., the dwelling of "Van den Vos Reynaerde" can be spelled as Malpertuis, Malpertus, or Malpertuus.

The 19th century philologist Karl Lachmann was among the first to apply a principled method for reconstructing stemmata from sets of manuscripts [Timpanaro, 2005]. Nowadays, a variety of methods exist. Many are borrowed from biology, where a similar problem, reconstruction of phylogenetic trees, is well-studied. However, these methods do not always fit the stemmatological context well. First, they assume that phylogenies are tree-shaped, while stemmata are DAGs.[5] Second, these trees contain only bifurcations, while stemmata can have multifurcations. Third, in most methods, the trees are such that each extant copy is at a leaf of the tree, whereas in stemmatology one extant copy may be an ancestor of another (and hence should be an internal node). Fourth, stemmatologists often have additional information, for instance about the time or place of origin of a manuscript, which ideally should be taken into account. Research continues to develop new algorithms better suited for the stemmatological context [Baret et al., 2006].

### The Task

Apart from reconstructing stemmata from data, stemmatologists are also interested in other types of analyses, which may, for instance, use a known stemma or a manually-constructed best-guess stemma as an input. These types of analysis can be very diverse. The data mining tasks we address in this section belong to this category.

The problem studied here assumes that a CRDAG representing a stemma of a tradition is given, as well as feature data about the manuscripts from the tradition. More specifically, the data include a feature for each location where variation is observed in the tradition represented by the stemma. For each extant manuscript in the tradition, the feature data describe its variant reading; the variant reading is unknown for the non-extant ones. For most features, it seems rather unlikely that the same variant reading originated multiple times independently; i.e., it is reasonable to assume there is one ancestor where the variant reading occurred for the first time (the “source” of the variant). Therefore, we say that *the feature is consistent with the stemma* if it is possible to indicate for each variant a single manuscript that may have been the origin of that variant. Since for some manuscripts the value of the feature is not known, checking consistency boils down to assigning a variant to each node in the CRDAG in such a way that, for each variant, the nodes having that variant form a CRDAG themselves. Note that one can imagine exceptions to the above, e.g., a new spelling of a word can be independently introduced in different copies.

### An $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ Solution

A first $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ solution used a binary relation `SameVariant` for representing that two manuscripts have the same variant reading and imposed two

---

[5] Some methods return phylogenetic networks, but these represent uncertainty about the real tree, which is different from claiming that the network represents the actual phylogeny.

constraints: (i) transitivity of `SameVariant` relation, (ii) manuscripts with the same variant reading have a common ancestor with that variant reading and are connected to that ancestor through manuscripts with that same variant reading. This resulted in a working version that could serve as a golden standard for the procedural code but was much slower than the latter.

Modeling transitive closures results in large grounding sizes and running time [Blockeel et al., 2013]. Hence, a major improvement can be expected when that can be avoided. Representing the variant reading as a function from manuscripts to variants allowed us to drop the transitivity constraint. The final improvement, resulting in the program below, came from learning more about the procedural code: it checks for connectedness by following a path to the original source manuscript of the variant reading and checks that there is a single such source for the variant reading. Expressing the latter as a single constraint resulted in a version that turned out to be faster than the incomplete procedural algorithm. The IDP model is shown in Listing 3.1 and explained below. We also show most of the procedural code, so that the reader can see how a number of satisfiability-checking tasks can be embedded in a single process.

Listing 3.1: Checking the consistency between stemma and features.

```
procedure main() {
  process("besoin");
  process("parzival");
  process("florilegium");
  process("sermon158");
  process("heinrichi");
}

/* ------ Knowledge base ------------------------ */
vocabulary V  is {
   type Manuscript;  type Variant;
   pred CopiedBy[Manuscript,Manuscript];
   func VariantReading[Manuscript -> Variant];
}
vocabulary Vtask is {
   includes  V;
   func SourceOf[Variant->Manuscript);
}
theory Ttask over Vtask is {
   ! x : (x ~= SourceOf(VariantReading(x))) =>
      ? y : CopiedBy(y,x) & VariantReading(y) = VariantReading(x);
}
```

```
/* ------ Check consistency between feature and stemma ---- */
procedure check(feature) {
  setvocabulary(feature,Vtask);
  return sat(Ttask,feature);
}

/* ------ Procedures for processing ----------------- */
procedure process(tradition) {
  io.write("Processing ",tradition,".\n");
  local path = "data/";
  local stemmafilename = path..tradition..".dot";
  local featurefilename = path..tradition..".json";
  processFiles(stemmafilename,featurefilename);
}
procedure processFiles(stemmafilename,featurefilename) {
  local stemma,nbnodes,nbedges = readStemma(stemmafilename);
  io.write("Stemma has ",nbnodes," nodes, ",nbedges, " edges.\n");
  local nbp,nbs,time = processFeatures(stemma,featurefilename);
  io.write("Found ",nbp," positive out of ",nbs," groupings ");
  io.write("in ",time," sec.\n");
}
procedure readStemma(stemmafilename) {
  /* 19 lines of lua code */
}
procedure processFeatures(stemma,featurefilename) {
  /* 23 lines of lua code
    a loop iterating over the features,
    -- compute feature as stemma extended with
      the feature specific data
    -- call check(feature)
    -- process the results
    finally, return the overall results */
}
```

The logical model is described in the "Knowledge base" section of the code. The vocabulary has been split in two parts. The vocabulary `V` is used to represent the input data: the stemma and the feature. It introduces the types `Manuscript` and `Variant`, the binary relation `CopiedBy` representing the parent-child pairs in the given structure of the stemma and the function `VariantReading` representing the known data about variant readings of manuscripts. The vocabulary `Vtask` extends `V` with the task-specific vocabulary. Only one extra

Table 3.1: The five traditions used in this work.

| Name | # manuscripts | # parent-child pairs | # features | # variant readings | |
|---|---|---|---|---|---|
| | | | | maximum | average |
| `Notre Besoin` | 13 | 13 | 44 | 5 | 2,18 |
| `Parzival` | 21 | 20 | 122 | 6 | 2,59 |
| `Florilegium` | 22 | 21 | 547 | 5 | 2,19 |
| `Sermon 158` | 34 | 33 | 270 | 3 | 2,12 |
| `Heinrichi` | 48 | 51 | 1042 | 17 | 4,84 |

function is needed namely `sourceOf`, which maps a variant reading to the manuscript that is the source of that variant reading. The theory `Ttask` consists of a single constraint; it states that a manuscript that is not the source of its own variant reading must have a parent with the same variant reading.

The remainder is procedural code. The procedure `process` uses concatenation to construct two filenames from the name of the tradition and passes these file names to the `processFiles` procedure; the ".dot" file contains the stemma data; the ".json" file the feature data. The `readStemma` procedure (code omitted) returns the input structure describing the stemma as well as the number of manuscripts (nodes) and parent-child pairs (edges). The `processFeatures` procedure (code omitted) iterates over the features in the file. For each feature, it constructs a feature structure by extending the stemma structure with the feature specific data. It then calls the `check` procedure. This procedure extends the feature structure with the symbols from the `Vtask` vocabulary (`setvocabulary(feature,Vtask)`) and then checks the colour-connectedness of the feature (`sat(Ttask,feature)`). The yes/no result is returned to the `processFiles` procedure which collects and returns the global results: number of consistent (positive) features, total number of features and time. The `processFiles` procedure prints these global data and returns to `main`.

As can be seen in the `main()` procedure, we used the code to perform consistency checking for the features of 5 traditions; two of them, `Sermon 158` and `Florilegium` are real traditions, with stemmata that have been constructed according to current philological best practice; the other three are artificial traditions, produced under test conditions by volunteers for the purposes of empirical research into stemmatological methods. We received the data from Tara Andrews. A website where such stemma data can be found is `http://byzantini.st/stemmaweb/`. Some information about the stemma we used is given in Table 3.1.

The IDP program determines consistency for all features and datasets in a matter of seconds[6]:

```
> main()
Processing besoin.
Stemma has 13 nodes and 13 edges.
Found 26 positive out of 44 groupings in 0 sec.
Processing parzival.
Stemma has 21 nodes and 20 edges.
Found 45 positive out of 122 groupings in 1 sec.
Processing florilegium.
Stemma has 22 nodes and 21 edges.
Found 431 positive out of 547 groupings in 2 sec.
Processing sermon158.
Stemma has 34 nodes and 33 edges.
Found 64 positive out of 270 groupings in 2 sec.
Processing heinrichi.
Stemma has 48 nodes and 51 edges.
Found 1 positive out of 1042 groupings in 12 sec.
```

Our largest benchmark is the `Heinrichi` dataset [Roos and Heikkilä, 2009]. This stemma about old Finnish texts includes 48 manuscripts, 51 `copiedBy` tuples and information about 1042 features. Processing all features takes 12 seconds with the IDP system, while it took 25 seconds with the original procedural code.

One can observe that rather few features are consistent with the stemma. This raises the question what is the minimal number of sources needed to explain the data. To solve that inference task, it suffices to replace the vocabulary extension `Vtask` and the theory `Ttask` in the knowledge base and to introduce the term to be minimized. As core procedure, `Check` is replaced by `minSources` and the processing of results has to be adjusted. The most relevant new parts are shown in Listing 3.2. The `IsSource` predicate is defined as manuscripts that do not have a parent with the same variant reading.

Listing 3.2: Minimize the number of sources.

```
/* ------ new parts of Knowledge base --------------- */
vocabulary Vms {
  includes V;
  pred IsSource[Manuscript];
}
theory Tms over Vms is {
   definition  {
```

---

[6] Using an Intel$^R$ Core$^{TM}$2 Duo CPU at 3.00GHz with 3.7 GB of RAM running Ubuntu and IDP 3.2.0 with the options stdoptions.groundwithbounds = false (disabling bounded grounding) and stdoptions.liftedunitpropagation = false (disabling lifted unit propagation).

```
      ! x : IsSource(x) <- ~? y : CopiedBy(y,x) &
                                  VariantReading(y) = VariantReading(x).
   }
}
term NbOfSources over Vms is #{ x : IsSource(x)};

/* ------ the core procedure ------------------- */
procedure minSources(feature) {
   setvocabulary(feature,Vms);
   return minimize(Tms,feature,NbOfSources)[1];
}
```

Although this is a minimization problem, processing the traditions is still a matter of seconds, except for the larger `Heinrichi` dataset which now requires about 5 minutes to process its 1042 features.

Other variations are of interest to the researchers. One variation, mentioned in [Andrews et al., 2012], considers the possibility that the scribe has copied from an older ancestor than the direct parent, thus reintroducing a variant. Playing with the relative penalty of introducing a new variant versus reverting to an older variant, one can obtain various explanations of interest to the stemmatologist. All these can be achieved with modifying a handful of lines in the model. Interesting about the above variant is that it uses a predicate `IndirectAncestor` that is defined in terms of the stemma data, so it can be computed once and reused when processing each of the features. The tight integration of the knowledge base with the procedural code makes this very easy as illustrated in Listing 3.3. The procedure `readStemma`, which constructs the stemma structure from the inputfile, is extended with the call `modelexpand(T,stemma)[1]`. The resulting model is the stemma structure extended with the true `IndirectAncestor` atoms. This structure, together with the other outputs of `readStemma`, is returned to the procedure `processFiles` which uses it to handle the features one by one.

Listing 3.3: Materializing a definition once and using it many times.

```
vocabulary V is {
   /* ... as in Listing 5 ... */
   pred IndirectAncestor[Manuscript,Manuscript];
}

theory T over V is {
   definition {
      ! x y : IndirectAncestor(x,y) <-
```

```
                ? z : CopiedBy(x,z) & IndirectAncestor(z,y);
    ! x y : IndirectAncestor(x,y) <-
                ? z : CopiedBy(x,z) & CopiedBy(z,y);
  }
}

procedure readStemma(stemmafilename) {
  local stemma = newstructure(V,"stemma");
  /* ... reading the stemma data ... */
  return modelexpand(T,stemma)[1], #nodes, #edges;
}
```

## 3.6 Related Work

Within several domains, research is targeting expressive specification languages and (to a lesser extent) multiple inference techniques within one language. While we do not aim at an extensive survey of related languages (e.g., [Marriott et al., 2008] has a section with such a survey), we do compare with a couple of them.

The B language [Abrial, 1996], a successor of Z, is a formal specification language developed specifically for the generation of procedural code. It is based on first-order logic and set theory, and supports quantification over sets. Event-B is a variant for the specification of event-based applications. The language Zinc, developed by Marriott et al. [Marriott et al., 2008], is a successor of OPL and intended as a specification language for constraint programming applications (mainly CSP and COP solving). It is based on first-order logic, type theory and constraint programming languages. Within ASP, a number of related languages, originating from logic programs, are being developed, such as Gringo [Gebser et al., 2009a] and DLV [Leone et al., 2006]. They support definitional knowledge and default reasoning. Implementations exist for inference techniques like stable model generation (related to model expansion), visualization, optimization and debugging. A comparison of ASP and FO(*ID*) can be found in [Denecker et al., 2012]. The language of the Alloy [Jackson, 2002] system is basically first-order logic extended with relational algebra operators, but with an object-oriented syntax, making it more natural to express knowledge from application domains centered around agents and their roles, e.g., security analysis.

The following are alternative approaches to model expansion (or closely related inference tasks). The solver-independent CP language Zinc [Marriott et al., 2008] is grounded to the language MiniZinc [Nethercote et al., 2007], supported by a range of search algorithms using various paradigms, as can be seen on `www.minizinc.org/challenge2012/results2012.html`. In the context of CASP, several systems ground to ASP extended with constraint atoms, such as Clingcon [Ostrowski and Schaub, 2012] and EZ(CSP) [Balduccini, 2011]. For search, Clingcon combines the ASP solver Clasp [Gebser et al., 2012b] with the CSP solver Gecode [Gecode Team, 2013], while EZ(CSP) combines an off-the-shelf ASP solver with an off-the-shelf CLP-Prolog system. The prototype CASP solver Inca [Drescher and Walsh, 2011a] searches for answer sets of a ground CASP program by applying Lazy Clause Generation (LCG) for arithmetic and all-different constraints. As opposed to extending the search algorithm, a different approach is to transform a CASP program to a pure ASP program [Drescher and Walsh, 2011b], afterwards applying any off-the-shelf ASP solver. CASP languages generally only allow a restricted set of expressions to occur in constraint atoms and impose conditions on where constraint atoms can occur. For example, none of the languages allows general atoms $P(\overline{c})$ with $P$ an uninterpreted predicate symbol. One exception is $\mathcal{AC}(\mathcal{C})$, a language aimed at integrating ASP and Constraint Logic Programming [Mellarkod et al., 2008]. As shown in [Lierler, 2012], the language captures the languages of both Clingcon and EZ(CSP); however, only subsets of the language are implemented [Gelfond et al., 2008].

## 3.7 Conclusion

Kowalski's 1974 paper laid the foundations for the field of Logic Programming, by giving the Horn-clause subset of predicate logic a procedural interpretation to use it for programming. More recently, progress in automated reasoning in fields such as SAT and CP made the exploration possible of more pure forms of declarative programming, gradually moving from declarative programming to declarative modeling, in which the user only has to care about the problem specification.

In this chapter, we took this development one step further and presented the knowledge base system IDP, in which knowledge is separated from computation. The knowledge representation language is both natural and extensible, cleanly integrating first-order logic with definitions, aggregates, partial functions and a type systems, and how to integrate it with a procedural language. IDP provides a range of inference engines and functionalities for tasks encountered often in practice.

From an application perspective, we presented the use of the various components of $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ and showed detailed specifications for some problems encountered by researchers in the field of stemmatology. $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ specifications proved to be of invaluable help for researchers trying to cope with stemma that go beyond tree structures [Andrews et al., 2012]. We obtained a specification that not only correctly handles arbitrary directed acyclic graphs, but also achieved better performance than the original (incomplete) procedural code. In addition, we gave an overview of some modeling patterns and of tools that were developed to support using IDP as a software engineering framework, such as declarative visualization and debugging. Various other applications were outlined in which IDP was used in different (research) contexts in a satisfactory way, showing good performance, decreased development time and/or good usability.

The results obtained from the various applications using IDP indicate that the system is coming of age. It was already known from the ASP-competitions that it compares pretty well with ASP systems in terms of performance [Denecker et al., 2009, Calimeri et al., 2011]. In contrast to ASP, which relies on the stable semantics [Gelfond and Lifschitz, 1988], it is based on first-order logic. The informal semantics of FO's connectives and of the novel language constructs is clear and easy to understand. This probably makes it easier for newcomers to start modeling. The core of an $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ specification consists on the one hand of formulas in first-order logic, which act as constraints, and on the other hand of definitions, which are close to the rules of traditional logic programs. What distinguishes $\mathrm{FO}(\cdot)$ from traditional logic programming is the use of non-Herbrand interpretations and correspondingly, the lack of constructor functions. This often leads to a simpler data representation and gives rise to elegant formulations. On the other hand, there are cases where the rich data structures that arise in Herbrand interpretations (compound terms, lists, trees, ...) are useful too. With the addition of constructor functions, they can already be represented, although support in the inference engines is still in the early stages. Another distinction is that the IDP framework offers other forms of inference, most notably deduction and propagation. A feature of the IDP system is the integration of procedures in $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ specification [De Pooter et al., 2011] and the clean separation between declarative and procedural components. As we illustrated in the stemmatology application, this allows a user to develop a whole workflow in an $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ specification.

# 4

# Model Expansion and Optimization Inference

Model generation is a widely used problem-solving paradigm. A problem is specified as a theory in a rich, declarative logic in such a way that models of the theory represent solutions to the problem. A closely related paradigm is bounded Model Expansion (MX). Here, a partial input structure interpreting a finite, known domain is expanded into a total structure that satisfies a given theory. In practice, these tasks are often complemented by an optimization function, to express that some solutions are preferred over others, which we referred to as *optimization* inference. These paradigms are studied in fields such as CP, MIP, ASP and KR.

A state-of-the-art approach is to reduce the input theory, formulated in an expressive logic, to a theory in a fragment of the language supported by some search algorithm, while preserving a suitable form of equivalence. Afterwards, the search algorithm is applied to effectively search for (optimal) models of the theory. For example bounded model expansion for the language FO($\cdot$) can be achieved by reducing FO($\cdot$)-theories to CNF and apply a SAT-solver. We refer to the former reduction process as *grounding* and the latter as *search*. With a slight abuse of notation, grounding is also used to refer to the outcome of the reduction process, the ground theory. The two-phase approach is commonly called *ground-and-solve*.

Traditionally, search algorithms have been thought of in terms of heuristics, propagation and (possibly) learning. With the advent of search engines for non-ground languages, another characteristic came (more) into the spotlight: the *size* of the ground theory. Naturally, it was already important, but it was less generally studied: SAT and MIP applications often use custom encoding scripts, in CP a straightforward grounding is usually applied. However, when considering non-ground languages, the blowup caused by grounding the input theory becomes an important issue as users turn to applications with large domains and complex constraints. Indeed, the size of the theory increases polynomially with the size of the domain and exponentially with the nesting depth of quantified variables. For example for a formula $\forall \overline{x} \in \overline{\tau} : \phi(x)$ or $\exists x : \phi(x)$, the grounding over a structure $\mathcal{I}$ contains $|\tau_1^{\mathcal{I}}| \times \ldots \times |\tau_n^{\mathcal{I}}|$ instantiations of $\phi(x)$. This blowup increases both the grounding and solving time. The effective handling of quantifiers in search problems is an important, unsolved problem, for example also studied in [Lefèvre and Nicolas, 2009] in the field of ASP and in [Ge and de Moura, 2009] in the field of SAT Modulo Theories (SMT).

In the next four chapters, we present approaches to address this problem in various ways. In this chapter, we present the grounding approach used in the optimization engine in IDP and various preprocessing techniques to reduce the size of the grounding. We show that allowing both interpreted and uninterpreted function symbols to occur in the grounding results in much smaller groundings. In the following chapter, we then develop the solving approach used in IDP, a search algorithm for the resulting ground $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$-theories that combines techniques from SAT (such as learning), ASP (propagation over rules), SMT (solver architecture) and CP (propagation over more complex constraints). The resulting algorithm is a novel approach to optimization for rich KR languages and fits into ongoing work in the fields of CP and ASP. The algorithm also demonstrates how preserving the structure of the problem longer (instead of, e.g., translating into CNF) has beneficial effects on search. Naturally, a modeler is not required to use function symbols in her specifications. For that reason, we show in Chapter 6 how deduction can be used to automatically detect functional dependencies and how they can be exploited to improve the inference engine further. In Chapter 7, we develop a *lazy* model expansion algorithm that improves the approach further by interleaving both phases.

The main results of this chapter have been published in [De Cat et al., 2013a] and [De Cat et al., 2013b].

The chapter is organized as follows. We elaborate on the problem of quantifier handling for optimization in Section 4.1, followed by a detailed discussion of the grounding algorithms in Section 4.2. In Section 4.3, we construct the complete inference engine, integrating important pre- and postprocessing techniques. Related work is presented in Section 4.4 and Section 4.5 concludes the chapter. A detailed treatment of the search algorithm itself is left to the following chapter, in which we also present a detailed experimental evaluation of both the search algorithm and the complete model expansion and optimization inference engine.

**Running Example.** In the following chapters, we use the well-known 2-D *packing* problem for the purpose of illustration [Lodi et al., 2002]. The problem consists of a set $Sq$ of rectangular objects, each with a fixed height and breadth known in advance, and a rectangular area with known width $w$ and breadth $b$, also integer numbers. The task is to place all objects at discrete coordinates within the rectangular area such that they are all completely inside and such that no two objects overlap each other. An example is shown in Figure 4.1 and one of its solutions in Figure 4.2.

The packing problem and its generalizations (higher dimensions, non-integer sizes or positions, non-rectangular objects, etc.) occur frequently in practice. One example is the design of the floor plan of, e.g., a hospital, where it has to be decided how to map a number of rooms, with different functions and sizes, on a floor. Another example is in the logistics sector, where better packing and stacking of luggage, containers, etc., results in less wasted space in transport vehicles. Packing, even in its simplest form, is a *hard* problem and is still extensively studied in various fields. Here, we do not aim at specifically improving on the state-of-the-art in solving packing problems, but we use it as an illustration of the presented ideas.

As running example, we use the following variation of the 2-D square-packing problem. The standard formulation (fit a number of square boxes within a rectangular area) is extended to illustrate more of $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$'s language features:

- Not all boxes have to be placed within the target area, but as many as possible.
- One of the largest boxes should be placed at the root of the coordinate system.[1]
- The user is only interested in the position of the boxes (not in the interpretation of any helper symbols).

The complete specification then consists of the vocabulary $\Sigma$ consisting of the types $id[id]$ (the square identifiers) and $nb[nb]$ (the relevant coordinate num-

---

[1] This acts as a very simple symmetry breaking formula.

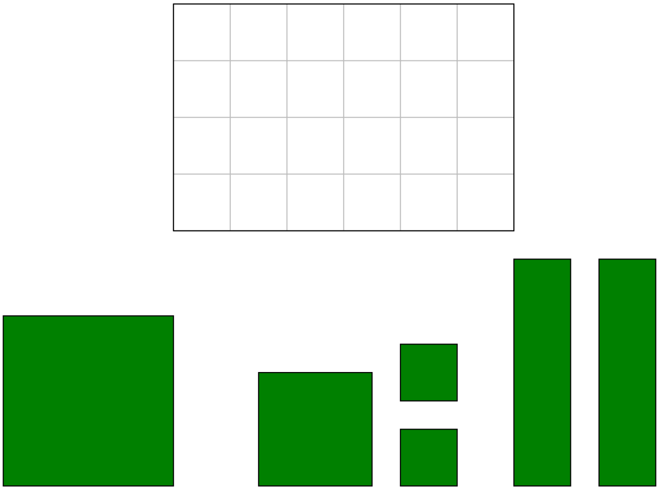


Figure 4.1: An example of a packing problem.

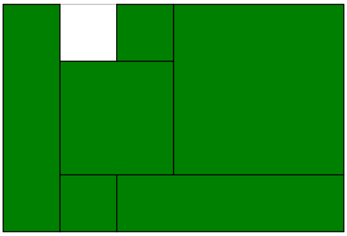


Figure 4.2: A solution of the packing problem in Figure 4.1.

bers), the predicate symbols $leftOf[id, id]$, $below[id, id]$ and $noOverLap[id, id]$ and the function symbols $largest[\mapsto id]$, $width[\mapsto nb]$ and $breadth[\mapsto nb]$ (of the target area), $size[id \mapsto nb]$, $xpos[id \mapsto nb]$ and $ypos[id \mapsto nb]$. The latter two functions are partial and they make up the output vocabulary $\Sigma_{out}$ (together with $id$ and $nb$). Theory $\mathcal{T}$ consists of the following sentences (types are omitted for readability):[2]

$$\forall id : 0 \leq xpos(id) \leq width - size(id) \land 0 \leq ypos(id) \leq breadth - size(id) \; (1)$$

$$\forall id_1 \; id_2 : noOverlap(id_1, id_2) \quad (2)$$

$$\left\{ \begin{array}{ll} \forall id_1 \; id_2 : leftOf(id_1, id_2) & \leftarrow xpos(id_1) + size(id_1) \leq xpos(id_2) \quad (3) \\ \forall id_1 \; id_2 : below(id_1, id_2) & \leftarrow ypos(id_1) + size(id_1) \leq ypos(id_2) \quad (4) \\ \forall id : noOverlap(id, id) & \leftarrow \top \quad (5a) \\ \forall id_1 \; id_2 : noOverlap(id_1, id_2) & \leftarrow leftOf(id_1, id_2) \lor leftOf(id_2, id_1) \\ & \qquad \lor below(id_1, id_2) \lor below(id_2, id_1) \; (5b) \end{array} \right\}$$

$$\forall id : denotes(xpos(id)) \Leftrightarrow denotes(ypos(id)) \qquad (6)$$

$$xpos(largest) = min[\mapsto nb] \land ypos(largest) = min[\mapsto nb] \qquad (7)$$

$$\left\{ \; \forall id_1 : largest = id_1 \leftarrow \forall id_2 : id_1 \neq id_2 \Rightarrow size(id_1) \geq size(id_2) \quad (8) \; \right\}$$

Sentence (1) expresses that every box should lie completely within the given area. Sentence (2) that boxes should not overlap, with not overlapping defined in the subsequent definition as boxes that are either the same or where one box is completely to the left of or below another box. The last two sentences then express the symmetry breaker.

The maximization term $c$ is $\#(\{id : denotes(xpos(id))\})$.

In Appendix B.1, an IDP specification of this problem is presented.

## 4.1 Optimization Inference

Recall from Chapter 3, optimization inference **optimize**$\langle \mathcal{T}, \mathcal{I}, c, \Sigma_{out} \rangle$ takes as input a theory $\mathcal{T}$, a structure $\mathcal{I}$ and a *cost* term $c$, all over the same vocabulary $\Sigma$, and an *output* vocabulary $\Sigma_{out} \subseteq V$. Solutions are then $\Sigma_{out}$-structures for which a $\Sigma$-expansion $\mathcal{J}$ exists that models $\mathcal{T}$ and expands $\mathcal{I}$ and such that $c$ is minimal in $\mathcal{J}$.

This inference captures Herbrand model generation and (bounded) model expansion, both of which were proposed as logic-based methods for constraint solving, in [East and Truszczyński, 2006] and [Mitchell and Ternovska, 2005], respectively.

Traditionally, model expansion algorithms (and thus also optimization algorithms) for rich (quantified) logics focused on predicate symbols. Function symbols were not allowed or dealt with by replacing them by their graph, as shown in the following example.

---

[2] Recall, $denotes(t)$ is a shorthand for $\exists x \in type(t) : t = x$.

**Example 4.1.1.** Consider our running example. Traditionally, a rule like rule (3)

$$\forall id_1\ id_2 : leftOf(id_1, id_2) \leftarrow pos_x(id_1) + size(id_1) \leq pos_x(id_2)$$

would be preprocessed before grounding by replacing the functions with their graph representation. This results in the rule

$$\forall id_1\ id_2 : leftOf(id_1, id_2) \leftarrow \exists x_1\ s_1\ x_2 : pos_x(id_1, x_1) \land size(id_1, s_1) \\ \land pos_x(id_2, x_2) \land x_1 + s_1 \leq x_2.$$

The drawback is already clear from this example: the quantifier depth has increased from two to five and two quantifications have been introduced over the coordinate type *nb*, which typically has a much larger interpretation than the other types involved.

The result of this replacement is, in effect, a theory that is untractable to ground for practical applications.

Recently, research has been done in ASP to incorporate techniques from CP, giving rise to the field of ASP modulo CSP (CASP) [Ostrowski and Schaub, 2012]. In CASP, the ASP language is extended with *constraint atoms*, atoms that stand for the constraints of a CSP problem [Lierler, 2012, Gebser et al., 2009c], and can, for example, contain function symbols. Second, search algorithms have been developed that allow ground constraint atoms (instead of only propositional atoms) in the input. This gives rise to more compact groundings that often also yield better propagation. Among those next generation systems are the systems Clingcon [Ostrowski and Schaub, 2012], EZ(CSP) [Balduccini, 2011], Mingo [Liu et al., 2012] and Inca [Drescher and Walsh, 2011a].

The work in this chapter fits in this line of work. We show that for $\text{FO}(\cdot)^{\text{IDP}}$, allowing the grounding to contain function terms in fact produces a form of "constraint atoms". In the above example, $pos_x(id_1) + size(id_1) \leq pos_x(id_2)$ is such an atom, for which efficient propagation techniques exist in the field of CP. We present a model expansion algorithm for $\text{FO}(\cdot)^{\text{IDP}}$ that exploits this idea. It consists of (i) an algorithm to ground $\text{FO}(\cdot)^{\text{IDP}}$ theories without eliminating all function symbols from the grounding and (ii) a search algorithm for general, ground $\text{FO}(\cdot)^{\text{IDP}}$. As different search algorithms often support different sets of function symbols, the grounding algorithm is *parametrized* by the set of functions allowed to occur in the grounding. The algorithms are implemented in the IDP system. The search algorithm itself is implemented as the separate solver MINISAT(ID), which is also used as solver in ASP and MiniZinc portfolios.

We impose the following restrictions on our **optimize**$\langle \mathcal{T}, \mathcal{I}, c, \Sigma_{out} \rangle$ task:

- Both $\mathcal{T}$ and $\mathcal{I}$ are well-typed.
- $\mathcal{I}$ interprets all types and these are all totally ordered.
- Nested aggregate terms occurring in a definition cannot contain any symbols defined in that definition.[3]

For now, we assume the theory contains no function definitions. In Section 6.3.3, we discuss how a theory with defined functions can be reduced to a theory without defined functions without blowing up the grounding.

## 4.2 Grounding to Parametrized Ground FO$(\cdot)^{\text{IDP}}$

This section describes an algorithm to construct the grounding of a theory $\mathcal{T}_{in}$ and a term $c_{in}$ over $\Sigma$ in the context of a three-valued (consistent) interpretation $\mathcal{I}_{in}$.[4] The algorithm transforms $\mathcal{T}_{in}$ and $c_{in}$ to a $\{\Sigma,\mathcal{I}\}$-equivalent ground —quantifier-free— theory $\mathcal{T}_g$ and term $c_g$ (the grounding) and a "mapping" theory $\mathcal{T}_m$ consisting of explicit definitions for symbols of $\Sigma$ that were eliminated from the input.

The grounding algorithm takes as parameter a set SuppF of "residual" function symbols, the function symbols allowed to occur in $\mathcal{T}_g$. Intuitively, these are the function symbols "supported" by the target search algorithm and which can hence occur in the grounding. In our algorithm, functions $f/n$ not in SuppF are replaced by their "graph" predicate symbol $G_f/n+1$. If SuppF is empty, then all atoms in the grounding will be domain atoms; by translating these into propositional symbols, such a theory can be mapped into an "equivalent" propositional theory.

We assume the optimization term $c_{in}$ is a constant. This is without loss of generality: any non-constant term $c_{in}$ can be handled by replacing $c_{in}$ by a new constant $c'[\mapsto \tau(c_{in})]$ and adding the sentence $c' = c_{in}$ to $\mathcal{T}_{in}$. Naturally, the optimization constant has to be a supported function.

We describe the grounding process as two stratified sequences of $\{\Sigma,\mathcal{I}\}$-equivalence preserving rewrite rules, rewriting the theories $\mathcal{T}_g$ and $\mathcal{T}_m$. Theory $\mathcal{T}_g$ is initialized as $\mathcal{T}_{in}$, $\mathcal{T}_m$ as the empty set. The rewrite rules operate on $\mathcal{T}_g$, substituting expressions by simpler ones, and sometimes introduce new definitions to $\mathcal{T}_g$ or $\mathcal{T}_m$.

---

[3] Although IDP does not impose this restriction, we introduce it here to simplify the presentation.

[4] Checking whether a structure is inconsistent can be expensive. In Section 4.3.1, we show how four-valued structures are handled efficiently.

Rewrite rules are denoted as, e.g., $\neg\neg\varphi \rightarrowtail \varphi$, which means that the rule replaces occurrences of $\neg\neg\varphi$ in $\mathcal{T}_g$ by $\varphi$.

### 4.2.1 Phase 1: Simplifying the Syntax

The first phase consists of iterated rewriting of $\mathcal{T}_g$ to obtain a normal form suitable for the effective grounding in phase 2. The phase serves two purposes. First, the resulting theory $\mathcal{T}_g$ will only contain function symbols that are part of SuppF. Second, the theory is normalized such that the subsequent grounding will be smaller, faster and introduce less additional symbols (which is well-known to have a detrimental impact on search). Considering the introduction of additional symbols, a rule-of-thumb is that a new symbol is introduced for each "context switch" when descending through the parse-tree, such as a conjunction nested below a disjunction or a product below an addition.

We first introduce rules for the first property, the only one required for correctness of the grounding algorithm. Afterwards, we present the rules to normalize the theory further.

#### Unnesting and Graphing Function Symbols

To replace function symbols in the theory, we define the transformations UNNEST and $G$, which both take as input a theory $\mathcal{T}_g$ and a set of supported functions SuppF. The former flattens applications of functions symbols $f$, the latter effectively replaces $f$ by its graph equivalent.

The aim of unnesting is to obtain a theory in which functions $f[\tau_1, \dots, \tau_{n-1} \mapsto \tau_n] \notin$ SuppF only occur as direct subterms of atoms $f(\bar{t}) = x$ or $x = f(\bar{t})$, with $x$ a variable or a domain element. To define unnesting, we first define the concept of *closest formula* of a term occurrence.

**Definition 4.2.1** (closest formula). For an occurrence of a term $t$ in a formula $\varphi$, the *closest formula* of $t$ is:

- an atom $a$ in $\varphi$, if $t$ is a direct subterm of $a$ and $a$ is not the head of a rule,
- the condition $\psi$ of a set in $\varphi$, if $t$ is the term of the set,
- the body of a rule in $\varphi$, if $t$ is a direct subterm of the head,
- (recursive case) the closest formula of $t'$ of which $t$ is a direct subterm.

The rule UNNEST is then basically the rewrite rule

$$\varphi[f(\bar{t})] \rightarrowtail \exists x \in \tau_n : f(\bar{t}) = x \wedge \varphi[f(\bar{t})/x]$$

with $f$ the function symbol $f[\tau_1, \ldots, \tau_{n-1} \mapsto \tau_n] \notin$ SuppF and $\varphi$ the closest formula of the relevant occurrence of $f(\bar{t})$. There are some special cases:

- The rule is not applied if $f(\bar{t})$ occurs in atoms $f(\bar{t}) = x$ or $x = f(\bar{t})$.
- It is also not applied if $f(\bar{t})$ occurs as the head $f(\bar{t}) = t'$ of a rule in a definition. As $f$ is then the defined symbol, we cannot just move $f(\bar{t})$ to the body. Instead, if $t'$ is not a variable, we replace $t'$ by $x$ and replace the body $\varphi$ of the rule by $\exists x \in \tau' : t' = x \wedge \varphi$, with $\tau'$ the type of $t'$.
- If the closest formula is the body of an aggregate set $\{\bar{y} : \psi : f(\bar{t})\}$, the set is rewritten as $\{\bar{y} :: x : f(\bar{t}) = x \wedge \psi : x\}$ instead.

After executing UNNEST until fixpoint, all occurrences of function symbols $f \notin$ SuppF are top symbols in equalities of the form $f(\bar{t}) = t$ or $t = f(\bar{t})$, with $\bar{t}$ and $t$ terms. If SuppF is empty, all terms $t_1, \ldots, t_n$ and $t$ are either domain elements or variables.

On the obtained theory, we apply GRAPH, which consists of the rules $f(\bar{t}) = x \rightarrowtail G_f(\bar{t} :: x)$ and $x = f(\bar{t}) \rightarrowtail G_f(\bar{t} :: x)$ and the interpretation of $G_f$ in $\mathcal{I}_{in}$ is set to the interpretation of $f$. In addition, the following sentences are added to $\mathcal{T}_g$ to guarantee $G_f$ behaves as a function (the second only if $f$ is total):

$$\forall \bar{x} \in \tau_1 \ldots \tau_{n-1} : \exists_{\leq 1} y \in \tau_n : G_f(\bar{x} :: y)$$

$$\forall \bar{x} \in \tau_1 \ldots \tau_{n-1} : \exists_{\geq 1} y \in \tau_n : G_f(\bar{x} :: y)$$

Last, the following "output" definition is added to $\mathcal{T}_m$ to handle consistency between $f$ and $G_f$

$$\{\forall \bar{x} :: y \in \tau_1, \ldots, \tau_n : f(\bar{x}) = y \leftarrow G_f(\bar{x} :: y)\}.$$

Naturally, the sentences and definition are only added once for any graphed function symbol $f$.

**Proposition 4.2.2.** *For a theory $\mathcal{T}$ and set of function symbols* SuppF*, the transformations* UNNEST *and* GRAPH *preserve models of $\mathcal{T}$.*

The proof is provided in Appendix A.1.

**Example 4.2.3.** Consider part of sentence (1) of our packing theory

$$\forall id : 0 \leq xpos(id) \leq width - size(id).$$

Assuming that SuppF contains only arithmetic operators, applying UNNEST results in the sentence

$$\forall id : \exists x\ s\ w \in nb^3 : xpos(id) = x \land size(id) = s \land width = w \land 0 \leq x \leq w - s.$$

Graphing the function terms then results in

$$\forall id : \exists x\ s\ w \in nb^3 : G_{xpos}(id, x) \land G_{size}(id, s) \land G_{width} w \land 0 \leq x \leq w - s.$$

From this example it is already clear why, in Chapter 5, we develop a search algorithm where SuppF contains all function symbols, so no additional variables have to be introduced.

We can already distinguish one case in which we do not have to unnest function terms not in SuppF. We say a term $t$ (formula $\varphi$) is *ground-evaluable* in a structure if all symbols in $t$ are two-valued. During grounding, we use the structure $\mathcal{I}_{in}$ to simplify ground-evaluable formulas and terms on-the-go. There is then no need to unnest terms $f(\bar{t})$ that are ground-evaluable in $\mathcal{I}_{in}$, which would introduce additional variables and complicate the grounding process. In the packing example, *size* is usually interpreted in $\mathcal{I}_{in}$ and as such, will not be unnested.

### Normalization

After unnesting and graphing, the following rules are applied to bring the theory in a form with a smaller grounding with less additional symbols. This is achieved among others by reducing the number of context switches and pushing quantifications lower in the parse-tree.

- PUSH-NEGATIONS applies the standard equivalence-preserving transformations to push negations down.[5]

$$\neg \bigwedge_{i \in S} \varphi_i \rightarrowtail \bigvee_{i \in S} \neg \varphi_i \qquad \neg \bigvee_{i \in S} \varphi_i \rightarrowtail \bigwedge_{i \in S} \neg \varphi_i$$

$$\neg \forall \bar{x} : \varphi \rightarrowtail \exists \bar{x} : \neg \varphi \qquad \neg \exists \bar{x} : \varphi \rightarrowtail \forall \bar{x} : \neg \varphi$$

$$\neg \neg \varphi \rightarrowtail \varphi \qquad \neg(t \sim t') \rightarrowtail t \not\sim t'$$

---

[5] Given a comparison $\sim$, we use $\not\sim$ to denote its negation.

- PUSH-QUANTIFIERS pushes down quantifiers to where they are relevant. The following rule pushes a variable $x_j$ down if it only occurs in some subformulas of a disjunction:

  $$\forall x_1 \ldots x_n : \bigvee_{\varphi \in S} \varphi \rightarrowtail \forall x_1 \ldots x_{j-1}\, x_{j+1} \ldots x_n : (\bigvee_{\varphi \in S_1} \vee (\forall x_j : \bigvee_{\varphi \in S_2}))$$

  with $S_1$ and $S_2$ the partition of $S$ such that $x_j$ does not occur in formulas in $S_1$ and occurs in all formulas in $S_2$. A similar rule is applied for all other combinations of universal/existential quantifications with conjunctive/disjunctive subformulas. For definitions, the following rule is also applied, which moves variables from the head to the body.

  $$\forall \overline{x} : head \leftarrow \varphi \rightarrowtail \forall x_1 \ldots x_{j-1}\, x_{j+1} \ldots x_n : head \leftarrow \exists x_j : \varphi$$

  Pushing $x_j$'s quantification down is only allowed if the interpretation of its type is not empty in $\mathcal{I}_{in}$. In that case, other simplifications are available (see below).

- FLATTEN combines subsequent identical operators into one:

  $$\begin{aligned} \bigwedge_{\varphi \in S} \varphi &\rightarrowtail \bigwedge_{\varphi \in S - \psi \cup S'} \varphi \qquad \textbf{if } \psi \in S \text{ and } \psi = \textstyle\bigwedge_{\varphi \in S'} \varphi \\ \bigvee_{\varphi \in S} \varphi &\rightarrowtail \bigvee_{\varphi \in S - \psi \cup S'} \varphi \qquad \textbf{if } \psi \in S \text{ and } \psi = \textstyle\bigvee_{\varphi \in S'} \varphi \\ \forall \overline{x} : \forall \overline{y} : \varphi &\rightarrowtail \forall \overline{x} :: \overline{y} : \varphi \\ \exists \overline{x} : \exists \overline{y} : \varphi &\rightarrowtail \exists \overline{x} :: \overline{y} : \varphi \\ agg(agg(S_1), agg(S_2)) &\rightarrowtail agg(S_1 \cup S_2) \end{aligned}$$

- CHECK-EXISTENCE syntactically replaces checks on whether a term denotes with the "denoting" shorthand, if the shorthand is supported by the search algorithm, in which case less grounding is necessary:[6]

  $$\exists x \in \tau : f(\bar{t}) = x \rightarrowtail denotes(f(\bar{t}))$$

  $$\forall x \in \tau : f(\bar{t}) \neq x \rightarrowtail \neg denotes(f(\bar{t}))$$

  The rule should only be applied if $\bar{t}$ does not contain $x$, otherwise it is not merely checking whether $f(\bar{t})$ has a value.

---

[6] Naturally, also for atoms $x = f(\bar{t})$.

It is easy to see that each of the above rewriting rules preserves equivalence. Additionally, they can be stratified in such a way that one set of rules is applied to fixpoint without invalidating rules applied earlier.

Most of the transformations are implemented in IDP as an operation on the parse-tree that takes as input a term, formula or theory and visits the input in a top-down, depth-first fashion, rewriting it on-the-go to satisfy the appropriate post conditions.

### 4.2.2 Phase 2: Grounding

The aim of the second phase is to produce the effective grounding, in an appropriate normal form so that relevant fragments can easily be supported by solvers. This normal form is the so-called Extended Conjunctive Normal Form (ECNF), defined as follows.

**Definition 4.2.4** (Extended CNF). An $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$-theory is in ECNF if all its sentences are either disjunctions of domain atoms $L_1 \vee \ldots \vee L_n$ or definitions where rules are of one the following forms (with all $L_i$'s domain literals and all $e_i$'s constants or domain elements):

$$P(\bar{e}) \leftarrow L_1 \wedge \ldots \wedge L_n \qquad P(\bar{e}) \leftarrow L_1 \vee \ldots \vee L_n$$

$$P(\bar{e}) \leftarrow Q(\bar{e}') \qquad P(\bar{e}) \leftarrow f(\bar{e}) \sim e_0$$

$$P(\bar{e}) \leftarrow agg(\{L_1 : e_1\} \cup \cdots \cup \{L_n : e_n\}) \sim e_0$$

Any $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$-theory over a finite structure $\mathcal{I}$ can be reduced to a $\{\Sigma, \mathcal{I}\}$-equivalent-theory in ECNF.

From now on, the domains of variables are made explicit in all expressions, written as $\forall \bar{x} \in \overline{D} : \varphi$ or $\{\bar{x} \in \overline{D} : \varphi : t\}$. Initially, $\overline{D}$ is $\tau_1^{\mathcal{I}_{in}} \times \cdots \times \tau_n^{\mathcal{I}_{in}}$, where $\tau_i$ is the type of $x_i$ (recall that $\mathcal{I}_{in}$ interprets all types).

The second phase then consists of applying the following set of rewrite rules to effectively instantiate variables and introduce new symbols to obtain the above normal form. The phase terminates when no more rules are applicable. For this phase, we introduce a term **undef** which represents a non-denoting term; it is only used during grounding and will not occur in the resulting ground theory.

- INSTANTIATE, for some $\overline{d} \in \overline{D}$:

$$\forall \overline{x} \in \overline{D} : \varphi \rightarrowtail \varphi[\overline{x}/\overline{d}] \land \forall \overline{x} \in \overline{D} - \overline{d} : \varphi$$

$$\exists \overline{x} \in \overline{D} : \varphi \rightarrowtail \varphi[\overline{x}/\overline{d}] \lor \exists \overline{x} \in \overline{D} - \overline{d} : \varphi$$

$$\forall \overline{x} \in \overline{D} : head \leftarrow \varphi \rightarrowtail head[\overline{x}/\overline{d}] \leftarrow \varphi[\overline{x}/\overline{d}], \quad \forall \overline{x} \in \overline{D} - \overline{d} : head \leftarrow \varphi$$

$$\{\overline{x} \in \overline{D} : \varphi : t\} \rightarrowtail \{\varphi[\overline{x}/\overline{d}] : t[\overline{x}/\overline{d}]\} \cup \{\overline{x} \in \overline{D} - \overline{d} : \varphi : t\}$$

- INTRODUCE TSEITIN $\varphi \rightarrowtail T_\varphi$, where $\varphi$ is an occurrence of a formula without free variables in $\mathcal{T}_g$ and $T_\varphi$ is a new propositional symbol. In addition, a rule is added that defines $T_\varphi$: if $\varphi$ occurs in a definition $\Delta$, the rule $T_\varphi \leftarrow \varphi$ is added to $\Delta$, otherwise, the singleton definition $\{T_\varphi \leftarrow \varphi\}$ is added to $\mathcal{T}_g$.

  The rule is not applied if $\varphi$ is a literal, a sentence or a rule body. Recall that a sufficient condition to preserve model equivalence is to not introduce Tseitins under negation, which is guaranteed by the rule PUSH-NEGATIONS applied in the previous phase and by restricting the nesting of aggregate terms over inductively defined symbols.

- INTRODUCE TERM $t \rightarrowtail c_t$, where $t$ is an occurrence of a term without free variables in $\mathcal{T}_g$ and $c_t$ is a newly introduced constant $c_t[\mapsto type(t)]$. The sentence $t = c_t$ is added to $\mathcal{T}_g$. Informally, the idea is to reduce the nesting depth of terms that are supported by the search algorithm to obtain ECNF expressions. The rule is not applied if $t$ is a domain element, a variable or a constant, or if $t$ occurs as $t \sim c'$ or $c' \sim t$.

- EVALUATE domain terms and atoms in the structure

$$f(\overline{d}) \rightarrowtail d' \qquad \textbf{if } f_{ct}^{\mathcal{I}_{in}}(\overline{d}) = \{d'\}$$

$$f(\overline{d}) \rightarrowtail \textbf{undef} \qquad \textbf{if } f_{pt}^{\mathcal{I}_{in}}(\overline{d}) = \emptyset \textbf{ or } d_i \notin type(arg_i(f))^{\mathcal{I}_{in}}$$

$$P(\overline{d}) \rightarrowtail \top \qquad \textbf{if } P(\overline{d})^{\mathcal{I}_{in}} = \mathbf{t}$$

$$P(\overline{d}) \rightarrowtail \bot \qquad \textbf{if } P(\overline{d})^{\mathcal{I}_{in}} = \mathbf{f} \textbf{ or } d_i \notin type(arg_i(P))^{\mathcal{I}_{in}}$$

  The rules for a symbol $s$ are not applied in definitions that define $s$, as loops through the definition might be lost (which leads to inconsistencies). More specifically, if the definition is monotone, replacements with $\bot$ are allowed.

Note that application of built-in symbols such as aggregates, arithmetic and constructed types is covered by EVALUATE, as they are interpreted in $\mathcal{I}_{in}$. We add two more interesting rules related to function terms under equality (and, similarly, in inequality):

$$
\begin{array}{ll}
f(\overline{d}) = d \rightarrowtail \top & \textbf{if } d \in f_{ct}^{\mathcal{I}_{in}}(\overline{d}) \\[1em]
f(\overline{d}) = d \rightarrowtail \bot & \textbf{if } d \in f_{cf}^{\mathcal{I}_{in}}(\overline{d})
\end{array}
$$

For aggregates, a similar rule is possible, which allows us to only partially ground aggregate sets. It follows from the fact that all aggregates are composable. For any aggregate function, we can straightforwardly derive minimum and maximum bounds given a set $S$ (even a quantified one). For example for an expression $sum(\{\overline{x} \in \overline{D} : \varphi : t\})$, the minimum bound $min_{bound}$ is $min[\mapsto type(t)]^{\mathcal{I}} \times |\overline{D}|$ and the maximum bound $max_{bound}$ is $max[\mapsto type(t)]^{\mathcal{I}} \times |\overline{D}|$. If we then have an atom $agg(S \cup \{\top : d\} \cup S') \sim d'$ in $\mathcal{T}_g$, simplifications are possible in various cases (not enumerated here). One example is in case of a sum aggregate with $>$ comparison operator: if $d + min(min_{bound}, 0) > d'$, then the atom evaluates to true.

- SPLIT *conjunctive sentences:* $\varphi_1 \wedge \ldots \wedge \varphi_n \rightarrowtail \varphi_1, \ldots, \varphi_n$.
- SIMPLIFY
  Basic simplifications

$$
\begin{array}{lr}
\neg\top \rightarrowtail \bot & \\
\neg\bot \rightarrowtail \top & \\
\varphi_1 \vee \ldots \vee \varphi_n \rightarrowtail \top & \textbf{if } \varphi_j = \top \\
\varphi_1 \wedge \ldots \wedge \varphi_n \rightarrowtail \bot & \textbf{if } \varphi_j = \bot \\
\varphi_1 \vee \ldots \vee \varphi_n \rightarrowtail \varphi_1 \vee \ldots \vee \varphi_{j-1} \vee \varphi_{j+1} \vee \ldots \vee \varphi_n & \textbf{if } \varphi_j = \bot \\
\varphi_1 \wedge \ldots \wedge \varphi_n \rightarrowtail \varphi_1 \wedge \ldots \wedge \varphi_{j-1} \wedge \varphi_{j+1} \wedge \ldots \wedge \varphi_n & \textbf{if } \varphi_j = \top \\
a \leftarrow \top,\ a \leftarrow \varphi \rightarrowtail a \leftarrow \top & \\
a \leftarrow \bot,\ a \leftarrow \varphi \rightarrowtail a \leftarrow \varphi &
\end{array}
$$

Quantification simplifications

$$
\begin{array}{lll}
\forall \overline{x} \in \overline{D} : \varphi & \rightarrowtail \top & \textbf{if } \overline{D} = \emptyset \text{ or } \varphi = \top \\
\exists \overline{x} \in \overline{D} : \varphi & \rightarrowtail \bot & \textbf{if } \overline{D} = \emptyset \text{ or } \varphi = \bot \\
\{\overline{x} \in \overline{D} : \varphi : t\} & \rightarrowtail \{\bot : \textbf{undef}\} & \textbf{if } \overline{D} = \emptyset \text{ or } \varphi = \bot
\end{array}
$$

Partial function simplifications

$$
\begin{array}{ll}
f(t_1, \ldots, t_n) \rightarrowtail \textbf{undef} & \textbf{if } t_j = \textbf{undef} \\[1em]
P(t_1, \ldots, t_n) \rightarrowtail \bot & \textbf{if } t_j = \textbf{undef}
\end{array}
$$

Aggregate simplifications

$$agg(S \cup \{\bot : t\}) \rightarrowtail agg(S)$$

$$agg(S \cup \{\varphi : d\} \cup \{\varphi : d'\}) \rightarrowtail agg(S \cup \{\varphi : agg(\{\top : d\} \cup \{\top : d'\})^{\mathcal{I}_{in}}\})$$

After application of the above rewrite rules, we obtain a theory in ECNF.

**Example 4.2.5.** Consider grounding our packing theory from the running example for the partial input structure $\{width = 1000, breadth = 1000,\ size(1) \rightarrow 250,\ size(2) \rightarrow 500,\ size(3) \rightarrow 750,\ size(4) \rightarrow 300\}$.

$$\begin{array}{l}
0 \le xpos(1) \le 750,\ 0 \le ypos(1) \le 750 \\
0 \le xpos(2) \le 500,\ 0 \le ypos(2) \le 500 \\
0 \le xpos(3) \le 250,\ 0 \le ypos(3) \le 250 \\
0 \le xpos(4) \le 700,\ 0 \le ypos(4) \le 700 \\
noOverlap(1,1) \\
\quad \vdots \\
noOverlap(4,4) \\
\left\{ \begin{array}{lll}
leftOf(1,1) & \leftarrow & xpos(1) + 250 \le xpos(1) \\
 & \vdots & \\
leftOf(4,4) & \leftarrow & xpos(4) + 300 \le xpos(4) \\
below(1,1) & \leftarrow & ypos(1) + 250 \le ypos(1) \\
 & \vdots & \\
below(4,4) & \leftarrow & ypos(4) + 300 \le ypos(4) \\
noOverlap(1,1) & & \\
noOverlap(1,2) & \leftarrow & leftOf(1,2) \lor leftOf(2,1) \\
 & & \quad \lor below(1,2) \lor below(2,1) \\
 & \vdots & \\
noOverlap(4,3) & \leftarrow & leftOf(4,3) \lor leftOf(3,4) \\
 & & \quad \lor below(4,3) \lor below(3,4) \\
noOverlap(4,4) & &
\end{array} \right\} \\
denotes(xpos(1)) \Leftrightarrow denotes(ypos(1)) \\
\qquad\qquad \vdots \\
denotes(xpos(4)) \Leftrightarrow denotes(ypos(4)) \\
xpos(largest) = 0,\ ypos(largest) = 0 \\
\{largest = 1\}
\end{array}$$

Note that our theory still contains atoms of the form $xpos(d) + 250 \leq xpos(d)$, as our algorithm contains no rules for simplifying arithmetic expressions. For example, putting the equality in canonical form would result in $250 \leq 0$, which evaluates to false.

**Theorem 4.2.6.** *For input $\mathcal{T}$, $\mathcal{I}$ and* SuppF*, let $\mathcal{T}_g$ and $\mathcal{T}_m$ be the computed theories at any time during the rewrite process. Then $\mathcal{T}$ and $\mathcal{T}_g \cup \mathcal{T}_m$ are $\{\Sigma, \mathcal{I}\}$-equivalent and $\mathcal{T}_m$ only consists of total definitions.*

*The rewrite-process terminates if all types are finite and the resulting theory $\mathcal{T}_g$ is in ECNF and contains only function symbols in* SuppF*.*

*proof-sketch.* The equivalence follows from the fact that each rewrite rule preserves $\{\Sigma, \mathcal{I}\}$-equivalence.

Theory $\mathcal{T}_m$ only consists of total definitions as only the GRAPH rule adds definitions to $\mathcal{T}_m$. For any such definition $\Delta$, no symbol defined in $\Delta$ occurs in the body of rules in $\Delta$, so $\Delta$ is total.

That the resulting theory is in ECNF if the process terminates, follows from the fact that for any theory that is not in ECNF, at least one rewrite rule is applicable.

Termination of phase 1 is straightforward. To prove termination of phase 2, it can be shown that a well-founded order exists on theories for the presented rewrite rules and that each application results in a theory ordered strictly lower. The order depends among others on the nesting depth of function symbols, the nesting and domain size of quantifications and the number of occurrences of symbols in the theory. ☐

### 4.2.3 Improved Termination

Termination is a desirable property, even more so as it implies that the grounding is finite. However, the guarantees provided by Theorem 4.2.6 are too weak for practical use: no guarantees are provided if some type has an infinite interpretation. The latter is, e.g., the case when the user uses arithmetic. Fortunately, we can slightly adapt the rewrite rules to provide the following hard guarantee: if all quantifications in $\mathcal{T}_{in}$ and all non-type, user-defined symbols are over finite types, the ground theory is finite. An advantage of this property is that its condition is easy for a user to understand and check (IDP also issues a warning if it is violated).

Using the rewrite rules as defined above, this property is not satisfied. Indeed, application of the UNNEST rule to a term $f(\bar{t})$ introduces a quantification over the output type of $f$, which is infinite for arithmetic functions and aggregates. To resolve this, we define an operation *derive-bound*, which takes as input a term $f(\bar{t})$ over a function $f[\tau_1, \dots, \tau_{n-1} \mapsto \tau_n]$ and a structure $\mathcal{I}$ and returns a (possibly new) type $\tau$. The new type contains at least all values $f(\bar{t})$ can take in any expansion of $\mathcal{I}$ (so is still correct), but can be smaller than $\tau_n^{\mathcal{I}}$. It is defined by going top-down through the parse tree of $f(\bar{t})$ and, at each step, returning the minimum and maximum bounds the term at hand can still take. These values are then combined appropriately for the arguments of built-in functions when going back up through the tree. If combination is not possible (e.g., for uninterpreted functions), the original output type is returned. A full presentation would lead us too far, but we provide an example below.

**Example 4.2.7.** Consider term $t$ of the form $\#(\{x \in \tau : P(x)\}) \times c$ and a structure $\mathcal{I}$ that interprets $\tau$ as $[1..100]$ and $type(c)$ as $[-10..10]$. The operation *derive-bound* then derives that the cardinality lies in the interval $[0, 100]$ and the bounds on $t$ are then $[-1000, 1000]$. This is a more precise bound than $\mathbb{Z}$ (the output type of product) and thus a new, finite type is created that is interpreted as $[-1000, 1000]$ that is used when unnesting $t$.

When UNNEST is applied to $f(\bar{t})$, it then assigns *derive-bound* $(f(\bar{t}),\mathcal{I}_{in})$ as domain of the newly introduced variable. Correctness of the claim follows from the fact that the bounds of any of the arithmetic and aggregate functions, applied to arguments of finite types, are themselves finite. There are two other places in the grounding algorithm where the type of a term is used to introduce new symbols: in the INTRODUCE TERM rule and when a constant is introduced in case the optimization term is not a constant. In both cases, we also apply *derive-bound* as this might result in a grounding over smaller types. In Section 4.3.5, we discuss additional (approximate) techniques the system uses to handle large and infinite types.

### 4.2.4 Reducing the Quantification Domain

At this point, it is useful to elaborate on an important optimization studied by Wittocx et al. [Wittocx et al., 2010], namely grounding *with bounds*. In the context of the KBS paradigm, it is another example of how intelligent reuse of one inference engine can improve other engines.

The INSTANTIATE rule presented above generally instantiates quantified variables with all elements in their type. In practice, we can see already in advance that some of these instantiations are not necessary. For example for a sentence $\forall x \in D : 1 \leq x \leq 10 \Rightarrow P(x)$, the left-hand side is false and hence the formula is trivially true for any instantiation other than assigning 1 to 10 to $x$. Consequently, we like for INSTANTIATE to result in $P(1) \land \ldots \land P(10)$, instead of first instantiating $x$ for all elements in $D$ and afterwards simplifying the obtained expression again.

To achieve this, we adapt INSTANTIATE as follows. For a formula $\forall \overline{x} \in \overline{\tau} : \varphi$, INSTANTIATE does not generate instantiations in $\overline{D}$, but only tuples of domain elements that are contained in $\{\overline{d} \mid \overline{d} \in \overline{D},\ \varphi_{pf}[\overline{x}/\overline{d}]^{\mathcal{I}_{in}} = \mathbf{t}\}$. Indeed, only because of instantiations for which the subformula is possibly false can the whole formula become false. This is then implemented by applying **query** inference to solve $\mathbf{query}_{pf}\langle \overline{x} \in \overline{D}, \varphi, \mathcal{I}_{in}\rangle$. For the above example, the result is the query $\mathbf{query}\left\langle x \in D, 1 \leq x \leq 10 \land P_{cf}(x), \mathcal{I}_{in}\right\rangle$.

Naturally, the query can safely return any overapproximation (results are bounded by $\overline{\tau}$ anyway). Hence, the query engine is allowed to optimize the query to balance the number of expected answers with the expected cost of generating those answers.

The complete improved INSTANTIATE rule is then changed from "select a $\overline{d}$ in $\overline{D}$" to "select a $\overline{d}$ from the results of query $q$", with $q$ one of the following queries.

- (skipping certainly true instances) For a formula $\forall \overline{x} \in \overline{D} : \varphi$, the query $\mathbf{query}_{pf}\langle \overline{x} \in \overline{D}, \varphi, \mathcal{I}_{in}\rangle$.
- (skipping certainly false instances) For a formula $\exists \overline{x} \in \overline{D} : \varphi$, $\mathbf{query}_{pt}\langle \overline{x} \in \overline{D}, \varphi, \mathcal{I}_{in}\rangle$. Similarly for set quantification.
- (skipping certainly false instances) For definitional quantification, we have to be extra careful to maintain equivalence. In general, we can only rely on the open symbols in the body of a rule, as otherwise loops might be lost. However, if the definition is total, we can replace $\overline{D}$ by the results of $\mathbf{query}_{pt}\langle \overline{x} \in \overline{D}, \varphi, \mathcal{I}_{in}\rangle$, for a rule $\forall \overline{x} \in \overline{D} : \mathit{head} \leftarrow \varphi$.[7]

The instantiation rule can be optimized even further by first querying whether there are any certain values. For example for $\forall x \in D : 1 \leq x \leq 10 \Rightarrow P(x)$, we could first solve $\mathbf{query}_{cf}\langle x \in D, 1 \leq x \leq 10 \Rightarrow P(x), \mathcal{I}_{in}\rangle$ to check whether there already is some $d \in [1,10]$ for which $P(d)$ is false in $\mathcal{I}_{in}$. If that is the case, the whole formula is replaced by false.

---

[7] Now, the grounding of a definition defining a symbol $s$, might have no rules for $s$ left. Whenever that happens, $s$ is made completely false in $\mathcal{I}_{in}$.

### 4.2.5 Concrete Algorithms and Implementation

The rewrite process of the previous section is not confluent. By imposing different rewrite strategies, it can be instantiated to a class of —sound— grounding algorithms.

**Example 4.2.8.** Consider for example a sentence $\forall x \in D : P(x) \vee (Q(x) \wedge R(x))$ with $Q$ completely true in $\mathcal{I}_{in}$. Suppose we instantiate $x$ with $d_1, d_1 \in D$. We can then first evaluate $Q(d_1)$ in $\mathcal{I}_{in}$, resulting in the subformula $P(d_1) \vee (\mathbf{t} \wedge R(d_1))$, which is simplified to the sentence $P(d_1) \vee R(d_1)$ that is added to $\mathcal{T}_g$. However, we could also first have introduced a Tseitin symbol for the context switch, resulting in $P(d_1) \vee T$ and $\{T \leftarrow Q(d_1) \wedge R(d_1)\}$. The latter is then simplified after evaluation to $\{T \leftarrow R(d_1)\}$.

To obtain a state-of-the-art grounding algorithm, one should select an instantiation that minimizes the number of traversals through formulas (in search for applicable rewrite rules), the memory and time complexity of the algorithm, the size of the grounding, . . . .

The rewrite strategy implemented in IDP takes the following main considerations into account:

1. The top priority is to minimize grounding size, followed by minimizing running time and memory usage.
2. INSTANTIATE is performed top-down and depth-first. This allows the grounder to simplify formulas early and reduces the memory overhead of storing partial results.
3. SIMPLIFY and EVALUATE are applied eagerly, as they may considerably reduce the size of formulas.
4. The number of introduced symbols should be minimized. E.g., by avoiding creating different Tseitin symbols for different occurrences of the same formula.

On the following pages, the main grounding algorithms specify this rewrite strategy in detail. Algorithm 1 is the top-level algorithm, while Algorithms 2 and 3 are responsible for grounding formulas and terms, respectively. We now discuss these in detail.

Algorithm 2 returns a sentence which is either a disjunction or conjunction of domain literals, together with a set of definitions (containing among others the definitions of new Tseitin symbols). Top-level Algorithm 1 start by applying

**Algorithm 1:** ground

```
Input: theory $\mathcal{T}_{in}$, structure $\mathcal{I}_{in}$
Output: $\mathcal{T}_g$, $\mathcal{T}_m$
1 $\mathcal{T}_{in}, \mathcal{T}_m$:= phase$_1$($\mathcal{T}_{in}, \mathcal{I}_{in}$);
2 $qm$ := initialize-queries($\mathcal{T}_{in}, \mathcal{I}_{in}$);
3 for sentence $s \in \mathcal{T}_{in}$ do
4     $\langle s_g, defs \rangle$ := reduce-ground($s, \emptyset, \mathcal{I}_{in}, qm$);
5     if $s_g$ is false then return unsatisfiable ;
6     $\mathcal{T}_g$ += $s_g$;
7     $\mathcal{T}_g \cup= defs$;
8 end
9 return $\langle \mathcal{T}_g, \mathcal{T}_m \rangle$;
```

the phase 1 simplifications on the whole theory. Afterwards, the algorithm proceeds by grounding one sentence at a time (applying Algorithm 2) and adding the ground sentence to $\mathcal{T}_g$. If that sentence is false, the algorithm terminates early and returns **unsatisfiable**. Otherwise, it continues by adding the definitions to $\mathcal{T}_g$. In Line 7, instead of simply adding all rules of $defs$ to $\mathcal{T}_g$, one can add rules defining Tseitin symbols only if the symbol that already occurs in $\mathcal{T}_g$. If some Tseitins have become irrelevant (e.g., they occurred in a disjunction in which some other disjunct turned out to be true) a stage will be reached where all the remaining rules define symbols not occurring in $\mathcal{T}_g$, at which point these remaining rules can be dropped.

Algorithm 1 can be optimized to produce a smaller grounding and to terminate early using the fact that IDP is closely integrated with its search algorithm. Theory $\mathcal{T}_g$ is in fact added directly to the search algorithm, without intermediate storage, and the search algorithm can then already apply its propagation rules to the part of the theory it has already seen (referred to as $propagate(\mathcal{T}_g, \mathcal{I}_{in})$). Any literals that are assigned by propagation during grounding can then be used to make $\mathcal{I}_{in}$ more precise, possibly resulting in less subsequent grounding. Propagations can also cause $\mathcal{I}_{in}$ to become inconsistent, in which case **unsatisfiable** is returned immediately. Thus, the Algorithm is extended by adding "$\mathcal{I}$:= propagate($\mathcal{T}_g$)" and "**if** $\mathcal{I}$ is inconsistent, **return unsatisfiable**" after Line 6.

The reduce-ground procedure is a wrapper (code omitted) that, depending on its first argument, calls ground-formula (Algorithm 2) or ground-term (Algorithm 3).

Algorithms 2 and 3 proceed by visiting the subformulas and/or subterms of

**Algorithm 2:** ground-formula (Ground a formula)

**Input**: formula $\varphi$, environment $env$, structure $\mathcal{I}_{in}$, querymap $qm$

**Output**: ground formula $\varphi_g$, ECNF definitions $defs$

```
switch $\varphi$ do
    case $[\neg]P(\bar{t})$
        $\bar{t}_g := \langle\rangle; defs := \emptyset;$
        foreach $t \in \bar{t}$ do
            $\left\langle t_g, defs' \right\rangle$ := reduce-ground($t$,$env$,$\mathcal{I}_{in}$, $qm$);
            if $t_g =$ undef then return $\langle \bot, \emptyset \rangle$; else $\bar{t}_g := \bar{t}_g :: t_g; defs \cup= defs';$
        end
        return $\left\langle [\neg]P(\bar{t}_g), defs \cup defs' \right\rangle$;
    case $\bigwedge_{\psi \in S} \psi$
        $\varphi_g := \top; defs := \emptyset;$
        foreach $\psi \in S$ do
            $\left\langle s_g, defs' \right\rangle$ := reduce-ground($\psi$,$env$,$\mathcal{I}_{in}$, $qm$);
            if $s_g = \bot$ then return $\langle \bot, \emptyset \rangle$; else $\varphi_g := \varphi_g \wedge s_g; defs \cup= defs';$
        end
        return $\left\langle \varphi_g, defs \right\rangle$;
    case $\exists \overline{x} \in \overline{D} : \psi$
        $\varphi_g := \bot; defs := \emptyset;$
        $q$ := initialize(qm($\varphi$),$env$));
        while hasAnswersLeft(q) do
            $\left\langle s_g, defs' \right\rangle$ := reduce-ground($\psi$, $env$+nextAnswer($q$), $\mathcal{I}_{in}$, $qm$);
            if $s_g = \top$ then return $\langle \top, \emptyset \rangle$; else $\varphi_g := \varphi_g \vee s_g, defs \cup = defs';$
        end
        return $\left\langle \varphi_g, defs \right\rangle$;
    cases $\bigvee_{\psi \in S} \psi$ and $\forall \overline{x} \in \overline{D} : \psi$ are similar to the above
    case definition $\Delta$
        $\Delta_g := \emptyset;$
        foreach rule $\forall \overline{x} \in \overline{D} : head \leftarrow \psi \in \Delta$ do
            $q$ := initialize($qm(\varphi)$,$env$));
            while hasAnswersLeft(q) do
                $\langle h, \emptyset \rangle$ := reduce-ground($head$, $env$+nextAnswer($q$), $\mathcal{I}_{in}$, $qm$, $D$);
                $\left\langle b, \Delta'_g \right\rangle$ := reduce-ground($\psi$, $env$, $\mathcal{I}_{in}$, $qm$);
                $\Delta_g$ += $h \leftarrow b$; $\Delta_g \cup = \Delta'_g$;
            end
        end
        return $\left\langle \Delta_g, \emptyset \right\rangle$;
endsw
```

**Algorithm 3:** ground-term (Ground a term)

```
Input: term $t$, environment $env$, structure $\mathcal{I}_{in}$, querymap $qm$
Output: ground term $t_g$, ECNF definitions $defs$
switch $t$ do
    case domain element $d$ return $\langle d, \emptyset \rangle$ ;
    case variable $v$ return $\langle env(v), \emptyset \rangle$ ;
    case function term $f(\bar{t})$
        $\bar{t}_g := \langle\rangle$; $defs := \emptyset$;
        foreach $t_i \in \bar{t}$ do
            $t_{i_g}, defs' :=$ reduce-ground($t_i$, $env$, $\mathcal{I}_{in}$, $qm$);
            $\bar{t}_g := \bar{t}_g :: t_{i_g}$; $defs \cup = defs'$;
        end
        return $\langle f(\bar{t}_g), defs \rangle$;
    case aggregate term $agg(\bigcup_{i \in [1,n]} \{\bar{x}_i : \varphi_i : t_i\})$
        $set := \emptyset$; $defs := \emptyset$;
        foreach $i \in [1,n]$ do
            $q :=$ initialize(qm($\{\bar{x}_i : \varphi_i : t_i\}$),$env$));
            while hasAnswersLeft($q$) do
                $next :=$ nextAnswer($q$);
                $\langle lit, defs' \rangle :=$ reduce-ground($\varphi_i$, $env + next$, $\mathcal{I}_{in}$, $qm$);
                $\langle t_g, defs'' \rangle :=$ reduce-ground($t_i$, $env + next$, $\mathcal{I}_{in}$, $qm$);
                $set \cup = \{lit : t_g\}$; $defs \cup = defs' \cup defs''$;
            end
        end
        return $\langle agg(set), defs \rangle$;
endsw
```

the formula, respectively term at hand. Whenever applicable, simplifications will be applied on the go. Next to a formula or term, the algorithms take as arguments a structure $\mathcal{I}_{in}$, an *environment env*, which maps variables to their instantiation, and a *query map qm*, which maps quantified formulas and sets to a "query object". The idea is that queries are expensive to construct (requiring, e.g., query optimization) and that the different queries for given formula or set will be identical except for the instantiation of free variables. Consequently, queries for all quantifications are created and optimized in an initial phase (*initialize-queries*), with a concrete instantiation of the free variables. The resulting "query-objects" provide three operations:

**initialize** takes an environment *env*, interpreting at least all free variables of the query, and initializes the query for *env*.

**nextAnswer** returns the next variable instantiation that is an answer of the query (over the *env* used to initialize it). The operation guarantees to never return duplicate instantiations.

**hasAnswersLeft** returns true iff the query has unseen results left.

Both algorithms then recursively visit subformulas and subterms. Each recursive call returns a ground expression and, possibly, a set of Tseitin definitions. After the recursive call, results are combined (depending on the type of formula or term) and returned upwards. Using the simplification rules discussed in the previous section, the algorithms check on the go whether they can stop early.

The recursive calls are to the operation *reduce-ground*. It starts by calling either $ground\text{-}formula$ or *ground-term* (depending on the first argument) and afterwards applies two postprocessing steps. First, EVALUATE is applied over $\mathcal{I}_{in}$, which returns its interpretation (if any). Second, INTRODUCE TSEITIN or INTRODUCE TERM are applied if the expression is not yet in ECNF. In addition, some amount of sharing is done, such as using $\neg T$ to Tseitinize $c \neq c'$ if $T$ was introduced earlier for $c = c'$.

To achieve the goal of minimizing the introduction of new symbols for the same formula/term, we further improve the algorithms by applying *folding*. Folding consists of taking a ground sentence/rule containing a Tseitin $T$ and checking whether replacing $T$ with its definition would still result in an ECNF expression. For example given a ground sentence $A \vee B \vee T$ with $T$ defined as $T \leftarrow C \vee D$, then the replacement results in the ECNF sentence $A \vee B \vee C \vee D$ and $T$ has been eliminated. This folding is done the moment expressions are added to the ground theory, both for predicate symbols (reasoning on $\wedge$ and $\vee$) and function symbols (reasoning on arithmetic operators).

## 4.3 Practical Optimization Engine

In previous sections, a grounding algorithm was presented. In combination with a search algorithm for ground $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$ theories (presented in Chapter 5), it can be used to solve **optimize**$\langle \mathcal{T}, \mathcal{I}, c, \Sigma_{out} \rangle$ problems. However, to obtain a state-of-the-art optimization engine, we extend it with a number of pre- and postprocessing techniques, which we discuss below. These techniques reduce the size of the grounding and/or improve search performance. In Section 4.3.3

they are then combined into the full optimization algorithm used in IDP. Afterwards, we discuss some practical considerations and the scalability of the approach.

Note that three of the preprocessing techniques described below (exploiting symmetries, exploiting input$*$ definitions and symbolic unit propagation) were not developed by me (see references in the separate sections). However, the techniques are discussed here as they constitute important building blocks to realize our major goal of reducing the impact of the modeling style on the size of the grounding and the performance of the inference engines.

The input structure of **optimize**$\langle \mathcal{T}, \mathcal{I}, c, \Sigma_{out} \rangle$ was $\mathcal{I}_{in}$. The techniques below sometimes derive more precise structures. Hence, from now on we use $\mathcal{I}$ as the "current" partial structure, which is initialized with $\mathcal{I}_{in}$ before preprocessing.

## 4.3.1 Preprocessing

### Checking Consistency

Previously, we assumed $\mathcal{I}$ was consistent. If the input structure is inconsistent, this should preferably be detected as soon as possible. Indeed, in that case we do not even have to start grounding. The structure can turn out to be inconsistent because of

- A domain element that is part of some type but not of some of its supertypes.
- A domain or atom term that belongs to a relation, but some of its arguments are not contained in their respective types.
- A function that maps to multiple values for the same domain tuple.
- A total function that has no value for some domain tuple.

Each of these can be checked by expressing the property as a formula that is true for variable instantiations that violate the property. For example, for an $n$-ary total function $f[\overline{\tau} \mapsto \tau']$, violations of the last property are instantiations of variables $\overline{x}$ in $\overline{\tau}$ for which the formula $\varphi = \forall y \in \tau' : f(\overline{x}) \neq y$ is true. Consequently, we can reuse one of our main inference engines and apply query inference to solve the task **query**$\langle \overline{x} \in \overline{\tau}, \varphi, \mathcal{I} \rangle$. Answers to the query are then indeed instantiations for which the property is violated.

This check is applied for all relevant queries for the above four properties. If any of these queries is not empty, **unsatisfiable** is returned and the matching instantiation(s) are returned to the user as debugging output. To improve efficiency, in practice part of the consistency check is done on-the-fly when a structure is being manipulated, for example during parsing.

### Reducing Quantification Depth using Functional Dependencies

In the previous section, we have discussed that function symbols can reduce the size of the grounding. However, the aim of the system is to hide such concerns from the user wherever possible.

One way to reduce the quantification depth is to detect that symbols can in fact be split in a number of symbols with a smaller arity. Consider, for example, the packing relation $size[id, nb]$, where the size of a box is modeled as a binary predicate symbol. If one could detect that the second argument *functionally depends* on the first argument (the first uniquely determines the value of the second), then it could be replaced by new function symbols $f_{size}[id \mapsto nb]$ instead. Variables $s_i$ can then be eliminated from the theory and thus reduce the quantification depth.

This technique, detecting that symbols can be split automatically and rewriting the theory to exploit this, is the focus of Chapter 6.

Second, *Skolemization* is applied to introduce even more function symbols. Skolemization eliminates existential quantifications by introducing additional (*Skolem*) function symbols:

$$\exists x \in \tau : \varphi \rightarrowtail \varphi[x/f(\overline{y})] \text{ in the context of free variables } \overline{y}[\overline{\tau}']$$

$$\text{with } f[\overline{\tau}' \mapsto \tau] \text{ a new function symbol}$$

Skolemization is only applied to existential quantification in monotone (positive) context, as otherwise it is not guaranteed to preserve equivalence. In anti-monotone context, it is allowed to apply Skolemization to universal quantifications; this is not done at the moment (recall, negations are pushed down during grounding).

### Exploiting Symmetries

It is well-known that when symmetries are present in a problem, they can cause a search algorithm to solve the same (sub)problem over and over. For example the "pigeonhole" problem "do $n$ pigeons fit in $n-1$ holes?" is a problem known to be hard for SAT-solvers. In our packing example, all boxes with the same size are symmetric: swapping any two such boxes in a valid solution will result in another valid solution.

Symmetries can be detected and broken on the propositional level, as shown for example in [Aloul et al., 2006], but for large problems, even the task of detecting symmetries becomes infeasible on the propositional level. Detecting symmetries on a first-order level is often an easier problem, as more structure of the problem is explicitly available. For example, for an FO specification of the pigeonhole problem, it is almost trivial to detect that all pigeons are interchangeable.

The symmetry-detection functionality in IDP detects a simple, frequently occurring form of symmetries: interchangeable domain elements. Two domain elements are considered interchangeable if they are of the same type and occur only symmetrically in interpreted predicates. The detected symmetries are either used by extending the theory with sentences that break the symmetries statically or they are passed directly to the search algorithm, which exploits them using dynamic symmetry-breaking methods [Devriendt et al., 2012]. The latter has the (theoretical) advantage that no parts of the search space are excluded a-priori, but only parts symmetric to already visited ones are avoided, and hence lend themselves better to techniques that construct the grounding on-the-fly, such as discussed in Chapter 7. In practice, static symmetry breaking usually outperforms dynamic approaches.

The techniques were integrated into IDP by Jo Devriendt in the context of his master's thesis and extended as presented in [Devriendt et al., 2012].

### Exploiting input$*$ Definitions

Top-down grounding techniques, as used in IDP, tend not to perform very efficiently in case of complex (inductive) definitions [Wittocx, 2010]. However, definitions with only two-valued parameters, so-called *input$*$* definitions, can be evaluated in advance instead of during the main grounding phase to make the input structure $\mathcal{I}$ more precise [Jansen et al., 2013].

First, a transformation is applied on the theory that splits definitions in their strongly connected components on the symbolic level (reducing the number of rules in a definition). Afterwards, if the theory contains a definition $\Delta$ with only two-valued open symbols, the current partial structure $\mathcal{I}$ is expanded by applying **delta model expansion** to $\Delta$ and $\mathcal{I}$. If this results in an inconsistent structure, **unsatisfiable** is returned. Otherwise, $\Delta$ is removed from the theory. Delta model expansion on a definition $\Delta$ and structure $\mathcal{I}$ proceeds by first translating $\Delta$ into a tabled Prolog program with the same well-founded model as $\Delta$ in $\mathcal{I}$. Afterwards, the Prolog program is queried for the interpretation of all defined symbols in $\Delta$ using XSB-Prolog and $\mathcal{I}$ is extended with the results. As discussed in [Jansen et al., 2013], this approach results in smaller grounding size and increased efficiency.

Next, consider total definitions $\Delta$ in the resulting $\mathcal{T}$ (after input$*$-evaluation) of which none of the defined symbols occurs outside of $\Delta$ nor is interpreted in $\mathcal{I}$, so-called *output$*$* definitions. Any structure that satisfies the theory without $\Delta$ and does not interpret symbols defined in $\Delta$, can be expanded to a model of the definition by applying delta model expansion. Consequently, output$*$ definitions do not have to be considered during search and do not have to be grounded. Instead, they are removed from $\mathcal{T}$ and stored separately, to be evaluated afterwards in a post-processing step. Obviously, a definition is also output$*$ if all its defined symbols only occur in other output$*$ definitions and it does not introduce loops over defined symbols over different output$*$ definitions.[8] The approach can be extended in a straightforward way to handle the case that not all defined symbols are part of $\Sigma_{out}$. In that case, some definitions need not be evaluated at all, not during search nor during pre- or postprocessing.

How to split the theory is implemented through bootstrapping: an IDP meta-theory expresses which definitions should be evaluated *before*, *during* or *after* search and which definitions can be forgotten altogether given $\Sigma_{out}$. Model expansion is then applied and the resulting model indicates how the theory should be split.

### Improving Grounding using Symbolic Unit Propagation

The grounding algorithm exploits information in the structure in two ways: it queries the type of a variable for instantiations and it evaluates terms and formulas in the structure at hand. Such an evaluation is only possible for a symbol $P$ if it is interpreted explicitly in $\mathcal{I}$. What if our theory contained sentences like

- $\forall x : P(x)$ (all atoms over $P$ are true), or
- $\forall x : Q(x)$ and $\forall x : Q(x) \Rightarrow P(x)$ (all atoms over $Q$ are true and an atom $Q(d)$ implies $P(d)$), or
- $P(1)$ (literal $P(1)$ is true)?

---

[8] Two total definitions do not necessarily share a model if there are loops over them. For example, the theory consisting only of the definitions $\{P \leftarrow Q\}$ and $\{Q \leftarrow \neg P\}$ has no models, even though both definitions are total.

In all three cases, the sentence(s) in fact fix (part of) the interpretation of $P$ and it would be beneficial if this could be exploited during grounding. Otherwise, our grounding algorithm does additional work that is eliminated again afterwards by propagation.

Observations like this gave rise to the algorithms presented in [Wittocx et al., 2010, Wittocx et al., 2013], where $\mathcal{I}$ is made more precise into a structure $\mathcal{I}'$, such that all models of $\mathcal{T}$ that expand $\mathcal{I}$ also expand $\mathcal{I}'$. This allows us to detect earlier that a formula does not need to be grounded and avoids wasting precious time on useless work. As a more precise structure allows us to create a smaller grounding and increase search performance, the ideal structure as input of the grounding step would be the *most precise* structure that captures all models of $\mathcal{T}$ that are more precise than $\mathcal{I}$. Of course, finding this ideal structure is a task that is even harder than the original problem.

Instead of searching for this ideal structure, IDP's approach is to execute a lifted (approximate) version of the (unit) propagation that would occur after grounding anyway. The result is stored as a symbolic representation of a structure. Namely, for each symbol $P$, $P_{ct}$ and $P_{cf}$ are associated with symbolic set expressions $S_{ct}$ and $S_{cf}$. The interpretation of $P_{ct}$ ($P_{cf}$) in $\mathcal{I}'$ is then $S_{ct}^{\mathcal{I}}$ ($S_{cf}^{\mathcal{I}}$). Consider, for example, a theory containing the sentences $\forall x : P(x) \Rightarrow Q(x)$ and $\forall x : P(x) \Rightarrow R(x)$. Symbolic unit propagation associates $\{x \mid P_{ct}(x)\}$ (interpreted in $\mathcal{I}$!) with $P_{ct}$, $\{x \mid P_{cf}(x) \lor Q_{cf}(x) \lor R_{cf}(x)\}$ with $P_{cf}$ and $\{x \mid P_{ct}(x)\}$ with $R_{ct}$. During the grounding phase, all queries for variable instantiations and the interpretation of atoms and terms are then evaluated (using query inference) relative to this symbolic interpretation, resulting in less instantiations and more precise interpretations. E.g., if in the above $Q$ is two-valued in $\mathcal{I}$, the second sentence will only be instantiated for $x'$s for which $Q(x)$ is not false in $\mathcal{I}$. Indeed, during grounding $x$ is queried for instantiations for which $P$ is not false and $R$ not true, and the interpretation of $P_{cf}$ now states that it is true if $Q_{cf}$ is true.

A symbolic representation of complete symbolic unit propagation (SUP) often consists of complex formulas, which are infeasible to query. However, we can suffice with any approximation of those formulas, as long as the resulting structure is at least as precise as $\mathcal{I}$. Consequently, a greedy simplification approach is used to balance the estimated cost of querying against the expected reduction in number of answers.

After SUP, we could turn $\mathcal{I}'$ from a symbolic into a concrete structure, in which case the approach is called Lifted Unit Propagation [Vaezipoor et al., 2011]. However, it is often beneficial to keep the structure symbolic as long as possible. It is, for example, easier to generate intelligent quantification bounds based on a symbolic representation and, in case an output vocabulary is given, we do not have to evaluate all symbolic interpretations. For symbols over infinite domains, this might even be untractable.

**Example 4.3.1.** Consider a graph application where we are looking for a path through a given graph. The vocabulary consists of a type *node*, predicate symbols $edge[node, node]$ and $path[node, node]$ and function symbols $start[\mapsto node]$, $end[\mapsto node]$. The theory contains the following sentences (among others):

$$\begin{array}{l} \forall x\, y : path(x,y) \Rightarrow edge(x,y) \\ \forall x : \#(\{y : path(x,y)\}) < 2 \\ \forall y : \#(\{x : path(x,y)\}) < 2 \\ \forall xy : path(x,y) \Rightarrow y = end \lor \exists z : path(y,z) \\ (\exists x : path(start,x)) \land (\exists x : path(x,end)) \end{array}$$

For example, for the fourth sentence, without SUP, $x$ and $y$ are instantiated with all possible nodes. With SUP, they are only instantiated with edges in the graph (typically a lot less).

Symbolic unit propagation removes the need of having proper "guards" for every quantification (e.g., adding an $edge(x,y)$ conjunct to the fourth sentence). Indeed, without such a technique, the user himself has to guarantee that the subformula of every quantification has the information necessary to sufficiently bound the variable instantiation. This is indeed an important modeling advice for most ASP systems. With SUP, such information is derived from the theory as a whole, resulting in a smaller need of tuning and structuring a specification.

Afterwards, consistency of the partial structure is checked (again), as using more information from the theory might have made it inconsistent. As part of the structure is now kept symbolic, more intelligent query simplification techniques can be applied, for example to check disjointness of $s_{ct}$ and $s_{cf}$ for symbol $s$.

Next to reducing the size of the grounding, the technique can also improve search performance because the search algorithm does not have to spend time in irrelevant parts of the search space, as demonstrated in [Vaezipoor et al., 2011].

### 4.3.2 Post-processing

The search algorithm returns a structure that is a model of the ground theory, which is typically over a larger vocabulary (containing Tseitin and Skolem symbols). The first postprocessing step is to project the structure on $\Sigma$.

However, we might still have a partial structure of which not all expansions are models. If the obtained structure is partial, it is possible that not all expansions are models if some symbols are defined in an output$*$ definition. Hence, as second step, output$*$ definitions are evaluated. The result is a structure of which indeed all expansions are models. Afterwards, we make the symbolic[9] structure concrete (which is consistent as we checked during preprocessing) for symbols in $\Sigma_{out}$ and project it into $\Sigma_{out}$, resulting in a solution to the original **optimize**$\langle \mathcal{T}, \mathcal{I}, c, \Sigma_{out} \rangle$ problem. If that solution is a partial structure, all two-valued expansions of it are also solutions, as discussed earlier. If symmetry-breaking was applied, additional solutions can now be generated by applying the symmetries to the solutions found.

### 4.3.3 Complete Optimization Algorithm

Combining all techniques presented in this chapter, Algorithm 4 presents the global optimization algorithm in IDP. After checking consistency of the input structure, it applies the described preprocessing operations, namely (in order) checking consistency, detecting and exploiting functions, symmetries and input$*$/output$*$ definitions and making $\mathcal{I}$ more precise through LUP. Afterwards, ground and search is applied and postprocessing takes place.

---

**Algorithm 4:** The **optimize**$\langle \mathcal{T}, \mathcal{I}, c, \Sigma_{out} \rangle$ algorithm in IDP

---

```
Input: $\mathcal{T}$, $\mathcal{I}$, c, $\Sigma_{out}$
Output: a voc($\mathcal{T}$)-structure or unsatisfiable
if inconsistent($\mathcal{I}$) then return unsatisfiable
$\mathcal{T}, c$ := detect-and-rewrite-functions($\mathcal{T}$, $c$)
$\pi$ := detect-symmetries($\mathcal{T}$, $\mathcal{I}$, $c$)
$\mathcal{T}, \mathcal{I}, defs_{out}$ := definition-handling($\mathcal{T}$, $\mathcal{I}$, $\Sigma_{out}$)
$\mathcal{I}_{symb}$ := approximation($\mathcal{T}$, $\mathcal{I}$)
if inconsistent($\mathcal{I}_{symb}$) then return unsatisfiable

$\mathcal{T}_g, c_g$ := ground($\mathcal{T}$, $c$, $\mathcal{I}_{symb}$)
$\mathcal{M}$ := search($\mathcal{T}_g$, $c_g$, $\mathcal{I}_{symb}$, $\pi$)
if inconsistent($\mathcal{M}$) then return unsatisfiable

$\mathcal{M}$ := evaluate-output$*$-defs($\mathcal{M}$, $defs_{out}$)
$\mathcal{M}$ := make-concrete($\mathcal{M}$, $\Sigma_{out}$)
return project($\mathcal{M}$,$\Sigma_{out}$)
```

---

[9] Recall, symbolic unit propagation results in a symbolic structure.

The order of preprocessing operations is derived from the observations that functional rewriting introduces additional input$*$ and output$*$ definitions and that approximation should happen on the part of the theory relevant for search, to obtain a smaller symbolic structure.

### 4.3.4 Practical Considerations

To allow for more practical usage, models are generated (and postprocessed) one at a time, after which search can be continued to find additional solutions.

For minimization, intermediate (suboptimal) models are returned one at a time. The sequence of suboptimal models is guaranteed to be ordered strictly descending according to the value of $c$. Consequently, minimization can be aborted at any time and return the best model found till then (in effect, an *anytime* algorithm). The system reports if optimality of the last model found was proven; afterwards, search can continue to find different models with the same (optimal) value for $c$.

Various options can be used to control the inference engine, ranging from changing which underlying engines are used to changing the semantics. A small overview of the main options:

- `stdoptions.groundwithbounds` indicates whether subformulas should be used to derive bounds on quantifications.
- `stdoptions.liftedunitpropagation` indicates whether lifted unit propagation should be done before search.
- `stdoptions.xsb` decides whether to evaluate input$*$ definitions with XSB Prolog (instead of including them in the ground-and-search phase).
- `stdoptions.symmetrybreaking` can be either `none`, `static` or `dynamic` to indicate whether symmetries should be detected and how they should be broken in the latter two cases (by adding symmetry breaking clauses or by applying symmetry propagation, respectively).
- `stdoptions.semantics` can be either `completion`, `stable`, or `wellfounded`, according to the preferred semantics for inductive definitions.

### 4.3.5 Scalability and Infinity

The reader might have noticed that structures and groundings can be very large or even infinite (for example, when a predicate or a quantified variable are typed over `int`). Model expansion (and thus, optimization) over infinite structures takes infinite time in general. In IDP, several techniques are applied that enable the system to address this issue and have been shown to work well in practice.

A first such technique has already been explained in Subsection 4.3.1: by intelligent reasoning over the entire theory, we can sometimes derive better variable bounds. Suppose, for example, that a theory contains formulas $\forall x \in \mathbb{N} : P(x) \Rightarrow Q(x)$ and $\forall y \in \mathbb{N} : P(y) \Rightarrow R(y)$, where $Q$ only ranges over a finite type, say $T$, but $P$ and $R$ range over $\mathbb{N}$. The first of these sentences guarantees that $P$ will only hold for values such that $Q$ holds, hence $P$ can only hold for values in the finite type $T$. Thus we know that the second sentence should only be instantiated for $y$'s in $T$, i.e., by deriving an improved bound for $y$, the grounding of the second sentence suddenly becomes finite. The first sentence can be handled similarly. We only ground this sentence for $y$'s in $T$ and maintain a symbolic interpretation expressing that $P$ is certainly false outside of $T$.

Second, the usage of a top-down, depth-first grounding algorithm has the advantage that the structure is evaluated *lazily*: (**i**) queries generate instantiations one at a time, and (**ii**) the interpretation of atoms and terms needs only to be retrieved for atoms and terms that effectively occur on the grounding. The same advantage applies for symbols that are interpreted by (complex) procedures: the procedures are only executed for effective occurrences.

The search algorithm maintains bounds on the interpretation of function terms, taking constraints in the grounding into account. Consider a constant $c \mapsto int$, which in itself would result in an infinite search space. However, combined with, e.g., a sentence $0 \geq c \geq 10$ in the grounding, the solver reduces $c \mapsto int$ to $c \mapsto [0, 10]$, a finite search space.

Last, the lazy model expansion technique, presented in Chapter 7, is built on the observation that the entire grounding is often not necessary during search. Instead, grounding is done on-the-fly during search, whenever satisfaction of a non-grounded part of the theory can no longer be guaranteed. For more details, we refer to Chapter 7.

## 4.4 Related Work

The techniques presented in this chapter fit in a more general effort towards more intelligent model expansion support for rich languages. Similar grounding and analysis techniques are applied in CP-, ASP- and FO-based model expansion algorithms.

Considering grounding techniques, several approaches exist. The ASP grounders Gringo [Gebser et al., 2011b] and the grounder [Faber et al., 2012] of the DLV system [Leone et al., 2006] use *semi-naive evaluation* to efficiently ground an ASP program. The techniques iteratively derives which rule bodies might still become true and ground those. In Gringo, this is combined with a query reordering strategy, in DLV more advanced optimization techniques from the field of databases are used, enabling them to provide some guarantees in case of an infinite Herbrand base. The FO$(ID)$ grounders of the MXG framework [Mitchell and Ternovska, 2005, Patterson et al., 2007] and of Enfragmo [Aavani et al., 2012] are based on algebraic database theory and work by bottom-up application of relation operators, starting from the initial interpretation. The FO grounder KodKod [Torlak and Jackson, 2007] of the Alloy system [Jackson, 2002] applies bottom-up grounding through matrix operations. In CP, the solver-independent languages Zinc [Marriott et al., 2008] and MiniZinc [Nethercote et al., 2007] can be *flattened* in a straightforward way to their ground fragment FlatZinc using the `mzn2fzn` tool. FlatZinc is supported by a range of search algorithms as can be seen on `www.minizinc.org/challenge2013/results2013.html`. For a more in-depth comparison between (some of) these grounding algorithms, we refer the reader to [Wittocx, 2010] and [Aavani, 2014].

Note that neither bottom-up nor top-down grounding is ideal in every situation. For example, bottom-up grounding suffers from issues with intermediate table size while it is more difficult to use the input structure effectively in top-down grounding. As a result, current state-of-the-art grounders often combine ideas from both types of grounding algorithms, such as magic-set transformation in DLV or top-down grounding in IDP combined with symbolic unit propagation. The main drawback to IDP's grounding algorithm is the fact that the symbolic unit propagation is not as fine-grained as semi-naive evaluation for highly recursive expressions. For example for rules of the form $\forall t : P(t+1) \leftarrow P(t)$, symbolic unit propagation cannot efficiently derive that if some $P(t)$ is certainly false, all $P(t_2)$, with $t < t_2$, will also always be false. Such expressions frequently occur in for example planning applications.

Within ASP, work is ongoing to extend the language to support "constraint atoms", which represent a CSP problem and need not be grounded to predicate logic. Examples of CASP systems are Clingcon [Ostrowski and Schaub, 2012], EZ(CSP) [Balduccini, 2011] and Inca [Drescher and Walsh, 2011a]. Constraint-ASP languages generally only allow a restricted set of expressions to occur in constraint atoms and impose conditions on where constraint atoms can occur. For example, none of the languages allows general atoms $P(\bar{c})$ with $P$ an uninterpreted predicate symbol. One exception is $\mathcal{AC}(\mathcal{C})$, a language aimed at integrating ASP and Constraint Logic Programming [Mellarkod et al., 2008]. As shown in [Lierler, 2012], $\mathcal{AC}(\mathcal{C})$ captures the languages of both Clingcon and EZ(CSP); however, only subsets of the language are implemented [Gelfond et al., 2008]. (Mini-)Zinc supports uninterpreted functions in the modeling language, as discussed in [Stuckey and Tack, 2013]. Most of the CASP systems ground to a language that supports such constraint atoms, hence, they also use an extended search algorithm to handle the richer language. We discuss these after presenting MINISAT(ID) in the next chapter.

An approach to reduce the number of introduced symbols is to apply common subexpression elimination, see e.g. [Rendl et al., 2009]. In IDP, a limited version of subexpression elimination is done for arithmetic operations and ground atoms. It is part of future work to implement a more complete version. In [Zhang and Yap, 2011], it is shown that functional dependencies can be exploited to reduce the number of variables in the problem, simplifying subsequent search.

## 4.5 Conclusion

In this chapter, we presented the workflow of the **optimize**$\langle \mathcal{T}, \mathcal{I}, c, \Sigma_{out} \rangle$ $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$-inference engine in IDP. The main component is the grounding algorithm, which has been parametrized to allow specific types of function symbols in the grounding. These function symbols are then exploited by the search algorithm that is presented in the next chapter. In this way, we can, without changes to the input language, support the next generation of search algorithms that integrate techniques from SAT, ASP and CP. In addition, we showed how a number of transformations and more complex pre- and postprocessing techniques can be integrated into the workflow, to reduce the blowup caused by grounding and to improve subsequent search. For most of the separate techniques, their advantage has been demonstrated in other publications. The experimental evaluation of the engine as a whole is left to the following chapter, where we first finalize it by presenting the MINISAT(ID) search algorithm.

The main drawback of the current grounding component is the lack of an intelligent grounding technique for highly recursive expressions, for example by integrating semi-naive evaluation. Investigating this is part of future work.

Interestingly, a few years ago, the modeling advice was to forgo function symbols whenever performance mattered, even though they are often more natural from a modeling point of view. Indeed, function symbols had to be graphed, which sometimes resulted in a larger grounding than a tuned predicate specification. At this moment, IDP users are actively encouraged to use function symbols: both from a modeling *and* performance point of view, they are now often superior.

To give users more freedom in writing models, there is a trend to integrate a wide range of (more-or-less) general techniques that each optimize inference for a particular class of expressions. E.g., in Example 4.2.5, we illustrated that additional rewrite rules for arithmetic expressions would be useful to simplify the theory further. To keep the complexity of inference engines under control, there is a need for a standardized way to develop and (re)use such optimizations.

# 5

# MiniSAT(ID): Ground $\text{FO}(\cdot)^{\mathtt{IDP}}$ Search Algorithm

The aim of this chapter is to present the MINISAT(ID) search algorithm, a search algorithm for general ground $\text{FO}(\cdot)^{\mathtt{IDP}}$. The work is based on a number of observations. First, the tremendous progress made in SAT-solving over the last decade, allowed researchers to tackle many new classes of search problems. Second, in CP and ASP it was already shown that a richer input language (and, hence, more of the original problem structure in the input) allows systems to reduce the size of the solver input and to increase search performance. The algorithm natively combines learning during search with efficient propagation for uninterpreted functions, arithmetic, aggregates and inductive definitions. The algorithm is able to add any symbol or sentence during search, which is for example crucial for the interleaving of grounding and search in Chapter 7. Its development is part of a larger trend to improve search by combining ideas from different fields such as SAT, CP and ASP, apparent, e.g., in the emerging field of Constraint ASP (CASP) [Lierler, 2012].

The main results have been published in [De Cat et al., 2013a].

The chapter is structured as follows. In Section 5.1, some background in search algorithms is provided. In Section 5.2, the MINISAT(ID) search algorithm is presented; it forms the last building block of IDP's **optimize**$\langle \mathcal{T}, \mathcal{I}, c, \Sigma_{out} \rangle$ engine. Afterwards, in Section 5.3, we present an experimental evaluation of the ideas presented in this and the previous chapter. Related work is discussed in Section 5.4, followed by a conclusion in Section 5.5.

## 5.1 Background

In the last decade, a lot of attention went to Conflict-Driven Clause-Learning (CDCL) algorithms because they successfully combined automated learning with a single, simple propagation rule (unit propagation) and with effective heuristics. In this thesis, we develop, among others, techniques to improve over CDCL to achieve even better performance in automated reasoning tasks. In this section, we give an overview of the main techniques used in CDCL search algorithms, which we sometimes refer to as *SAT-solvers*. For more information, we refer the reader to [Marques Silva et al., 2009].

### 5.1.1 Basics of CDCL

The basic CDCL algorithm takes a CNF theory $\mathcal{T}$ as input and returns a model of the theory or **unsatisfiable** if the theory has no models. The state of the algorithm is characterized as a tuple $\langle \mathcal{I}, C \rangle$ with

- $\mathcal{I}$ a mapping of input atoms to their truth value (**t**, **f** or **u**), the decision level and order in which they were assigned. They are also labelled with either the clause that derived them or with $d$ if it was a decision. Abusing notation, we use this mapping also as the partial $voc(\mathcal{T})$-structure that can be straightforwardly derived from it.
- $C$ a set of clauses, with clauses derived during search labelled with $l$.

The transition rules typically applied in a CDCL algorithm are the following:

**Decide**:

$$\langle \mathcal{I}, C\rangle \quad \longrightarrow \quad \left\langle \mathcal{I}+l^{d}, C\right\rangle \quad \textbf{if} \quad l^{\mathcal{I}} = \mathbf{u}$$

**Unitpropagate**:

$$\langle \mathcal{I}, C\rangle \quad \longrightarrow \quad \left\langle \mathcal{I}+l^{\varphi \vee l}, C\right\rangle \quad \textbf{if} \begin{cases} l^{\mathcal{I}} = \mathbf{u} \\ \varphi \vee l \in C, \varphi^{\mathcal{I}} = \mathbf{f} \end{cases}$$

**Learn**:

$$\langle \mathcal{I}, C\rangle \quad \longrightarrow \quad \left\langle \mathcal{I}, C+\varphi^{l}\right\rangle \quad \textbf{if} \begin{cases} C \models \varphi \\ \varphi \notin C \end{cases}$$

**Backtrack**:

$$\left\langle M+l^{d}+N, C\right\rangle \quad \longrightarrow \quad \langle M, C\rangle$$

**Fail**:

$$\langle \mathcal{I}, C\rangle \quad \longrightarrow \quad \textbf{unsatisfiable} \quad \textbf{if} \quad \text{some clause } c \in C \text{ is false at the root level}$$

These rules are then combined into a complete search algorithm, for example the one shown in Algorithm 5. The algorithm first applies unit propagation to the input; afterwards it goes into a decide-propagate-learn loop, which terminates when either a model is found or when a conflict has been found at the root level, indicating that no models exist.

### 5.1.2 Concrete Instantiation of the Rules

Current state-of-the-art instantiations of the CDCL skeleton outlined above share a number of important features. We give an overview.

**Decision Heuristic.** The order in which atoms are chosen and what value they are assigned is crucial for any SAT-solver, and various well-performing domain independent heuristics have been devised. One example of a domain-independent variable-selection heuristic is the Variable-State Independent Decaying-Sum (VSIDS) heuristic. VSIDS keeps a mapping from atoms to a score reflecting how often that atom has been part of a conflict clause. Whenever a decision has to be taken, the unassigned atom with the highest score is selected. VSIDS reflects the idea that variables that lead to conflicts often should be selected earlier to spend less time in finding conflicts. Value-selection heuristics are usually simpler, even in state-of-the-art solvers, such as always assigning choice atoms true or always false, choosing their value randomly or dependent on the ratio of positive and negative atom occurrences in the theory. Currently,

**Algorithm 5:** Outline of a general CDCL algorithm.

```
Input: a CNF theory $\mathcal{T} = \{c_1, \ldots, c_n\}$
Output: a model of $\mathcal{T}$ or unsatisfiable if $\mathcal{T}$ has no models
$\mathcal{I}$:= $\emptyset$; $dl$ := 0;
$C$ := $\{c_1, \ldots, c_n\}$;
unitpropagate($\mathcal{I}$,$C$);
if conflict then return unsatisfiable;
while not satisfied($\mathcal{I}$,$C$) do
    $\langle var, val \rangle$ := chooseVarAndValue($\mathcal{I}$,$C$);
    $\mathcal{I}[var]$ := $val$; $dl$ := $dl + 1$;
    unitpropagate($\mathcal{I}$,$C$);
    if conflict then
        $\langle clause, level \rangle$ := analyzeConflict($\mathcal{I}$,$C$);
        if level<0 then return unsatisfiable;
        backtrack($\mathcal{I}$,$C$,$level$);
        $C$ := $C + clause$; $dl$ := $level$;
    end
end
return $\mathcal{I}$;
```

the most effective value-selection heuristic consists of tracking the value atoms had previously (before they were assigned unknown again by backtracking), and always reassign a variable its previous value. If an atom has not been assigned previously, one of the other heuristics is applied.

**2-Watched Literal Scheme.** A CDCL algorithm spends most of its time doing unit propagation. Unit propagation is done until fixpoint, by checking for propagation of assigned literals in the order they were assigned, until all have been checked or a conflict has been found.[1] Time is mainly spent checking whether a clause is unit or false, as assigning it or raising a conflict are generally cheap. The most important technique devised to speed up checking for propagation is the *2-watched literal scheme*. The idea is that, in fact, it is not necessary to check every clause after every change in the interpretation. Indeed, as long as there are two literals in any clause that are not both false, no propagation or conflict is possible. This is implemented by maintaining a mapping $w$ from literals $l$ to clauses containing $\neg l$, such that at least two literals point to any clause $c$. We then say that a literal $l$ is a *watch* for a clause $c$ if $c \in w(l)$. If a literal becomes true, only the clauses it watches are rechecked

---

[1] Following the order of assignments turned out to result in better learned clauses.

for propagation and conflict and its watches are updated. The watches have to either be both not false, or one can be false if the other is the last non-false literal in the clause. The result is a propagation algorithm that takes amortized constant time to propagate, instead of time linear in the size of the clauses.

**Clause Learning.** The `Learn` rule adds entailed clauses to the theory. However, it is intractable to add all entailed clauses to the theory, as this comes down to all possible resolutions between any number of clauses in the theory, which is worst-case exponential. Instead, a "conflict-driven" approach is taken: whenever a conflict is found, resolution is applied on-the-fly to learn a clause that would have implied the same propagation at an earlier decision level, as follows. Consider a clause $c$ has become false, denoted as the *conflict* clause, with literal $l$ the last literal in $c$ that was assigned false. In that case, resolution is possible between $c$ and the clause $c'$, the *reason* for making $l$ false, resulting in a clause $c_1$. Afterwards, this process is repeated until some *Unique Implication Point* (UIP) is reached, a clause $c_i$ in which only one literal was assigned in the current decision level. At a UIP, back-tracking to the level the second-last literal was assigned will result in $c_i$ being unit. Hence, at UIP, the resulting clause, called the *learned* clause, is added to the theory. There are several UIP, with the last one being at the last decision literal itself. Generally, stopping at the first UIP turned out to result in better performance in practice.

**Simplify.** Given a CNF theory and a set of true literals $L$, various simplifications can be applied to the theory, such as:

- erasing clauses with true literals, erasing false literals from clauses,
- erasing clauses that are a superset of other clauses,
- deriving that two literals will always be equivalent and replacing one by the other everywhere (e.g. by detecting clauses $A \Rightarrow B$ and $B \Rightarrow A$),
- ...

Most SAT-solvers apply some of these simplifications, with $L$ the set of literals true at the root level. They can be applied in a pre-processing step or also during search (in which case it is called *in-processing*), for example after *restarts*.

**Simp-true-lit**:
$\langle \mathcal{I}, C\, c\rangle \quad \longrightarrow \langle \mathcal{I}, C\rangle \quad \textbf{if} \quad l^{\mathcal{I}} = \mathbf{t}, level(l) = 0, l \in c$
**Simp-contained**:
$\langle \mathcal{I}, C\, c\rangle \quad \longrightarrow \langle \mathcal{I}, C\rangle \quad \textbf{if} \quad c' \subseteq c, c' \in C$
...

**Restarts.** Now and again, a SAT-solver backtracks to the root level, or *restarts*, and drops part of the learned clauses. Afterwards, search continues normally, but practice has learned that it often goes into a different part of the search space because (**i**) the theory has changed and the heuristic has accumulated more information since last time and (**ii**) because most heuristics include a random component. Restarts are usually initiated after a threshold on the number of conflicts has been reached. After each restart, the threshold is increased. Several approaches exist on how to increase this number and which clauses to delete.[2] Clause deletion might result in the same search space being visited multiple times.

**Restart**:

$$\left\langle M\, l^d\, N, C \right\rangle \quad \longrightarrow \quad \langle M, C \rangle \quad \textbf{if} \quad level(l) = 1 \text{ (no decision in } M\text{)}$$

**Forget**:

$$\left\langle \mathcal{I}, C\, \varphi^l \right\rangle \quad \longrightarrow \quad \langle \mathcal{I}, C \rangle$$

**Efficient Implementation.** To efficiently implement above operations, atoms are represented as natural numbers such that they can be used as indices into arrays mapping atoms to their truth value, reason clause, watched clauses, .... Clauses are then represented as arrays of integers themselves and their watches are kept at the first two positions. In addition to the assignment array (atoms to their truth value), a *trail* is maintained, a chronological list of the assigned literals, and a mapping from decision levels to indices to the position of the decision literal in the trail.

## 5.2 MiniSAT(ID) Transition Rules

In this section, we present the search and optimization algorithm MiniSAT(ID), which takes as input an ECNF theory $\mathcal{T}_g$, an optimization term $c$ with $c$ a constant, and a 3-valued input structure $\mathcal{I}_{in}$. The algorithm is an extension of the existing SAT(ID) search algorithm (no optimization) described in [Mariën et al., 2008], which takes as input a function-free ECNF theory where definitions, if present, are total. In addition, input to MiniSAT(ID) can contain

---

[2] Increasing the threshold is (also) necessary to guarantee completeness of the SAT-solver.

function terms instead of just Boolean atoms, an input structure (originally, a set of unit clauses sufficed) and an optimization term.

First, recall that the ECNF format consists of disjunctions of domain atoms (clauses) and definitions with rules of one the following forms (with all $L_i$'s domain literals and all $e_i$'s constants or domain elements):

$$P(\bar{e}) \leftarrow L_1 \wedge \ldots \wedge L_n \qquad P(\bar{e}) \leftarrow L_1 \vee \ldots \vee L_n$$

$$P(\bar{e}) \leftarrow Q(\bar{e}') \qquad P(\bar{e}) \leftarrow f(\bar{e}) \sim e_0$$

$$P(\bar{e}) \leftarrow agg(\{L_1 : e_1\} \cup \cdots \cup \{L_n : e_n\}) \sim e_0$$

### 5.2.1 Adapting CDCL

We adapt the CDCL schema outlined in Section 5.1 to our ECNF setting. The state consists of

- an ECNF theory $\mathcal{T}_s$, initialized as $\mathcal{T}_g$.
- a sequence $\mathcal{I}$ of true domain literals, ordered by the time at which literals were derived.
- a set *Dec* of *decidable* domain atoms.

Both $\mathcal{I}$ and *Dec* are initialized to the empty set. The former is used to track the progress of the search, the latter to track the atoms the search algorithm is allowed to choose. During search, sentences can be added to $\mathcal{T}_s$, which are annotated by $^f$ if they can be safely forgotten again later.

The standard CDCL rewrite rules are then slightly adapted as follows:

- DECIDE selects domain atoms $a$ from *Dec* that are unknown in $\mathcal{I}$ and adds $a$ or $\neg a$ to $\mathcal{I}$,
- PROPAGATE$_{UP}$ works on clauses in $\mathcal{T}_s$,
- LEARN$_{UP}$ adds $^f$ annotated clauses to $\mathcal{T}_s$,
- FORGET only discards $^f$ annotated clauses from $\mathcal{T}_s$.

To apply learning, the algorithm needs to be able to *explain* propagations in terms of true literals. For unit propagation, this explanation is the clause that triggered propagation. For the additional propagation rules, they will support the generation of such an explanation clause for any of their propagations.

Naturally, the three-valued interpretation $\mathcal{I}_{in}$ is used during search, to guarantee correspondence with the input structure. It will however remain fixed and hence need not be included in the state explicitly.

We say a domain literal or domain term *occurs* in a state if it occurs either in $\mathcal{T}_s$, $\mathcal{I}$ or *Dec*.

In a traditional SAT-solver, *Dec* is not very relevant: atoms are not added dynamically and the theory only consists of clauses over propositional symbols. In our case, symbols will be added dynamically, for example to lazily encode functions by a set of atoms, and $\mathcal{T}_s$ can contain ECNF constructs instead of just propositional atoms. For now, we assume *Dec* always (in any state) contains at least all domain atoms in $\mathcal{T}_s$. In Chapter 7, we show how a more fine-grained treatment of the decidable literals allows search to terminate early.

As an initial step of the algorithm, definitions $\Delta$ in $\mathcal{T}_s$ are simplified. If $\Delta$ is not recursive (or if it can be stratified), it can be split in a set of subdefinitions $\Delta_1, \ldots, \Delta_n$ as shown in [Denecker and Ternovska, 2008]. These are added to $\mathcal{T}_s$ and $\Delta$ is removed from it, to avoid checking for unfounded loops over $\Delta$ during search.

Additional transition rules, presented in the following subsections, then serve to perform propagation on the non-clausal components of $\mathcal{T}_s$. In the SAT(ID) algorithm, the following (sets of) transition rules were already defined:

- COMPLETION adds the completion of a (propositional) definition $\Delta$ to $\mathcal{T}_s$. The rule is executed once for each definition, in the initialization phase.
- PROPAGATE$_{ufs}$ checks for unfounded sets [Van Gelder et al., 1991] in a propositional definition $\Delta$ given $\mathcal{I}$. If an unfounded set $U$ is found, the rule propagates all its atoms as **f** (i.e., it appends $\neg U$ to $\mathcal{I}$.). The propagation to prevent unfounded loops over $U$ is explained in terms of the value of atoms external to the set that could have prevented an unfounded loop.
- AGGREGATE checks for propagation over pseudo-Boolean aggregate expressions by reasoning on the bounds of the aggregate function over $\mathcal{I}$.

Efficient, incremental evaluation of the latter two rules is highly non-trivial. Current state-of-the-art algorithms were presented in [Mariën et al., 2008] and [Gebser et al., 2012b]. For the aggregate rule, we developed an efficient approach in the context of this work, published in [De Cat and Denecker, 2010].

### 5.2.2 Approach to Extend to Full ECNF

We will generate the function-free theory *lazily* during search instead of eagerly up-front. The work is based on the technique of *Lazy Clause Generation (LCG)*, presented in [Stuckey, 2010]. LCG, developed in the context of Constraint Programming, alleviates the blowup of creating the full propositional theory in advance, by only generating the clauses representing (explaining) propagation over that theory when they would contribute to the search, i.e., the moment they would result in propagation. First, it consists of selecting an appropriate encoding of functions as sets of domain atoms. Second, the technique can leverage the existing constraint propagation algorithms to check for propagation on the original constraints, and whenever a propagation is derived, a clause that represents that propagation is added to the propositional theory. Third, in the context of a CDCL algorithm, the clause does not need to be added the moment it would result in propagation, but is in fact only necessary during clause-learning.

Our transition rules apply a slight extension of LCG, namely *lazy constraint generation*, as a sentence is not necessarily decomposed into clauses, but sometimes also in other types of ECNF sentences. In this respect, it is closely related to the work of Drescher et al. [Drescher and Walsh, 2011a] in the context of CASP.

To simplify presentation, we present some transition rules not for the definitional rules as they appear in ECNF but for equivalences ($L \Leftrightarrow c \sim c'$, $L \Leftrightarrow agg(S) \sim c'$, $L \Leftrightarrow P(\bar{e})$ and $L \Leftrightarrow f(\bar{e}) \sim e'$). While they are not directly part of ECNF, they are obtained straightforwardly as the completion of sets of rules. The structure of the presentation of transition rules is as follows:

- Clauses and definitions consisting only of domain literals are covered by existing rules as discussed above.
- The handling of function symbols is covered in Section 5.2.3.
- Sentences of the form $L \Leftrightarrow c \sim c'$ are handled in Section 5.2.4.
- Aggregate sentences of the form $L \Leftrightarrow agg(S) \sim c'$ are handled in Section 5.2.5.
- "General" ground sentences of the form $L \Leftrightarrow P(\bar{e})$ and $L \Leftrightarrow f(\bar{e}) \sim e'$ are handled in Section 5.2.6.
- Last, in Section 5.2.7, we show how to handle definitions that consist not only of domain literals.

We first introduce all rules only considering total functions; in Section 5.2.9, we extend it to partial functions. Section 5.2.11 adds transition rules to solve optimization problems. Afterwards, we discuss how to find multiple (optimal) models.

### Handling Quantifications

A significant part of this thesis is concerned with improving the handling of quantifiers for model expansion and one might wonder how lazy constraint generation fits in this picture. As mentioned above, we are in fact lazily constructing a ground, function-free, clausal theory, call it the *target theory* $\mathcal{T}_{target}$. The question which naturally poses itself is whether this $\mathcal{T}_{target}$ itself can be represented as the result of a grounding process? In fact, the answer is yes, and the transition rules described below can be seen as a grounding algorithm that is highly optimized to delay the grounding of specific types of sentences as much as possible, until the resulting grounding would cause propagation.

Following this idea, we present our transition rules according to the following schema. First, we present a set of (non-ground) sentences of the form $\forall \overline{x} : \varphi \Rightarrow \psi$, with $\varphi$ a conjunction of literals. In principle, we would add all instantiations of this formula to the ground theory $\mathcal{T}_s$. Instead of adding all instances eagerly, we could do this only in case the left-hand is true. Hence, second, we discuss how to quickly find instances of $\overline{x}$ for which $\varphi$ holds in $\mathcal{I}$ and $\psi$ does not, and how to check that no such instance exists. Last, if $\psi$ is a (conjunction of) literal(s), when $\varphi$ is true, we could add $\psi$ to the interpretation directly and only effectively generate the instance when an explanation is required for the propagation. Thus, third, for each rule we discuss when the instances are effectively added to $\mathcal{T}_s$.

The aim of such a presentation is two-fold. First, presenting the original theory gives a formal account of what the algorithms derive, making it easier to check correctness. Second, in Chapter 7, we develop a more general framework to interleave grounding and search, by automatically deriving conditions on when sentences should (not yet) be grounded during search. After reading Chapter 7, it will become clear that the techniques presented here are a highly optimized instantiations of the general framework, which delays more grounding and is more efficient than the general algorithm for the specific classes of sentences considered here.

### 5.2.3 Function Symbols

To handle sentences containing function term $f/n$ with codomain $D = \{d_1, \ldots, d_n\}$ (recall, all types are interpreted in $\mathcal{I}_{in}$) in a solver that only decides on domain atoms, we use the *order encoding* [Tamura et al., 2009]. Domain term $c$ of the form $f(\overline{d})$ are encoded through a new unary predicate symbol $E_{c\leq}$, where the truth of atoms $E_{c\leq}(d)$ expresses that $c \leq d$ holds. Trivially, any atom $E_{c\leq}(d)$ with $d \notin D$ is interpreted true if $d > d_n$, false if $d < d_1$ and as $E_{c\leq}(d')$ otherwise, with $d'$ the domain element in $D$ closest to and smaller than $d$.

In the sequel, we use $\lceil c \leq v \rceil$, with $v$ a variable or domain element, to denote the atom $E_{c\leq}(v)$. We introduce the (partial) functions *next* and *prev* which take a domain element and map it to the next, respectively previous, element in the order. Other inequality operators can then be defined in terms of $\leq$:

$$\lceil c \leq v \rceil \equiv E_{c\leq}(v) \qquad \lceil c \geq v \rceil \equiv \neg \lceil c < v \rceil$$

$$\lceil c < v \rceil \equiv \lceil c \leq next(v) \rceil \qquad \lceil c > v \rceil \equiv \lceil c \geq prev(v) \rceil$$

For (dis)equality, we introduce a new unary predicate symbol $E_{c=}$, defined as

$$\{\forall x : E_{c=}(x) \leftarrow \lceil c \leq x \rceil \land \lceil c \geq x \rceil\}.$$

We then use $\lceil c = v \rceil$ as a shorthand for $E_{c=}(v)$ and $\lceil c \neq v \rceil$ for $\neg \lceil c = v \rceil$.[3]

**Definition 5.2.1** (Bounds and range)**.** For each domain term $c$ and structure $\mathcal{I}$, we define the *minimum bound* $\min_c(\mathcal{I})$ and *maximum bound* $\max_c(\mathcal{I})$ as $\min(\{d \in D \mid \lceil c \leq d \rceil^{\mathcal{I}} = \mathbf{t}\}$ and $\max(\{d \in D \mid \lceil c \geq d \rceil^{\mathcal{I}} = \mathbf{t}\})$, respectively. If the set is empty, $\min_c(\mathcal{I}) = d_n$ and $\max_c(\mathcal{I}) = d_1$.

The bounds indicate the range between which $c$ can still take values (in $D$ naturally). The values $\min_c(\mathcal{I})$ and $\max_c(\mathcal{I})$ can be computed from $\mathcal{I}$ in every state, but an efficient algorithm should store and adapt them incrementally, as their values are required often, to evaluate propagation over other sentences.

The order encoding for a (total) function then consists of the sentences

$$\forall x \in D \setminus \{d_n\} : \forall y \in D : \lceil c \leq x \rceil \land y > x \Rightarrow \lceil c \leq y \rceil \text{, and} \tag{5.1}$$

$$\forall x \in D \setminus \{d_1\} : \forall z \in D : \lceil c \geq x \rceil \land z < x \Rightarrow \lceil c \geq z \rceil . \tag{5.2}$$

---

[3] In fact, allowing atoms over $E_{c=}$ results in a hybrid of the order and the unary encoding. However, we will only use equalities if they are explicitly present in the input and as they are explicitly defined in terms of the order encoding, we will usually refer to presented approach as the order encoding.

The propagation rule ENCODE is responsible to add part of the grounding of the above formulas to $\mathcal{T}_s$. It is applied to a domain term $c$ the first time it occurs in the state. Two approaches are used. (**i**) For small codomains $D$ ($|D| < 100$), the encoding is added to $\mathcal{T}_s$ eagerly, by instantiating sentence 5.1 for all instantiations of $x$ and for instantiations of $y = next(x)$. (**ii**) For larger (and infinite) codomains, the eager approach is likely to explode the grounding. Instead, we construct it lazily (cfr. [Stuckey, 2010]). If the codomain is finite, the unit clauses $\lceil c \leq d_n \rceil$ and $\lceil c \geq d_1 \rceil$ are added. Whenever an atom $\lceil c \leq d \rceil$ occurs in the state (e.g., because it was propagated by some constraint), we instantiate $x$ with $d$ and $y$ with the closest larger $d'$ ($z$ with the closest smaller $d'$) for which $\lceil c \leq d' \rceil$ occurs in the state if it exists. It follows that in the infinite case, the minimum and maximum bound are undefined in some interpretations.

**Proposition 5.2.2.** *Both approaches (**i**) and (**ii**) are complete: in any state, any literal that can be propagated from the full instantiation of the encoding can also be propagated from the instances made by (**i**), respectively (**ii**).*

Note that approach (**ii**) in the limit (all atoms have been introduced) results in a grounding that is twice as large as the grounding of approach (**i**).

For the lazy encoding however, there is one issue left: it is perfectly possible that the ENCODE rule is at fixpoint and all atoms $\lceil c \leq d \rceil$ have been assigned, but the codomain of possible values for $c$ is not a singleton. Indeed, we need a way to introduce more atoms $\lceil c \leq d \rceil$ as long as the CDCL algorithm has not assigned $c$ a single value. We add the rule ADDATOM:

**ADDATOM:**

$$\mathcal{I} \mid Dec \longrightarrow \mathcal{I} \mid Dec :: \lceil c \leq d \rceil \ \textbf{if} \begin{cases} \text{atoms } \lceil c \leq d' \rceil \text{ in } Dec \text{ are assigned,} \\ c \text{ is not assigned, and} \\ \lceil c \leq d \rceil \notin Dec, d \in D \end{cases}$$

The rule introduces an additional atom of the encoding of $c$ if all introduced ones are assigned but do not yet fix the value of $c$. We select the domain element $d$ in $D$ either randomly from those for which $\lceil c \leq d \rceil \notin Dec$ or, if the interval between the current bounds is finite, we alternate between selecting a value in the middle between the bounds or a value next to one of the bounds. The latter approach keeps the possibility open that $c$ can be assigned without introducing more of its codomain (if $\lceil c \leq d \rceil$ becomes true and $\lceil c \leq prev(d) \rceil$ false).

**Proposition 5.2.3.** *In any state in which all atoms $\lceil c \leq d \rceil$ in the state are assigned and* ENCODE *and* ADDATOM *are at fixpoint, each $c$ is assigned in $\mathcal{I}$.*

Similarly to approach (**ii**), the rule defining equality atoms ($\forall x : E_{c=}(x) \leftarrow \lceil c \leq x \rceil \land \lceil c \geq x \rceil$) is instantiated lazily by ENCODE: $x$ is instantiated with $d$ whenever an atom $\lceil c = d \rceil$ occurs in the state for the first time.

A model $\mathcal{M}$ can be obtained from a satisfying state by starting from $\mathcal{I}_{in} \cup \mathcal{I}$ and expanding it by interpreting domain terms $c$ that occur in the state by $\min_c(\mathcal{I})$ (in any satisfying state, $\min_c(\mathcal{I})$ will be equal to $\max_c(\mathcal{I})$). This construction of $\mathcal{M}$ is the task of the EXTRACT-MODEL rule, which then adds $\mathcal{M}$ to the set of models already found.

**Example 5.2.4.** Consider the theory $\mathcal{T}_g$ consisting only of the sentence $P \Leftrightarrow f(1) \leq 3$, with $f$ typed as $f(\tau) : \tau'$ and $\tau'$ interpreted as $D = \{1, 2, 3\}$. The rule ENCODE adds the following sentences to $\mathcal{T}_s$.

$$\begin{array}{ll} \lceil f(1) \leq 1 \rceil \Rightarrow \lceil f(1) \leq 2 \rceil & \lceil f(1) \leq 2 \rceil \Rightarrow \lceil f(1) \leq 3 \rceil \\ \lceil f(1) \leq 3 \rceil & \lceil f(1) \geq 1 \rceil \\ \lceil f(1) \geq 2 \rceil \Rightarrow \lceil f(1) \geq 1 \rceil & \lceil f(1) \geq 3 \rceil \Rightarrow \lceil f(1) \geq 2 \rceil \end{array}$$

Note that it does not add instantiations for any other term $f(d)$, $d \neq 1$, which is important if $\tau$ has a large interpretation. Hence, the eventual result of model expansion is partial, as it does not interpret those other domain terms. Any interpretation for those terms results in a model of the theory. For example, the interpretation $\mathcal{I} = \{P, \lceil f(1) \leq 3 \rceil, \lceil f(1) \geq 3 \rceil\}$ is precise enough: all structures more precise than $\mathcal{I}$ are models of $\mathcal{T}_g$.

If also $\tau'$ has a large interpretation, say $D = \{1, \ldots, 1.000.000\}$, the initial call to ENCODE only adds the sentences $\lceil f(1) \leq 1.000.000 \rceil$ and $\lceil f(1) \geq 1 \rceil$. As both will then be propagated but the value of $f(1)$ is not fixed yet, ADDATOM introduces the domain atom $\lceil f(1) \leq 500.000 \rceil$ and ENCODE adds the sentences $\lceil f(1) \geq 500.000 \rceil \Rightarrow \lceil f(1) \geq 1 \rceil$ and $\lceil f(1) \leq 500.000 \rceil \Rightarrow \lceil f(1) \leq 1.000.000 \rceil$. Afterwards, additional atoms are only introduced when $\lceil f(1) \leq 500.000 \rceil$ has been decided (except that $\lceil f(1) \leq 3 \rceil$ will be added because of the sentence $P \Leftrightarrow f(1) \leq 3$).

The order encoding is selected over encoding the function as its graph $E_{c=}(d)$ as the encoding of inequalities is smaller, the encoding of more constraints is more compact, and choices on encoding atoms more often eliminate subsets of the codomain (instead of just one value). A disadvantage is that performance can degrade for specifications in which the order on elements has no relevant meaning in the original application domain. A more in-depth comparison can be found, e.g., in [Stuckey, 2010].

We often do not annotate the added sentences as "learned" in $\mathcal{T}_s$. This is on purpose: it is not necessary to track whether they are still present or might have to be added again by their corresponding rules.

### 5.2.4 Comparison Constraint

The transition rule COMPARISON applies to *comparison* constraints $P \Leftrightarrow c \leq c'$, with $P$ a domain atom and $c$ and $c'$ domain terms with codomains $D$ and $D'$, respectively. The propagations we consider over such a constraint can be represented as the following sentences.

$$\begin{array}{lll}
\forall x \in D \cup D' : & \lceil c \leq x \rceil \wedge \lceil c' \geq x \rceil & \Rightarrow P. \\
\forall x \in D \cup D' : & \lceil c > x \rceil \wedge \lceil c' < x \rceil & \Rightarrow \neg P. \\
\forall x \in D : & \lceil c' \leq x \rceil \wedge P & \Rightarrow \lceil c \leq x \rceil . \\
\forall x \in D : & \lceil c' \geq x \rceil \wedge \neg P & \Rightarrow \lceil c > x \rceil . \\
\forall x \in D' : & \lceil c \geq x \rceil \wedge P & \Rightarrow \lceil c' \geq x \rceil . \\
\forall x \in D' : & \lceil c \leq x \rceil \wedge \neg P & \Rightarrow \lceil c' < x \rceil .
\end{array}$$

These sentences, together with the encoding of $c$ and $c'$, are $\{\Sigma, \mathcal{I}\}$-equivalent to the original constraint.

Instantiations of the sentences are generated as follows. Rule COMPARISON checks for each of the non-ground sentences whether the body is true and the head isn't, for instantiations of $x$ with $\min_c$, $\max_c$, $\min_{c'}$ and $\max_{c'}$. This is checked whenever one of those values increases (for the minimum bound) or decreases (for the maximum bound) and whenever $P$ becomes assigned.

It is well-known that reasoning on the bounds for a comparison constraint results in complete propagation. In other words, when PROPAGATE$_{UP}$, ENCODE, ADDATOM and COMPARISON are at fixpoint, none of the above sentences has an instantiation with a true left-hand side and a false or unknown right-hand side.

**Example 5.2.5.** Consider a constraint $P \Leftrightarrow c \leq c'$, where $c$ has codomain $[3, 10]$, $c'$ has codomain $[7, 20]$ and $P$ is true in $\mathcal{I}$. When $\mathcal{I}$ is expanded to $\mathcal{I}'$ by making $\lceil c \geq 8 \rceil$ true, COMPARISON checks for $x = 8$ which of the left-hand sides are true, which is the case for the sentence $\lceil c \geq 8 \rceil \wedge P \Rightarrow \lceil c' \geq 8 \rceil$. As $P$ is also true in $\mathcal{I}'$, the sentence is added to $\mathcal{T}_s$ and PROPAGATE$_{UP}$ derives $\lceil c' \geq 8 \rceil$.

Sentences for comparison operators other than $\leq$ are not necessary, as they can be rewritten (in a preprocessing step) into sentences of the form $P \Leftrightarrow c \leq c'$:

$$P \Leftrightarrow c \geq c' \equiv P \Leftrightarrow c' \leq c$$

$$P \Leftrightarrow c > c' \equiv (\neg P) \Leftrightarrow c \leq c'$$

$$P \Leftrightarrow c < c' \equiv (\neg P) \Leftrightarrow c \geq c'$$

$$P \Leftrightarrow c = c' \equiv \begin{cases} P \Leftrightarrow P_1 \wedge P_2, \\ P_1 \Leftrightarrow c \leq c', \\ P_2 \Leftrightarrow c' \leq c \end{cases}$$

$$P \Leftrightarrow c \neq c' \equiv (\neg P) \Leftrightarrow c = c'$$

### 5.2.5 Aggregates

Next, we introduce propagation rules for sentences of the form $P \Leftrightarrow agg(\{L_1 : e_1\{\cup \ldots \cup \{L_n : e_n\}) \leq e'$ where $agg$ is an aggregate function. We refer to $L_i$ as *condition* literals and $e_i$ as *weight* terms.

As above, other comparison operators can be rewritten into constraints over $\leq$. Cardinality aggregates are rewritten into sum aggregates, as any cardinality term $\#(\{L_1, \ldots, L_n\})$ is equivalent to a sum term $sum(\{L_1 : 1\} \cup \ldots \cup \{L_n : 1\})$. The rules for minimum and product aggregates are not presented, as they are similar to those for maximum and sum respectively (the treatment of product aggregates is further complicated by the non-monotonicity of product for terms with negative values).

The rule ENCODE$_{\text{MAX}}$ rewrites the sentence $P \Leftrightarrow \max(\{L_1 : e_1\} \cup \ldots \cup \{L_n : e_n\}) \leq e'$ into the sentences

$$\begin{array}{ll} P \wedge L_i \Rightarrow e_i \leq e' & \text{for each } i \in [1,n]\text{, and} \\ \neg P \Rightarrow \bigvee_{i \in [1,n]} (L_i \wedge e_i > e') & \end{array}$$

The rewriting preserves equivalence and captures all possible propagation. As the rewriting consists of only $n + 1 + 3 \times n$ ground clauses, it is done eagerly for all such sentences in $\mathcal{T}_s$.[4]

---

[4] The $3 \times n$ factor arises because of the encoding of $n$ Tseitin equivalences, required to obtain a clausal representation.

The rule $\textsc{Encode}_{\mathbf{SUM}}$ enforces *bounds* consistency on sentences $P \Leftrightarrow \sum(S) \leq 0$.[5] Similarly to above, these bounds are defined as follows.

**Definition 5.2.6** (aggregate bounds)**.** The minimum bound of a term $\sum(S)$ in an interpretation $\mathcal{I}$ with $S$ the set $\{L_1 : e_1\} \cup \ldots \cup \{L_n : e_n\}$, denoted $min_{\sum(S)}(\mathcal{I})$, is defined as

$$min_{\sum(S)}(\mathcal{I}) = \sum_{i|i\in[1,n],L_i^{\mathcal{I}}=\mathbf{t}} min_{e_i}(\mathcal{I}) + \sum_{i|i\in[1,n],L_i^{\mathcal{I}}=\mathbf{u}} min(0, min_{e_i}(\mathcal{I})).$$

Similarly, the maximum bound $max_{\sum(S)}(\mathcal{I})$ is defined as

$$max_{\sum(S)}(\mathcal{I}) = \sum_{i|i\in[1,n],L_i^{\mathcal{I}}=\mathbf{t}} max_{e_i}(\mathcal{I}) + \sum_{i|i\in[1,n],L_i^{\mathcal{I}}=\mathbf{u}} max(0, max_{e_i}(\mathcal{I})).$$

The following propagations are applied to sentences $P \Leftrightarrow \sum(S) \leq 0$:

**Head propagation** $P$ is derived if $max_{\sum(S)}(\mathcal{I}) \leq 0$; if $min_{\sum(S)}(\mathcal{I}) > 0$, then $\neg P$ is derived.

**Condition propagation** If $P$ is true, then $L_i$ is derived if $min_{\sum(S)}(\mathcal{I}) \leq 0$ and $min_{\sum(S-L_i,e_i)}(\mathcal{I}) > 0$. If both conditions are false, $\neg L_i$ is derived. Similar cases apply if $P$ is false (reasoning on *max* bounds).

**Weight propagation** If $P$ is true, then $e_i \leq n$ is derived if $min_{\sum(S)}(\mathcal{I}) \leq 0$ and $min_{\sum(S-L_i,e_i+L_i,n+1)}(\mathcal{I}) > 0$. A similar case applies if $P$ is false (reasoning on *max* and deriving $e_i \geq n$).

It is easy to see that no weight propagation applies if the corresponding condition is false. Naturally, we only need to propagate the minimum, respectively maximum, $n$ for which the property holds.

These propagations cannot be expressed as a compact set of FO sentences. Indeed, we need to express properties about subsets of $S$, for which second-order logic is required. We do not go into further details here.

The reason for propagating $P$ can be derived by analysing which assignments reduced $max_{\sum(S)}(\mathcal{I})$ sufficiently such that it is now below 0. A sufficient explanation (which is always the case) are $L_i$ (if true in $\mathcal{I}$), $\neg L_i$ (if true in $\mathcal{I}$), $\lceil e_i \leq \max_{e_i}(\mathcal{I})\rceil$ and $\lceil e_i \geq \min_{e_i}(\mathcal{I})\rceil$, for every $i$ in $[1, n]$. However, such an explanation is overly large (and hence too specific for proper learning).

---

[5] An atom $\sum(S) \leq e'$ is equivalent with $\sum(S \cup \{\top : e'\}) \leq 0$.

**Example 5.2.7.** Consider the sentence $P \Leftrightarrow sum(\{L_1 : c_1\} \cup \{L_2 : c_2\}) \leq 0$, with $[-10, 10]$ the codomain of both $c_1$ and $c_2$. Under the interpretation $\{\lceil c_1 \leq -1 \rceil, \neg L_1, \lceil c_2 \leq -10 \rceil\}$, the sentence entails $P$. Its explanation clause is

$$\lceil c_1 \leq -1 \rceil \land \neg L_1 \land \lceil c_2 \leq -10 \rceil \Rightarrow P,$$

while the subset-minimal explanation clause is in fact

$$\neg L_1 \land \lceil c_2 \leq -10 \rceil \Rightarrow P.$$

Various strategies have been proposed to select a more appropriate explanation (see, e.g., [Stuckey, 2010]):

1. Add literals in the order / inverse order of the trail, until the set explains the propagation.
2. Construct the explanation in the order of the largest/smallest current weight, until the set explains the propagation.
3. Prefer literals assigned in the current decision level.
4. Prefer literals already in the partially built learned clause.
5. Apply a (subset)-minimization step in which it is attempted to remove literals from the explanation.

Propagation is checked whenever the interpretation of the head, one of the conditions or the bounds of one of the terms change. Some time was spend to develop a more specific approach to reduce the overhead of checking for propagation. The core idea was to derive an (approximately) minimal subset of watches in each state, as illustrated in the following example.

**Example 5.2.8.** Consider a sentence $\top \Leftrightarrow \#(\{L_1, \ldots, L_m\}) \geq n$, with $n$ a fixed integer, a type of constraint that frequently occurs in, e.g., scheduling and configuration applications. Here, we do not need to check for propagation whenever any one of the literals $L_i$ changes. Indeed, a generalization of the 2-watched literal can be applied, where we watch only $n + 1$ literals $L_i$. It can be shown that as long as none of those are false, no propagation ensues.

However, the more intelligent approach turned out to significantly complicate implementation and provided only a small performance gain on a small number of benchmarks. Consequently, this branch of research was not investigated further.

### Optimizations

The implementation of aggregate propagation can be highly optimized in case all weights are known in advance (so-called pseudo-Boolean aggregates). It is for example possible to keep the aggregate set sorted by weights, which allows the weight from which propagation would ensue to be found in time logarithmic in the size of the set. In such case, it becomes worthwhile to efficiently update the aggregate bounds when the interpretation changes, instead of recomputing it. Furthermore, weights of the form $n \times t$ with $n$ a known integer can be handled straightforwardly as part of the sum rules, without the need for introducing an additional constant for the product. The optimized transition rules are presented in [De Cat and Denecker, 2010].

VSIDS is adapted to bump the value of literals that are propagated using $\textsc{Encode}_{\textbf{sum}}$, which turned out to improve search. Aggregate sentences over small sets (less than 4 literals) are compiled directly to SAT using techniques presented in [Codish et al., 2011].

## 5.2.6 General Ground Atoms

Sentences of the form $P \Leftrightarrow Q(\bar{e})$ and $P \Leftrightarrow f(\bar{e}) \sim e'$, with at least one element of $\bar{e}$ a domain term (recall, $e_i$ is either a domain element or a domain term) are handled by the transition rule $\textsc{Encode}_{\textbf{general}}$. The rule waits until all domain terms in $\bar{e}$ are assigned. At that moment, an instantiation of one of the following sentences is generated:[6]

$$\begin{array}{l} \forall \bar{x} \in dom_{\bar{e}} : \lceil \bar{e} = \bar{x} \rceil \Rightarrow (P \Leftrightarrow Q(\bar{x})), \text{respectively} \\ \forall \bar{x} \in dom_{\bar{e}} : \lceil \bar{e} = \bar{x} \rceil \Rightarrow (P \Leftrightarrow f(\bar{x}) \sim e'). \end{array}$$

Tseitin introduction is applied to generate sentences in ECNF. Note that in case of $P \Leftrightarrow f(\bar{e}) \sim e'$, we need not wait until $e'$ is instantiated.

**Example 5.2.9.** The *element constraint* `element`($c$,$A$,$i$), from the field of CP, expresses that a constant $c$ takes the value at index $i$ of array $A$. An array can be seen as a function $f_A$ from indices to values. Such a constraint can then be expressed in ECNF as the sentence $f_A(i) = c$ and handled lazily as described above: nothing happens until $i$ has been assigned; when it is assigned to an element $d$, the single comparison sentence $f_A(d) = c$ is generated and added to $\mathcal{T}_s$. It is then handled by the rules introduced earlier.

This approach also works when $A$ ($f_A$) is very large or not completely known in advance, which CP systems typically cannot handle efficiently.

---

[6] We use $\lceil \bar{e} = \bar{x} \rceil$ as a shorthand for $\bigwedge_{i \in [1,|\bar{e}|]} \lceil e_i = x_i \rceil$.

It is sometimes possible to derive propagation even before $\bar{e}$ is completely instantiated, for example if $Q$ or $f$ are partially interpreted. For example, ENCODE$_{\text{GENERAL}}$ is optimized to handle basic arithmetic functions ($+$,$-$,$*$,$/$, minimum and maximum) by transforming the atom $f(\bar{e}) \sim e'$ (with $f$ an arithmetic function) into the equivalent aggregate atom. E.g., $P \Leftrightarrow e_1 - e_2 \sim e_3$ is transformed into the equivalent $P \Leftrightarrow sum(\{\mathbf{t} : e_1\} \cup \{-1 \times e_2\}) \sim e_3$. It is part of future work to develop an efficient way to check for more propagation for general symbols $Q$ and $f$, depending on the bounds of $\bar{e}$, $e'$ and the current interpretation of $f$ and $Q$.

**Example 5.2.10.** Even the quite straightforward rule as presented here can be indispensable. For the following riddle, IDP was unable to solve the task without ENCODE$_{\text{GENERAL}}$ as constructing the grounding was intractable. The riddle goes as follows: "To determine my age, it suffices to know that my current age in 2013 is halfway between two consecutive primes, that my age's prime factors do not sum to a prime number, and that I was born in a prime year.".

In IDP, this can be modeled as

```
vocabulary V is {
   type Nb isa int ;
   func Age[−>Nb];
   pred Prime(Nb);
   func YearOfBirth[−>Nb];
}
theory T over V is {
   { Prime(x) <− x>1 & !y: 1 < y < x => ~ (x % y = 0) }

   Age = 2013−YearOfBirth;
   Prime(YearOfBirth);

   ?x1 x2: Prime(x1) & Prime(x2) & x1 < Age < x2 &
          ~(?y: Prime(y) & x1 < y < x2) & Age = (x2 + x1)/2;

   ~Prime(sum { x : Prime(x) & 1 < x =< Age & Age % x = 0 : x });
}
structure S over V is { Nb = {0..2013} }
```

Unexpectedly (to the original riddler), IDP proved that there is not a unique solution: 48 different solutions exist! There is only one with an age below 100 however, namely $Age = 26$. Without any manual optimizations (e.g., by restricting the range of $Age$ or $YearOfBirth$), IDP takes half a second to find a solution. If skolemization is disabled (`x1` and `x2` can be skolemized), this time increases to a bit over 20 seconds; also disallowing general ground atoms blows up the grounding so much it becomes too large to construct.

### 5.2.7 Definitions with Function Terms

In the standard case (no function terms), definitions are handled by applying the rules COMPLETION and PROPAGATE$_{ufs}$ and the new rule WELLFOUNDED. The latter rule *checks* whether a definition $\Delta$ still has a two-valued model, by applying the complete, bottom-up well-founded construction to $\Delta$ as described in [Van Gelder, 1993], given a two-valued interpretation of the opens of $\Delta$. If the definition has a three-valued well-founded interpretation, the negation of all open literals is raised as conflict. For total definitions, WELLFOUNDED can be dropped as it will never result in a conflict.

To handle definitions which contain function terms, we introduce the rules COMPLETION′, UNFOUNDED′ and WELLFOUNDED′. These are extended versions of the above ones (and, hence, incorporate their functionality).

Consider a definition $\Delta$ defining, among others, the symbol $P$ by the rules $\{P(\overline{e}_1) \leftarrow \varphi_1, \dots, P(\overline{e}_n) \leftarrow \varphi_n\}$. The completion of $P$ for $\Delta$ is then the (non-ground) sentence $\forall \overline{x} : P(\overline{x}) \Leftrightarrow \bigvee_{i \in [1,n]} \lceil \overline{e}_i = \overline{x} \rceil \land \varphi_i$. The rule COMPLETION′ adds the equivalent sentences

$$\varphi_i \Rightarrow P(\overline{e}_i) \qquad \text{for each } i \in [1,n] \tag{5.3}$$

$$\forall \overline{x} : P(\overline{x}) \Rightarrow \left( \bigvee_{i \in [1,n]} \lceil \overline{e}_i = \overline{x} \rceil \land \varphi_i \right) \tag{5.4}$$

The former sentence is added eagerly for each $i$ (as it is already ground).[7] The latter sentence is added lazily, by instantiating $\overline{x}$ with $\overline{d}$ whenever an atom $P(\overline{d})$ becomes true (for the first time).

An issue with the condition on the instantiation of sentence 5.4 is that propagations might be missed. Indeed, the sentence is not instantiated as long as $P(\overline{d})$ is not true; however, if $\left(\bigvee_{i \in [1,n]} \left\lceil \overline{e}_i = \overline{d} \right\rceil \land \varphi_i\right)$ is false, then $\neg P(\overline{d})$ is entailed. If $P(\overline{d})$ does not occur in $\mathcal{T}_s$ (and is never added by other rules), it might remain unknown, even when search has finished. The resulting interpretation is then not a model as not all two-valued extensions are models. To resolve this, we first note that search does not finish until all domain terms

---

[7] This might require Tseitin introduction to obtain ECNF sentences.

in $\bar{e}_i$, for all $i$, are decided and all sentences 5.3 are satisfied (as they are part of $\mathcal{T}_s$). In that case, there are no more sentences in the completion that could force an unknown domain atom $P(\overline{d})$ to be true. As $P$ is defined, we then know that $P(\overline{d})$ (and thus all unassigned domain atoms over $P$) have to be false. Extending the interpretation in this way, denoted as the rule DEFINED-FALSE, restores soundness.

For UNFOUNDED′ and WELLFOUNDED′, we take an approach similar to the approach for general ground atoms: we delay the application of both rules for a definition $\Delta$ until all domain terms occurring in $\Delta$ are assigned. In such situations, replacing all domain terms in $\Delta$ with their interpretation results in a definition $\Delta_{inst}$ to which the existing definition rules PROPAGATE$_{ufs}$ and WELLFOUNDED can be applied (recall that WELLFOUNDED is anyway only applied when all open symbols are known). It only remains to properly adapt explanations $EC$ generated by these rules: the explanations are only valid conditionally because we had to substitute constants with domain elements in order to obtain $\Delta_{inst}$. So instead of adding $EC$ to $\mathcal{T}_s$, we add

$$\bigwedge_{c|c \text{ occurs in } \Delta} \left\lceil c = c^{\mathcal{I}} \right\rceil \Rightarrow EC.$$

**Example 5.2.11.** Consider part of a graph application consisting of a function *next* mapping nodes to nodes and a constant *start* of type node. Suppose the aim is to compute the reachability relation $r$ of nodes reachable from *start* through *next*, defined as $\{r(start), \quad \forall x : r(next(x)) \leftarrow r(x)\}$.[8] In the context of an interpretation $\mathcal{I} = \{start = a, next(a) = b, next(b) = a, next(c) = c\}$ with domain $\{a, b, c\}$, the definition reduces to the rules $r(a)$, $r(b) \leftarrow r(a)$, $r(a) \leftarrow r(b)$, and $r(c) \leftarrow r(c)$, to which PROPAGATE$_{ufs}$ can be applied. PROPAGATE$_{ufs}$ derives $\neg r(c)$, with the associated explanation $(\lceil start = a\rceil \wedge \lceil next(a) = b\rceil \wedge \lceil next(b) = a\rceil \wedge \lceil next(c) = c\rceil) \Rightarrow \neg r(c)$.

There is one issue left: we have to check whether the value of all domain elements in the head of $\Delta_{inst}$ are within the type associated to the position in which they occur. Indeed, according to the semantics, out-of-type is treated as if the according body is false. Hence, types are checked against $\mathcal{I}_{in}$ and those rules are removed from $\Delta_{inst}$.

---

[8] The size of the grounding of this definition is linear in the size of the domain, instead of quadratic if functions would be graphed.

### 5.2.8 Pre-interpretation Over Some Symbols

As discussed above, the grounding algorithm gets as input a partial, consistent input interpretation $\mathcal{I}_{in}$, parts of which can be implicit (e.g., interpretations of numerical functions) or described symbolically (e.g., ranges $0..n$). The information in $\mathcal{I}_{in}$ should be passed to the search algorithm, but we do not want to add $\mathcal{I}_{in}$ as constraints to the theory, for the same reason as we do not want to eagerly generate the full propositional grounding.

Instead, the following transition rule takes care of adding just enough of $\mathcal{I}_{in}$ to obtain interpretations that are consistent with it. Rule CHECK-$\mathcal{I}_{in}$ adds a clause $A$ or $\neg A$ to $\mathcal{T}_s$ for every domain atom $A$ that occurs in the state for which $A^{\mathcal{I}_{in}} = \mathbf{t}$, respectively $A^{\mathcal{I}_{in}} = \mathbf{f}$.[9] Similarly, for a domain term $c$ that occurs in the state, CHECK-$\mathcal{I}_{in}$ adds atoms $\lceil c = d \rceil$ or $\lceil c \neq d \rceil$ to $\mathcal{T}_s$ if $d \in c^{\mathcal{I}}_{in}$, respectively $d \notin c^{\mathcal{I}}_{in}$. Again, these clauses are not annotated as learned clauses and, hence, it suffices to apply CHECK-$\mathcal{I}_{in}$ once for any domain atom or domain term whenever it is added to the state.

**Example 5.2.12.** Consider a theory $\mathcal{T}_s$ with constraint $P(c) \lor \neg P(c')$, with $P$ over a large domain $D$ and interpreted in $\mathcal{I}_{in}$. Adding clauses $P(d)$ or $\neg P(d)$ for every domain element $d \in D$ would blow up the size of the grounding. However, lazily adding such an atom whenever a value for $c$ or $c'$ is chosen, results in a theory where only the relevant literals are asserted.

### 5.2.9 Partial Function Symbols

For partial functions, a bit more care is required. We assume that for every domain term $c$ over a partial function, a unique propositional atom exists that is (implicitly) defined as $\exists x : c = x$. As this coincides with the meaning of our shorthand for whether terms denote, we will refer to that atom also as $denotes(c)$ .

The encoding scheme is adapted as follows: (**i**) the order encoding is kept identical, (**ii**) the minimum and maximum bounds become partial functions, which are undefined if $c$ is not denoting. (**iii**) a symmetry breaker is added, which fixes the interpretation of the encoding atoms if $c$ is not denoting:

$$\forall x \in D : \neg denotes(c) \Rightarrow \lceil c \leq x \rceil .$$

Indeed, the sentence forces all (introduced) atoms over $E_{c\leq}$ to be true if $denotes(c)$ is false.

---

[9] By definition of "false in an interpretation", CHECK-$\mathcal{I}_{in}$ automatically takes care of well-typedness checking: if, for an atom $P(\overline{d})$, one of its arguments is not in the interpretation of the associated type, $P(\overline{d})$ is false in $\mathcal{I}_{in}$ and, hence, $\neg P(\overline{d})$ wil be added to $\mathcal{T}_s$ by CHECK-$\mathcal{I}_{in}$.

Models are constructed by interpreting $c$ with $\min_c(\mathcal{I})$ if $c$ is total *or* if $denotes(c)$ is true in $\mathcal{I}$; otherwise, $c$ is interpreted by **undef**.

To correctly handle partial function occurrences, we adapt the transition rules as follows.

**(Dis)equality.** The definition of $E_{c=}$ is adapted by adding $denotes(c)$ to the body of its defining rule.

**Comparison.** For a comparison constraint of the form $P \Leftrightarrow c \leq c'$, the following sentences are added, which express the relation between $P$ and whether $c$ and $c'$ are denoting.

$$\begin{array}{ll}
\neg denotes(c) & \Rightarrow \neg P \\
\neg denotes(c') & \Rightarrow \neg P \\
P & \Rightarrow denotes(c) \land denotes(c')
\end{array}$$

These sentences are naturally added eagerly. Next, there are three original sentences with $\neg P$ on the left-hand side or $P$ on the right-hand side: they are changed to reflect that they should only fire if the terms are denoting:

$$\begin{array}{lll}
\forall x \in D \cup D' : & \lceil c \leq x \rceil \land \lceil c' \geq x \rceil & \Rightarrow P. \\
\forall x \in D : & \lceil c' \geq x \rceil \land \neg P & \Rightarrow \lceil c > x \rceil . \\
\forall x \in D' : & \lceil c \leq x \rceil \land \neg P & \Rightarrow \lceil c' < x \rceil .
\end{array}$$

These have to be changed to reflect that the implications only hold if the terms are denoting. This results in the sentences

$$\begin{array}{lll}
\forall x \in D \cup D' : & \lceil c \leq x \rceil \land \lceil c' \geq x \rceil \land denotes(c) \land denotes(c') & \Rightarrow P. \\
\forall x \in D : & \lceil c' \geq x \rceil \land \neg P \land denotes(c) \land denotes(c') & \Rightarrow \lceil c > x \rceil . \\
\forall x \in D' : & \lceil c \leq x \rceil \land \neg P \land denotes(c) \land denotes(c') & \Rightarrow \lceil c' < x \rceil .
\end{array}$$

**Aggregates.** The considered aggregate functions themselves are total, but terms in the set or on the right-hand side can be over partial functions. The right-hand side is handled similarly to the approach in comparison sentences. For possibly non-denoting terms in the set, recall that they have the same effect as if the condition was false. Hence, in $\textsc{Encode}_{\textbf{MAX}}$, it suffices to replace atoms $L_i$ by $L_i \land denotes(c_i)$. For $\textsc{Encode}_{\textbf{SUM}}$, we change the definition of $\min_{agg(S)}$ and $\max_{agg(S)}$ to treat $\neg denotes(c_i)$ as if $L_i$ is false. Note that explanations will then also contain atoms $(\neg)denotes(c_i)$.

**General Ground Atoms.** The original encoding for a sentence $P \Leftrightarrow Q(\overline{e})$ builds upon the fact that all the terms will denote. Indeed, it lazily grounds the sentence $\forall \overline{x} \in dom_{\overline{e}} : \lceil \overline{e} = \overline{x} \rceil \Rightarrow (P \Leftrightarrow Q(\overline{x}))$, which only propagates when all arguments have a value. To cater for partial functions, we add the sentence $\neg denotes(e) \Rightarrow \neg P$ for each $e \in \overline{e}$, to indicate that $P$ has to be false if some arguments has no value. Similarly for sentences $P \Leftrightarrow f(\overline{e}) \sim e'$.

**Definitions.** To apply UNFOUNDED′ and WELLFOUNDED′, constants are replaced by their interpretation. Whenever a constant in the body is non-denoting, its corresponding atom is replaced by **f**. If a constant in the head is non-denoting, the rule is removed from $\Delta_{inst}$.

**Pre-interpretation.** Considering the interpretation of $c$ in $\mathcal{I}_{in}$, the established rules are in fact sufficient, as they use atoms over $E_{c=}$, of which we already adapted the definition to handle $denotes(c)$ . However, if $\mathcal{I}_{in}$ interprets $c$ as non-denoting, we add the sentence $\neg denotes(c)$ to $\mathcal{T}_s$ instead of adding all possible inequalities $c \neq d$, to reduce the size of the grounding.

**Optimization Term.** In any model, an optimization term $c$ is required to have a value. Hence, we add $denotes(c)$ as sentence to $\mathcal{T}_s$.

### 5.2.10 Complete Search Algorithm

Next to the set of transition rules, a search algorithm consists of an execution order $\ll$ on those rules. The execution order can have a great impact on the efficiency of the search. E.g., whenever FAIL is possible, it is useless to propagate further; PROPAGATE$_{UP}$ is preferred over PROPAGATE$_{ufs}$ because it is cheaper and often derives more propagation; etc.

An important concern when lazily constructing the propositional theory is to prevent the same expression from being generated multiple times, preferably without having to explicitly keep track of this. This can be solved by ordering the transition rules in such a way that propagation is checked over sentences of type $x$ before transition rules that might generate such $x$-sentences. As most of the rules only generate sentences when they would propagate, they would only regenerate sentences whenever they have been forgotten. For rules which generate sentences earlier, such as ENCODE$_{\text{GENERAL}}$ and COMPLETION′, some additional state needs to be maintained.

The result is the following order, where preprocessing rules are not included (such as $\textsc{Encode}_{\textbf{max}}$).

$$\begin{aligned}&\textsc{Fail} \ll \textsc{Learn}_{UP} \ll \textsc{Propagate}_{UP} \ll \textsc{check-}\mathcal{I}_{in}\\ &\quad\ll \textsc{addatom} \ll \textsc{Encode} \ll \textsc{Comparison} \ll \textsc{Completion}'\\ &\quad\ll \textsc{Encode}_{\textbf{sum}} \ll \textsc{Encode}_{\textbf{general}} \ll \textsc{Unfounded}'\\ &\quad\ll \textsc{Decide} \ll \textsc{Wellfounded}' \ll \textsc{Defined-false} \ll \textsc{extract-model}\end{aligned}$$

**Theorem 5.2.13** (Soundness and completeness)**.** *For any ECNF theory $\mathcal{T}$ and consistent interpretation $\mathcal{I}_{in}$ over $\Sigma$ $(\mathcal{T})$, the algorithm terminates and returns an interpretation $\mathcal{I}$, consistent with $\mathcal{I}_{in}$, such that all two-valued extensions of $\mathcal{I} \cup \mathcal{I}_{in}$ are models of $\mathcal{T}$, or* FAIL *if no models of $\mathcal{T}$ exist that extend $\mathcal{I}_{in}$.*

By default, MINISAT(ID) finds models that correspond to the well-founded semantics. In addition, it supports both stable semantics (so is a true ASP solver) and completion semantics. For stable semantics the WELLFOUNDED′ rule is dropped, for the completion semantics, also the UNFOUNDED′ rule is dropped.

### 5.2.11 Optimization and Finding Multiple Models

To solve optimization tasks with an optimization domain term $c$, we extend the presented algorithm with the additional rules MINIMIZE and MAXIMIZE. The relevant rule (depending on whether we are minimizing or maximizing) is applied whenever a model has been found, and adds a constraint that expresses that better models should be found:

$$\begin{array}{llll}\textbf{minimize}: & & & \\ \quad \mathcal{M}\mathcal{T}_s & \longrightarrow \mathcal{T}_s :: (c < c^{\mathcal{M}}) & \textbf{if} & \mathcal{M} \text{ is a model of } \mathcal{T}_s\\ \textbf{maximize}: & & & \\ \quad \mathcal{M}\mathcal{T}_s & \longrightarrow \mathcal{T}_s :: (c > c^{\mathcal{M}}) & \textbf{if} & \mathcal{M} \text{ is a model of } \mathcal{T}_s\end{array}$$

The result is a branch-and-bounds approach. If we are only interested in the optimal solution, we can order these rules before EXTRACT-MODEL. If search terminates and a model has been found, it has been proven to be the optimal one. However, users are frequently interested in the sequence of models, with decreasing cost, or want an anytime algorithm in which the best solution found to date is returned. The sequence can be obtained by ordering the rules after EXTRACT-MODEL, so models are stored first.

To find multiple models, we add an "invalidation" clause whenever a model has been found, a clause that ensures that the obtained model cannot be found again. An obvious approach is to add the negation of the model (as the conjunction of all its true literals) to the theory. More efficient approaches are possible. First, consider the case where the output vocabulary $\Sigma_{out}$ is the full vocabulary. In that case, it is sufficient to add the negation of the decision literals. Indeed, these are the only degrees of freedom of the model that was found, their negation prevents finding the same model again. This approach is also used in MINISAT(ID), by the rule INVALIDATE, which adds the clause $\neg l_1 \vee \ldots \neg l_n$, with $\{l_1, \ldots, l_n\}$ the set of all decision literals in $\mathcal{I}$. More efficient approaches have been studied in the context of ASP [Gebser et al., 2009b]; they are specifically more efficient in case one is looking for a large number of models. Second, in case $\Sigma_{out}$ is a subset of the full vocabulary, we start from the negation of all (assigned) $\Sigma_{out}$-literals. We might further reduce this set by applying resolution using clauses that propagated the negation of literals in the (current) invalidation clause. In the limit, we obtain a subset of the set of decision literals.

INVALIDATE immediately adds the invalidation clause to the theory, which raises a conflict and, hence, leads to learning and backtracking. This approach is preferred over applying a restart before adding the clause to the theory, which often reduces performance for optimization problems.

### After Finding Unsatisfiable

There are some related tasks that can be solved efficiently if search is able to continue after **unsatisfiable** has been found.

1. Finding multiple optimal models: it has to be proven that $c^{\mathcal{M}}$ was minimal by obtaining **unsatisfiable** when $\mathcal{T}_s$ is extended with $c < c^{\mathcal{M}}$.
2. Optimization by deriving both a minimum and a maximum bound for $c$. It is shown that this can result in significantly better results in some application domains (see, e.g., `www.csi.ucd.ie/staff/jpms/soft/soft.php` and [Andres et al., 2012]). This can happen for example when the minimum bound is much closer to the final optimum than the maximum bound.
3. *Multi-criteria* optimization: the input consists not of a single optimization term but an ordered sequence $\{c_{1,o_1}, \ldots, c_{n,o_n}\}$ of optimization terms, with $o_i$ either "minimize" or "maximize". The task is to find a model that is optimal for $c_{1,o_1}$, optimal for $c_{2,o_2}$ among the models optimal for $c_{1,o_1}$, etc.

In each of the cases, theory $\mathcal{T}_s$ has been extended with a sentence $S$ that we want to retract again afterwards. For case 1 for example, after finding unsatisfiable if

$\mathcal{T}_s$ contains $c < c^{\mathcal{M}}$, we want to remove $c < c^{\mathcal{M}}$ from $\mathcal{T}_s$, as we know now that the previous value $c^{\mathcal{M}'}$ was optimal. If we remove $c < c^{\mathcal{M}}$ from $\mathcal{T}_s$, we would like the resulting theory to imply that $c = c^{\mathcal{M}'}$. However, during search $\mathcal{T}_s$ might have been extended or simplified based on the assumption that $c < c^{\mathcal{M}}$ is true.

The restore the theory to a previous state, two additional transition rules are added, namely SAVE and RESET. The idea is that whenever SAVE is called, the current theory $\mathcal{T}_s$ is saved, such that RESET returns the theory to the state when SAVE was last called.[10] For optimization, SAVE is then called just before a new assumption is added to the theory. Afterwards, if **unsatisfiable** was found, RESET is called, after which search can continue or another, more relaxed, assumption is added to the theory.

## 5.3 Experiments

In this section, we provide an experimental evaluation of the ideas and algorithms presented in this chapter and the previous one. The search algorithm for ground FO$(\cdot)^{\text{IDP}}$, presented in this chapter, is implemented as the independent solver MINISAT(ID). The optimization inference for full FO$(\cdot)^{\text{IDP}}$ is implemented as an inference engine in IDP. At its core, MINISAT(ID) was built from the SAT-solver MiniSAT [Eén and Sörensson, 2003]. We reuse its implementation of VSIDS, unit propagation and conflict analysis almost unchanged.

The presented algorithms have not been fully implemented yet. More specifically, only functions with an integer codomain are supported, maximization is not directly supported (but can often be cast as minimization) and function terms in definitions still have to be unnested and replaced by their graph by the grounder (unless they occur in atoms over open symbols). The former was never an issue in the considered benchmarks, support for the latter could further improve the experimental results presented in Chapter 6 (as discussed there).

Next to ground FO$(\cdot)^{\text{IDP}}$, MINISAT(ID) supports several other standardized input languages for search problems, namely

- Conjunctive Normal Form (CNF), the language used by SAT-solvers,
- ground LParse [Syrjänen, 1998], the standard ground ASP dialect, and

---

[10] In MINISAT(ID), SAVE and RESET are optimized to only (re-) storing what changed in $\mathcal{T}_s$.

- FlatZinc [Nethercote et al., 2007], the ground fragment of the CP language (Mini)-Zinc.

We evaluate our optimization engine as follows. In Section 5.3.1, we evaluate the effect of supporting function symbols in the ground format, in terms of performance and grounding size. Second, in Section 5.3.2, we compare IDP and MINISAT(ID) with other systems from the fields of CP and ASP.

### 5.3.1 Ground $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$ Versus Propositional $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$

First, we evaluate the effect of supporting function symbols in the ground format, in terms of performance and grounding size. As benchmarks, we used the benchmarks and instances of the 2013 ASP competition [Alviano et al., 2013] in the NP complexity class.[11] These were complemented by the classic CP benchmarks of `Disjunctive Scheduling` and `Packing` (specification and instances were taken from the 2011 ASP competition [Calimeri et al., 2011]) and the `Concrete Deliv.` problem described in [Asbach et al., 2009], which our group studied in some detail in collaboration with the CODes group at Kulak.

Two different setups were used. The `ground` setup applies MX with SuppF the set of all functions with integer codomains. The `ground` setup, using the algorithm described in this chapter, is compared to the (function-free) reference setup `prop`, which uses SuppF $= \emptyset$. As discussed earlier, the latter results in an effectively propositional grounding, in which the search collapses to the original MINISAT(ID) algorithm presented in [Mariën et al., 2008]. For each benchmark, we randomly selected up-to 30 instances (if that many were available).

In Table 5.1, we report on the performance per benchmark, measured as the number of solved instances, the average total time for the *solved* instances, and the average size of the grounding; bold numbers indicate clear winners. The time limit was 1000 seconds, the memory limit 5 GB, — indicates no results were available.[12]

As Table 5.1 shows, the techniques described in this chapter are crucial to solve some problems, such as `Graceful Graphs` and `Concrete Delivery`, `Disjunctive Scheduling`, `Packing` and `Incremental Scheduling`. The opposite does not hold: there are no problems where `prop` could solve significantly more instances than `ground`. Looking at the problems where the same number of instances

---

[11] Available at `www.mat.unical.it/aspcomp2013/OfficialProblemSuite`.

[12] Experiments were run with IDP-3.1.3 on a 64-bit Ubuntu 12.04 system with an Intel Core i5 3570 processor and 8 GB of RAM.

| Benchmark | # inst. | # solved | avg. time | avg. size |
|---|---|---|---|---|
| Bottle Filling | 30 | 30(30) | 99(98) | $9 \times 10^5(1 \times 10^6)$ |
| Graceful Graphs | 30 | **19**(3) | **131**(489) | $\mathbf{8 \times 10^5}(3 \times 10^7)$ |
| Incremental Sched. | 30 | **20**(0) | 3(—) | $\mathbf{5 \times 10^3}$(—) |
| NoMystery | 30 | 27(28) | 52(64) | $2 \times 10^5(3 \times 10^5)$ |
| Perm. P. Matching | 30 | **30**(22) | **3**(44) | $\mathbf{5 \times 10^4}(1 \times 10^7)$ |
| Ricochet Robots | 30 | 15(15) | 402(408) | $2 \times 10^7(2 \times 10^7)$ |
| Sokoban | 30 | 17(17) | 115(113) | $5 \times 10^5(5 \times 10^5)$ |
| Solitaire | 27 | 23(22) | 10(9) | $3 \times 10^4(3 \times 10^4)$ |
| Stable Marriage | 30 | 30(30) | 124(123) | $3 \times 10^7(3 \times 10^7)$ |
| Weighted Sequence | 30 | 30(30) | 3(12) | $\mathbf{2 \times 10^3}(5 \times 10^5)$ |
| Disjunctive Sched. | 21 | **21**(5) | **2**(27) | $\mathbf{3 \times 10^3}(1 \times 10^7)$ |
| Packing | 30 | **30**(9) | **1**(138) | $\mathbf{1 \times 10^4}(1 \times 10^7)$ |
| Crossing Minim. * | 30 | 9(11) | **48**(128) | $\mathbf{8 \times 10^3}(4 \times 10^5)$ |
| Still-Life * | 26 | 3(4) | **1**(41) | $\mathbf{1 \times 10^4}(5 \times 10^4)$ |
| Valves Location * | 30 | **6**(2) | 156(**49**) | $\mathbf{2 \times 10^6}(6 \times 10^6)$ |
| Concrete Deliv. * | 30 | **18**(0) | **171**(—) | $\mathbf{7 \times 10^5}$(—) |

Table 5.1: Experimental results for the setups ground (function terms allowed in the grounding) and prop (pseudo-propositional grounding). The table shows the number of instances, the number of solved instances, the average time take (in seconds) and the average grounding size (in number of atoms), formatted as ground (prop). Clear winners are shown in bold. For optimization problems (annotated by *), the results reflect instances solved to optimality.

were solved, in all benchmarks the average solving time is similar, and often significantly better, for ground. The average grounding size is in line with the above results: problems with a much smaller grounding are typically solved much faster, and vice versa. Note that there are no benchmarks where using ground leads to a larger average grounding size compared to prop.

For the optimization problems, we also compared the best solutions found within the time limit in case both approaches did not prove optimality (not shown). For Connected, Maximum Density Still-Life, the best solution was found by prop in 18 cases opposed to only once by ground, and similarly for Crossing Minimization (14 to 4). The situation is reversed for Valves Location (3 to 27) and Concrete Delivery (0 to 26), where ground clearly outperforms prop. It is not yet clear if there is a specific reason why either setup is better at solving a specific optimization task.

As $\textsc{Encode}_{\textsc{General}}$ detects propagation quite late, function term occurrences as arguments of non-built-in symbols are unnested by default in IDP and also in the `ground` setup. We also experimented with a setup where no such unnesting was done. The results (not shown in detail), show that for most benchmarks, there was no difference. However, on the `Perm. P. Matching` and `Crossing Minim.` benchmarks, a slowdown was observed: (mostly) the same instances got solved as for `ground`, but the average solving time was significantly higher. Hence, the performance lies between that of `prop` and `ground`, and, as expected, ENCODE$_{\text{GENERAL}}$ only provides an advantage in cases where the grounding becomes too large if generalized atoms are not allowed.

### 5.3.2 Comparison With Other Systems

A second set of experimental results demonstrate where IDP stands in relation to similar systems in the ASP and CP community. For the comparison with ASP, we discuss the results achieved in the 2011 and 2013 ASP competitions. For the comparison with CP, we discuss experiments done by Amadini et al. [Amadini et al., 2013], in the context of developing a MiniZinc portfolio solver.

The ASP competitions typically feature two tracks: a System and a Model-and-Solve track. In the Model-and-Solve track, each participating team submits not only their system but also their own encoding for each of the benchmark problems. In the System track, the encodings are fixed (in a prescribed ASP language) and a submission consists only of an ASP system. The Model-and-Solve track allows teams to exploit all features of their system (also the input language). However, this also means that results depend on which benchmark properties a team was aware of and exploited, such as functional dependencies or symmetries. It is part of future work to support the standard ASP-Core-2 language in IDP to participate in future System tracks.

**MiniSAT(ID) as an ASP Solver.** To evaluate MINISAT(ID) as an ASP solver, the solver was submitted to the 2011 ASP competition System track as an alternative backend to the award-winning ASP-grounder GRINGO.[13] Hence, MINISAT(ID) could be directly compared with the ASP-solver CLASP, the standard Gringo backend and the winner of the competition.

Table 5.2, compiled from the ASP 2011 competition website `www.mat.unical.it/aspcomp2011/SystemTrackFinalResults`, shows the results achieved by the participating ASP systems in the P and NP categories of the System Track (only claspd and cmodels supported the beyond-NP category). The results show that

---

[13] GRINGO produces a ground ASP program in Lparse syntax.

| System | Total Score | Instance Score | Time Score |
|---|---|---|---|
| claspfolio | 818 | 535 | 283 |
| clasp | 810 | 520 | 290 |
| minisatid | 781 | 500 | 281 |
| claspd | 758 | 500 | 258 |
| cmodels | 694 | 465 | 229 |
| lp2diffz3 | 572 | 405 | 167 |
| sup | 541 | 380 | 161 |
| lp2sat2gminisat | 495 | 365 | 130 |
| lp2sat2minisat | 481 | 355 | 126 |
| lp2sat2lminisat | 472 | 350 | 122 |
| smodels | 449 | 295 | 154 |

Table 5.2: Results of the System Track of the 2011 ASP competition.

MINISAT(ID)'s performance lies very close to that of CLASP, with only CLASP and CLASPFOLIO (a portfolio approach that tunes CLASP's options) solving more instances. If we only look at the problems in the NP class, MINISAT(ID) and CLASP performed effectively eye-to-eye, even receiving identical scores.

**MiniSAT(ID) as a MiniZinc Solver.** In the context of developing a MiniZinc portfolio system, Amadini et al. [Amadini et al., 2013] compared 12 different MiniZinc solvers on a dataset of 4642 Constraint Satisfaction Problems.[14] MiniZinc specifications can contain heuristic information that solvers can exploit to improve search; however, this information is ignored by MINISAT(ID), which always applies its domain-independent heuristic. In the case of MINISAT(ID) and several other solvers, the tool MZN2FZN was run as a preprocessor to reduce MiniZinc specifications to FlatZinc. Version 3.9.3 of MINISAT(ID) was used.

The results are shown in Table 5.3.[15] For each solver, the table presents the Average Solving Time (AST) and the Percentage of Solved Instances (PSI). The table allows us to conclude that MINISAT(ID) is the best performing MiniZinc-system of those compared, with a smaller average solving time than any other system and solving 10% more benchmarks than the runner-up (g12cpx).

An interesting observation is that MINISAT(ID) does not support any global constraints (alldifferent, circuit, …) as opposed to several other solvers. Such constraints allow smaller encodings and more efficient propagation algorithms.

---

[14] Recall, CSPs are decision problems. Also, MiniZinc does not contain inductive definition constructs at the time of writing.

[15] Courtesy of Roberto Amadini and colleagues.

| Solver | AST (sec.) | PSI (%) |
|---|---|---|
| minisatid | 950.91 | 51.62 |
| g12cpx | 1126.98 | 41.68 |
| fzn2smt | 1143.47 | 38.13 |
| ortools | 1316.25 | 30.65 |
| g12lazyfd | 1306.10 | 30.31 |
| gecode | 1354.65 | 29.51 |
| izplus | 1350.42 | 28.05 |
| bprolog | 1423.45 | 24.73 |
| jacop | 1435.123 | 24.67 |
| g12fd | 1424.80 | 23.57 |
| mistral | 1525.83 | 16.91 |
| g12mip | 1597.54 | 12.58 |

Table 5.3: Experimental evaluation of MiniZinc solvers on the CSPs in Benchmark Set B [Amadini et al., 2013].

Currently, Amadini et al. are in the process of repeating the experiment for a benchmark set of Constraint Optimization Problems (COPs). According to the authors, preliminary results indicate that MINISAT(ID) performs similarly well on COPs, but definitive results are not yet available at the time of writing.

**IDP as an ASP System.** To compare IDP to other ASP systems, we can look to the results of the previous ASP competitions [Alviano et al., 2013, Calimeri et al., 2011, Denecker et al., 2009], where we participated with IDP in the Model-and-Solve Track. In 2009 and 2011, we used the model expander $IDP^2$, in 2013 we participated with $IDP^3$.

In all three competitions, we aimed to evaluate to what extent automated reasoning systems could handle natural modelings of problems. Hence, we submitted specifications which stayed close to a natural representation of the benchmark problem, instead of relying on non-obvious problem properties, such as non-trivial symmetries or specifically handling subclasses of easily solvable instances.

In all three competitions, IDP achieved fourth place (out of 16 (2009), 6 (2011) and 7 (2013) participants). The winner of all competitions was the GRINGO-CLASP system of the Potsdam ASP group. For many benchmarks, we were able to solve quite a number of instances, which shows that indeed, IDP is indeed able to solve problems using a natural specification. However, as expected, we were often outperformed by more optimized encodings and non-default search heuristics. Some examples: the IDP theory of the

`Reverse Folding` benchmark was less than 50 lines, while EZCSP's ASP encoding is over 300 lines long (and indeed performs significantly better). For `Crossing Minimization`, 10 lines of FO(·)$^{IDP}$ against 50 for GRINGO-CLASP, for `Connected, Maximum Density Still-Life`, 50 against 100.

In the 2013 competition, IDP was disqualified for some benchmarks because of specification bugs that were detected too late in the competition. After fixing the specifications, we reran the experiments with IDP-3.1.1 and with the GRINGO-CLASP submission to the competition. The results are shown in Table 5.4, where the number of solved instances (out of ten instances) is shown for each benchmark. For optimization problems (*), the number of instances for which optimality was proven is counted. We also reran the four benchmarks of the System Track to which we participated (annotated by $_{core}$).

The results show that GRINGO-CLASP solved more instances than IDP (122 instances against 113) and often required less time to solve an instance (not shown). A striking example is `Crossing Minimization`, where GRINGO-CLASP found the optimal solution and proved it was optimal in 9 out of 10 instances, against 0 for IDP. It turns out their encoding contains some sophisticated symmetry breaking which performs quite well. However, IDP solved more instances in 6 out of 17 benchmarks.

These results allow us to conclude that IDP is a state-of-the-art ASP system, but that an optimized encoding can still make a large difference.

## 5.4 Related Work

The work presented in this chapter fits in a more general effort to combine techniques from SAT and CP, ASP and KR to improve search by combining automatic heuristics with richer input language that preserve more problem structure.

In the context of CASP [Lierler, 2012], several systems ground to ASP extended with specific constraint atoms, such as Clingcon [Ostrowski and Schaub, 2012] and EZ(CSP) [Balduccini, 2011]. For search, Clingcon tightly integrates the ASP solver Clasp [Gebser et al., 2012b] with Gecode [Gecode Team, 2013], a well-known CSP solver. EZ(CSP) combines an ASP solver with an CLP-Prolog system and provides different integration schemes [Balduccini and Lierler, 2013] (from treating both solvers as black boxes to a Clingcon-like approach) and supports partial, non-Herbrand functions [Balduccini, 2013]. The CASP solver Inca [Drescher and Walsh, 2011a] searches for answer sets of a ground CASP program by applying LCG for arithmetic and all-different constraints.

| Benchmark | # solved IDP | # solved Gringo-Clasp |
|---|---|---|
| Perm. P. Matching | 10 | 10 |
| Valves Location * | **7** | 4 |
| Still-Life * | 2 | **3** |
| Graceful Graphs | 3 | **9** |
| Bottle Filling | 10 | 10 |
| NoMystery | **9** | 6 |
| Sokoban | **7** | 5 |
| Ricochet Robots | 7 | **10** |
| Crossing Minim. * | 0 | **9** |
| Solitaire | 8 | **9** |
| Weighted Sequence | 10 | 10 |
| Stable Marriage | 10 | 10 |
| Incremental Sched. | **6** | 5 |
| Visit All $_{core}$ | 6 | **7** |
| Knight's Tour $_{core}$ | **1** | 0 |
| Maximal Clique $^{*}_{core}$ | 0 | **1** |
| Graph Colouring $_{core}$ | **7** | 4 |

Table 5.4: Experimental results for benchmarks of the 2013 ASP competition. For optimization problems (*), # solved reflects the number of instances for which optimality was proven. Winners are shown in bold.

More recently, the Chuffed solver was extended to support inductive definitions (using stable model semantics) in combination with uninterpreted functions [Aziz et al., 2013].

Instead of extending the search algorithm, a different approach is to reduce the input further, referred to as *compilation*, and apply an off-the-shelf solver for the reduced language. Examples are reducing CSP to SAT [Tamura et al., 2009, Metodi and Codish, 2012], reducing ASP to MIP [Liu et al., 2012], ASP to SMT [Janhunen et al., 2009] or ASP to SAT [Janhunen, 2004] and reducing CASP to ASP [Drescher and Walsh, 2011b].

SMT systems typically support general uninterpreted functions during search. For this, they use congruence closure algorithms [Nelson and Oppen, 1980, Nieuwenhuis and Oliveras, 2007], which reason symbolically on nested function application and are generally able to find inconsistencies earlier than the approach we presented here. These algorithms are typically not well suited for the kind of applications considered in ASP and CP, where the domains are large and non-trivial. On the other hand, they are very powerful tools in verification, where a common aim is to prove that no models exist (which represent counterexamples or bugs). Work is ongoing to integrate congruence closure in MINISAT(ID).

## 5.5 Conclusion

We presented the MINISAT(ID) search algorithm for the ground fragment of FO(·)$^{\text{IDP}}$. To the best of our knowledge, it is the first algorithm for the full ground fragment of FO(·)$^{\text{IDP}}$ (allowing for example nested function occurrences, inductive definitions and partial functions). The implementation, although it does not yet fully implement the presented algorithm, is currently one of the best free-search MiniZinc solvers and is on-par (although less feature-rich) than the award-winning ASP solver Clasp. It is also one of the first open-source implementations of LCG.

MINISAT(ID) is designed to be an extensible search framework (which was not elaborated upon in this text) that allows developers to easily extend the input language and the transition rules. At this moment, next to what was presented in this chapter, the solver supports dynamic symmetry breaking [Devriendt et al., 2012], interfaces to integrate it with a grounder (used in Chapter 7), supports a variety of input and output languages, and has preliminary support for Quantified Boolean Formulas (second-order quantification over propositional symbols [Franco and Martin, 2009]) through recursive calls of the search algorithm.

Experimental results of the whole optimization engine show that the grounding size can be significantly reduced while obtaining similar or improved search performance. Although a detailed comparison with other systems is nontrivial, we can conclude from the ASP competition that IDP performs quite well in comparison to other ASP systems. Specifically for more natural encodings, the various analysis tools and automatic transformations in IDP turn out to be an important advantage. It is part of future work to implement an ASP-Core-2 parser; this will allow us to compare IDP with other ASP systems on truly the same input. Another part of future work is to implement the rules COMPLETION′, UNFOUNDED′ and WELLFOUNDED′, which will allow us to natively handle function terms in definitions.

# 6

# Deduction Inference for Exploitation of Functional Dependencies

In Chapter 4, it was shown that using function symbols has important advantages from the model expansion perspective. Supporting terms over function symbols natively results in smaller groundings (as no new variables have to be introduced) and ideas from Constraint Programming can be used to obtain better propagation. However, the previous discussion was from the point of view of efficiency of inference engines. To obtain the benefits of a more compact grounding and a better performance, the user has to use these constructs in his declarative specifications. While function symbols can make specifications more concise and readable, a modeler is still free to use predicate symbols and there are various reasons why a modeler might do just that. For example, she might not be aware of the better expected performance or might find predicate symbols easier to understand or use. Also, it may happen that a functional dependency only holds for a particular problem instance (e.g., a graph where each vertex has exactly one outgoing edge). Lastly, it might be that a specification is derived from a language that does not support function symbols, such as standard ASP.

In this chapter, we explore to what extent this burden can be removed from

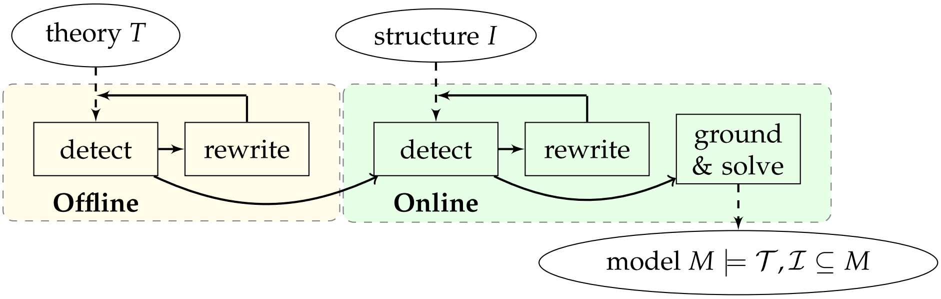


Figure 6.1: Workflow. In a first "offline" phase, the theory is used to detect functional dependencies and functions are introduced until no more can be found (or a time-out is reached). This is repeated in the "online" phase, now combined with the input structure. The transformed theory is then passed to the ground-and-solve algorithm.

the modeler. Namely, we are interested in whether we can still obtain the improved performance shown in previous chapters, without having the explicit information on which relations are functional. We uncover functional dependencies in declarative problem specifications using deduction and exploit them with a transformation that introduces functions and, in the process, eliminates quantified variables and splits symbols into multiple symbols with smaller arity. This results in a more compact grounding and more efficient search. Note that the problem of splitting symbols in an automated way in symbols with lower arity is a widely recognized, unsolved problem, within various fields of A.I., as noted for example within the ASP community by Gebser et al. [Gebser et al., 2011a]. The results presented in this chapter are a partial solution to this problem.

We do this in the context of $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$, with the assumption that all definitions are total. The techniques are implemented as part of the IDP system. The same ideas could be applied in the context of (Constraint) ASP languages. The analysis can be performed on theories both with and without input structure, giving rise to the workflow of Figure 6.1.

**Example 6.0.1.** Consider a scheduling application involving some events (*events*) to be planned, each exactly once, over a large period of time (*time*). A total order $<$ on events is given. One possible constraint is that the planning of events has to follow their order. In $\mathrm{FO}(\cdot)$, this can be represented as the theory consisting of the following sentences, with a grounding size of

$\|event\|^2 \times \|time\|^2$ (typically measured in number of ground atoms):

$$\forall e : \exists_{=1} t : planned(e, t),$$
$$\forall e_1\ e_2\ t_1\ t_2 : e_1 < e_2 \land planned(e_1, t_1) \land planned(e_2, t_2) \Rightarrow t_1 < t_2.$$

However, if we can prove that the second argument of *planned* depends functionally on the first, then *planned* can be replaced by a function symbol, say $f_{planned} : event \mapsto time$. By equivalence preserving transformations, a theory with a grounding size of only $\|event\|^2$ can then be obtained, namely $\forall e_1\ e_2 : e_1 < e_2 \Rightarrow f_{planned}(e_1) < f_{planned}(e_2)$.

The grounding contains constraint atoms $f_{planned}(e_1) < f_{planned}(e_2)$. In CASP Clingcon syntax, the constraint atom is written as $f_{planned}(E1)\$ < f_{planned}(E2)$.

The task is an example of the more general problem of detecting useful properties that are implicit in a specification. Different variations of this problem have already been studied in the literature. The problem has two important features that makes it difficult to solve, namely that (i) the space of possible properties is usually very large and (ii) proving whether a property holds can be hard. Because of this, very diverse solutions have been studied in practice. In the field of CP, Mears et al. [Mears et al., 2008] search for symmetries in a problem specification by generating multiple solutions for several instances of the same benchmark, from which symmetry candidates are extracted, which can then be verified using, e.g., theorem provers. In the field of constraint-based data mining, Guns et al. [Guns et al., 2013] use a unification-based approach in Prolog to syntactically map a constraint specification on constraint specifications for which efficient data mining algorithms exist to solve them. The approximation step, discussed in Chapter 4, is another example, where implied sentences are derived through a fixpoint procedure that reasons over unit propagation schemes for the parse tree of the given theory. Last, the conflict-driven clause learning approach fits in this scheme, as it derives relevant resolvents of clauses in the input specification on the fly.

The main results in this chapter are published in [De Cat and Bruynooghe, 2013].

The chapter is organized as follows. In Section 6.1, we present the detection algorithm, in Section 6.2 how deduction is handled by IDP and, in Section 6.3, how to exploit detected dependencies through theory transformations. In Section 6.4, experimental results are presented; conclusions are presented in Section 6.5.

## 6.1 Detecting Functional Dependencies

Consider again the packing problem, introduced in Chapter 3. A predicate-based FO($\cdot$)$^{\mathrm{IDP}}$-specification of the problem might consist of the theory shown below, over the vocabulary with types *id* and *nb* and additional predicate symbols $pos[id, nb, nb]$, $size[id, nb]$, $area[nb, nb]$, $leftOf[id, id]$, $below[id, id]$ and $noOverlap[id, id]$.

$$
\begin{array}{ll}
\forall id : \exists_{=1}(x\ y) : pos(id, x, y). & (1)\\
\forall id_1\ id_2 : id_1 \neq id_2 \Rightarrow noOverlap(id_1, id_2). & (2)\\
\forall id\ x\ y\ a\ b : (pos(id, x, y) \land area(a, b) \land size(id, s)) & \\
\qquad\qquad \Rightarrow (x \geq 0 \land y \geq 0 \land x + s \leq a \land y + s \leq b). & (3)
\end{array}
$$

$$
\left\{
\begin{array}{lll}
\forall id_1\ id_2 : leftOf(id_1, id_2) & \leftarrow \exists x_1\ y_1\ x_2\ y_2\ s_1 : pos(id_1, x_1, y_1) & \\
& \quad \land size(id_1, s_1) \land pos(id_2, x_2, y_2) \land x_1 + s_1 \leq x_2 & (4)\\
\forall id_1\ id_2 : below(id_1, id_2) & \leftarrow \exists x_1\ y_1\ x_2\ y_2\ s_1 : pos(id_1, x_1, y_1) & \\
& \quad \land size(id_1, s_1) \land pos(id_2, x_2, y_2) \land y_1 + s_1 \leq y_2 & (5)\\
\forall id_1\ id_2 : noOverlap(id_1, id_2) & \leftarrow leftOf(id_1, id_2) \lor leftOf(id_2, id_1) & \\
& \quad \lor below(id_1, id_2) \lor below(id_2, id_1) & (6)
\end{array}
\right\}
$$

The sentences express respectively the constraints (as FO sentences) that (1) each square is placed at exactly one position, (2) no squares overlap and (3) each square fits completely inside the specified area. The defined concepts have the same meaning as in previous chapters.

The grounding of this theory can become very large. For example, rule (4) has a grounding size in the order of $n^2a^2b^2$ instances, with $n$ the number of squares and $a$ and $b$ respectively the length and width of the area. However, *pos*, *size* and *area* in fact represent respectively 2, 1 and 2 functional relationships. Indeed, as shown previously, the body of (4) could be replaced by the formula $pos_x(id_1) + size(id_1) \leq pos_x(id_2)$, with $pos_x$ and *size* are functions derived from the predicates *pos* and *size*. The latter formula has a grounding size in the order of only $n^2$ atoms.

First we introduce some additional notation. An *index set* $S$ is a set of the form $\langle s_1, \ldots, s_m \rangle$ that is a subsequence of $[1, n]$. With $\bar{t} = \langle t_1, \ldots, t_n \rangle$ a tuple and an index set $S$, $\bar{t}|_S$ denotes the tuple $\langle t_{s_1}, \ldots, t_{s_m} \rangle$ while $S^c$ denotes the complement of $S$ with respect to $[1, n]$, i.e., the elements of $S$ are removed. An expression $(t_i \mid i \in S)$ is used as a shorthand for $(t_{s_1}, \ldots, t_{s_m})$. Abusing notation, we denote predicate and function symbols as predicates and functions, respectively.

A predicate has a *functional dependency* from a set of arguments $S$ denoted by their index ($S$ is an index set) to an argument at index $j$ if a value for the arguments at $S$ uniquely determines the value of the argument at $j$. For an $n-1$-ary function, the output can be considered as the $n^{th}$ argument; a function (obviously) always has a dependency from the $n-1$ input arguments to the output argument, but also other dependencies can be present. A functional dependency can be formalized as a mapping from an index set to an argument position.

**Definition 6.1.1** (Functional dependency). Let $P[\tau_1,\ldots,\tau_n]/f[\tau_1,\ldots,\tau_{n-1}\mapsto\tau_n]$ be the signature of a predicate/function, $S$ an index set over $[1,n]$, $j$ an index in $S^c$, and $\mathcal{T}$ a theory in which $P/f$ occurs. We have a *partial functional dependency* from $S$ to $j$ if, in each model $\mathcal{I}$ of $\mathcal{T}$, it holds that for each tuple $\overline{d}_r \in (\tau_1^{\mathcal{I}}\times\ldots\times\tau_n^{\mathcal{I}})|_S$, all tuples $\overline{d}'$ in $P^{\mathcal{I}}/f^{\mathcal{I}}$ for which $\overline{d}'\Big|_S = \overline{d}_r$ have the same value for $d'_j$ (called the *uniqueness* property). We have a *total functional dependency* if, in addition, for each tuple $\overline{d}_r \in (\tau_1^{\mathcal{I}}\times\ldots\times\tau_n^{\mathcal{I}})|_S$, there is a tuple $\overline{d}'$ in $P^{\mathcal{I}}/f^{\mathcal{I}}$ for which $\overline{d}'\Big|_S = \overline{d}_r$ (called the *existence* property).

The *uniqueness* property expresses that a tuple in the index set $S$ maps to at most one value for the $j^{th}$ argument; the *existence* property that there is such a value for each well-typed tuple over $S$. Dependencies as above are denoted by $d\,\langle P[\tau_1,\ldots,\tau_n],S,j\rangle/\ d\,\langle f[\tau_1,\ldots,\tau_{n-1}\mapsto\tau_n],S,j\rangle$, a subscript *total* (*partial*) is added to denote total (partial) dependencies whenever relevant. The index set $S$ and the argument $j$ are called, respectively, the *domain* and the *codomain* of the functional dependency. Note that in the case of functions, the codomain (index $n$) of the function can be part of the domain of a dependency. For example, a bijective function $f[\tau_1\mapsto\tau_2]$ has the dependencies $d\,\langle f[(\tau_1\mapsto\tau_2],\{1\},2\rangle$ and $d\,\langle f[\tau_1\mapsto\tau_2],\{2\},1\rangle$.

**Definition 6.1.2** (Function constraints). For an index set $S$ and index $j$, the existence property states that for a predicate $P[\overline{\tau}]$ (a function $f[\overline{\tau}\mapsto\tau_n]$) in a theory $\mathcal{T}$, $\mathcal{T}$ entails the *existence constraint*, namely, for a predicate, the sentence

$$\forall(x_i \mid i\in S) : \bigwedge_{i\in S}\tau_i(x_i) \Rightarrow \exists(x_i \mid i\in S^c) : \bigwedge_{i\in S^c}\tau_i(x_i)\wedge P(\overline{x})$$

and, for a function, the sentence

$$\forall(x_i \mid i\in S) : \bigwedge_{i\in S}\tau_i(x_i) \Rightarrow \exists(x_i \mid i\in S^c) : \bigwedge_{i\in S^c}\tau_i(x_i)\wedge f(\overline{x}) = x_n.$$

The uniqueness property states that $\mathcal{T}$ entails the *uniqueness constraint*, namely, for a predicate, the sentence

$$\forall\overline{x}\,\overline{x}' : P(\overline{x})\wedge P(\overline{x}')\wedge \overline{x}|_S = \overline{x}'|_S \Rightarrow x_j = x'_j$$

and, for a function, the sentence

$$\forall \overline{x}\, \overline{x}' : f(\overline{x}) = x_n \land f(\overline{x}') = x'_n \land (\overline{x} :: x'_n)\big|_S = (\overline{x}' :: x_n)\big|_S \Rightarrow x_j = x'_j.$$

In what follows, shorthands $C_{exists}(P[\overline{\tau}], S, j)$, respectively, $C_{exists}(f[\overline{\tau} \mapsto \tau_n], S, j)$, $C_{unique}(P[\overline{\tau}], S, j)$ and $C_{unique}(f[\overline{\tau} \mapsto \tau_n], S, j)$, are used for these constraints.

### 6.1.1 Detection Algorithm

Proposition 6.1.2 is the basis for a straightforward detection algorithm that iterates over all predicate and function symbols of a given theory $\mathcal{T}$. For each symbol and each of its possible index sets $S$ and argument positions $j \in S^c$, it applies deduction (see Section 6.2) to check whether the corresponding uniqueness property is entailed by $\mathcal{T}$. If so, we have a partial functional dependency. If, in addition, also the corresponding existence property is entailed, a total functional dependency is detected. Whenever a dependency is detected, the theory can be rewritten to make the dependency explicit (see Section 6.3) and the detection algorithm continues with the new theory, until all possible dependencies have been checked.

Using a theorem prover for checking the entailment of the constraints, the algorithm has two issues. First, checking whether a particular property holds may take an excessive amount of time, especially when the property does not hold. So a time-out is necessary. Second, the number of potential dependencies is exponential in the arity of symbols, so another time-out is needed. This means we have to use an anytime algorithm and have to decide on the order in which we iterate over all candidate dependencies. However, the following proposition allows us to prune the search.

**Proposition 6.1.3** ([Armstrong, 1974]). *For a theory $\mathcal{T}$, a total (partial) functional dependency of a position $j$ on an index set $S$ for a symbol in $\mathcal{T}$ implies a total (partial) functional dependency of $j$ on all index sets of the given symbol that are supersets of $S$ and do not contain $j$.*

This proposition offers two opportunities to prune the search. First, if one can prove $\neg C_{unique}(P[\overline{\tau}], S, j)$ (or $\neg C_{unique}(f[\overline{\tau} \mapsto \tau_n], S, j)$, then there is no dependency of $S$ on $j$ and one need not consider subsets of $S$. Second, if a dependency from $S$ to $j$ is found, one need not consider supersets of $S$. Our current implementation only exploits the latter and starts from the smallest candidate index sets (starting with $\emptyset$, i.e., a constant argument). Each time one is found, the theory is rewritten (see next section) and detection continues on the new theory. However, that way, one likely never analyses the largest index sets (due to time-out), while dependencies involving them are quite frequent for predicates. So, for predicates, before exploring index sets from small to large, we first check for a dependency with an index set of size $n-1$ and store it when found. Then we process index sets from small to large. If the algorithm aborts because of a time-out, the stored dependency, if present and not pruned, is used to rewrite the theory.

The detection algorithm resulting from using detection algorithm can be used both *offline* and *online*. In an offline setting, the theory is optimized (often without time bounds) to improve subsequent uses. If the functional dependency is only present in the instance at hand, it might also be worthwhile to do online detection. Consider for example an instance of a graph problem where each node has exactly one outgoing edge. In that case, the algorithm takes as input a $\Sigma$-theory $\mathcal{T}$ and a $\Sigma$-structure $\mathcal{I}$ and the first step consists of transforming $\mathcal{T}$ into $\mathcal{T}'$ such that a structure $\mathcal{M}$ is a model of $\mathcal{T}'$ iff $\mathcal{M}|_\Sigma$ expand $\mathcal{I}$ and is a model of $\mathcal{T}$; afterwards the detection algorithm is applied to $\mathcal{T}$. Such transformation can be accomplished in a straightforward way by adding the appropriate unique-names axioms, domain-closure axioms and atomic sentences.[1]

## 6.2 Deduction in IDP

As already mentioned, we use deduction to check for functional dependencies. More specifically, we are interested in whether $\textbf{entails}\left\langle \mathcal{T}, \mathcal{T}_f \right\rangle$ is true, with $\mathcal{T}_f$ a theory expressing some function constraint and $voc(\mathcal{T}) \supseteq voc(\mathcal{T}_f)$. In this section, we go into detail in how IDP supports such inference.

As starting point, we observe that off-the-shelf theorem provers are currently available for various standardized logics [Sutcliffe, 2009], with yearly competitions comparing their performance [Sutcliffe, 2013, Sutcliffe, 2012]. For $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$, no theorem provers are yet available, but there are two interesting alternatives, namely the logics FOF and TFF_ARITH. The former is a syntactical variant of FO as introduced in this thesis, the latter is a variant of $\mathrm{FO}(\mathbb{Z})$. Consequently, a straightforward approach for $\textbf{entails}\left\langle \mathcal{T}, \mathcal{T}_f \right\rangle$ is to transform any $\mathcal{T}$ and $\mathcal{T}_f$ not using other language extensions than arithmetic to TFF_ARITH and apply an off-the-shelf prover (or to FOF if also no arithmetic is used).

---

[1] Note that online detection is not possible for infinite structures, as the UNA and DCA cannot be expressed.

But can we do better? In our application at hand, for example, $\mathcal{T}_f$ happens to consist only of FO sentences, but $\mathcal{T}$ contains definitions and aggregates. In general, FO$(\cdot)^{\mathtt{IDP}}$ cannot be translated to FO, but for some extensions this is possible. For the others, note that we are often only interested in positive answers (whether $\mathcal{T}$ indeed entails $\mathcal{T}_f$, as in that case, we can improve the theory). In that case, we are allowed to transform the left-hand side theory ($\mathcal{T}$) into a (weaker) FO($\mathbb{Z}$) theory $\mathcal{T}'$, as long as $\mathcal{T}' \models \mathcal{T}$. In that case, positive answers of $\mathcal{T}' \models \mathcal{T}_f$ imply $\mathcal{T} \models \mathcal{T}_f$, by transitivity of the $\models$ relation.

In the next section, we show how FO$(\cdot)^{\mathtt{IDP}}$ theories can be converted to strong $\Sigma$-equivalent FO($\mathbb{Z}$) theories. The resulting transformation results in highly recursive theories however, which current day theorem provers cannot handle efficiently as they still poorly support proofs by induction (as can, e.g., be observed on the inductive proving track of the theorem proving competition). To that end, in the subsequent section, we present weaker transformation to FO($\mathbb{Z}$) and to FO. The latter transformations are applied in the deduction engine in IDP.

### 6.2.1 Transforming FO($\cdot$) to FO($\mathbb{Z}$)

First, we introduce the *strong* FO($\cdot$)-*to*-FO($\mathbb{Z}$) transformation ("strong" as it preserves $\Sigma$-equivalence), by showing for each language extension how it can be transformed, in the order the transformations are applied.

#### Typed Symbols

For every predicate symbol $P[\tau_1, \ldots, \tau_n]$, type information is made explicit by adding the sentence $\forall x_1 \ldots x_n : P(x_1, \ldots, x_n) \Rightarrow \tau_1(x_1) \land \ldots \land \tau_n(x_n)$. For every function symbol $f[\tau_1, \ldots, type_{n-1} \mapsto \tau_n]$, the sentence $\forall x_1 \ldots x_n : f(x1, \ldots, x_{n-1}) = x_n \Rightarrow \tau_1(x_1) \land \ldots \land \tau_n(x_n)$ is added.

For every type $\tau$ with a supertype $\tau'$, a sentence $\forall x : \tau(x) \Rightarrow type'(x)$ is added.

For a type $\tau$ constructed by a set of function symbols $\{f_1[\overline{\tau}_1 \mapsto \tau], \ldots, f_m[\overline{\tau}_m \mapsto \tau]\}$, the UNA and DCA are added as follows. The UNA, expressing that all images are unique, is formalized as the FO sentences $\forall \overline{x}\ \overline{x}' : \overline{\tau}_i(\overline{x}) \land \overline{\tau}_j(\overline{x}') \land f_i(\overline{x}) = f_j(\overline{x}') \Rightarrow i = j \land \overline{x} = \overline{x}'$, for each $i, j \in [1, m]$ (taking care of the arities and adding type information). The DCA, expressing that no other elements than the constructed ones are part of $\tau$, cannot be expressed in FO. It can however be expressed as the inductive definition

$$\left\{ \begin{array}{l} \forall x : \tau(x) \leftarrow \exists \overline{y} : \overline{\tau}_1(\overline{y}) \wedge f_1(\overline{y}) = x \\ \qquad\qquad \vdots \\ \forall x : \tau(x) \leftarrow \exists \overline{y} : \overline{\tau}_m(\overline{y}) \wedge f_m(\overline{y}) = x \end{array} \right\}$$

### Partial Functions

Partial functions $f[\tau_1, \dots, \tau_{n-1} \mapsto \tau_n]$ in $\Sigma$ are unnested and replaced by their graph $G_f[\tau_1, \dots, \tau_n]$.

Concerning **entails**$\langle \mathcal{T}_1, \mathcal{T}_2 \rangle$, any partial function in $voc(\mathcal{T}_2)$ has to be replaced simultaneously in $\mathcal{T}_1$ and $\mathcal{T}_2$ (by the same predicate symbol). In that case, it is straightforward to see that entailment carries over (even if the vocabulary changes).

### Aggregates

Atoms $\#\{\overline{x} : \varphi\} \geq n$, with $n$ a natural number (cardinality aggregates with a known bound), are replaced by

$$\exists \overline{x}_1 \dots \overline{x}_n : ( \bigwedge_{i,j \in [1,n], i \neq j} \overline{x}_i \neq \overline{x}_j) \bigwedge_{i \in [1,n]} \varphi[\overline{x}/\overline{x}_i].$$

Other comparison operators are rewritten in a similar way.

Other aggregates are transformed in a $\Sigma$-equivalence preserving way as suggested in Section 5.4 of [Pelov, 2004]. Recall that all aggregate functions in $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ are *decomposable*, i.e., aggregates for which $agg(S) = agg(\{agg(S'), agg(S \setminus S')\})$, for sets of domain elements $S$ and $S' \subset S$. Our transformation of $agg(\{\overline{x} \in \overline{\tau} : \varphi : t\})$ is based on the decomposability of the considered aggregates and the fact that there is a total order on all domain elements, with a function $MIN$ (minimum) and a (partial) function $PRED$ (predecessor) and that this order can be extended to a total order over tuples of domain elements. Assuming $neutral_{agg}$ is the neutral element of $agg$, we define an accumulator function $acc[\overline{\tau} \mapsto \mathbb{N}]$ (a well-known Prolog programming pattern) as:

$$\left\{\begin{array}{ll} acc(MIN(\overline{\tau})) = t[\overline{x}/MIN(\overline{\tau})] & \leftarrow \varphi[\overline{x}/MIN(\overline{\tau})] \\ acc(MIN(\overline{\tau})) = neutral_{agg} & \leftarrow \neg\varphi[\overline{x}/MIN(\overline{\tau})] \\ \forall \overline{x} \in \overline{\tau} : acc(\overline{x}) = agg(acc(PRED(\overline{x})), t) & \leftarrow \varphi \land \neg(\overline{x} = MIN(\overline{\tau})) \\ \forall \overline{x} \in \overline{\tau} : acc(\overline{x}) = acc(PRED(\overline{x})) & \leftarrow \neg\varphi \land \neg(\overline{x} = MIN(\overline{\tau})) \end{array}\right\}$$

This definition is equivalent to the FO formulas

$$\begin{array}{l} acc(MIN(\overline{\tau})) = v \equiv (v = t[\overline{x}/MIN(\overline{\tau})] \land \varphi[\overline{x}/MIN(\overline{\tau})]) \lor \\ \qquad\qquad\qquad (v = neutral_{agg} \land \neg\varphi[\overline{x}/MIN(\overline{\tau})]) \text{ and} \\ acc(\overline{x}) = v \equiv \neg(\overline{x} = MIN(\overline{\tau})) \land \\ \qquad\qquad (v = agg(acc(PRED(\overline{x})), t) \land \varphi) \lor (v = acc(PRED(\overline{x})) \land \neg\varphi) \end{array}$$

The aggregate term itself is then replaced by $acc(MAX(\overline{\tau}))$.

The only remaining issue is to add a definition of the minimum and maximum functions for two terms. Indeed, cardinality is reduced to addition, and addition and product are supported in FO($\mathbb{Z}$).

### Inductive Definitions

Definitions $\Delta$ are rewritten using a $voc(\Delta)$-equivalence preserving transformation based on ideas in [Janhunen et al., 2009, Pelov and Ternovska, 2005] as follows. The completion has models that are not well-founded when the definition contains positive loops. To eliminate such models, Janhunen et al. [Janhunen et al., 2009] used the idea of *level mappings* for converting ground rule sets to propositional logic. Pelov et al. [Pelov et al., 2007] elaborated on this in the context of the well-founded semantics. The idea is to introduce a function symbol $l_P[\overline{\tau} \mapsto \mathbb{N}]$ for every defined predicate or function symbol $P[\overline{\tau}]$, the *level* of atoms over $P$ ( we only work out the predicate case in detail). The function $l_P(\overline{d})$ is axiomatized to be 0 if $P(\overline{d})$ is false, and otherwise it states that the level $l_P(\overline{d})$ of atom $P(\overline{d})$ is strictly larger than the level of any atoms that were positively used to derive the truth of $P(\overline{d})$. Interpretations that satisfy these constraints, together with the completion, are then guaranteed to not contain positive or mixed loops.

Recall, without loss of generality, we can assume that each predicate (or function) is defined by a single rule. Under that assumption, it suffices to consider the following cases. For simplicity, assume $l_P[\overline{\tau} \mapsto \mathbb{N}]$ also exists for all non-defined symbols $P[\overline{\tau}]$, mapping to 0 for each atom over $P$.

- $\forall \overline{x} : \neg P(\overline{x}) \Rightarrow l_P(\overline{x}) = 0$ (we assume all well-founded models are total).

- $\forall \overline{x} : P(\overline{x}) \leftarrow \neg Q(\overline{x})$. The constraint is $\forall \overline{x} : P(\overline{x}) \Rightarrow l_P(\overline{x}) \geq 0$.
- $\forall \overline{x} : P(\overline{x}) \leftarrow \forall \overline{y} : Q(\overline{x} :: \overline{y})$. The constraint is $\forall \overline{x} :: \overline{y} : P(\overline{x}) \Rightarrow l_P(\overline{x}) > l_Q(\overline{x} :: \overline{y})$.
- $\forall \overline{x} : P(\overline{x}) \leftarrow \exists \overline{y} : Q(\overline{x} :: \overline{y})$. The constraint is $\forall \overline{x} : P(\overline{x}) \Rightarrow \exists \overline{y} : l_P(\overline{x}) > l_Q(\overline{x} :: \overline{y}) \wedge Q(\overline{x} :: \overline{y})$.
- $\forall \overline{x} : P(\overline{x}) \leftarrow Q_1(\overline{x}) \vee \ldots \vee Q_n(\overline{x})$. The constraint is $\forall \overline{x} : P(\overline{x}) \Rightarrow ((Q_1(\overline{x}) \wedge l_P(\overline{x}) > l_{Q_1}(\overline{x})) \vee \ldots \vee (Q_n(\overline{x}) \wedge l_P(\overline{x}) > l_{Q_n}(\overline{x})))$.
- $\forall \overline{x} : P(\overline{x}) \leftarrow Q_1(\overline{x}) \wedge \ldots \wedge Q_n(\overline{x})$. The constraint is $\forall \overline{x} : P(\overline{x}) \Rightarrow l_P(\overline{x}) > l_{Q_1}(\overline{x}) \wedge \ldots \wedge l_P(\overline{x}) > l_{Q_n}(\overline{x})$.

**Example 6.2.1.** To convert the $\Sigma$-definition

$$\left\{ \begin{array}{l} \forall x \in \tau : P(x) \leftarrow Q(x) \\ \forall x \in \tau : Q(x) \leftarrow P(x) \end{array} \right\}$$

to FO($\mathbb{Z}$), the function symbols $l_P[\tau \mapsto \mathbb{N}]$ and $l_Q[\tau \mapsto \mathbb{N}]$ are introduced. Using these, the definition is transformed into the sentences

$$\begin{array}{l} \forall x \in \tau : \neg P(x) \Rightarrow l_P(x) = 0 \\ \forall x \in \tau : \neg Q(x) \Rightarrow l_Q(x) = 0 \\ \forall x \in \tau : P(x) \Rightarrow l_Q(x) > l_P(x) \\ \forall x \in \tau : Q(x) \Rightarrow l_P(x) > l_Q(x). \end{array}$$

It can be seen easily that both theories have the same, unique $\Sigma$-model, namely one in which both $P$ and $Q$ are completely false.

### Typed Quantifications

Formulas $\forall \overline{x} \in \overline{\tau} : \varphi$ are replaced by $\forall \overline{x} : (\bigwedge_{i \in [1,n]} \tau_i(x_i)) \Rightarrow \varphi$. Typed variables in existential quantifications are transformed in a similar fashion (and sets are no longer present).

## 6.2.2 Weak Transformation

The strong FO($\cdot$)-to-FO($\mathbb{Z}$) transformation preserves $\Sigma$-equivalence and hence also all functional dependencies. However, the transformation of inductive definitions and of aggregates (for which an accumulator function is used), often results in large formulas (with lots of arithmetic) and can hence substantially increase the run-time of the theorem prover. Also, the proof of many properties over them would require reasoning by induction over the natural numbers. Alternatively, a *weak* FO($\cdot$)-*to*-FO($\mathbb{Z}$) transformation can be applied.

Note that, for any FO($\cdot$)$^{\mathrm{IDP}}$-theory $\mathcal{T}$, replacing any formula in positive (negative) context by true (false) or dropping any definition results in a weaker theory. The *weak* FO($\cdot$)-*to*-FO($\mathbb{Z}$) is identical to the strong version except that:

**w1** The rule for $\#\{\overline{x} : \varphi\} \geq n$ is only applied for small $n$ (e.g., $n < 3$),

**w2** Aggregates are unnested and afterwards, atoms containing aggregate terms are replaced by true if they occur in a positive context, by false if in a negative context and by $P(\overline{x})$ otherwise, with $\overline{x}$ the free variables of the atom and $P$ a new predicate.

**w3** A definition is replaced by its completion.

**Proposition 6.2.2.** *Let $\mathcal{T}_s$ be the strong and $\mathcal{T}_w$ be the weak* FO($\cdot$)-*to*-FO($\mathbb{Z}$) *transformation of $\mathcal{T}$. With the understanding that a partial function $f$ in $\mathcal{T}$ corresponds to a predicate $G_f$ in $\mathcal{T}_s$ and $\mathcal{T}_w$, it holds that: (i) a dependency is entailed by $\mathcal{T}$ if it is entailed by $\mathcal{T}_s$ and (ii) a dependency entailed by $\mathcal{T}_w$ or $\mathcal{T}_s$ on symbols in $voc(\mathcal{T})$ is entailed by $\mathcal{T}$.*

*Proof sketch.* (i) holds because the transformation rules, except the rule for partial functions, preserve strong $\Sigma$-equivalence. For partial function, a tuple $\overline{d}$ is part of the interpretation of $f$ in a model $\mathcal{I}$ of $\mathcal{T}$ iff it is part of the corresponding model of $G_f$ in $\mathcal{T}_s$. (ii) holds because rule w1 preserves strong equivalence and rules w2 and w3 make sure that formulas are replaced by weaker formulas, hence models are preserved and only extra models can be created, so no new functional dependencies can be introduced by the transformation. □

Lastly, we also define the *weak* FO($\cdot$)-*to-FO* transformation, to be able to use the larger range of provers for FO. Intuitively, FO($\mathbb{Z}$) extends FO with the natural numbers: a set of domain elements and a set of arithmetic functions and constants with fixed interpretation. Consequently, we can view an FO($\mathbb{Z}$) theory as a (weaker) FO theory by dropping the interpretation of those symbols. The weak FO($\cdot$)-to-FO takes as input a theory $\mathcal{T}$, applies the weak FO($\cdot$)-to-FO($\mathbb{Z}$) and considers the result as an FO theory.

**Example 6.2.3.** Applying the weak FO($\cdot$)-to-FO($\mathbb{Z}$) transformation on the square-packing example replaces constraint (1) (in fact a cardinality constraint) by the sentences $\forall id : id(id) \Rightarrow \exists x\ y : pos(id, x, y)$ and $\forall id_1\ x_1\ y_1\ x_2\ y_2 : pos(id_1, x_1, y_1) \wedge pos(id_1, x_2, y_2) \Rightarrow x_1 = x_2 \wedge y_1 = y_2$ (w1). Rule 1 is applied to all predicates; e.g., for the predicate $pos(id, nb, nb)$, the sentence $\forall id\ x\ y :$

$pos(id, x, y) \Rightarrow id(id) \land nb(x) \land nb(y)$ is added. As for the definition part, rule w3 replaces the three rules by their completion. Equivalence is preserved in this case.

For the resulting theory, the FO prover SPASS [Weidenbach et al., 2009] (the default prover in IDP) can prove, e.g., $C_{unique}(pos[id, x, y], \{1\}, 2)$ and $C_{exists}(pos[id, x, y], \{1\}, 2)$, i.e., that each square has exactly one x-coordinate in all models of the theory.

## 6.3 Rewriting the Theory

As said in the previous section, each time a dependency is detected, the theory is rewritten into an equivalent theory. First, we explain how a theory is rewritten in case a functional dependency is detected for a symbol that is not defined. Next, we extend the method to defined symbols. Finally we describe how the detection and rewriting of functional dependencies is integrated into our model expansion methodology.

### 6.3.1 Rewriting of Non-defined Symbols

A theory can entail multiple functional dependencies on the same symbol and it is not clear what is the best way to exploit all of them. E.g., for a bijection, we have to decide which one to use in the rewriting. For the other dependency, we then have to decide whether to add the function constraints to the theory[2]. Different choices affect grounding size and search behaviour differently. Our heuristic is to use dependencies that result in symbols with lower arities. Hence we do not look for dependencies on functions with $\#(S) = n - 1$.

The rewriting transformation dep-reduce takes a theory $\mathcal{T}$ and a dependency $d$ over a non-defined symbol and produces a new $\mathcal{T}'$ in which $d$ is explicit as follows.

**Definition 6.3.1** (dep-reduce)**.** The rewriting of a functional dependency $d$ of the form $d\left\langle f[\overline{\tau} \mapsto \tau_n], S, j \right\rangle$ over a function symbol starts by a preprocessing phase. Occurrences of a term $f(\overline{t})$ are unnested if their closest parent that is not a function term is an aggregate set or the head of a rule. In addition, if $n \in S$, $f$ is unnested everywhere and replaced by its graph $G_f$; the input dependency is rewritten as $d\left\langle G_f[\overline{\tau} :: \tau_n], S, j \right\rangle$.

---

[2] Redundant constraints can improve search performance.

The main rewriting then distinguishes between a dependency on predicate or function symbols.

- Let $d\,\langle f[\overline{\tau} \mapsto \tau_n], S, j\rangle$ be a (partial) functional dependency ($n \notin S$) with $\#(S) < n-1$. For the occurrences of $f(\overline{t})$, we identify two cases. In both, a new (partial) function $f_d[\overline{\tau}|_S \mapsto \tau_j]$ is introduced.

  $j \neq n$ : A new function $f_r[\overline{\tau}|_{\{j\}^C} \mapsto \tau_n]$ is added; $f_r$ is partial iff $f$ is. Occurrences of $f(\overline{t})$ are eliminated by replacing atoms $a[f(\overline{t})]$ by $a[f(\overline{t})/f_r(\overline{t}|_{\{j\}^C})] \wedge t_j = f_d(\overline{t}|_S)$.

  $j = n$ : A new predicate $P_r[\overline{\tau}]$ is introduced. Occurrences of $f(\overline{t})$ are eliminated by replacing atoms $a[f(\overline{t})]$ by $a[f(\overline{t})/f_d(\overline{t}|_S)] \wedge P_r(\overline{t})$.

- Let $d\,\langle P[\overline{\tau}], S, j\rangle$ be a (partial) functional dependency for a predicate $P$. If $\#(S) = n-1$, a new (partial) function $f_d[\overline{\tau}|_S \mapsto \tau_j]$ is introduced and each atom $P(\overline{t})$ is replaced by the atom $t_j = f_d(\overline{t}|_S)$; otherwise, also a predicate $P_r[\overline{\tau}|_{\{j\}^C}]$ is introduced and atoms $P(\overline{t})$ are replaced by $P_r(\overline{t}|_{\{j\}^C}) \wedge t_j = f_d(\overline{t}|_S)$.

To translate a model of the new theory into the vocabulary of the original theory, we have to define the replaced symbol in terms of the new symbols (in fact, this is only necessary if the symbol is part of the output vocabulary of the problem at hand).

**Definition 6.3.2** (Definition introduction)**.** We distinguish three cases.

- If the preprocessing replaced $f[\overline{\tau} \mapsto \tau_n]$ by $G_f[\overline{\tau} :: \tau_n]$ then add

$$\left\{\forall \overline{x}, x_n \in \overline{\tau}, \tau_n : f(\overline{x}) = x_n \leftarrow G_f(\overline{x} :: x_n)\right\}.$$

- If $P[\overline{\tau}]$ was replaced, then add the definition

$$\begin{array}{ll} \{\forall \overline{x} \in \overline{\tau} : P(\overline{x}) \leftarrow f_d(\overline{x}|_S) = x_j\} & \text{if \#(S)=n-1,} \\ \left\{\forall \overline{x} \in \overline{\tau} : P(\overline{x}) \leftarrow P_r(\overline{x}|_{\{j\}^C}) \wedge f_d(\overline{x}|_S) = x_j\right\} & \text{otherwise.} \end{array}$$

- If $f[\overline{\tau} \mapsto \tau_n]$ was replaced, add

$$\begin{array}{ll} \left\{\forall \overline{x} \in \overline{\tau} : f(\overline{x}) = f_r(\overline{x}|_{\{j\}^C}) \leftarrow f_d(\overline{x}|_S) = x_j\right\} & \text{if } j \neq n \text{ and } n \notin S, \\ \{\forall \overline{x} \in \overline{\tau} : f(\overline{x}) = f_d(\overline{x}|_S) \leftarrow P_r(\overline{x})\} & \text{otherwise } (j = n). \end{array}$$

**Proposition 6.3.3.** *Let $\mathcal{T}$ be a theory and d a functional dependency of $\mathcal{T}$ for a symbol not defined in $\mathcal{T}$. Let $\mathcal{T}'$ be the theory obtained after applying the* dep-reduce *rewriting of Definition 6.3.1 for d and the definition introduction of Definition 6.3.2. Then $\mathcal{T}'$ is strongly $voc(\mathcal{T})$-equivalent with $\mathcal{T}$.*

The proof of Proposition 6.3.3 (in fact, the proof of the more general Proposition 6.3.9 presented below) is provided in Appendix A.2.

Before extending the approach to defined symbols, we provide some examples and complete the rewrite strategy to effectively reduce the number of quantifications.

**Example 6.3.4.** Consider rule (4) of our packing problem; applying dep-reduce for the functional dependency $d_{total}\,\langle pos[id,nb,nb],\{1\},2\rangle$ introduces a function we rename as $pos_x[id \mapsto nb]$ and a relation $pos_r[id,nb]$. This produces the definition

$$\left\{\begin{array}{l} \forall id_1\ id_2 : leftOf(id_1,id_2) \leftarrow \exists x_1\ y_1\ x_2\ y_2\ s_1 : \\ \qquad\qquad pos_r(id_1,y_1) \land x_1 = pos_x(id_1) \land size(id_1,s_1) \\ \qquad\qquad \land pos_r(id_2,y_2) \land x_2 = pos_x(id_2) \land x_1 + s_1 \leq x_2 \end{array}\right\}$$

Applying definition introduction would then add the definition

$$\{\forall id\ y : pos(id,pos_x(id),y) \leftarrow pos_r(id,y)\}\,.$$

For the new theory, another functional dependency can be proven, namely $d_{total}\,\langle pos_r[id,nb],\{1\},2\rangle$. Again applying dep-reduce introduces the function $pos_y[id \mapsto nb]$. Lastly, we apply dep-reduce for the dependency $d_{total}\,\langle size[id,nb],\{1\},2\rangle$, introducing the function $size[id \mapsto nb]$. After these steps, the definition is rewritten into

$$\left\{\begin{array}{l} \forall id_1\ id_2 : leftOf(id_1,id_2) \leftarrow \exists x_1\ y_1\ x_2\ y_2\ s_1 : \\ \qquad\qquad y_1 = pos_y(id_1) \land x_1 = pos_x(id_1) \land s_1 = size(id_1) \\ \qquad\qquad \land y_2 = pos_y(id_2) \land x_2 = pos_x(id_2) \land x_1 + s_1 \leq x_2. \end{array}\right\}$$

## 6.3.2 Reducing the Number of Variables

While we have now replaced the symbol *pos* by symbols $pos_x$ and $pos_y$ of lower arity in our running example, we have not eliminated any variables. However, postprocessing can do so. For example, the body of the rule of the above example can be simplified into $\exists s_1 : size(id_1, s_1) \land pos_x(id_1) + s_1 \leq pos_x(id_2)$. The $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$ grounder as introduced in previous chapters does a poor job on such formulas, as it is aimed at grounding human-written theories. To that end, we add additional parse-tree transformations for theories (applied on formulas in NNF). In the rewriting, only atoms $x = f(\bar{t})$ are used where the type of $x$ is the output type of $f$. If the type of $x$ is empty, the atom is false.

The transformation VARIABLE-REPLACE$\langle\mathcal{T}\rangle$ replaces variables in $\mathcal{T}$ by equivalent terms, according to the following rules, until fixpoint.

- If an atom $x = f(\bar{t})$, with $x$ a variable, is a conjunct of a formula $\varphi$, replace $x$ by $f(\bar{t})$ in all other conjuncts.
- Similarly, if an atom $x \neq f(\bar{t})$ is a disjunct of a formula $\varphi$, replace $x$ by $f(\bar{t})$ in all other disjuncts.
- Any set $\{\bar{x} : x_i = f(\bar{t}) \land \varphi : x_i\}$, with $x_i \in \bar{x}$, is replaced by $\{\bar{x} : \varphi : f(\bar{t})\})$.
- For any rule $\forall \bar{x} : head \leftarrow \varphi$ where $\varphi$ has a conjunct $x = f(\bar{t})$, occurrences of $x$ in the head are replaced by $f(\bar{t})$.

The transformation VARIABLE-ELIMINATION$\langle\mathcal{T}\rangle$ removes quantifications from a theory $\mathcal{T}$ by applying the following rules.

- For any formula $\exists \bar{x} : \varphi$, if the only occurrence of $x_i \in \bar{x}$ is in a conjunct $x_i = f(\bar{t})$ of $\varphi$, the conjunct is removed if $f$ is total, otherwise it is replaced by $denotes(f(\bar{t}))$. The same transformation is applied for rules $\forall \bar{x} : head \leftarrow \varphi$ and for sets $\{\bar{x} : \varphi : t\}$.
- For any formula $\forall \bar{x} : \varphi$, if the only occurrence of $x_i \in \bar{x}$ is in a disjunct $x_i \neq f(\bar{t})$ of $\varphi$, the disjunct is removed if $f$ is total, otherwise it is replaced by $\neg denotes(f(\bar{t}))$[3].
- If also a structure $\mathcal{I}$ is given, then any formula of the form $\forall x : \varphi$ or $\exists x : \varphi$, such that $x$ does not occur in $\varphi$, is replaced by $\varphi$, if the type of $x$ is not empty in $\mathcal{I}$.

Both transformations preserve equivalence (without proof). Naturally, the transformations are also applied for atoms $f(\bar{t}) = x$. They can be extended for atoms $x = t'$ where $t'$ is not of the form $f(\bar{t})$ and to the case where the type of $x$ is different from the output type of $f$, not detailed here.

---

[3] A formula $\forall x : \neg(x = f(\bar{t}))$ is equivalent with $\neg\exists x : x = f(\bar{t})$ which is equivalent with $\neg denotes(f(\bar{t}))$.

**Example 6.3.5** (Continued from Example 6.3.4). Applying transformation VARIABLE-REPLACE$\langle\mathcal{T}\rangle$ to the definition obtained after dep-reduce has the following result.

$$\left\{\begin{array}{l} \forall id_1\ id_2 : leftOf(id_1, id_2) \leftarrow \exists x_1\ y_1\ x_2\ y_2\ s_1 : \\ \qquad\qquad\qquad y_1 = pos_y(id_1) \wedge x_1 = pos_x(id_1) \wedge s_1 = size(id_1) \\ \qquad\qquad\qquad \wedge y_2 = pos_y(id_2) \wedge x_2 = pos_x(id_2) \\ \qquad\qquad\qquad \wedge pos_x(id_1) + size(id_1) \leq pos_x(id_2) \end{array}\right\}$$

And as the final step, applying VARIABLE-ELIMINATION$\langle\mathcal{T}\rangle$ then results in

$$\left\{\ \forall id_1\ id_2 : leftOf(id_1, id_2) \leftarrow pos_x(id_1) + size(id_1) \leq pos_x(id_2)\ \right\}.$$

If we now apply the rewriting strategy, as presented above, to the whole packing example, we end up with the following theory, which is significantly more compact and readable.

$$\begin{array}{l} \forall id_1\ id_2 : id_1 \neq id_2 \Rightarrow noOverlap(id_1, id_2) \\ \forall id : pos_x(id) \geq 0 \wedge pos_y(id) \geq 0 \\ \forall id : pos_x(id) + size(id) \leq area_w \wedge pos_y(id) + size(id) \leq area_h \\ \left\{\begin{array}{ll} \forall id_1\ id_2 : leftOf(id_1, id_2) & \leftarrow pos_x(id_1) + size(id_1) \leq pos_x(id_2) \\ \forall id_1\ id_2 : below(id_1, id_2) & \leftarrow pos_y(id_1) + size(id_1) \leq pos_y(id_2) \\ \forall id_1\ id_2 : noOverlap(id_1, id_2) & \leftarrow leftOf(id_1, id_2) \vee leftOf(id_2, id_1) \\ & \quad \vee below(id_1, id_2) \vee below(id_2, id_1) \end{array}\right\} \\ \left\{\begin{array}{ll} \forall id : pos(id, pos_x(id), pos_y(id)) & \leftarrow \top \\ \forall id : size(id, size(id)) & \leftarrow \top \\ \forall id : area(area_w, area_h) & \leftarrow \top \end{array}\right\} \end{array}$$

**Example 6.3.6.** As a second example of the ideas presented here, consider the problem of scheduling courses at a university. A naive modeler might use a symbol *planned*/5 to associate a session with its student group, classroom, time slot and teacher all at once. The restriction that a teacher cannot teach multiple sessions at the same time might then be expressed by a sentence

$$\forall s\ sg\ c\ ts\ te : planned(s, sg, c, ts, te) \Rightarrow$$

$$\neg\exists s_2\ sg_2\ c_2 : s_2 \neq s \wedge planned(s_2, sg_2, c_2, ts, te)$$

It can be considered a naive encoding as the sentence has an impractical grounding size in the order of $|sessions|^2 \times |groups|^2 \times |rooms|^2 \times |slots| \times |teachers|$ atoms. However, as all those relations are functional, the detection and rewriting of functional dependencies will split *planned* in four functions and reduce the above sentence to

$$\forall s\ s_2 : s \neq s_2 \land teacher(s) = teacher(s_2) \Rightarrow \neg(slot(s) = slot(s_2))$$

This is the theory an experienced modeler would construct, but generated from the specification of an inexperienced user!

### 6.3.3 Rewriting of Defined Symbols

So far, we have only handled dependencies for non-defined symbols. When the symbol $s$ is defined, the rewriting dep-reduce is first applied to the whole theory, but we are left with occurrences of $s$ in the heads of its defining rules. Afterwards, these rules are replaced by rules for the new symbols introduced by dep-reduce. This is achieved by the following definition.

**Definition 6.3.7** (Define new symbols)**.** Rules defining the new symbols are added as follows, distinguishing several cases.

- A dependency $d\ \langle f[\overline{\tau} \mapsto \tau_n], S, j\rangle$ with $j \neq n$ and $n \notin S$. Each rule $f(\overline{x}) = x_n \leftarrow \varphi$ is replaced by $f_d(\overline{x}|_S) = x_j \leftarrow \varphi$ and $f_r(\overline{x}|_{\{j\}^C}) = x_n \leftarrow \varphi$.
- A dependency $d\ \langle f[\overline{\tau} \mapsto \tau_n], S, j\rangle$ with $j = n$. Each rule $f(\overline{x}) = x_n \leftarrow \varphi$ is replaced by $f_d(\overline{x}|_S) = x_n \leftarrow \varphi$ and $P_r(\overline{x}) \leftarrow \varphi$.
- A dependency $d\ \langle P[\overline{\tau}], S, j\rangle$. Each rule $P(\overline{x}) \leftarrow \varphi$ is replaced by $f_d(\overline{x}|_S) = x_j \leftarrow \varphi$ and $P_r(\overline{x}|_{\{j\}^C}) \leftarrow \varphi$; the latter only if $\#(S) < n - 1$.

**Example 6.3.8.** Consider a weighted graph defined by its edges $edge(node, node)$ and a partial cost function $cost[node, node \mapsto weight]$ (positive weights) defined for each edge. The relation $minreach[node, weight]$, that expresses whether a node is reachable from a given start node $start[\mapsto node]$ and the minimal cost for reaching it from $start$, can be defined as follows.

$$\left\{ \begin{array}{ll} & minreach(start, 0) \\ \forall x : & minreach(x, min(\{y\ c_y : edge(y,x) \land minreach(y, c_y) \\ & \qquad\qquad\qquad\qquad\qquad : cost(y,x) + c_y\})) \end{array} \right\}$$

This definition entails the dependency $d_{partial}\ \langle minreach[node, weight], \{1\}, 2\rangle$, so can be rewritten as follows, introducing the new partial function symbol $cost_{reach}[node \mapsto weight]$.

$$\left\{ \begin{array}{ll} & cost_{reach}(start) = 0 \\ \forall x : & cost_{reach}(x) = min(\{y : edge(y,x) : cost(y,x) + cost_{reach}(y)\}) \end{array} \right\}$$

**Proposition 6.3.9.** *Let $\mathcal{T}$ be a theory and $d$ a functional dependency of $\mathcal{T}$. Let $\mathcal{T}'$ be the theory obtained after applying the* dep-reduce *rewriting of Definition 6.3.1 for $d$, the definition introduction of Definition 6.3.2 and the definition of new symbols of Definition 6.3.7. Then $\mathcal{T}'$ is strongly $voc(\mathcal{T})$-equivalent with $\mathcal{T}$.*

The proof is provided in Appendix A.2.

### Handling Defined Functions

The search algorithm as presented in Chapter 4 has no support for defined function symbols. Most existing approaches first transform such definitions into their graph equivalent. Such a transformation increases the quantifier nesting and defies the purpose of introducing those functions in the first place. To the best of our knowledge, there is only one search algorithm that natively supports defined function symbols, which was presented very recently by Aziz et al. [Aziz et al., 2013]; it is an approach in the context of stable semantics that works by defining a function by providing rules for its lower or upper bounds.

Here, we present an alternative approach, that transforms a theory with defined functions into a theory without them, with the guarantee that the size of the grounding only increases by a constant factor. More specifically, it does not introduce quantifications over types that were not already quantified elsewhere in the theory.

Consider again the final, total definition in Example 6.3.8 (call it $\Delta$), namely

$$\left\{ \begin{array}{ll} & cost_{reach}(start) = 0 \\ \forall x: & cost_{reach}(x) = min(\{y : edge(y,x) : cost(y,x) + cost_{reach}(y)\}) \end{array} \right\}$$

To handle the defined function $cost_{reach}$, we remove $\Delta$ from the theory and replace it as follows. First, the completion of the rules defining $cost_{reach}$ ($\text{comp}(cost_{reach}, \Delta)$), is added to the theory. Second, we need to prevent positive loops over $cost_{reach}$. The crucial observation here is that, as $cost_{reach}$ is a function and any model will satisfy the completion, no positive loops are possible over two atoms $cost_{reach}(d) = d'$ and $cost_{reach}(d) = d''$ with $d' \neq d''$. Hence, can project the codomain away, as we only need to care about loops over heads that define different domain terms over $cost_{reach}$. For this, a new predicate symbol $red_{cost_{reach}}[node]$ is introduced and a sentence is added that expresses that $red_{cost_{reach}}(d)$ is true iff the corresponding domain term $cost_{reach}(d)$ is denoting. Last, a definition for $red_{cost_{reach}}$ is created that will have the same positive loops as $\Delta$. This is done by (i) replacing the head of the rule defining $cost_{reach}$ with $red_{cost_{reach}}(x)$ and adding the original head conjunctively to the body, and (ii) adding an atom $red_{cost_{reach}}(t)$ conjunctively to every occurrence of a term $cost_{reach}(t)$ except to the defined term occurrence (just moved to the body). The result is the following set of sentences, model-equivalent to $\Delta$.

$$\begin{array}{l} \mathrm{comp}(cost_{reach}, \Delta) \\ \forall x : red_{cost_{reach}}(x) \Leftrightarrow denotes(cost_{reach}(x)) \\ \left\{ \begin{array}{l} red_{cost_{reach}}(start) \\ \forall x : red_{cost_{reach}}(x) \leftarrow cost_{reach}(x) = min(\{y : edge(y,x) \land red_{cost_{reach}}(y) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad : cost(y,x) + cost_{reach}(y)\}) \end{array} \right\} \end{array}$$

### 6.3.4 Integration Within Model Expansion

When our theory rewriting is done as part of a task such as model expansion, we cannot just replace a symbol with a set of symbols of lower arity. Indeed, the input structure is over the original vocabulary and also the model should be presented to the user in the original vocabulary. The rules given in Definition 6.3.2 provide the link between both vocabularies. They are used during the grounding phase to translate the given input structure into corresponding input in the new vocabulary. They are not needed during the solving (hence need not be grounded), but are used again to translate the model expressed in the new vocabulary back to the original vocabulary. As discussed previously, also the latter can done efficiently (without grounding), by bottom-up or tabled evaluation of the definitions.

## 6.4 Experimental Results

To experimentally verify the ideas presented in this chapter, we implemented the detection and rewriting algorithm in IDP. Currently, we only implemented the weak FO($\cdot$)-to-FO transformations, as there are few FO($\mathbb{Z}$) provers available and we do not expect much gain from current day theorem provers as usually inductive reasoning is required. To detect dependencies, we experimented with the award-winning FOF provers SPASS [Weidenbach et al., 2009], Princess [Rümmer, 2008] and Vampire [Kovács and Voronkov, 2013].

As benchmarks, we used the benchmarks of the System track of the 2013 ASP competition [Alviano et al., 2013]. For these benchmarks, encodings are provided by the competition in the ASP-core-2 language, an ASP language not supporting uninterpreted functions (so ideal for our evaluation purposes). In our experiments, we used faithful translations of these encodings and instances to FO($\cdot$)$^{\mathrm{IDP}}$. We added type information (required by IDP) and, more importantly, constraints on the input structure if these were specified in the natural language benchmark description. Such constraints were often not modeled, but are crucial for offline detection. We complemented these benchmarks with the examples on scheduling (denoted "OO-Database" in the results) and packing (denoted "Packing") presented in this chapter. For the latter, instances of the 2011 ASP competition were used, for the former new instances were crafted.

For the resulting specifications, we did two types of experiments.[4] In the first series of experiments, the detection and rewriting algorithm was applied to each of the encodings to measure how many functional dependencies were detected and how much the rewriting reduced the number of quantified variables. In the second series of experiments, we evaluated the effect on the performance of model expansion to effectively solve the benchmarks.

### 6.4.1 Detection and Rewriting Experiments

For the first series of experiments, we ran the FO provers Vampire 3.0, Princess CASC-2013-06-14 and SPASS 3.7 offline on all the encodings, using the weak $\mathrm{FO}(\cdot)$-to-FO transformation.

In Table 6.1, we show the number of functional dependencies each prover managed to detect, given either a timeout of 2 or of 10 seconds per prover call. Out of 19 benchmarks, functional dependencies were detected in all but 3 benchmarks. The results show that SPASS is clearly the strongest prover for our class of problems (while Vampire is in fact the strongest FOF prover according to the competition), and for SPASS, the timeout of 2 seconds is generally sufficient (only in one case, more functions were detected with 10 seconds timeout). Princess with a 10 second timeout comes close to the results of SPASS, but both Princess with a 2 second timeout and both setups of Vampire perform clearly inferior. Specifically Vampire is able to detect almost no functions whatsoever.

In accordance with these results, we decided to select SPASS as the default prover, with a default timeout of 2 seconds. In Table 6.2, more detailed results of that setup are provided. In the 16 benchmarks in which functions were detected, 45% of the detected dependencies were partial, of which 75% were detected in two benchmarks. The subsequent rewrite transformation erased on average 52% of all quantified variables, with peaks above 85%, and was able to strongly reduce the size of the grounding. Total detection time ranged from

---

[4] All experiments were run on an 64-bit Ubuntu 13.10 system with a quad-core 2.53 Ghz processor and 8 Gb of RAM.

less than 1 second to 450 seconds and was directly proportional to the space of index sets (depending on the number of symbols and their arity).

| | SPASS | | Princess | | Vampire | |
|---|---|---|---|---|---|---|
| Benchmark | 10*s* | 2*s* | 10*s* | 2*s* | 10*s* | 2*s* |
| Perm. P. Matching | 3 | 3 | 3 | 0 | 0 | 0 |
| Valves Location | 1 | 1 | 1 | 1 | 1 | 1 |
| Still-Life | 0 | 0 | 0 | 0 | 0 | 0 |
| Graceful Graphs | 2 | 2 | 2 | 0 | 0 | 0 |
| Bottle Filling | 4 | 4 | 4 | 4 | 4 | 4 |
| NoMystery | 16 | 16 | 11 | 0 | 0 | 0 |
| Ricochet Robots | 13 | 13 | 5 | 0 | 0 | 0 |
| Reachability | 0 | 0 | 0 | 0 | 0 | 0 |
| Visit All | 2 | 2 | 2 | 1 | 0 | 0 |
| Knight's Tour | 6 | 4 | 0 | 0 | 0 | 0 |
| Crossing Minim. | 3 | 3 | 3 | 0 | 0 | 0 |
| Solitaire | 3 | 3 | 0 | 0 | 0 | 0 |
| Weighted Sequence | 6 | 6 | 0 | 0 | 0 | 0 |
| Stable Marriage | 5 | 5 | 1 | 0 | 0 | 0 |
| Incremental Sched. | 8 | 8 | 0 | 0 | 0 | 0 |
| Maximal Clique | 0 | 0 | 0 | 0 | 0 | 0 |
| Graph Colouring | 1 | 1 | 1 | 1 | 0 | 0 |
| Database | 5 | 5 | 5 | 0 | 5 | 5 |
| Packing | 5 | 5 | 5 | 0 | 0 | 0 |

Table 6.1: The total number of functional dependencies detected for each benchmark of the 2013 ASP competition. Results are provided in function of the prover used and the time limit for calls to the prover.

Close inspection of the results showed that, as expected, the prover was unable to detect functional dependencies in expressions where inductive reasoning was required to prove the dependency. This is for example the case for sentences of the form $\forall x : P(x, next(t)) \Leftrightarrow (x = initialvalue \land t = start) \lor (\ldots P(y,t)\ldots)$, which occur frequently in, e.g., planning problems. It becomes even more tedious if time is modeled over the natural numbers, replacing $next(t)$ with $t+1$, etc. After a discussion with Philip Rüemmer, developer of the Princess prover, it became clear that automated induction proofs is still a mostly unsolved theorem proving task. Neither Princess, SPASS nor Vampire support the required inductive reasoning. While one could organize the proof by first proving the property for $t = start$ and afterwards proving the induction step, our current implementation does not.

As the detection time is already significant in the offline (structure-independent)

| Benchmark | #t–#p | $\#var_{in}$ | $\#var_{out}$ | % red | # calls | time |
|---|---|---|---|---|---|---|
| Perm. P. Matching | 3–0 | 21 | 8 | 62% | 24 | 25.49 |
| Valves Location | 1–0 | 27 | 24 | 11% | 89 | 20.01 |
| Still-Life | 0–0 | 31 | 31 | 0% | 30 | 50.01 |
| Graceful Graphs | 1–1 | 32 | 21 | 34% | 26 | 2.02 |
| Bottle Filling | 4–0 | 26 | 14 | 46% | 38 | 1.99 |
| NoMystery | 2–14 | 155 | 84 | 46% | 211 | 405.08 |
| Ricochet Robots | 0–13 | 109 | 77 | 29% | 256 | 510.08 |
| Reachability | 0–0 | 5 | 5 | 0% | 30 | 67.98 |
| Visit All | 0–2 | 26 | 6 | 77% | 43 | 21.01 |
| Knight's Tour | 0–4 | 80 | 68 | 15% | 102 | 209.05 |
| Crossing Minim. | 3–0 | 42 | 23 | 45% | 82 | 29.74 |
| Solitaire | 0–3 | 114 | 54 | 53% | 94 | 202.07 |
| Weighted Sequence | 6–0 | 50 | 39 | 22% | 30 | 53.01 |
| Stable Marriage | 5–0 | 24 | 3 | 88% | 6 | 0.42 |
| Incremental Sched. | 6–2 | 89 | 45 | 46% | 146 | 59.03 |
| Maximal Clique | 0–0 | 7 | 7 | 0% | 20 | 0.78 |
| Graph Colouring | 1–0 | 9 | 4 | 6% | 12 | 0.58 |
| Database | 5–0 | 15 | 2 | 87% | 2 | 0.2 |
| Packing | 5–0 | 37 | 8 | 78% | 67 | 10.99 |

Table 6.2: Detailed results for the SPASS setup with 2 seconds prover timeout. It shows the number of total (#t) and partial (#p) functions detected, the number of variable quantifications in the input ($\#var_{in}$) and after rewriting ($\#var_{out}$) and their relative reduction (%), the number of prover calls and the total time taken (in seconds) by the detection and rewriting algorithm.

case, we expect the online setup will be too costly for the current set of theorem provers if large domains or interpretations are involved, as the prover has to reason on the UNA and DCA. For the online case, we conclude it is probably better to use other approaches. For example for interpreted symbols, the function constraints can be directly evaluated in the structure.

### 6.4.2 Model Expansion Experiments

In a second series of experiments, we evaluated the effect of the rewriting on the performance. For this, we selected all benchmarks in which total functions were detected.[5] The transformation in Section 6.3.3 to remove defined function

---

[5] At the time of these experiments, partial functions were not completely supported yet; naturally, the performance of benchmarks in which no functions were detected would not be affected.

| Benchmarks<br>Decision | # inst. | # solved | avg. time (sec.) | avg. size |
|---|---|---|---|---|
| `Packing` | 10 | **10**(0) | **1.52**(84.03) | **1.97$^5$**(7.86$^7$) |
| `Perm. P. Matching` | 10 | **10**(3) | **75.60**(514.49) | **6.90$^4$**(4.23$^8$) |
| `Graceful Graphs` | 10 | 10(10) | 20.37(**16.74**) | 7.42$^6$(7.18$^6$) |
| `Bottle Filling` | 10 | 10(10) | **15.83**(142.65) | 1.09$^6$(5.85$^5$) |
| `Graph Colouring` | 10 | 10(10) | 0.54(0.22) | 1.54$^4$(1.54$^4$) |
| `Database` | 5 | **4**(0) | **39.54**(76.88) | **3.59$^5$**(1.76$^7$) |
| Optimization | # inst. | # optimal | # best sol | avg. size |
| `Valves Location` | 10 | **5** (0) | **10** (0) | 2.28$^6$(2.76$^6$) |
| `Crossing Minim.` | 10 | **1** (0) | **6** (4) | **3.31$^4$**(1.02$^5$) |

Table 6.3: Model expansion results comparing the `pred` and `func` setups. Reports the number of instances, number of instances solved (to optimality), average time taken and average grounding size (# atoms) for the *solved* instances. For optimization benchmarks, it is also reported which setup found the best solution how often. Results are shown as `func`**-result (**`pred`**-result)**.

symbols was not implemented yet.

For each selected benchmark, we randomly selected 10 instances of the 2013 ASP competition, 10 for the `Packing` problem and 5 for the database problem. A time limit of 1000 seconds and a memory limit of 3 Gb were used. The results are shown in Table 6.3, reporting the number of (optimal) solutions found, the average solving time and the average size of the grounding.

We refer to the setup using the original encodings as `pred` and the setup using the preprocessed encodings as `func`.

The results show that, even without support for partial functions, function detection results in significant improvements. In all tested benchmarks, `func` never solves less instances (optimally) than `pred`. Only for `Crossing Minim.`, `pred` finds a better solution than `func` within the time limit in four out of ten cases. The solving time for `func` is always significantly less, with even a 50-fold speedup for `Packing`, except for a 25% slowdown on `Graceful Graphs`. The grounding size is reduced by orders of magnitude for four out of 8 benchmarks and never significantly higher.

To summarize, offline detection of functional dependencies is certainly worthwhile, as detected dependencies can result in a significant performance boost for the solver, while the performance is unaffected when none are detected. Whether to use online detection depends on the application at hand as the proving overhead could be significant.

## 6.5 Conclusion

The main contribution of this chapter are an approach to deduction in the context of FO($\cdot$) and an approach to detect and exploit functional dependencies in declarative problem statements. The developed inference engine provides IDP with support for deduction and allows us to plug in any theorem prover supporting a standardized theorem proving language. Experimental detection results showed that SPASS performs significantly better than the other tested provers on our benchmark set. We have also shown that it is not a trivial theorem proving problem and hence an interesting new prover benchmark, as none of the provers was able to prove all entailed functional dependencies or proved partial dependencies while in fact total ones were entailed.

Moreover, as described in the previous chapter, it facilitates the automatic exploitation of the support for uninterpreted functions in the latest generation of grounders and search algorithms. Model expansion results with such a setup show a significant improvement in terms of problems solved, solving time and grounding size, with very few benchmarks negatively affected.

The detection results also show a large number of detected partial functions. Extrapolating from the significant advantages for supporting total functions in a model expansion engine, we expect similar result for partial functions. Hence, part of future work is to extend the implementation to support partial functions, as described in Chapters 4 and 5.

Another part of future work is to extract other types of implicit knowledge, such as types, definitional totality and more complex finite-domain constraints such as *all-different*. It remains an open question whether theorem provers with additional integer reasoning capabilities, such as Melia [Baumgartner et al., 2012] and Z3 [de Moura and Bjørner, 2008], are capable of proving the presence of more functional dependencies.

# 7

# Interleaving Grounding and Search

While ground-and-solve is a state-of-the-art approach for solving search problems, a bottleneck is the blowup caused by grounding. In previous chapters, we developed techniques to mediate this using preprocessing approaching, such as function detection, allowing a richer ground language as input to the search algorithm and developing a search algorithm that lazily grounds specific types of constraints during grounding.

In this chapter, we present a novel approach to remedy this bottleneck, called *lazy model expansion*, where the grounding is generated lazily (on-the-fly) during search, instead of up-front. The approach works by associating *justifications* to the non-ground parts of the theory. A valid justification for a non-ground formula is a recipe to expand a partial structure into a more precise (partial) structure that satisfies the formula. Given a partial structure and a valid justification for each of the non-ground formulas, a (total) structure can be constructed that satisfies all the non-ground formulas and extends the given structure. Consequently, model generation can be limited to the grounded part of the theory; if a model is found for that part, it can be extended to a model of the whole theory. However, a new assignment during model generation can conflict with one of the justifications. In that case, an alternative justification needs to be sought. If none is found, the associated formula can be split in two parts, one part that is grounded and one part for which a valid justification is still available.

**Example 7.0.1.** Consider the `Sokoban` toy problem, a planning problem where a robot has to push blocks around on a 2-D grid to arrange them in a given goal configuration. A constraint on the move action is that the target position of the moved block is currently empty, which can be expressed as

$$\forall(t,b,p) \in \mathit{time} \times \mathit{block} \times \mathit{pos} : \mathit{move}(b,p,t) \Rightarrow \mathit{empty}(p,t). \tag{7.1}$$

As it is not known in advance how many time steps are needed, one ideally wants to assume a very large or even infinite number of steps. Using ground-and-solve, this blows-up the size of the grounding.

Given a partial plan $P$ (a structure) that only moves block $b_1$ to position $p_1$ at time $t_1$ and in which $\mathit{empty}(p_1,t_1)$ is true, the partial plan does not violate this constraint. In that case, there is a simple way to extend the plan so that it certainly satisfies the constraint: do not move any more blocks. Indeed, the recipe "$\mathit{move}(b,p,t)$ is false except for $\mathit{move}(b_1,p_1,t_1)$" can be used to *construct* a complete plan in which sentence 7.1 is satisfied.

We do not immediately apply this recipe, but use it as a *default*, to be applied after a (partial) plan has been found that satisfies the goal configuration. However, if during search it is decided to move a block $b_2$ at time $t_2$ to position $p_2$, this conflicts with the recipe, so we are no longer guaranteed it will result in a complete, valid plan. In that case, one can split up sentence (7.1) into sentences (7.2), for the instantiation $\langle \mathit{time}/t_2, \mathit{block}/b_2, \mathit{pos}/p_2\rangle$, and (7.3), for the remainder. For sentence (7.3), the recipe still holds if $\mathit{move}(b_2,p_2,t_2)$ is added as (second) exception, so only sentence (7.2) needs to be grounded and taken into account during search.

$$\mathit{move}(b_2,p_2,t_2) \Rightarrow \mathit{empty}(p_2,t_2) \tag{7.2}$$

$$\forall(t,b,p) \in \mathit{time} \times \mathit{block} \times \mathit{pos} \backslash (t_2,b_2,p_2) : \mathit{move}(b,p,t) \Rightarrow \mathit{empty}(p,t) \tag{7.3}$$

Lazy grounding is a form of constraint propagation. In general, a constraint propagator for a constraint $c$ is responsible for detecting that an assignment to variables in $c$ violates the constraint and to derive consequences of $c$ under (partial) assignments. Either by observing that a particular variable is forced to take a specific value or by adding extra constraints to the constraint store. Lazy grounding acts as a propagator for non-ground constraints. As soon as a constraint's justification is violated, it adds some new ground constraints to the store and creates/reuses justifications for the remaining non-ground parts of the constraint. Interestingly, it can be shown (see Section 7.4) that justifications can be chosen in such a way that propagation that would occur in the fully grounded theory is not delayed.

In this chapter, we restrict ourselves to FO($ID$) [Denecker and Ternovska, 2008], the language which extends FO with inductive definitions. Extending the approach to other FO extensions is part of future work, but possible approaches are already discussed towards the end of the chapter. The main contributions of the chapter are:

- A theoretical framework for *lazy model expansion*. By aiming at minimally instantiating quantified variables, it paves the way for a solution to the long-standing problem of handling quantifiers in search problems, encountered, e.g., in the fields of ASP [Lefèvre and Nicolas, 2009] and SAT Modulo Theories [Ge and de Moura, 2009]. The framework also generalizes existing approaches that are related to the grounding bottleneck such as incremental domain extension [Claessen and Sörensson, 2003] and lazy clause generation [Stuckey, 2010].
- A complete algorithm for lazy model expansion for the logic FO($ID$), the extension of FO with inductive definitions (a language closely related to ASP as shown in [Denecker et al., 2012]). This includes efficient algorithms to derive consistent sets of justifications and to maintain them throughout changes in a partial structure (e.g., during search).
- An implementation extending the IDP knowledge base system and experiments that illustrate the power and generality of lazy grounding.

The chapter is organized as follows. In Section 7.1, the necessary background and notations are introduced. Formal definitions of the concepts used for lazy grounding are presented in Section 7.2, followed by a presentation of the relevant algorithms and optimizations in Sections 7.3 and 7.4. Experimental evaluation is provided in Section 7.5, followed by a discussion on related and future work and a conclusion. A preliminary version of this work appeared as [De Cat et al., 2012].

## 7.1 Preliminaries

In this section, we review some relevant concepts and introduce notations.

### 7.1.1 FO($ID$)

Without loss of generality, we limit FO and FO($ID$) to its function-free fragment. Recall, function symbols can always be replaced by graph predicates [Enderton, 2001]. We also assume that, in any definition, any domain atom is defined by at most one rule; this is again without loss of generality.

For any set of domain atoms $S$, we use $\mathcal{I}|_S$ to denote the restriction of $\mathcal{I}$ to $S$: $a^{\mathcal{I}|_S} = a^{\mathcal{I}}$ if $a \in S$ and $a^{\mathcal{I}|_S} = \mathbf{u}$ otherwise. If $S$ is a set of predicate symbols, then $\mathcal{I}|_S$ is the restriction of $\mathcal{I}$ to the set of all domain atoms of these symbols. We call $\mathcal{I}$ a two-valued structure of $S$ if $\mathcal{I}$ is two-valued on domain atoms of $S$ and unknown otherwise.

#### Model Semantics

Below, we present a formalization of the well-founded semantics using the notion of *justification*. As explained in [Denecker and Ternovska, 2008], it is a methodological guideline in FO($ID$) that the user write total definitions. The techniques presented in this chapter require that definitions are total.

We always work in the context of a known domain $D$. To simplify the presentation, we use a slightly different notion of the well-founded semantics, in terms of defined domain atoms (over $D$). Given a definition $\Delta$, a domain atom $P(\overline{d})$ is *defined* by $\Delta$ if there exists a rule $\forall \overline{x} \in \overline{D} : P(\overline{x}) \leftarrow \varphi \in \Delta$ such that $\overline{d} \in \bar{D}$. Otherwise $P(\overline{d})$ is *open* in $\Delta$. A domain literal $[\neg]P(\overline{d})$ is defined by $\Delta$ if $P(\overline{d})$ is defined by $\Delta$. The sets of defined and open domain literals of $\Delta$ are then denoted as $\mathrm{def}(\Delta)$ and $\mathrm{open}(\Delta)$, respectively.

The result is that a definition $\Delta$ does not impose a value for undefined domain atoms, even if other domain atoms over the same symbol are defined by $\Delta$.

When a definition defines a predicate $P$ only for $\overline{x}$ in $\overline{D}'$, a subset of the interpretation $\overline{D}$ of $type(\overline{x})$, one can restore the original well-founded semantics by adding the rule $\forall \overline{x} \in \overline{D} \setminus \overline{D}' : P(\overline{x}) \leftarrow \bot$.

**Canonical Theories.** To simplify the presentation, the lazy grounding techniques are presented here for theories of the form $\{P_{\mathcal{T}}, \Delta\}$, with $P_{\mathcal{T}}$ a propositional symbol, and $\Delta$ a single definition with function-free rules. This is without loss of generality. First, as mentioned above, a theory can be made function-free using standard techniques. Second, multiple definitions can be combined into one as described in [Denecker and Ternovska, 2008, Mariën et al., 2004], for example if a stratification exists or by introduction of additional predicate symbols. Furthermore, we assume negation only occurs in front of atoms and compound formulas are built by conjunction, disjunction and (existential and universal) quantification. We also assume that $D$ is the domain of structures.

The methods below can be extended to standard theories with functions, FO axioms and multiple definitions, and such extended methods have been implemented in our system. However, this introduces a number of rather irrelevant technicalities which we want to avoid here.

### Justifications

Recall that structures correspond one-to-one to sets of domain literals and that, by assumption, for each defined domain atom $P(\overline{d})$ there is a unique rule $\forall \overline{x} \in \overline{D} : P(\overline{x}) \leftarrow \varphi \in \Delta$ such that $\overline{d} \in \overline{D}$.

**Definition 7.1.1** (Direct justification)**.** A *direct justification* for a defined domain literal $P(\overline{d})$ (respectively $\neg P(\overline{d})$) is a consistent set $S$ of domain literals such that for the rule $\forall \overline{x} \in \overline{D} : P(\overline{x}) \leftarrow \varphi$ of $\Delta$ such that $\overline{d} \in \overline{D}$, it holds that $\varphi[\overline{x}/\overline{d}]^S = \mathbf{t}$ (respectively $\varphi[\overline{x}/\overline{d}]^S = \mathbf{f}$).

**Definition 7.1.2** (Justification)**.** A *justification* over $\Delta$ is a graph $J$ on the set of domain literals such that if $(l, l') \in J$, then $\{l'' \mid (l, l'') \in J\}$ is a direct justification of $l$.

With a justification $J$, we can associate the total function from domain literals $l$ to sets $J(l) = \{l'' \mid (l, l'') \in J\}$ of domain literals. If $J(l) \neq \emptyset$, then $l$ is a defined literal and $J(l)$ is a direct justification of $l$. In the sequel we say that $J$ is defined in $l$ if $J(l) \neq \emptyset$. A justification is denoted as a set of pairs $l \rightarrow S$, with $S$ the direct justification of $l$.

**Example 7.1.3.** Consider a domain $D = \{d_1, \ldots, d_n\}$ and the definition $\Delta$

$$\left\{ \begin{array}{ll} \forall x \in D : P(x) & \leftarrow Q(x) \lor R(x) \\ \forall x \in D : Q(x) & \leftarrow P(x) \end{array} \right\}$$

The unique (minimal) direct justification for $Q(d_i)$ is $\{P(d_i)\}$ and for $\neg Q(d_i)$ is $\{\neg P(d_i)\}$. Direct justifications for $P(d_i)$ are both $\{Q(d_i)\}$ and $\{R(d_i)\}$ while the only (minimal) direct justification for $\neg P(d_i)$ is $\{\neg Q(d_i), \neg R(d_i)\}$. Atoms $R(d_i)$ are open and have no direct justification.

**Definition 7.1.4** (Justification subgraph)**.** Let $J$ be a justification over $\Delta$. The justification *for a literal $l$* is the subgraph $J_l$ of nodes and edges of $J$ reachable from $l$. The justification *for a set of literals* $L$ is the subgraph $J_L$ of nodes and edges of $J$ reachable from any $l \in L$.

A justification $J$ is *total* for $l$ if $J$ is defined in each defined literal reachable from $l$; it is total for a set of literals $L$ if it is total for each literal in $L$; it is total if it is total for all literals for which $J$ is defined. A justification $J$ is *consistent with* a structure $\mathcal{I}$ when none of the literals in $J$ is false in $\mathcal{I}$.

A path in a justification $J$ is a sequence $l_0 \rightarrow l_1 \rightarrow \ldots$ such that, if $l_i \rightarrow l_{i+1}$, then there is an edge from $l_i$ to $l_{i+1}$ in $J$. An infinite path may be cyclic or not.

**Definition 7.1.5.** A *cycle* in a justification $J$ is the set of domain literals on an infinite path of $J$. A cycle is *positive* if the path has a tail consisting of positive literals; it is *negative* if the path has a tail consisting of negative literals; it is *mixed* otherwise.

Cycles are finite iff the path is cyclic. If the domain $D$ is finite, cycles are always finite. Intuitively, a justification provides an argument to the truth of its nodes. If this argument contains a positive cycle, we consider it unfounded; negative cycles, though, are accepted.

**Definition 7.1.6** (Justifies)**.** A justification $J$ over $\Delta$ *justifies* a set of literals $L$ defined in $\Delta$ (a set of literals *has a justification*) if (i) $J_L$ is total for $L$; (ii) cycles in $J_L$, if any, are negative; (iii) the set of literals in $J_L$ is consistent.

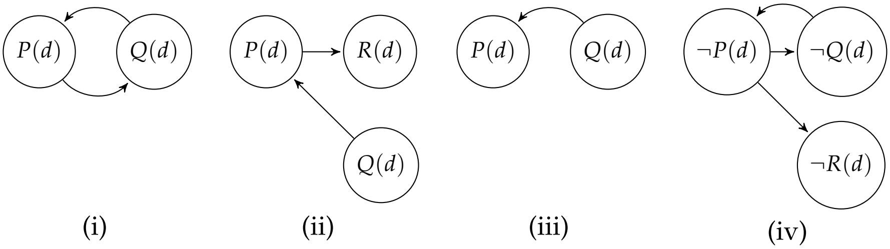


Figure 7.1: Justifications for definition $\Delta$ in Example 7.1.3, with $d \in D$.

**Example 7.1.7.** In Figure 7.1, we show a few possible justifications ((i) to (iv)) over definition $\Delta$ from Example 7.1.3, that contain the defined domain atoms $P(d)$ and $Q(d)$ ($d \in D$). Justification (ii) justifies $Q(d)$ and (iv) justifies $\neg Q(d)$; (iii), however, is not total for $Q(d)$ and (i) has a positive cycle. Given a structure $\mathcal{I} = \{P(d)\}$, justifications (i), (ii) and (iii) are consistent with it, but not (iv) as $\neg P(d)$ as false in $\mathcal{I}$.

The relationship between justifications and the well-founded semantics has been investigated in different publications [Denecker and De Schreye, 1993, Mariën, 2009]. Below we recall the results on which this chapter relies. The first result states that if $J$ justifies all literals in $L$, then any model $\mathcal{I}$ of $\Delta$ in which the leaves of $J_L$ are true, satisfies all literals in $L$ and in $J_L$.

**Proposition 7.1.8.** *If $J$ is a justification over $\Delta$ that justifies a set of domain literals $L$ then all literals in $J_L$ are true in every model of $\Delta$ in which the (open) leaves of $J_L$ are true.*

For an interpretation $\mathcal{I}_{open}$ that is two-valued for $open(\Delta)$ an in which the leaves are true, the structure $\mathcal{I} = wf_{\Delta}(\mathcal{I}_{open})$ can be computed in time polynomial in the size of the domain, as shown in [Chen and Warren, 1996]. When $\Delta$ is total, $wf_{\Delta}(\mathcal{I}_{open})$ is the two-valued well-founded model of $\Delta$; otherwise, $wf_{\Delta}(\mathcal{I}_{open})$ can be a partial structure.

**Example 7.1.9** (Continued from Example 7.1.7)**.** Consider justification (ii), with $R(d)$ the only literal open in the subgraph for $L = \{Q(d)\}$. Take $\mathcal{I}_{open}$ to be the structure that makes $R(d)$ true and all other domain atoms over $R$ false. Then $\mathcal{I}_{open}$ is an open($\Delta$)-structure $\geq_p \{R(d)\}$. The well-founded evaluation assigns $P(d)$ true (as $R(d)$ is true) and afterwards assigns $Q(d)$ true (as $P(d)$ is now true). Hence, $wf_{\Delta}(\mathcal{I}_{open})$ is a two-valued model of $\Delta$ ($\Delta$ is total) and expands $L$ and all literals reachable from $L$.

**Proposition 7.1.10.** *If $\mathcal{I}$ is a model of $\Delta$, a justification $J$ over $\Delta$ exists that justifies each defined domain literal true in $\mathcal{I}$ and contains only domain literals true in $\mathcal{I}$.*

**Corollary 7.1.11.** *In case $\Delta$ is total, if a justification $J$ over $\Delta$ justifies a set of domain literals $L$, then every two-valued $open(\Delta)$-structure consistent with $J_L$ can be extended in a unique way to a model of $\Delta$ that satisfies all literals of $L$.*

Hence, for a canonical theory $\{P_{\mathcal{T}}, \Delta\}$ (recall, $\Delta$ is total), the theory is satisfiable iff a justification $J$ exists that justifies $P_{\mathcal{T}}$.

### Grounding

For ease of presentation, we start with the grounding algorithm presented below, a simplified version of the algorithm presented in Chapter 4, as the basis of the lazy MX algorithm.

A grounder takes as input a theory $\mathcal{T}$ over vocabulary $\Sigma$, a partial structure $\mathcal{I}$ with domain $D$, interpreting at least $\top$ and $\bot$, and returns a ground theory $\mathcal{T}'$ that is strongly $\Sigma$-equivalent with $\mathcal{T}$ in $\mathcal{I}$. Theory $\mathcal{T}'$ is then called a *grounding* of $\mathcal{T}$ given $\mathcal{I}$. Recall, we assume $\mathcal{T}$ is a canonical theory of the form $\{P_{\mathcal{T}}, \Delta\}$.

One way to compute the grounding is using a top-down process on the theory, iteratively applying grounding steps to direct subformulas of the rule or formula at hand. The grounding algorithm may replace subformulas by new predicate symbols as follows. Let $\varphi[\overline{x}]$ be a formula in $\mathcal{T}$ and let $\overline{D}$ be the domains of $\overline{x}$. *Tseitin transformation* replaces $\varphi$ by the atom $T_\varphi(\overline{x})$, with $T_\varphi$ a new $|\overline{x}|$-ary predicate symbol called a *Tseitin* symbol, and extends $\Delta$ with the rule $\forall \overline{x} \in \overline{D} : T_\varphi(\overline{x}) \leftarrow \varphi$. The new theory is strongly $\Sigma$-equivalent to the original one [Vennekens et al., 2007].

The procedure one_step_ground, outlined in Figure 6, performs one step in the grounding process. Called with a formula or rule $\varphi$ in canonical form, the algorithm replaces all direct subformulas with Tseitin symbols and returns a pair consisting of a ground part $G$ (rules or formulas) and a possibly non-ground part $R$ (rules). If $\varphi$ is a formula, then $G$ consists of ground formulas. Replacing $\varphi$ by the returned ground formulas and extending $\Delta$ with the returned rules produces a theory that is strongly $voc(\mathcal{T})$-equivalent to the original. If $\varphi$ is a rule from $\Delta$, it is replaced by both sets of returned rules and again, the new theory is strongly $voc(\mathcal{T})$-equivalent to the original.

**Algorithm 6:** The one_step_ground algorithm.

```
Function one_step_ground (formula or rule $\varphi$)
    switch $\varphi$ do
        case $[\neg]P(\overline{d})$ return $\langle\{\varphi\},\emptyset\rangle$;
        case $P(\overline{d}) \leftarrow \psi$
            $\langle G,\Delta\rangle$ := one_step_ground($\psi$);
            return $\langle\{P(\overline{d}) \leftarrow \bigwedge_{g\in G} g\},\Delta\rangle$;
        case $\psi_1 \vee \ldots \vee \psi_n$
            return $\langle\{\bigvee_{i\in[1,n]} T_{\psi_i}\},\{T_{\psi_i} \leftarrow \psi_i \mid i \in [1,n]\}\rangle$;
        case $\psi_1 \wedge \ldots \wedge \psi_n$
            return $\langle\{T_{\psi_i} \mid i \in [1,n]\},\{T_{\psi_i} \leftarrow \psi_i \mid i \in [1,n]\}\rangle$;
        case $\forall \overline{x} \in \overline{D} : P(\overline{x}) \leftarrow \psi$
            return $\langle\emptyset,\{P(\overline{x})[\overline{x}/\overline{d}] \leftarrow \psi[\overline{x}/\overline{d}] \mid \overline{d} \in \overline{D}\}\rangle$;
        case $\exists \overline{x} \in \overline{D} : \psi[\overline{x}]$
            return $\langle\{\bigvee_{d\in D} T_{\psi[\overline{x}/\overline{d}]}\},\{T_{\psi[\overline{x}/\overline{d}]} \leftarrow \psi[\overline{x}/\overline{d}] \mid \overline{d} \in \overline{D}\}\rangle$;
        case $\forall \overline{x} \in \overline{D} : \psi[\overline{x}]$
            return $\langle\{T_{\psi[\overline{x}/\overline{d}]} \mid \overline{d} \in \overline{D}\},\{T_{\psi[\overline{x}/\overline{d}]} \leftarrow \psi[\overline{x}/\overline{d}] \mid \overline{d} \in \overline{D}\}\rangle$;
        end
    endsw
```

Grounding a theory then boils down to applying one_step_ground on the sentence $P_\mathcal{T}$ (which copies $P_\mathcal{T}$ to the ground part) and on each rule of the theory and repeatedly applying one_step_ground on the returned rules $R$ (all returned sentences and rules in $G$ are ground). We use `ground` to refer to the algorithm for this overall process.

As discussed in Chapter 4, various improvements exist, such as returning $\top/\bot$ for atoms interpreted in $\mathcal{I}$ and returning $\bot$ from conjunctions whenever a false conjunct is encountered (analogously for disjunctions and quantifications).

Also, algorithm `one_step_ground` introduces a large number of Tseitin symbols. State-of-the-art grounding algorithms use a number of optimizations to reduce the number of such symbols. As these optimizations are not directly applicable to the techniques presented in this chapter, we start from the naive `one_step_ground` algorithm. In Section 7.4, we present an optimized version of `one_step_ground` that introduces less Tseitin symbols and hence results in smaller groundings.

## 7.2 Lazy Grounding and Lazy Model Expansion

We use the term *lazy grounding* to refer to the process of partially grounding a theory and the term *lazy model expansion* (lazy MX) for the process that interleaves lazy grounding with model expansion over the grounded part. In Section 7.2.1, we formalize a framework for lazy model expansion of FO($ID$) theories; in Section 7.2.2, we formalize the instance of this framework that is the basis of our current implementation; in Section 7.2.3, we illustrate its operation.

### 7.2.1 Lazy Model Expansion for FO($ID$) Theories

The input of the lazy MX algorithm consists of a canonical theory $\mathcal{T} = \{P_{\mathcal{T}}, \Delta\}$ and an input structure $\mathcal{I}_{in}$ with domain $D$ interpreting at least $\top$ and $\bot$. We assume that $\Delta$ is total (in $\mathcal{I}_{in}$). The task is then to find models of $\Delta$ in which $P_{\mathcal{T}}$ is true. This is achieved by interleaving lazy grounding with search on the already grounded part. We first focus on the lazy grounding.

The initial input to lazy grounding consists of $P_{\mathcal{T}}$ (handled separately), $\Delta$ and $\mathcal{I}_{in}$. For subsequent steps, the input consists of a set of rules still to be grounded, an already grounded theory and a three-valued structure that is an expansion of the initial input structure.

Each subsequent grounding step can replace non-ground rules by ground rules and might introduce new rules. Hence, the state of the grounding includes a set $\Delta_{\mathrm{g}}$ of ground rules and a set $\Delta_{\mathrm{d}}$ of (possibly) non-ground rules. The definitions have the property that $\Delta_{\mathrm{g}} \cup \Delta_{\mathrm{d}}$ (in what follows abbreviated as $\Delta_{\mathrm{gd}}$)

is voc($\Delta$)-equivalent with the original definition $\Delta$ and hence, $\Delta_{\mathrm{gd}}$ is total. The grounding procedure will guarantee that, at all times, $\Delta_{\mathrm{g}}$ and $\Delta_{\mathrm{d}}$ are total.

Given a partial structure $\mathcal{I}_{in}$ and the rule sets $\Delta_{\mathrm{g}}$ and $\Delta_{\mathrm{d}}$, the key idea behind lazy model expansion is (**i**) to use a search algorithm to search for a model $\mathcal{I}$ of $\Delta_{\mathrm{g}}$ that is an expansion of $\mathcal{I}_{in}$ in which $P_{\mathcal{T}}$ is true; (**ii**) to maintain a justification $J$ such that the literals true in $\mathcal{I}$ and defined in $\Delta_{\mathrm{d}}$ are justified over $\Delta_{\mathrm{gd}}$ and that $J$ is consistent with $\mathcal{I}$; (**iii**) to interleave steps (**i**) and (**ii**) and to move parts of $\Delta_{\mathrm{d}}$ to $\Delta_{\mathrm{g}}$ when some literal defined in $\Delta_{\mathrm{d}}$ that needs to be justified cannot be justified.

Thus, to control lazy model expansion, it suffices to maintain a state $\langle \Delta_{\mathrm{g}}, \Delta_{\mathrm{d}}, J, \mathcal{I} \rangle$ consisting of the grounded rules $\Delta_{\mathrm{g}}$, the rules $\Delta_{\mathrm{d}}$ yet to be grounded, a justification $J$, and a three-valued structure $\mathcal{I}$. Initially, $\mathcal{I}$ is $\mathcal{I}_{in}$, $\Delta_{\mathrm{d}}$ is $\Delta$, $\Delta_{\mathrm{g}} = \emptyset$, and $J$ is the empty graph.

Lazy model expansion searches over the space of *acceptable* states.

**Definition 7.2.1** (Acceptable state)**.** A tuple $\langle \Delta_{\mathrm{g}}, \Delta_{\mathrm{d}}, J, \mathcal{I} \rangle$ of a theory with an atomic sentence $P_{\mathcal{T}}$, a total definition $\Delta$, and an input structure $\mathcal{I}_{in}$ is an *acceptable* state if (**i**) $\Delta_{\mathrm{gd}}$, $\Delta_{\mathrm{g}}$ and $\Delta_{\mathrm{d}}$ are total definitions and $\Delta_{\mathrm{gd}}$ is strongly voc($\Delta$)-equivalent with $\Delta$, (**ii**) no domain atom is defined in both $\Delta_{\mathrm{g}}$ and $\Delta_{\mathrm{d}}$, (**iii**) $J$ is a justification over $\Delta_{\mathrm{gd}}$, (**iv**) $\mathcal{I}$ is an expansion of $\mathcal{I}_{in}$, (**v**) the set $L$ of literals true in $\mathcal{I}$ and defined in $\Delta_{\mathrm{d}}$ are justified by $J$, and (**vi**) $J_L$ is consistent with $\mathcal{I}$.

The lazy model expansion algorithm tries to compute an acceptable state in which $P_{\mathcal{T}}$ is justified. By Corollary 7.1.11, this would entail that a model of $\mathcal{T}$ exists; it can be computed efficiently through well-founded model computation. In intermediate states, the justification may be non-total for $P_{\mathcal{T}}$, contain positive or mixed cycles, or be inconsistent.

Note (**iii**) that the justification must be over $\Delta_{\mathrm{gd}}$. Indeed, assume some literal $l$ is justified over $\Delta_{\mathrm{d}}$. Its justification graph can have a leaf that is defined in $\Delta_{\mathrm{g}}$ and that depends positively or negatively on $l$. Then every attempt to extend this justification graph to a total justification graph that justifies $l$ over $\Delta_{\mathrm{gd}}$ might fail because of a forbidden cycle.

**Proposition 7.2.2.** *Let $\langle \Delta_g, \Delta_d, J, \mathcal{I} \rangle$ be an acceptable state. $\Delta_{gd}$ has a well-founded model that expands all literals true in $\mathcal{I}$ and defined in $\Delta_d$.*

*Proof.* Let $L$ be the set of literals true in $\mathcal{I}$ and defined in $\Delta_{\mathrm{d}}$. As the state is acceptable, $J$ justifies the literals of $L$. Hence, by Corollary 7.1.11, there exists a well-founded model that expands $L$. □

**Example 7.2.3.** Consider the theory $\{P_{\mathcal{T}}, \Delta\}$, with $\Delta$ the definition

$$\left\{ \begin{array}{l} P_{\mathcal{T}} \leftarrow T_1 \vee T_2 \vee T_3 \\ T_1 \leftarrow \forall x \in D : Q(x) \\ T_2 \leftarrow \forall x \in D : R(x) \\ T_3 \leftarrow \exists x \in D : \neg Q(x) \end{array} \right\}$$

Let $\mathcal{I}$ be the structure $\{P_{\mathcal{T}}, T_1\}$ (hence, $T_2$ and $T_3$ are unknown), and $\Delta_{\mathrm{g}}$ and $\Delta_{\mathrm{d}}$ the definitions consisting of the first rule, respectively the remaining rules. Furthermore, let $J$ be $\{T_1 \rightarrow \{Q(d) \mid d \in D\}\}$. The tuple $\langle \Delta_{\mathrm{g}}, \Delta_{\mathrm{d}}, J, \mathcal{I} \rangle$ is then an acceptable state. Indeed, $T_1$ is the only literal in $\mathcal{I}$ that is defined in $\Delta_{\mathrm{d}}$ and it is justified by $J$. The well-founded evaluation, after assigning $\top$ to the open literals of $J$ (i.e., to $\{Q(d) \mid d \in D\}$), derives that $T_1$ is true. Moreover, because $\mathcal{I}$ is a model of $\Delta_{\mathrm{g}}$, $P_{\mathcal{T}}$ is also true in such a well-founded model. Note that $R$ can be interpreted randomly, as no $R$-atoms occur in $\mathcal{I}$ or $J$.

The following theorem states conditions on when the obtained expansion is also a model of $\mathcal{T}$.

**Theorem 7.2.4.** *Let $\langle \Delta_g, \Delta_d, J, \mathcal{I} \rangle$ be an acceptable state of a theory $\mathcal{T} = (P_{\mathcal{T}}, \Delta)$ with input structure $\mathcal{I}_{in}$ such that $P_{\mathcal{T}}$ is true in $\mathcal{I}$ and $\mathcal{I}|_{voc(\Delta_g)}$ is a model of $\Delta_g$. Then there exists a model $\mathcal{M}$ of $\mathcal{T}$ that expands $\mathcal{I}|_{voc(\Delta_g)}$.*

*Proof.* $\mathcal{I}|_{voc(\Delta_{\mathrm{g}})}$ is a model of $\Delta_{\mathrm{g}}$. It follows from Proposition 7.1.10 that there exists a justification $J_g$ over $\Delta_{\mathrm{g}}$ that justifies every true defined literal of $\Delta_{\mathrm{g}}$ and that consists of only domain literals true in $\mathcal{I}|_{voc(\Delta_{\mathrm{g}})}$. We now have two justifications: $J$ and $J_g$. We combine them in one $J_c$ as follows: for each defined literal $l$ of $\Delta_{\mathrm{gd}}$, if $J$ is defined in $l$, we set $J_c(l) = J(l)$; otherwise, we set $J_c(l) = J_g(l)$.

We verify that $J_c$ justifies $P_{\mathcal{T}}$. First, it is total in $P_{\mathcal{T}}$. Indeed, any path from $P_{\mathcal{T}}$ either consists of literals defined in $\Delta_{\mathrm{g}}$, and then it is a branch of the total $J_g$ over $\Delta_{\mathrm{g}}$, or it passes to a literal $l'$ defined in $\Delta_{\mathrm{d}}$, which is justified by $J$ according to condition (**v**) and hence $(J_c)_{l'} = J_{l'}$ is total. As such, from $P_{\mathcal{T}}$ we cannot reach a defined literal of $\Delta_{\mathrm{gd}}$ in which $J_c$ is undefined. Second, $J_c$ does not contain mixed or positive cycles starting from $P_{\mathcal{T}}$. This is because any path from $P_{\mathcal{T}}$ is either a path in $J_g$ or it has a tail in $J$, and neither of these contain mixed or positive cycles. Finally, the set of literals reachable from $P_{\mathcal{T}}$ in $J_c$ is consistent. Also this we can see if we look at paths in $J_c$ from $P_{\mathcal{T}}$: at first we follow $J_g$ which consists of true literals in $\mathcal{I}$, then we may get into a path of $J$ which contains literals that are consistent with $\mathcal{I}$. In any case, it is impossible to reach both a literal and its negation.

It follows from Proposition 7.1.8 that there exists a model of $\Delta_{\mathrm{gd}}$ that expands $\mathcal{I}|_{voc(\Delta_{\mathrm{g}})}$ and in which $P_{\mathcal{T}}$ is true. Since $\Delta_{\mathrm{gd}}$ is strongly equivalent with $\Delta$, the proposition follows. $\square$

Recall that effectively computing such a model $\mathcal{M}$ can be achieved by well-founded evaluation of $\Delta_{\mathrm{gd}}$, with polynomial data complexity, starting from any two-valued open$(\Delta_{\mathrm{gd}}) - \mathit{structure}$ expanding $\mathcal{I}|_{voc(\Delta_{\mathrm{g}})}$.

In the above theorem, it is required that $\mathcal{I}$ is a model of $\Delta_{\mathrm{g}}$. Actually, we do not need to compute a two-valued model of $\Delta_{\mathrm{g}}$. It suffices to search for a partial structure and a justification that justifies $P_{\mathcal{T}}$. So, we can relax this requirement at the expense of also maintaining justifications for literals true in $\mathcal{I}$ and defined in $\Delta_{\mathrm{g}}$.

**Corollary 7.2.5.** *Let $\langle \Delta_g, \Delta_d, J, \mathcal{I} \rangle$ be an acceptable state of a theory $\mathcal{T} = \{P_{\mathcal{T}}, \Delta\}$ with input structure $\mathcal{I}_{in}$ such that $P_{\mathcal{T}}$ is true in $\mathcal{I}$ and $J$ justifies $P_{\mathcal{T}}$ over $\Delta_{gd}$. Then there exists a model $\mathcal{M}$ of $\mathcal{T}$ that expands $\mathcal{I}|_S$ with $S$ the set of defined literals in $J_{P_{\mathcal{T}}}$.*

Failure to find a model of $\Delta_{\mathrm{g}}$ expanding $\mathcal{I}_{in}$ in which $P_{\mathcal{T}}$ is true implies the lack of models of $\mathcal{T}$ expanding $\mathcal{I}_{in}$. The former is the case when an inconsistency is detected in $\Delta_{\mathrm{g}}$ after initializing $\mathcal{I}$ with $P_{\mathcal{T}}$. If $\Delta_{\mathrm{g}}$ has no such models, then it has an unsatisfiable core, i.e., a set of rules from $\Delta_{\mathrm{g}}$ such that no model exists that expands $\mathcal{I}$. Hence, it is also an unsatisfiable core for $\mathcal{T} = (P_{\mathcal{T}}, \Delta)$. To find an unsatisfiable core, one can, for example, use techniques described in [Torlak et al., 2008].

The above formalization is for a theory $\{P_{\mathcal{T}}, \Delta\}$ possibly containing an inductive definition. In the simpler case that the original input theory $\mathcal{T}$ was a set of FO sentences $\{\phi_1, \ldots, \phi_n\}$, the above method boils down to maintaining a grounded theory $\mathcal{T}_g$ and a non-ground theory $\mathcal{T}_d$, both consisting of FO sentences. For the non-ground part, a justification manager can maintain direct justifications as sets of literals that make the sentences of $\mathcal{T}_g$ true. Major simplifications are that the justification manager never needs to find justifications that make a formula false and need not guard for cycles in the justification graph but only maintain that the direct justifications are consistent with one another and with $\mathcal{I}$.

**Example 7.2.6.** Consider the theory consisting of the sentences $P \Rightarrow Q$ and $Q \Rightarrow \forall x \in D : R(x)$. We can then keep both sentences in the non-grounded part by taking $\{\neg P\}$ as justification for the former sentence and $\{\neg Q\}$ for the latter. The union of both is consistent and hence would result in an acceptable state for the empty interpretation.

### 7.2.2 Practical Constructions for $\mathrm{FO}(ID)$ Theories

Roughly speaking, our lazy model expansion framework consists of two components. On the one hand, a standard model expansion algorithm that operates on $\{P_{\mathcal{T}}, \Delta_{\mathrm{g}}\}$ and, on the other hand, a justification manager that maintains a justification over $\Delta_{\mathrm{gd}}$ and lazily grounds $\Delta_{\mathrm{d}}$. Lazy model expansion performs search over the space of acceptable states and aims at reaching a state where Theorem 7.2.4 (or Corollary 7.2.5) is applicable. To avoid slowing down the search during model expansion, the work done by the justification manager and the lazy grounding must be limited. To achieve this, we have designed a system in which the justification manager has no access to the grounded definition $\Delta_{\mathrm{g}}$ and need not restore its state when the search algorithm backtracks over the current structure $\mathcal{I}$. The justification manager only has access to $\mathcal{I}$ and maintains justifications that are restricted to $\Delta_{\mathrm{d}}$. In particular, a literal defined in $\Delta_{\mathrm{g}}$ is not allowed in a direct justification. Our justification manager maintains the following properties:

- Literals in direct justifications are either open in $\Delta_{\mathrm{gd}}$ or defined in $\Delta_{\mathrm{d}}$.
- All direct justifications in $J$ are kept consistent with each other and with the current structure $\mathcal{I}$.
- The justification graph defined by $J$ has no positive or mixed cycles and is total.

To distinguish acceptable states that meet these additional requirements from acceptable states as defined in Definition 7.2.1, we call them *default acceptable states*; they can be defined as:

**Definition 7.2.7** (Default acceptable state)**.** A state $\langle \Delta_{\mathrm{g}}, \Delta_{\mathrm{d}}, J, \mathcal{I} \rangle$ is a default acceptable state if it is an acceptable state and, in addition, (**i**) literals in direct justifications are either open in $\Delta_{\mathrm{gd}}$ or defined in $\Delta_{\mathrm{d}}$, and (**ii**) $J$ justifies the set of all literals for which $J$ is defined.

It follows that default acceptable states satisfy two extra conditions: they do not justify literals defined in $\Delta_{\mathrm{d}}$ in terms of literals defined in $\Delta_{\mathrm{g}}$, and the set of all literals in $J$ is consistent. For an acceptable state it suffices that those in $J_L$ are consistent. Since default acceptable states are acceptable states, Theorem 7.2.4 and Corollary 7.2.5 also hold for default acceptable states.

During standard model expansion, the main state-changing operations are expanding $\mathcal{I}$ by making literals true, either through choice or propagation, and by backjumping. When $S = \langle \Delta_{\mathrm{g}}, \Delta_{\mathrm{d}}, J, \mathcal{I} \rangle$ is a default acceptable state and model expansion modifies $\mathcal{I}$ into $\mathcal{I}'$, the new state $\langle \Delta_g, \Delta_d, J, \mathcal{I}' \rangle$ is not necessarily a default acceptable state. The following propositions identify situations where acceptability is preserved.

**Proposition 7.2.8.** *Let $\langle \Delta_g, \Delta_d, J, \mathcal{I} \rangle$ be a default acceptable state, L a set of literals unknown in $\mathcal{I}$ and $\mathcal{I}'$ the consistent structure $\mathcal{I} \cup L$. If (**i**) literals of L either are not defined in $\Delta_d$ or have a direct justification in J and (**ii**) no direct justification in J contains the negation of a literal in L, then $\langle \Delta_g, \Delta_d, J, \mathcal{I}' \rangle$ is a default acceptable state.*

*Proof.* As the literals true in $\mathcal{I}$ and defined in $\Delta_d$ have a direct justification, it follows from (**i**) that all literals true in $\mathcal{I}'$ and defined in $\Delta_d$ have a direct justification. As all justifications in $J$ are consistent with $\mathcal{I}$, then, by (**ii**), they are also consistent with $\mathcal{I}'$. Hence, $J$ justifies all literals true in $\mathcal{I}'$ and defined in $\Delta_d$. □

**Proposition 7.2.9.** *Let $\langle \Delta_g, \Delta_d, J, \mathcal{I} \rangle$ be a default acceptable state. Then the state $\langle \Delta_g, \Delta_d, J, \mathcal{I}' \rangle$ with $\mathcal{I}' <_p \mathcal{I}$ is a default acceptable state.*

*Proof.* The justification $J$ justifies all literals defined in $\Delta$ and true in $\mathcal{I}$. As $\mathcal{I}'$ is a subset of $\mathcal{I}$, $J$ justifies all literals defined in $\Delta$ and true in $\mathcal{I}'$. □

The justification $J$ has to be revised when model expansion adds a literal $l$ to $\mathcal{I}$ and the conditions of Proposition 7.2.8 are not satisfied. The justification manager attempts to revise the direct justifications of literals where the negation of $l$ occurs in their direct justification. When $l$ is defined in $\Delta_d$ and has no direct justification, the justification manager attempts to find one. When the justification manager fails to revise $J$ such that an acceptable state is obtained, the rule defining the offending literal is grounded and moved to $\Delta_g$. If possible, the rule is first split up and only a small part is grounded. At that point, propagation can interrupt the justification manager and infer the truth of additional literals, or detect an inconsistency, at which point the model expansion algorithm performs backjumping. In both cases, the justification manager has to resume the revision of the justification until an acceptable state is reached. Once an acceptable state is reached, model expansion can select and assign a choice literal. These processes are described in more detail in the next section.

As described above, we do not allow literals defined in $\Delta_g$ in the direct justifications of literals defined in $\Delta_d$. The reason is that forbidden loops over both $\Delta_g$ and $\Delta_d$ can only be detected by also maintaining a justification in $\Delta_g$ (which we currently do not do). However, it is feasible to relax this condition by performing some static analysis over the rules of the definition. One case is when the body of a rule has no defined literals: literals defined by such a rule cannot be part of a cycle. A step further is to analyse the dependency graph: a literal defined in $\Delta_g$ is allowed in the direct justification of another literal when both literals do not belong to the same strongly connected component of the dependency graph. Indeed, in that case, they cannot be part of the same cycle.

### 7.2.3 An Example

Our example uses a theory $\mathcal{T}$ which states that a symmetric graph ($edge/2$) exists where at least one node is reachable (predicate $R/1$) from the root node ($root$). Both *edge* and *root* are not interpreted, the input structure $\mathcal{I}$ only interprets the domain $D$ as the set of domain elements $\{d_1, \ldots, d_n\}$.

The justification manager uses a *local* approach; initially, no literal has a direct justification. When the justification $J$ does not meet the requirements for an acceptable state, a revision is made locally: the manager tries to find a direct justification for the offending literal without modifying direct justifications of other literals.

$$\begin{array}{l} P_{\mathcal{T}} \\ \left\{\begin{array}{lll} P_{\mathcal{T}} & \leftarrow C_1 \wedge C_2 & (1) \\ C_1 & \leftarrow \exists x \in D : \neg root(x) \wedge R(x) & (2) \\ C_2 & \leftarrow \forall (x\, y) \in D^2 : edge(x,y) \Rightarrow edge(y,x) & (3) \\ \forall x \in D : root(x) & \leftarrow x = d_1 & (4) \\ \forall x \in D : R(x) & \leftarrow root(x) \vee \exists y \in D : edge(x,y) \wedge R(y) & (5) \end{array}\right\} \end{array}$$

The lazy MX algorithm, sketched above, proceeds as follows:

1. The initial default acceptable state is $\langle \Delta_g, \Delta_d, J, \mathcal{I} \rangle$ in which $\Delta_g$ and $\mathcal{I}$ are empty, $J$ is empty and $\Delta_d = \Delta$.
2. Propagation over $\{P_{\mathcal{T}}, \Delta_g\}$ sets $\mathcal{I}$ to $\{P_{\mathcal{T}}\}$. $P_{\mathcal{T}}$ is defined in $\Delta_d$ so the state becomes unacceptable as $P_{\mathcal{T}}$ is not justified in $J$. Setting the direct justification of $P_{\mathcal{T}}$ to $\{C_1, C_2\}$ does not help as the justification for $P_{\mathcal{T}}$ is not total in $\Delta_d$; instead rule (1) is moved to $\Delta_g$.
3. Unit propagation sets $\mathcal{I}$ to $\{P_{\mathcal{T}}, C_1, C_2\}$. Now $C_1$ and $C_2$ have to be justified. Consider first $C_2$ and rule 3. As *edge* is open, we can take a direct justification that sets all negative *edge* literals true (setting all positive *edge* literals true would be equally good). This justifies $C_2$ and avoids the grounding of the rule defining $C_2$.

4. Literal $C_1$ cannot be justified (with the local approach). However, as rule 2 is existentially quantified, one can avoid grounding the whole rule. One can perform a Tseitin transformation to isolate one instance and then only ground that instance. For the purpose of illustration, we make the worst choice possible and instantiate $x$ with $d_1$:

   $$\left\{ \begin{array}{lll} C_1 & \leftarrow (\neg root(d_1) \wedge R(d_1)) \vee T & (2a) \\ T & \leftarrow \exists x \in D \setminus \{d_1\} : \neg root(x) \wedge R(x) & (2b) \end{array} \right\}$$

   Rule 2a is moved to $\Delta_g$ and a default acceptable state is reached.

5. No further propagation is possible, so a choice has to be made. As $C_1$ is true, the body of rule 2a has to become true. Preferably not selecting a Tseitin (this would trigger more grounding), the first disjunct is selected by model expansion and propagation extends the structure with $root(d_1) = \mathbf{f}$ and $R(d_1) = \mathbf{t}$. The literal $root(d_1)$ is defined in $\Delta_d$ by rule 4 but cannot be justified with the local approach. To avoid fully grounding it, it is transformed in rule 4a that defines $root(d_1)$ and rule 4b that defines $root$ for the other domain elements:

   $$\left\{ \begin{array}{lll} root(d_1) & \leftarrow \top & (4a) \\ \forall x \in D \setminus \{d_1\} : root(x) & \leftarrow x = d_1 & (4b) \end{array} \right\}$$

   Rule 4a is moved to $\Delta_g$. Note that it has no defined literals in the body, hence it is safe to use $root(d_1)$ in direct justifications. As a consequence of extending $\Delta_g$, propagation detects an inconsistency. After backtracking to $\mathcal{I} = \{P_{\mathcal{T}}, C_1, C_2\}$, the subsequent propagation sets the structure $\mathcal{I}$ to $\{P_{\mathcal{T}}, C_1, C_2, root(d_1), T\}$. Still not in a default acceptable state ($T$ is not justifiable using the local approach), rule 2b is further transformed to split off another instance.

   $$\left\{ \begin{array}{lll} T & \leftarrow (\neg root(d_2) \wedge R(d_2)) \vee T_2 & (2ba) \\ T_2 & \leftarrow \exists x \in D \setminus \{d_1, d_2\} : \neg root(x) \wedge R(x) & (2bb) \end{array} \right\}$$

   Rule 2ba is moved to $\Delta_g$, while rule 2bb remains in $\Delta_d$.

6. Again, the search avoids the new Tseitin, selecting the first disjunct in rule 2ba which propagates $\neg root(d_2)$ and $R(d_2)$. The literal $\neg root(d_2)$ is defined in $\Delta_d$, but can be justified by $\top$ (the body of rule 4b is false for $x = d_2$). The literal $R(d_2)$ cannot be justified (as all *edge* literals are false in the current justification graph) and rule 5 is transformed to split off the $d_2$ instance. Actually, this instance in turn has a disjunctive body with a complex subformula, so to avoid grounding the subformula, we break it

up in two parts and introduce another Tseitin.

$$\left\{\begin{array}{lll} R(d_2) & \leftarrow root(d_2) \vee T_3 & (5aa) \\ T_3 & \leftarrow \exists y \in D : edge(d_2, y) \wedge R(y) & (5ab) \\ \forall x \in D \setminus \{d_2\} : R(x) & \leftarrow root(x) & \\ & \quad \vee \exists y \in D : edge(x, y) \wedge R(y) & (5b) \end{array}\right\}$$

Rule 5aa is moved to $\Delta_g$, the others remain in $\Delta_d$.

7. The structure $\mathcal{I}$ is now $\{P_{\mathcal{T}}, C_1, C_2, root(d_1), T, \neg root(d_2), R(d_2)\}$, hence propagation on rule 5aa in $\Delta_g$ extends it with $T_3$. However, $T_3$ cannot be justified and rule 5ab is transformed to avoid its full grounding. We split off the $d_1$ case

   $$\left\{\begin{array}{lll} T_3 & \leftarrow (edge(d_2, d_1) \wedge R(d_1)) \vee T_4 & (5aba) \\ T_4 & \leftarrow \exists y \in D \setminus \{d_1\} : edge(d_2, y) \wedge R(y) & (5abb) \end{array}\right\}$$

   Rule 5aba is moved to $\Delta_g$ while rule 5abb remains in $\Delta_d$.

8. Again search selects the first disjunct and propagates $T_3$, $edge(d_2, d_1)$ and $R(d_1)$. The literal $R(d_1)$ is defined in $\Delta_d$, but $root(d_1)$ is a direct justification for it. While $root(d_1)$ is defined in $\Delta_g$, it is safe to use it in a direct justification as discussed earlier. The literal $edge(d_2, d_1)$ is in conflict with the direct justification for $C_2$ (rule 3). To handle this conflict, we split off the affected instance ($x = d_2, y = d_1$). Literal $T_5$ inherits the unaffected part of the direct justification of $C_2$ (all negative literals over *edge* except $edge(d_2, d_1)$).

   $$\left\{\begin{array}{lll} C_2 & \leftarrow (edge(d_2, d_1) \Rightarrow edge(d_1, d_2)) \wedge T_5 & (3a) \\ T_5 & \leftarrow \forall (x\ y) \in D^2 \setminus \{(d_2, d_1)\} : edge(x, y) \Rightarrow edge(y, x) & (3b) \end{array}\right\}$$

   Rule 3a is moved to $\Delta_g$ while rule 3b remains in $\Delta_d$.

9. Propagation on rule 3a extends $\mathcal{I}$ with $edge(d_1, d_2)$ and $T_5$. The literal $edge(d_1, d_2)$ is in conflict with the direct justification for $T_5$ (rule 3b) and part of rule 3b has to be grounded. First, we split off the instance $\{x = d_1, y = d_2\}$ as follows.

   $$\left\{\begin{array}{lll} T_5 & \leftarrow (edge(d_1, d_2) \Rightarrow edge(d_2, d_1)) \wedge T_6 & (3ba) \\ T_6 & \leftarrow \forall (x\ y) \in D^2 \setminus \{(d_2, d_1), (d_1, d_2)\} : & \\ & \qquad edge(x, y) \Rightarrow edge(y, x) & (3bb) \end{array}\right\}$$

   Rule 3ba is moved to $\Delta_g$ while rule 3bb remains in $\Delta_d$ and inherits the remainder of the direct justification (all negative literals over *edge* except $edge(d_1, d_2)$ and $edge(d_2, d_1)$). Propagation on rule 3ba extends $\mathcal{I}$ with $T_6$; while defined in $\Delta_d$, it is justified.

By now, $\Delta_g$ consists of the rules 1, 2a, 4a, 2ba, 5aa, 5aba, 3a, and 3ba, and the residual definition $\Delta_d$ consists of the rules 4b, 2bb, 5b, 5abb, and 3bb. The current structure $\mathcal{I}$ is $\{P_{\mathcal{T}}, C_1, C_2, root(d_1), \neg root(d_2), edge(d_2, d_1), edge(d_1, d_2), R(d_1), R(d_2), T, T_3, T_5, T_6\}$ and is a model of $P_{\mathcal{T}} \cup \Delta_g$.

Of these literals, $\neg root(d_2)$, $R(d_1)$ and $T_6$ are defined in $\Delta_d$. Literal $\neg root(d_2)$, defined by rule 4b has $\{\top\}$ as direct justification. Literal $R(d_1)$, defined by rule 5b, has $\{root(d1)\}$ as direct justification. Literal $T_6$, defined by rule 3bb has as direct justification the set of all negative *edge* literals over $D$ except $edge(d_1, d_2)$ and $edge(d_2, d_1)$. To obtain a full model of the theory, $\mathcal{I}$ is extended with the literals of the above direct justifications. In this case, this assigns all open literals and the model can be completed by the well-founded model computation over $\Delta_{gd}$. Actually, this can be done without grounding the definition [Jansen et al., 2013].

## 7.3 Justification Management

In Section 7.2.2, we have instantiated our general framework, developed in Section 7.2.1, for a justification manager that only has access to $\Delta_d$. In the example of Section 7.2.3, the justification was constructed on demand, i.e., each time some literal needed a (different) direct justification, the body of its defining rule was analysed and a justification was extracted. If that failed, part of the rule was grounded. This was called the *local approach*. One can also imagine a *global approach*, where more rules of $\Delta_d$ are considered at once in an attempt to select direct justifications that minimize the grounding of the rules as a whole. Obviously, a global approach will be more time consuming, so should not be applied every time an adjustment of the justification is required. In this section, we describe both approaches.

In the presentation of the algorithms, we assume quantifiers range over a single variable, variable names are not reused in multiple quantifications and the operation nnf reduces a formula to its negation normal form.

### 7.3.1 The Local Approach

Algorithm 7 shows the top level of the lazy_mx model expansion algorithm. The theory passed to the algorithm consists of the fact $P_{\mathcal{T}}$ and the definition $\Delta$; $\mathcal{I}_{in}$ is the input structure to be expanded. Definitions $\Delta_d$ and $\Delta_g$ are initialized with $\Delta$ and the empty definition, respectively, and $\mathcal{I}$ is initialized with $\mathcal{I}_{in}$. The set of ground sentences $sent_g$ is initialized with the fact $P_{\mathcal{T}}$ and the initial

**Algorithm 7:** The lazy_mx lazy model expansion algorithm.

```
Function lazy_mx (atomic sentence $P_{\mathcal{T}}$, definition $\Delta$, structure $\mathcal{I}_{in}$)
    Output: either a model of $\Delta_{\text{g}}$ and $J$ or false
    $sent_{\text{g}}$ := $\{P_{\mathcal{T}}\}$; $\Delta_{\text{g}}$ := $\emptyset$; $\Delta_{\text{d}}$ := $\Delta$; $J$ := $\emptyset$; $\mathcal{I}$ := $\mathcal{I}_{in}$; $q_{ch}$ := $\emptyset$;
    while true do
        $L$ := propagate($sent_{\text{g}} \cup \Delta_{\text{g}}, \mathcal{I}$);
        $\mathcal{I}$ := $\mathcal{I} \cup L$;
        foreach $l \in L$ do $q_{ch}$:=check_literal($l$,$q_{ch}$);
        if $\mathcal{I}$ is inconsistent then
            $sent_{\text{g}}$ += learn nogood;
            if conflict at root level then return false;
            $\mathcal{I}$ := $\mathcal{I}$ at state of backjump point;
        else if $q_{ch}$ is not empty then
            $(l, q_{ch})$ := dequeue($q_{ch}$);
            lazy_ground($l$);
        else if $\mathcal{I}$ is a model of $sent_g \cup \Delta_g$ then
            return $\mathcal{I}$, $J$;
        else
            select choice literal $l$; $\mathcal{I}$ := $\mathcal{I} + l$;
            $q_{ch}$:=check_literal($l$,$q_{ch}$);
        end
    end
Function check_literal (literal $l$, literal queue $q_{ch}$)
    Data: global $\Delta_{\text{d}}$ and $J$    Output: updated queue
    if $l$ defined in $\Delta_d$ and $J(l) = undef$ then $q_{ch}$ := enqueue($l$,$q_{ch}$) ;
    foreach $l'$ such that $\neg l \in J(l')$ do $q_{ch}$ := enqueue($l'$,$q_{ch}$) ;
    return $q_{ch}$;
```

justification $J$ is empty. An auxiliary (FIFO) queue $q_{ch}$ is initialized as empty. This queue keeps track of literals for which the direct justification needs to be checked.

The main loop performs model expansion over $sent_{\text{g}} \cup \Delta_{\text{g}}$, interleaved with work by the justification manager towards establishing a default acceptable state. The model expansion part consists of propagation (the call to propagate), the test for inconsistency of the current state (with clause learning and backjumping), the test whether a model of $sent_{\text{g}} \cup \Delta_{\text{g}}$ is found (returning the model and the justification) and of a choice step that selects an unknown literal in $sent_{\text{g}} \cup \Delta_{\text{g}}$ and assigns it a value. The test for a model is only performed in a default acceptable state (i.e., when the queue $q_{ch}$ is empty) as this ensures the well-founded model computation can expand the current structure $\mathcal{I}$ —extended with the direct justifications of all literals— into a model of the whole theory. Also the choice step only takes place in a default acceptable state, to ensure that the search space is limited to the state space of default acceptable states. Literals that become assigned (by propagation or choice) need a direct justification if they are defined in $\Delta_\mathrm{d}$, hence they are queued by check_literal if they do not have one. Also, assigned literals of which the negation is used in a direct justification need to be queued, as this direct justification is now invalid. When no propagation is possible and the current structure is consistent, a literal is removed from the queue and its direct justification is updated by the lazy_ground function. Literals are processed one at a time as lazy_ground may extend $\Delta_\mathrm{g}$, which may allow for extra propagation. As propagation is a fast and powerful operation, it is given priority over updating the justifications. Second, propagation might result in conflict and backtracking, after which some justifications might not need an update anymore, as discussed below.

### Lazy Grounding of One Rule

The function lazy_ground, Algorithm 8, checks the direct justification of its input literal. As the lazy model expansion algorithm may have backtracked since the literal was queued, it may well be that the direct justification of the literal is again acceptable as it is. Otherwise, namely when the literal is true in $\mathcal{I}$, defined in $\Delta_\mathrm{d}$ and has no direct justification, the function justify, Algorithm 9, is called to construct a new direct justification. If a direct justification already existed, justify first checks whether it is still valid. If justify did not find a valid direct justification, part of the rule has to be grounded, which is done through the call to split_and_ground. The old justification, if any, is passed as argument to split_and_ground, as we show that it can help to guide the grounding process.

Before going into more details, we first analyse which properties we want to maintain in the justification. The direct justifications of literals in the $q_{ch}$ queue might be invalid, hence they should not be considered as part of the current justification. The global invariants we maintain are:

- neither positive nor mixed cycles (negative ones are allowed),
- literals in the justification are consistent with each other.

For direct justifications of literals not on the queue, we maintain that

- they do not contain literals defined in $\Delta_\mathrm{g}$ (unless such a literal is safe, i.e., it can never be part of a cycle),

**Algorithm 8:** The lazy grounding of a literal.

```
Function lazy_ground (literal $l$)
    Data: global $\Delta_d$, $J$ and $\mathcal{I}$
    $success := true$;
    $j_{old} := J(l)$;
    if $l \in \mathcal{I}$ and $l$ defined in $\Delta_d$ and $J(l) = undef$ then
        $success :=$ justify($l$);
    else if $J(l) \neq undef$ then
        $success :=$ justify($l$);
    end
    if $success = false$ then
        $J := J[l \to false]$;
        split_and_ground($l, j_{old}$);
    end
```

- the literals they contain that are defined in $\Delta_d$ either have a direct justification or are on the queue themselves.

These invariants imply that a default acceptable state is reached when the $q_{ch}$ queue is empty. Indeed, it follows from the invariants that the current justification is total in that situation and hence all literals that have a direct justification are justified (Definition 7.1.6). Due to the policy followed to queue literals, the current justification is also consistent with $\mathcal{I}$ while all literals true in $\mathcal{I}$ and defined in $\Delta_d$ have a justification, hence $\langle \Delta_g, \Delta_d, J, \mathcal{I} \rangle$ is a default acceptable state.

**Algorithm 9:** The justify algorithm.

```
Function justify (literal $l$)
    Data: global $\Delta_d$ and $J$    Result: update to $J$
    Output: $true$ if a justification was found, $false$ otherwise
    if $J(l)$ exists and obeys the invariants then return $true$ ;
    $\varphi :=$ body of the rule defining $l$;
    if $l$ is a negative literal then $\varphi :=$ nnf$(\neg\varphi)$ ;
    $dj :=$ build_djust($l, \varphi,$ init_just($l$));
    if $dj = false$ then return $false$ ;
    else $J := J[l \to dj]$; return $true$ ;
```

The function justify, described in Algorithm 9 starts by checking whether the direct justification of the literal obeys the invariants on direct justifications as well as the global invariants on the justification graph extended with this direct justification; if so, nothing needs to be done. To keep these checks tractable, they are done in an approximate way on the symbolic level. If the invariants are not satisfied, build_djust is called, a function that attempts to find a valid direct justification. If found, the justification is updated and *true* is returned; if not, *false* is returned.

### Building a Direct Justification

The purpose of build_djust, Algorithm 10, is to find a direct justification for literal $l$ that obeys the above-mentioned invariants. It is a recursive function which takes three parameters: (**i**) the literal $l$, (**ii**) the formula to be made true by the direct justification (initially the whole body of the rule defining the literal; note that the initialization takes the negation of the rule when the literal is negative), (**iii**) a description of the direct justification derived so far, initialized through init_just$(l)$.

Before going into detail, we discuss how to represent direct justifications. Basically, we could represent a direct justification as a set of ground literals. However, this set can be quite large and using a ground representation might hence defy the purpose of lazy grounding. Instead, we represent a direct justification as a pair $\langle L, B\rangle$ with $L$ a set of possibly non-ground literals and $B$ a set of bindings $x_i \in D_i$ with $x_i$ a variable and $D_i$ a domain. A set of bindings $\{x_1 \in D_1, \ldots, x_n \in D_n\}$ represents the set of variable substitutions $S_B = \{\{x_1/d_1, \ldots, x_n/d_n\} \mid d_i \in D_i \text{ for each } i \in [1, n]\}$. The set of ground literals represented by $\langle L, B\rangle$ is then $\{l\theta \mid l \in L \text{ and } \theta \in S_B\}$. The direct justification of a literal $P(\overline{d})$, defined by a rule $\forall \overline{x} \in \overline{D} : P(\overline{x}) \leftarrow \varphi$, is initialized by init_just$(l)$ as $\langle \emptyset, \{x_1 \in \{d_1\}, \ldots, x_n \in \{d_n\}\}\rangle$. In effect, $B$ selects the appropriate variable instantiation from the domains to identify the relevant rule instantiation, while the set of literals is empty.

The build_djust algorithm searches for a set of literals making $\varphi$ true. It works by recursively calling itself on subformulas of $\varphi$ and composing the results afterwards into a larger justification for $\varphi$.

The base case is when the formula is a literal. To make that literal true, all instances of the literal under binding $B$ must be true, hence, the set of literals $L$ is extended with the literal itself. The resulting direct justification has to satisfy all invariants, which is checked by the call to valid: it returns *true* for a call valid$(l, dj)$ if $dj$ satisfies the invariants to be (part of) a direct justification for $l$ and $J[l \rightarrow dj]$ satisfies the invariants on the justification.

A universally quantified formula $\forall x \in D : \psi$ has to be true for each instance of the quantified variable. Hence, in the recursive call, the binding $B$ is extended

**Algorithm 10:** The build_djust algorithm.

```
Function build_djust ($literal\ l, formula\ \varphi\ and\ justification\ \langle L, B\rangle$)
    Input: $B$ binds all free variables of $\varphi$
    Output: either a direct justification or false
    switch $\varphi$ do
        case $\varphi$ is a literal
            if $valid(l, \langle L \cup \{\varphi\}, B\rangle)$ then return $\langle L \cup \{\varphi\}, B\rangle$;
            else return false;
        case $\forall x \in D : \psi$
            return $build\_djust(l, \psi, \langle L, B \cup \{x \in D\}\rangle)$;
        case $\exists x \in D : \psi$
            if $D$ is large then
                return $build\_djust(l, \psi, \langle L, B \cup \{x \in D\}\rangle)$;
            else
                foreach $d_i \in D$ do
                    $\langle L', B'\rangle$ := build_djust$(l, \psi, \langle L, B \cup \{x \in \{d_i\}\}\rangle)$;
                    if $\langle L', B'\rangle \neq$ false then return $\langle L', B'\rangle$;
                end
                return false;
            end
        case $\varphi_1 \wedge \ldots \wedge \varphi_n$
            foreach $i \in [1, n]$ do
                $\langle L', B'\rangle$ := build_djust$(l, \varphi_i, \langle L, B\rangle)$;
                if $\langle L', B'\rangle =$ false then return false;
                else $\langle L, B\rangle := \langle L', B'\rangle$;
            end
            return $\langle L, B\rangle$;
        case $\varphi_1 \vee \ldots \vee \varphi_n$
            foreach $i \in [1, n]$ do
                $\langle L', B'\rangle$ := build_djust$(l, \varphi_i, \langle L, B\rangle)$;
                if $\langle L', B'\rangle \neq$ false then return $\langle L', B'\rangle$;
            end
            return false;
        end
    endsw
```

with $x \in D$. For an existentially quantified formula, it suffices that one instance is true. Hence, a minimal approach is to try each instance separately until one succeeds; if all fail, *false* is returned. However, we do not want to iterate over each domain element if $D$ is large, which would be similar to constructing the grounding itself. Instead, if $D$ is large, we extend the binding with $x \in D$.

Conjunction is similar to universal quantification, except that explicit iteration over each conjunct is needed. As soon as one conjunct fails, the whole conjunction fails. Disjunction is similar to existential quantification.

Note that the order in which build_djust iterates over subformulas and instantiations has an effect on the justification that is found.

**Example 7.3.1.** Consider the following rule over a large domain $D$.

$$H \leftarrow \forall x \in D : \neg P(x) \vee (\exists y \in D : Q(x,y) \wedge \neg R(x,y))$$

Assume that $J$ is empty and we have no loops to keep track of. Applying build_djust to $H$ then results in either the justification $\{\neg P(x) \mid x \in D\}$ or the justification $\{Q(x,y) \mid x \in D, y \in D\} \cup \{\neg R(x,y) \mid x \in D, y \in D\}$, depending on the order of iteration.

### Partially Grounding a Rule

The last bit of the lazy model expansion algorithm handles the case where no (new) justification could be found for a defined literal $l$. A straightforward solution would be to call one_step_ground on the associated rule, add the resulting ground rule(s) to $\Delta_g$ and add rules over newly introduced (Tseitin) symbols to $\Delta_d$. Indeed, these new symbols are unknown in $\mathcal{I}$, so can be kept safely in $\Delta_d$ for now, even without a direct justification. However, in many cases such an operation results in too much grounding.

**Example 7.3.2.** Consider a rule $r_1$ of the form $\forall x \in D : P(x) \leftarrow \varphi$ in a situation where no justification can be found for atom $P(d)$. Straightforwardly applying one_step_ground to $r_1$ would instantiate $x$ with all elements in $D$, resulting in $|D|$ rules, while in fact it would suffice to split $r_1$ in two rules, one for the instance $x = d$ and one for the remainder. Another example applies to a rule $r_2$ of the form $H \leftarrow \forall x \in D : Q(x) \vee R(x)$ and a direct justification $J[H] = \langle Q(x), \{x \in D\} \rangle$. When $Q(d)$ becomes false (assume $R$ is not a valid alternative justification), one_step_ground instantiates $x$ with all elements in $D$. Instead, it is be better to split off the instantiation $x = d$ and introduce a Tseitin for the remainder. For the latter, removing $Q(d)$ from the original justification results in a valid direct justification.

The split_and_ground algorithm (Algorithm 11) has to ground part of the rule defining a given literal $l$, say $P(\overline{d})$. The first step is to split off the rule instance for which the rule defining $l$ has to be grounded (the call to split). Let $\forall \overline{x} \in \overline{D} : P(\overline{x}) \leftarrow \varphi$ be the rule in $\Delta_\mathrm{d}$ that defines $P(\overline{d})$. We then replace that rule by $\forall \overline{x} \in \overline{D} - \overline{d} : P(\overline{x}) \leftarrow \varphi$ in $\Delta_\mathrm{d}$ and additionally return the rule $P(\overline{d}) \leftarrow \varphi[\overline{x}/\overline{d}]$. Afterwards, we apply one_step_ground and add the rules to their corresponding definitions.[1]

**Algorithm 11:** The split_and_ground algorithm.

```
Function split_and_ground (literal $l$, justification $j_{old}$)
    Input: $l$ is defined in $\Delta_\mathrm{d}$, $j_{old}$ is the previous justification (if any)
    Result: update to $\Delta_\mathrm{g}$, $\Delta_\mathrm{d}$, $J$, and $q_{ch}$
    $r$ := split($l$,$j_{old}$);
    $(\Delta'_\mathrm{g}, \Delta'_\mathrm{d})$ := one_step_ground($r$);
    $\Delta_\mathrm{g} \cup = \Delta_\mathrm{g}'$; $\Delta_\mathrm{d} \cup = \Delta_\mathrm{d}'$;
```

The result of split_and_ground is that definition $\Delta_\mathrm{gd}$ that is "more" ground than the previous one. The limit is a ground definition $\Delta_\mathrm{gd}$ in which $\Delta_\mathrm{d}$ is empty and $\Delta_\mathrm{g}$ is strongly $voc(\Delta)$-equivalent with $\Delta$.

Even if no justification was found, we can do better than just splitting off $l$ and applying one_step_ground, as shown in Example 7.3.2. First, splitting can be made significantly more intelligent, which is discussed in Section 11. Second, we can improve one_step_ground to only ground part of expressions if possible, which we described below.

**Improving one_step_ground.** Applying one_step_ground to a rule $l \leftarrow \varphi$ iterates over all subformulas/instantiations of $\varphi$. For example if $\varphi$ is the sentence $\exists x \in D : P(x)$, the result are $|D|$ new rules and as many new Tseitin symbols. Instead, depending on the value of $l$, we can suffice with only introducing one (or some) of these subformulas, as shown in Algorithm 12, which extends the switch statement of one_step_ground with two higher-priority cases. If $l$ is true, we can suffice by grounding one disjunct/existential instantiation and delay the rest by Tseitin introduction. If $l$ is false, we take a similar approach for conjunction/universal quantification.

---

[1] Recall, rules returned by one_step_ground always have a unique head. Those in $\Delta_\mathrm{d}'$ even a unique new Tseitin as head.

**Algorithm 12:** Additional cases for the one_step_ground algorithm.

```
switch $r$ do
    case $l \leftarrow \varphi_1 \lor \ldots \lor \varphi_n$ and $\mathcal{I}(l) \neq \mathbf{f}$
        choose $i \in [1, n]$;
        $\Delta_\mathrm{d}$ += $T \leftarrow \bigvee_{j \in \{1 \ldots n\} - i} \varphi_j$ ;
        $r := L \leftarrow \varphi_i \lor T$;
    case $l \leftarrow \exists x \in D : \varphi$ and $\mathcal{I}(l) \neq \mathbf{f}$
        choose $d \in D$;
        $\Delta_\mathrm{d}$ += $T \leftarrow \exists x \in D \backslash d : \varphi$;
        $r := l \leftarrow \varphi[x/d] \lor T$;
    analogous cases for $\land$ and $\forall$ and $\mathcal{I}(l) \neq \mathbf{t}$
endsw
```

### Algorithmic Properties

Correctness and termination of the presented algorithms is discussed in the following theorem.

**Theorem 7.3.3** (Correctness and termination)**.** *If algorithm lazy_mx returns an interpretation $\mathcal{I}$, then expanding $\mathcal{I}$ with $J$, applying the well-founded evaluation over $\Delta_{gd}$ and restricting it to $voc(\mathcal{T})$ results in a model of $\mathcal{T}$. If the algorithm returns false, no interpretation exists which is more precise than $\mathcal{I}_{in}$ and satisfies $\mathcal{T}$.*

*Algorithm lazy_mx terminates if $\mathcal{I}$ is finite. Otherwise, termination is possible but not guaranteed.*[2]

*Proof.* If lazy_mx returns an interpretation $\mathcal{I}$, $\mathcal{I}$ is a model of $\Delta_\mathrm{g}$ and $q_{ch}$ is empty. Given the properties of split_and_ground, after applying lazy_ground for a literal $l$, no more constructions are violated by $l$ and any rule defining $l$ has a construction valid in $\mathcal{I}$ or is part of $\Delta_\mathrm{g}$. Hence if $q_{ch}$ is empty, we are in a default acceptable state. In that case, the model construction is correct as shown in Theorem 7.2.4. If lazy_mx returns *false*, it has been proven that $\Delta_\mathrm{g}$ has no models in $\mathcal{I}_{in}$. In that case, there can also be no models of $\Delta_\mathrm{gd}$ and hence $\mathcal{T}$ also has no models expanding $\mathcal{I}_{in}$.

Without calls to lazy_ground, the search algorithm terminates for any finite $\Delta_\mathrm{g}$. lazy_ground itself produces an ever-increasing ground theory $\Delta_\mathrm{g}$ with the full grounding as limit. Hence, lazy_mx always terminates if $\mathcal{I}_{in}$ is finite. If $\mathcal{I}_{in}$

---

[2] It is possible to change the integration of lazy_ground in lazy_mx to guarantee termination if a finite model exists, see Section 7.4.2.

is infinite, the limit of $\Delta_g$ is an infinite grounding, so termination cannot be guaranteed. □

### Symbolic Justifications, Incremental Querying and Splitting

The algorithms presented above are sound and complete. There are however two important observations that create room for improvements. In both cases, it revolves around not treating justifications as just a set of literals but taking the formula from which they were derived into account.

**Incremental Querying.** First, we observe that if multiple justifications exist for (subformulas of) a formula $\varphi$, we can delay grounding even more.

**Example 7.3.4.** Consider the formula $\forall x \in D : P(x) \lor Q(x)$, where both $\langle\{P(x)\}, \{x \in D\}\rangle$ and $\langle\{Q(x)\}, \{x \in D\}\rangle$ are justifications. From that, we could derive the justification: for each $d \in D$, make either $P(d)$ *or* $Q(d)$ true. Hence, *only* when for one $d$, both $P(d)$ and $Q(d)$ become false is more grounding necessary.

We can do this automatically by making the following changes to build_djust.

- The algorithm is allowed to select multiple disjunctions / existential quantifications even if a valid justification was already found for one (Lines 14 and 28).
- build_djust does not build a justification, but builds a *justification formula* from the subformulas/instantiations it selects. For example, **return** build_djust$(l, \psi, \langle L, B \cup \{x \in D\}\rangle)$ in the universal quantification case is changed to **return** $\forall x \in D :$ build_djust$(l, \psi, \langle L, B \cup \{x \in D\}\rangle)$.
- The validity check (valid) is extended to only return true if the justification formula is not false.

It is straightforward to see that a justification formula $\psi$ of a formula $\varphi$ entails $\varphi$. From $\psi$, the justification itself can be derived directly as the set of non-false literals in the (full) grounding of $\psi$.

Naturally, by allowing more complex formulas (instead of a conjunction of universally quantified literals), it becomes more expensive to track whether a formula has become false after changes to $\mathcal{I}$. This is in fact the *incremental query* problem. In the experiments, we limit the depth of the allowed formulas and use a straightforward (expensive) incremental query algorithm to track whether justification formulas have become false.

**Body Splitting.** Second, if a literal $l$ violates the justification $j$ of a formula $\varphi$, it is often possible to split $j$ and $\varphi$ each in two parts: a small part to which the violation really applies and a large remainder for which we can directly reuse (a reduced version of) $j$.

**Example 7.3.5.** Consider the rule $h \leftarrow \forall x \in D : P(x)$ with justification $\langle\{P(x)\}, \{x \in D\}\rangle$ for $h$ true in $\mathcal{I}$. When $P(d)$ becomes true, it is easy to see that we can split off the "violating instantiation" by rewriting the original rule into $h \leftarrow P(d) \land T$ and adding the rule $T \leftarrow x \in D \backslash d : P(x)$. Crucially, a justification for the latter can be derived from the original justification, namely $\langle\{P(x)\}, \{x \in D \backslash d\}\rangle$. The latter rule can hence be added to $\Delta_\mathrm{d}$ and its justification to $J$ and we only are only left with the former rule.

This is the responsibility of the split operation, which takes as input a literal $l$ defined in $\Delta_\mathrm{d}$ by a rule $r = h \leftarrow \varphi$ and its old justification formula $j_{old}$. Next to splitting of the rule instance based on $l$ (as described earlier), its task is to identify subformulas of $\varphi$ for which $j_{old} \backslash l$ is a justification formula and apply Tseitin transformation on them.

**Proposition 7.3.6.** *Given a literal $l$ in a justification $j$ of a sentence $\varphi$. If the full grounding of a subformula $\psi$ of $\varphi$ does not contain $\neg l$, then $j \backslash l$ is a justification of $\psi$.*

Assume $v$ is the domain literal that violated a justification and resulting in the current call to lazy grounding ($v$ can be computed from the state or, better still, can be passed down from lazy_mx). Assume we know all variable instantiations $\{\overline{x} = \overline{d}_1, \ldots, \overline{x} = \overline{d}_n\}$ under which $\varphi$ contains occurrences of a literal $\neg v$. Then the following algorithm can be used to efficiently split $\varphi$ for $v$. For each variable instantiation, say $\overline{x} = \overline{d}_i$, visit $r$ recursively and depth-first. Whenever a quantification $\forall x \in D : \varphi$ is encountered with $x$ equal to $x_j \in \overline{x}$, replace it by $(\forall x \in D - d_j : \varphi) \land \varphi[x = d_j]$. Tseitin transformation is applied to the left-hand conjunct. This results in a new rule $r_v$ for which $j_{old} \backslash v$ is a justification. Afterwards, the right-hand conjunct is visited recursively. The result is a set of new rules for which no new justification has to be sought and a smaller rule $r'$ which are then passed to one_step_ground. Correctness follows from the fact the $j_{old} \backslash v$ is a valid justification, none of the new rules contains $v$, and from the correctness of Tseitin transformation.

**Example 7.3.7.** In Example 7.3.1, justifications were sought for $H$ in the rule

$$H \leftarrow \forall x \in D : \neg P(x) \lor (\exists y \in D : Q(x,y) \land \neg R(x,y)).$$

Assume we selected the justification $j = \{Q(x,y) \mid x \in D, y \in D\} \cup \{\neg R(x,y) \mid x \in D, y \in D\}$. When $l = Q(d_1, d_2)$ becomes false, $j$ is no longer consistent with $\mathcal{I}$; $j \backslash l$, however, is still consistent with $\mathcal{I}$, but is not a justification of

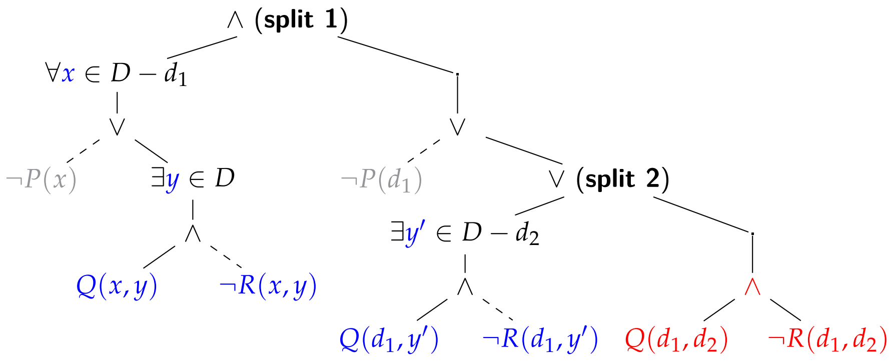


Figure 7.2: The rule $\forall x \in D : \neg P(x) \vee \exists y \in D : Q(x,y) \wedge \neg R(x,y)$ is split for a violating literal $Q(d_1,d_2)$. The original justification without $Q(d_1,d_2)$ is a justification of the left-hand side of all splits, with the justification formula shown in blue. The remaining non-justified formula is shown in red.

the whole body. On the other hand, $j \backslash l$ is a justification for the subformula $\neg P(x) \vee \exists y \in D : Q(x,y) \wedge \neg R(x,y)$ for each instantiation of $x$ different from $d_1$. Consequently, we can split the quantification $\forall x \in D :$ into $x \in D - d_1$ and $x = d_1$ and apply Tseitin transformation to the former. Afterwards, we recursively visit the latter formula and apply a similar reasoning to the existential quantification. The operations on the formula are illustrated in Figure 7.2. The result are the following rules, where the rule for $H$ is now even ground.

$$\left\{ \begin{array}{l} H \leftarrow T_1 \wedge (\neg P(d_1) \vee (T_2 \vee (Q(d_1,d_2) \wedge \neg R(d_1,d_2)))) \\ T_1 \leftarrow \forall x \in D \backslash \{d_1\} : \neg P(x) \vee \exists y \in D : Q(x,y) \wedge \neg R(x,y) \\ T_2 \leftarrow \exists y \in D \backslash \{d_2\} : Q(d_1,y) \wedge \neg R(d_1,y) \end{array} \right\}$$

Obtaining all variable instantiations, required to apply split, can be done during an initial visit of $\varphi$ by verifying all possible ways to obtain $l$. However, this information can be directly extracted from build_djust: for each selected literal occurrence $l$, we keep track of the path taken through the parse tree of $\varphi$: which subformulas/instantiations were selected to get to $l$.

**Example 7.3.8.** Assume the rule $C_1 \leftarrow \forall (x\ y) \in D^2 : \neg edge(x,y) \vee edge(y,x)$ has to be constructed true, $J$ is empty and $\mathcal{I}$ does not interpret *edge*. If build_djust recursively visits the body of the rule until $\neg edge(x,y)$, $\neg edge(x,y)$ is returned as it is a valid literal to use. Going up one level, we store that for $\neg edge(x,y) \vee edge(y,x))$, we selected $\{\neg edge(x,y)\}$. Assuming no more disjuncts are selected, $\neg edge(x, y)$ is returned again. Going back up through both quantifications, we store that, for both quantifications, we selected $D$ as the set of relevant domain elements, and build_djust returns the justification formula $\forall (x\ y) \in D^2 : \neg edge(x, y)$.

### 7.3.2 The Global Approach

Finding justifications using the greedy local approach can easily lead to more grounding than necessary. Consider for example the sentences $\forall x \in D : P(x)$ and $\forall x \in D : P(x) \Rightarrow Q(x)$. Applying the local approach to the second sentence first (with an empty $J$), might result in the construction which makes atoms over $P$ false. Applying the local approach to the first sentence then finds no valid justification, requiring to fully ground it. The global approach takes a *set* of rules as input and tries to select justifications for them such that the *expected* grounding size of the whole set is minimal.

We cast the task, called the *optimal justification* problem, as a problem on a graph as follows. The graph consists of two types of nodes $R$ ("rule") and $J$ ("justification"). Each rule node corresponds to a rule in $\Delta_{\mathrm{d}}$ and a truth value (true, false or unknown) which indicates whether the head is true or false or whether it is kept in $\Delta_{\mathrm{d}}$ but not be assigned a justification. Each justification node corresponds to a (symbolic) set of literals. Next, there are three types of edges $V$ ("valid"), $C$ ("conflict") and $D$ ("depends on"). There is a valid edge from a justification node to true or false rule nodes that are directly justified by it. There is a conflict edge between rule nodes of the same rule and between justification nodes that contain opposite literals. There is also a conflict edge between a true (false) rule node that defines $l$ and a justification node that contains $\neg l$ ($l$) and between an unknown rule node that defines $l$ and a justification node that contains either $l$ or $\neg l$. There is a depends edge from a true (false) rule node to a justification node if the justification contains negative (positive) literals defined by the rule. The idea is then to select rule and justification nodes such that

- Each selected true or false rule node is connected with a $V$ edge to at least one selected justification (justified). Each selected justification is connected through a $V$ edge to a selected true or false rule node.
- Selected nodes are not connected by a $C$ edge (no conflicts).
- There are neither positive nor mixed cycles in the subgraph consisting of the $D$ and $V$ edges connecting selected nodes (the selected justifications justify the selected rules).

From a selection satisfying these constraints, an initial $J$ can be built as follows. $J[(\neg)l \rightarrow j]$ if $l$ is defined in $r$, its true (false) rule node was selected and $j$ is the union of the justifications of its associated selected justification nodes. All literals defined by rules for which no rule node was selected are added to the initial $q_{ch}$ queue, to be handled by the local approach.

This type of problem is somewhat related to the NP-hard *hitting set* (or *set cover*) problem [Karp, 1972]: given a set of "top" and "bottom" nodes and edges between them, the task is to find a minimal set of bottom nodes such that each top node has an edge to at least one selected bottom node.

Given a default acceptable state $\langle \Delta_g, \Delta_d, J, \mathcal{I} \rangle$, the input for the optimal justification problem is generated as follows. For any rule in $\Delta_d$, a node is constructed for each of the three truth values (only one when the truth value is known in $\mathcal{I}$) and also their conflict edges are added. Justification nodes and their $V$ edges are obtained using a (straightforward) adaptation of `build_djust` which returns a set of possible justifications that make the head of a rule true (false). E.g., for a rule $\forall \overline{x} \in \overline{D} : P(x) \leftarrow \varphi$, `build_djust` is called with the literal $P(x)$ and the binding is initialized at $\{\overline{x} \in \overline{D}\}$. Conflict and depends-on edges are derived by checking dependencies between justifications and between rules and justifications. To keep this efficient, it is done on the symbolic level.

**Example 7.3.9.** Consider the theory of our running example, after $P_{\mathcal{T}}$, $C_1$ and $C_2$ have been propagated to be true. Definition $\Delta_d$ is then

$$\left\{ \begin{array}{lll} C_1 & \leftarrow \exists x \in D : \neg root(x) \land R(x) & (2) \\ C_2 & \leftarrow \forall (x\, y) \in D^2 : edge(x,y) \Rightarrow edge(y,x) & (3) \\ \forall x \in D : root(x) & \leftarrow x = d_1 & (4) \\ \forall x \in D : R(x) & \leftarrow root(x) \lor \exists y \in D : edge(x,y) \land R(y) & (5) \end{array} \right\}$$

The associated optimal construction set input is shown in Figure 7.3. Literal $C_1$ and $C_2$ are true in $\mathcal{I}$, hence there is only one rule node for rules (2) and (3). Neither $root(x)$ nor $\neg root(x)$ can be justified for all $x \in D$, hence they have been omitted.

There are four solutions that are subset-maximal with respect to rule nodes, namely the following rule node selections:

$$\{\langle R, \mathbf{u}\rangle, \langle root, \mathbf{u}\rangle, \langle C_2, \mathbf{t}\rangle\} \tag{a}$$

$$\{\langle R, \mathbf{f}\rangle, \langle C_2, \mathbf{t}\rangle\} \tag{b}$$

$$\{\langle R, \mathbf{t}\rangle, \langle C_2, \mathbf{t}\rangle\} \tag{c}$$

$$\{\langle C_1, \mathbf{t}\rangle, \langle C_2, \mathbf{t}\rangle\} \tag{d}$$

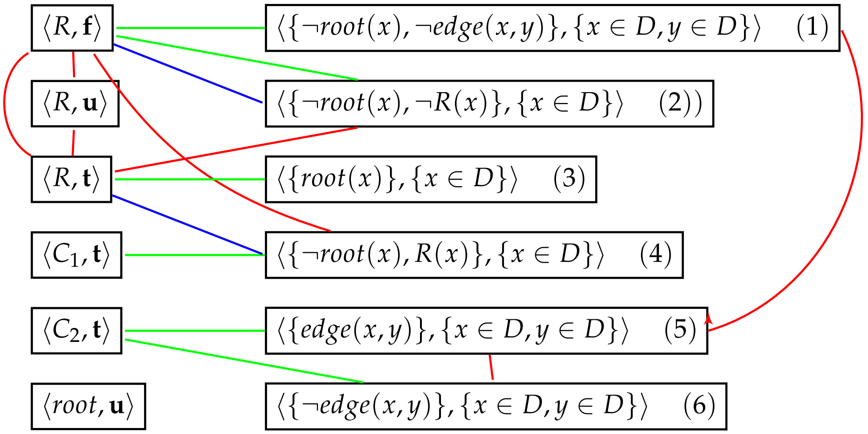

$\langle R, \mathbf{f} \rangle$
$\langle R, \mathbf{u} \rangle$
$\langle R, \mathbf{t} \rangle$
$\langle C_1, \mathbf{t} \rangle$
$\langle C_2, \mathbf{t} \rangle$
$\langle root, \mathbf{u} \rangle$
$\langle \{\neg root(x), \neg edge(x,y)\}, \{x \in D, y \in D\} \rangle$ (1)
$\langle \{\neg root(x), \neg R(x)\}, \{x \in D\} \rangle$ (2))
$\langle \{root(x)\}, \{x \in D\} \rangle$ (3)
$\langle \{\neg root(x), R(x)\}, \{x \in D\} \rangle$ (4)
$\langle \{edge(x,y)\}, \{x \in D, y \in D\} \rangle$ (5)
$\langle \{\neg edge(x,y)\}, \{x \in D, y \in D\} \rangle$ (6)

Figure 7.3: The graph that is part of the input to the optimal justification problem of Example 7.3.9. Rule nodes are shown on the left, justification nodes on the right; valid edges are shown in green, conflict edges in red and depends-on edges in blue. For readability, conflicts between justifications and unknown rule nodes are not shown.

For each of these, multiple justification selections are possible (also shown in Figure 7.3). For $C_1$, we have to select justification 4, but for $C_2$ we can choose from justification 5 or 6 (but not both).

The objective is then not to maximize the number of selected rule nodes, but to minimize the expected grounding size. To obtain an estimate of the expected grounding size, the following conditions were taken into account:

- It should depend on the size of the grounding of the rule.
- Assigning multiple justifications to a rule should result in a lower estimate as the rule will only be grounded when all are false.
- Shared variables result in less matching instantiations.
- In most practical applications, the number of false atoms far exceeds the number of true ones in any model. Hence, positive literals in a justification should have a higher cost than negative ones.

We approximate the expected grounding size with the function $exp_{size}$ which takes as input a rule $r$, the selected type of justification (rule not selected ($\mathbf{n}$), no justification (**u**), making the rule true (**t**) or false (**f**)) and the set of justifications $J$. It is then defined as

$$\begin{aligned}
exp_{size}(r, \mathbf{n}, \emptyset) &= size(r) \\
exp_{size}(r, \mathbf{u}, \emptyset) &= size(r) \times 0.1 \\
exp_{size}(r, \mathbf{f}, J) &= size(r) \times \prod_{J_i \in J} (0.3 \times |\text{pos. lits.} \in J_i| \\
&\qquad\qquad + 0.1 \times |\text{neg. lits.} \in J_i|) \\
exp_{size}(r, \mathbf{t}, J) &= size(r) \times 0.3 \times \prod_{J_i \in J} (0.3 \times |\text{pos. lits.} \in J_i| \\
&\qquad\qquad + 0.1 \times |\text{neg. lits.} \in J_i|)
\end{aligned}$$

Below, we define the function *size*, an approximation of final grounding size of a rule, which takes the local approach into account. For an existential quantification and disjunction, we take the logarithm to reflect that it generally results in less grounding than universal quantification or conjunction in the same context, as Tseitin introduction can be used to ground it only partially.

$$\begin{aligned}
size(L) &= 1 \\
size(\exists x \in D : \varphi) &= log(D) \times size(\varphi) \\
size(\forall x \in D : \varphi) &= D \times size(\varphi) \\
size(\phi_1 \wedge \ldots \wedge \varphi_n) &= \sum_{i \in [1,n]} size(\varphi_i) \\
size(\phi_1 \vee \ldots \vee \varphi_n) &= log(n) \times \frac{\sum_{i \in [1,n]} size(\varphi_i)}{n} \\
size(L \leftarrow \varphi) &= size(\varphi) + 1
\end{aligned}$$

Solutions to the optimal justification problem should minimize the term $\sum_{r \in \Delta_d} exp_{size}(r, t(r), J(r))$ with $t(r)$ the type of justification and $J(r)$ the constructions assigned to $r$.

**Example 7.3.10** (Continued from Example 7.3.9)**.** The size of rule $C_1$ is $1 + log(D) \times 2$, that of $C_2$ is $1 + D^2 \times log(2)$ and that of $R$ is $D \times (1 + log(2) \times (1 + log(D) \times 2)/2$. Consider assigning justification 4 to $C_1$: this results in an expected cost for that rule of $(1 + log(D) \times 2) \times 0.3 \times 1$ (as the construction relies on making $R$ true). Additionally, it would force the grounding of the rule defining $R$, increasing the cost with the size of the rule for $R$. The optimal solution for the problem in Figure 7.3 is then rule node selection (a) with justification 6 for $C_2$. Its cost is the sum of $(1 + D^2 \times log(2) \times 0.3$ (for justification 6) and $1 + log(D) \times 2$ (the expected size of the rule for $C_1$). With this solution, only the rule for $C_1$ has to be passed to the local approach.

To solve the optimal justification problem, IDP's optimization inference itself is applied to a (meta-level) declarative specification of the task.[3] For larger theories $\mathcal{T}$, the problem turns out to be quite hard, so two approximations were considered to reduce the search space. First, the number of selected justifications for any rule was limited to 2. Second, as the values of $size(r)$ can grow quite large, the approximation $\lceil \log(size(r)) \rceil$ is used (a standard approach). The resulting specification could be solved to optimality within seconds for all tested theories. During lazy model expansion, the global approach is applied in the initial phase when all Tseitin literals representing sentences in the original theory have been propagated true.

## 7.4 Optimizations and Discussion

In this section, a number of optimizations is discussed that improve specific parts of the presented algorithms. First, we discuss heuristic choices affecting in the different algorithms. Second, it is briefly discussed how the create an even smaller grounding and how to avoid confusing the search algorithm with too many spurious Tseitin variables. Finally, we discuss how to extend the approach to the logic $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$, an extension of $\mathrm{FO}(ID)$, and to related inference tasks.

### 7.4.1 Heuristics

The algorithms leave room for a number of heuristic choices, that can have an important effect on the performance. We now briefly discuss these choices. As a guideline for our decisions, the following principles are used:

- Avoid leaving the search process without enough information to make an informed decision; for example avoid losing too much (unit) propagation or introducing too many Tseitin symbols.

---

[3] The specification is part of IDP's public distribution.

- Prevent creating a grounding that is too large; this may for example happen as the result of a very long propagate-ground sequence.

Recall, the goal is not to create a minimal grounding, but to solve model expansion problems while avoiding a too large grounding.

In `split_and_ground`, when handling a disjunction or existential quantification, there is a choice on how many disjuncts to expand. If we expand one instantiation at a time for a rule $h \leftarrow \exists x \in D : P(x)$, as done in Algorithm 12 (lines 3 and 7), iterative application results in a ground theory

$$\begin{aligned} h &\leftarrow P(d_1) \lor T_1 \\ T_1 &\leftarrow P(d_2) \lor T_2 \\ T_2 &\leftarrow P(d_3) \lor T_3 \\ &\vdots \\ T_n &\leftarrow \exists x \in D \setminus \{d_1, d_2, \dots, d_n\} : P(x). \end{aligned}$$

If we adapt the algorithm to introduce $n$ elements at a time, we obtain

$$\begin{aligned} h &\leftarrow P(d_1) \lor \ldots \lor P(d_n) \lor T \\ T &\leftarrow \exists x \in D \setminus \{d_1, d_2, \dots, d_n\} : P(x). \end{aligned}$$

In the experiments, adding 10 instantiations of an existential quantification and 3 disjuncts of a disjunction at a time turned out to give the best balance between introducing too many Tseitin atoms and grounding too much.

The thrashing behaviour that leads to the above-mentioned propagate-grounding chains for a disjunction or existential quantification can have several origins. One if them is the truth value assigned to first-time choice atoms by the search algorithm. For example, in the SAT-solver MiniSAT (used in the IDP system), initially each atom is assigned **f**. This causes the above sentence $\exists x \in D : P(x)$ to be fully grounded (as a conflict only arises when the full disjunction is grounded). To remedy this, two search-related heuristics are changed. First, the initial truth value is randomized (still favouring false), by assigning true initially with probability 0.2. Second, search algorithms typically *restart* after an (ever-increasing) threshold on the number of conflicts, sometimes caching the truth value assigned to atoms (*polarity caching*). This allows them to take learned information into account in the search heuristic while staying approximately in the same part of the search space. In case of lazy grounding, we might want to jump to another part of the search space when we come across long propagate-ground sequences. To this end, we introduce the concept of *randomized restarts*, which take place after an (ever-increasing) threshold on the number of times $\Delta_g$ is extended and randomly flipping some of the cached truth values. The initial threshold on the number of extensions is 100; it is doubled after each restart.

In addition, build_djust always returns *false* if it is estimated that the formula has a small grounding. Indeed, grounding them can help the search. In our experiments, a formula is considered small if its (estimated) grounding size is below $10^4$ atoms. The same strategy is used in split_and_ground: whenever the formula to which one_step_ground would be applied is sufficiently small, ground is applied instead, to completely ground the remaining formula.

## 7.4.2 Other Issues

We briefly discuss a number of further considerations that have an important effect on the behaviour of the algorithm, but for which a deeper discussion is out of the scope of this text.

**Propagation Guarantees.** In build_djust, we can vary how many domain element/disjuncts to select for justifying disjunctive knowledge (currently the whole domain, respectively one disjunct). Selecting only one object will result in minimal grounding, but also only guarantees *consistency*: if the justification becomes false, the associated sentence might already be false. Consequently, if its grounding had already been part of $\Delta_g$, it might have resulted in earlier propagation. There is an important subclass of justifications that guarantee that propagation is not delayed.

**Definition 7.4.1** (unit-propagation-safe). A justification $J$ of a rule $r$ of the form $h \leftarrow \varphi$ is *unit-propagation-safe* if (**i**) the head is true (false) in $\mathcal{I}$ and the justification for $h$ ($\neg h$) can be partitioned in two valid justification or (**ii**) the head is unknown in $\mathcal{I}$ and $J$ is to be a valid justification for $\varphi$ and contains $\neg h$ or is a valid justification for $\neg\varphi$ and contains $h$.

Informally, propagation might get delayed if a justification is still available, but it is the only way in which the associated sentence can still become true. By guaranteeing two disjoint justifications are available, we are sure no unit propagation (or detection of a conflict) might be delayed. This is in fact a generalization of the two-watched literal scheme [Moskewicz et al., 2001], an important optimization in SAT-solvers: for each clause, it is sufficient to maintain 2 non-false literals and only check propagation for the clause when either one becomes false. To incorporate propagation-safe justifications in our algorithm, some changes have to be made, such as adapting build_djust to only return such justifications. Also, we cannot just delay grounding of a rule as long as its head is not relevant (as its body might already be true or false). It is part of future work to experiment with this type of justifications. An obvious disadvantage is that it results in earlier grounding, but this might be offset by improved search. On the other hand, note that the presented algorithms lose propagation only **once**: when the justification becomes false, the relevant sentence is grounded, hence afterwards it is checked for propagation as it has become part of $\Delta_g$.

**Cheap Propagation Checks.** In lazy_mx, for each assigned literal it is checked whether it is defined in $\Delta_d$ and whether it violates any justifications. To implement this cheaply, a mapping is maintained from literals in $\Delta_g$ to whether they are defined in $\Delta_d$ and in which justifications its negation occurs. This mapping is created whenever a literal is first added to $\Delta_g$ and maintained whenever justifications change. Consequently, the performance of the search loop is unaffected as long as literals are assigned for which the mapping is empty.

**Grounding Order Effects.** In split_and_ground, a domain element/disjunct has to be selected when applying Tseitin transformation. In the experimental section, we have taken the simple approach of selecting the minimal object in the lexicographical order. The result is that for example for planning problems where the order on time is the lexicographical order, lazy grounding is closely related to incrementally increasing the time interval considered. It is part of future work to compare other selection heuristics, such as first selecting an instantiation which is already true/false or has already been introduced in the ground theory.

**Stopping Early.** In Algorithm 7, we took the standard stopping criterion used in most search algorithms (Line 14): to stop when $\mathcal{I}$ is two-valued (on $sent_g \cup \Delta_g$) and no conflict has been found, as it is now certainly a model of the theory. In our setting, we would prefer to stop earlier, ideally as soon as $\mathcal{I}$ is a total justification for $P_{\mathcal{T}}$, which is correct as shown in Corollary 7.2.5. Otherwise, we might ground more than necessary. Indeed, for example assigning literals defined in $\Delta_d$ by rules without a justification results in violated rules and possibly further grounding, even when the literal was not necessary for the justification of $P_{\mathcal{T}}$. Our algorithms only chooses literals that are watched by some formula/rule. It guarantees that if all watched literals have been assigned and no conflict ensues, that the current partial structure satisfies the ground theory and hence search can stop.

**Alternative Interleaving of Grounding and Search.** Grounding is applied during the search process as soon as unit propagation has taken place. The result is a focus on the current location in the search space, but with the danger of grounding too much if there is no solution in that part of the space. Alternatively, we could apply the opposite strategy, namely to ground as late as possible: only apply additional grounding when the search algorithm terminates without ever having found a model in an acceptable default state. Such a strategy is well-known from the fields of incremental proving and planning, where the domain (number of time steps) is only increased after search over the previous, smaller bound has finished. The latter strategy guarantees a minimal grounding. A prototype of this strategy has been implemented in IDP with good results on planning problems.

**Approximate Justifications.** The build_djust algorithm only returns valid, symbolic justifications. Hence, even if only one literal violates a potential justification, build_djust returns *false*. Consider, for example, the formula $\forall x \in D : P(x) \Rightarrow \varphi$ and a structure $\mathcal{I}$ in which $P$ is unknown except for $P(d_1)^{\mathcal{I}} = \mathbf{t}$. Here, build_djust will not return the justification that makes $P$ completely false (as $P(d_1)$ is true in $\mathcal{I}$) and as a result, $x$ will be instantiated with all elements in $D$ in the subsequent grounding step (assuming no justifications can be found on $\varphi$). We can improve over this as follows: within build_djust, we estimate the number of literals that would violate a potential justification. If this number is small enough compared to the number of potential instantiations, the (invalid) justification is still returned. However, the justification is queried for instantiations that are false in $\mathcal{I}$ and call lazy_ground is applied to each of those, as if they were violations during search. Hence, for the above example, "making $P$ false" will be returned as a justification and lazy_ground will be applied to $P(d_1)$, only instantiating $x$ with $d_1$.

**Decision Literals.** Sometimes, we can delay grounding even longer using the idea that in fact, the search algorithm itself also works towards justifying $P_{\mathcal{T}}$. Indeed, suppose we have the following sentences, reflecting some case-based reasoning scheme.

$$P_1 \Rightarrow \varphi_1 \qquad P_2 \Rightarrow \varphi_2 \qquad \dots \qquad P_n \Rightarrow \varphi_n$$

Further suppose the rest of the theory entails that only one $P_i$ will ever be true. In that case, we can delay grounding of any of these sentences until some $P_i$ has become true (and only ground that one sentence then), without the need to maintain justifications. However, to guarantee soundness, we have to add all $P_i$ to the *decision list* of the search algorithm: the list of atoms that are guaranteed to be assigned in any (partial) interpretation that is returned as a model. Then, we are sure some $P_i$ will become true (forced by the rest of the theory) and thus the relevant sentence will certainly be grounded. In our implementation, the global approach takes this possibility into account to attempt to find an even larger part of the theory that does not yet have to be grounded.

### 7.4.3 Extension to $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$

So far, we have described a lazy model expansion algorithm for function-free $\mathrm{FO}(ID)$. However, $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$, the knowledge base language of the IDP system, supports a richer input language that includes partial functions, aggregates, arithmetic and types. More work is needed to fully exploit the potential of lazy grounding for these extensions. However, a straightforward implementation is to slightly adapt build_djust (Algorithm 10): the literal case is extended to return *false* when the literal is not a function-free literal and an extra case, also returning *false*, is added to handle expressions that do not belong to function-free $\mathrm{FO}(ID)$. For example, given a rule $h \leftarrow \forall x : P(x) \lor Q(f(x))$, $P(x)$ can be used in a justification but $Q(f(x))$ cannot. This is the approach used in the experiments of Section 7.5.

### 7.4.4 Related Inference Tasks

The bulk of the chapter focuses on model expansion (MX) for $\mathrm{FO}(ID)$ theories $\mathcal{T}$, for which solutions are structures which are two-valued on $voc(\mathcal{T})$. Often, one is only interested in a small subset of the symbols in $voc(\mathcal{T})$. This is for example the case for model generation for $\exists\mathrm{SO}(ID)$, the language which extends $\mathrm{FO}(ID)$ with existential quantification over relations. An $\exists\mathrm{SO}(ID)$ problem $\exists P_1, \ldots, P_n : \mathcal{T}$ with an initial structure $\mathcal{I}$, relation symbols $P_1, \ldots, P_n$, and $\mathcal{T}$ an $\mathrm{FO}(ID)$ theory, can be solved by model generation for the $\mathrm{FO}(ID)$ theory $\mathcal{T}$ with initial structure $\mathcal{I}$ and by dropping the interpretation of the symbols $P_1, \ldots, P_n$ in the models. Another example is query evaluation for $\mathrm{FO}(ID)$: given a theory $\mathcal{T}$, an initial structure $\mathcal{I}$ and a formula $\varphi$ with free variables $\overline{x}$ (all in $\mathrm{FO}(ID)$), the purpose of evaluating the query $\langle \mathcal{T}, \mathcal{I}, \varphi \rangle$ is to find assignments of domain elements $\overline{d}$ to $\overline{x}$ such that a model of $\mathcal{T}$ exists that expands $\mathcal{I}$ and in which $\varphi[\overline{x}/\overline{d}]$ is true. To solve it by model expansion in FO($ID$), a new predicate symbol $T$ is introduced and answers to the query are tuples of domain elements $\overline{d}$ such that $T(\overline{d})$ is true in a model of the theory $\mathcal{T}$ extended with the sentence $\exists \overline{x} \in \overline{D} : T(\overline{x})$ and the definition $\{\forall \overline{x} \in \overline{D} : T(\overline{x}) \leftarrow \varphi\}$.

In both cases, approaches using (standard) model expansion compute a total interpretation and afterwards drop all unnecessary information, which is quite inefficient. Lazy model expansion can save a lot of work by only partially grounding the theory. However, once a model is found for the grounded part, the justifications and the remaining definitions are used to expand the structure to a model of the full theory. Although this expansion is obtained in polynomial time, it is still inefficient when afterwards a large part of the model is dropped.

To remedy this, we define a variant of the model expansion task, denoted *restricted* MX. Restricted MX takes as input a theory $\mathcal{T}$, a structure $\mathcal{I}$ and an additional list of symbols $O$, called *output symbols*.[4] Solutions are then structures $M$ which are two-valued on all symbols in $O$ and for which an expansion exists that extends $\mathcal{I}$ and is a model of $\mathcal{T}$. Adapting lazy grounding to solve restricted MX can be done through an analysis of which justifications need not be added (completely) to the structure, splitting $\Delta_{\mathrm{gd}}$ into multiple definitions and only evaluating those defining output symbols or symbols on which those depend (using a stratification argument).

The above-mentioned inference tasks can be cast trivially to restricted MX problems and lazy restricted MX then greatly improves the efficiency with respect to ground-and-solve, as shown in the experimental section.

The extension of FO($ID$) with *procedurally interpreted* symbols [De Cat et al., 2014] provides another class of interesting problems. Such predicate symbols have a fixed interpretation, but to know whether a tuple belongs to the predicate, a procedural function has to be executed. Such an approach provides a clean way to combine declarative and procedural specifications. Consider for example a symbol $isPrime(\mathbb{N})$ that is interpreted by a procedure which executes an efficient prime-verification algorithm and returns true iff the given argument is prime. We are generally not interested in the complete interpretation of $isPrime$, so it can be cast as a restricted MX problem with $isPrime$ not in $O$. Solving such a problem using lazy grounding then has the benefit of only executing the associated function *during* search for relevant atoms $isPrime(d)$. Also for this task, we show an experimental evaluation in the next section.

---

[4] Within the ASP community, they are sometimes referred to as "show" predicates.

## 7.5 Experiments

The IDP system has a state-of-the-art model expansion engine, as can be observed from previous Answer-Set Programming competitions ([Denecker et al., 2009], [Calimeri et al., 2012] and [Alviano et al., 2013]). The lazy model expansion algorithms were implemented in the IDP system, by extending the existing algorithms [De Cat et al., 2013a]. The implementation is preliminary in the sense that direct justifications are only derived for sentences, but not for definitions. However, the grounding of rule instances can be delayed until their head is assigned and Tseitin introduction can be applied to disjunctive bodies. As non-inductive definitions can be equivalently expressed as their completion, this does not significantly restrict the applicability of the implementation.

We tested three different setups: IDP with the standard ground-and-solve approach (referred to as `g&s`), IDP with lazy model expansion (`lazy`) and the award-winning ASP system Gringo-Clasp (`ASP`). All experiments were run on an 64-bit Ubuntu 13.10 system with a quad-core 2.53 Ghz processor and 8 Gb of RAM. A timeout of 1000 seconds and a memory limit of 3 Gb was used; out-of-time is indicated by `T`, out-of-memory by `M`. We used IDP version 3.2.1-lazy, Gringo 3.0.5 and Clasp 2.1.2-st.[5]

The section is organized as follows. In Section 7.5.1, we evaluate the overhead of completely grounding a theory using the presented approach. In Section 7.5.2, we evaluate the effect of lazy grounding on a number of benchmarks of the ASP competition. Afterwards, in Section 7.5.3, a number of additional properties of the presented algorithms are demonstrated.

### 7.5.1 Lazy Grounding Complexity

To quantify the overhead of lazy model expansion over the standard grounding algorithm, we evaluated the time necessary to completely ground a theory over a given structure in both IDP setups. To this end, we minimally adapted the lazy grounding implementation to immediately add literals in the ground theory or in a justification to the $q_{ch}$ queue and by setting the threshold under which a formula is considered "small" to 0 atoms. As discussed previously, we expected the (naive) incremental querying of justifications to be a bottleneck.

We devised six benchmarks to test various aspects of the novel algorithm that set it apart from non-incremental grounding. The tested aspects are:

---

[5] Benchmarks, experimental data and complete results are available at `dtai.cs.kuleuven.be/krr/files/experiments/lazygrounding2014experiments.tar.gz`.

1. Existential quantification using Tseitin transformation.
2. Definitions without justification for the body.
3. Sentences without disjunctive justifications.
4. The effect of intelligent grounding techniques that derive smaller bounds on variable instantiations [Wittocx et al., 2010, Wittocx et al., 2013]. We distinguish between standard "grounding without LUP" (for Lifted Unit Propagation), which derives bounds for a quantified variable based only on its subformula, and "grounding with LUP", which also incorporates information from the rest of the theory.
5. Nested universal quantification for a sentence of the form $\forall \overline{x} : P(\overline{x}) \Rightarrow \forall \overline{y} : Q(\overline{x}, \overline{y})$. Recall, the implementation of the local approach only derives justifications on one quantification level. As a result, first the justification is found which sets $P$ false; when this wakes up, it results in splitting the formula and a new justification that sets $Q$ true.
6. Sentences with disjunctive justifications with shared variables.

Experiments were done with domains of size 10, 20, 30, 40 and 50.

In all experiments, the overhead for the time required to solve the initial optimization problem (for the global approach) was always around 0.02 second, so in itself is negligible. The results for the first three experiments are not shown as the differences were negligible. The results for the latter three are shown in Figure 7.4. As expected, there is a significant overhead to handle more complex justification formulas, especially when variables are shared (in which case significant query optimization is possible). However, these results are a worst-case view of the overhead: in practice, we expect that often, constructing the full grounding will not be necessary in the first place. Still, it is an important part of future research to improve the querying algorithm to reduce it to an acceptable overhead.

### 7.5.2 ASP Competition Benchmarks

Second, we selected benchmarks from previous ASP competitions to evaluate the lazy grounding algorithm in a more realistic setting. For most of the available benchmarks, the encoding can usually be sufficiently optimized so that the grounding is not too large. Hence, to show the improvements of lazy grounding over ground-and-solve, we focused on benchmarks in which a natural, predicate encoding can easily result in a large grounding. We also

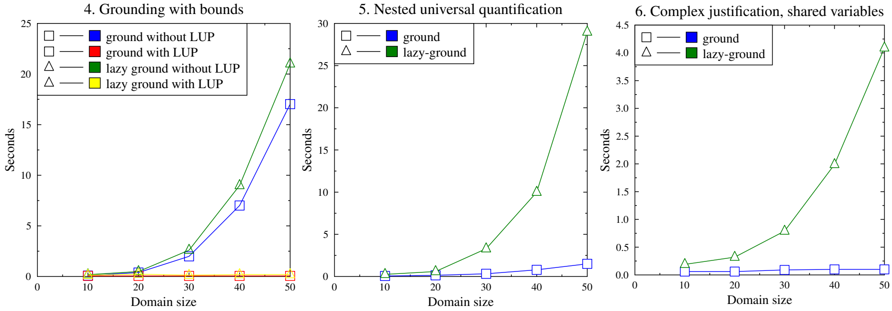


Figure 7.4: Time overhead of lazy grounding over ground-and-solve when completely grounding the input theory, for benchmarks 4, 5 and 6.

selected some benchmarks that are known to be hard, to evaluate the effect of lazy model expansion on search. We selected the following problems (see the competition websites for complete descriptions):

- `Reachability`: Given a directed graph, determine whether a path exists between two given nodes.
- `Labyrinth`: A planning problem where the aim is to manipulate a graph such that the goal node becomes reachable from the start node.
- `Packing`: Given a 2-D rectangle and a number of squares, fit all squares into the grid without overlaps.
- `Disjunctive Scheduling`: Schedule a number of actions with a given earliest start and latest end time with additional constraints on precedence and disjointness.
- `Sokoban`: A planning problem where a robot has to push a number of blocks to goal positions, constrained by a 2-D maze.
- `Graph Colouring`: Given a graph, assign a colour to each node (from a given set of colours), such that no directly connected nodes have the same colour.
- `Stable Marriage`: Given a set of men and women and a set of match preferences, find an assignment that is "stable": no swap results in a better or equal match.
- `Hydraulic Planning`: A planning problem where the task is to turn on a number of valves in a pipe system in the proper order such that some goal nodes become pressurized.

| | # solved | | | avg. time (sec.) | | |
|---|---|---|---|---|---|---|
| benchmark | g&s | lazy | ASP | g&sIDP | lazy IDP | ASP |
| `Sokoban` | 9 | 5 | **10** | 370 | 1538 | **22** |
| `Disjunctive Sched.` | 4 | **10** | 2 | 1933 | **92** | 2400 |
| `Packing` | 9 | **10** | 1 | 352 | **51** | 2704 |
| `Labyrinth` | **6** | 5 | **6** | 1286 | 1851 | **1015** |
| `Reachability` | 1 | **10** | 4 | 2706 | **7** | 1807 |
| `Stable Marriage` | 0 | **10** | 5 | 3000 | **350** | 1563 |
| `Graph Colouring` | **7** | 3 | 4 | **1013** | 2103 | 1842 |

Table 7.1: The number of solved instances for the ASP benchmarks (10 instances each) and the average time taken.

For each of these, we randomly selected 10 of the 2011 competition instances (2013 for `Stable Marriage`). For `Stable Marriage`, `Graph Colouring` and `Reachability`, we based our encodings on the available ASP-Core-2 encodings. For `Packing`, `Disjunctive Scheduling`, `Hydraulic Planning` we constructed a natural IDP encoding and made a faithful translation to ASP. For the more complex benchmarks of `Labyrinth` and `Sokoban`, we used the original IDP and Gringo-Clasp specifications sent in to the 2011 competition. For the lazy model expansion, we replaced cardinality expressions by their FO encoding as for the former no justifications are derived yet; this also increases the size of the full grounding. Note that the encoding used is not necessarily the most efficient one. Specifically for `Disjunctive Scheduling` and `Hydraulic Planning`, more efficient encodings are available, using uninterpreted functions and an encoded algorithm respectively.

For `Hydraulic Planning`, none of the instances could be solved by any setup. For the other benchmarks, the number of solved instances and average time are shown in Table 7.1; the average grounding size for the IDP setup is shown in Table 7.2. For unsolved instances, we took three times the time threshold as time and $10^{10}$ as grounding size in case of memory overflow.[6] Memory overflows only happened in `Reachability` (7 times for `g&s`), `Labyrinth` (4 times for `g&s`, once for `lazy`) and `Packing` (once for `g&s`); all other unsolved instances were caused by a time-out.

The results show that lazy model expansion solved more instances than the other setups in four out of seven cases. In those cases, the problems also got solved significantly below the time threshold. In five out of seven cases, the (final) grounding size was smaller for lazy model expansion, even orders of

---

[6] IDP has automatic symmetry breaking, the cause of the difference between `g&s` and `ASP` for `Graph Colouring`.

| | grounding size (# atoms) | |
|---|---|---|
| benchmark | `g&s` | `lazy` |
| `Sokoban` | $\mathbf{2.5 \times 10^4}$ | $7.8 \times 10^5$ |
| `Disjunctive Sched.` | $6.8 \times 10^7$ | $\mathbf{3.5 \times 10^6}$ |
| `Packing` | $1.0 \times 10^9$ | $\mathbf{1.4 \times 10^7}$ |
| `Labyrinth` | $4.0 \times 10^9$ | $\mathbf{2.0 \times 10^9}$ |
| `Reachability` | $7.1 \times 10^9$ | $\mathbf{1.5 \times 10^4}$ |
| `Stable Marriage` | $4.4 \times 10^7$ | $\mathbf{1.5 \times 10^7}$ |
| `Graph Colouring` | $1.1 \times 10^4$ | $1.6 \times 10^4$ |

Table 7.2: The average grounding size for the ASP benchmarks for both IDP setups. For the `lazy` setup, the size of the final ground theory was taken.

magnitude in three cases. For `Sokoban`, `Labyrinth` and `Graph Colouring`, lazy model expansion was outperformed by ground-and-solve, indicating that the loss of information outweighed the gain of grounding less up-front. E.g., for `Sokoban`, the `lazy` final grounding size was even higher than for `g&s` (possible due to the FO encoding of cardinalities), indicating that a large part of the search space was explored. For `Stable Marriage`, the relatively small difference in grounding size leads us to believe that the different search heuristic was the main factor and not lazy grounding itself.

For hydraulic planning (for which no instance was solved), we observed that for ground-and-solve, the computation went out of memory, while lazy model expansion was always able to start searching, even though it was not able to find any solutions. A similar observation was made during experiments with the `Airport Pickup` ASP-2011 benchmark, a fairly standard scheduling problem (transporting passengers by taxis taking into account fuel consumption) except that no upper bound on time is provided. Hence any ground-and-solve approach would need to construct an infinite grounding. Applying straightforward lazy model expansion also resulted in a grounding that was too large. Instead, we implemented a prototype of an alternative approach in which grounding is not done as soon as a justification becomes violated, but is delayed until the solver has proven that the currently ground theory is unsatisfiable (has too few time steps to solve the task), see also Section 7.4.2. The prototype only solved one of ten instances (instead of none), but for the other instances, the grounding was not the problem: the search heuristic took too long at each of the separate time intervals $1..n$ to get up to a sufficient $n$ for which the problem was solvable (also with the standard search heuristic).

The presented results show that, although often beneficial, lazy model expansion can be a considerable overhead for some hard search problems. Therefore, the logical step is with the current algorithms is then a *portfolio approach*, which runs both the ground-and-solve and lazy model expansion in tandem. We expect such an approach to come close to the best solution in all presented benchmarks. However, on all the problems considered, lazy model expansion could start search much earlier than ground-and-solve, even though it got lost more often during search. This leads us to believe that combining lazy model expansion with a better heuristic (or possible a user-specified one) will allow it to demonstrate its full capabilities.

### 7.5.3 Specific Experiments

Next to the ASP competition benchmarks, some experiments were conducted using constructed benchmarks/instances to illustrate specific properties of the lazy grounding algorithm.

The first part of Table 7.3 shows the results of scalability experiments. For each of the benchmarks `Packing`, `Sokoban` and `Disjunctive Scheduling`, we selected a very simple instance and evaluated the effect of increasing the domain size (either time or grid size) by orders of magnitude. The results shows that for each of the instances, lazy model expansion scales much better than ground-and-solve, both for IDP and Gringo-Clasp and for satisfiable and unsatisfiable instances. However, for `Disjunctive Scheduling` the solving time still increases significantly. The reason is that the lazy heuristics are still naive and make uninformed choices too often.

The second part of the table shows results for some crafted benchmarks:

- `Dynamic reachability`, the example described in Section 7.2.3.
- Lazy evaluation of `procedurally interpreted` symbols, using a simple theory over the prime numbers. As described in Section 7.4.4, a predicate symbol $isPrime/1$ is interpreted by a procedure that returns true if the argument is prime.
- A predicate encoding of a `function` with a huge domain (note that a state-of-the-art encoding using aggregates or uninterpreted functions instead of a predicate encoding would perform better).
- A `modelgeneration`-like application, where the aim is to also find the domain of the structure.

| benchmark | lazy | g&s | ASP |
|---|---|---|---|
| Packing-10 | 0.2 | 2.0 | 0.1 |
| Packing-25 | 0.3 | 2.0 | 0.1 |
| Packing-50 | 1.1 | 10.03 | 5.8 |
| Sokoban-$10^3$ | 0.31 | 0.3 | 0.1 |
| Sokoban-$10^4$ | 0.5 | 20.0 | 1.1 |
| Sokoban-$10^5$ | 2.6 | T | 68.0 |
| Disjunctive Sched.-sat-$10^3$ | 0.39 | 0.49 | 0.07 |
| Disjunctive Sched.-sat-$10^4$ | 13.04 | 16.05 | 17.44 |
| Disjunctive Sched.-sat-$10^5$ | 164.18 | M | M |
| Disjunctive Sched.-unsat-$10^3$ | 0.24 | 0 49 | 0.09 |
| Disjunctive Sched.-unsat-$10^4$ | 4.11 | 16.04 | 19.85 |
| Disjunctive Sched.-unsat-$10^5$ | 164.2 | M | M |
| dynamic reachability | 0.18 | M | M |
| procedural | 1.24 | M | M |
| function | 0.79 | M | M |
| modelgeneration | 0.19 | M | M |

Table 7.3: Additional experiments on crafted benchmarks, one instance each.

For each one, a faithful ASP encoding was constructed. The results show a significant improvement of lazy model expansion over ground-and-solve on all examples.

**Closer to Inherent Complexity?** During the modeling phase of an application, different encodings are typically tested out, in an attempt to improve performance or to locate bugs. While modeling our experimental benchmarks, we noticed that simplifying a theory by dropping constraints often resulted in a dramatic reduction in the time lazy model expansion took to find a model. Standard model expansion, on the other hand, was much less affected by such simplifications. In our opinion, this observation, while hardly definitive evidence, is another indication that the presented algorithms are able to derive justifications for parts of a theory that can be satisfied cheaply. In that way, the approach is able to distinguish better between problems which are inherently difficult and problems which would just have a large grounding.

## 7.6 Related Work

Lazy model expansion offers a solution for the blowup of the grounding that often occurs in the ground-and-solve model expansion methodology for FO($ID$) theories. ASP and Sat-Modulo-Theories techniques also process theories that can have a large grounding; the constraint store of CP and Mixed Integer Programming and the clauses of SAT can be considered the equivalent of a grounded theory (they are often derived from quantified descriptions such as "$c_i < c_j$ **for all** $i$ and $j$ for which …") and can also become very large. Various authors have reported a blowup problem in these related paradigms [Lefèvre and Nicolas, 2009, Ge and de Moura, 2009] and a multitude of techniques has been developed to address it. We distinguish four general approaches.

First, concerning grounding up-front, research has been done towards *reducing the size of the grounding* itself through (**i**) *static analysis* of the input to derive bounds on variable instantiations [Wittocx et al., 2010, Wittocx et al., 2013], (**ii**) techniques to *compile* specific types of sentences into more compact ground sentences [Tamura et al., 2009, Metodi and Codish, 2012], (**iii**) detect parts that can be evaluated polynomially [Leone et al., 2006, Gebser et al., 2011b, Jansen et al., 2013] and (**iv**) detect parts that are not relevant to the task at hand (e.g., in the context of query problems) [Leone et al., 2006]. Naturally, each of these approaches can be used in conjunction with lazy grounding to further reduce the size of the grounding. In IDP for example, lazy grounding is already combined with (**i**) and (**iii**).

Second, the size of the grounding can be reduced by *enriching* the language the search algorithm supports. For example, ASP solvers typically support ground aggregates (interpreted second-order functions such as cardinality or sum that take sets as arguments), and CP and SMT solvers support (uninterpreted) functions. More recently, the Constraint-ASP paradigm was developed [Ostrowski and Schaub, 2012], that integrates ASP and CP by extending the ASP language with *constraint* atoms. These are interpreted as constraints in a CSP problem and can thus be handled using CP techniques. Various CASP solvers are already available, such as Clingcon [Ostrowski and Schaub, 2012], Inca [Drescher and Walsh, 2011a], Ezcsp [Balduccini, 2011], Mingo [Liu et al., 2012] and IDP [De Cat et al., 2013a]. Inca and IDP in fact implement Lazy Clause Generation [Stuckey, 2010], an optimized form of lazy grounding for specific types of constraints. The language HEX-ASP [Eiter et al., 2005] also extends ASP, this time with *external* atoms that represent (higher-order) external function calls.

Third, *incremental grounding* is a technique well-known from model generation, theorem proving and planning. For these tasks, the domain is typically not fixed in advance, but part of the structure being sought, such as the number of time steps in a planning problem. Incremental grounding then works by grounding the problem for an initial guess of (the number of elements in) the domain. Afterwards, search is applied; if no model was found, the domain is extended and more grounding is done. This is iterated until a model is found or a bound on the maximum domain size is hit (if one is known). This technique is applied, e.g., in the prover Paradox [Claessen and Sörensson, 2003] and the ASP solver IClingo [Gebser et al., 2008].

Fourth, and closest to lazy grounding itself, is a large body of research devoted to delaying the grounding of specific types of expressions until necessary (for example until they result in propagation). Propagation techniques on the first-order level that delay grounding until propagation ensues have been researched within ASP [Lefèvre and Nicolas, 2009, Dal Palù et al., 2009a, Dal Palù et al., 2009b, Dao-Tran et al., 2012] and within CP [Stuckey, 2010]. Such techniques can be used in conjunction with lazy grounding as they derive more intelligent justifications for specific types of constraints than presented here. For example, in [Dao-Tran et al., 2012] the authors present an efficient algorithm for bottom-up propagation in a definition. Within SMT, various theory propagators work by lazily transforming their theory into SAT, such as [Bruttomesso et al., 2007] for the theory of Bit Vectors. Ge et al. [Ge and de Moura, 2009] investigated quantifier handling by combining heuristic instantiation methods with research into decidable fragments of FO theories, as these can be efficiently checked for models. In [Saptawijaya and Pereira, 2013], an abduction framework is extended to lazily generate part of the relevant sentences. In search algorithms themselves, justifications (or *watches*) are used to derive when a constraint will not result in propagation or is already satisfied, and hence need not be checked in the propagation phase. In [Nightingale et al., 2013], it is shown how maintaining (short) justifications can significantly reduce the cost of the propagation phase.

In fact, a well-known technique already exists that combines search with lazy instantiation of quantifiers, namely *skolemization*, where existentially quantified variables are replaced by newly introduced function symbols. Universal quantifications are handled by instantiating them for those introduced function symbols. Reasoning on consistency can, e.g., be achieved by congruence closure algorithms, capable of deriving consistency without effectively assigning an interpretation to the function symbols. These techniques are used in Tableau theorem proving [Hähnle, 2001] and SMT solvers [Detlefs et al., 2005]. Formula [Jackson et al., 2013] interleaves creating a ground program and giving it to an SMT solver, iterating when symbolic guesses proved to be wrong. Skolemization-based techniques typically work well in case only a small number of constants needs to be introduced, but have difficulty in case the relevant domain is large. One can also see that lazy grounding (with support for function symbols) could incorporate skolemization by adapting the rules for grounding existential and universal quantification. We expect skolemization to be complementary to lazy grounding, but an in-depth investigation is part of future work.

In the field of probabilistic inference, several related techniques have been developed that also rely on lazy instantiation. First, the Problog system uses a form of static dependency analysis to ground a (probabilistic) program in the context of a given query, by constructing all possible ways to derive the query in a top-down fashion [Kimmig et al., 2011]. Second, so-called *lazy inference*, applied e.g. in *LazySAT* [Singla and Domingos, 2006], exploits the fact that, for the considered inference, a (fixed) *default* assumption exists under which an expression certainly does not contribute to the probabilities. Hence, expressions for which the assumption certainly holds do not have to be considered during search. Third, *cutting place inference* [Riedel, 2009] applies lazy inference in an interleaved setting, only constructing the part of the program for which the assumptions are not satisfied.

## 7.7 Future Work

In future, several aspects of the presented work will be investigated further.

One aspect is extending support to lazy ground more complex expressions, including aggregate expressions and (nested) function terms. Consider for example the sentence $(\sum_{x \in D \text{ and } P(x)} f(x)) > 3$, which expresses that the sum of terms $f(d)$ for which the atom $P(d)$ is true, with $d \in D$, $P$ a predicate and $f$ a function, should be larger than 3. One can observe that it is not necessary to ground the whole sentence up-front. E.g., if $f$ maps to the natural numbers (hence positive), the set $\{P(d_1), f(d_1) > 3\}$ is a minimal justification. Even if no easy justification can be found, we can suffice by grounding only part of the sentence and delay the remainder. E.g., we can create the ground sentence $(\sum_{P(d_1)} f(d_1)) > 3 \lor T$, with $T$ a Tseitin symbol defined as $(\sum_{P(d_1)} f(d_1)) + (\sum_{x \in D \setminus d_1 \text{ and } P(x)} f(x)) > 3$. Indeed, in any model of the sentence in which $T$ is false, the original inequality is satisfied.

A second aspect is whether there are advantages to grounding earlier, for example to guarantee no propagation is lost, or grounding later, possibly reducing the size of the grounding even more. For example, consider the sentences $P \Rightarrow \phi$ and $\neg P \Rightarrow \psi$, with $\phi$ and $\psi$ both large formulas for which no justification was found. Instead of grounding at least one of the sentences, we might add $P$ to the list of atoms the search algorithm has to assign and only ground either of the sentences when $P$ has been assigned a value (it might even be that unsatisfiable is detected before grounding either one).

Lastly, what about lazy *forgetting*? As the ground theory is extended when making the structure more precise, the ground theory could be reduced again during backtracking. By storing the justification violations that caused which grounding, we can derive which grounding can be forgotten again if the violation is no longer problematic (e.g., after backtracking). For this, an algorithm needs to be developed which tracks grounding/splitting dependencies between rules given their justifications. This closely resembles techniques used in tableau theorem proving and SMT, where the theory at hand can be compacted when moving to a different part of the search space.

## 7.8 Conclusion

Solvers used in the domains of SAT, SMT and ASP are often confronted with problems that are too large to ground. Lazy model expansion, the technique described in this chapter, interleaves grounding and search in order to avoid the grounding bottleneck. The technique builds upon the concept of a justification, a deterministic recipe to extend an interpretation such that it satisfies certain constraints. A theoretical framework has been developed for lazy model expansion for the language FO($ID$) and algorithms have been presented to derive and maintain such justifications and to interleave grounding with state-of-the-art CDCL search algorithms. The framework aims at bounded model expansion, in which all domains are finite, but is also an initial step towards handling infinite domains efficiently. Experimental evaluation has been provided, using an implementation in the IDP system, in which lazy model expansion was compared with a state-of-the-art ground-and-solve approach. The experiments showed considerable improvement over ground-and-solve in existing benchmarks as well as in new applications. The main disadvantage is the less-informed search algorithm, caused by the delay in propagation and the introduction of (more) additional symbols. This leads us to believe a portfolio approach, heuristically selecting between ground-and-solve and lazy model expansion, would result in an overall improvement.

# 8

# Conclusion

The main contributions and conclusions are summarized in this chapter. Afterwards, some directions for future work are discussed.

## 8.1 Contributions and Conclusions

The aim of this dissertation was two-fold. On the one hand, to evaluate the KBS paradigm in practice, by building a KBS and evaluating its applicability. On the other hand, to improve the state-of-the-art for the core inference tasks of model expansion and optimization. The result are the following contributions.

- The *knowledge base system* IDP was developed with the declarative logic $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ as its knowledge base language. It provides an integration with the procedural language Lua and a range of inference engines such as querying, deduction, model expansion and optimization. Several applications have been discussed and a case study has been worked out, which show that IDP is indeed applicable to a broad range of tasks. In the considered applications, IDP demonstrated good or superior performance, natural modeling capabilities and reduced development time compared to procedural solutions.

- Throughout the chapters, we showed that the KBS paradigm simplifies *reuse* of inference engines, as they are integrated in a single system. Support for additional tasks such as declarative debugging, definition postprocessing and function detection were implemented with a minimal amount of additional code. Hence, inference engine development in the context of a KBS has the important advantage of directly carrying over improvements to all engines that make use of it. For example, improvements to the deduction engine will immediately have an effect on model expansion through the detection of additional functional dependencies.

- An optimization engine was developed, building on earlier research in the KRR group [Mariën, 2009, Wittocx, 2010]. We focused on the problem of *quantifier instantiation* in the ground-and-solve approach, which causes the intermediate theory to blow up, and on the robustness of the engine, to reduce the effect of (naive) modeling on performance. We formalized the model expansion workflow in terms of rewrite and transition rules and implemented it as part of IDP. To address the instantiation problem, we first discussed the importance of existing preprocessing techniques such as symbolic unit propagation, grounding with bounds and definition preprocessing. Afterwards, we developed the following three novel approaches to further improve optimization.
  - In Chapter 5, we showed that *allowing function symbols to occur in the ground theory* significantly reduces the size of the intermediate theory. We developed the search algorithm MINISAT(ID) for general ground $\text{FO}(\cdot)^{\text{IDP}}$ and showed that techniques from SAT, ASP and CP can be cleanly integrated, resulting in a state-of-the-art optimization engine. This solves a first type of instantiation problem by not grounding the implicit quantifications up-front to which function terms give rise. In one set of experiments, MINISAT(ID) was shown to be the best free-search MiniZinc system out of 12 compared systems.
  - To reduce the burden of having to distinguish function and predicate symbols for performance's sake, we developed a method to automatically *detect function symbols* in Chapter 6. The method uses deduction inference, which was implemented by reducing $\text{FO}(\cdot)^{\text{IDP}}$ theories to weaker FO theories and applying a state-of-the-art FO theorem prover (SPASS). Whenever functional dependencies are detected, they are made explicit by rewriting the theory. This operation removes quantifications and hence reduces the size of the grounding, so is another step towards solving the instantiation problem. Interestingly, the approach is also a partial solution

to the long-standing problem of *automatically reducing the arity of symbols*. Experimental evaluation demonstrate that model expansion performance of predicate specifications could be significantly improved using this approach.

- Finally, in Chapter 7, we developed the *lazy model expansion* framework, a general approach on how to tightly interleave grounding and search to solve model expansion problems. The framework, developed for FO($ID$), is based on maintaining justifications for non-ground formulas: as long as such a justification can still be made true, the associated formula does not need to be considered by search just yet and can be kept non-ground. Only when no justification can be found in the current structure is additional grounding necessary. The result is a theoretical framework, which captures many existing techniques, and an implementation in IDP of an instantiation of the framework, which maintains justifications symbolically. Experiments show lazy model expansion significantly improves over ground-and-solve on selected benchmarks, but that better heuristics are required to further exploit its potential.

From these contributions, we can conclude that IDP is a maturing KBS system and that the KBS paradigm has significant advantages as a software engineering paradigm. Most importantly, it allows application engineers to reduce development time while offering similar or even increased efficiency. The rich language, based on FO, is indeed experienced as quite natural, even by people not familiar with declarative approaches. The developed optimization algorithm is a state-of-the-art inference engine which can be applied to a variety of problems. The integration of various inference engines within one system demonstrated many points of possible reuse and the integration in a procedural language balances declarativity with usability.

## 8.2 Future Research Directions

As with any dissertation, many novel and promising ideas arose during the research, most of which could not be studied in detail. In this section, we give an overview of the ideas which we feel are promising directions for future research. Given the contributions outlined above, they can be largely grouped in two categories. In Section 8.2.1, we discuss further research related to the KBS paradigm and IDP as an instantiation. In Section 8.2.2, we go into improvements to the optimization inference.

### 8.2.1 KBS Paradigm

As future research into the KBS paradigm, we outline several interesting directions below.

**Extending the KBS Language.** IDP's KBS language, $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$, is already a rich KR language, consisting of predicate logic extended with inductive definitions, aggregates, arithmetic, partial functions and a type system. However, there are various important concepts that cannot be expressed naturally in $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$. One example is causal or probabilistic information, such as "A teacher is sick with probability 0.05" or "A stallion and a mare that are put in the same field may cause the birth of a foal". Another example are expressions over sets of elements, such as "for all paths $P$ in a graph, it holds that ...". We would need second-order logic (or higher-order logic by extension to sets of sets of ... ) to naturally model such statements.

**Increasing Robustness.** During the development of IDP, one issue became apparent: the aim to make inference engines more robust has a significant impact on the complexity of the system. We observe that often, when presented with a (naive) specification that performs poorly, it is quite clear to an experienced modeler how to increase it performance. In some cases, a general pattern can be extracted and the engine is improved to automate the optimization. We already discussed some examples in this text, such as pushing quantifications down and introducing Tseitin symbols for formulas occurring multiple times. However, there is a large number of such possible optimizations and while each improves the performance for some benchmarks, it also makes the system more complex, has no effect on many other benchmarks or might even reduce performance on those. The effect is that the these systems become highly complex over time, in the aim of automatically tackling more benchmarks efficiently. Such systems are increasingly complex to develop, maintain and compare. This trend can be observed, for example, in the fields of ASP and CP: an increasingly smaller set of increasingly complex systems dominate the field. We think that to make such systems viable in the long run and to allow novel research without too much implementation overhead, it is crucial to investigate how this complexity can be addressed, for example by modularizing inference engines and reusing components over different research groups.

**Increasing Functionality.** A KBS system revolves around its inference engines. To increase its applicability, it is important to support additional functionality.

Within our research group, we are currently working on various extensions, such as (**i**) integration with other procedural languages (Scala and Haskell), (**ii**) temporal reasoning tasks, such as progression and verification, and (**iii**) generic support for meta-level reasoning, which would for example make bootstrapping (see Section 3.5.3) easier to apply.

### 8.2.2 Improving Search

Concerning search, one direction of future research is to extend the presented algorithms to further. First, lazy grounding can be improved from FO($ID$) to full $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$ by adding support for functions and aggregates. Second, the approach to detect functional dependencies could be generalized to detect global constraints. Applying a search algorithm with support for such globals would allow an even smaller grounding and improved search performance.

A final direction of future research are heuristics. In this thesis, we usually applied VSIDS as our search technique. In several cases, we needed additional heuristics, for example to guide the lazy encoding of functions, or needed to adapt the search heuristics, for example to reduce thrashing in lazy model expansion. These adaptations were always relatively naive, and for example for lazy grounding, experiments showed that the potential of a small grounding is sometimes lost as search (still) goes off in the wrong direction. We think this could be mediated by either developing better domain-independent heuristics or by adding the possibility to include heuristics (declaratively?) as part of the specification, similar to techniques used in, e.g., Constraint Programming.

# A

# Proofs

In this chapter, we provide the proofs that were omitted in the main body of the text.

## A.1 Proof of Proposition 4.2.2

*Proof.* We assume without loss of generality that SuppF consists of one function symbol $f[\tau_1, \ldots, \tau_{n-1} \mapsto \tau_n]$.

We distinguish the following operations for UNNEST:

- Let $\mathcal{T}'$ be derived from $\mathcal{T}$ by the replacement of a rule $\forall \bar{x} : P(\bar{t}) \leftarrow \varphi$ by $\forall \bar{x}, y : P(t_1, \ldots, t_{j-1}, y, t_{j+1}, \ldots, t_n) \leftarrow y = t_j \wedge \varphi$. Let $\mathcal{M}$ be a model of $\mathcal{T}$; we show that it is also a model of $\mathcal{T}'$. It suffices to show that every $P(\overline{d})$, element of $\mathcal{M}$ supported by the original rule in $\mathcal{T}$, is supported by the new rule in $\mathcal{T}'$. That $P(\overline{d})$ is supported means there is a variable assignment $\{\overline{x}/\overline{d}_x\}$ such that $P(\bar{t}\{\overline{x}/\overline{d}_x\}^{\mathcal{M}}) = P(\overline{d})$ and that $\varphi\{\overline{x}/\overline{d}_x\}^{\mathcal{M}} = \top$. But then $y = t_j \wedge \varphi$ is true in $\mathcal{M}$ under the variable assignment $\{\overline{x}/\overline{d}_x, y/t_j\{\overline{x}/\overline{d}_x\}^{\mathcal{M}}\}$. We can conclude from this observation that $\langle t_1, \ldots, t_{j-1}, y, t_{j+1}, \ldots, t_n \rangle \{\overline{x}/\overline{d}_x, y/t_j\{\overline{x}/\overline{d}_x\}^{\mathcal{M}}\}^{\mathcal{M}} = \bar{t}\{\overline{x}/\overline{d}_x\}^{\mathcal{M}} = \overline{d}$ and that $P(\overline{d})$ is supported by the new rule of $\mathcal{T}'$. Hence,

$\mathcal{M}$ is also a model of $\mathcal{T}'$. Similarly, we can show that every model of $\mathcal{T}'$ is a model of $\mathcal{T}$; this completes the proof for this case.

- Let $\mathcal{T}'$ be derived from $\mathcal{T}$ by the replacement of a formula $\varphi$ by $\exists y : \varphi[t/y] \wedge y = t$. Let $\mathcal{I}$ be an interpretation and let $\{\overline{x}/\overline{d}\}$ be a variable assignment for the free variables $\overline{x}$ of $\varphi$. If $\varphi\{\overline{x}/\overline{d}\}^{\mathcal{I}}$ is true then $\varphi[t/y] \wedge y = t$ is true in $\mathcal{I}$ for the variable assignment $\{\overline{x}/\overline{d}, y/t\{\overline{x}/\overline{d}\}^{I}\}$ and hence also $\exists y : \varphi[t/y] \wedge y = t$ is true in $\mathcal{I}$ for the variable assignment $\{\overline{x}/\overline{d}\}$. Similarly, one can show that $\varphi[t]$ is false under variable assignment $\{\overline{x}/\overline{d}\}$ in $\mathcal{I}$ iff $\exists y : \varphi[t/y] \wedge y = t$ is. Hence models are preserved by this rewriting.
- Let $\mathcal{T}'$ be derived from $\mathcal{T}$ by the replacement of $agg(\{\overline{x} : \varphi : t\})$ by $agg(\{\overline{x}, y : \varphi \wedge y = t : y\})$. Let $\{\overline{x}/\overline{d}\}$ be a variable assignment for which $\varphi$ is true in $I$. In this case $t\{\overline{x}/\overline{d}\}^{I}$ is added to the set to be aggregated[1]. It follows that $\varphi \wedge y = t$ is true for exactly one variable assignment extending $\{\overline{x}/\overline{d}\}$, namely $\{\overline{x}/\overline{d}, y/t\{\overline{x}/\overline{d}\}^{I}\}$. Hence, $y\{\overline{x}/\overline{d}, y/t\{\overline{x}/\overline{d}\}^{I}\}$ is added to the set to be aggregated. But $y\{\overline{x}/\overline{d}, y/t\{\overline{x}/\overline{d}\}^{I}\} = t\{\overline{x}/\overline{d}\}^{I}$ hence the same value is added. Also, for all variable assignments $\{\overline{x}/\overline{d}\}$ for which $\varphi$ is false, $\varphi \wedge y = t$ is false for all variable assignments extending $\{\overline{x}/\overline{d}\}$ with an assignment for $y$. Hence the rewriting preserves the value of the aggregate expression and the models of the theory.

Considering GRAPH, model preservation can be shown as follows. First, we observe that the interpretation of an $n$-ary predicate symbol $P$ is isomorphic that of an $n-1$-ary function symbol if for every $n-1$-ary tuple of domain elements $\overline{d}$ there is at most one domain element $d'$ such that $P(\overline{d} :: d')$ is true and (if $f$ is total) there is at least one such element for every $\overline{d}$ in the corresponding domain. This is exactly the restriction imposed by the sentences added to the theory by graphing. In combination with the output definition, this guarantees that in every model, $f$ and $G_f$ will be isomorphic. □

## A.2 Proof of Proposition 6.3.9

First, we introduce some notations. Given a term $t$ with free variable $\overline{x}$ and a variable assignment $\{\overline{x}/\overline{d}\}$, we denote with $t\{\overline{x}/\overline{d}\}^{\mathcal{I}}$ the interpretation of $t$ in $\mathcal{I}$ under that assignment (a domain element). Similar for a formula $\varphi$, $\varphi\{\overline{x}/\overline{d}\}^{\mathcal{I}}$ is the interpretation of $\varphi$ under that assignment (a truth value).

---

[1] Nothing is added when $t$ contains a function that is not defined for the variable assignment; for that variable assignment, the atom $y = t$ is false.

Our rewriting performs a number of operations to unnest and graph terms. As discussed in Proposition 4.2.2, this operation preserves models. Given that lemma, we can assume without loss of generality that the arguments of rule heads are distinct variables. Now, we can turn our attention to the replacement of predicate and function symbols by new ones in dep-reduce.

*Proof of Proposition 6.3.9.* We distinguish several cases.

- The functional dependency $d\,\langle P[\overline{\tau}], S, j\rangle$ holds in $\mathcal{T}$ with $\#(S) < n-1$ and $\mathcal{T}'$ is derived from $\mathcal{T}$ by replacing $P(\overline{t})$ everywhere by $P_r(\overline{t}|_{\{j\}^C}) \wedge t_j = f_d(\overline{t}|_S)$ and adding the function $f_d[\overline{t}|_S \mapsto \tau_j]$. Let $\mathcal{M}$ be a total interpretation of $\mathcal{T}$. Now construct a total interpretation $\mathcal{M}'$ of $\mathcal{T}'$ by copying $\mathcal{M}$ for all symbols except $P$ and by adding the atoms $P_r(\left.\overline{d}\right|_{\{j\}^C})$ and $f_d(\left.\overline{d}\right|_S) = \left.\overline{d}\right|_{\{j\}}$ for every atom $P(\overline{d})$ in $\mathcal{M}$. Note that $f_d$ defined in this way is a function because of the functional dependency in $\mathcal{T}$. The expressions $P(\overline{t})$ and $P_r(\overline{t}|_{\{j\}^C}) \wedge t_j = f_d(\overline{t}|_S)$ have the same free variables, say $\overline{x}$. Moreover, given the relationship between $\mathcal{M}$ and $\mathcal{M}'$, it holds that $P(\overline{t})\{\overline{x}/\overline{d}_x\}^{\mathcal{M}} \equiv (P_r(\overline{t}|_{\{j\}^C}) \wedge t_j = f_d(\overline{t}|_S))\{\overline{x}/\overline{d}_x\}^{\mathcal{M}'}$ for every variable assignment $\{\overline{x}/\overline{d}_x\}$ hence $\mathcal{M}$ is a model of $\mathcal{T}$ iff $\mathcal{M}'$ is a model of $\mathcal{T}'$. Now, let $\mathcal{M}$ be a model of $\mathcal{T}$ and $\mathcal{M}'$ the corresponding model of $\mathcal{T}'$. Adding the definition $\{\forall \overline{x} \in \overline{\tau} : P(\overline{x}) \leftarrow P_r(\overline{x}|_{[1,n-1]-j}) \wedge f_d(\overline{x}|_S) = x_j\}$ to $\mathcal{T}'$, as prescribed by Definition 6.3.2, $\mathcal{M}'$ is extended with every atom $P(\overline{d})$ that is true in $\mathcal{M}$ and both theories are strongly $voc(\mathcal{T})$-equivalent.

  When $P$ is a defined symbol, every rule $P(\overline{x}) \leftarrow \varphi$ is replaced by the rules $f_d(\overline{x}|_S) = x_j \leftarrow \varphi$ and $P_r(\overline{x}|_{\{j\}^C}) \leftarrow \varphi$ (Definition 6.3.7). Given the correspondence between $\mathcal{M}$ and $\mathcal{M}'$, clearly, an atom $P(\overline{d})$ is supported under $\mathcal{M}$ by the rule in $\mathcal{T}$ iff $P_r(\left.\overline{d}\right|_{\{j\}^C})$ and $f_d(\left.\overline{d}\right|_S) = d_j$ are supported under $\mathcal{M}'$ by the two rules in $\mathcal{T}'$. Hence both theories are also strongly $voc(\mathcal{T})$-equivalent in this case.

- The case of a functional dependency $d\,\langle P[\overline{\tau}], S, j\rangle$ in $\mathcal{T}$ with $\#(S) = n-1$ is very similar and the proof is omitted.

- The functional dependency $d\,\langle f[\overline{\tau} \mapsto \tau_n], S, j\rangle$ holds in $\mathcal{T}$ with $\#(S) < n-1$, $j \neq n$, $n \notin S$, and $\mathcal{T}'$ is derived from $\mathcal{T}$ by adding the functions $f_d[\overline{\tau}|_S \mapsto \tau_j]$ and $f_r[\overline{\tau}|_{\{j\}^C} \mapsto \tau_n]$ and replacing everywhere atoms $a[f(\overline{t})]$ by $a[f(\overline{t})/f_r(\overline{t}|_{\{j\}^C})] \wedge t_j = f_d(\overline{t}|_S)$. Also here, given a total interpretation

$\mathcal{M}$ of $\mathcal{T}$, we can construct a total interpretation $\mathcal{M}'$ of $\mathcal{T}'$ by copying the interpretations of all symbols but $f$ and extending it with $f_d(\overline{d}\big|_S) = d_j$ and $f_r(\overline{d}\big|_{\{j\}^C}) = d_n$ for every atom $f(\overline{d}) = d_n$ in $\mathcal{M}$. Note that $f_d$ and $f_r$ defined in this way are functions because $f$ is a function and the functional dependency holds in $\mathcal{T}$. Also here, we can argue that $a[f(\bar{t})]$ and $a[f(\bar{t})/f_r(\bar{t}|_{\{j\}^C})] \land t_j = f_d(\bar{t}|_S)$ have the same free variables $\overline{x}$ and hence the same truth value for every variable assignment $\{\overline{x}/\overline{d}_x\}$ under respectively $\mathcal{M}$ and $\mathcal{M}'$. Hence $\mathcal{M}$ is a model of $\mathcal{T}$ iff $\mathcal{M}'$ is a model of $\mathcal{T}'$. Adding the rule $\{\forall \overline{x}, x_n \in \overline{\tau}, \tau_n : f(\overline{x}) = x_n \leftarrow f_r(\overline{x}|_{\{j\}^C}) = x_n \land f_d(\overline{x}|_S) = x_j\}$ as prescribed by Definition 6.3.2, to $\mathcal{T}'$, $\mathcal{M}'$ is extended with every atom $f(\overline{d}) = d_n$ that is true in $\mathcal{M}$ and both theories are strongly $voc(\mathcal{T})$-equivalent.

When $f$ is a defined symbol, every rule $f(\overline{x}) = x_n \leftarrow \varphi$ is replaced by the rules $f_d(\overline{x}|_S) = x_j \leftarrow \varphi$ and $f_r(\overline{x}|_{\{j\}^C}) = x_n \leftarrow \varphi$ (Definition 6.3.7). Given the correspondence between $\mathcal{M}$ and $\mathcal{M}'$, clearly, an atom $f(\overline{d}) = d_n$ is supported under $\mathcal{M}$ by the rule in $\mathcal{T}$ iff $f_r(\overline{d}\big|_{\{j\}^C}) = d_n$ and $f_d(\overline{d}\big|_S) = d_j$ are supported under $\mathcal{M}'$ by the two rules in $\mathcal{T}'$. Hence both theories are strongly $voc(\mathcal{T})$-equivalent in this case.

- The case of a functional dependency $d\ \langle f[\overline{\tau} \mapsto \tau_n], S, j\rangle$ with $\#(S) = n - 1$ and $j = n$ is very similar to the previous case and the proof is omitted.

- Also the case where the preprocessing step replaces atoms of the form $x = f(\bar{t})$ by $G_f(\bar{t} :: x)$ with $G_f[\overline{\tau} :: \tau_n]$ a new predicate is very similar to the above cases and we also omit its proof.

It follows that rewriting a theory $\mathcal{T}$ according to Definitions 6.3.1, 6.3.7, and 6.3.2 produce a strongly $voc(\mathcal{T})$-equivalent theory. □

# B

# Complete Specifications

In this chapter, we provide complete IDP specifications of some of the problems presented in the main body of the text.

## B.1 2-D Square Packing

```
vocabulary V is {
    type id;            type nb;
    func width[->nb];   func breadth[->nb];
    pred size[id->nb];  func largest[->id];
    partial func xpos[id->nb];
    partial func ypos[id->nb];
}
vocabulary Vout is { includes V.xpos;  includes V.ypos }

theory T over V is {
    0=<xpos(id)=<width-size(id) & 0=<ypos(id)=<breadth-size(id);
    definition {
        leftOf(id1,id2)     <- xpos(id1)+size(id1)=<xpos(id2);
        below(id1,id2)      <- ypos(id1)+size(id1)=<ypos(id2);
        noOverlap(id1,id2) <- leftOf(id1,id2) | leftOf(id2,id1)
                              | below(id1,id2) | below(id2,id1);
    }

     xpos(largest)=min[->nb]; ypos(largest)=min[->nb];
     definition {
         largest=id1 <- !id2: id1~=id2 => size(id1)>=size(id2);
     }
}

term C over V is #({id: denotes(xpos(id))});

structure I over V is {
   id = {s1;s2;s3;s4;s5;s6;s7;s8;s9;s10};
   width = 1000;
   breadth = 500;
   size = {s1->325; ...};
}
procedure pack() is {
    m = minimize(T,I,C,Vout);
    print("Area = "..value(C,m));
    return m;
}
```

## B.2 Sudoku Generation

First, a generic part in which some knowledge about 2-D grids is modeled and a procedure to make an empty square grid of given size.

```
namespace grid is {
   vocabulary simplegridvoc is {
      type Row isa nat
      type Col isa nat
   }

   // Make a grid of size (nrrows x nrcolumns)
   procedure makegrid(nrrows, nrcolumns) is {
      local rows = range(1,nrrows)
      local cols = range(1,nrcolumns)
      local struct = newstructure(simplegridvoc,"grid")
      struct[simplegridvoc.Row.type] = rows
      struct[simplegridvoc.Col.type] = cols
      return struct
   }

   // Make a square grid of dimension dim
   procedure makesquaregrid(dim) is {
      return makegrid(dim,dim)
   }
}
```

Second, background knowledge about Sudoku puzzles and a procedure to make an empty Sudoku grid.

```
vocabulary sudvoc is {
   includes vocabulary grid.simplegridvoc
   type Num isa nat
   type Block isa nat
   func Sudoku[Row,Col -> Num]
   pred InBlock[Block,Row,Col]
}
theory sudtheo over sudvoc is {
   !r n: ?=1 c: Sudoku(r,c) = n;
   !c n: ?=1 r: Sudoku(r,c) = n;
   !b n: ?=1 r c: InBlock(b,r,c) & Sudoku(r,c) = n;
    definition {
      !c r b: InBlock(b,r,c) <- b = ((r-1) - ((r-1)%3))
                                        + ((c-1) - ((c-1)%3))/3 + 1
   }
}
procedure createSudokugrid(n) is {
   local g = grid.makesquaregrid(n)
   setvocabulary(g,sudvoc)
   g[sudvoc.Num.type] = range(1,n)
   g[sudvoc.Block.type] = range(1,n)
   return g
}
```

The effective code to generate a new Sudoku puzzle.

```
procedure generate(size) is {
    local g = createSudokugrid(size)

    stdoptions.nbmodels = 2
    local currsols = modelexpand(sudtheo,g)
    while #currsols > 1 do
        repeat
            col = math.random(1,size)
            row = math.random(1,size)
            num = currsols[1][sudvoc.Sudoku](row,col)
        until num ~= currsols[2][sudvoc.Sudoku](row,col)

        maketrue(g[sudvoc.Sudoku].graph,{row,col,num})
        currsols = modelexpand(sudtheo,g)
    end

    -- try to remove elements
    local change = true
    while change do
        change = false
        local cttab = { }
        for t in tuples(g[sudvoc.Sudoku].graph.ct) do
            local pos = math.random(1,math.max(#cttab,1))
            table.insert(cttab,pos,t)
        end
        for i,v in ipairs(cttab) do
            makeunknown(g[sudvoc.Sudoku].graph,v)
            currsols = modelexpand(sudtheo,g)
            if #currsols == 1 then
                change = true
                showtext(g)
                break
            else
                maketrue(g[sudvoc.Sudoku].graph,v)
            end
        end
    end
    return g
}
```

Finally, a procedure to print an ASCII version of the Sudoku, given a Sudoku puzzle structure.

```
procedure showtext(struc) is {
   grid = struc[sudvoc.Sudoku].graph.ct
   local table = { }
   for row = 1,9 do table[row] = { } end
   for tuple in tuples(grid) do
      table[tuple [1]][ tuple [2]] = tuple[3]
   end
   for row = 1,9 do
      local str = ""
      for col = 1,9 do
         local content = table[row][col]
         if content then str = str.."  "..content.."  "
         else str = str.."  .  "
         end
      end
      print(str)
   end
   print()
}
```

# Bibliography

[Aavani, 2014] Aavani, A. (2014). *Enfragmo: A System for Grounding Extended First-Order Logic to SAT*. PhD thesis, Faculty of Applied Sciences, Simon Fraser University, Vancouver, Canada.

[Aavani et al., 2012] Aavani, A., Wu, X. N., Tasharrofi, S., Ternovska, E., and Mitchell, D. G. (2012). Enfragmo: A system for modelling and solving search problems with logic. In Bjørner, N. and Voronkov, A., editors, *LPAR*, volume 7180 of *LNCS*, pages 15–22. Springer.

[Abrial, 1996] Abrial, J.-R. (1996). *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA.

[Abrial et al., 2010] Abrial, J.-R., Butler, M. J., Hallerstede, S., Hoang, T. S., Mehta, F., and Voisin, L. (2010). Rodin: An open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466.

[Aloul et al., 2006] Aloul, F., Sakallah, K., and Markov, I. (2006). Efficient symmetry breaking for Boolean satisfiability. *IEEE Transactions on Computers*, 55(5):549–558.

[Alviano et al., 2013] Alviano, M., Calimeri, F., Charwat, G., Dao-Tran, M., Dodaro, C., Ianni, G., Krennwallner, T., Kronegger, M., Oetsch, J., Pfandler, A., Pührer, J., Redl, C., Ricca, F., Schneider, P., Schwengerer, M., Spendier, L. K., Wallner, J. P., and Xiao, G. (2013). The fourth answer set programming competition: Preliminary report. In Cabalar, P. and Son, T. C., editors, *LPNMR*, volume 8148 of *LNCS*, pages 42–53. Springer.

[Amadini et al., 2013] Amadini, R., Gabbrielli, M., and Mauro, J. (2013). Features for building CSP portfolio solvers. *CoRR*, abs/1308.0227.

[Andres et al., 2012] Andres, B., Kaufmann, B., Matheis, O., and Schaub, T. (2012). Unsatisfiability-based optimization in clasp. In [Dovier and Santos Costa, 2012], pages 211–221.

[Andrews et al., 2012] Andrews, T., Blockeel, H., Bogaerts, B., Bruynooghe, M., Denecker, M., De Pooter, S., Macé, C., and Ramon, J. (2012). Analyzing manuscript traditions using constraint-based data mining. In *COmbining COnstraint solving with MIning and LEarning (CoCoMile)*.

[Andrews and Macé, 2012] Andrews, T. and Macé, C. (2012). Beyond the tree of texts: Building an empirical model of scribal variation through graph analysis of texts and stemmata. *Literary and Linguistic Computing*, 28(4):504–521.

[Apt, 2003] Apt, K. R. (2003). *Principles of Constraint Programming*. Cambridge University Press.

[Armstrong, 1974] Armstrong, W. W. (1974). Dependency structures of data base relationships. *IFIP Congress*, pages 580–583.

[Asbach et al., 2009] Asbach, L., Dorndorf, U., and Pesch, E. (2009). Analysis, modeling and solution of the concrete delivery problem. *European Journal of Operational Research*, 193(3):820 – 835.

[Aziz et al., 2013] Aziz, R. A., Chu, G., and Stuckey, P. J. (2013). Stable model semantics for founded bounds. *TPLP*, 13(4-5):517–532.

[Balduccini, 2011] Balduccini, M. (2011). Industrial-size scheduling with ASP+CP. In [Delgrande and Faber, 2011], pages 284–296.

[Balduccini, 2013] Balduccini, M. (2013). ASP with non-herbrand partial functions: A language and system for practical use. *TPLP*, 13(4-5):547–561.

[Balduccini and Lierler, 2013] Balduccini, M. and Lierler, Y. (2013). Integration schemas for constraint answer set programming: a case study. *TPLP*, 13(4-5-Online-Supplement).

[Baral, 2003] Baral, C. (2003). *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

[Baret et al., 2006] Baret, P., Macé, C., Robinson, P., Peersman, C., Mazza, R., Noret, J., Wattel, E., M., V. M., P., R., Lantin, A., Canettieri, P., Loreto, V., Windram, H., Spencer, M., Howe, C., Albu, M., and Dress, A. (2006). Testing methods on an artificially created textual tradition. In *The evolution of* *texts: Confronting stemmatological and genetical methods*, pages 255–283. Istituti editoriali e poligrafici internazionali, Pisa.

[Bartholomew and Lee, 2012] Bartholomew, M. and Lee, J. (2012). Stable models of formulas with intensional functions. In [Brewka et al., 2012].

[Baumgartner et al., 2012] Baumgartner, P., Pelzer, B., and Tinelli, C. (2012). Model evolution with equality - revised and implemented. *J. Symb. Comput.*, 47(9):1011–1045.

[Biere et al., 2009] Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors (2009). *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

[Blockeel et al., 2012] Blockeel, H., Bogaerts, B., Bruynooghe, M., De Cat, B., De Pooter, S., Denecker, M., Labarre, A., Ramon, J., and Verwer, S. (2012). Modeling machine learning and data mining problems with FO(·). In [Dovier and Santos Costa, 2012], pages 14–25.

[Blockeel et al., 2013] Blockeel, H., Bruynooghe, M., Bogaerts, B., De Cat, B., De Pooter, S., Denecker, M., Labarre, A., Ramon, J., and Verwer, S. (2013). Predicate logic as a modeling language: Modeling and solving some machine learning and data mining problems with IDP3. *CoRR*, abs/1309.6883. To appear in TPLP.

[Brewka et al., 2012] Brewka, G., Eiter, T., and McIlraith, S. A., editors (2012). *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012*. AAAI Press.

[Brewka et al., 2011] Brewka, G., Eiter, T., and Truszczyński, M. (2011). Answer set programming at a glance. *CACM*, 54(12):92–103.

[Bruttomesso et al., 2007] Bruttomesso, R., Cimatti, R., Franzén, A., Griggio, A., Hanna, Z., Nadel, E., Palti, A., and Sebastiani, R. (2007). A lazy and layered SMT(BV) solver for hard industrial verification problems. In *Computer Aided Verification (CAV), LNCS*. Springer.

[Bruynooghe, 1981] Bruynooghe, M. (1981). Solving combinatorial search problems by intelligent backtracking. *Inf. Process. Lett.*, 12(1):36–39.

[Buchholz et al., 1981] Buchholz, W., Feferman, S., Pohlers, W., and Sieg, W. (1981). *Iterated Inductive Definitions and Subsystems of Analysis : Recent Proof-Theoretical Studies*, volume 897 of *Lecture Notes in Mathematics*. Springer.

[Calimeri et al., 2012] Calimeri, F., Ianni, G., and Ricca, F. (2012). The third open answer set programming competition. *CoRR*, abs/1206.3111.

[Calimeri et al., 2011] Calimeri, F., Ianni, G., Ricca, F., Alviano, M., Bria, A., Catalano, G., Cozza, S., Faber, W., Febbraro, O., Leone, N., Manna, M., Martello, A., Panetta, C., Perri, S., Reale, K., Santoro, M. C., Sirianni, M., Terracina, G., and Veltri, P. (2011). The third answer set programming system competition: Preliminary report of the system competition track. In *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 388–403. Springer.

[Chen and Warren, 1996] Chen, W. and Warren, D. S. (1996). Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74.

[Claessen and Sörensson, 2003] Claessen, K. and Sörensson, N. (2003). New techniques that improve MACE-style model finding. In *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*.

[Codish et al., 2011] Codish, M., Fekete, Y., Fuhs, C., and Schneider-Kamp, P. (2011). Optimal base encodings for pseudo-Boolean constraints. In Abdulla, P. A. and Leino, K. R. M., editors, *TACAS*, volume 6605 of *LNCS*, pages 189–204. Springer.

[Dal Palù et al., 2009a] Dal Palù, A., Dovier, A., Pontelli, E., and Rossi, G. (2009a). Answer set programming with constraints using lazy grounding. In [Hill and Warren, 2009], pages 115–129.

[Dal Palù et al., 2009b] Dal Palù, A., Dovier, A., Pontelli, E., and Rossi, G. (2009b). GASP: Answer set programming with lazy grounding. *Fundam. Inform.*, 96(3):297–322.

[Dao-Tran et al., 2012] Dao-Tran, M., Eiter, T., Fink, M., Weidinger, G., and Weinzierl, A. (2012). Omiga : An open minded grounding on-the-fly answer set solver. In del Cerro, L. F., Herzig, A., and Mengin, J., editors, *JELIA*, volume 7519 of *LNCS*, pages 480–483. Springer.

[De Cat et al., 2014] De Cat, B., Bogaerts, B., Bruynooghe, M., and Denecker, M. (2014). Predicate logic as a modelling language: The IDP system. *CoRR*, abs/1401.6312.

[De Cat et al., 2013a] De Cat, B., Bogaerts, B., Devriendt, J., and Denecker, M. (2013a). Model expansion in the presence of function symbols using constraint programming. In *ICTAI*, pages 1068–1075. IEEE.

[De Cat and Bruynooghe, 2013] De Cat, B. and Bruynooghe, M. (2013). Detection and exploitation of functional dependencies for model generation. *Theory and Practice of Logic Programming (TPLP)*, 13(4-5):471–485.

[De Cat and Denecker, 2010] De Cat, B. and Denecker, M. (2010). DPLL(Agg): An efficient SMT module for aggregates. In Mitchell, D. and Ternovska, E., editors, *Third Workshop on Logic and Search, 2010*.

[De Cat et al., 2012] De Cat, B., Denecker, M., and Stuckey, P. J. (2012). Lazy model expansion by incremental grounding. In [Dovier and Santos Costa, 2012], pages 201–211.

[De Cat et al., 2013b] De Cat, B., Jansen, J., and Janssens, G. (2013b). IDP3: Combining symbolic and ground reasoning for model generation.

[de Moura and Bjørner, 2008] de Moura, L. M. and Bjørner, N. (2008). Z3: An efficient SMT solver. In Ramakrishnan, C. R. and Rehof, J., editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer.

[De Pooter et al., 2011] De Pooter, S., Wittocx, J., and Denecker, M. (2011). A prototype of a knowledge-based programming environment. In Tompits, H., Abreu, S., Oetsch, J., Pührer, J., Seipel, D., Umeda, M., and Wolf, A., editors, *INAP/WLP*, volume 7773 of *Lecture Notes in Computer Science*, pages 279–286. Springer.

[Decroix et al., 2013] Decroix, K., Lapon, J., De Decker, B., and Naessens, V. (2013). A formal approach for inspecting privacy and trust in advanced electronic services. In Jürjens, J., Livshits, B., and Scandariato, R., editors, *ESSoS*, volume 7781 of *LNCS*, pages 155–170. Springer.

[Delgrande and Faber, 2011] Delgrande, J. P. and Faber, W., editors (2011). *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, volume 6645 of *LNCS*. Springer.

[Denecker, 1998] Denecker, M. (1998). The well-founded semantics is the principle of inductive definition. In Dix, J., del Cerro, L. F., and Furbach, U., editors, *JELIA*, volume 1489 of *LNCS*, pages 1–16. Springer.

[Denecker et al., 2001] Denecker, M., Bruynooghe, M., and Marek, V. W. (2001). Logic programming revisited: Logic programs as inductive definitions. *ACM Transactions on Computational Logic (TOCL)*, 2(4):623–654.

[Denecker and De Schreye, 1993] Denecker, M. and De Schreye, D. (1993). Justification semantics: A unifying framework for the semantics of logic programs. In Pereira, L. M. and Nerode, A., editors, *LPNMR*, pages 365–379. MIT Press.

[Denecker et al., 2012] Denecker, M., Lierler, Y., Truszczynsky, M., and Vennekens, J. (2012). A Tarskian informal semantics for answer set programming. In Dovier, A. and Santos Costa, V., editors, *Technical Communications of the 28th International Conference on Logic Programming,*, pages 277–289. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[Denecker and Ternovska, 2008] Denecker, M. and Ternovska, E. (2008). A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic (TOCL)*, 9(2):14:1–14:52.

[Denecker and Vennekens, 2007] Denecker, M. and Vennekens, J. (2007). Well-founded semantics and the algebraic theory of non-monotone inductive definitions. In Baral, C., Brewka, G., and Schlipf, J. S., editors, *LPNMR*, volume 4483 of *LNCS*, pages 84–96. Springer.

[Denecker and Vennekens, 2008] Denecker, M. and Vennekens, J. (2008). Building a knowledge base system for an integration of logic programming and classical logic. In [García de la Banda and Pontelli, 2008], pages 71–76.

[Denecker and Vennekens, 2014] Denecker, M. and Vennekens, J. (2014). The well-founded semantics is the principle of inductive definition, revisited. In *International Conference on Principles of Knowledge Representation and Reasoning*. To appear.

[Denecker et al., 2009] Denecker, M., Vennekens, J., Bond, S., Gebser, M., and Truszczyński, M. (2009). The second answer set programming competition. In [Erdem et al., 2009], pages 637–654.

[Detlefs et al., 2005] Detlefs, D., Nelson, G., and Saxe, J. B. (2005). Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473.

[Devriendt et al., 2012] Devriendt, J., Bogaerts, B., de Cat, B., Denecker, M., and Mears, C. (2012). Symmetry propagation: Improved dynamic symmetry breaking in SAT. In *ICTAI*, pages 49–56. IEEE.

[Dovier and Santos Costa, 2012] Dovier, A. and Santos Costa, V., editors (2012). *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*, volume 17 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[Doyle, 1979] Doyle, J. (1979). A truth maintenance system. *Artif. Intell.*, 12(3):231–272.

[Drescher and Walsh, 2011a] Drescher, C. and Walsh, T. (2011a). Conflict-driven constraint answer set solving with lazy nogood generation. In Burgard, W. and Roth, D., editors, *AAAI*. AAAI Press.

[Drescher and Walsh, 2011b] Drescher, C. and Walsh, T. (2011b). Translation-based constraint answer set solving. In Walsh, T., editor, *IJCAI*, pages 2596–2601. IJCAI/AAAI.

[East and Truszczyński, 2006] East, D. and Truszczyński, M. (2006). Predicate-calculus-based logics for modeling and solving search problems. *ACM Transactions on Computational Logic (TOCL)*, 7(1):38–83.

[Eén and Sörensson, 2003] Eén, N. and Sörensson, N. (2003). An extensible SAT-solver. In Giunchiglia, E. and Tacchella, A., editors, *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer.

[Eiter et al., 2005] Eiter, T., Ianni, G., Schindlauer, R., and Tompits, H. (2005). A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In Kaelbling, L. P. and Saffiotti, A., editors, *IJCAI*, pages 90–96. Professional Book Center.

[Enderton, 2001] Enderton, H. B. (2001). *A Mathematical Introduction To Logic*. Academic Press, second edition.

[Erdem et al., 2009] Erdem, E., Lin, F., and Schaub, T., editors (2009). *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *LNCS*. Springer.

[Faber et al., 2012] Faber, W., Leone, N., and Perri, S. (2012). The intelligent grounder of DLV. In Erdem, E., Lee, J., Lierler, Y., and Pearce, D., editors, *Correct Reasoning*, volume 7265 of *Lecture Notes in Computer Science*, pages 247–264. Springer Berlin Heidelberg.

[Franco and Martin, 2009] Franco, J. and Martin, J. (2009). *A History of Satisfiability*, chapter 1, pages 3–74. Volume 185 of [Biere et al., 2009].

[Frisch and Stuckey, 2009] Frisch, A. M. and Stuckey, P. J. (2009). The proper treatment of undefinedness in constraint languages. In Gent, I., editor, *Principles and Practice of Constraint Programming - CP 2009*, volume 5732 of *Lecture Notes in Computer Science*, pages 367–382. Springer Berlin Heidelberg.

[García de la Banda and Pontelli, 2008] García de la Banda, M. and Pontelli, E., editors (2008). *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, volume 5366 of *LNCS*. Springer.

[Ge and de Moura, 2009] Ge, Y. and de Moura, L. M. (2009). Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Bouajjani, A. and Maler, O., editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 306–320. Springer Berlin Heidelberg.

[Gebser et al., 2008] Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Thiele, S. (2008). Engineering an incremental ASP solver. In [García de la Banda and Pontelli, 2008], pages 190–205.

[Gebser et al., 2011a] Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2011a). Challenges in answer set solving. In Balduccini, M. and Son, T. C., editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, volume 6565 of *Lecture Notes in Computer Science*, pages 74–90. Springer.

[Gebser et al., 2012a] Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2012a). *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

[Gebser et al., 2011b] Gebser, M., Kaminski, R., König, A., and Schaub, T. (2011b). Advances in Gringo series 3. In [Delgrande and Faber, 2011], pages 345–351.

[Gebser et al., 2009a] Gebser, M., Kaminski, R., Ostrowski, M., Schaub, T., and Thiele, S. (2009a). On the input language of ASP grounder Gringo. In [Erdem et al., 2009], pages 502–508.

[Gebser et al., 2009b] Gebser, M., Kaufmann, B., and Schaub, T. (2009b). Solution enumeration for projected Boolean search problems. In van Hoeve, W. J. and Hooker, J. N., editors, *CPAIOR*, volume 5547 of *LNCS*, pages 71–86. Springer.

[Gebser et al., 2012b] Gebser, M., Kaufmann, B., and Schaub, T. (2012b). Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89.

[Gebser et al., 2009c] Gebser, M., Ostrowski, M., and Schaub, T. (2009c). Constraint answer set solving. In [Hill and Warren, 2009], pages 235–249.

[Gecode Team, 2013] Gecode Team (2013). Gecode: Generic constraint development environment. Available from `http://www.gecode.org`.

[Gelfond and Lifschitz, 1988] Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In Kowalski, R. A. and Bowen, K. A., editors, *ICLP/SLP*, pages 1070–1080. MIT Press.

[Gelfond et al., 2008] Gelfond, M., Mellarkod, V. S., and Zhang, Y. (2008). Systems integrating answer set programming and constraint programming. In Denecker, M., editor, *Second Workshop on Logic and Search, 2008*, pages 145–152.

[Green et al., 2012] Green, T. J., Aref, M., and Karvounarakis, G. (2012). Logicblox, platform and language: A tutorial. In Barceló, P. and Pichler, R., editors, *Datalog*, volume 7494 of *LNCS*, pages 1–8. Springer.

[Guns et al., 2013] Guns, T., Dries, A., Tack, G., Nijssen, S., and De Raedt, L. (2013). MiningZinc: A modeling language for constraint-based mining. In Rossi, F., editor, *IJCAI*. IJCAI/AAAI.

[Hähnle, 2001] Hähnle, R. (2001). Tableaux and related methods. In Robinson, J. A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, pages 100–178. Elsevier and MIT Press.

[Heyman, 2013] Heyman, T. (2013). *A Formal Analysis Technique for Secure Software Architectures*. PhD thesis, Department of Computer Science, KU Leuven.

[Hill and Warren, 2009] Hill, P. M. and Warren, D. S., editors (2009). *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, volume 5649 of *LNCS*. Springer.

[IDPDraw, 2012] IDPDraw (2012). IDPDraw: Finite structure visualization. `http://dtai.cs.kuleuven.be/krr/software/visualisation`.

[Ierusalimschy et al., 1996] Ierusalimschy, R., Henrique de Figueiredo, L., and Celes, W. (1996). Lua – an extensible extension language. *Software: Practice and Experience*, 26(6):635–652.

[Jackson, 2002] Jackson, D. (2002). Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM'02)*, 11(2):256–290.

[Jackson et al., 2013] Jackson, E. K., Bjorner, N., and Schulte, W. (2013). Open-world logic programs: A new foundation for formal specifications. *Tech. Report*, pages 1–19.

[Janhunen, 2004] Janhunen, T. (2004). Representing normal programs with clauses. In de Mántaras, R. L. and Saitta, L., editors, *ECAI*, pages 358–362. IOS Press.

[Janhunen et al., 2009] Janhunen, T., Niemelä, I., and Sevalnev, M. (2009). Computing stable models via reductions to difference logic. In [Erdem et al., 2009], pages 142–154.

[Jansen et al., 2013] Jansen, J., Jorissen, A., and Janssens, G. (2013). Compiling input$*$ FO$(\cdot)$ inductive definitions into tabled Prolog rules for IDP$^3$. *Theory and Practice of Logic Programming (TPLP)*, 13(4-5):691–704.

[Karp, 1972] Karp, R. (1972). Reducibility among combinatorial problems. In Miller, R. and Thatcher, J., editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press.

[Kimmig et al., 2011] Kimmig, A., Demoen, B., De Raedt, L., Santos Costa, V., and Rocha, R. (2011). On the implementation of the probabilistic logic programming language ProbLog. *TPLP*, 11(2-3):235–262.

[Kleene, 1952] Kleene, S. C. (1952). *Introduction to Metamathematics*. Van Nostrand.

[Kovács and Voronkov, 2013] Kovács, L. and Voronkov, A. (2013). First-order theorem proving and Vampire. In Sharygina, N. and Veith, H., editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer Berlin Heidelberg.

[Kowalski, 1974] Kowalski, R. A. (1974). Predicate logic as programming language. In *IFIP Congress*, pages 569–574.

[Labarre and Verwer, 2014] Labarre, A. and Verwer, S. (2014). Merging partially labelled trees: Hardness and an efficient practical solution. *IEEE/ACM transactions on Computational Biology and Bioinformatics*. To appear.

[Lefèvre and Nicolas, 2009] Lefèvre, C. and Nicolas, P. (2009). The first version of a new ASP solver : ASPeRiX. In [Erdem et al., 2009], pages 522–527.

[Leone et al., 2006] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. (2006). The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3):499–562.

[Lierler, 2012] Lierler, Y. (2012). On the relation of constraint answer set programming languages and algorithms. In Hoffmann, J. and Selman, B., editors, *AAAI*. AAAI Press.

[Lifschitz, 2012] Lifschitz, V. (2012). Logic programs with intensional functions. In [Brewka et al., 2012].

[linq, ] linq. Linq (language-integrated query. `http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx`.

[Liu et al., 2012] Liu, G., Janhunen, T., and Niemelä, I. (2012). Answer set programming via mixed integer programming. In [Brewka et al., 2012].

[Lodi et al., 2002] Lodi, A., Martello, S., and Monaci, M. (2002). Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241 – 252.

[Mariën, 2009] Mariën, M. (2009). *Model Generation for ID-Logic*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium.

[Mariën et al., 2004] Mariën, M., Gilis, D., and Denecker, M. (2004). On the relation between ID-Logic and answer set programming. In Alferes, J. J. and Leite, J. A., editors, *JELIA*, volume 3229 of *LNCS*, pages 108–120. Springer.

[Mariën et al., 2008] Mariën, M., Wittocx, J., Denecker, M., and Bruynooghe, M. (2008). SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In Kleine Büning, H. and Zhao, X., editors, *SAT*, volume 4996 of *LNCS*, pages 211–224. Springer.

[Marques Silva et al., 2009] Marques Silva, J. P., Lynce, I., and Malik, S. (2009). Conflict-driven clause learning SAT solvers. In [Biere et al., 2009], pages 131–153.

[Marriott et al., 2008] Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P. J., Garcia de la Banda, M., and Wallace, M. (2008). The design of the Zinc modelling language. *Constraints*, 13(3):229–267.

[Mears et al., 2008] Mears, C., García de la Banda, M. J., Wallace, M., and Demoen, B. (2008). A novel approach for detecting symmetries in CSP models. In Perron, L. and Trick, M. A., editors, *CPAIOR*, volume 5015 of *LNCS*, pages 158–172. Springer.

[Mellarkod et al., 2008] Mellarkod, V. S., Gelfond, M., and Zhang, Y. (2008). Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, 53(1-4):251–287.

[Mendonça de Moura and Bjørner, 2011] Mendonça de Moura, L. and Bjørner, N. (2011). Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77.

[Metodi and Codish, 2012] Metodi, A. and Codish, M. (2012). Compiling finite domain constraints to SAT with BEE. *TPLP*, 12(4-5):465–483.

[Mitchell and Ternovska, 2005] Mitchell, D. G. and Ternovska, E. (2005). A framework for representing and solving NP search problems. In Veloso, M. M. and Kambhampati, S., editors, *AAAI*, pages 430–435. AAAI Press / The MIT Press.

[Mitchell et al., 2006] Mitchell, D. G., Ternovska, E., Hach, F., and Mohebali, R. (2006). Model expansion as a framework for modelling and solving search problems. Technical Report TR 2006-24, Simon Fraser University, Canada.

[Moskewicz et al., 2001] Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *DAC'01*, pages 530–535. ACM.

[Nelson and Oppen, 1980] Nelson, G. and Oppen, D. C. (1980). Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364.

[Nemhauser and Wolsey, 1988] Nemhauser, G. L. and Wolsey, L. A. (1988). *Integer and Combinatorial Optimization*. John Wiley and Sons, New York.

[Nethercote et al., 2007] Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., and Tack, G. (2007). Minizinc: Towards a standard CP modelling language. In Bessiere, C., editor, *CP'07*, volume 4741 of *LNCS*, pages 529–543. Springer.

[Nieuwenhuis and Oliveras, 2007] Nieuwenhuis, R. and Oliveras, A. (2007). Fast congruence closure and extensions. *Inf. Comput.*, 205(4):557–580.

[Nightingale et al., 2013] Nightingale, P., Gent, I. P., Jefferson, C., and Miguel, I. (2013). Short and long supports for constraint propagation. *J. Artif. Intell. Res. (JAIR)*, 46:1–45.

[Ostrowski and Schaub, 2012] Ostrowski, M. and Schaub, T. (2012). ASP modulo CSP: The clingcon system. *TPLP*, 12(4-5):485–503.

[Patterson et al., 2007] Patterson, M., Liu, Y., Ternovska, E., and Gupta, A. (2007). Grounding for model expansion in k-guarded formulas with inductive definitions. In Veloso, M. M., editor, *IJCAI*, pages 161–166.

[Pelov, 2004] Pelov, N. (2004). *Semantics of Logic Programs with Aggregates*. PhD thesis, K.U.Leuven, Leuven, Belgium.

[Pelov et al., 2007] Pelov, N., Denecker, M., and Bruynooghe, M. (2007). Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming (TPLP)*, 7(3):301–353.

[Pelov and Ternovska, 2005] Pelov, N. and Ternovska, E. (2005). Reducing inductive definitions to propositional satisfiability. In Gabbrielli, M. and Gupta, G., editors, *ICLP*, volume 3668 of *LNCS*, pages 221–234. Springer.

[Reiter, 1980] Reiter, R. (1980). Equality and domain closure in first-order databases. *Journal of the ACM*, 27(2):235–249.

[Reiter, 1982] Reiter, R. (1982). Towards a logical reconstruction of relational database theory. In Brodie, M. L., Mylopoulos, J., and Schmidt, J. W., editors, *On Conceptual Modelling (Intervale)*, pages 191–233. Springer.

[Rendl et al., 2009] Rendl, A., Miguel, I., Gent, I. P., and Gregory, P. (2009). Common subexpressions in constraint models of planning problems. In Bulitko, V. and Beck, J. C., editors, *SARA*. AAAI.

[Riedel, 2009] Riedel, S. (2009). Cutting plane MAP inference for Markov logic. In *Workshop Statistical Relational Learning 2009*.

[Roos and Heikkilä, 2009] Roos, T. and Heikkilä, T. (2009). Evaluating methods for computer-assisted stemmatology using artificial benchmark data sets. *Literary and Linguistic Computing*, 24(4):417–433.

[Rümmer, 2008] Rümmer, P. (2008). A constraint sequent calculus for first-order logic with linear integer arithmetic. In Cervesato, I., Veith, H., and Voronkov, A., editors, *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 274–289. Springer.

[Russell, 1905] Russell, B. (1905). On denoting. *Mind*, 14(56):479–493.

[Saptawijaya and Pereira, 2013] Saptawijaya, A. and Pereira, L. M. (2013). Towards practical tabled abduction in logic programs. In Correia, L., Reis, L. P., and Cascalho, J., editors, *EPIA*, volume 8154 of *LNCS*, pages 223–234. Springer.

[Shanahan, 1997] Shanahan, M. (1997). *Solving the Frame Problem - a Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press.

[Singla and Domingos, 2006] Singla, P. and Domingos, P. (2006). Memory-efficient inference in relational domains. In Gil, Y. and Mooney, R. J., editors, *AAAI*, pages 488–493. AAAI Press.

[Stuckey, 2010] Stuckey, P. J. (2010). Lazy clause generation: Combining the power of SAT and CP (and MIP?) solving. In *CPAIOR*, pages 5–9.

[Stuckey and Tack, 2013] Stuckey, P. J. and Tack, G. (2013). Minizinc with functions. In Gomes, C. and Sellmann, M., editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 7874 of *Lecture Notes in Computer Science*, pages 268–283. Springer Berlin Heidelberg.

[Sutcliffe, 2009] Sutcliffe, G. (2009). The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362.

[Sutcliffe, 2012] Sutcliffe, G. (2012). The CADE-23 automated theorem proving system competition - CASC-23. *AI Commun.*, 25(1):49–63.

[Sutcliffe, 2013] Sutcliffe, G. (2013). The 6th IJCAR automated theorem proving system competition - CASC-J6. *AI Commun.*, 26(2):211–223.

[Syrjänen, 1998] Syrjänen, T. (1998). Implementation of local grounding for logic programs with stable model semantics. Technical Report B18, Helsinki University of Technology, Finland.

[Tamura et al., 2009] Tamura, N., Taga, A., Kitagawa, S., and Banbara, M. (2009). Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272.

[Thielscher, 2000] Thielscher, M. (2000). Representing the knowledge of a robot. In Cohn, A. G., Giunchiglia, F., and Selman, B., editors, *KR*, pages 109–120. Morgan Kaufmann.

[Timpanaro, 2005] Timpanaro, S. (2005). *The Genesis of Lachmann's Method*. University of Chicago Press. Translated by Most, G.W.

[Torlak et al., 2008] Torlak, E., Chang, F. S.-H., and Jackson, D. (2008). Finding minimal unsatisfiable cores of declarative specifications. In Cuéllar, J., Maibaum, T. S. E., and Sere, K., editors, *FM*, volume 5014 of *LNCS*, pages 326–341. Springer.

[Torlak and Jackson, 2007] Torlak, E. and Jackson, D. (2007). Kodkod: A relational model finder. In Grumberg, O. and Huth, M., editors, *TACAS*, volume 4424 of *LNCS*, pages 632–647. Springer.

[Tseitin, 1968] Tseitin, G. S. (1968). On the complexity of derivation in the propositional calculus, *Zapiski nauchnykh seminarov. LOMI*, 8:234–259. English translation of this volume: Studies in Constructive Mathematics and Mathematical Logic, Part 2, A. O. Slisenko, eds. Consultants Bureau, N.Y., 1970, pp. 115-125.

[Vaezipoor et al., 2011] Vaezipoor, P., Mitchell, D., and Mariën, M. (2011). Lifted unit propagation for effective grounding. *CoRR*, abs/1109.1317.

[van Fraassen, 1966] van Fraassen, B. (1966). Singular terms, truth-value gaps and free logic. *Journal of Philosophy*, 63(17):481–495.

[Van Gelder, 1993] Van Gelder, A. (1993). The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47(1):185–221.

[Van Gelder et al., 1991] Van Gelder, A., Ross, K. A., and Schlipf, J. S. (1991). The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650.

[Vennekens et al., 2007] Vennekens, J., Mariën, M., Wittocx, J., and Denecker, M. (2007). Predicate introduction for logics with a fixpoint semantics. Part I: Logic programming. *Fundamenta Informaticae*, 79(1-2):187–208.

[Vlaeminck et al., 2009] Vlaeminck, H., Vennekens, J., and Denecker, M. (2009). A logical framework for configuration software. In Porto, A. and López-Fraguas, F. J., editors, *PPDP*, pages 141–148. ACM.

[Weidenbach et al., 2009] Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., and Wischnewski, P. (2009). SPASS version 3.5. In Schmidt, R. A., editor, *CADE*, volume 5663 of *LNCS*, pages 140–145. Springer.

[Wittocx, 2010] Wittocx, J. (2010). *Finite Domain and Symbolic Inference Methods for Extensions of First-Order Logic*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium.

[Wittocx et al., 2013] Wittocx, J., Denecker, M., and Bruynooghe, M. (2013). Constraint propagation for first-order logic and inductive definitions. *ACM Trans. Comput. Logic*, 14(3):17:1–17:45.

[Wittocx et al., 2010] Wittocx, J., Mariën, M., and Denecker, M. (2010). Grounding FO and FO(ID) with bounds. *Journal of Artificial Intelligence Research*, 38:223–269.

[Zhang and Yap, 2011] Zhang, Y. and Yap, R. H. C. (2011). Solving functional constraints by variable substitution. *TPLP*, 11(2-3):297–322.

# Curriculum Vitae

Broes De Cat was born on August 9th 1986, in Boom, Belgium. After high school, at the Sint-Theresia College in Kapelle-op-den-Bos, he started with the studies of Civil Engineering at the Katholieke Universiteit Leuven (KUL) in 2004. In 2009, he received the degree of Master of Engineering in Computer Science (option Artificial Intelligence, magna cum laude). His master's thesis, titled "Development of Model Revision Algorithms with Applications in Trainscheduling and Network Configuration", was supervised by Prof. Dr. Marc Denecker.

In September 2009, he joined the DTAI (Declaratieve Talen en Artificiële Intelligentie) group of the Department of Computer Science at KUL, to investigate inference engines for rich logics, under the supervision of prof. dr. Marc Denecker. From January 2010 onwards, his research was funded by a personal grant ("strategische onderzoeksbeurs") from the Agency for Innovation by Science and Technology in Flanders (IWT).

# List of Publications

All publications are available on `http://www.cs.kuleuven.be/publicaties/lirias/mypubs.php?unum=U0063917`.

## Journal and Book Articles

- B. De Cat and M. Bruynooghe. "Detection and exploitation of functional dependencies for model generation". In: Theory and Practice of Logic Programming, volume 13, issue 4-5, pages 471-485, 2013.
- P. Hou, B. De Cat and M. Denecker. "FO(FD): Extending classical logic with rule-based fixpoint definitions". In: Theory and Practice of Logic Programming, volume 10, issue 4-6, pages 581-596, 2010.
- B. De Cat, M. Denecker, P. Stuckey. "Interleaving grounding and search". Submitted to: Journal of A.I. Research.
- B. De Cat, B. Bogaerts, M. Bruynooghe and M. Denecker. "Predicate logic as a modeling language: the IDP3 system". To be published in M. Kifer and A. Liu, "Declarative Logic Programming: Theory, Systems, and Applications".

## Peer-reviewed Articles at Conferences and Workshops

- B. De Cat, B. Bogaerts, M. Denecker and J. Devriendt. "Model expansion in the presence of function symbols using constraint programming". In: International Conference on Tools For Artificial Intelligence, Washington D.C., 4-6 Nov 2013.
- H. Blockeel, B. Bogaerts, M. Bruynooghe, B. De Cat, S. De Pooter, M. Denecker, A. Labarre, J. Ramon, S. Verwer. "Modeling Machine Learning and Data Mining Problems with FO($\cdot$)". In: A. Dovier and V. Santos Costa (eds.), Internationcal Conference on Logic Programming, Budapest, 4-8 Sept 2012, Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary, volume 17, pages 14 -25, Schloss Daghstuhl - Leibniz-Zentrum für Informatik.
- B. De Cat, M. Denecker and P. Stuckey. "Lazy model expansion by incremental grounding". In: A. Dovier and V. Santos Costa (eds.), International Conference on Logic Programming, Budapest, 4-8 Sept 2012, Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary, volume 17, pages 201-211, Schloss Dagstuhl - Leibniz-Zentrum für Informatik.
- J. Devriendt, B. Bogaerts, C. Mears, B. De Cat and M. Denecker. "Symmetry propagation: Improved dynamic symmetry breaking in SAT". In: ICTAI, Athens, Greece, 7-9 November 2012, Proceedings of the 24th IEEE International Conference on Tools with Artificial Intelligence, ICTAI'12.
- B. De Cat, C. Machiels, G. Janssens and M. Denecker. "Regularity requirements in university course timetabling". In: Workshop on Preferences and Soft Constraints, Perugia, 12 September 2011, Proceedings of the 11th Workshop on Preferences and Soft Constraints (SofT 2011), pages 31-45.
- B. De Cat and M. Denecker. "DPLL(Agg): An efficient SMT module for aggregates". In: Workshop on Logic and Search (LaSh), 26th International Conference on Logic Programming, Edinburgh, 2010.
- M. Bruynooghe, B. De Cat, J. Drijkoningen, D. Fierens, J. Goos, B. Gutmann, A. Kimmig, W. Labeeuw, S. Langenaken, N. Landwehr, W. Meert, E. Nuyts, R. Pellegrims, R. Rymenants, S. Segers, I. Thon, J. Van

Eyck, G. Van den Broeck, T. Vangansewinkel, L. Van Hove, J. Vennekens, T. Weytjens, L. De Raedt. “An exercise with statistical relational learning systems”. In: P. Domingos and K. Kersting (eds.), International Workshop on Statistical Relational Learning, Leuven, Belgium, 2-4 July 2009.

- J. Wittocx, B. De Cat and M. Denecker. “Towards computing revised models for FO theories”. In: S. Abreu and D. Seipel (eds.), 18th International Conference on Applications of Declarative Programming and Knowledge Management, INAP 2009, Évora, Portugal, November 3-5, 2009, Revised Selected Papers, LNCS, volume 6547, pages 199-211.
- B. De Cat, J. Jansen and G. Janssens. “IDP3: Combining symbolic and ground reasoning for model generation”. In: Workshop on Grounding and Transformations for Theories with Variables (GTTV), 12th International Conference on Logic Programming and Nonmonotonic Reasoning, La Coruña, 15 Sept 2013.

## Peer-reviewed Abstracts

- J. Devriendt, B. Bogaerts, C. Mears, B. De Cat and M. Denecker. “Symmetry propagation: Improved dynamic symmetry breaking in SAT”. In: SymCon, Quebec City, 6-8 October 2012.
- J. Wittocx, B. De Cat and M. Denecker. “The IDP system”. In: P. Bouvry, L. van der Torre, E. Dubois and T. Latour (eds.), Benelux conference on artificial intelligence, Luxembourg, 25-26 October 2010, Proceedings of the 22nd Benelux Conference on Artificial Intelligence.

## Posters and Presentations at Miscellaneous Events

- “IDP: Model-expansion system for extended first-order logic”, 13th International Conference on Principles of Knowledge Representation and Reasoning (KR), 2012.
- “IDP3: Next-generation knowledge management system”, Flanders Scientific Research Community (WOG) Declarative Methods in Computer Science, 2013.
- “Recent advances in automated search”, University of Nebraska-Omaha, 2013.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
SCIENTIFIC COMPUTING GROUP
Celestijnenlaan 200A box 2402
B-3001 Heverlee
broes.decat@cs.kuleuven.be
people.cs.kuleuven.be/broes.decat