**KATHOLIEKE UNIVERSITEIT LEUVEN**
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

# MODEL GENERATION FOR ID-LOGIC

Promotor :
Prof. Dr. M. DENECKER

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

**Maarten MARIËN**

Februari 2009

**KATHOLIEKE UNIVERSITEIT LEUVEN**
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

# MODEL GENERATION FOR ID-LOGIC

Jury :
Prof. Dr. C. Vandecasteele, voorzitter
Prof. Dr. M. Denecker, promotor
Prof. Dr. ir. M. Bruynooghe
Prof. Dr. ir. G. Janssens
Prof. Dr. ir. P. Dutré
Prof. Dr. D. G. Mitchell (Simon Fraser University, Canada)
Prof. Dr. I. Niemelä (Helsinki University of Technology, Finland)

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

**Maarten MARIËN**

U.D.C. 681.3*I2

Februari 2009

©Katholieke Universiteit Leuven – Faculteit Ingenieurswetenschappen
Arenbergkasteel, B-3001 Heverlee (Belgium)

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

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

D/2009/7515/4
ISBN 978–94–6018–021–7

# Abstract

In the domain of *knowledge representation and reasoning*, one studies knowledge: what types of knowledge there are, how often these are used, how they can be expressed in a formal language, etc. An important goal of knowledge representation is to develop formal languages (logics) that can be used to express a wide range of problems, to develop automated reasoning methods for these languages, and to develop efficient implementations of these reasoning methods.

ID-logic is a knowledge representation language that extends classical logic with inductive definitions. It can express a large variety of practical problems in an intuitive way, and has therefore been promoted as a useful knowledge representation language.

Model generation is a very general and widely applicable automated reasoning method. The topic of this dissertation is *propositional model generation for ID-logic.* As such, this work offers an important contribution to the development of automated reasoning methods for ID-logic.

The main part of this dissertation is concerned with the propositional fragment of ID-logic, called PC(ID), and with model generation algorithms for it. We provide two alternative semantical characterizations of PC(ID), both of which yield important insights into the underlying structure of PC(ID) theories, and both of which therefore contribute to the understanding of the model generation task for PC(ID). Also, the second characterization offers a vocabulary-preserving transformation of PC(ID) theories to propositional logic.

We then study practical model generation algorithms for PC(ID). We discuss a number of possible strategies, provide various propagation rules, and present algorithms for these rules. We have also implemented a propositional model generator for PC(ID).

The rest of the dissertation is concerned with ID-logic itself. We discuss a methodology of knowledge representation in ID-logic and provide some examples. We also extend ID-logic with aggregate expressions, thereby extending the applicability of model generation for ID-logic. We study propositional model generation algorithms for this extension, and have implemented such algorithms. Finally, we compare ID-logic to a related formalism, namely ASP, and provide a transformation of ID-logic theories to ASP theories.

# Acknowledgements

Breaking with tradition, I will start this part of my thesis—undoubtedly the part that is read the most often—with an anecdote. It sets in 1986, on my first Communion day, in the zoo. It was a bright day. At one point, somewhere between the zebras and the camels, my mother asked me, pleasantly:

—“Well, Maarten, is this the most beautiful day of your life?” She had failed to add “…so far,” so, rational young little fellow I apparently was at the age of seven, I answered hesitatingly:

—“I don’t think so. . . .”

—“Then when *is* the most beautiful day of your life?,” my mother pressed on. Tough question: I paused to think. (Good habit, that.) What day to choose? My whole future life spun itself in front of my eyes: six more years of primary school, then six more years of secondary school. Then some years of university, I reckoned (but I never imagined *more* university was to follow *after* that), followed by…a job. A lifetime of work. Until…

—“…my retirement day!,” I exclaimed.

My family still laughs at me for being lazy enough to long for my retirement day at the age of seven. Of course, now that I’ve learned about reasoning with uncertainty, and about temporal logics and such, I know much better. Or . . . do I? Well, for the fun of it: I stand by my words! Which is not to say today is not a happy day.

Falling back in line with tradition, then, I want to thank the people that have helped make this a happy day for me. First, and foremost, there is my promoter, Marc Denecker. Marc has achieved the admirable feat of being both a good promoter and a friend. Throughout the years he has given me guidance, support, and trust. His guidance was especially useful in the early phases of my Ph.D. career—I remember several long sessions where Marc explained technical issues to me, while I was keeping track of his story and its many (many!) ramifications. His support came both in the form of motivational support and technical support; his influence on the contents of this text is omnipresent. His trust became important in the later phases of my career, when I had become more of an independent scientist than I was. His friendship, finally, has resulted in many joint activities: travelling, climbing, visiting parties. Marc, thanks.

I’ve been lucky to have had the help of Maurice Bruynooghe on several occasions, not least on early proof-reading of my thesis. The speed with which

he is able to not only gain insight in new scientific material, but actually suggest useful improvements, has always amazed me. I'd like to thank Maurice also for his unrelenting but extremely helpful eye for the tiniest of mistakes.

I would like to thank also the other members of my reading committee, Gerda Janssens, Phil Dutré, David Mitchell and Ilkka Niemelä, for their many valuable comments, and Carlo Vandecasteele for presiding the jury. Especially the foreign jury members, David Mitchell and Ilkka Niemelä, both of whom I've had the pleasure to visit in their home universities, have made important suggestions for improving the text.

Collaborating with my colleague Johan Wittocx has been a pleasure throughout the years. Many were the times that I had implemented a new feature but the implementation turned out to contain a bug, and Johan quickly suggested useful ways to locate it; many were the times that I relied on his organizational talent to run some experiments; many were the times, in short, that I appreciated Johan's collaborative support.

Life as a Ph.D. student was made agreeable through life outside of academia. This, in turn, was made agreeable with the help of many people, whom I want to thank for many reasons. In no particular order, I'm grateful to all of the following people:

**for friendship:** Álvaro, Fien, Anneleen, Fabián, Daan, Joost, Stephen, Hanne, Marc, Siesel, Bert, Élisa, and everybody else in the lists below;

**for being excellent travel partners:** Fien, Álvaro and Lien, Marc, Fabián, Anneleen and Siesel, Caroline and Dries, Daan, Dirk, Hanne, Johan, Nikolay, Maurice, Rudradeb;

**for saving my life:** by securing my climbing rope: Daan, Anneleen, Marc, Tijl, Álvaro, Fabián, Hanne;

**for being my photography buddy:** Álvaro;

**for losing many a game of go against me:** and for teaching me a good lesson at it too: Robby, Peter, John, Kris, Kwinten, Nelis, Frank, and many others;

**for sharing many coffee breaks:** Joost, Stephen, Johan, Hanne, Mai, Marc, Álvaro, Anneleen;

**for burning many a candle for me:** my grandparents;

**for continued support and love:** my sisters Goedele en Neleke, their children, Evelien, Amanda, Luna and Yara, and, last but definitely not least, my parents.

Maarten Mariën, 17 February 2009.

# Contents

**Abstract** **i**

**Acknowledgements** **iii**

**List of Symbols** **ix**

**1 Introduction** **1**
1.1 General background and goal of the thesis . . . . . . . . . . . . . 3
1.1.1 Significance of ID-logic in knowledge representation . . . 5
1.1.2 Application areas . . . . . . . . . . . . . . . . . . . . . . . . 6
1.2 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
1.3 Structure of the text . . . . . . . . . . . . . . . . . . . . . . . . 9

**2 Preliminaries** **11**
2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.2 Classical logic . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.2.1 Herbrand interpretations . . . . . . . . . . . . . . . . . . . 14
2.2.2 Propositional logic and SAT . . . . . . . . . . . . . . . . . 15
2.2.3 Three-valued and four-valued semantics . . . . . . . . . . 16
2.3 Well-founded and stable model semantics . . . . . . . . . . . . . 18
2.3.1 Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.3.2 Semantics . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.4 ID-logic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.4.1 Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.4.2 Semantics . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

**3 Knowledge representation in ID-logic** **31**
3.1 On modelling in ID-logic . . . . . . . . . . . . . . . . . . . . . . 31
3.1.1 Definitional knowledge . . . . . . . . . . . . . . . . . . . . 32
3.1.2 Modularity . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
3.1.3 First-order logic sentences . . . . . . . . . . . . . . . . . . 34
3.1.4 First-order logic rule bodies . . . . . . . . . . . . . . . . . 35
3.1.5 On the use of function symbols . . . . . . . . . . . . . . . 35
3.1.6 Non-Herbrand models . . . . . . . . . . . . . . . . . . . . 36

3.2 Model expansion . . . 37
3.3 Examples . . . 39
3.3.1 Block's world . . . 39
3.3.2 Hamiltonian circuit . . . 40
3.3.3 $N$-queens . . . 41
3.3.4 Transitive reduction . . . 41
3.4 The IDP system . . . 43
3.5 Grounding . . . 44
3.6 Satisfiability of propositional ID-logic . . . 45
3.7 Conclusions . . . 46

**4 Semantical analysis of PC(ID) 47**
4.1 Introduction . . . 47
4.2 Preliminaries . . . 48
4.2.1 Completion . . . 48
4.2.2 Definitional Normal Form . . . 48
4.2.3 Some graph concepts . . . 50
4.3 Justification semantics . . . 51
4.3.1 Justifications . . . 51
4.3.2 Properties of justifications . . . 54
4.3.3 Three-valued semantics . . . 58
4.3.4 Simplifications . . . 59
4.3.5 Related work . . . 64
4.4 Loop formula semantics . . . 64
4.4.1 Loop formulas . . . 65
4.4.2 Relation with ASP . . . 72
4.4.3 Corollaries and observations . . . 73
4.4.4 Related work . . . 77
4.5 Conclusions . . . 78

**5 SAT(ID) algorithms 81**
5.1 Introduction . . . 81
5.2 Preliminaries . . . 82
5.2.1 SAT Modulo Theories (SMT) . . . 82
5.2.2 Transition systems . . . 83
5.2.3 SMT solving techniques . . . 87
5.2.4 Normal form . . . 89
5.3 SAT(ID) transition rules . . . 89
5.3.1 Specific transition rules . . . 90
5.3.2 Concrete transition systems . . . 94
5.4 Algorithms . . . 100
5.4.1 UnitPropDef . . . 100
5.4.2 Relevant loop algorithms . . . 101
5.4.3 BwLoop . . . 103
5.4.4 FwLoop and $\Delta$-Propagate . . . 109
5.4.5 Simplify . . . 111

5.5 Discussion of the BwLoop algorithm . . . 114
5.5.1 Search space restricted to non-false literals . . . 114
5.5.2 Cycle sources . . . 115
5.5.3 The search space $HP$ . . . 118
5.5.4 Remarks and related work . . . 119
5.6 Enhancements to DPLL . . . 121
5.6.1 Clause learning, backjumping and restarts . . . 122
5.6.2 Heuristics . . . 123
5.6.3 Preprocessing . . . 124
5.7 MiniSat(ID) . . . 124
5.7.1 Transition system and strategy . . . 124
5.7.2 Motivation of this strategy . . . 127
5.7.3 Implementation details . . . 128
5.7.4 Evaluation . . . 130
5.8 Conclusions and related work . . . 139

**6 Aggregates 143**
6.1 Introduction . . . 143
6.2 Preliminaries . . . 143
6.2.1 Examples of ID-logic extended with aggregates . . . 143
6.2.2 Ground aggregate expressions . . . 145
6.2.3 Grounding of aggregates . . . 148
6.3 Aggregates: algorithms . . . 150
6.3.1 At-most-one statements . . . 150
6.3.2 Reductions to SAT . . . 151
6.3.3 Non-recursive aggregate expressions . . . 152
6.3.4 Recursive aggregate expressions . . . 161
6.4 Evaluation . . . 171
6.5 Conclusions . . . 172

**7 A comparison of ID-logic and ASP 175**
7.1 Introduction . . . 175
7.2 Transformations . . . 176
7.2.1 ID-logic to ASP . . . 176
7.2.2 ASP to ID-logic . . . 185
7.3 ID-logic modelling principles in ASP . . . 185
7.4 Model expansion vs. Herbrand model generation . . . 188
7.4.1 General introduction . . . 188
7.4.2 Problem transformations . . . 189
7.5 Conclusion . . . 190

**8 Conclusion 193**
8.1 Summary of contributions . . . 193
8.2 Future work . . . 194

**A Extended Clausal Normal Form (ECNF) 197**
A.1 Definitions . . . 198
A.2 Aggregate expressions . . . 199
A.3 Exists-unique and at-most-one expressions . . . 199

**B Problem encodings 201**
B.1 Hamiltonian circuit . . . 201
B.2 Sokoban puzzle . . . 202
B.3 Hitori puzzle . . . 205
B.4 Transitive opening . . . 205
B.5 Social golfer . . . 206
B.6 Magic series . . . 206

**C Complete version of BwLoop algorithm 209**

**Bibliography 211**

**Publication List 223**

**Biography 225**

# List of Symbols

| | |
|---|---|
| $\Sigma$ | A vocabulary, page 11 |
| $vocab(T)$ | The vocabulary of theory $T$, page 12 |
| $I$ | An interpretation, page 12 |
| $\mathrm{dom}(I)$ | The domain of interpretation $I$, page 12 |
| $S^I$ | The interpretation of symbol $S$ in $I$, page 12 |
| $I[\overline{x}/\overline{d}]$ | The interpretation that has the same domain as $I$, interprets $\overline{x}$ by $\overline{d}$ and coincides with $I$ on all other symbols, page 13 |
| $\boldsymbol{f}$ | The truth value "false", page 13 |
| $\boldsymbol{t}$ | The truth value "true", page 13 |
| $\boldsymbol{u}$ | The truth value "unknown", page 16 |
| $P(\overline{d})$ | A domain atom, page 13 |
| $I[P(\overline{d})/\boldsymbol{f}]$ | The interpretation $I'$ that is identical to $I$ except that $I'(P(\overline{d})) = \boldsymbol{f}$. Similar for $\boldsymbol{t}$, page 20 |
| $I[U/\boldsymbol{f}]$ | The interpretation $I'$ that is identical to $I$ except that $I'(P(\overline{d})) = \boldsymbol{f}$ for each $P(\overline{d}) \in U$. Similar for $\boldsymbol{t}$, page 20 |
| $\overline{S}$ | ($S$ is a set of literals) The set $\{\neg l \mid l \in S\}$, page 12 |
| $\widehat{S}$ | ($S$ is a set of literals) The set $S \cup \overline{S}$, page 12 |
| $\models$ | The satisfaction relation between an interpretation and a theory, or the entailment relation between two theories, page 14 |
| $Def(\Delta)$ | The defined predicates of $\Delta$, page 25 |
| $Open(\Delta)$ | The open symbols of $\Delta$, page 25 |
| $\mathrm{wfm}_\Delta(I_O)$ | Well-founded model of definition $\Delta$ extending $Open(\Delta)$-interpretation $I_O$, page 26 |

| Symbol | Description |
|---|---|
| $T_1 \cong T_2$ | ID-logic theories $T_1$ and $T_2$ have the same models, page 27 |
| $\varphi_l$ | The body of the rule defining $l$ if $l$ is an atom, or its negation if $l$ is a negative literal. Also used to denote the set of literals in that body, page 48 |
| $comp(\Delta)$ | Completion of definition $\Delta$, page 48 |
| $\mathcal{C}_{\text{lits}}$ | Conjunctively defined literals, page 48 |
| $\mathcal{D}_{\text{lits}}$ | Disjunctively defined literals, page 48 |
| $\bot$ | The empty disjunction, page 19 |
| $\top$ | The empty conjunction, page 19 |
| $\mathcal{GL}(\Delta, I)$ | Gelfond-Lifschitz reduct of logic program $\Delta$ under interpretation $I$, page 22 |
| $\mathcal{L}$-$MX$ | Model expansion for logic $\mathcal{L}$, page 37 |
| $\mathcal{L}$-$MX_{\langle T,\sigma \rangle}, MX_{\langle T,\sigma \rangle}$ | Parameterized model expansion, with theory $T$ and input vocabulary $\sigma$ (for logic $\mathcal{L}$), page 37 |
| $DJ(l)$ | Direct justification for literal $l$, page 51 |
| $J$ | A justification, page 51 |
| $DJ_J(l)$ | ($J$ is a justification, $l$ a literal) Direct justification for $l$ in $J$, page 51 |
| $d_J(l)$ | ($l \in \mathcal{D}_{\text{lits}}$, $J$ is a justification) The literal $x$ for which $DJ_J(l) = \{x\}$, page 51 |
| $\circlearrowleft$ | The set of unavoidably false literals, page 59 |
| $\Delta^{\varnothing}$ | Loop-simplification of $\Delta$, page 59 |
| $\mathcal{C}^{\text{ext}}(L)$ | External conjuncts of $L$, page 65 |
| $\mathcal{D}^{\text{ext}}(L)$ | External disjuncts of $L$, page 65 |
| $LF_\Delta(L)$ | Loop formula of $L$ in $\Delta$, page 66 |
| $LF_\Delta$ | The relevant loop formulas of $\Delta$, page 68 |
| $M \parallel F$ | A state with state sequence $M$ and theory $F$, page 83 |
| $\Longrightarrow$ | A transition relation, page 84 |
| $\mathsf{R}$ | A transition rule, page 84 |
| $\xRightarrow{\mathsf{R}}$ | Transition relation specified by transition rule $\mathsf{R}$, page 84 |

| | |
|---|---|
| $\mathcal{A}_{\text{lits}}$ | Literals defined by an aggregate expression, page 146 |
| $S.min^{M}_{Aggr}$ | Known minimum value of weighted set $S$ in interpretation $M$ over $Aggr$, page 152 |
| $S.max^{M}_{Aggr}$ | Known maximum value of weighted set $S$ in interpretation $M$ over $Aggr$, page 152 |
| $DJ(l)$ | Direct justification for literal $l$, extended to aggregates, page 161 |
| $All^{+}(l)$ | The union of all positive direct justifications for $l \in \mathcal{A}_{\text{lits}}$, page 164 |
| $All^{-}(l)$ | The union of all negative direct justifications for $l \in \mathcal{A}_{\text{lits}}$, page 164 |
| $\mathcal{G}^{\text{ext}}(L)$ | Generalized external disjuncts of $L$, page 164 |
| $GLF_{\Delta}(L)$ | Generalized loop formula $L$ in $\Delta$, page 165 |
| $\equiv_{\Sigma}$ | Equivalence up to $\Sigma$, page 176 |

# Chapter 1

# Introduction

This thesis deals with the topic of *model generation for ID-logic.* ID-logic is a formal language: it extends classical logic with inductive definitions. It has been promoted as a useful knowledge representation language. Model generation is a quite general type of reasoning that can be applied to a wide variety of practical computational problems. We elaborate on these claims further on.

Without going into the formal specifics, we illustrate some of the topics of this thesis with an informal example, based on an example from Mumick et al. (1990).

**Example 1.1.** (Company control) A large company A wants to acquire control over a company B via the stock-market. It will have control when the sum of shares of B that are in A's control exceeds 50%; the shares in A's control are those that A owns, and those that other companies that are controlled by A own. Thus, if A controls a third company C, then the shares that C owns of B contribute to A's potential control over B. Additional relevant domain knowledge is that the total number of shares of a company is 100%, as well as some concrete data: A owns 20% of its own shares, B owns 10% of C's shares, and C owns 40% of A's shares.

There may be several solutions for A to gain control of B: A may buy new shares in B, or it may buy shares in other companies that possess shares in B. Different solutions have different costs and may differ also on other more strategic aspects (e.g., A may want to avoid buying shares in B, to hide its intentions). Thus, A is interested in viewing different solutions to evaluate them along several criteria.

We illustrate the use of ID-logic and the method of problem solving by means of model generation in the context of this problem. The first step of this method is to describe the knowledge of this domain. The crucial concept in this problem is that of *control.* This is a nice example of a recursively defined concept. In ID-logic, this concept is expressed by the following definition:

$$\left\{ \begin{array}{ll} \forall x,y \ \big(Controls(x,y) \leftarrow Sum\{\underline{s},z \mid & (x = z \lor Controls(x,z)) \\ & \land Owns(z,y) = s\} > 50\big) \end{array} \right\}. \quad (1.1)$$

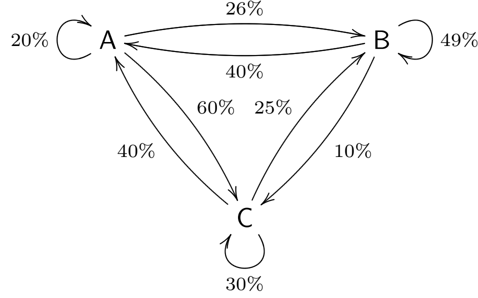



Figure 1.1: A shareholdership function for Example 1.1

Informally, this definition reads: "Company $x$ controls company $y$ if the sum of shares $s$ of $y$ controlled by $x$, i.e. those that $x$ itself owns and those that other companies $z$ controlled by $x$ own, exceeds 50." Note that this definition contains a *sum aggregate*. The summation is over a multiset of values $s$; we have denoted this multiset by the set of tuples $(\underline{s}, z)$.

We express also the additional domain knowledge, and the assertion that A should control B, as ID-logic formulas:

$$\forall c \; Sum\{\underline{s}, x \mid s = Owns(x, c)\} = 100, \tag{1.2}$$

$$Owns(\mathsf{A}, \mathsf{A}) = 20, \tag{1.3}$$

$$Owns(\mathsf{B}, \mathsf{C}) = 10, \tag{1.4}$$

$$Owns(\mathsf{C}, \mathsf{A}) = 40, \tag{1.5}$$

$$Controls(\mathsf{A}, \mathsf{B}). \tag{1.6}$$

We have now expressed the relevant knowledge in a formal *ID-logic theory* (formulas (1.1)–(1.6)). The next (and final) step of the problem solving method is to compute solutions for this theory, i.e., situations in which all requirements are satisfied. This can be done using the techniques developed in this thesis.

One of the possible solutions is shown in Figure 1.1. Indeed, this ownership function makes sure that each of (1.1)–(1.6) is satisfied. This solution is represented in logic as *a model*, $M$, of the theory (1.1)–(1.6). $M$'s *domain* is the set of companies $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$ and values $\{0, 1, \ldots, 100\}$, and $M$ interprets $Owns$ as

$$\begin{array}{lll} (\mathsf{A}, \mathsf{A}) \mapsto 20, & (\mathsf{B}, \mathsf{A}) \mapsto 40, & (\mathsf{C}, \mathsf{A}) \mapsto 40, \\ (\mathsf{A}, \mathsf{B}) \mapsto 26, & (\mathsf{B}, \mathsf{B}) \mapsto 49, & (\mathsf{C}, \mathsf{B}) \mapsto 25, \\ (\mathsf{A}, \mathsf{C}) \mapsto 60, & (\mathsf{B}, \mathsf{C}) \mapsto 10, & (\mathsf{C}, \mathsf{C}) \mapsto 30, \end{array}$$

and $Controls$ as

$$(\mathsf{A}, \mathsf{A}), \qquad (\mathsf{A}, \mathsf{B}), \qquad (\mathsf{A}, \mathsf{C}).$$

A controls C because it has 60% of its shares. Therefore, it controls 26%+25% = 51% of B's shares, and therefore A controls B as well. Therefore A controls all of its own shares, hence it controls itself. B and C control nobody.

## 1.1 General background and goal of the thesis

This thesis falls within the broader research area of *knowledge representation and reasoning* (KR&R). In this domain, one studies knowledge: what types of knowledge there are, how they are used, how they can be expressed in a formal language, etc. An important goal of this field is to develop formal languages (logics) that can be used to express a wide range of problem domains, to develop automated reasoning methods for these languages, and to develop efficient implementations of these reasoning methods. This goal is an important step towards *declarative problem solving*: a method of (computational) problem solving whereby the human expert *declares* his knowledge of the problem in a formal specification (theory), and then simply applies a program that implements the appropriate automated reasoning task to this theory, to solve the problem.

First-order logic (FO) is an example of a well-known and widely used KR language. It has many assets: it is very expressive, has a clear and objective informal semantics which corresponds to its formal semantics, and since it is so widely studied, there exist many types of reasoning for it. Examples include *deductive reasoning*, implemented by theorem provers, *querying*, implemented amongst others by SQL database systems, and *model generation*, implemented by model generators or satisfiability solvers.

A type of knowledge that is used frequently throughout mathematics, and that is present in almost any non-trivial domain, is *definitional knowledge*. Despite the expressivity of FO, it cannot express some sorts of definitional knowledge, namely *inductive definitions*. We give two examples of inductive definitions that cannot be expressed in FO:[1]

- the transitive closure $TC$ (reachability) of an arbitrary binary relation $R$. It is usually defined inductively by the following two rules:
  - a tuple $(x, y)$ is in $TC$ if it is in $R$;
  - a tuple $(x, y)$ is in $TC$ if there is some $z$ such that both $(x, z)$ and $(z, y)$ are in $TC$.

  This definition is inductive because the second rule makes use of the concept being defined: $(x, y)$ is in $TC$ if ... are in $TC$;
- the *control* relation from Example 1.1: it defines which company controls which other company in terms of the ownership function, and the control relation itself.

It is therefore natural from a KR point of view to extend FO with a formal language construct to express definitional knowledge. This is precisely what ID-logic does. It offers a syntax and a formal semantics to express inductive

[1] The fact that transitive closure cannot be expressed in FO is well-known (e.g., Libkin, 2004, Proposition 3.1). The fact that the control relation cannot be expressed in FO is an easy consequence: let $R$ be an arbitrary relation, then if we construct the $Owns$ function such that for any $(a, b) \in R$, $Owns(a, b) = 51$ and for any $(a, b) \notin R$, $Owns(a, b) = 0$, the $Controls$ relation as defined in (1.1) is the transitive closure of $R$.

definitions (IDs); this formal semantics corresponds to the informal semantics of definitions as used in mathematics (Denecker, 1998). Hence, ID-logic extends first-order logic with inductive definitions. ID-logic was originally defined by Denecker (2000), and further developed by Denecker and Ternovska (2004, 2008). It has been shown to be a useful language for knowledge representation, for instance by Denecker and Ternovska (2007).

Other language constructs that enhance modelling flexibility can be added to FO, or to ID-logic, as well. This has been done with *aggregate expressions* such as the sum aggregate in Example 1.1 by Pelov et al. (2007).

The semantics of ID-logic is specified as an extension of the standard model-theoretic semantics of FO. A model describes a situation of the specification's domain that satisfies all requirements of the specification. For instance, in the company control example (Example 1.1), the model $M$ specifies a situation representing a particular state of shareholdership, and the corresponding control relation. Examples where it is useful to find a model of a given theory abound. We elaborate on practical application areas in Section 1.1.2, and sketch a simple example here. Consider a course scheduling problem: we can describe in an ID-logic theory that different courses should be taught at different moments, that they should be taught in classrooms that are big enough to hold all subscribers to the course, etc. The desired solution is a schedule that satisfies all requirements: each model of the ID-logic theory describes such a schedule.

Hence, an important goal of the KR&R research group of Leuven is to develop the automated reasoning method *finite model generation* for ID-logic, and to develop efficient algorithms and implementations for it. With this purpose, we have developed a system called IDP in close collaboration with Johan Wittocx. This system can handle finite model generation for full first-order logic—and for ID-logic with aggregate expressions, which extends first-order logic.

In the current state of the art, finite model generation problems are often solved—e.g., in the IDP system—in two phases:

1. a *grounding* phase, in which a given first-order theory and a finite domain are reduced to an equivalent variable-free, or *propositional*, theory;

2. a *propositional model generation* phase, in which the models of a given propositional theory are computed.

This thesis has the second phase, *propositional model generation for ID-logic*, as its main topic.

The propositional fragment of FO is called *propositional calculus* (PC); that of ID-logic is called PC(ID). The model generation problem for PC is in practice equivalent to the *satisfiability* problem for PC, which is called SAT. Likewise, the model generation problem for PC(ID) (in other words, the propositional model generation problem for ID-logic) is called SAT(ID). Thus, this thesis seeks to develop techniques to solve the SAT(ID) problem.

The research domain of SAT has thrived in recent years; many highly efficient "SAT solvers" have been developed (see, e.g., Mitchell, 2005). We therefore

realize the goal of this thesis by *extending* such SAT solvers with techniques to handle inductive definitions (IDs) and aggregate expressions.

### 1.1.1 Significance of ID-logic in knowledge representation

The type of knowledge under consideration in this work, and representable in ID-logic, can be described as *precise and discrete knowledge*. However, in practical settings of knowledge representation, precise information is often lacking. This phenomenon has been the motivation for the research areas of nonmonotonic and common-sense reasoning. Nevertheless, inductive definitions have a relevant role to play in some of the most important common-sense knowledge principles.

One of the first nonmonotonic reasoning principles to be investigated was the *Domain Closure Assumption* by Reiter (1982). It is the assumption that the domain of discourse contains no other objects than those named by ground terms. This assumption cannot be expressed in classical first-order logic, but it can be in ID-logic, as we demonstrate in Section 3.1.6. The Domain Closure Assumption is often applied in combination with the *Unique Name Axioms* by Reiter (1980), which express that different terms represent different objects. This combination is implicitly present in logics with Herbrand model semantics (cf. Section 2.2.1), such as logic programming.

An important principle of *default reasoning* is the *Closed-World Assumption* (CWA) by Reiter (1977), which assumes any proposition to be false unless it can be proven from the given theory. Typically this principle is applied to a theory as a whole. ID-logic, in contrast, is suitable to model such defaults in a localized way, namely within inductive definitions. Indeed, when a definition contains no undefined symbols, we can interpret it as a set of material implications augmented with the CWA, instead of the standard view of a set of definitional rules. The principle of CWA also led to the development of the *Local Closed-World Assumption* (Cortés Calabuig, 2008), which applies a closed-world assumption on a well-defined domain, and an open-world assumption elsewhere. Also this assumption can suitably be expressed in ID-logic.

Temporal reasoning is a major subdiscipline of KR&R. Inductive definitions have an important contribution to offer to this field. This was illustrated by Denecker and Ternovska (2007), who formalized in ID-logic the most general representation to date of *Situation Calculus*. Involving iterated inductive definitions, this representation could correctly deal with arbitrary ramifications of actions.

Closely related to the previous, also *causal knowledge* has been widely studied in KR&R. The Situation Calculus formalisation of (Denecker and Ternovska, 2007) builds on the observation that the construction process of an inductive definition formally mimics the physical process of the propagation of causes and effects in a dynamic system. Also CP-logic (Vennekens, 2007), a logic for representing causal processes, is based on the same construction process, using the well-founded semantics.

Finally, ID-logic extends FO, which can be considered the best-known logic for knowledge representation; many other logics have been compared to FO.

### 1.1.2 Application areas

Examples of problems on which finite model generation has been applied with great success are many and come both from industry and from academia. They include various types of *electronic design analysis* such as hardware testing and verification (Gupta et al., 2006), logic synthesis, FPGA routing (Nam et al., 1999); of *software analysis* methods such as formal verification (Prasad et al., 2005) and computer aided verification (Heljanko and Niemelä, 2003; Gupta and Malik, 2008); *artificial intelligence* domains such as planning (Kautz and Selman, 1992), diagnostic reasoning (Balduccini and Gelfond, 2003), decision support (Nogueira et al., 2001); and domains from other sciences such as biology (Erdem and Türe, 2008), linguistics (Konczak and Vogel, 2005), and many others. The referenced works are but a minimal selection of the vast body of work that discusses applications of finite model generation.

We clarify the methodology of problem solving by means of finite model generation by elaborating on a concrete example. It concerns a decision support system by Nogueira et al. (2001) for the NASA's space shuttle. The reaction control system (RCS) of the space shuttle is the system responsible for performing precise manoeuvers in space. It consists both of mechanical components such as fuel and oxidizer tanks, plumbing, valves and jets, and of electronic circuitry to control the valves and direct the jets. All components are subject to failure, e.g., valves can get stuck. The space shuttle has built-in redundancy to cope with such failures. However, finding a suitable plan to correctly manoeuver the space shuttle even in the presence of failures is a complex task. Nogueira et al. solved this task by formally modelling the RCS's functionality as a Stable logic program. Given a set of failures and a manoeuvering goal, the computational task of finding a suitable plan can then be solved by finding finite models of that program.

We show part of such a formal model of the RCS's functionality.[2] The plumbing system consists of *nodes* connected by *pipes*. Such nodes can be tanks containing helium or propellant fuel, junctions between the pipes, and jets. Pipes may or may not have a valve, that can be open or closed at any timepoint (depending on whether or not they are stuck, and depending on the instructions issued by an astronaut, e.g. by pressing a button, which is connected to the valve through electronic circuitry). Any node or any valve might contain a leak. We first represent which nodes are *linked*, i.e., between which nodes fluids can flow uninterruptedly.

$$\left\{\begin{array}{l} \forall n_1, n_2, t\ Linked(n_1, n_2, t) \leftarrow \\ \qquad \neg Leaking(n_1) \wedge \neg Leaking(n_2) \wedge \\ \qquad \Big(Pipe(n_1, n_2) \vee \\ \qquad \ \big(\exists v\ PipeWithValve(n_1, n_2, v) \wedge \\ \qquad \qquad Position(v, t) = Open \wedge \neg Leaking(v)\big)\Big) \end{array}\right\} \tag{1.7}$$

[2]Our representation here slightly deviates from the one in (Nogueira et al., 2001), but the essence is the same.

Definition (1.7) defines nodes $n_1$ and $n_2$ to be linked at (discrete) timepoint $t$, if neither of them contains a leak, and they are connected by a direct pipe, or by a pipe that contains a non-leaking valve that is open at time $t$. Using this concept, it is straightforward to define whether a fluid can reach from $n_1$ to $n_2$ at time $t$:

$$\left\{\begin{array}{ll} \forall n_1, n_2, t\ CanReach(n_1, n_2, t) \leftarrow & Linked(n_1, n_2, t), \\ \forall n_1, n_2, t\ CanReach(n_1, n_2, t) \leftarrow & \exists n_3\ CanReach(n_1, n_3, t) \wedge \\ & \quad Linked(n_3, n_2, t) \end{array}\right\}. \tag{1.8}$$

Definition (1.8) defines that a fluid can reach from $n_1$ to $n_2$ at time $t$ if they are linked at that time, or if a fluid can reach from $n_1$ some intermediate node $n_3$ at time $t$, which itself is linked to $n_2$ at time $t$.

Propellant tanks may need to be pressurized at certain timepoints: this is the case when helium from a helium tank can reach the propellant tank.

$$\left\{\begin{array}{ll} \forall htk, ptk, t\ Pressurizes(htk, ptk, t) \leftarrow & HeliumTank(htk) \wedge \\ & PropellantTank(ptk) \wedge \\ & CanReach(htk, ptk, t) \end{array}\right\} \tag{1.9}$$

Finally, we can describe constraints on the plumbing system, such as "a tank should never be pressurized through two different helium tanks at the same time". We can use the previously defined concept *Pressurizes* to express this constraint:

$$\begin{aligned} &\forall ptk, t\ PropellantTank(ptk) \supset \\ &\qquad \neg\big(\exists\, htk_1, htk_2\ htk_1 \neq htk_2 \wedge Pressurizes(htk_1, ptk, t) \wedge \\ &\qquad\qquad Pressurizes(htk_2, ptk, t)\big). \end{aligned} \tag{1.10}$$

The formal theory modelling all of the RCS's functionality contains many more similar definitions and constraints. Certain input data for this theory is fixed, such as the layout of the plumbing system (which nodes are helium tanks, which are junctions, between which nodes there are pipes, etc.); certain input data varies over different flights of the space shuttle, such as which nodes are leaking, which valves are stuck, and which manoeuver is desired. The theory augmented with the (finite) input data may have a number of (finite) *models*. Any such model describes concrete relations and functions for all the concepts in the theory, e.g., it describes per valve when it is open, it describes for all nodes $n_1$ and $n_2$ when a fluid can reach from $n_1$ to $n_2$, etc. Some of these relations represent a plan (a sequence of instructions issued by an astronaut) that will execute the desired manoeuver correctly, even in the presence of faults. If there is no such model, there does not exist a correct plan.

Thus, the methodology is as follows:

- a knowledge engineer (in the example, engineers from NASA) models the problem domain in a formal theory;
- for a specific problem in the domain, the specific input data is gathered;

- the specific problem is solved by automatically generating (a) model(s) of the given theory and data.

The advantage of this approach towards computational problem solving lies in the fact that the complex domain knowledge of the problem at hand is represented in a declarative way. This causes solutions to be elaboration tolerant with respect to changes in the domain knowledge, and in general significantly simpler, smaller and more flexible than those produced by “direct” approaches—(ad hoc) imperative implementations of (ad hoc) algorithms for solving the computational problem. For instance, Boenn et al. (2008) report on their experience with automatic composition of melodic and harmonic music: two previous direct approaches resulted in programs of respectively 8,000 and 88,000 lines of (imperative) code, whereas the declarative approach using finite model generation by Boenn et al. yielded a logic theory of 191 lines, augmented with 500 lines of scripting code to provide desired functionality such as creating actual audio files.

## 1.2 Contributions

The first part of the thesis is a semantical study of propositional inductive definitions (hence of PC(ID)) from the viewpoint of model generation. We provide two independent alternative characterizations of the semantics of propositional inductive definitions. The first is based on *justification semantics* (Denecker and De Schreye, 1993), the second on *loop formula semantics* (Lin and Zhao, 2004). These two semantics will prove essential for developing the algorithms later on in this thesis. These characterizations provide an alternative to the model-theoretic semantics of PC(ID).

Based on these results, we develop algorithms to solve the SAT(ID) problem. More precisely, we develop algorithms for reasoning on PC(ID) theories, and study how to integrate them in current SAT solvers. We use a framework by Nieuwenhuis et al. (2006) to discuss different strategies for solving the SAT(ID) problem, and explain how our algorithms can be used to implement those strategies. We implement a SAT(ID) solver, called MiniSat(ID), as an extension of the SAT solver MiniSat (Eén and Sörensson, 2003), and we illustrate that it has state-of-the-art performance.

We then discuss a further extension of ID-logic with *aggregate expressions*. We use a semantics developed by Pelov et al. (2007) that provides a natural extension of the semantics of inductive definitions used in ID-logic. This extension further enhances the applicability of ID-logic model generation. We also develop algorithms to solve the corresponding extended satisfiability problem.

Finally, we make a comparison of ID-logic and Answer Set Programming (ASP) (see, e.g., Niemelä, 1999), or more precisely, of ID-logic and Stable logic programs. This logic is closely related to ID-logic. Therefore we relate our work to ASP—and we do so also throughout the rest of the thesis.

We summarize our contributions:

1. a characterization of propositional IDs using justifications, and one using loop formulas;
2. several algorithms to solve the SAT(ID) problem;
3. an implementation of some of these algorithms;
4. algorithms to solve the SAT(ID) problem extended with aggregate expressions under Pelov et al.'s semantics;
5. a detailed comparison of ID-logic and Stable logic programs.

This work has profited from a collaboration with colleague Johan Wittocx. Especially, initial understanding of the properties of justifications (Chapter 4) and of the BwLoop algorithm (Chapter 5) was gained through this collaboration. However, all technical contributions of Chapters 4–7—definitions, propositions, proofs—are the author's contribution.

## 1.3 Structure of the text

The text is structured as follows:

- Chapter 2 introduces the background required for the rest of the text: it introduces first-order logic, logic programming under the stable semantics and under the well-founded semantics, and ID-logic.
- In Chapter 3 we present ID-logic from a knowledge representation point of view: we give examples and provide some methodological principles of modelling in ID-logic. We then discuss the IDP system, and make the bridge to the rest of the thesis by briefly explaining the grounding phase. Some of the contributions in this chapter were published in (Mariën, Gilis, and Denecker, 2004; Mariën, Wittocx, and Denecker, 2006; Wittocx, Mariën, and Denecker, 2008c).
- Chapter 4 provides two different characterizations of the semantics of propositional IDs: one in justification semantics, the other in loop formula semantics. The contributions in this chapter regarding justifications have been published in (Mariën, Mitra, Denecker, and Bruynooghe, 2005; Mariën, Wittocx, and Denecker, 2007b; Mariën, Wittocx, Denecker, and Bruynooghe, 2008); the contributions regarding loop formulas have not been published before.
- We develop algorithms for the SAT(ID) problem in Chapter 5. We present these algorithms in Nieuwenhuis et al.'s framework, which enables us to discuss different strategies for solving the SAT(ID) problem. We present the SAT(ID) solver MiniSat(ID), provide experimental results to evaluate some of the possible strategies, and evaluate MiniSat(ID)'s performance. Most contributions in this chapter have been published in (Mariën,

Mitra, Denecker, and Bruynooghe, 2005; Mariën, Wittocx, and Denecker, 2007a,b; Mariën, Wittocx, Denecker, and Bruynooghe, 2008).

- In Chapter 6 we provide algorithms for the SAT(ID) problem extended with aggregate expressions under Pelov et al.'s semantics. These contributions have not been published before.

- In Chapter 7 we compare ID-logic and Stable logic programs as knowledge representation languages. We provide a transformation from ID-logic theories to Stable logic programs, and vice versa. We also briefly compare two methods for finite model generation: Herbrand model generation and model expansion. The contributions in this chapter were published in (Mariën, Gilis, and Denecker, 2004; Mariën, Wittocx, and Denecker, 2006).

- Finally, we summarize the results of this thesis and conclude in Chapter 8.

# Chapter 2

# Preliminaries

## 2.1 Introduction

In this chapter we introduce concepts from the domains of classical logic and of logic programming that will be used throughout the rest of the thesis. In particular, we define several logics.

A *logic* consists of a syntax and a semantics. The syntax is specified by defining what constitutes a well-formed statement, or theory, in the logic; the semantics is given by defining what constitutes a *model* of a given well-formed statement. A third component of a logic is informal in nature and is not always stated: its *informal semantics*. It describes how a well-formed statement in the logic should be read, and therefore how it can intuitively be understood.

The logics that we define here are classical first-order logic, logic programming under the well-founded semantics and under the stable semantics, and ID-logic. Stable logic programs is defined as a particular sublogic of logic programming under the stable semantics.

In this chapter we give a non-constructive characterization of models of ID-logic theories. In later chapters we will provide algorithms to construct, or generate, such models.

## 2.2 Classical logic

We begin by recalling some essential concepts from classical logic; for a full exposition, see, e.g., (Enderton, 2001).

A *vocabulary*, which we will often denote by $\Sigma$, is a set of variables, and predicate and function symbols, each with an associated arity. We often denote a symbol $S$ with arity $n$ by $S/n$. 0-ary function symbols are called *constants*; variables and constants are called *object symbols*. A *term* is inductively defined as either an object symbol or an expression of the form $F(t_1, \ldots, t_n)$, where $F$ is an $n$-ary function symbol, and all the $t_i$ are terms. An *atom* is an expression

of the form $P(t_1, \ldots, t_n)$, where $P$ is an $n$-ary predicate symbol, and all the $t_i$ are terms.

A vocabulary is called *propositional* if it consists only of 0-ary predicate symbols ("propositions").

The set of *first-order logic (FO) formulas* is inductively defined as:

- an atom is an FO formula;
- if $\varphi$ is an FO formula, then so is $\neg\varphi$;
- if $\varphi$, $\psi$ are FO formulas, then so is $\varphi \vee \psi$;
- if $\varphi$ is an FO formula and $x$ is a variable, then $\exists x\ \varphi$ is an FO formula. Occurrences of $x$ in $\varphi$ are said to be in the *scope* of the quantifier $\exists x$.

We use the following standard abbreviations: $\varphi \wedge \psi$ for $\neg(\neg\varphi \vee \neg\psi)$, $\varphi \supset \psi$ for $\neg\varphi \vee \psi$, $\varphi \equiv \psi$ for $(\varphi \supset \psi) \wedge (\psi \supset \varphi)$, and $\forall x\ \varphi$ for $\neg(\exists x\ \neg\varphi)$. Also, we abbreviate $\bigwedge_{s \in S} s$ by $\bigwedge S$, and $\bigvee_{s \in S} s$ by $\bigvee S$. A *literal* is an atom $P(t_1, \ldots, t_n)$ or its negation $\neg P(t_1, \ldots, t_n)$, called respectively a *positive* and a *negative* literal. For a literal $l$, we identify $\neg\neg l$ with $l$. For a set of literals $S$, we denote by $\overline{S}$ the set $\{\neg l \mid l \in S\}$, and by $\widehat{S}$ the set $S \cup \overline{S}$.

The informal reading of an FO formula is well-known: "$\wedge$" means "and" (conjunction), "$\vee$" means "or" (disjunction), "$\exists x$" means "there exists an $x$ such that", etc.

A variable $x$ is a free variable of a formula $\varphi$ if $x$ appears in $\varphi$, and is not in the scope of a quantifier. An FO *sentence* is a closed formula, i.e., one without free variables. An FO *theory* is a set of FO sentences. For a theory $T$, we denote its vocabulary by $vocab(T)$. We say that $T$ is a theory *over vocabulary* $\Sigma$ if $vocab(T) \subseteq \Sigma$. A finite theory is formally equivalent to one sentence, consisting of the conjunction of all sentences of the theory.

**Example 2.1.** Let $\Sigma$ contain the constant symbol $Philosopher$ and the predicate symbols $Man/1$ and $Mortal/1$. In this vocabulary we can describe properties of men and of mortals, and of one philosopher. Then the following is a well-formed FO theory over $\Sigma$:

$$T_{2.1} = \left[ \begin{array}{l} Man(Philosopher), \\ \forall x\ Man(x) \supset Mortal(x) \end{array} \right].$$

The informal reading of the two sentences of $T_{2.1}$ is "the philosopher is a man" and "all objects that are men are mortal".

Let $\Sigma$ be a vocabulary. A $\Sigma$-*interpretation* $I$, or simply interpretation if $\Sigma$ is clear from the context, consists of a *domain* $\mathrm{dom}(I)$, and an interpretation of each predicate and function symbol $S$ of $\Sigma$, denoted $S^I$. An interpretation of an object symbol $o \in \Sigma$ is a domain element $d \in \mathrm{dom}(I)$; an interpretation of a predicate symbol $P/n \in \Sigma$ is an $n$-ary relation on $\mathrm{dom}(I)$; an interpretation of a function symbol $F/n \in \Sigma$ is an $n$-ary function on $\mathrm{dom}(I)$. An interpretation can be understood as denoting a state of the described domain. A *pre-interpretation*

of $\Sigma$ consists of a domain and an interpretation of all object and function symbols of $\Sigma$.

Throughout the text, we implicitly assume $=/2$ to be part of any non-propositional vocabulary; it is an interpreted predicate symbol that always represents the identity relation in the domain, i.e., $=/2^I = \{(d,d) \mid d \in \mathrm{dom}(I)\}$, for any interpretation $I$. We use $=/2$ in infix-notation.

**Example 2.2.** Example 2.1 continued. Consider the $\Sigma$-interpretation $I_{2.2}$ with domain $\{\mathsf{Snoopy}, \mathsf{Socrates}\}$ that interprets $Philosopher^{I_{2.2}} = \mathsf{Socrates}$, $Man^{I_{2.2}} = \{\mathsf{Socrates}\}$, and $Mortal^{I_{2.2}} = \{\mathsf{Snoopy}, \mathsf{Socrates}\}$. This interpretation describes a world with two objects, called Snoopy and Socrates, and with the properties that Socrates is a man but Snoopy is not, and both are mortals, and where the object symbol *Philosopher* denotes Socrates.

We use the following alternative terminology and notation. We represent the truth value *true* by $\boldsymbol{t}$, and *false* by $\boldsymbol{f}$. For a given vocabulary $\Sigma$ and interpretation $I$, a *domain atom* is of the form $P(\overline{d})$, where $P/n$ is a predicate symbol of $\Sigma$, and $\overline{d} \in \mathrm{dom}^n(I)$. We denote by $I(P(\overline{d})) = \boldsymbol{t}$ that $\overline{d} \in P^I$ and by $I(P(\overline{d})) = \boldsymbol{f}$ that $\overline{d} \notin P^I$.

**Example 2.3.** Examples 2.1 and 2.2 continued. We have the following truth values in $I_{2.2}$:

$$I_{2.2}(Man(\mathsf{Socrates})) = \boldsymbol{t} \qquad I_{2.2}(Mortal(\mathsf{Socrates})) = \boldsymbol{t}$$
$$I_{2.2}(Man(\mathsf{Snoopy})) = \boldsymbol{f} \qquad I_{2.2}(Mortal(\mathsf{Snoopy})) = \boldsymbol{t}.$$

We sometimes restrict a $\Sigma$-interpretation $I$ to a vocabulary $\sigma \subset \Sigma$. This restriction, denoted $I|_\sigma$, is the $\sigma$-interpretation with the same domain as $I$, and same interpretations on all symbols of $\sigma$. Conversely, we may extend a $\sigma$-interpretation to a $\Sigma$-interpretation. A special case is when $\sigma$ consists of all object and function symbols of $\Sigma$: thus we may extend a pre-interpretation of $\Sigma$ to a $\Sigma$-interpretation.

For an arbitrary term $t$ and interpretation $I$, we denote by $t^I$ the domain element $o^I$ if $t$ is an object symbol $o$, and the domain element $F^I(t_1^I, \ldots, t_n^I)$ if $t$ is a term $F(t_1, \ldots, t_n)$. We extend the notation $t^I$ to tuples of terms $\overline{t}$. If $\overline{x}$ is a tuple of variables of $\Sigma$, $n = |\overline{x}|$ and $\overline{d} \in \mathrm{dom}^n(I)$, we denote by $I[\overline{x}/\overline{d}]$ the interpretation that has the same domain as $I$, interprets $\overline{x} = (x_1, \ldots, x_n)$ by $\overline{d} = (d_1, \ldots, d_n)$, and coincides with $I$ on all other symbols.

**Example 2.4.** Consider a vocabulary $\Sigma$ that includes a function symbol $Fib/1$ and the constant symbol *Zero*. Consider also an interpretation $I$ with domain $\mathrm{dom}(I) = \mathbb{N}$ that interprets *Zero* by the natural number 0, and $Fib/1$ by the Fibonacci function $f : \mathbb{N} \to \mathbb{N}$ defined by

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1, \\ f(n-1) + f(n-2) & \text{if } n > 1. \end{cases}$$

As an example, we give the meaning of the term $Fib(Fib(Zero))$ in $I$. It is

$$\begin{aligned}\big(Fib(Fib(Zero))\big)^I &= Fib^I(Fib^I(Zero^I))\\ &= Fib^I(Fib^I(0))\\ &= Fib^I(1)\\ &= 1.\end{aligned}$$

We inductively define when an interpretation $I$ *satisfies* an FO formula $\varphi$, or $I$ *is a model of* $\varphi$, denoted $I \models \varphi$:

- $I \models P(\bar{t})$ if $\bar{t}^I \in P^I$;
- $I \models \varphi \lor \psi$ if $I \models \varphi$ or $I \models \psi$;
- $I \models \neg\varphi$ if $I \not\models \varphi$;
- $I \models \exists x\ \varphi$ if there exists a $d \in \mathrm{dom}(I)$ for which $I[x/d] \models \varphi$.

We also denote $I \models \varphi$ by $I(\varphi) = \boldsymbol{t}$, and $I \not\models \varphi$ by $I(\varphi) = \boldsymbol{f}$. For a theory $T$, $I \models T$ iff $I \models \varphi$ for every sentence $\varphi \in T$; $I$ is said to be a *model* of $T$. For theories or sentences $\varphi, \psi$, we define that $\varphi$ *entails* $\psi$, denoted $\varphi \models \psi$, iff for every model $I$ of $\varphi$, also $I \models \psi$ holds. This notation generalizes also the notation for *validity* of a sentence $\varphi$: by $\models \varphi$ we mean $\emptyset \models \varphi$, i.e., $\varphi$ is valid.

**Example 2.5.** Examples 2.1–2.3 continued. We have that $I_{2.2} \models T_{2.1}$. This means that $I_{2.2}$ denotes a *possible state* of the world described by $T$.

Indeed, $I_{2.2} \models Man(Philosopher)$, because $Philosopher^{I_{2.2}} = \mathsf{Socrates}$, and $\mathsf{Socrates} \in Man^{I_{2.2}}$. Also $I_{2.2} \models \forall x\ Man(x) \supset Mortal(x)$, because for each domain element $d \in \mathrm{dom}(I_{2.2})$, $I_{2.2}[x/d] \models Man(x) \supset Mortal(x)$. We illustrate this: for $d = \mathsf{Snoopy}$, $I_{2.2}[x/\mathsf{Snoopy}] \models Man(x) \supset Mortal(x)$ because $\mathsf{Snoopy} \notin Man^{I_{2.2}}$, hence the implication is satisfied; for $d = \mathsf{Socrates}$, $I_{2.2}[x/\mathsf{Socrates}] \models Man(x) \supset Mortal(x)$ because both $\mathsf{Socrates} \in Man^{I_{2.2}}$ and $\mathsf{Socrates} \in Mortal^{I_{2.2}}$, hence the implication is again satisfied.

The informal meaning of $T_{2.1}$, "the philosopher is a man, and all men are mortal", is also satisfied by $I_{2.2}$.

**Example 2.6.** Examples 2.1–2.3 and 2.5 continued. We illustrate also the concept of entailment. We have that

$$Man(Philosopher) \land \big(\forall x\ Man(x) \supset Mortal(x)\big) \models Mortal(Philosopher).$$

Indeed, in any possible world where the philosopher is a man and all men are mortal, the philosopher is mortal.

### 2.2.1 Herbrand interpretations

In logic programming, the domain is often restricted to the *Herbrand universe*, i.e., the set of all ground (variable-free) terms of the vocabulary. A *Herbrand*

*pre-interpretation* is a pre-interpretation that has the Herbrand universe as its domain and interprets each constant and function symbol by itself. A *Herbrand interpretation* is an interpretation that extends a Herbrand pre-interpretation.[1]

The models of a theory that contains the *Unique Names Axioms* (UNA) (Reiter, 1980) and the *Domain Closure Assumption* (DCA) (Reiter, 1982) are isomorphic to Herbrand interpretations.

The UNA can be expressed in FO as follows:

- a sentence $\forall \overline{x}\, \forall \overline{y}\; F_i(\overline{x}) \neq F_j(\overline{y})$, for all function symbols $F_i \neq F_j$ (including constants) in the vocabulary, and where $\overline{x}$ and $\overline{y}$ have the appropriate arity for $F_i$ and $F_j$;
- a sentence $\forall \overline{x}\ \forall \overline{y}\ F_i(\overline{x}) = F_i(\overline{y}) \supset \overline{x} = \overline{y}$, for all non-constant function symbols $F_i$ in the vocabulary, and where $\overline{x}$ and $\overline{y}$ have the appropriate arity for $F_i$.

The DCA expresses that every object in the domain is represented by a variable-free term of the vocabulary. In general the DCA cannot be represented in FO. An exception is when all function symbols of the vocabulary are constants, $C_1, \ldots, C_N$; it is then the sentence

$$\forall x\ (x = C_1 \vee \cdots \vee x = C_N).$$

This expresses that every object in the domain is represented by one of the constants.

However, the general case of the DCA (when there are non-constant function symbols) can be represented in ID-logic, using an inductive definition. We give this definition of the DCA in ID-logic in Section 3.1.6.

### 2.2.2 Propositional logic and SAT

Recall that a vocabulary is called *propositional* if it consists only of 0-ary predicate symbols. We call the predicate symbols of a propositional vocabulary *atoms*.

When $\Sigma$ is a propositional vocabulary, we assume the pre-interpretation has an empty domain. We then treat a $\Sigma$-interpretation as a function $\Sigma \rightarrow \{\boldsymbol{f}, \boldsymbol{t}\}$. We formally define the relation: let $\Sigma$ be a propositional vocabulary, $I$ a $\Sigma$-interpretation. Then for any atom $p \in \Sigma$, $I(p) = \boldsymbol{t}$ if $p^I = \{()\}$, where $()$ is the zero-ary tuple, and $I(p) = \boldsymbol{f}$ if $p^I = \emptyset$.

The fragment of FO restricted to propositional vocabularies is called *propositional calculus* (PC). A much-used fragment of PC is *conjunctive normal form* (CNF). A theory is said to be in CNF if it is a conjunction of *clauses*, where a clause is a disjunction of literals. Equivalently, a CNF theory is a *set* of clauses. An arbitrary PC theory $T$ can be transformed in linear time into a CNF theory

[1] A Herbrand interpretation is sometimes defined as *a set of ground atoms*—an object of a different type then a classical interpretation. However, there is a simple isomorphism between Herbrand interpretations as we define them here, and Herbrand interpretations as sets of ground atoms.

with an extended vocabulary, whose models have a one-to-one correspondence to the models of $T$ (Tseitin, 1968). The advantage of CNF over full PC for computational purposes is that CNF has a very simple structure.

The Boolean *satisfiability problem* (SAT) is the problem of deciding whether a given PC theory has a model. A famous result by Cook (1971) states that the SAT problem is NP-complete.

**Example 2.7.** Let $\Sigma$ be $\{p, q, r\}$, and let $T_{2.7}$ be the following CNF theory:

$$\begin{aligned}&(p \lor q),\\&(\neg p \lor \neg q),\\&(q \lor \neg r),\\&(\neg p \lor r).\end{aligned}$$

Then $T_{2.7}$ is satisfiable; indeed, the following interpretation is a model of $T_{2.7}$: $I_{2.7} = \{p \mapsto \boldsymbol{f}, q \mapsto \boldsymbol{t}, r \mapsto \boldsymbol{t}\}$. The first three clauses each contain one true literal in $I_{2.7}$, the last clause contains two.

Decision procedures for SAT were developed by Davis and Putnam (1960) en Davis et al. (1962); the procedure proposed in the latter paper is now commonly referred to as “the Davis-Putnam-Logemann-Loveland (DPLL) procedure”. Early implementations of this and other procedures were not widely successful, largely until Marques-Silva and Sakallah (1999) added *clause learning* to the DPLL procedure. Contemporary implementations of these procedures (“SAT solvers”) are very efficient (Mitchell, 2005), and have many practical applications in industry. The research domain of SAT is highly active (Kleine Büning and Zhao, 2008).

### 2.2.3 Three-valued and four-valued semantics

The purpose of propositional model generation techniques is to *construct* a model. During this construction, the eventual truth value of some atoms is still unknown—initially, *every* truth value is unknown. We treat “unknown” as a third truth value, and represent it by $\boldsymbol{u}$. This construction may also (temporally) lead to inconsistency: when some atom is set to both true and false. We treat “inconsistent” as a fourth truth value, and represent it by $\boldsymbol{i}$. In the rest of this section we extend the classical semantics of FO as given above to four-valued semantics, and explain the relation between three- and four-valued semantics.

Logic programming semantics such as the well-founded semantics (see further) are inherently three-valued. Our definition of the well-founded semantics depends on the three-valued semantics presented here. The four-valued semantics we present here is due to Belnap (1977), and generalizes Kleene's three-valued semantics.

A *four-valued* $\Sigma$*-interpretation* is a pair $I = (I_1, I_2)$ of two $\Sigma$-interpretations that extend the same pre-interpretation. Intuitively, $I_1$ underestimates what is true, and $I_2$ overestimates it. Let $P(\overline{d})$ be a domain atom. Then

– $I(P(\overline{d})) = \boldsymbol{t}$ if both $I_1(P(\overline{d})) = \boldsymbol{t}$ and $I_2(P(\overline{d})) = \boldsymbol{t}$;
– $I(P(\overline{d})) = \boldsymbol{f}$ if both $I_1(P(\overline{d})) = \boldsymbol{f}$ and $I_2(P(\overline{d})) = \boldsymbol{f}$;
– $I(P(\overline{d})) = \boldsymbol{u}$ if $I_1(P(\overline{d})) = \boldsymbol{f}$ and $I_2(P(\overline{d})) = \boldsymbol{t}$; and
– $I(P(\overline{d})) = \boldsymbol{i}$ if $I_1(P(\overline{d})) = \boldsymbol{t}$ and $I_2(P(\overline{d})) = \boldsymbol{f}$.

If for every predicate symbol $P \in \Sigma$, $P^{I_1} \subseteq P^{I_2}$ holds, then $I = (I_1, I_2)$ is said to be *consistent*. We denote this by $I_1 \subseteq I_2$. A *three-valued* $\Sigma$*-interpretation* is a consistent four-valued one. Observe that if $I$ is a three-valued interpretation, $I_1(P(\overline{d})) = \boldsymbol{t}$ implies $I_2(P(\overline{d})) = \boldsymbol{t}$, and $I_2(P(\overline{d})) = \boldsymbol{f}$ implies $I_1(P(\overline{d})) = \boldsymbol{f}$, hence $I(P(\overline{d}))$ cannot be $\boldsymbol{i}$.

A three-valued $\Sigma$-interpretation $(I_l, I_u)$ *approximates* any two-valued $\Sigma$-interpretation $I$ that has the same pre-interpretation as $I_l$ and $I_u$, and for which $I_l \subseteq I \subseteq I_u$. $I_l$ is a lower bound on the approximated interpretations (anything that is true in $I_l$, is true in any approximated interpretation), $I_u$ is an upper bound on the approximated interpretations (anything that is false in $I_u$, is false in any approximated interpretation).

**Example 2.8.** Examples 2.1–2.3 continued. Consider the three-valued $\Sigma$-interpretation $I_{2.8} = (I_l, I_u)$ with the same domain as before, $\{\mathsf{Snoopy}, \mathsf{Socrates}\}$, and with the pre-interpretation of $Philosopher$ as before: $Philosopher^{I_{2.8}} = \mathsf{Socrates}$. Let the lower bound $I_l$ have $Man^{I_l} = \emptyset$, and $Mortal^{I_l} = \{\mathsf{Snoopy}\}$, and let the upper bound $I_u$ have $Man^{I_u} = \{\mathsf{Socrates}\}$, and $Mortal^{I_u} = \{\mathsf{Snoopy}, \mathsf{Socrates}\}$.

We have

$$I_{2.8}(Man(\mathsf{Socrates})) = \boldsymbol{u} \qquad I_{2.8}(Mortal(\mathsf{Socrates})) = \boldsymbol{u}$$
$$I_{2.8}(Man(\mathsf{Snoopy})) = \boldsymbol{f} \qquad I_{2.8}(Mortal(\mathsf{Snoopy})) = \boldsymbol{t}.$$

This three-valued interpretation describes a situation in which it is unknown whether Socrates is a man, but it is certain that Snoopy is not a man, and in which it is unknown whether Socrates is mortal, but it is certain that Snoopy is mortal.

For a four-valued interpretation $I = (I_1, I_2)$ and a formula $\varphi$, we define $I \models_4 \varphi$ by induction as:

- $I \models_4 P(\overline{t})$ if $\overline{t}^{I_1} \in P^{I_1}$;
- $I \models_4 \varphi \lor \psi$ if $I \models_4 \varphi$ or $I \models_4 \psi$;
- $I \models_4 \neg\varphi$ if $(I_2, I_1) \not\models_4 \varphi$;
- $I \models_4 \exists x\ \varphi$ if there exists a $d \in \mathrm{dom}(I)$ for which $(I_1[x/d], I_2[x/d]) \models_4 \varphi$.

Observe that the two-valued satisfaction relation can be defined by means of the four-valued one: for a two-valued interpretation $I$ and a formula $\varphi$, we have that $I \models \varphi$ iff $(I, I) \models_4 \varphi$.

We introduce the notation $I(\varphi) = \boldsymbol{v}$, where $I$ is a four-valued interpretation, $\varphi$ a formula, and $\boldsymbol{v}$ a four-valued truth value.

- $I(\varphi) = \boldsymbol{t}$, meaning $\varphi$ is certainly true in $I$, if $I \models_4 \varphi$ and $I \not\models_4 \neg\varphi$;
- $I(\varphi) = \boldsymbol{f}$, meaning $\varphi$ is certainly false in $I$, if $I \models_4 \neg\varphi$ and $I \not\models_4 \varphi$;
- $I(\varphi) = \boldsymbol{u}$, meaning $\varphi$'s value is unknown in $I$, if $I \not\models_4 \varphi$ and $I \not\models_4 \neg\varphi$;
- $I(\varphi) = \boldsymbol{i}$, meaning $\varphi$'s value is inconsistent in $I$, if $I \models_4 \varphi$ and $I \models_4 \neg\varphi$.

When $I$ is a three-valued interpretation, $I \models_4 \varphi$ implies $I(\varphi) = \boldsymbol{t}$ and $I \models_4 \neg\varphi$ implies $I(\varphi) = \boldsymbol{f}$. Note that $I \models_4 \neg\varphi$ implies $I \not\models_4 \varphi$, but not vice versa: it is possible that both $I \not\models_4 \varphi$ and $I \not\models_4 \neg\varphi$ hold—then $I(\varphi) = \boldsymbol{u}$. Also, the following result relates the $\models_4$ relation for three-valued interpretations to the $\models$ relation for two-valued interpretations.

**Proposition 2.1.** *Let $I = (I_l, I_u)$ be a three-valued $\Sigma$-interpretation, and let $I'$ be a two-valued $\Sigma$-interpretation with $I_l \subseteq I' \subseteq I_u$. Let $\varphi$ be an FO formula over $\Sigma$. Then $I \models_4 \varphi$ implies $I' \models \varphi$, and $I' \models \varphi$ implies $(I_u, I_l) \models_4 \varphi$.*

The first part of this result says that for three-valued $I$, if $I \models_4 \varphi$, then $\varphi$ it is true in any $I'$ approximated by $I$, hence is certainly true. The second part says that if $\varphi$ is known to be true in some two-valued interpretation approximated by $I = (I_l, I_u)$, it is certainly true in the interpretation $(I_u, I_l)$, which overestimates what is true, and underestimates what is false.

In practice, we will never work with inconsistent interpretations or truth values; throughout the text all four-valued interpretations will be consistent and therefore three-valued, unless explicitly mentioned.

We treat a propositional three-valued $\Sigma$-interpretation as a function $\Sigma \to \{\boldsymbol{f}, \boldsymbol{u}, \boldsymbol{t}\}$. We then obtain the following truth tables for three-valued semantics:

| $\neg$ | $\boldsymbol{t}$ | $\boldsymbol{f}$ | $\boldsymbol{u}$ |
|---|---|---|---|
| | $\boldsymbol{f}$ | $\boldsymbol{t}$ | $\boldsymbol{u}$ |

| $\wedge$ | $\boldsymbol{t}$ | $\boldsymbol{f}$ | $\boldsymbol{u}$ |
|---|---|---|---|
| $\boldsymbol{t}$ | $\boldsymbol{t}$ | $\boldsymbol{f}$ | $\boldsymbol{u}$ |
| $\boldsymbol{f}$ | $\boldsymbol{f}$ | $\boldsymbol{f}$ | $\boldsymbol{f}$ |
| $\boldsymbol{u}$ | $\boldsymbol{u}$ | $\boldsymbol{f}$ | $\boldsymbol{u}$ |

| $\vee$ | $\boldsymbol{t}$ | $\boldsymbol{f}$ | $\boldsymbol{u}$ |
|---|---|---|---|
| $\boldsymbol{t}$ | $\boldsymbol{t}$ | $\boldsymbol{t}$ | $\boldsymbol{t}$ |
| $\boldsymbol{f}$ | $\boldsymbol{t}$ | $\boldsymbol{f}$ | $\boldsymbol{u}$ |
| $\boldsymbol{u}$ | $\boldsymbol{t}$ | $\boldsymbol{u}$ | $\boldsymbol{u}$ |

We sometimes extend two-valued $\sigma$-interpretations to three-valued $\Sigma$-interpretations, for $\sigma \subsetneq \Sigma$; if $I$ is a $\sigma$-interpretation and $S$ a predicate symbol in $\Sigma \setminus \sigma$, then we define the extension $I'$ to be unknown in all domain atoms of $S$. Formally, for any $P/n \in \Sigma$, $\overline{d} \in \mathrm{dom}^n(I)$, $I'(P(\overline{d})) = I(P(\overline{d}))$ if $P/n \in \sigma$, and $I'(P(\overline{d})) = \boldsymbol{u}$ otherwise. $I'$ is called the *empty extension* of $I$ to $\Sigma$.

## 2.3 Well-founded and stable model semantics

### 2.3.1 Syntax

A *rule* over $\Sigma$ is an expression of the form[2]

$$\forall \overline{x} \ \left(P(\overline{t}) \leftarrow \varphi\right), \tag{2.1}$$

[2]This syntax generalizes the syntax conventionally used in logic programming; we instantiate this to the *normal logic programming* syntax further on.

where $P$ is a predicate symbol of $\Sigma$, $\bar{t}$ is a tuple of terms of $\Sigma$, $\varphi$ is an arbitrary FO formula over $\Sigma$, and the free variables of both $\bar{t}$ and $\varphi$ are a subset of $\overline{x}$. We often leave the universal quantification of the rule implicit. Note that the symbol "$\leftarrow$" is a new symbol. We call $P$ the *head* of the rule, $\varphi$ the *body*. The empty disjunction is written as $\bot$, the empty conjunction as $\top$, when used in rule bodies. A rule with $\top$ as body is called a *fact*. Note that $\models \top$ and $\models \neg\bot$.

For simplicity of notation, and without loss of generality, we assume that rules are of the form $\forall \overline{x}\ (P(\overline{x}) \leftarrow \varphi)$, i.e., the terms in the head and the free variables in the body are precisely the variables $\overline{x}$. This form can be obtained by replacing any non-variable term $t$ in the head by a new variable $x$, and replacing the body $\varphi$ by $\varphi \wedge x = t$, and by subsequently replacing each rule of the form $\forall \overline{x}\,\overline{y}\ (P(\overline{x}) \leftarrow \varphi)$ by $\forall \overline{x}\ (P(\overline{x}) \leftarrow \exists \overline{y}\ \varphi)$.

We define a *logic program* to be a set of rules.

A *normal logic program* is a set of rules, all of which have a conjunction of literals as their body. In the context of stable model semantics for normal logic programs we use the notation standard in the literature: we use the symbol "`:-`" instead of "$\leftarrow$", "`not`" instead of "$\neg$", and "," instead of "$\wedge$", we make the universal quantification of rules implicit, and we omit the set notation $\{\ \cdot\ \}$. Thus, a rule of a normal logic program has the form

$$P(\bar{t}) \texttt{ :- } L_1, \ldots, L_m, \texttt{not}\, L_{m+1}, \ldots, \texttt{not}\, L_n, \tag{2.2}$$

where the $L_i$ are atoms of the vocabulary with free variables $\overline{x}$.

A *constraint* is an expression of the form

$$\texttt{:- } L_1, \ldots, L_m, \texttt{not}\, L_{m+1}, \ldots, \texttt{not}\, L_n. \tag{2.3}$$

Still in the context of stable model semantics, we treat a constraint (2.3) as syntactic sugar for a rule

$$False \texttt{ :- } L_1, \ldots, L_m, \texttt{not}\, L_{m+1}, \ldots, \texttt{not}\, L_n, \texttt{not}\, False,$$

where $False$ is a predicate symbol not in $\Sigma$. In next section, Example 2.13, we will explain that this forces at least one of $L_1, \ldots, L_m, \neg L_{m+1}, \ldots, \neg L_n$ to be false.

### 2.3.2 Semantics

It is customary to define the well-founded and stable model semantics for *propositional* logic programs. One then uses the Herbrand pre-interpretation to derive the first-order semantics from the propositional one. We follow this approach for the stable model semantics. However, in this work we also want to define non-Herbrand well-founded models. To this end we define the well-founded semantics on a first-order level. For a definition of the stable model semantics on a first-order level, we refer to (Denecker, 1993).

The well-founded semantics was first defined for propositional normal logic programs by Van Gelder et al. (1991), and generalized to logic programs by

Van Gelder (1993). We deviate from these definitions and follow an alternative definition by Denecker and Vennekens (2007), where the well-founded model is constructed as the limit of a *well-founded sequence*.

Let $I$ be a three-valued interpretation and $P(\overline{d})$ a domain atom. Then we denote by $I[P(\overline{d})/\boldsymbol{t}]$ respectively $I[P(\overline{d})/\boldsymbol{f}]$ the interpretation $I'$ that is identical to $I$ except that $I'(P(\overline{d})) = \boldsymbol{t}$ respectively $I'(P(\overline{d})) = \boldsymbol{f}$. We extend the notation to sets $U$ of domain atoms.

**Definition 2.1** (Well-founded sequence)**.** Let $\Delta$ be a logic program over $\Sigma$, $I$ a three-valued $\Sigma$-interpretation. A *well-founded sequence for* $\Delta$ *from* $I$ is any sequence $\langle I_i \rangle_{0 \leq i \leq n}$ of three-valued $\Sigma$-interpretations, such that $I_0 = I$, and each $I_{i+1}$ is derived from $I_i$ as follows:

*i)* $I_{i+1} = I_i[P(\overline{d})/\boldsymbol{t}]$ for some domain atom $P(\overline{d})$ such that $I_i(P(\overline{d})) = \boldsymbol{u}$, and there is a rule $\forall \overline{x}\ (P(\overline{x}) \leftarrow \varphi)$ in $\Delta$ with $I_i[\overline{x}/\overline{d}](\varphi) = \boldsymbol{t}$; or

*ii)* $I_{i+1} = I_i[U/\boldsymbol{f}]$ for some set of domain atoms $U$ such that for each $P(\overline{d}) \in U$ the following conditions hold:

- $I_i(P(\overline{d})) = \boldsymbol{u}$, and
- for each rule $\forall \overline{x}\ (P(\overline{x}) \leftarrow \varphi)$ in $\Delta$, $I_i[\overline{x}/\overline{d}, U/\boldsymbol{f}](\varphi) = \boldsymbol{f}$.

Such a set $U$ is called an *unfounded set with respect to* $I_i$.

Note that for predicates $P$ that do not occur in the head of a rule of $\Delta$, any domain atom $P(\overline{d})$ that is unknown in the given (initial) three-valued interpretation can be made false by an application of the second derivation rule (*ii*), since the qualification "for each rule $\forall \overline{x}\ (P(\overline{x}) \leftarrow \varphi)$ in $\Delta$ ..." is trivially satisfied. Or, to put it more succinctly: non-defined predicates are always false. This is a form of *closed-world reasoning* (Reiter, 1977).

**Example 2.9.** We illustrate the concept of unfounded set on a propositional vocabulary. Let $\Delta_{2.9} =$

$$\left\{ \begin{array}{l} p \leftarrow q, \\ q \leftarrow p \wedge r \end{array} \right\},$$

and consider the three-valued interpretation $I$ with $I(p) = I(q) = \boldsymbol{u}$, $I(r) = \boldsymbol{t}$. Then $\{p, q\}$ is an unfounded set with respect to $I$. We have that $I' = I[\{p,q\}/\boldsymbol{f}]$ is the interpretation with $I'(p) = I'(q) = \boldsymbol{f}$, $I'(r) = \boldsymbol{t}$; and indeed, both $I'(q) = \boldsymbol{f}$ and $I'(p \wedge r) = \boldsymbol{f}$ hold, where $q$ is the body of the only rule with $p$ in the head, and $p \wedge r$ is the body of the only rule with $q$ in the head.

A well-founded sequence is *terminal* if it cannot be extended. Denecker and Vennekens (2007) showed that, for a given logic program $\Delta$ and three-valued interpretation $I$, any terminal well-founded sequence for $\Delta$ from $I$ has the same limit.

Let $\Sigma$ be a vocabulary. Denote by $I_{H(\Sigma)}$ the three-valued $\Sigma$-interpretation that is the empty extension to $\Sigma$ of the Herbrand pre-interpretation for $\Sigma$. Hence, $I_{H(\Sigma)}$ is the three-valued Herbrand interpretation with $I_{H(\Sigma)}(P(\overline{d})) = \boldsymbol{u}$ for any domain atom $P(\overline{d})$.

**Definition 2.2** (Well-founded model)**.** Let $\Delta$ be a logic program over $\Sigma$. Then the *well-founded model of* $\Delta$ is the limit of some (any) well-founded sequence for $\Delta$ from $I_{H(\Sigma)}$.

**Example 2.10.** Let $\Delta_{2.10}$ =

$$\left\{\begin{array}{rl} (R(A,B) & \leftarrow \top), \\ (R(B,C) & \leftarrow \top), \\ (R(B,D) & \leftarrow \top), \\ \forall x,y\ (TC(x,y) & \leftarrow R(x,y)), \\ \forall x,y\ (TC(x,y) & \leftarrow \exists z\ TC(x,z) \land TC(z,y)) \end{array}\right\},$$

and let $\Sigma = vocab(\Delta_{2.10})$. We interpret $(R(A,B) \leftarrow \top)$ as an abbreviation for $\forall x,y\ (R(x,y) \leftarrow x = A \land y = B)$, and similarly for other rules. We construct the well-founded model of $\Delta_{2.10}$.

The Herbrand pre-interpretation has domain $\{\mathsf{A},\mathsf{B},\mathsf{C},\mathsf{D}\}$, and interprets the constants $A, B, C, D$ by the domain elements of the same name. $I_{H(\Sigma)}$ is the empty extension of this pre-interpretation to $\{R/2, TC/2\}$. We construct a well-founded sequence for $\Delta_{2.10}$ from $I_{H(\Sigma)}$.

- We have $I_0 = I_{H(\Sigma)}$.
- We find $I_1 = I_0[R(\mathsf{A},\mathsf{B})/\boldsymbol{t}]$, $I_2 = I_1[R(\mathsf{B},\mathsf{C})/\boldsymbol{t}]$, and $I_3 = I_2[R(\mathsf{B},\mathsf{D})/\boldsymbol{t}]$ from the first three rules, applying derivation rule (*i*).
- From rule $\forall x,y\ (TC(x,y) \leftarrow R(x,y))$ we then further derive by (*i*) the interpretations $I_4$, $I_5$ and $I_6$, which make respectively $TC(\mathsf{A},\mathsf{B})$, $TC(\mathsf{B},\mathsf{C})$ and $TC(\mathsf{B},\mathsf{D})$ true.
- Applying rule (*i*), we find $I_7 = I_6[TC(\mathsf{A},\mathsf{C})/\boldsymbol{t}]$. Indeed, using the substitution $z/\mathsf{B}$, we have that $I_6[(x,y)/(\mathsf{A},\mathsf{C})]\big(\exists z\ TC(x,z) \land TC(z,y)\big) = \boldsymbol{t}$.
- Similarly, $I_8 = I_7[TC(\mathsf{A},\mathsf{D})/\boldsymbol{t}]$.
- We now apply derivation rule (*ii*). The only rules with $R(x,y)$ as a head have either $x = A \land y = B$, $x = B \land y = C$, or $x = B \land y = D$ as body, and therefore, for any two-tuples $\overline{d}$ other than $(\mathsf{A},\mathsf{B})$, $(\mathsf{B},\mathsf{C})$ or $(\mathsf{B},\mathsf{D})$, we have that $I_8[\overline{x}/\overline{d}](\varphi) = \boldsymbol{f}$ for any rule $\forall \overline{x}\ (R(\overline{x}) \leftarrow \varphi)$ in $\Delta_{2.10}$. Hence, $\{R(\mathsf{A},\mathsf{A}),\ldots,R(\mathsf{D},\mathsf{D})\}$ is an unfounded set with respect to $I_8$, and we find $I_9 = I_8[\{R(\mathsf{A},\mathsf{A}),\ldots,R(\mathsf{D},\mathsf{D})\}/\boldsymbol{f}]$.
- Finally, consider the remaining unknown domain atoms: $U = \{TC(\mathsf{A},\mathsf{A}), TC(\mathsf{B},\mathsf{A}),\ldots,TC(\mathsf{D},\mathsf{D})\}$. This is an unfounded set with respect to $I_9$; we can therefore apply rule (*ii*), obtaining $I_{10} = I_9[U/\boldsymbol{f}]$. We show that this is indeed the case. All domain atoms in $U$ are instantiations of $TC(x,y)$; the rules with $TC(x,y)$ in the head have bodies $R(x,y)$ respectively $\exists z\ TC(x,z) \land TC(z,y)$. Indeed we find
  - for $TC(\mathsf{A},\mathsf{A})$, that both $I_9[(x,y)/(\mathsf{A},\mathsf{A})]\big(R(x,y)\big) = \boldsymbol{f}$ and $I_9[(x,y)/(\mathsf{A},\mathsf{A})]\big(\exists z\ TC(x,z) \land TC(z,y)\big) = \boldsymbol{f}$;

– for $TC(\mathsf{B},\mathsf{A})$, that both $I_9[(x,y)/(\mathsf{B},\mathsf{A})]\big(R(x,y)\big) = \boldsymbol{f}$ and $I_9[(x,y)/(\mathsf{B},\mathsf{A})]\big(\exists z\ TC(x,z) \land TC(z,y)\big) = \boldsymbol{f}$;
– ...
– for $TC(\mathsf{D},\mathsf{D})$, that both $I_9[(x,y)/(\mathsf{D},\mathsf{D})]\big(R(x,y)\big) = \boldsymbol{f}$ and $I_9[(x,y)/(\mathsf{D},\mathsf{D})]\big(\exists z\ TC(x,z) \land TC(z,y)\big) = \boldsymbol{f}$.

$I_{10}$ is two-valued, and therefore terminal. Thus, the interpretation $I_{10}$ with $R^{I_{10}} = \{(\mathsf{A},\mathsf{B}),(\mathsf{B},\mathsf{C}),(\mathsf{B},\mathsf{D})\}$ and $TC^{I_{10}} = \{(\mathsf{A},\mathsf{B}),(\mathsf{A},\mathsf{C}),(\mathsf{A},\mathsf{D}),(\mathsf{B},\mathsf{C}),(\mathsf{B},\mathsf{D})\}$ is a well-founded model of $\Delta_{2.10}$. Observe that $TC$ is the transitive closure of $R$.

In general, the well-founded model may be three-valued.

**Example 2.11.** We illustrate this on a definition with a propositional vocabulary. Let $\Delta_{2.11}$ =

$$\left\{ \begin{array}{l} p \leftarrow \neg q, \\ q \leftarrow \neg p \end{array} \right\},$$

and $\Sigma = \{p,q\}$. Then $I_{H(\Sigma)}$ is the well-founded model of $\Delta_{2.11}$. Note that $I_{H(\Sigma)}(p) = I_{H(\Sigma)}(q) = \boldsymbol{u}$. Indeed, neither the first nor the second rule of a well-founded sequence can be applied on $I_{H(\Sigma)}$. Therefore the well-founded sequence consisting of only this (initial) interpretation is terminal.

We now define the stable model semantics of logic programs. It is defined for propositional logic programs.

If a propositional logic program is *positive*, i.e. it has no negative literals in any rule body, then it has a *least model*. Let $\Sigma$ be a propositional vocabulary, $\Delta$ a positive logic program over $\Sigma$. Consider the set of $\Sigma$-interpretations $\mathcal{I} = \{I \mid \text{for any } p \in \Sigma \text{ with } I(p) = \boldsymbol{t}, \exists\,(p \leftarrow \varphi) \in \Delta \text{ with } I(\varphi) = \boldsymbol{t}\}$. Then *the least model of* $\Delta$ is the interpretation $I \in \mathcal{I}$ for which the set $\{p \mid I(p) = \boldsymbol{t}\}$ is subset-minimal.

The least model of $\Delta$ coincides with its well-founded model. It can be derived by first applying the first derivation rule ($i$) of well-founded sequences (making domain atoms true) exhaustively, and then applying the second derivation rule ($ii$) on all remaining unknown atoms.

We introduce the *reduct* program transformation originally defined by Gelfond and Lifschitz (1988). This definition assumes $\Delta$ contains no nesting of negation; this can be easily generalized.

**Definition 2.3** (Gelfond-Lifschitz reduct)**.** Let $\Delta$ be a propositional logic program over $\Sigma$, $I$ a $\Sigma$-interpretation. Then the *Gelfond-Lifschitz reduct* $\mathcal{GL}(\Delta, I)$ is the logic program obtained from $\Delta$ by replacing all occurrences (in rule bodies) of negative literals $\neg P$ by $\bot$ if $I(P) = \boldsymbol{t}$, and by $\top$ if $I(P) = \boldsymbol{f}$.

Note that the Gelfond-Lifschitz reduct of any program under any interpretation is a positive logic program, and therefore has a least model.

**Definition 2.4** (Stable model)**.** Let $\Delta$ be a propositional logic program over vocabulary $\Sigma$. Then a $\Sigma$-interpretation $I$ is a *stable model of* $\Delta$ iff $I$ is the least model of $\mathcal{GL}(\Delta, I)$.

We then define the logic of *Stable logic programs* as normal logic programs (including constraints) under the stable model semantics.

**Example 2.12.** Let $\Delta_{2.12}$ be the normal logic program

$$p \texttt{ :- not } q,$$
$$q \texttt{ :- not } p.$$

Both $I_1 = \{p \mapsto \boldsymbol{t}, q \mapsto \boldsymbol{f}\}$ and $I_2 = \{p \mapsto \boldsymbol{f}, q \mapsto \boldsymbol{t}\}$ are stable models of $\Delta_{2.12}$. For instance, we have that $\mathcal{GL}(\Delta_{2.12}, I_1) =$

$$p \texttt{ :- } \top,$$
$$q \texttt{ :- } \bot,$$

which indeed has $I_1$ as its least model. Note that the well-founded model of $\Delta_{2.12}$ is $\{p \mapsto \boldsymbol{u}, q \mapsto \boldsymbol{u}\}$ (cf. Example 2.11).

**Example 2.13.** Let $\Delta_{2.13}$ be a normal logic program containing the rule

$$False \texttt{ :- } L_1, \ldots, L_m, \texttt{not } L_{m+1}, \ldots, \texttt{not } L_n, \texttt{not } False,$$

and no other rules with $False$ in the head. Recall that this is how a constraint of the form $\texttt{:- } L_1, \ldots, L_m, \texttt{not } L_{m+1}, \ldots, \texttt{not } L_n$ is defined. Consider an interpretation $I$ with $I(False) = \boldsymbol{t}$. The reduct $\mathcal{GL}(\Delta_{2.14}, I)$ contains the rule $False \texttt{ :- } L_1, \ldots, \bot$. Its least model $I'$ has $I'(False) = \boldsymbol{f}$. Since $I(False) \neq I'(False)$, $I$ cannot be a stable model of $\Delta_{2.13}$.

Consider instead an interpretation $I$ with $I(False) = \boldsymbol{f}$. Then the reduct $\mathcal{GL}(\Delta_{2.14}, I)$ contains the rule $False \texttt{ :- } L_1, \ldots, \top$. In order for $I$ to be a stable model of $\Delta_{2.13}$, the least model $I'$ of the reduct should have $I'(False) = I(False) = \boldsymbol{f}$. Therefore, at least one of $L_1, \ldots, L_m$ must be false, or at least one of $L_{m+1}, \ldots, L_n$ true, to satisfy $\Delta_{2.13}$.

Whereas the well-founded model (of a propositional logic program) is unique and may be three-valued, the stable models are not unique, and are two-valued. However, the following result establishes a strong relationship between well-founded models and stable models.

**Proposition 2.2** (Van Gelder et al., 1991)**.** *If the well-founded model of $\Delta$ is two-valued, then $\Delta$ has a unique stable model, and it coincides with the well-founded model.*

The inverse, however, is not true. Below is an example with a unique stable model which is not its well-founded model.

**Example 2.14.** Examples 2.12 and 2.13 continued. Let $\Delta_{2.14}$ =

$$\begin{array}{ll} p & \texttt{:-}\ \texttt{not}\, q, \\ q & \texttt{:-}\ \texttt{not}\, p, \\ & \texttt{:-}\ Q. \end{array}$$

Due to the constraint $\texttt{:-}\ q$, $\Delta_{2.14}$ has a unique stable model, namely $I_1 = \{p \mapsto \boldsymbol{t}, q \mapsto \boldsymbol{f}\}$. The well-founded model of $\Delta_{2.14}$, however, is $\{p \mapsto \boldsymbol{u}, q \mapsto \boldsymbol{u}\}$.

## 2.4 ID-logic

We now introduce ID-logic, the logic for which this thesis seeks to develop efficient model generation algorithms. It was originally introduced by Denecker (2000), and further developed by Denecker and Ternovska (2004, 2008).

### 2.4.1 Syntax

ID-logic is a logic that intends to provide a formal syntax and semantics for *definitions*, and extend FO with them, as they are constructs that occur often in mathematics, but in general cannot be expressed in FO.

As an example, we recall our own definition of the satisfaction relation from Section 2.2:

> "We inductively define when an interpretation $I$ *satisfies* an FO formula $\varphi$, denoted $I \models \varphi$:
>
> - $I \models P(\bar{t})$ if $\bar{t}^I \in P^I$;
> - $I \models \varphi \lor \psi$ if $I \models \varphi$ or $I \models \psi$;
> - $I \models \neg\varphi$ if $I \not\models \varphi$;
> - $I \models \exists x\ \varphi$ if there exists a $d \in \mathrm{dom}(I)$ for which $I[x/d] \models \varphi$."

We observe the following syntactical structure (also in other examples):

- ◦ a declaration that one or more concepts will be defined in terms of other concepts;
- ◦ the actual definition, consisting of a set of *rules*. Each of these rules is of the form "[defined concept] **if** [logical expression]".

These observations justify the following formal syntax, and its informal meaning.

Like a logic program, a *definition* is a set of rules. Recall that a rule is of the form (2.1):

$$\forall \bar{x}\ \left(P(\bar{t}) \leftarrow \varphi\right).$$

In ID-logic, we call the rule symbol "$\leftarrow$" *definitional implication*, and in particular, we distinguish it from (the inverse of) material implication, "$\supset$". If $\Delta$ is a definition over vocabulary $\Sigma$, we denote by $Def(\Delta)$ the set of predicate

symbols, called the *defined symbols*, appearing in the head of some rule of $\Delta$, and by $Open(\Delta)$, called the *open symbols*, the symbols $\Sigma \setminus Def(\Delta)$.

The *informal reading* of an ID-logic definition $\Delta$ is of a mathematical definition of the concepts $Def(\Delta)$, i.e., "we define $Def(\Delta)$ (by induction) as [the set of rules in $\Delta$]". The definitional implication "$\leftarrow$" should be read as "if"; a body of a rule gives a sufficient condition to derive the head of the rule. The set of all rule bodies of a definition gives a necessary condition to derive a defined predicate. This declarative reading is supported by the formal semantics (see next section), which was shown by Denecker (1998) to coincide with the semantics of definitions as occurring in mathematics, for most common types of definitions.

An *ID-logic theory* is a set of definitions and FO sentences.[3] Thus ID-logic extends first-order logic with a definition language construct. As such, ID-logic has recently often been called FO(ID).

**Example 2.15.** The following theory $T_{2.15}$ is a well-formed ID-logic theory, consisting of four definitions and three FO sentences.

$$\left\{ \ \forall x,y\ (Sibling(x,y) \leftarrow x \neq y \land \exists z\ Parent(z,x) \land Parent(z,y)) \ \right\},$$
$$\left\{ \ \forall x,y\ (Brother(x,y) \leftarrow Male(x) \land Sibling(x,y)) \ \right\},$$
$$\left\{ \ \forall x,y\ (Sister(x,y) \leftarrow Female(x) \land Sibling(x,y)) \ \right\},$$
$$\forall x,y\ Husband(x,y) \equiv Wife(y,x),$$
$$\forall x,y\ Husband(x,y) \supset Male(x) \land Female(y),$$
$$\forall x\ Male(x) \equiv \neg Female(x),$$
$$\left\{ \begin{array}{lll} \forall x,y & (Uncle(x,y) & \leftarrow \exists z\ Brother(x,z) \land Parent(z,y)), \\ \forall x,y & (Uncle(x,y) & \leftarrow \exists z\ Husband(x,z) \land Aunt(z,y)), \\ \forall x,y & (Aunt(x,y) & \leftarrow \exists z\ Sister(x,z) \land Parent(z,y)), \\ \forall x,y & (Aunt(x,y) & \leftarrow \exists z\ Wife(x,z) \land Uncle(z,y)) \end{array} \right\}. \qquad (\Delta_{2.15})$$

Consider definition $\Delta_{2.15}$. It is a *definition by simultaneous induction*: an uncle is defined in terms of an aunt, and conversely, an aunt in terms of an uncle. The informal meaning of $\Delta_{2.15}$ corresponds to the definition of the uncle and aunt relations in natural language. In particalur, consider a married couple, Ann and Bob, and neither of the two has siblings. Then, although there is a circular dependency (Ann is the aunt of some person $y$ if Bob is the uncle of $y$, and vice versa), it is clear that Ann is not an aunt of anybody, nor is Bob an uncle of anybody.

### 2.4.2 Semantics

Unlike the well-founded and the stable model semantics, an inductive definition may have a model *for every given interpretation of* $Open(\Delta)$. To define this, we begin by introducing a parameterized version of the well-founded semantics.

[3] ID-logic is sometimes defined to have a more general syntax, whereby definitions are allowed to be nested as subformulas in FO sentences (see, e.g., Denecker and Ternovska, 2008).

**Definition 2.5** (Well-founded model extending $I_O$)**.** Let $\Delta$ be a definition over $\Sigma$, and let $I_O$ be an $Open(\Delta)$-interpretation. Then define $I'_O$ as the empty three-valued extension of $I_O$ to $Def(\Delta)$ (i.e., all domain atoms of $Def(\Delta)$ are unknown). The *well-founded model of* $\Delta$ *extending* $I_O$, denoted $\text{wfm}_\Delta(I_O)$, is the limit of any well-founded sequence for $\Delta$ from $I'_O$.

**Definition 2.6** (Model of $\Delta$)**.** Let $\Delta$ be a definition over $\Sigma$, and let $I$ be a (two-valued) $\Sigma$-interpretation. Then $I$ *is a model of* $\Delta$, denoted $I \models \Delta$, iff $I = \text{wfm}_\Delta(I|_{Open(\Delta)})$.

We point out the correspondences and differences with the well-founded model of a logic program:

- a model of a definition is by definition two-valued, the well-founded model of a logic program may be three-valued;
- a definition $\Delta$ may have many (or no) models: for every $Open(\Delta)$-interpretation at most one; a logic program has a unique well-founded model;
- the well-founded model of a logic program is a Herbrand model, a model of a definition may have an arbitrary pre-interpretation;
- if the well-founded model $I$ of a logic program $\Delta$ is two-valued, then $I$ is also a model of definition $\Delta$, namely, with $I$ false for every domain atom of the non-defined symbols of $\Delta$. Or conversely, if $\Delta$ is a definition, and $\Delta'$ is the logic program obtained from $\Delta$ by adding $\forall \overline{x}\ (P(\overline{x}) \leftarrow \bot)$ for each $P \in Open(\Delta)$, then if the well-founded model of $\Delta'$ is two-valued, it is a model of $\Delta$.

We restate the original intention of ID-logic: to provide a formal syntax and semantics for definitions as they occur in mathematics (and add this to FO). It was shown by Denecker (1998) that the formal semantics of definitions coincides with the semantics of definitions as occurring in mathematics. Intuitively, $Open(\Delta)$ are the concepts that are *used* in the definition $\Delta$, and $Def(\Delta)$ are the concept that are being defined.

**Example 2.16.** Let $\Delta_{2.16} =$

$$\left\{ \begin{array}{l} \forall x, y\ (TC(x,y) \leftarrow R(x,y)), \\ \forall x, y\ (TC(x,y) \leftarrow \exists z\ TC(x,z) \wedge TC(z,y)) \end{array} \right\}.$$

We have $Def(\Delta_{2.16}) = \{TC\}$, $Open(\Delta_{2.16}) = \{R\}$. Consider the $Open(\Delta_{2.16})$-interpretation $I_O$ with $\text{dom}(I_O) = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}$ and $R^{I_O} = \{(\mathsf{A}, \mathsf{B}), (\mathsf{B}, \mathsf{C}), (\mathsf{B}, \mathsf{D})\}$. Refer to Example 2.10 to see that the well-founded model of $\Delta_{2.16}$ extending $I_O$ is $\text{wfm}_{\Delta_{2.16}}(I_O) = I_{2.16}$ with $\text{dom}(I_{2.16}) = \text{dom}(I_O)$, $R^{I_{2.16}} = R^{I_O}$ and $TC^{I_{2.16}} = \{(\mathsf{A}, \mathsf{B}), (\mathsf{A}, \mathsf{C}), (\mathsf{A}, \mathsf{D}), (\mathsf{B}, \mathsf{C}), (\mathsf{B}, \mathsf{D})\}$. Thus $I_{2.16} \models \Delta_{2.16}$.

With the semantics of definitions in place, the semantics of ID-logic is a simple extension of FO semantics.

**Definition 2.7** (Model of an ID-logic theory)**.** Let $T$ be an ID-logic theory over $\Sigma$, and $I$ a (two-valued) $\Sigma$-interpretation. Then $I$ *is a model of* $T$, denoted $I \models T$, iff $I \models \varphi$ for every FO sentence or definition $\varphi \in T$.

**Example 2.17.** Example 2.16 continued. Let $T_{2.17} = \{\Delta_{2.16}, \neg(\exists x\ TC(x,x))\}$. Then any interpretation $I$ in which $R^I$ describes an acyclic graph is a model of $T_{2.17}$. For instance, $I_{2.16} \models T_{2.17}$, because $I_{2.16} \models \Delta_{2.16}$, and $I_{2.16} \models \neg(\exists x\ TC(x,x))$.

We define also the entailment relation $\models$ between ID-logic theories $T_1$ and $T_2$ as it was done for FO: $T_1 \models T_2$ if for every model $I \models T_1$, also $I \models T_2$ holds. We derive from this the congruence relation $\cong$: $T_1 \cong T_2$ if $T_1 \models T_2$ and $T_2 \models T_1$, i.e., $T_1$ and $T_2$ have the same models. Note that a particular case is when $T_1$ or $T_2$ consists of a single definition.

ID-logic has both monotone and non-monotone aspects. The logic as a whole is monotone: if a sentence $\Psi$ or a definition $\Delta$ is added to a theory $T$, the consequences of $T$ are retained (and may grow). I.e., $T \models \varphi$ implies $T \cup \{\Psi\} \models \varphi$ and $T \cup \{\Delta\} \models \varphi$. Non-monotonicity is present *within* definitions: if a rule is added to a definition $\Delta$, the consequences of $\Delta$ may decrease.

### Propositional ID-logic

**Definition 2.8** (PC(ID), SAT(ID))**.** The propositional fragment of ID-logic is called *PC(ID)* . The satisfiability problem of PC(ID) is called *SAT(ID)*.

Observe that PC(ID) is an extension of PC with propositional inductive definitions (IDs).

As we will see in Chapter 5, the SAT(ID) problem of a given PC(ID) theory is solved in practice by searching for a model of that theory. Hence, if we can solve the SAT(ID) problem, we can also solve the model generation problem for PC(ID), i.e., the *propositional* model generation problem for ID-logic.

*Remark* 2.1. The semantics of propositional inductive definitions is just derived from that on the first-order level. Therefore, the intended semantics (informal meaning) of propositional inductive definitions is derived from the intended semantics of first-order definitions.

Consider, for instance, the definition $\left\{\ p \leftarrow q, \quad q \leftarrow p\ \right\}$. Its declarative reading is "we define $p, q$ as: $p$ is true if $q$ is true, and $q$ is true if $p$ is true". Formally, the only model of this definition is $\{p \mapsto \boldsymbol{f}, q \mapsto \boldsymbol{f}\}$. We illustrate why this is also the intended model.

Recall $\Delta_{2.15}$ from Example 2.15. If we instantiate this definition with constants *Ann* and *Bob*, who are married and have no siblings, and some constant $Y$ denoting an arbitrary person, and then simplify the definition, we obtain:

$$\left\{\begin{array}{ll} Uncle(Bob,Y) & \leftarrow Aunt(Ann,Y), \\ Aunt(Ann,Y) & \leftarrow Uncle(Bob,Y) \end{array}\right\},$$

which is of the form $\left\{\ p \leftarrow q, \quad q \leftarrow p\ \right\}$. As we have seen in Example 2.15, its informal meaning is that $Uncle(Bob,Y)$ and $Aunt(Ann,Y)$ are false.

**Totality**

Mathematicians often follow certain linguistic conventions when formulating inductive definitions. For instance, for non-monotone inductive definitions, they often specify that the definition is by induction *on some given well-founded order*. That is, the definition is meant to be used in a setting where that well-founded order is imposed, and is ill-defined otherwise.

Whether or not a (syntactically well-constructed) definition is (semantically) well-defined is expressed in ID-logic using the semantical concept of *totality*. The formal definition of totality is based on the observation that if the well-founded order over which the induction is specified is *not* imposed, then the definitional construction process may not be able to derive the truth or falsity of certain tuples in the defined relation—the well-founded model may be three-valued.

**Definition 2.9** (Totality)**.** Let $\Delta$ be a definition over $\Sigma$, $T$ an ID-logic theory over $\Sigma$. Then $\Delta$ *is total with respect to* $T$ iff for any $\Sigma$-model $I$ of $T$, $\mathrm{wfm}_\Delta(I|_{Open(\Delta)})$ is two-valued. $\Delta$ is *total* if it is total with respect to the empty theory (and therefore with respect to *any* theory).

Intuitively, then, a definition $\Delta$ is a valid definition only in conjunction with a theory $T$ such that $\Delta$ is total with respect to $T$.

It is easy to derive that positive definitions are always total. We show an example of a definition that is total only with respect to a specific theory.

**Example 2.18.** Van Gelder et al. (1991) introduced the following definition of the winning states of a two-player game with alternating players: $\Delta_{2.18} =$

$$\left\{ \;\forall x\; (Win(x) \quad \leftarrow \exists y\; Move(x,y) \land \neg Win(y)) \;\right\}.$$

Here, $Move/2$ describes the valid transitions between states of the game; a winning state is one for which there exists a move (transition) to a losing state.

$\Delta_{2.18}$ is not total. For instance, its well-founded model extending $I_O$ with $\mathrm{dom}(I_O) = \{1,2\}$ and $Move^{I_O} = \{(1,2),(2,2)\}$ is three-valued (it is unknown in both $Win(1)$ and $Win(2)$).

But the definition is intended to be used in situations where the $Move/2$ relation describes an acyclic graph with finite branches, i.e., where there is a well-founded order $<$ on the game states such that $Move(x,y)$ implies $y < x$. This is expressed formally by saying that $\Delta_{2.18}$ is total with respect to $T_{2.18}$, where $T_{2.18}$ is the following theory:

$$\begin{aligned}
&\forall x,y\; Move(x,y) \supset y < x,\\
&\neg(\exists x\; x < x),\\
&\forall x,y,z\; x < y \land y < z \supset x < z,\\
&\left\{\; \forall x\; (F_<(x) \leftarrow \forall y\; y < x \supset F_<(y)) \;\right\},\\
&\forall x\; F_<(x).
\end{aligned}$$

Indeed, when the $Move/2$ relation satisfies the above, then the well-founded model of $\Delta_{2.18}$ can be constructed by first assigning $Win$ to be $\boldsymbol{f}$ for all final

states (from which no move is possible), and then propagating these values to earlier states along the given order; since the graph described by $Move/2$ is acyclic, all states will in this way eventually be assigned a two-valued truth value.

# Chapter 3

# Knowledge representation in ID-logic

This chapter presents ID-logic from a knowledge representation point of view. We provide some methodological principles of modelling in ID-logic in Section 3.1, and some examples of modellings in ID-logic in Section 3.3. Most of these examples fit in the computational paradigm of *model expansion*—a generalization of finite model generation—which we introduce in Section 3.2.

In Section 3.4, we then present a concrete system to solve model expansion problems for ID-logic: the IDP system. Such problems are solved in two phases: a grounding phase, which reduces an ID-logic theory and a finite domain to a PC(ID) theory, and a propositional model generation phase, which finds models of a given PC(ID) theory. We present a simple grounding technique in Section 3.5, and establish some simple complexity results about the satisfiability problem for PC(ID) in Section 3.6; these two sections make the bridge to later chapters, which elaborate on the second phase. We conclude in Section 3.7.

## 3.1 On modelling in ID-logic

The work in this section is based on (Mariën et al., 2004, 2006). In accordance with practical use in the IDP system, we use many-sorted logic (see, e.g., Enderton, 2001; for details of the IDP system, see Wittocx and Mariën, 2008). When needed, we will start the declaration of a theory by a declaration of types (sorts) and vocabulary as follows:

| | |
|---|---|
| **types:** | [A declaration of all types of the vocabulary] |
| **vocabulary:** | [A declaration of the vocabulary, including type information]. |

We will sometimes use arithmetic operations such as $+$, $-$, $abs(\cdot)$. It is beyond the scope of this work to elaborate on IDP's treatment of such operations; instead, we refer to (Wittocx et al., 2008a).

### 3.1.1 Definitional knowledge

An important form of knowledge, present in almost any non-trivial domain, is *definitional knowledge*: knowledge of certain concepts in the domain that are deterministically defined in terms of other concepts. Examples abound: for instance, in the domain of planar geometry, the concept *square* is defined in terms of simpler concepts:

$$\text{An object is a } \textit{square} \text{ if it is a } \textit{rectangle} \text{ that is } \textit{equilateral}; \tag{3.1}$$

in the domain of propositional logic, the concept of *satisfaction* ($\models$) between an interpretation and a formula, is defined in terms of the structure of the formula and the interpretation of trivial formulas:

$$\begin{array}{l} \bullet\ I \models \Psi \text{ if } \Psi \text{ is an atom, } P\text{, and } I(P) = \boldsymbol{t}, \\ \bullet\ I \models \Psi \text{ if } \Psi \text{ is } \Psi_1 \wedge \Psi_2 \text{ and both } I \models \Psi_1 \text{ and } I \models \Psi_2, \\ \bullet\ I \models \Psi \text{ if } \Psi \text{ is } \neg\Phi \text{ and not } I \models \Phi; \end{array} \tag{3.2}$$

and so on for many domains.

ID-logic provides an explicit language construct to represent this type of knowledge. Thus the intuitive definitions (3.1) and (3.2) can be straightforwardly represented as ID-logic definitions (3.3) respectively (3.4) (using the obvious interpretations for all symbols used):

$$\left\{\ Square(x) \leftarrow Equilateral(x) \wedge Rectangle(x)\ \right\}, \tag{3.3}$$

$$\left\{ \begin{array}{ll} \forall I, P & \big(\models(I,P) \leftarrow IsAtom(P) \wedge True(I,P)\big), \\ \forall I, \Psi & \big(\models(I,\Psi) \leftarrow IsConj(\Psi,\Psi_1,\Psi_2) \wedge \models(I,\Psi_1) \wedge \models(I,\Psi_2)\big), \\ \forall I, \Psi & \big(\models(I,\Psi) \leftarrow IsNeg(\Psi,\Phi) \wedge \neg\models(I,\Phi)\big) \end{array} \right\}. \tag{3.4}$$

Hence, ID-logic supports a natural, modular representation of definitions as formal objects; it is part of our ID-logic modelling methodology to use the definition construct *exclusively* for this purpose. Since in standard mathematical practice, definitions are always well-defined, and well-defined definitions are total (with respect to the theory in which they occur, cf. Section 2.4.2), an interesting consequence is that theories that are modelled according to this methodology never contain non-total definitions.

Throughout this section we will refer to an example from the well-known *block's world* domain. We give a full account of this example in Section 3.3.1; here we mention some details for illustrative purposes. The block's world domain contains amongst others the concepts $On$, $Free$, and $Move$. The latter describes the movements of blocks to other blocks over time and is usually the relation we want to construct; $On$ and $Free$ are concepts of which we have definitional knowledge. The former specifies which blocks are on top of which other blocks at any given timepoint, and is defined in terms of the initial positioning of blocks and their movements over time, the latter is of a block having no other block on

top of it at a given timepoint. It is easy to see that definitions (3.5) and (3.6) are a correct representation of these concepts:

$$\left\{ \begin{array}{lll} \forall b_1, b_2 & \big(On(b_1, b_2, Init) \leftarrow & InitiallyOn(b_1, b_2)\big), \\ \forall b_1, b_2, t & \big(On(b_1, b_2, t) \leftarrow & Move(b_1, b_2, t-1)\big), \\ \forall b_1, b_2, t & \big(On(b_1, b_2, t) \leftarrow & On(b_1, b_2, t-1) \wedge \\ & & \neg(\exists b_3\ Move(b_1, b_3, t-1))\big) \end{array} \right\}, \tag{3.5}$$

$$\left\{\ \forall b, t\ \big(Free(b,t) \leftarrow \neg(\exists b'\ On(b', b, t))\big)\ \right\}. \tag{3.6}$$

### 3.1.2 Modularity

Part of our ID-logic modelling methodology is to provide *separate* definitions whenever possible. This enhances modularity and readability. Examples where such a separation is not possible, are definitions by simultaneous induction, such as the definition of uncles and aunts in Example 2.15. In the same example, however, we did provide separate definitions for the concepts sibling, brother, and sister, since they can be defined independently.

Note that as a side-effect of the fact that ID-logic allows users to write separate definitions, one can introduce different definitions for the same concept. While for general methodological purposes this feature should not be considered more than just an interesting side-effect, the following example illustrates that it may sometimes be useful.

**Example 3.1.** Consider definition (3.1) of the concept *square* on the one hand, and the next definition on the other hand:

An object is a *square* if it is a *rhombus* that is *orthogonal*. (3.7)

Both definitions can coexist in one ID-logic theory:

$$T_{3.1} = \left[ \begin{array}{l} \left\{\ \forall x\ (Square(x) \leftarrow Equilateral(x) \wedge Rectangle(x))\ \right\}, \\ \left\{\ \forall x\ (Square(x) \leftarrow Orthogonal(x) \wedge Rhombus(x))\ \right\} \end{array} \right].$$

This theory contains two definitions for the predicate $Square/1$. Note that together these definitions constrain their open predicates; for example, $T_{3.1}$ logically entails the formula $\forall x\ (Equilateral(x) \wedge Rectangle(x) \equiv Orthogonal(x) \wedge Rhombus(x))$.

The rationale behind our proposed principle of providing separate definitions, i.e., separate *modules of knowledge*, whenever possible is very simple: it is much easier to verify whether several small definitions correctly model your intentions than it is to verify the same for one large definition. Similarly, in FO, one prefers to write small sentences, each of which represents a single isolated property, over writing a large conjunction of sentences.

When a knowledge engineer writes separate definitions of each of which he knows that it correctly models his intentions, he need not even consider the question of whether the resulting theory is still correct. By contrast, if he

merges the separate definitions into one big definition, the question becomes very relevant.

For instance, in the block's world domain, suppose that we want to define the strategy of moving always the first (according to a given total order $<$ over blocks) free block to the last free block. This can be modelled by the definition

$$\left\{ \ \forall b_1, b_2, t \ \big( Move(b_1, b_2, t) \leftarrow \forall b' \ Free(b', t) \supset b' \geq b_1 \wedge b' \leq b_2 \big) \ \right\}. \qquad (3.8)$$

Note that definitions (3.5), (3.6) and (3.8) have a circular dependency: $On$ depends (a.o.) on $Move$, $Move$ on $Free$, and $Free$ on $On$. Therefore it is not at all obvious whether the definition consisting of all the rules of definitions (3.5)–(3.8) together also correctly models our intentions. In fact it does, as can be shown using the modularity theorem by Vennekens and Denecker (2005); however, having to prove this is an inconvenience easily avoided by applying our principle.

### 3.1.3 FO sentences

Naturally, many domains also contain non-definitional knowledge. In our ID-logic methodology we represent such knowledge by FO sentences.[1] These sentences should be as *declarative* as possible, i.e., their informal reading should be as close as possible to the natural language reading of the knowledge being expressed. It is one of the strong points of FO that many natural language statements *can* be declaratively expressed in it.

An example in the block's world domain is the knowledge that a precondition for moving block $x$ to block $y$, is that both are free:

$$\forall x \forall y \forall t \ Move(x, y, t) \supset Free(x, t) \wedge Free(y, t). \qquad (3.9)$$

Also existentially quantified sentences (constraints) are often useful. Consider the statement "there is a final timepoint after which no moves are made". This could be expressed in ID-logic (FO) by the sentence

$$\exists t \ \forall t' \ \big( t' \geq t \supset \neg(\exists x, y \ Move(x, y, t')) \big). \qquad (3.10)$$

For comparison, assume that a constant $F$, representing the final timepoint, is given; the representation of the original statement then becomes

$$\forall t \ \big( t \geq F \supset \neg(\exists x, y \ Move(x, y, t)) \big). \qquad (3.11)$$

Notice the shared structure of expressions (3.10) and (3.11); this is an indication of the fact that the declarative reading of these expressions is close to the natural language statement being represented.

[1]Naturally, domains with other types of knowledge, such as causal knowledge, statistical knowledge, ..., may require other logics altogether; it is out of the scope of this work to consider these.

### 3.1.4 FO rule bodies

ID-logic allows arbitrary FO formulas as rule bodies. Recall, for instance, the rule in definition (3.5):

$$\forall b_1, b_2, t \ \big(On(b_1, b_2, t) \leftarrow On(b_1, b_2, t-1) \wedge \neg(\exists b_3 \ Move(b_1, b_3, t-1))\big).$$

Note that because FO rule bodies are allowed, it is possible to write a set of rules $\forall \overline{x} \ P(\overline{x}) \leftarrow \varphi_1[\overline{x}], \ldots, \forall \overline{x} \ P(\overline{x}) \leftarrow \varphi_N[\overline{x}]$ as one single rule, $\forall \overline{x} \ P(\overline{x}) \leftarrow \varphi_1[\overline{x}] \vee \cdots \vee \varphi_N[\overline{x}]$. (They are formally equivalent.)

A good methodological practice is to have each rule in a definition correspond to one sufficient condition for making the defined concept true; informally, to one "case". Consider, for instance, the natural language specification of when one block is on top of another block in the block's world: "a block is on top of another one on the initial timepoint *if* it is so given initially; a block is on top of another one *if* it was moved there on the previous timepoint; and also *if* it was already there on the previous timepoint and has not been moved since." The three rules of definition (3.5) correspond to these three cases.

When the syntax of rule bodies is restricted, as in normal logic programs, this methodological practice cannot be followed. For instance, to represent the above rule for $On/3$ in normal logic programming syntax, a new concept representing the subformula $(\exists b_3 \ Move(b_1, b_3, t-1))$ has to be introduced and defined.

We will not elaborate further on the meaning of different types of definitions, and the methodological impact this has on writing sets of rules. Suffice it here to say that the semantics of definitions provided by ID-logic captures the mathematical concept of "definition", and that this unifies several types of definitions, to wit: non-inductive definitions, monotone inductive definitions, non-monotone inductive definitions over a well-founded order, iterated inductive definitions, simultaneous inductive definitions, etc. For a comprehensive account, we refer to (Denecker, 1998).

### 3.1.5 On the use of function symbols

Since ID-logic works with classical interpretations, function symbols can be used freely. Functions abound in computational problem solving; often the solution of some computational problem is a function. Consider for instance the graph colouring example.

**Example 3.2.** A *graph colouring* of a given graph, represented here by $Edge$, with a given set of colours, is a function, represented here by $Colour$, that assigns a colour to each vertex of the graph, such that neighbouring vertices are assigned different colours. The following theory models this problem.

$$\begin{aligned} \textbf{types:} \ & Vtx, Clr \\ \textbf{vocabulary:} \ & Edge(Vtx, Vtx), Colour(Vtx) : Clr \\ \forall v_1 \forall v_2 \ & Edge(v_1, v_2) \supset (Colour(v_1) \neq Colour(v_2)) \end{aligned} \tag{3.12}$$

We have seen typical uses of *constant* symbols in expressions (3.5) and (3.11): the initial timepoint $Init$, and the final timepoint $F$. Often the intended interpretation of constant symbols is given, but this is not necessarily the case. Consider, for instance, a concept such as the mayor of a town. We may wish to represent this concept by a function symbol $Mayor$, without knowing who it represents.

In finite model generation, a function can be represented by a predicate symbol called its *graph*; this requires the addition of an axiom stating that this predicate symbol represents a function. For instance, the function $Colour(Vtx) : Clr$ above can be represented by $PColour(Vtx, Clr)$; this requires the axioms

$$\forall v\ \exists c\ PColour(v, c),$$
$$\forall v, c_1, c_2\ PColour(v, c_1) \wedge PColour(v, c_2) \supset c_1 = c_2.$$

However, it is good methodological practice to use function symbols whenever the expressed concept *is* a function. The reason is that this enhances readability for at least two reasons:

- no extra axioms are necessary;
- function symbols are used as *terms* and therefore often require fewer variables to be used. Compare, e.g., the following two representations of "... are assigned different colours":

  $$Colour(v_1) \neq Colour(v_2),$$
  $$\neg(\exists c\ PColour(v_1, c) \wedge PColour(v_2, c)).$$

  Observe, also, that the intended interpretation really is that of a term, e.g., $Colour(v)$ represents the domain object "the colour of $v$".

### 3.1.6 UNA and DCA not default

Recall from Section 2.2.1 that all models of a theory augmented with the Unique Names Axioms (UNA) and the Domain Closure Assumption (DCA) are isomorphic with a Herbrand model. Such a model has a one-to-one correspondence between variable-free terms and domain elements. Thus, each domain element is represented by a unique term.

The UNA and DCA are not assumed in ID-logic. Therefore, ID-logic theories may have non-Herbrand models. However, both UNA and DCA can be represented in ID-logic; UNA can actually be represented in FO, but DCA cannot. We represent the DCA here in ID-logic. It expresses that every object in the domain we are describing is represented by a variable-free term of the vocabulary. A theory that contains the DCA but not the UNA has the property that in its models, each domain element is represented by at least one variable-free term, and possibly by more.

$$\left\{\begin{array}{rcl} U(C_1) & \leftarrow & \top, \\ & \vdots & \text{(a rule for each constant } C_i \text{ in the vocabulary)} \\ U(C_M) & \leftarrow & \top, \\ \forall \overline{x}\ \big(U(F_1(\overline{x})) & \leftarrow & \bigwedge_{x_i \in \overline{x}} U(x_i)\big), \\ & \vdots & \text{(a rule for each non-constant function} \\ & & \text{symbol } F_i \text{ in the vocabulary)} \\ \forall \overline{x}\ \big(U(F_N(\overline{x})) & \leftarrow & \bigwedge_{x_i \in \overline{x}} U(x_i)\big) \end{array}\right\},$$

$\forall x\ U(x)$.

The free use of function symbols in ID-logic (cf. Section 3.1.5) is a consequence of the fact that UNA and DCA are not default.

The IDP system (see Section 3.4) also allows a localized form of UNA and DCA. It uses a typed language; when declaring a type, the language allows a declaration of a "local Herbrand universe": a set of constants with that type, and interpreted by themselves in any model. This may be useful for certain fixed domains, e.g. a type *direction* with constants $\{N, E, S, W\}$.

## 3.2 Model expansion

Given a theory, model expansion is the computational task of finding a model that expands a given interpretation of a given subvocabulary of the theory. This task generalizes finite model generation, and has many practical applications. Mitchell and Ternovska (2005) proposed model expansion for FO and for ID-logic as a framework for solving NP problems.

**Definition 3.1** ($MX$)**.** Let $\mathcal{L}$ be a logic. *Model expansion for* $\mathcal{L}$, denoted $\mathcal{L}$-$MX$ is the following decision problem. An instance of the problem consists of a $\mathcal{L}$-theory $T$, a vocabulary $\sigma \subseteq vocab(T)$, and a finite $\sigma$-interpretation $I_\sigma$. The problem is to decide whether there exists a $vocab(T)$-interpretation $I$ such that $I \models T$ and $I|_\sigma = I_\sigma$.

We denote $\mathcal{L}$-$MX$ simply by $MX$ if $\mathcal{L}$ is clear from the context. The vocabulary $\sigma$ is called the *input vocabulary*, the finite $\sigma$-interpretation $I_\sigma$ is called *input interpretation*, the vocabulary $vocab(T) \setminus \sigma$ is called *expansion vocabulary*, and the $vocab(T)$-model $I$ is called an *expansion model of* $I_\sigma$ *for* $T$.

Consider $\mathcal{L} = FO$ and $\mathcal{L} =$ ID-logic. Mitchell and Ternovska (2005) proved that FO-$MX$ and ID-logic-$MX$ are NEXP-complete.

**Definition 3.2** (Parameterized $MX$)**.** Let $\mathcal{L}$ be a logic, $T$ a $\mathcal{L}$-theory, $\sigma$ a subvocabulary of $vocab(T)$. Then the problem $\mathcal{L}$-$MX_{\langle T,\sigma\rangle}$ is the problem of deciding, for a given finite $\sigma$-interpretation $I_\sigma$, whether there exists an expansion model of $I_\sigma$ for $T$.

Observe that if $\sigma = vocab(T)$, then $MX_{\langle T,\sigma\rangle}$ reduces to model checking, while if $\sigma = \emptyset$, the problem is that of deciding the existence of a model of $T$

with a given finite size. Hence (parameterized) model expansion generalizes finite model generation for the case where model size is given.

**Example 3.3.** Recall the graph colouring problem in Example 3.2. Our ID-logic encoding of this problem was the following theory $T_{3.3}$:

$$\begin{aligned} \textbf{types: } & Vtx, Clr \\ \textbf{vocabulary: } & Edge(Vtx, Vtx), Colour(Vtx) : Clr \\ & \forall v_1 \forall v_2\ Edge(v_1, v_2) \supset (Colour(v_1) \neq Colour(v_2)). \end{aligned}$$

Let the input vocabulary $\sigma$ be $\{Edge(Vtx, Vtx)\}$. Then the problem $MX_{\langle T_{3.3}, \sigma \rangle}$ is the problem of deciding for a given finite $\sigma$-interpretation, i.e., for a given finite graph and set of colours, whether the graph is colourable.

For instance, let the input interpretation be $I_\sigma$: $I_\sigma$ has domains $Vtx = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ and $Clr = \{\mathsf{R}, \mathsf{G}, \mathsf{B}\}$ and interprets $Edge$ by $Edge^{I_\sigma} = \{(\mathsf{a}, \mathsf{b}), (\mathsf{b}, \mathsf{c}), (\mathsf{c}, \mathsf{a})\}$. This interpretation represents the graph $\mathsf{a} \to \mathsf{b} \to \mathsf{c}$ (with an edge from $\mathsf{c}$ back to $\mathsf{a}$) and three colours.

Then $MX_{\langle T_{3.3}, \sigma \rangle}$ with input $I_\sigma$ is satisfiable (i.e., the answer to the decision problem is "yes"), as witnessed by, for instance, the expansion model $I$ that extends $I_\sigma$ with $Colour^I = \{\mathsf{a} \mapsto \mathsf{R}, \mathsf{b} \mapsto \mathsf{G}, \mathsf{c} \mapsto \mathsf{B}\}$.

Parameterized $MX$ for FO is implemented in the MXG solver by Mohebali (2007).

For a given $\mathcal{L}$-theory $T$, input vocabulary $\sigma$, and decision problem $X$ on finite $\sigma$-structures, we say that $\mathcal{L}$-$MX_{\langle T, \sigma \rangle}$ *expresses* $X$ when a finite $\sigma$-interpretation $I_\sigma$ belongs to $X$ iff there exists an expansion model of $I_\sigma$ for $T$. Mitchell and Ternovska proved that parameterized model expansion for FO and for ID-logic *captures* NP. This means the following:

- for any $T$ and $\sigma \subseteq vocab(T)$, the problem $MX_{\langle T, \sigma \rangle}$ is in NP;
- for any NP decision problem $X$ on the class of finite $\sigma$-interpretations, there is a theory $T$ with $vocab(T) \supseteq \sigma$ such that $MX_{\langle T, \sigma \rangle}$ expresses $X$.

In general, the expressivity of the logic $\mathcal{L}$ determines the complexity class that is captured by $\mathcal{L}$-$MX$ and by $\mathcal{L}$-$MX_{\langle \cdot, \cdot \rangle}$.

Mitchell et al. (2006) presented a similar result for NP *search* problems, to the effect that for every NP search problem, there is an ID-logic theory $T$ and a vocabulary $\sigma \subseteq vocab(T)$ such that the expansion models of any finite $\sigma$-interpretation $I_\sigma$ have a one-to-one correspondence to solutions of the NP problem for input $I_\sigma$.

This means that parameterized $MX$ is ideally suited as a computational paradigm for solving search problems (with logic). In this paradigm, a problem is solved by computing expansion models of a specific theory in some logic $\mathcal{L}$. To this end, a human expert should model the theory such that its expansion models correspond to the solutions of the problem at hand.[2] This paradigm is very closely related to the *Answer Set Programming* paradigm, which we discuss in Chapter 7.

[2]Depending on the choice of $\mathcal{L}$, it may not always be possible to achieve a one-to-one

## 3.3 Examples

In this section, we illustrate the use of model expansion for ID-logic on a number of examples, most of which have been published before in (Mariën et al., 2006; Wittocx et al., 2008c). Also other works demonstrate that ID-logic is a valuable knowledge representation language; for instance, Denecker and Ternovska (2007) presents in ID-logic the most general version of *situation calculus* to date.

### 3.3.1 Block's world

We have partly discussed the *block's world* before. Here we give a full representation of the problem.

A table and a number of blocks are given; the blocks are initially stacked in some given configuration. Also a finite number of timesteps is given, as well as a goal configuration of the blocks. The task is to find a sequence of moves that lead from the initial configuration to the goal configuration. A move consists of a putting a block that is free (has no blocks on top of it) on the table, or on top of another free block. There can be only one move per timestep, and the sequence should be completed in the given number of timesteps.

In the representation below, we use a type $Block$ to represent the set of blocks and the table, and a type $Time$ to represent the timepoints. The symbols in the input vocabulary are $InitOn/2$ and $GoalOn/2$, representing the initial and goal configuration, and constants $Table$, $Init$ and $Final$, representing respectively the table, the initial and the final timepoint.

The meaning of the expansion predicates $On/3$, $Move/3$ and $Free/2$ is clear.

Observe that the table is treated as a special block which is always free and cannot be moved. We use the *exists unique* quantifier "∃!", which is defined by $\exists!\overline{x}\ \varphi[\overline{x}]$ if $\big(\exists\overline{x}\ \varphi[\overline{x}]\big) \wedge \big(\forall\overline{x}_1,\overline{x}_2\ \varphi[\overline{x}_1] \wedge \varphi[\overline{x}_2] \supset \overline{x}_1 = \overline{x}_2\big)$.

$$
\begin{aligned}
\textbf{types:}\ & Block, Time \\
\textbf{input vocabulary:}\ & InitOn(Block, Block), \\
& GoalOn(Block, Block), \\
& Table : Block,\ Init : Time,\ Final : Time \\
\textbf{expansion vocabulary:}\ & On(Block, Block, Time), \\
& Move(Block, Block, Time), \\
& Free(Block, Time)
\end{aligned}
$$

correspondence. In such cases, a many-to-one correspondence (many expansion models to one solution) may be the best achievable result. However, Mitchell et al. (2006) show that in the case of ID-logic a one-to-one correspondence can always be achieved.

$$\left\{\begin{array}{lll} \forall b_1, b_2 & \big(On(b_1, b_2, Init) \leftarrow & InitiallyOn(b_1, b_2)\big), \\ \forall b_1, b_2, t & \big(On(b_1, b_2, t) \leftarrow & Move(b_1, b_2, t-1)\big), \\ \forall b_1, b_2, t & \big(On(b_1, b_2, t) \leftarrow & On(b_1, b_2, t-1) \wedge \\ & & \neg(\exists b_3\ Move(b_1, b_3, t-1))\big) \end{array}\right\}, \tag{3.13}$$

$$\left\{\begin{array}{ll} \forall b, t & \big(Free(b, t) \leftarrow \ \neg(\exists b'\ On(b', b, t))\big), \\ \forall t & \big(Free(Table, t) \leftarrow \ \top\big) \end{array}\right\}, \tag{3.14}$$

$$\neg(\exists b, t\ Move(Table, b, t)), \tag{3.15}$$

$$\forall b_1, b_2, t\ Move(b_1, b_2, t) \supset Free(b_1, t) \wedge Free(b_2, t), \tag{3.16}$$

$$\forall t\ \exists! b_1, b_2\ Move(b_1, b_2, t), \tag{3.17}$$

$$\forall b_1, b_2\ GoalOn(b_1, b_2) \supset On(b_1, b_2, Final). \tag{3.18}$$

### 3.3.2 Hamiltonian circuit

A *Hamiltonian circuit* of a given graph is a closed path that visits every vertex exactly once.

In the representation below we use one type, $Vtx$. The input vocabulary consists of one predicate symbol, *Edge*/2, representing the edges of the graph. The expansion vocabulary consists of the constant symbol *Start*, representing an arbitrary vertex, and the predicate symbols $Ham/2$, representing the edges of the Hamiltonian circuit, and *Reached*/1, representing the vertices that are reached by the path $Ham/2$ when leaving from *Start*.

**types:** $Vtx$

**input vocabulary:** $Edge(Vtx, Vtx)$

**expansion vocabulary:** $Ham(Vtx, Vtx), Reached(Vtx), Start : Vtx$

$$\forall x, y\ Ham(x, y) \supset Edge(x, y), \tag{3.19}$$

$$\forall x\ \exists! y\ Ham(x, y), \tag{3.20}$$

$$\forall y\ \exists! x\ Ham(x, y), \tag{3.21}$$

$$\left\{\begin{array}{ll} \forall x\ (Reached(x) \leftarrow & Ham(Start, x)), \\ \forall x\ (Reached(x) \leftarrow & \exists y\ Ham(y, x) \wedge Reached(y)) \end{array}\right\}, \tag{3.22}$$

$$\forall x\ Reached(x). \tag{3.23}$$

$$\tag{3.24}$$

Note the difference between the inductive definition (3.22) of *Reached*/1, and the sentence

$$\forall x\ Reached(x) \equiv (\exists y\ Ham(y, x) \wedge Reached(y)). \tag{3.25}$$

In the latter, there is no requirement that a vertex $v$ is reachable by $Ham/2$ *from some fixed start vertex*, in order for $v$ to be in the *Reached*/1 relation. E.g., an interpretation $I$ with $\text{dom}(I) = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, $Start^I = \mathsf{a}$, $Ham^I = \{(\mathsf{b}, \mathsf{c}), (\mathsf{c}, \mathsf{b})\}$ and $Reached^I = \{b, c\}$ satisfies (3.25) but not definition (3.22).

### 3.3.3 $N$-queens

The $N$-queens problem is a well-known combinatorial problem whereby $N$ chess queens must be positioned on an $N \times N$ grid such that no two queens can capture each other (by moving over rows, columns or diagonals).

We use a type $D$ to represent the domain $\{1, \ldots, N\}$, and no input vocabulary. The expansion vocabulary consists of one predicate symbol, $Queen/2$, which represents the grids on which a queen is positioned. This representation uses arithmetic such as $abs(\cdot)$ end $-/2$, which is interpreted in the obvious way.

**types:** $D$
**input vocabulary:** —
**expansion vocabulary:** $Queen(D, D)$

$$\forall r\ \exists! c\ Queen(r, c), \tag{3.26}$$

$$\forall c\ \exists! r\ Queen(r, c), \tag{3.27}$$

$$\begin{aligned} \forall r_1, r_2, c_1, c_2\ abs(r_2 - r_1) = abs(c_2 - c_1) \supset \\ \neg(Queen(r_1, c_1) \wedge Queen(r_2, c_2)). \end{aligned} \tag{3.28}$$

Observe that both here and in the previous example, we could have used function symbols instead of the predicate symbols $Queen/2$ and $Ham/2$. However, it is not obvious from the natural language specification of the problem in either example that the solution is a function. Therefore we consider it a matter of choice whether or not to represent the concepts as a function. By contrast, the desired solution of the graph colouring problem (cf. Example 3.2) is clearly a function; there is no reason not to use a function symbol there.

### 3.3.4 Transitive reduction

The *transitive closure* $T$ of a binary relation $R$ is the least relation that contains $R$ and is transitively closed. It can be represented by the following definition in ID-logic:

$$\left\{ \begin{array}{l} \forall x, y\ \big(T(x, y) \leftarrow R(x, y)\big), \\ \forall x, y\ \big(T(x, y) \leftarrow \exists z\ T(x, z) \wedge T(z, y)\big) \end{array} \right\}. \tag{3.29}$$

Conversely, the *transitive reduction* of a binary relation $T$ is a minimal relation $R$ which has $T$ as its transitive closure. For arbitrary $T$, neither the existence nor the uniqueness of such a relation $R$ is guaranteed.

In the representation below we use the type $D$ to represent $T$'s (arbitrary) domain. The input vocabulary is $\{T/2\}$. The expansion vocabulary consists of two predicate symbols: $R/2$, the desired transitive reduction relation, and $TS/4$, representing the transitive closure of a relation *smaller* than $R$: for each $u$ and $v$, the binary relation $TS(\cdot, \cdot, u, v)$ denotes the transitive closure of $R \setminus \{(u, v)\}$.

Observe that this representation uses the definition of transitive closure as given above.

**types:** $D$
**input vocabulary:** $T(D, D)$
**expansion vocabulary:** $R(D, D), TS(D, D, D, D)$

$$\left\{ \begin{array}{l} \forall x, y \ \big(T(x, y) \leftarrow R(x, y)\big), \\ \forall x, y \ \big(T(x, y) \leftarrow \exists z \ T(x, z) \land T(z, y)\big) \end{array} \right\}, \tag{3.30}$$

$$\left\{ \begin{array}{l} \forall x, y, u, v \ \big(TS(x, y, u, v) \leftarrow R(x, y) \land \neg(x = u \land y = v)\big), \\ \forall x, y, u, v \ \big(TS(x, y, u, v) \leftarrow \exists z \ (TS(x, z, u, v) \land TS(z, y, u, v))\big) \end{array} \right\}, \tag{3.31}$$

$$\forall x, y \ (R(x, y) \supset \neg TS(x, y, x, y)). \tag{3.32}$$

Recall that the $MX$ computational paradigm requires the human expert to model the theory such that its expansion models correspond to the solutions of the problem at hand. Since is not trivial that expansion models of the above theory correspond to transitive reductions, we formally prove it. In the following, let $\mathcal{T}$ be the theory (3.31)–(3.32).

**Proposition 3.1.** *The model expansion problem* $MX_{\langle \mathcal{T}, \{T\} \rangle}$ *correctly models the transitive reduction problem of* $T$.

*Proof.* We denote by $tc(X)$ the transitive closure of a relation $X$.

Indeed, if $I$ is a solution to this MX problem, then we have the following:

- Since $I \models$ (3.30), we have $tc(R^I) = T^I$.
- We show that $R^I$ is subset-minimal. Assume towards contradiction that there exists a binary relation $R'$ such that $R' \subsetneq R^I$, and $tc(R') = T^I$.
  - Choose a tuple $(a, b) \in R^I$ and $\notin R'$
  - $(a, b) \in T^I$ because $T^I = tc(R^I) \supseteq R^I$.
  - Therefore, and using $tc(R') = T^I$, there exists a sequence $(a, c_1)$, $(c_1, c_2)$, $\ldots$, $(c_n, b)$ of tuples, all of which are in $R'$.
  - Since $I \models$ (3.32), $(a, b, a, b) \notin TS^I$. Hence, and using $I \models$ (3.31), there does not exist a sequence $(a, c'_1), (c'_1, c'_2), \ldots, (c'_n, b)$ in $R^I \setminus \{(a, b)\}$.
  - Because $R^I \setminus \{(a, b)\} \supseteq R'$, the sequence $(a, c_1)$, $\ldots$, $(c_n, b)$ consists of tuples in $R^I \setminus \{(a, b)\}$, hence, with $c'_1 = c_1$, $c'_2 = c_2$, $\ldots$, $c'_n = c_n$, we have found a sequence $(a, c'_1), (c'_1, c'_2), \ldots, (c'_n, b)$ in $R^I \setminus \{(a, b)\}$. Contradiction.

Conversely, let $R''$ be a transitive reduction of $T$. Then it is trivial to construct a structure $I$ that is a solution to the MX problem, with $R^I = R''$. □

## 3.4 The IDP system

The previous sections motivate the use of ID-logic as a knowledge representation language, and model expansion as a computational paradigm. Following these motivations, we have built an ID-logic-$MX$ solver called IDP (Mariën et al., 2006; Wittocx et al., 2008c; Wittocx and Mariën, 2008). The implementation of the IDP system is joint work with Johan Wittocx.

The input language of IDP is ID-logic with many extensions; a theory in the IDP language is a set of FO sentences and inductive definitions. The extensions include:

- aggregates, both in FO sentences and in rule bodies (cf. Chapter 6);
- quantifiers with numerical constraints;
- order-sorted types;
- arithmetic; and
- partial functions.

For a complete description of the IDP language and of the semantics of these extensions, we refer to (Wittocx et al., 2008c; Wittocx and Mariën, 2008). A complete IDP specification consists of a type and vocabulary declaration, a declaration of the input vocabulary $\sigma$, a declaration of the *output vocabulary*, and a theory in the IDP language. "Output vocabulary" is the subset of the expansion vocabulary that suffices to describe problem solutions. E.g., in the transitive reduction encoding of Section 3.3.4, the relation $R/2$ suffices to describe the solution; $TS/4$ is just used as an auxiliary symbol.

As an example, we encode the graph colouring problem from Example 3.2 in IDP. The ASCII symbol `!` means $\forall$, `~=` means $\neq$, and otherwise, the syntax is self-explanatory.

```
Given:
   type { Clr Vtx }
   Edge(Vtx,Vtx)
Find:
   Colour(Vtx) : Clr
Satisfying:
   ! v1 v2 : Edge(v1,v2) => Colour(v1) ~= Colour(v2).
```

The IDP system consists of two components: a grounder, GidL (Wittocx et al., 2008a), and a propositional model generator, MiniSat(ID) (Mariën et al., 2008).

GidL's task is to transform a given ID-logic-$MX$ problem with a given input interpretation to a PC(ID) theory, the models of which correspond to the expansion models. Its algorithms have been described in (Wittocx et al., 2008a,b). The output language of GidL is Extended CNF (ECNF, see Section 5.2.4), a

normal form for PC(ID) theories. It extends CNF with propositional definitions and propositional aggregate expressions.

There are two propositional solvers for the IDP system: MidL (Mariën et al., 2007a,b) and the more recent MiniSat(ID) (Mariën et al., 2008). The former supports CNF with propositional definitions, the latter supports full ECNF. We discuss the implementation of the latter at length in Chapters 5 and 6; suffice it to say here that it is an extension of the SAT solver MiniSat (Eén and Sörensson, 2003).

The performance of the IDP system as a whole has recently been tested by Wittocx et al. (2008c); it compared favourably to similar systems, scoring best on at least three aggregated measures over a wide range of problems.

## 3.5 Grounding

In the IDP system, the grounding phase is performed by GidL. The development of efficient grounding techniques, and the implementation of GidL, is the work of Johan Wittocx (Wittocx, Mariën, and Denecker, 2008a,b). To make this text self-contained, we present here a simple but very naive grounding technique. The techniques used in GidL produce significantly smaller PC(ID) theories than the one presented here.

First we remove all function symbols from the given theory. For any function symbol $F/n$, introduce a new predicate symbol $P_F/(n+1)$, and add the axiom $\forall \overline{x}\ \exists! y\ P_F(\overline{x}, y)$. Replace all occurrences of a function symbol $F$ in the form $A(\dots, F(\overline{t}), \dots)$ by $\exists t'\ A(\dots, t', \dots) \wedge P_F(\overline{t}, t')$, except in the head of rules of a definition. If $\forall \overline{x}\ \big(A(\dots, F(\overline{t}), \dots) \leftarrow \varphi\big)$ is a rule of a definition, then replace it by $\forall \overline{x}, t'\ \big(A(\dots, t', \dots) \leftarrow P_F(\overline{t}, t') \wedge \varphi\big)$.

We now assume that the given theory is function-free. We further assume that a finite domain $D$ is given.[3] For each domain atom $\mathsf{d} \in D$, we introduce a new constant symbol $d$. For a formula $\Psi[\overline{x}]$ and a tuple $\overline{\mathsf{d}} \in D^{|\overline{x}|}$, we represent by $\Psi[\overline{x}/\overline{\mathsf{d}}]$ the propositional formula obtained from $\Psi[\overline{x}]$ by replacing all occurrences of $\overline{x}$ by $\overline{d}$. Then the grounding of an FO formula $\varphi$ is the formula $Gr_D(\varphi)$ defined by:

- if $\varphi = \forall \overline{x}\ \Psi[\overline{x}]$, it is $\bigwedge_{\overline{\mathsf{d}} \in D^{|\overline{x}|}} Gr_D(\Psi[\overline{x}/\overline{\mathsf{d}}])$;
- if $\varphi = \exists \overline{x}\ \Psi[\overline{x}]$, it is $\bigvee_{\overline{\mathsf{d}} \in D^{|\overline{x}|}} Gr_D(\Psi[\overline{x}/\overline{\mathsf{d}}])$;
- if $\varphi = \varphi_1 \wedge \varphi_2$, it is $Gr_D(\varphi_1) \wedge Gr_D(\varphi_2)$;
- if $\varphi = \varphi_1 \vee \varphi_2$, it is $Gr_D(\varphi_1) \vee Gr_D(\varphi_2)$;
- if $\varphi = \neg\Psi$, it is $\neg Gr_D(\Psi)$.

[3] Note that in practice, a finite *input interpretation* is given, i.e., more than just a finite domain. GidL exploits the given input interpretation to reduce the size of the resulting PC(ID) theory.

The grounding of a rule $\forall \overline{x}\ (P(\overline{x}) \leftarrow \Psi[\overline{x}])$ in an inductive definition is the set of rules $P(\overline{d}) \leftarrow Gr_D(\Psi[\overline{x}/\overline{\mathsf{d}}])$, for each $\overline{\mathsf{d}} \in D^{|\overline{x}|}$.

Note that $Gr_D(\varphi)$ is variable-free, but may not have a propositional vocabulary. To convert $Gr_D(\varphi)$ into a PC(ID) theory, we have to create a new vocabulary, in which we convert every atom $P(\overline{d})$ corresponding to a domain atom $P(\overline{\mathsf{d}})$ into a separate propositional atom; this is a trivial operation.

**Example 3.4.** The grounding $Gr_{\{a,b\}}(\Delta_{3.4})$ of $\Delta_{3.4} =$

$$\left\{ \begin{array}{l} \forall x, y\ TC(x,y) \leftarrow R(x,y), \\ \forall x, y\ TC(x,y) \leftarrow \exists z\ TC(x,z) \wedge R(z,y) \end{array} \right\}$$

is the definition

$$\left\{ \begin{array}{l} TC(a,a) \leftarrow R(a,a), \quad TC(b,a) \leftarrow R(b,a), \\ TC(a,b) \leftarrow R(a,b), \quad TC(b,b) \leftarrow R(b,b), \\ TC(a,a) \leftarrow (TC(a,a) \wedge R(a,a)) \vee (TC(a,b) \wedge R(b,a)), \\ TC(a,b) \leftarrow (TC(a,a) \wedge R(a,b)) \vee (TC(a,b) \wedge R(b,b)), \\ TC(b,a) \leftarrow (TC(b,a) \wedge R(a,a)) \vee (TC(b,b) \wedge R(b,a)), \\ TC(b,b) \leftarrow (TC(b,a) \wedge R(a,b)) \vee (TC(b,b) \wedge R(b,b)) \end{array} \right\}.$$

If $T$ is an ID-logic theory, then for every model $I$ of $Gr_D(\bigwedge T)$ with $\mathrm{dom}(I) = D$ and $d^I = \mathsf{d}$ for every $\mathsf{d} \in D$, $I|_{vocab(T)} \models T$. Conversely, for every model $I$ of $T$ with $\mathrm{dom}(I) = D$, there is an extension $I'$ of $I$ with $I' \models Gr_D(\bigwedge T)$ and $d^{I'} = \mathsf{d}$. We can therefore solve the model generation problem of $T$ in domain $D$ by solving the *propositional* model generation problem of $Gr_D(\bigwedge T)$.

## 3.6 Satisfiability of propositional ID-logic

We give some simple complexity results about the SAT(ID) problem. In Chapter 5, we discuss algorithms for the SAT(ID) problem at length.

**Proposition 3.2.** *SAT(ID) is NP-complete.*

*Proof.* (Membership) Checking whether a propositional interpretation is a well-founded model can be done in polynomial time, e.g. in quadratic time using an algorithm by Van Gelder (1993). It is easy to define an algorithm that uses such a well-founded semantics algorithm and finds SAT(ID) models in polynomial time on a non-deterministic Turing machine.

(Hardness) Any SAT problem is trivially also a SAT(ID) problem. □

Recall Definition 2.9 of totality of a definition $\Delta$ with respect to a theory $T$: for each $I \models T$, the well-founded model of $\Delta$ extending $I|_{Open(\Delta)}$ must be two-valued. Though deciding totality is not directly relevant for the SAT(ID) problem, it is an interesting problem, not least because stable and well-founded models coincide for total definitions.

**Proposition 3.3.** *Deciding whether a given propositional inductive definition $\Delta$ is total with respect to a given propositional theory $\Phi$ is a* co-NP*-complete problem.*

*Proof.* (Membership) Any interpretation $I$ such that $I \models \Phi$ and such that the well-founded model of $\Delta$ extending $I|_{Open(\Delta)}$ is not two-valued, is a certificate for the *non*-totality $\Delta$ with respect to $\Phi$. Both checking whether $I \models \Phi$ and whether the well-founded model of $\Delta$ extending $I|_{Open(\Delta)}$ is two-valued can be done in polynomial time.

(Hardness) Consider the definition $\Delta = \left\{ \; p \leftarrow \neg p \wedge \Phi \; \right\}$. $\Delta$ is total with respect to the empty theory iff $\Phi$ is unsatisfiable. Thus we have found an instance of our decision problem that is equivalent to a co-NP-hard decision problem, namely UNSAT. □

**Proposition 3.4.** *Let $\Delta$ be a definition that is total with respect to some theory $T$. Then $\Delta$ is satisfiable.*

Since it is part of our ID-logic methodology to write definitions that are total, we expect most definitions to be satisfiable. This, of course, has no implications on the satisfiability of a PC(ID) theory that contains only total definitions, not even when the theory without the definitions is also satisfiable. The reason is that the definitions may share vocabulary with the rest of the theory (and exert extra constraints on it).

## 3.7 Conclusions

We have presented ID-logic from a knowledge representation point of view, and highlighted some methodological principles of modelling in ID-logic. We mention some in summary:

- represent definitional knowledge by definitions;
- represent definitions as separate modules;
- represent non-definitional knowledge by FO sentences;
- represent each "case" of an informal definition by one rule in the formal definition;
- use function symbols whenever the represented concept is clearly a function.

We have followed the suggestion of Mitchell and Ternovska (2005) to use the model expansion problem as a computational paradigm to solve NP problems. We then gave a number of examples of model expansion for ID-logic, and presented the IDP system, a system for ID-logic-$MX$. Finally, we have presented a simple grounding technique, and established some results about SAT(ID).

# Chapter 4

# Semantical analysis of propositional definitions

## 4.1 Introduction

In this chapter we perform a semantical analysis of PC(ID), and more specifically, of propositional inductive definitions (IDs). The central question is: when is a given interpretation a model of a propositional definition?

We give two independent characterizations of (models of) IDs. The first one (Theorem 4.1) is a graph theoretical characterization of IDs; it uses graph structures called *justifications*. The second one (Theorem 4.2) is a PC characterization of IDs; it involves propositional formulas called *loop formulas*.

Such a semantical analysis is important in its own right, because it yields new insights into the logic. For instance, our second result defines a transformation of PC(ID) theories to PC theories. It also offers us a new way of characterizing the correspondences and the differences between PC(ID) and propositional Stable logic programs.

Importantly, these results also form the theoretical underpinnings to develop algorithms for solving the SAT(ID) problem. This is the topic of Chapter 5. In that chapter, we use both characterizations that are developed here. For instance, we develop an algorithm that seeks to produce a loop formula, by searching for possible justifications. This chapter forms the link between such algorithms and the original semantics of inductive definitions.

Section 4.2 introduces a normal form for IDs, which we use throughout the text, and presents some graph concepts. In Section 4.3, justification semantics is introduced, and some derived concepts useful for SAT(ID) algorithms are defined. Section 4.4 defines loop formulas for inductive definitions, and discusses some observations derived from the general result.

## 4.2 Preliminaries

### 4.2.1 Completion

Clark (1978) defined the *completion* of a logic program. Here we similarly define the completion of a propositional inductive definition (ID) $\Delta$, denoted $comp(\Delta)$: it is the propositional formula

$$\bigwedge_{p \in Def(\Delta)} \Big( p \equiv \bigvee \{ \varphi \mid (p \leftarrow \varphi) \in \Delta \} \Big). \tag{4.1}$$

For a PC(ID) theory $T$, denote by $comp(T)$ the theory obtained from $T$ by replacing each definition $\Delta \in T$ by $comp(\Delta)$.

The following result is well-known:

**Proposition 4.1.** *Let $\Delta$ be a propositional ID. Then $\Delta \models comp(\Delta)$.*

Note that for any given $Open(\Delta)$-interpretation $I_O$, $\Delta$ has a unique model extending $I_O$ (or possibly no model at all, if it is non-total). $comp(\Delta)$ on the other hand may have multiple models extending $I_O$.

**Example 4.1.** Let $\Delta_{4.1} = \left\{ \; p \leftarrow p \lor a \; \right\}$. Then $comp(\Delta_{4.1}) = (p \equiv p \lor a)$. $\Delta_{4.1}$ has two models: $\{p \mapsto \boldsymbol{t}, a \mapsto \boldsymbol{t}\}$, $\{p \mapsto \boldsymbol{f}, a \mapsto \boldsymbol{f}\}$; $comp(\Delta_{4.1})$ has the same two models, and the additional model $\{p \mapsto \boldsymbol{t}, a \mapsto \boldsymbol{f}\}$.

### 4.2.2 Definitional Normal Form

Recall that for a set of literals $S$, we denote by $\overline{S}$ the set $\{\neg s \mid s \in S\}$, and by $\widehat{S}$ the set $S \cup \overline{S}$.

A propositional definition $\Delta$ is in *definitional normal form (DefNF)* if each $p \in Def(\Delta)$ is defined by exactly one rule, denoted $p \leftarrow \varphi_p$, the body of which is either a disjunction or a conjunction of literals. We call $p \leftarrow \varphi_p$ *the rule defining* $p$. Slightly abusing notation, in some contexts $\varphi_p$ will be used to denote *the set of literals* in the body of the rule defining $p$. E.g., $l \in \varphi_p$ means that $l$ is one of the conjuncts or disjuncts of $\varphi_p$. We extend this notation to negative defined literals: for $l \in \overline{Def(\Delta)}$, we denote by $\varphi_l$ the formula $\neg\varphi_{\neg l}$, which is a conjunction if $\varphi_{\neg l}$ is a disjunction, and vice versa. Then the literals $l' \in \varphi_l$ are the negations of the conjuncts or disjuncts of $\varphi_{\neg l}$.

Observe that for DefNF definitions $\Delta$ the completion simplifies to

$$comp(\Delta) = \bigwedge_{p \in Def(\Delta)} p \equiv \varphi_p, \tag{4.2}$$

i.e., a simple replacement of $\leftarrow$ by $\equiv$. It then follows from Proposition 4.1 that for any $p \in Def(\Delta)$, $\Delta \models (p \equiv \varphi_p)$.

We partition the set of defined literals $\widehat{Def(\Delta)}$ in two:

$$\mathcal{D}_{\text{lits}} = \{p \mid \varphi_p \text{ is a disjunction}\} \cup \{\neg p \mid \varphi_p \text{ is a conjunction}\},$$
$$\mathcal{C}_{\text{lits}} = \{p \mid \varphi_p \text{ is a conjunction}\} \cup \{\neg p \mid \varphi_p \text{ is a disjunction}\}.$$

For atoms $p \in Def(\Delta)$ for which $\varphi_p$ is a singleton, we treat $\varphi_p$ as a conjunction. Also, we interpret $\top$ and $\bot$ here as literals of $Open(\Delta)$, with fixed truth values $\boldsymbol{t}$ respectively $\boldsymbol{f}$.

We call literals in $\mathcal{D}_{\text{lits}}$ *disjunctively defined*, and literals in $\mathcal{C}_{\text{lits}}$ *conjunctively defined*. Observe that from $\Delta \models (p \equiv \varphi_p)$ for defined atoms $p$ we can derive $\Delta \models (\neg p \equiv \neg\varphi_p)$; for a negative literal $l = \neg p$ that is disjunctively defined, $\varphi_p$ is a conjunction, so $\neg\varphi_p$ is a disjunction; for a conjunctively defined negative literal $l = \neg p$, $\neg\varphi_p$ is a conjunction. Hence, all disjunctively defined literals are equivalent to the disjunction of their body literals, and all conjunctively defined literals are equivalent to the conjunction of their body literals.

**Example 4.2.** Let $\Delta_{4.2}$ =

$$\left\{ \begin{array}{l} p \leftarrow q \vee r, \\ q \leftarrow p \wedge \neg s \end{array} \right\}.$$

Then we have $\mathcal{D}_{\text{lits}} = \{p, \neg q\}$, $\mathcal{C}_{\text{lits}} = \{\neg p, q\}$, and $comp(\Delta_{4.2}) = (p \equiv q \vee r) \wedge (q \equiv p \wedge \neg s)$. For the disjunctively defined literals $p$ and $\neg q$, we have $\varphi_p = \bigvee\{q, r\}$ and $\varphi_{\neg q} = \bigvee\{\neg p, s\}$, and consequently it holds that $\Delta_{4.2} \models p \equiv \bigvee\{q, r\}$ and $\Delta_{4.2} \models \neg q \equiv \bigvee\{\neg p, s\}$.

An arbitrary propositional definition can be transformed in linear time into an equivalent (up to the original vocabulary) DefNF definition using *predicate introduction*. The transformation starts by merging all rules with the same head atom into one rule as follows: $p \leftarrow \varphi_1, \dots, p \leftarrow \varphi_n$ becomes $p \leftarrow \varphi_1 \vee \cdots \vee \varphi_n$, for all defined atoms $p$. It then simply applies the following two steps until DefNF form is reached:

- replace a positively occurring subformula $\psi$ of some rule body by a newly introduced atom, say $p_\psi$;
- introduce a new rule $p_\psi \leftarrow \psi$.

Its correctness is proven by Wittocx et al. (2006).

As an example, we show how a definition in normal logic programming form is transformed to DefNF. Let the definition consist of rules $p_i \leftarrow \psi_{i,j}$, for $1 \leq i \leq n$ and for $1 \leq j \leq n_i$, where the $\psi_{i,j}$ are conjunctions of literals. The DefNF definition obtained after our transformation consists of disjunctive rules of the form $p_i \leftarrow \bigvee_{1 \leq j \leq n_i} p_{i,j}$, for $1 \leq i \leq n$, and of conjunctive rules of the form $p_{i,j} \leftarrow \psi_{i,j}$, for $1 \leq i \leq n$ and for $1 \leq j \leq n_i$. The $p_{i,j}$ are new atoms.

Besides the obvious advantage of simplified theoretical results, the use of DefNF has also practical advantages.

- It is very similar to the widely used conjunctive normal form (CNF); in fact, transforming $comp(\Delta)$ to CNF is trivial. This means that in practical implementations of SAT(ID) solvers, the data structures that are used by SAT solvers to represent CNF clauses can be reused to represent DefNF rules.

- Anger et al. (2006a) studied the behaviour of truth propagation algorithms with and without the possibility to assign a truth value to a rule body

in normal logic programming. They found that the former potentially behaved exponentially better than the latter. Under our transformation to DefNF above, rule bodies are represented by atoms, and therefore, normal truth propagation algorithms for DefNF automatically behave like the faster truth propagation algorithms for the corresponding normal logic programming definition. Their work was done in the context of stable model semantics, but carries over to ID semantics.

In the following sections, we study the satisfaction relation $I \models \Delta$ for DefNF definitions $\Delta$ with vocabulary $\Sigma$. The results generalize, with appropriate adaptions of definitions, to arbitrary propositional definitions.

### 4.2.3 Some graph concepts

A *(directed) graph* is a tuple $(V, E)$ of *vertices* $V$ and *edges* $E$. The edges form a binary relation over vertices. A *path* from $v_1 \in V$ to $v_2 \in V$ is a non-empty sequence of edges $(v_1, v'), (v', v''), \ldots, (v^{(n)}, v_2)$, each of which is in $E$. For a graph $G = (V, E)$, we denote by $v \in G$ that $v \in V$ and by $(v_1, v_2) \in G$ that $(v_1, v_2) \in E$. For graphs $G = (V, E)$ and $G' = (V', E')$, $G'$ is a *subgraph* of $G$ if $V' \subseteq V$ and $E' \subseteq E$.

Given a graph $G = (V, E)$, a set of vertices $S \subseteq V$ is said to be *strongly connected* if for any $v_1, v_2 \in S$, the graph contains a path through $S$ from $v_1$ to $v_2$. A *loop* of a graph is a non-empty, strongly connected set of vertices. For instance, if $(v, v) \in G$, then the singleton $\{v\}$ is a loop.

A *strongly connected component* of $G = (V, E)$ is a maximal subgraph $G' = (V', E')$ of $G$ such that $V'$ is strongly connected. By $\mathcal{SCC}_G$ we denote the set $\{V' \mid G' = (V', E')$ is a strongly connected component of $G\}$. Note that in general some vertices may not occur in any loop of $G$, and will therefore not be represented in $\mathcal{SCC}_G$. We define *the SCC-partition* of $G$, denoted $\mathcal{SCC}^+_G$, as the set $\mathcal{SCC}_G \cup \big\{\{v\} \mid v \in G$ and $v$ does not occur in a loop of $G\big\}$. Observe that $\mathcal{SCC}^+_G$ indeed forms a partition of $V$.

We define a relation $\preceq_G$ on the SCC-partition of $G$: for $S_1, S_2 \in \mathcal{SCC}^+_G$, $S_1 \preceq_G S_2$ if for some $v_1 \in S_1$, $v_2 \in S_2$, there is a path in $G$ from $v_2$ to $v_1$, or if $S_1 = S_2 = \{v\}$, where $v$ does not occur in a loop of $G$. Observe that for $S_1, S_2 \in \mathcal{SCC}_G$, $S_1 \preceq_G S_2$ implies that for *any* $v_1 \in S_1$, $v_2 \in S_2$, there is a path in $G$ from $v_2$ to $v_1$. Also, $S_1 \preceq_G S_2$ and $S_2 \preceq_G S_1$ implies that $S_1 = S_2$. The order $\preceq_G$ is well-founded if $G$ is finite. Hence $\preceq_G$ is a partial order. Note that $\preceq_G$ is now also defined on the strongly connected components of $G$, since $\mathcal{SCC}_G \subseteq \mathcal{SCC}^+_G$. We will often refer to a *minimal* strongly connected component of $G$: this is a component $G' = (V', E')$ such that there is no $V'' \in \mathcal{SCC}_G$ with $V'' \neq V'$ and $V'' \preceq_G V'$. Note that in general there may be several such minimal components.

We will sometimes encounter finite graphs $G$ with the property that for every $v \in G$, there is a $v' \in G$ such that $(v, v') \in G$. It follows that such a graph contains at least one loop, and therefore also at least one minimal strongly connected component. Furthermore, such a minimal strongly connected component

is also minimal in $\mathcal{SCC}^+_G$, i.e., it has no outgoing edges.

When considering subgraphs of a given graph, the following simple property of their strongly connected components is often useful:

**Proposition 4.2.** *Let $G$ be a graph, and $G'$ a subgraph of $G$. Then for any $S' \in \mathcal{SCC}^+_{G'}$, there exists an $S \in \mathcal{SCC}^+_G$ such that $S' \subseteq S$.*

## 4.3 A graph theoretical characterization of definitions

This section introduces *justification semantics*. The results in this section are based on the work of (Mariën et al., 2005, 2007b, 2008), and build on concepts first defined by Denecker and De Schreye (1993).

### 4.3.1 Justifications

The truth values of defined symbols are determined by the truth values of open symbols. We want to identify, for each defined literal, what would constitute sufficient reason for it to be true.[1] Consider an atom $c$ defined by the rule $c \leftarrow c_1 \wedge \ldots \wedge c_N$: in order for $c$ to be true, *all* of $\{c_1, \ldots, c_N\}$ should be true. On the other hand, in order for an atom $d$ defined by the rule $d \leftarrow d_1 \vee \ldots \vee d_N$ to be true, it suffices that *one* $d_i$ be true. We call such direct (local) dependencies $c \rightsquigarrow \{c_1, \ldots, c_N\}$, $d \rightsquigarrow d_i$ *direct justifications*.

**Definition 4.1** (Direct justification)**.** Let $l \in \widehat{Def(\Delta)}$. A *direct justification for $l$*, denoted $DJ(l)$, is a set of literals such that:

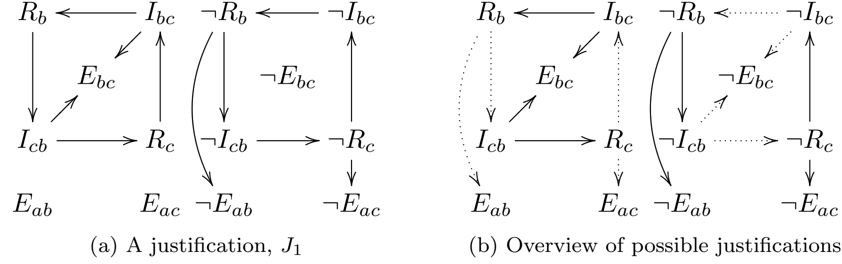
- if $l \in \mathcal{D}_{\text{lits}}$, $DJ(l)$ is a singleton $\{l'\}$, for $l' \in \varphi_l$;
- if $l \in \mathcal{C}_{\text{lits}}$, $DJ(l) = \varphi_l$.

However, such direct justifications on their own do not suffice yet to capture the deterministic relationship between open and defined symbols imposed by a definition. To capture this relationship we may use a graph structure, built out of direct dependencies as above. This motivates the following definition:

**Definition 4.2** (Justification)**.** A *justification $J$ for $\Delta$* is a directed graph $(V, E)$ where $V = \widehat{\Sigma}$, and $E$ is such that for each $l \in \widehat{Def(\Delta)}$, the set $\{l' \mid (l, l') \in E\}$ is a direct justification for $l$, denoted $DJ_J(l)$, and for each $l \in \widehat{Open(\Delta)}$, $E$ contains no edges leaving $l$.

Observe that a justification $J$ is uniquely determined by the function $d_J : \mathcal{D}_{\text{lits}} \rightarrow \widehat{\Sigma}$ defined by $d_J(l) = x$ if $DJ_J(l) = \{x\}$, for each $l \in \mathcal{D}_{\text{lits}}$.

[1]Note that identifying reasons for *literals* to be true is the same as identifying reasons for *atoms* to be either true or false.




(a) A justification, $J_1$ (b) Overview of possible justifications

Figure 4.1: Justifications for $\Delta_{4.3}$.

**Example 4.3.** Consider irreflexive undirected graphs with nodes $\{a, b, c\}$. We represent the edge between nodes $x$ and $y$ by $E_{xy}$. Then $R_x$ (for $x \in \{b, c\}$) in the following definition expresses the reachability of $x$ from $a$ ($I_{xy}$ expresses the reachability of $y$ from $a$ via $x$):

$$\Delta_{4.3} = \left\{ \begin{array}{ll} R_b \leftarrow E_{ab} \vee I_{cb}, & I_{cb} \leftarrow R_c \wedge E_{bc}, \\ R_c \leftarrow E_{ac} \vee I_{bc}, & I_{bc} \leftarrow R_b \wedge E_{bc} \end{array} \right\}.$$

Then Figure 4.1a shows a justification, $J_1$, for $\Delta_{4.3}$. $J_1$ is characterized by $d_{J_1}(R_b) = I_{cb}$, $d_{J_1}(R_c) = I_{bc}$, $d_{J_1}(\neg I_{bc}) = \neg R_b$, and $d_{J_1}(\neg I_{cb}) = \neg R_c$.

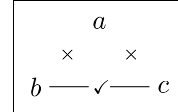
Figure 4.1b shows an overview of all possible justifications for $\Delta_{4.3}$, whereby possible edges are shown as dotted arrows. For each literal from which possible edges leave, exactly one of the edges must be chosen in an actual justification.

The intuitions given for the concept of justification also motivate the definition of the *support* property.

**Definition 4.3** (Support)**.** Let $J$ be a justification for $\Delta$, $I$ a $\Sigma$-interpretation. Then $J$ *supports* $I$ if for each $l \in \widehat{Def(\Delta)}$, $I(l) = I(\bigwedge DJ_J(l))$.

The conjunction $\bigwedge DJ_J(l)$ is true in $I$ if all literals in the direct justification of $l$ in $J$ are true. If that conjunction is true, then $l$ itself must be true.

**Example 4.4.** Example 4.3 continued. Consider the interpretation $I_1 = \{R_b \mapsto \boldsymbol{t}, R_c \mapsto \boldsymbol{t}, I_{bc} \mapsto \boldsymbol{t}, I_{cb} \mapsto \boldsymbol{t}, E_{bc} \mapsto \boldsymbol{t}, E_{ab} \mapsto \boldsymbol{f}, E_{ac} \mapsto \boldsymbol{f}\}$. This interpretation represents the graph shown on the right, and the statements that $b$ and $c$ are reachable from $a$, and that $b$ is reachable via $c$, and $c$ via $b$.




The justification $J_1$ shown in Figure 4.1a supports $I_1$.

Let $J$ be a justification and $L$ a loop in $J$. $L$ is a set of literals. We call $L$ *positive*, *negative*, or *mixed*, depending on whether it consists respectively of atoms only, of negative literals only, or of both. A *non-negative* loop is either positive or mixed, i.e., it is a loop that contains at least one atom.

Still building on the intuitions behind the concept of justification, we point out that our explanation contains a causal component: a defined literal $l$ is

Figure 4.2: Justifications for $\Delta_{4.7}$.

true *because* all literals in $\{l' \mid (l, l') \in J\}$ are true, for a given justification $J$. This implies that loops in a justification graph are unwanted: they reflect a cyclic, and therefore flawed, chain of reasons. However, it can be seen from the definition of the well-founded semantics (Definition 2.2) that some sets of atoms—unfounded sets—can "legitimately" cause each other to be false. In other words, negative loops are no problem. This motivates the definition of the *loop-safeness* property.

**Definition 4.4** (Loop-safeness)**.** Let $J$ be a justification for $\Delta$, $I$ a $\Sigma$-interpretation. Then *$J$ is loop-safe in $I$* iff for any non-negative loop in $J$, all literals in the loop are false in $I$.

**Example 4.5.** Examples 4.3–4.4 continued. $J_1$ is not loop-safe in $I_1$, since $\{R_b, I_{cb}, R_c, I_{bc}\}$ is a loop in $J_1$ that contains an atom, yet also contains true literals. Note that the loop $\{\neg R_b, \neg I_{cb}, \neg R_c, \neg I_{bc}\}$ is harmless: it contains only negative literals.

Naturally, we are interested in the combination of support and loop-safeness.

**Definition 4.5** (Witness)**.** Let $J$ be a justification for $\Delta$, $I$ a $\Sigma$-interpretation. Then *$J$ is a $\Delta$-witness for $I$* iff $J$ supports $I$ and $J$ is loop-safe in $I$. If $\Delta$ is clear from the context, we simply say $J$ is a witness.

**Example 4.6.** Examples 4.3–4.5 continued. There cannot be a $\Delta_{4.3}$-witness $J$ for $I_1$: if the support property is not to be violated, then $d_J(R_b) = I_{cb}$ and $d_J(R_c) = I_{bc}$, but then $\{R_b, I_{cb}, R_c, I_{bc}\}$ is a loop; if the loop-safeness property is not to be violated, all literals in it must be false, which they are not in $I_1$.

$J_1$ in Figure 4.1a is a witness for $I_2 = \{R_b \mapsto \boldsymbol{f},\ R_c \mapsto \boldsymbol{f},\ I_{bc} \mapsto \boldsymbol{f},\ I_{cb} \mapsto \boldsymbol{f},\ E_{bc} \mapsto \boldsymbol{f},\ E_{ab} \mapsto \boldsymbol{f},\ E_{ac} \mapsto \boldsymbol{f}\}$.

**Example 4.7.** Consider the definition $\Delta_{4.7} = \left\{\ p \leftarrow a \vee \neg p\ \right\}$. There are two justifications for $\Delta_{4.7}$, shown in Figure 4.2. $J_1$ cannot be loop-safe in any interpretation, since the loop $\{p, \neg p\}$ contains an atom and necessarily contains a true literal. Consider an interpretation $I$ that has $J_2$ as its witness. Then $J_2$ supports $I$, hence $I(p) = I(a)$, and $I(\neg p) = I(p \wedge \neg a)$. This is only possible for the interpretation $I = \{p \mapsto \boldsymbol{t}, a \mapsto \boldsymbol{t}\}$.

With the concept of witness in place, we come to our main result. This result formalizes our intuitions: if there exists a justification that satisfies all criteria

it should—support and loop-safeness—to represent a sound set of reasons for defined literals to be true in a given interpretation, i.e., if there exists a witness for the interpretation, it must be a model of the definition, and vice versa.

**Theorem 4.1.** *Let $I$ be a $\Sigma$-interpretation. Then $I \models \Delta$ iff there exists a $\Delta$-witness for $I$.*

**Example 4.8.** Example 4.7 continued. $I = \{p \mapsto \boldsymbol{t}, a \mapsto \boldsymbol{t}\}$ is the only model of $\Delta_{4.7}$. Note that $\Delta_{4.7}$ is not total, because its well-founded model extending $\{a \mapsto \boldsymbol{f}\}$ is three-valued: $\{p \mapsto \boldsymbol{u}, a \mapsto \boldsymbol{f}\}$.

We prove this result in Section 4.3.2, where we first introduce some preliminary results about justifications.

### 4.3.2 Properties of justifications

We start by some results regarding the *support* property.

**Proposition 4.3.** *Let $I$ be a $\Sigma$-interpretation. Then $I \models comp(\Delta)$ iff there exists a justification $J$ for $\Delta$ that supports $I$.*

*Proof.* We do a case analysis on the two types of rules of $\Delta$.

**$(p \leftarrow p_1 \vee \cdots \vee p_N) \in \Delta$.** If $I \models comp(\Delta)$, we have $I \models p \equiv p_1 \vee \cdots \vee p_N$. Suppose $I(p) = \boldsymbol{t}$. Then at least one $p_i$ is true in $I$. Then let $d_J(p) = p_i$: the support condition is thereby satisfied in $p$. Also, $\neg p$ is then false: the condition is then also satisfied in $\neg p$. Suppose instead $I(p) = \boldsymbol{f}$. Then all $p_i$ are also false in $I$. Hence, whichever $p_i$ is chosen as $d_J(p)$, the support condition is satisfied in $p$. Since all $\neg p_i$ are true in $I$, also $I(\neg p) = I(\bigwedge_i \neg p_i)$ is satisfied.

The reverse also is easy to verify (if the support condition is satisfied in $p$, then $I \models p \equiv p_1 \vee \cdots \vee p_N$).

**$(p \leftarrow p_1 \wedge \cdots \wedge p_N) \in \Delta$.** This is symmetric to the previous case.

□

The following concept, derived from justifications, will be useful throughout the text.

**Definition 4.6** (Subjustification)**.** Let $J$ be a justification for $\Delta$ and $l \in \widehat{Def(\Delta)}$. The *subjustification* of $J$ starting in $l$, denoted $Sub(J, l)$, is the subgraph of $J$ consisting of $l$ and all literals reachable from $l$ in $J$.

**Proposition 4.4.** *Let $J$ be a justification for $\Delta$, $I$ a $\Sigma$-interpretation, and let $J$ support $I$.*

1. *Let $l \in \widehat{Def(\Delta)}$ be a literal with $I(l) = \boldsymbol{t}$. Then all literals in $Sub(J, l)$ are true in $I$.*

2. *Let $L$ be a loop in $J$. Then $L$ consists entirely of false, or entirely of true literals in $I$.*

A special case of item 2 is when $L \in \mathcal{SCC}_J$.

*Proof.* We prove item 1 by induction: $l$ is true (in $I$), and if all literals reachable from $l$ in $J$ in $i$ steps are true, then all literals reachable from $l$ in $J$ in $i+1$ steps must be true. Indeed: let $l'$ be a literal reachable from $l$ in $J$ in $i$ steps, and $l'$ is true. Since $J$ supports $I$, it follows that all literals in $DJ_J(l')$ are true in $I$. Since all literals reachable from $l$ in $J$ in $i+1$ steps are member of $DJ_J(l')$ for some $l'$ reachable from $l$ in $J$ in $i$ steps, this proves the induction step.

It follows from the definition of $Sub(J, l)$ that a loop $L$ is a subset of $Sub(J, l)$ for any $l \in L$. Hence item 2 is an easy consequence of item 1. □

We further make a few observations that may help gain insight in the structure of a witness. The proofs of these statements follow trivially from the definition of loop-safeness and from the above results. Let $J$ be a witness for $I$.

- Let $S \in \mathcal{SCC}_J$. If $S$ contains an atom, it consists of false literals in $I$.
- Let $S \in \mathcal{SCC}_J$. If $S$ is mixed, then so is $\overline{S}$. No atom $p \in \overline{S}$ occurs in a loop in $J$, i.e., $\neg\exists S' \in \mathcal{SCC}_J : p \in S'$. Hence $\overline{S}$ as a whole is not a loop in $J$.
- Let the graph $G$ be the restriction of $J$ to true literals in $I$. Then $\mathcal{SCC}_G$ contains only negative loops.

We are now ready to prove Theorem 4.1, which states that $I \models \Delta$ iff there exists a $\Delta$-witness for $I$. The proof follows from Lemmas 4.1 and 4.2, which correspond to the "only if" and "if" cases, respectively.

**Lemma 4.1.** *Let $I$ be a $\Sigma$-interpretation, and let $I \models \Delta$. Then there exists a $\Delta$-witness for $I$.*

We recall Definition 2.1 of a *well-founded sequence*, and instantiate it here for propositional definitions in DefNF. A well-founded sequence for a DefNF definition $\Delta$ from a three-valued interpretation $I_o$ is any sequence $\langle I_i \rangle_{0 \leq i \leq n}$ of three-valued interpretations, such that $I_0 = I_o$, and such that for $0 \leq i < n$, $I_{i+1}$ is derived from $I_i$ by changing $I_i$ according to one of the following rules:

- $I_{i+1}(p) = \boldsymbol{t}$ for an atom $p \in \mathcal{D}_{\text{lits}}$ with $I_i(p) = \boldsymbol{u}$ and $I_i(b) = \boldsymbol{t}$ for at least one $b \in \varphi_p$;
- $I_{i+1}(p) = \boldsymbol{t}$ for an atom $p \in \mathcal{C}_{\text{lits}}$ with $I_i(p) = \boldsymbol{u}$ $I_i(b) = \boldsymbol{t}$ for each $b \in \varphi_p$;
- $I_{i+1}(p) = \boldsymbol{f}$ for all $p \in U$, for some set of atoms $U$ for which, for each $p \in U$, $I_i(p) = \boldsymbol{u}$ and
  - if $p \in \mathcal{D}_{\text{lits}}$, then $I_{i+1}(b) = \boldsymbol{f}$ for each $b \in \varphi_p$;
  - if $p \in \mathcal{C}_{\text{lits}}$, then $I_{i+1}(b) = \boldsymbol{f}$ for at least one $b \in \varphi_p$.

$U$ is called an unfounded set with respect to $I_i$.

Let $I_o$ be a three-valued interpretation that is two-valued on $Open(\Delta)$ and interprets all atoms in $Def(\Delta)$ by $\boldsymbol{u}$. Then if $\langle I_i \rangle_{0 \leq i \leq n}$ is a well-founded sequence from $I_o$ and $I_n$ is two-valued, then $I_n \models \Delta$.

*Proof.* (of Lemma 4.1)
Let $I$ be a model of $\Delta$. Hence there is a well-founded sequence $\langle I_i \rangle_{0 \leq i \leq n}$ as above, with $I_0$ the empty extension of $I|_{Open(\Delta)}$ to $\Sigma$ and with $I_n = I$.

We introduce a (stage level) function $f : \widehat{\Sigma} \to \{0, \ldots, n\}$, and define it as follows: for each $l \in \widehat{\Sigma}$, $f(l) = i$ iff $I_i(l) \neq \boldsymbol{u}$, and either $i = 0$ or $I_{i-1}(l) = \boldsymbol{u}$. Since $I_n = I$, $I_n$ is two-valued and $f$ is total. Note that $f(l) = f(\neg l)$ for any $l$.

We now use $f$ to define a justification $J$ (by defining the function $d_J$) and to prove that it is a witness for $I$.

- For each atom $l \in \mathcal{D}_{\text{lits}}$ with $I(l) = \boldsymbol{t}$, let $d_J(l) = b$ for a literal $b \in \varphi_l$ with $I_{f(l)-1}(b) = \boldsymbol{t}$. Note that $f(b) < f(l)$ and $I(b) = I(l)$ follows.

- For each negative literal $l \in \mathcal{D}_{\text{lits}}$ with $I(l) = \boldsymbol{f}$, let $d_J(l) = l'$ for some arbitrary $l' \in \varphi_l$. We have that $I_{f(l)-1}(b) = \boldsymbol{t}$ for any $b \in \varphi_{\neg l}$, and therefore again $f(l') < f(l)$ and $I(l') = I(l)$.

- For each atom $l \in \mathcal{D}_{\text{lits}}$ with $I(l) = \boldsymbol{f}$, let $d_J(l) = l'$ for some arbitrary $l' \in \varphi_l$. We have $I_{f(l)}(b) = \boldsymbol{f}$ for each $b \in \varphi_l$, hence $f(l') \leq f(l)$ and $I(l') = I(l)$.

- For each negative literal $l \in \mathcal{D}_{\text{lits}}$ with $I(l) = \boldsymbol{t}$, let $d_J(l) = b$ for a literal $b \in \varphi_l$ with $I_{f(p)}(b) = \boldsymbol{t}$. This exists because $\neg l$ is an atom $\in \mathcal{C}_{\text{lits}}$ that became false in $I_{f(\neg l)}$. We again have $f(b) \leq f(l)$ and $I(b) = I(l)$.

$J$ supports $I$: for literals in $\mathcal{D}_{\text{lits}}$ it follows from the definition of $d_J$; for literals in $\mathcal{C}_{\text{lits}}$ it follows from the construction of $I$. We show that $J$ is also loop-safe in $I$.

Observe that for each $(l, l') \in J$ such that $I(l) = \boldsymbol{t}$ (and therefore also $I(l') = \boldsymbol{t}$), if $l$ is an atom, then $f(l) > f(l')$; if $l$ is a negative literal, then $f(l) \geq f(l')$. Let $G$ be the subgraph $G$ of $J$, defined as the restriction of $J$ to true literals in $I$. Hence any loop in $G$ consists of negative literals. By Proposition 4.4, item 2 and because $J$ supports $I$, it follows that any loop in $J$ is either also a loop in $G$, and therefore a negative loop, or it consists entirely of false literals in $I$. Therefore $J$ is also loop-safe in $I$. □

We first illustrate the "if" case of Theorem 4.1 on an example.

**Example 4.9.** Examples 4.3–4.6 continued. Recall the justification $J_1$ from Figure 4.1a, which is a $\Delta_{4.3}$-witness for the interpretation $I_2$ that maps every atom to false.

We use $J_1$ to construct a sequence of three-valued interpretations with limit $I_2$, and show that the sequence is a well-founded sequence.

- Initially, $I_{2,0} = \{E_{bc} \mapsto \boldsymbol{f}, E_{ab} \mapsto \boldsymbol{f}, E_{ac} \mapsto \boldsymbol{f}\} \cup \{p \mapsto \boldsymbol{u} \mid p \in Def(\Delta_{4.3})\}$. Currently the literals $U = \{R_b,\ \neg R_b,\ R_c,\ \neg R_c,\ I_{bc},\ \neg I_{bc},\ I_{cb},\ \neg I_{cb}\}$ are unassigned, and for none of them, all its children in $J_1$ are assigned.
- Therefore we know that the restriction of $J_1$ to $U$ must contain (a) loop(s). In fact, at least one such loop is positive (we elaborate on the reasons in the proof). Indeed: the loops are $L = \{R_b, R_c, I_{bc}, I_{cb}\}$ and $\overline{L}$; $L$ is a positive loop.
- Since $J_1$ is loop-safe in $I_2$, all atoms in $L$ are false in $I_2$.
- We show that $L$ is an unfounded set with respect to $I_{2,0}$. (In the proof it is shown that there must exist such a loop $L$.) Each literal in $L \cap \mathcal{C}_{\text{lits}}$ (i.e. $I_{bc}$ and $I_{cb}$) has a child in $J_1$ that is $\in L$, and therefore false in $I_{2,0}[L/\boldsymbol{f}]$. For each literal $l$ in $L \cap \mathcal{D}_{\text{lits}}$ (i.e. $R_b$ and $R_c$), all literals in $\varphi_l$ are either $\in L$ or false in $I_{2,0}$.
- Therefore, $I_{2,1} = I_{2,0}[\{R_b, R_c, I_{bc}, I_{cb}\}/\boldsymbol{f}]$ is a valid extension of the well-founded sequence.
- $I_{2,1}$ is two-valued and therefore terminal; it is equal to $I_2$.

**Lemma 4.2.** *Let $I$ be a $\Sigma$-interpretation. If there exists a $\Delta$-witness for $I$, then $I \models \Delta$.*

*Proof.* Let $I$ be a two-valued interpretation, and $J$ a witness for $I$. We will use $J$ to construct a sequence $\langle I_i \rangle_{0 \leq i \leq n}$ of three-valued interpretations. We prove that the sequence is a well-founded sequence, and that $I_n = I$, and therefore $I \models \Delta$. We denote by $\text{dom}(I_i)$ the set of atoms for which $I_i$ is two-valued.

The induction hypothesis, for each $i$, is that $I_i(p) = I(p)$ for each $p \in \text{dom}(I_i)$, and if $i > 0$, then the relation between $I_{i-1}$ and $I_i$ satisfies the criteria of a well-founded sequence.

Let $I_0 = I|_{Open(\Delta)}$. $I_0$ clearly satisfies the induction hypothesis.

Suppose we have constructed $I_i$, and $i$ satisfies the induction hypothesis. We denote by $D_i$ the set of literals $\widehat{\text{dom}(I_i)}$, and by $U_i$ the unassigned literals $\widehat{\Sigma} \setminus D_i$.

If $U_i$ is empty, then $I_i$ is two-valued, and therefore $I_i = I$, and we are done.

If there exists a literal $l$ in $U_i$ with $DJ_J(l) \subseteq D_i$, then we extend the sequence with $I_{i+1}$, the interpretation derived from $I_i$ by changing $I_{i+1}(l)$ to $I(l)$. It is easy to see that then $i+1$ satisfies the induction hypothesis.

If there is no such literal left and $U_i$ is not empty, then we will show that a set of atoms can be found that is an unfounded set with respect to $I_i$. Let the graph $G$ be the restriction of $J$ to $U_i$. Since for each $l \in U_i$, $DJ_J(l)$ contains an $l' \in U_i$, $G$ contains a loop, and contains a minimal strongly connected component $L$. Note that $L$ has no outgoing edges in $G$. In $J$, however, literals $l \in \mathcal{C}_{\text{lits}} \cap L$ may have outgoing edges, i.e., for some literals $l' \in \varphi_l$, $l'$ may be in $D_i$.

Now let the graph $G'$ be the restriction of $J$ to $\overline{L}$. Observe that since $\overline{L} \subseteq U_i$, $G'$ is a subgraph of $G$. We show that also each $l \in G'$ has a child in $G'$:

- if $l \in \mathcal{D}_{\text{lits}}$, then because there is no literal $l'$ in $U_i$ with $DJ_J(l') \subseteq D_i$, we have $d_J(l) \in U_i$. We also have $\neg l \in \mathcal{C}_{\text{lits}}$, and because $\neg l \in L$, each $l' \in \varphi_{\neg l} \cap U_i$ is also in $L$, hence $d_J(l) \in \overline{L}$, hence $d_J(l) \in G'$;

- if $l \in \mathcal{C}_{\text{lits}}$, then $d_J(\neg l) \in L$, hence $\neg d_J(\neg l) \in \overline{L}$, hence $d_J(l) \in G'$.

Thus, $G'$ contains a minimal strongly connected component $L'$. Because $J$ is a witness for $I$, for any loop in $J$ that contains an atom, all literals in it are false in $I$. Since both $L$ and $L'$ are loops in $J$, and $\overline{L'} \subseteq L$, it is impossible that both $L'$ and $L$ contain an atom.

Suppose $L'$ contains an atom. Hence $L$ contains no atoms at all, or $\overline{L}$ contains only atoms. We show that all atoms in $\overline{L}$ must be false. Assume towards contradiction that some $l \in \overline{L}$ has $I(l) = \boldsymbol{t}$. Since $J$ supports $I$, we find $I(l') = \boldsymbol{t}$ for all $l' \in Sub(J, l)$ by Proposition 4.4. Also, any $l' \in \overline{L}$ contains at least one child in $G'$; if $I(l') = \boldsymbol{t}$ then that child is also in in $Sub(J, l)$. Thus we find a loop $L'' \subseteq \overline{L}$ in $G'$, for which all literals are true. Since $L''$ is also a loop in $J$ and $J$ is a witness for $I$, $L''$ contains no atoms. Contradiction.

Thus, still supposing $L'$ contains an atom, we have that $\overline{L}$ is a loop consisting of false atoms in $I$. We show that $\overline{L}$ is an unfounded set with respect to $I_i$.

- Take any $l \in \mathcal{D}_{\text{lits}} \cap \overline{L}$. By construction of $L$, for each $l' \in \varphi_l$ we have either $l' \in \overline{L}$ or $l' \in D_i$. In the former case $I(l') = \boldsymbol{f}$ is already known, in the latter case $I(l') = \boldsymbol{f}$ follows from the fact that $J$ supports $I$ (if $I(l') = \boldsymbol{t}$, then $I(\bigwedge \varphi_{\neg l}) = \boldsymbol{f}$, hence, by support, $I(\neg l) = \boldsymbol{f}$, hence $I(l) = \boldsymbol{t}$). We find $I(l') = \boldsymbol{f}$ for each $l' \in \varphi_l$ as desired.

- Take any $l \in \mathcal{C}_{\text{lits}} \cap \overline{L}$. By construction of $L$, $\neg d_J(\neg l) \in \overline{L}$, i.e., we find some $l' \in \varphi_l$ with $I(l') = \boldsymbol{f}$ as desired, namely $l' = \neg d_J(\neg l)$.

Now suppose that $L'$ contains no atoms. Then $\overline{L'}$ consists entirely of atoms, and because $L$ is a loop in $J$, $J$ is loop-safe in $I$ and $\overline{L'} \subseteq L$, they are all false in $I$. In the same way as in the previous case we find that $\overline{L'}$ is an unfounded set with respect to $I_i$.

In either case, let $I_{i+1}$ be the interpretation derived from $I_i$ by changing $I_{i+1}(l)$ to $I(l)$ (which is $\boldsymbol{f}$) for each $l$ in the unfounded set. Again $i + 1$ satisfies the induction hypothesis. □

*Proof.* (of Theorem 4.1) This is a direct consequence of Lemmas 4.1 and 4.2. □

### 4.3.3 Three-valued semantics

Theorem 4.1 is an interesting result for SAT(ID) solving: it tells us that we can verify that an interpretation $I$ is a model by finding a witness for it, or prove that it is not a model by showing that there exists no witness for it. However, in practical algorithms, we will *construct* a model: during its construction, we have a three-valued interpretation. We will prefer to reject candidate interpretations before they are two-valued. To this end we extend the result, and the definition of witness, to three-valued semantics.

**Definition 4.7** (Three-valued support)**.** Let $J$ be a justification for $\Delta$, $I$ a three-valued $\Sigma$-interpretation. Then $J$ *three-valuedly supports* $I$ iff, for each $l \in \widehat{Def(\Delta)}$, if $I(\bigwedge DJ_J(l)) \neq \boldsymbol{u}$ then $I(l) = I(\bigwedge DJ_J(l))$.

Recall that the truth of the conjunction $\bigwedge DJ_J(l)$ can be seen as a reason for the truth of a defined literal $l$. The requirement for three-valued support is that for each defined literal $l$, whenever that conjunction is two-valued in $I$, $l$ itself must have the same truth value in $I$. It is then possible that a defined literal already has a two-valued truth value, while its body is still three-valued. This state may come up often in practical SAT(ID) algorithms.

Observe that any justification 3-valuedly supports the empty interpretation, and that for 2-valued interpretations the notions of support and of 3-valued support coincide.

**Example 4.10.** Consider the definition $\Delta_{4.10} = \left\{ \begin{array}{ll} p \leftarrow a, & q \leftarrow \neg a \end{array} \right\}$, and the three-valued interpretation $I = \{p \mapsto \boldsymbol{f}, q \mapsto \boldsymbol{f}, a \mapsto \boldsymbol{u}\}$. There exists but one justification for $\Delta_{4.10}$, and it 3-valuedly supports $I$. However, it supports neither $I \cup \{a \mapsto \boldsymbol{f}\}$ nor $I \cup \{a \mapsto \boldsymbol{t}\}$.

**Definition 4.8** (Witness, generalized)**.** Let $J$ be a justification for $\Delta$, $I$ a 3-valued $\Sigma$-interpretation. Then $J$ *is a witness for* $I$ iff $J$ 3-valuedly supports $I$ and $J$ is loop-safe in $I$.

This concept generalizes the concept of witness for two-valued interpretations: if $I$ is a two-valued interpretation, then a generalized witness $J$ for $(I, I)$ is a witness for $I$.

During SAT(ID) solving we may wish to reject three-valued interpretations that cannot be strengthened to a two-valued interpretation that has a witness.

### 4.3.4 Simplifications

#### Loop-safeness

Definition 4.4 of loop-safeness involves not only a condition on loops, but also a condition on truth values of literals in those loops. We may wonder whether the latter can be eliminated, i.e. whether the concept of loop-safeness can be made independent of interpretations. The following example illustrates that for some definitions, any justification will contain loops with atoms: incorporating a condition on truth values in the definition of loop-safeness therefore seems appropriate.

**Example 4.11.** Let $\Delta_{4.11} = \left\{ \begin{array}{ll} p \leftarrow q, & q \leftarrow p \end{array} \right\}$. There is a unique justification $J$ for $\Delta_{4.11}$; it contains the loops $\{p, q\}$ and $\{\neg p, \neg q\}$. The former contains an atom, and since this loop cannot be avoided ($J$ is unique), its literals have to be false. Hence $\{p \mapsto \boldsymbol{f}, q \mapsto \boldsymbol{f}\}$ is the only model of $\Delta_{4.11}$.

We can simplify the concept of loop-safeness however by identifying the atoms that are unavoidably part of a loop in any justification: they have to be false in any model.

**Definition 4.9** ($\Delta^{\emptyset}$)**.** We define the set of *unavoidably false* literals $\circlearrowright$ as $\{l \in \widehat{Def(\Delta)} \mid$ for any justification $J$ for $\Delta$, $Sub(J, l)$ contains a non-negative loop$\}$.

The *loop-simplification of* $\Delta$, denoted $\Delta^{\varnothing}$, is the definition $\{\ p \leftarrow \neg p\ \}$ if $\circlearrowright$ contains both some atom $p$ and its negation $\neg p$; else it is the definition obtained from $\Delta$ by replacing

- $p \leftarrow \varphi_p$ by $p \leftarrow \bot$ for each atom $p \in \circlearrowright$; and
- $p \leftarrow \varphi_p$ by $p \leftarrow \top$ for each negative literal $\neg p \in \circlearrowright$.

If the set of unavoidably false literals contains both an atom and its negation, the definition is unsatisfiable. Otherwise, all literals in the set are indeed unavoidably false, i.e., false in any model of the definition. In the loop-simplification, all the unavoidable loops are broken by the replacements.

The following proposition follows easily:

**Proposition 4.5.** $\Delta \cong \Delta^{\varnothing}$.

The interesting part of $\Delta^{\varnothing}$ is the set of rules that have *not* been changed: it corresponds to that part of $\Delta$ that really depends on $Open(\Delta)$. A SAT(ID) solver could compute (the truth values imposed by)[2] $\Delta^{\varnothing}$ in an initialization phase.

**Example 4.12.** Let $\Delta_{4.12} =$

$$\left\{ \begin{array}{lll} p \leftarrow q \vee r, & q \leftarrow p, & r \leftarrow p, \\ s \leftarrow t \vee a, & t \leftarrow s & \end{array} \right\}.$$

An overview of possible justifications $J$ for $\Delta_{4.12}$ is shown in Figure 4.3. We have that $p \in \circlearrowright$, because when $J$ is such that $d_J(p) = q$, then $\{p, q\}$ is a positive loop; when $d_J(p) = r$, then $\{p, r\}$ is a positive loop. Also $q \in \circlearrowright$ because if $d_J(p) = q$, $\{p, q\}$ is a positive loop, and if $d_J(p) = r$, $q$ depends on the positive loop $\{p, r\}$; $r \in \circlearrowright$ for analogous reasons. $s \notin \circlearrowright$, because if $d_J(s) = a$, $Sub(J, s)$ is loop-free. Non of the negative literals is in $\circlearrowright$, because the loops in their subjustifications are negative.

Hence $\circlearrowright = \{p, q, r\}$, and $\Delta^{\varnothing}_{4.12} =$

$$\left\{ \begin{array}{lll} p \leftarrow \bot, & q \leftarrow \bot, & r \leftarrow \bot, \\ s \leftarrow t \vee a, & t \leftarrow s & \end{array} \right\}.$$

$\Delta^{\varnothing}_{4.12}$ (and hence $\Delta_{4.12}$) has two models: $\{p \mapsto \boldsymbol{f}, q \mapsto \boldsymbol{f}, r \mapsto \boldsymbol{f}, s \mapsto \boldsymbol{t}, t \mapsto \boldsymbol{t}, a \mapsto \boldsymbol{t}\}$, $\{p \mapsto \boldsymbol{f}, q \mapsto \boldsymbol{f}, r \mapsto \boldsymbol{f}, s \mapsto \boldsymbol{f}, t \mapsto \boldsymbol{f}, a \mapsto \boldsymbol{f}\}$. The truth values $\{p \mapsto \boldsymbol{f}, q \mapsto \boldsymbol{f}, r \mapsto \boldsymbol{f}\}$ can be computed in an initialization phase.

We can now return to our initial concern regarding loop-safeness. The following result shows that $\Delta^{\varnothing}$ has a nice property: for any $I \models \Delta^{\varnothing}$, there exists a $\Delta^{\varnothing}$-witness that has no non-negative loops at all. Thus, for loop-simplified definitions, loop-safeness is indeed independent of interpretations.

[2]In a practical implementation, the replacements by $\top$ and $\bot$ naturally do not have to be executed explicitly.

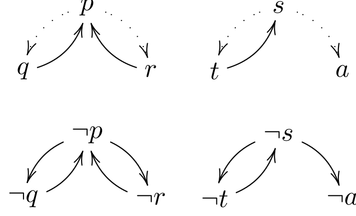



Figure 4.3: Overview of possible justifications for $\Delta_{4.12}$.

**Proposition 4.6.** *Let $I$ be a $\Sigma$-interpretation. $I \models \Delta$ iff there exists a justification for $\Delta^{\emptyset}$ that supports $I$, and in which all loops are negative.*

*Proof.* We have $I \models \Delta$ iff $I \models \Delta^{\emptyset}$ by Proposition 4.5, and $I \models \Delta^{\emptyset}$ iff there exists a $\Delta^{\emptyset}$-witness for $I$ by Theorem 4.1. We still need to prove that if there exists a $\Delta^{\emptyset}$-witness for $I$, there also exists one in which all loops are negative.

Suppose, towards contradiction, the contrary: that any $\Delta^{\emptyset}$-witness $J$ for $I$ contains a non-negative loop. Let $J$ be such a witness, and $L$ a non-negative loop in $J$. All literals in $L$ are false in $I$ because $J$ is a witness. Suppose, first, that $L \cap \mathcal{D}_{\text{lits}} = \emptyset$. Then all literals in $L$ are unavoidably false, and this contradicts the fact that $\Delta^{\emptyset}$ is a loop-simplification. Therefore $L \cap \mathcal{D}_{\text{lits}} \neq \emptyset$; let $l \in L \cap \mathcal{D}_{\text{lits}}$. Consider the justification $J'$ obtained from $J$ by setting $d_{J'} := l'$, for some literal $l' \in \varphi_l$ with $l' \neq d_J(l)$. By $I \models \Delta^{\emptyset}$ we have that all literals in $\varphi_l$ are false in $I$, therefore $J'$ also supports $I$, and since it coincides with $J$ everywhere but in $l$, it is therefore also a $\Delta^{\emptyset}$-witness for $I$. Hence $J'$ also contains a non-negative loop.

- Suppose $J'$ also contains a non-negative loop, $L'$, through $l$. The above procedure of changing the justification in one literal of $L \cap \mathcal{D}_{\text{lits}}$, or of $L' \cap \mathcal{D}_{\text{lits}}$, can be repeated: if for any such change (recursively), the new justification contains a loop through the literal where the justification was changed, then for any justification $J''$, we have that $Sub(J'', l)$ contains a non-negative loop. Hence $l$ is again unavoidably false, which contradicts the fact that $\Delta^{\emptyset}$ is a loop-simplification.

- Therefore there exists a set of such changes from $J$, such that in the resulting witness $J^{(3)}$, $Sub(J^{(3)}, l)$ contains no non-negative loops. However, $J^{(3)}$ is a $\Delta^{\emptyset}$-witness for $I$, and therefore contains a non-negative loop which is totally independent of $L$. We can repeat the same process on this loop, resolving the loop, and finding another new independent non-negative loop. This therefore leads to an infinite number of independent loops, which contradicts the fact that the number of loops is finite.

□

**Stable justifications**

In Definition 2.6 we defined inductive definition semantics using the well-founded semantics *extending interpretations of the open symbols*. We now introduce a similar definition of the stable model semantics; it will help us relate the semantics of total inductive definitions to the stable model semantics (Corollary 4.1). This corollary will be used in Chapter 5 to relate the computational properties of SAT(ID) and ASP.

**Definition 4.10** (Stable model extending $I_O$)**.** Let $I$ be a $\Sigma$-interpretation, $I_O$ an $Open(\Delta)$-interpretation, and $I'_O$ the empty extension of $I_O$ to $\Sigma$. Then ${}^{I_O}\Delta$ is the definition obtained from $\Delta$ by removing rules $(p \leftarrow \varphi_p)$ for which $I'_O(\varphi_p) = \boldsymbol{f}$, and replacing rules $(p \leftarrow \varphi_p)$ for which $I'_O(\varphi_p) = \boldsymbol{t}$ by $(p \leftarrow \top)$. *$I$ is a stable model of $\Delta$ extending $I_O$* iff $I|_{Def(\Delta)}$ is a stable model of ${}^{I_O}\Delta$ and $I|_{Open(\Delta)} = I_O$. We denote this by $I \models_{st} \Delta$.

**Example 4.13.** Let $\Delta_{4.13} = \left\{\ p \leftarrow \neg q \vee a, \quad q \leftarrow \neg p\ \right\}$. Then $\{a \mapsto \boldsymbol{t}, p \mapsto \boldsymbol{t}, q \mapsto \boldsymbol{f}\} \models_{st} \Delta_{4.13}$, $\{a \mapsto \boldsymbol{f}, p \mapsto \boldsymbol{t}, q \mapsto \boldsymbol{f}\} \models_{st} \Delta_{4.13}$, and $\{a \mapsto \boldsymbol{f}, p \mapsto \boldsymbol{f}, q \mapsto \boldsymbol{t}\} \models_{st} \Delta_{4.13}$. Note that there are two stable models extending $\{a \mapsto \boldsymbol{f}\}$.

It follows straightforwardly from a result by Van Gelder et al. (1991) that if a definition $\Delta$ is total with respect to a theory $T$, then for each $I \models T$, $\Delta$ has a unique stable model extending $I|_{Open(\Delta)}$, and it coincides with the well-founded model of $\Delta$ extending $I|_{Open(\Delta)}$.

We adapt the definitions and results of Section 4.3.1 to stable semantics. This has the advantage of simplifying matters, which may also lead to a computational advantage.

**Definition 4.11** (Stable justification)**.** A *stable justification for* $\Delta$ is a directed graph $(V, E)$ where (1) $V = \widehat{\Sigma}$, and $E$ such that each atom $p \in \mathcal{D}_{\text{lits}}$ has exactly one edge $(p, l) \in E$, where $l \in \varphi_p$; (2) each atom $p \in \mathcal{C}_{\text{lits}}$ has edges $(p, l) \in E$ for each $l \in \varphi_p$; and (3) $E$ has no other edges.

A stable justification can be seen as a simplification of a (general) justification: it has only edges leaving from defined *atoms*, not from their negative literals. This implies that the only type of loops possible in a stable justification are positive loops.

**Example 4.14.** Let $\Delta_{4.14} =$

$$\left\{\begin{array}{l} p \leftarrow \neg q, \\ q \leftarrow \neg p, \\ r \leftarrow r \end{array}\right\}.$$

Figure 4.4a illustrates a (the only) stable justification for $\Delta_{4.14}$, Figure 4.4b illustrates a (the only) justification for $\Delta_{4.14}$.

**Definition 4.12** (Stable witness)**.** Let $J$ be a stable justification for $\Delta$, $I$ a $\Sigma$-interpretation. *$J$ stably supports $I$* if, for each $p \in Def(\Delta)$, $I(p) = I(\bigwedge DJ_J(l))$. *$J$ is stably loop-safe in $I$* if any loop in $J$ consists of false atoms in $I$. *$J$ is a stable witness for $I$* iff $J$ stably supports $I$ and is stably loop-safe in $I$.

(a) Stable justification (b) (General) justification

Figure 4.4: Justifications for $\Delta_{4.14}$.

A stable witness for an interpretation has the same function as a witness, only now for stable semantics: its existence entails that the interpretation is a stable model. This result can be derived as a simplification of the proof of Theorem 4.1.

**Proposition 4.7.** *Let $I$ be a $\Sigma$-interpretation. $I \models_{st} \Delta$ iff there exists a stable witness for $I$.*

**Example 4.15.** Example 4.14 continued. $\Delta_{4.14}$ does not have any model, because its only justification (cf. Figure 4.4b) cannot be a witness for any interpretation. Indeed, for loop-safeness, the literals in both loops $\{p, \neg q\}$ and $\{\neg p, q\}$ all need to be false. However, $\Delta_{4.14}$ does have stable models (extending $\emptyset$). We have both $\{p \mapsto \boldsymbol{t}, q \mapsto \boldsymbol{f}, r \mapsto \boldsymbol{f}\} \models_{st} \Delta$ and $\{p \mapsto \boldsymbol{f}, q \mapsto \boldsymbol{t}, r \mapsto \boldsymbol{f}\} \models_{st} \Delta$. Indeed, the stable justification of Figure 4.4a is a stable witness for both interpretations. Note that $r$ must be false because $\{r\}$ is a loop.

From Van Gelder et al.'s result mentioned earlier and Proposition 4.7 we derive the following result.

**Corollary 4.1.** *Let $\Delta$ be total with respect to $T$, and $I \models T$. Then $I \models \Delta$ iff there exists a stable witness for $I$.*

Recall that we expect most definitions to be total with respect to the rest of the theory in which they occur. Since stable justifications have a smaller domain than justifications, we may therefore prefer searching for a stable witness over searching for a witness.

We adapt also Definition 4.9 of the loop-simplification of a definition to stable semantics.

**Definition 4.13** (Stable loop-simplification)**.** We define the set of *stable unavoidably false* literals $\circlearrowright_{st}$ as $\{l \in Def(\Delta) \mid$ for any stable justification $J$ for $\Delta$, $Sub(J, l)$ contains a loop$\}$. The *stable loop-simplification of* $\Delta$ is the definition obtained from $\Delta$ by replacing $p \leftarrow \varphi_p$ by $p \leftarrow \bot$ for each atom $p \in \circlearrowright_{st}$.

Since the only type of loops possible in a stable justification are positive loops, $\circlearrowright_{st}$ is defined as a set of atoms, not literals. We can therefore only derive falsity of defined atoms here.

Observe that the condition of loop-safeness simplifies to *loop-freeness* for stable loop-simplifications.

**Example 4.16.** Examples 4.14–4.15 continued. The stable loop-simplification of $\Delta_{4.14}$ is

$$\left\{ \begin{array}{l} p \leftarrow \neg q, \\ q \leftarrow \neg p, \\ r \leftarrow \bot \end{array} \right\}.$$

### 4.3.5 Related work

Justifications for the well-founded semantics were first defined, albeit in a different form, by Denecker and De Schreye (1993), and further elaborated by Denecker (1993). To the best of our knowledge, our works in (Mariën et al., 2005, 2007b, 2008) are the only other publications using justifications for the well-founded semantics.

In the context of stable model semantics, similar notions have been defined. For instance, Pontelli and Son's (2006) justifications are very similar to our stable justifications. They used the concept for debugging of Stable logic programs: a justification provides a *reason* for the truth of an atom in a stable model, and can therefore be used to explain unintended truths. Anger et al. (2005) used graph structures with comparable characteristics to implement the *nomore++* system for the stable model semantics.

In a wider context, graph structures that "justify" interpretations have been used for various purposes, such as for Prolog proof verification (Roychoudhury et al., 2000), and for local search for SAT (Järvisalo et al., 2008).

## 4.4 A propositional logic characterization of definitions

This section introduces *loop formula semantics*. We characterize propositional inductive definitions in propositional logic. In other words, we establish a reduction from PC(ID) to PC. This reduction introduces new formulas (in the original vocabulary) to the theory.

While there may be too many new formulas to make this reduction useful as a practical means of computing PC(ID) models, it is quite possible to add only those formulas that are relevant during the search for such models—a significantly smaller number of formulas. As such, these formulas can be considered as representing ID-based propagations. We elaborate on this extensively in Chapter 5. The general idea of PC formulas that represent propagation for an extension of PC is called *theory propagation* in the domain of satisfiability modulo theories, and is discussed in, e.g., (Ganzinger et al., 2004; Nieuwenhuis et al., 2006).

The new formulas are called *loop formulas*: propositional formulas which eliminate specific unwanted models of a definition's completion. The idea of loop formulas was proposed, for stable model semantics, by Lin and Zhao (2004).

After we have established the main result in Section 4.4.1, we discuss the relation of our loop formulas with Lin and Zhao's in Section 4.4.2, and discuss

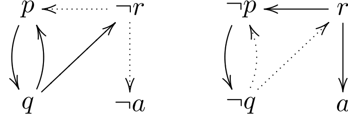



Figure 4.5: Dependency graph of $\Delta_{4.17}$

some consequences in Section 4.4.3.

### 4.4.1 Loop formulas

For a given DefNF definition $\Delta$, we want to define a set of formulas $\Phi(\Delta)$ such that $\Delta \cong comp(\Delta) \cup \Phi(\Delta)$. It is clear that these formulas should involve literals that depend on themselves, hence the following concept:

**Definition 4.14** (Dependency graph)**.** The *dependency graph* of $\Delta$ is the graph with vertices $\widehat{\Sigma}$ and edges $(l, l')$ for each $l \in \widehat{Def(\Delta)}$ and $l' \in \varphi_l$. We call a loop in $\Delta$'s dependency graph simply *a loop in* $\Delta$.

**Example 4.17.** Let $\Delta_{4.17}$ =

$$\left\{\begin{array}{l} p \leftarrow q, \\ q \leftarrow p \wedge \neg r, \\ r \leftarrow \neg p \wedge a \end{array}\right\}.$$

The dependency graph of $\Delta_{4.17}$ is shown in Figure 4.5. Dotted edges leave from literals in $\mathcal{D}_{\text{lits}}$, solid edges from literals in $\mathcal{C}_{\text{lits}}$. The loops in $\Delta_{4.17}$ are $\{p, q\}$, $\{p, q, \neg r\}$, $\{\neg p, \neg q\}$, and $\{\neg p, \neg q, r\}$. The models of $\Delta_{4.17}$ are $\{a \mapsto \boldsymbol{t}, p \mapsto \boldsymbol{f}, q \mapsto \boldsymbol{f}, r \mapsto \boldsymbol{t}\}$ and $\{a \mapsto \boldsymbol{f}, p \mapsto \boldsymbol{f}, q \mapsto \boldsymbol{f}, r \mapsto \boldsymbol{f}\}$.

Note that if $L$ is a loop in a definition $\Delta$, then so is $\overline{L}$. Also note that the dependency graph of $\Delta$ is the union of all justifications for $\Delta$. A loop of any justification for $\Delta$ is a loop in $\Delta$, but not necessarily vice versa.

If $L$ is a loop in $\Delta$, then the literals in $L$ depend on other literals in $L$, but they may also depend on literals external to $L$.

**Definition 4.15** (External disjuncts, conjuncts)**.** Let $L \subseteq \widehat{Def(\Delta)}$ be a set of defined literals. We define *L's external disjuncts* $\mathcal{D}^{\text{ext}}(L)$ as $\left(\bigcup_{l \in \mathcal{D}_{\text{lits}} \cap L} \varphi_l\right) \setminus L$, and *L's external conjuncts* $\mathcal{C}^{\text{ext}}(L)$ as $\left(\bigcup_{l \in \mathcal{C}_{\text{lits}} \cap L} \varphi_l\right) \setminus L$.

**Example 4.18.** Example 4.17 continued. We give $\mathcal{D}^{\text{ext}}(L)$ and $\mathcal{C}^{\text{ext}}(L)$ for each loop $L$ in $\Delta_{4.17}$.

| $L$ | $\{p, q\}$ | $\{p, q, \neg r\}$ | $\{\neg p, \neg q\}$ | $\{\neg p, \neg q, r\}$ |
|---|---|---|---|---|
| $\mathcal{D}^{\text{ext}}(L)$ | $\emptyset$ | $\{\neg a\}$ | $\{r\}$ | $\emptyset$ |
| $\mathcal{C}^{\text{ext}}(L)$ | $\{\neg r\}$ | $\emptyset$ | $\emptyset$ | $\{a\}$ |

We elaborate on some instances from this table: consider the loop $\{p, q\}$. Neither $p$ nor $q$ belongs to $\mathcal{D}_{\text{lits}}$, hence $\mathcal{D}^{\text{ext}}(\{p, q\}) = \emptyset$. Consider instead the loop

$\{p, q, \neg r\}$. $\neg r$ is $\in \mathcal{D}_{\text{lits}}$ and it has body literals $\varphi_{\neg r} = \{p, \neg a\}$. We therefore have $\mathcal{D}^{\text{ext}}(\{p, q, \neg r\}) = \{p, \neg a\} \setminus \{p, q, \neg r\} = \{\neg a\}$.

The intuition behind the concept of loop and that of external disjuncts, can best be expressed by a formula.

**Definition 4.16** (Loop formula)**.** For any set of defined literals $L$, *$L$'s loop formula* $LF_{\Delta}(L)$ is defined as

$$\bigvee L \supset \bigvee \mathcal{D}^{\text{ext}}(L).$$

We have defined the concept of *loop formula* for arbitrary sets $L$ of defined literals, but it is most intuitive when $L$ is a loop. $L$'s loop formula then says: "if any literal in the loop is true, also some external disjunct should be true". In other words, it says that literals in a loop need some justification from literals outside the loop in order to be true. Observe that we could have equivalently formulated this as: "if all external disjuncts of a loop are false, then the literals in the loop should be false too", corresponding to the rewriting $\neg \bigvee \mathcal{D}^{\text{ext}}(L) \supset \neg \bigvee L$ of $L$'s loop formula. A special case is when a loop has no external disjuncts: then its loop formula simply says that all literals in the loop have to be false.

We show on an example that (and why) this intuition certainly holds true for *some* loops, but not for all.

**Example 4.19.** Examples 4.17–4.18 continued. Consider again loop $L = \{p, q, \neg r\}$, and consider how some well-founded sequence could derive any literal in $L$ to be true. To make $p$ true, first $q$ should be made true. To make $q$ true, first both $p$ and $\neg r$ should be made true. Finally, to make $\neg r$ true, first $p$ *or* $\neg a$ should be made true. Hence, only the truth of $\neg a$ can justify the truth of any literal of $L$. Observe that $\neg a$ is the only external disjunct of $L$. Hence, the loop formula of $L$, $p \vee q \vee \neg r \supset \neg a$, is satisfied. Observe that this is indeed the case in the models of $\Delta_{4.17}$.

Now consider also $L' = \{\neg p, \neg q\}$. Because $\{p, q\}$ is an unfounded set with respect to the empty interpretation, a well-founded sequence can make the literals of $L'$ true at any point; no justification from external literals is needed. The loop formula of $L'$, $\neg p \vee \neg q \supset r$, is not satisfied. Note that an unfounded set is always a set of *atoms*.

We try to build an intuition of which loops are *relevant*, i.e., for which loops the loop formula is satisfied. From the above example, we might guess that any loop that contains an atom is relevant. This is not the case, as the following example illustrates.

**Example 4.20.** Examples 4.17–4.19 continued. Consider also the loop $L'' = \{\neg p, \neg q, r\}$. Observe that $L' = \{\neg p, \neg q\}$, of which we showed in Example 4.19 that its loop formula is not satisfied, is a subloop of $L''$. The literals of $L'$ do not need external justification to be true. Thus, a well-founded sequence could make the literals of $L''$ true by first making the literals of $L'$ ($\neg p$ and $\neg q$) true,

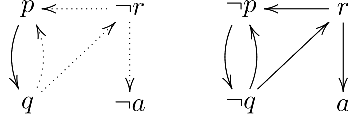



Figure 4.6: Dependency graph of $\Delta_{4.21}$

and then propagating this truth to the remaining literal, $r$. Hence also $L''$ does not need external justification because $L'$ acts as its "justifying kernel". The loop formula of $L''$ is not satisfied.

The previous example refines our intuition of which loops are relevant: those that do not contain a "justifying kernel". We now elaborate on this concept: it is already clear that a justifying kernel is a negative subloop; however, some extra conditions need to be applied.

**Example 4.21.** Consider $\Delta_{4.21} =$

$$\left\{ \begin{array}{l} p \leftarrow q, \\ q \leftarrow p \vee \neg r, \\ r \leftarrow \neg p \wedge a \end{array} \right\},$$

the dependency graph of which is shown in Figure 4.6. The only difference with $\Delta_{4.17}$ from Example 4.17 is that $q$ is now defined by a disjunction instead of a conjunction. $\Delta_{4.17}$ is not total: it has no model with $a \mapsto \boldsymbol{t}$. Its only model is $\{a \mapsto \boldsymbol{f}, p \mapsto \boldsymbol{t}, q \mapsto \boldsymbol{t}, r \mapsto \boldsymbol{f}\}$.

We consider the loop $L = \{\neg p, \neg q, r\}$ of $\Delta_{4.21}$. It has a subloop, $L' = \{\neg p, \neg q\}$, which is negative. We show that nevertheless, $L'$ alone cannot justify the truth of the literals of $L$. Consider again how a well-founded sequence could make $L$'s literals true. To make $r$ true, first both $\neg p$ and $a$ should be made true. $\neg p$ and $\neg q$ could be made true together (i.e., the atoms $\{p, q\}$ made false as an unfounded set), but then first $r$ should be true (since $q$ depends disjunctively on $\neg r$, or $\neg q$ conjunctively on $r$). Indeed, $\{p, q\}$ is an unfounded set *with respect to* $\{r \mapsto \boldsymbol{t}\}$. Since $r \in L$, this means that $L'$ cannot act as justifying kernel for $L$. The relevant property, therefore, is that $\mathcal{C}^{\text{ext}}(L') \cap L = \{r\} \neq \emptyset$.

Building on the above intuitions, we now formally define the concepts *justifying kernel* and *relevant loop.*

**Definition 4.17** (Justifying kernel, Relevant loop)**.** Let $L$, $L'$ be loops in $\Delta$. Then $L'$ is a *justifying kernel for* $L$ iff $L' \subseteq L$, $L'$ is negative, and $\mathcal{C}^{\text{ext}}(L') \cap L = \emptyset$. $L$ is a *relevant loop* iff it does not contain a justifying kernel.

**Example 4.22.** Examples 4.17–4.20 continued. $\{p, q\}$ and $\{p, q, \neg r\}$ are relevant loops in $\Delta_{4.17}$; $\{\neg p, \neg q\}$ is a justifying kernel for itself and for $\{\neg p, \neg q, r\}$.

Some properties immediately follow from the definition of justifying kernels and relevant loops:

- if a loop $L$ is positive, then it cannot contain a justifying kernel, and therefore it is a relevant loop;
- if a loop $L$ is negative, then it is its own justifying kernel, and therefore is not a relevant loop. I.e., a necessary condition for being a relevant loop is containing an atom;
- if $L'$ is a justifying kernel for $L$, then either $L' = L$ or $\mathcal{D}^{\text{ext}}(L') \cap L \neq \emptyset$. Indeed, if both $\mathcal{D}^{\text{ext}}(L') \cap L = \emptyset$ and $L' \subsetneq L$ were true, then because also $\mathcal{C}^{\text{ext}}(L') \cap L = \emptyset$ holds, literals in $L'$ could not reach literals in $L \setminus L'$, and hence $L$ would not be a loop.

Having now defined relevant loops, we can define the set of relevant loop formulas.

**Definition 4.18** (Relevant loop formulas)**.** The *relevant loop formulas of* $\Delta$ are

$$LF_\Delta = \bigcup_{L \text{ is a relevant loop of } \Delta} LF_\Delta(L).$$

**Example 4.23.** Examples 4.17–4.20, 4.22 continued. We have

$$\begin{aligned} LF_{\Delta_{4.17}} &= LF_{\Delta_{4.17}}(\{p,q\}) \cup LF_{\Delta_{4.17}}(\{p,q,\neg r\}) \\ &= \left(\bigvee\{p,q\} \supset \bigvee \emptyset\right) \cup \left(\bigvee\{p,q,\neg r\} \supset \bigvee\{\neg a\}\right) \\ &= (\neg p \wedge \neg q) \wedge (p \vee q \vee \neg r \supset \neg a)\,. \end{aligned}$$

The following proposition formally states the above intuitions: the loop formula of a relevant loop is satisfied in any model of $\Delta$.

**Proposition 4.8.** *A model of* $\Delta$ *is a model of* $LF_\Delta(L)$ *for any relevant loop* $L$*.*

*Proof.* Let $I$ be a model of $\Delta$. We assume towards contradiction that for some relevant loop $L$ in $\Delta$, we have $I \not\models LF_\Delta(L)$.

If $I \not\models LF_\Delta(L)$, then $L$'s external disjuncts are false, i.e. $I \models \neg \bigvee \mathcal{D}^{\text{ext}}(L)$, and $L$ itself contains a true literal, i.e., $I \models \bigvee L$. Let $l \in L$ have $I(l) = \boldsymbol{t}$.

By Theorem 4.1, $I$ has a $\Delta$-witness $J$. Then by Proposition 4.4, all literals in $Sub(J,l)$ are true. Since all literals of $\mathcal{D}^{\text{ext}}(L)$ are false, $d_J(l') \in L$ for each $l' \in \mathcal{D}_{\text{lits}} \cap L$, and if $l' \in Sub(J,l)$, then so is $d_J(l')$. Also for each $l' \in \mathcal{C}_{\text{lits}} \cap L \cap Sub(J,l)$ we find that some $l'' \in \varphi_{l'}$ is in $Sub(J,l) \cap L$, because $L$ is a loop. Let $G$ be the subgraph of $J$ obtained by restricting $J$ to $Sub(J,l) \cap L$. Then it follows that $G$ contains a minimal strongly connected component $L'$. Note that $L'$ is also a loop in $J$ and of $\Delta$. By its minimality, $L'$ has no outgoing edges in $G$, hence we find $\mathcal{C}^{\text{ext}}(L') \cap L = \emptyset$. Since $L'$ consists of true literals in $I$, and $J$ is loop-safe in $I$, $L'$ is a negative loop.

Hence, we have found a justifying kernel $L'$ for $L$, which contradicts the fact that $L$ is a relevant loop. □

The following is an easy consequence:

**Lemma 4.3.** $\Delta \models comp(\Delta) \cup LF_\Delta$.

*Proof.* Let $I \models \Delta$. Then $I \models comp(\Delta)$ is a long-standing result (see Proposition 4.1), and $I \models LF_\Delta$ follows trivially from Proposition 4.8 and the definition of $LF_\Delta$. □

We now show that we don't just obtain the above entailment result, but a general equivalence result.

**Theorem 4.2.** $\Delta \cong comp(\Delta) \cup LF_\Delta$.

To prove the result in the other direction, we introduce an auxiliary concept called *partial justification*.

**Definition 4.19.** (Partial justification) A *partial justification for* $\Delta$ is a partial function $PJ : \mathcal{D}_{\text{lits}} \to \widehat{\Sigma}$ such that for each literal $l$ in its domain $\text{dom}(PJ)$, $PJ(l) \in \varphi_l$.

The *empty* partial justification is the one with the empty domain. Observe also that if $J$ is a justification or $\Delta$, then $d_J$ is a partial justification with $\text{dom}(d_J) = \mathcal{D}_{\text{lits}}$.

**Definition 4.20.** A justification $J$ for $\Delta$ is an *instance* of a partial justification $PJ$ for $\Delta$ if for each $l \in dom(PJ)$, $d_J(l) = PJ(l)$.

**Definition 4.21.** (Restricted dependency graph, loop of a partial justification) The *restricted dependency graph* of a partial justification $PJ$ for $\Delta$ is the subgraph $Dep(PJ)$ of the dependency graph of $\Delta$ obtained by removing, for each $l \in \text{dom}(PJ)$, all edges leaving $l$, except $(l, PJ(l))$.
A loop in $Dep(PJ)$ is called *a loop of* $PJ$.

We distinguish again between negative, positive and mixed loops. A non-negative loop is one that contains an atom.

Observe that the restricted dependency graph of the empty partial justification is nothing else than the dependency graph itself. Thus, the concept of a partial justification can be seen as a generalization of the notions of dependency graph and justification.

**Definition 4.22.** A partial justification $PJ$ *supports* an interpretation $I$ if $I(l) = I(PJ(l))$ for each literal $l \in \text{dom}(PJ)$.

**Definition 4.23.** (Dangerous loop) We call a loop $L$ of a partial justification *dangerous* in interpretation $I$ if it is non-negative and contains a true literal in $I$.

**Definition 4.24.** A partial justification $PJ$ is *loop-safe* in $I$ if it contains no dangerous loops in $I$. I.e., all non-negative loops consist of false literals.

On the set of partial justifications for $\Delta$, we can define a precision order: $PJ \leq_p PJ'$ if $\text{dom}(PJ) \subseteq \text{dom}(PJ')$ and for each $l \in \text{dom}(PJ)$, $PJ(l) = PJ'(l)$. Clearly, if $J$ is an instance of $PJ$, then $PJ \leq_p d_J$.

The following result is straightforward:

**Proposition 4.9.** *Let $PJ, PJ'$ be partial justifications with $PJ \leq_p PJ'$. Then a loop of $PJ'$ is a loop of $PJ$.*

As a consequence, a loop of any partial justification is a loop of the empty partial justification and hence of $\Delta$, and a loop of an instance $J$ of some partial justification $PJ$ is a loop of $PJ$. In particular, if $PJ$ has no dangerous loops in $I$, then none of its instances has, i.e., all instances of a loop-safe partial justification are loop-safe.

**Proposition 4.10.** *Let $I$ be an interpretation with $I \models comp(\Delta)$ and $PJ$ a partial justification that supports $I$ and is loop-safe in $I$. Then $PJ$ has an instance that is a witness for $I$.*

*Proof.* Let $J$ be an arbitrary instance of $PJ$ for which $I(l) = I(d_J(l))$ for all $l \notin \mathrm{dom}(PJ)$ (this exists because $I \models comp(\Delta)$). Then clearly $J$ supports $I$. As shown above, any instance of $PJ$ is loop-safe in $I$, hence $J$ is a witness. □

We now prove the main result in the other direction.

**Lemma 4.4.** $comp(\Delta) \cup LF_\Delta \models \Delta$.

*Proof.* Let $I$ be a model of $comp(\Delta) \cup LF_\Delta$. To prove that $I \models \Delta$, we will show that $I$ has a $\Delta$-witness. All references to dangerous loops and loop-safeness are with respect to $I$, which is fixed throughout the proof.

We construct a sequence $\langle PJ_i \rangle_{0 \leq i \leq n}$ of increasingly precise partial justifications. The sequence is constructed in such a way that the following induction hypotheses hold for each $i$: *a*) $PJ_i$ supports $I$; *b*) $PJ_j <_p PJ_i$ for all $0 \leq j < i$ (i.e. we have increasing precision); and *c*) any dangerous loop of $PJ_i$ is disjoint from $\mathrm{dom}(PJ_i)$. Moreover, the last element $PJ_n$ of the sequence will be proven to be loop-safe. It then follows from part *a* of the hypothesis and Proposition 4.10 that $PJ_n$ has an instance that is a witness for $I$, and by Theorem 4.1, we obtain that $I \models \Delta$.

Note that because of its strictly increasing precision, the length of this sequence is bound by $|\mathcal{D}_{\mathrm{lits}}|$. For the same reason, the set of dangerous loops of $PJ_i$ decreases with increasing $i$. In fact, each new partial justification in the sequence is constructed such that this decrease is also strict.

We take for $PJ_0$ the empty partial justification. Clearly the induction hypothesis holds. Assume that we obtained $PJ_i$. If $PJ_i$ has no dangerous loops, it is loop-safe; then set $n := i$ and we are done. Otherwise, take a subset maximal dangerous loop $L$. Such a loop exists since the set of dangerous loops of $PJ_i$ is not empty. Note that for an arbitrary dangerous loop $L'$ of $PJ_i$ with $L' \cap L \neq \emptyset$, $L'$ is a subset of $L$. Also note that by induction hypothesis $c$, $\mathrm{dom}(PJ_i) \cap L = \emptyset$.

There are two cases:

**There is a literal $d \in \mathcal{D}^{\mathbf{ext}}(L)$ such that $I(d) = \boldsymbol{t}$.** Let $l \in L \cap \mathcal{D}_{\mathrm{lits}}$ be a literal such that $d \in \varphi_l$. Since $I \models comp(\Delta)$, we also have $I(l) = \boldsymbol{t}$. Define $PJ_{i+1}$ by extending $PJ_i$ with $PJ_{i+1}(l) = d$. Parts $a$ and $b$ of the induction hypothesis are clearly satisfied for $i+1$. As for part $c$, assume towards contradiction that $PJ_{i+1}$ has a dangerous loop $L'$ such that $L' \cap \mathrm{dom}(PJ_{i+1}) \neq$

$\emptyset$. By Proposition 4.9, this is a loop of $PJ_i$ as well, hence it follows from the induction hypothesis $c$ that $L' \cap \mathrm{dom}(PJ_{i+1}) = \{l\}$. Hence, $L'$ is a loop of $PJ_i$ with a non-empty intersection with $L$. Since the latter is maximal, we have $L' \subseteq L$. On the other hand, since $L'$ is a loop of $PJ_{i+1}$ and $l \in L'$, we have also $PJ_{i+1}(l) = d \in L'$, which is impossible since $d \notin L$. Thus, we obtain the desired contradiction. It follows that $c$ holds for $PJ_{i+1}$.

**There is no literal $d \in \mathcal{D}^{\mathbf{ext}}(L)$ such that $I(d) = \boldsymbol{t}$.** Consider the set $T$ of true literals of $L$: $T = \{l \in L \mid I(l) = \boldsymbol{t}\}$. Note that $T$ is not empty since $L$ is dangerous. Let $l$ be an arbitrary element of $T$. It is easy to see that $Dep(PJ_i)$ has an edge from $l$ to some $l' \in T$. Indeed, since $I \models comp(\Delta)$ it follows that $I \models \varphi_l$. If $l \in \mathcal{C}_{\mathrm{lits}}$, then every body literal is true, and since $L$ is a loop, at least one body literal belongs to $L$, and hence to $T$. If $l \in \mathcal{D}_{\mathrm{lits}}$, at least one body literal $l'$ is true, and since every literal in $\mathcal{D}^{\mathrm{ext}}(L)$ is false, $l'$ belongs to $L$, and hence to $T$. It follows that the graph $Dep(PJ_i)$ restricted to $T$ contains loops. Consider the strongly connected components of this graph and take a minimal one, say $L'$. Minimal means that no $l \in L'$ has an edge in this graph to a literal $l' \in T \setminus L'$.

We first show that, although $L'$ is a loop of $\Delta$, it cannot be a relevant loop. Indeed: assume towards contradiction that it is. Then $I \models LF_\Delta(L')$. Since $I \models \bigvee L'$ by construction of $L'$, we find $I \models \bigvee \mathcal{D}^{\mathrm{ext}}(L')$. However, all literals of $\mathcal{D}^{\mathrm{ext}}(L')$ are false in $I$ because either they belong to $\mathcal{D}^{\mathrm{ext}}(L)$, and in this case of the case analysis $\mathcal{D}^{\mathrm{ext}}(L)$ contains no true literals, or they belong to $L$, and can therefore not be true by construction of $L'$. Contradiction.

Hence, $L'$ has a justifying kernel, say $L''$, which is a non-empty negative loop consisting of negative literals such that $\mathcal{C}^{\mathrm{ext}}(L'') \cap L' = \emptyset$. It follows that $\mathcal{C}^{\mathrm{ext}}(L'')$ is a subset of $\mathcal{C}^{\mathrm{ext}}(L')$. By the construction of $L'$ as the minimal component, we also have $\mathcal{C}^{\mathrm{ext}}(L') \subseteq \mathcal{C}^{\mathrm{ext}}(L)$, hence we find $\mathcal{C}^{\mathrm{ext}}(L'') \subseteq \mathcal{C}^{\mathrm{ext}}(L)$.

Now, define $PJ_{i+1}$ by setting, for each $l \in \mathcal{D}_{\mathrm{lits}} \cap L''$, $PJ_{i+1}(l)$ to a literal in $\varphi_l \cap L''$ (this is possible, since $L''$ is a negative loop). Given that $I \models comp(\Delta)$ and $I \models \bigwedge L''$ (by construction), it is easy to see that $I(l) = I(PJ_{i+1}(l))$, for every $l \in L''$. Parts $a$ and $b$ of the induction hypothesis are clearly satisfied for $i+1$. To prove $c$, assume towards contradiction that $PJ_{i+1}$ contains a dangerous loop $L_1$ such that $L_1 \cap \mathrm{dom}(PJ_{i+1}) \neq \emptyset$. Similar to the first case of the case analysis, we find that $L_1 \cap \mathrm{dom}(PJ_{i+1}) \subseteq L''$ and therefore, $L_1 \subseteq L$. On the other hand, since $L_1$ is dangerous, it contains an atom $a$, and $a \notin L''$ because $L''$ is a negative loop. Since $L_1$ is a loop of $PJ_{i+1}$, there is a path in $L_1$ through $Dep(PJ_{i+1})$ from some negative literal $l \in L'' \cap L_1$ to $a$. Consider the first pair $(l_1, l_2)$ on this path such that $l_1 \in L''$ and $l_2 \notin L''$. By construction of $PJ_{i+1}$, if $l_1 \in \mathcal{D}_{\mathrm{lits}}$, then also $l_2 \in L''$, hence $l_1 \in \mathcal{C}_{\mathrm{lits}}$ and $l_2 \in \mathcal{C}^{\mathrm{ext}}(L'')$. Since $\mathcal{C}^{\mathrm{ext}}(L'') \subseteq \mathcal{C}^{\mathrm{ext}}(L)$, as we have seen, it follows that $l_2 \notin L$, which contradicts with $l_2 \in L_1 \subseteq L$. It follows that $c$ is satisfied.

In both cases we have shown how to extend the sequence with a partial justification $PJ_{i+1}$ in such a way that the induction hypotheses are fulfilled. This concludes the proof. □

*Proof.* (of Theorem 4.2) This is a direct consequence of Lemmas 4.3 and 4.4. □

**Example 4.24.** Consider $\Delta_{4.7} = \left\{ \; p \leftarrow a \vee \neg p \; \right\}$ (cf. Examples 4.7–4.8), and $\Delta_{4.24} = \left\{ \; p \leftarrow a \wedge \neg p \; \right\}$. We have $LF_{\Delta_{4.7}} = p \vee \neg p \supset a = a$, and $LF_{\Delta_{4.24}} = p \vee \neg p \supset \neg a = \neg a$. Hence we find the congruences $\Delta_{4.7} \cong \big((p \equiv a \vee \neg p) \wedge a\big)$ and $\Delta_{4.24} \cong \big((p \equiv a \wedge \neg p) \wedge \neg a\big)$.

The unique model of $\Delta_{4.7}$ is $\{p \mapsto \boldsymbol{t}, a \mapsto \boldsymbol{t}\}$; that of $\Delta_{4.24}$ is $\{p \mapsto \boldsymbol{f}, a \mapsto \boldsymbol{f}\}$. Both are non-total definitions.

*Remark* 4.1. Note that $\bigvee L \supset \bigvee \mathcal{D}^{\mathrm{ext}}(L)$ can be rewritten in CNF as follows:

$$\bigwedge_{l \in L} \left( \neg l \vee \bigvee \mathcal{D}^{\mathrm{ext}}(L) \right). \tag{4.3}$$

Also note that for any loop $L$, $\bigvee L \supset \bigvee \mathcal{D}^{\mathrm{ext}}(L)$ entails $\bigwedge L \supset \bigvee \mathcal{D}^{\mathrm{ext}}(L)$, because $L$ is non-empty.

### 4.4.2 Relation with ASP

We can partition the set of relevant loops $\mathcal{L}$ into positive loops $\mathcal{L}^+$ and mixed loops $\mathcal{L}^\pm$, and accordingly we can partition $LF_\Delta$ into $LF_\Delta^+ = \bigcup_{L \in \mathcal{L}^+} LF_\Delta(L)$ and $LF_\Delta^\pm = \bigcup_{L \in \mathcal{L}^\pm} LF_\Delta(L)$. We show that the loop formulas introduced by Lin and Zhao (2004) for ASP are equivalent to $LF_\Delta^+$.

We recall Lin and Zhao's results. Let $\Delta_{nlp}$ be a normal logic program, i.e., a set of rules with bodies that are conjunctions of literals. The *positive dependency graph* of $\Delta_{nlp}$ is the graph with vertices $\Sigma$ and edges $\{(p, q) \mid (p \leftarrow \bigwedge B) \in \Delta_{nlp}, q \in B\}$. Let $L$ be a loop in the positive dependency graph. Then define[3]

$$R^+(L, \Delta_{nlp}) = \left\{ \bigwedge B \,\middle|\, (p \leftarrow \bigwedge B) \in \Delta_{nlp}, p \in L, \exists q \; (q \in B \wedge q \in L) \right\}, \text{ and}$$
$$R^-(L, \Delta_{nlp}) = \left\{ \bigwedge B \,\middle|\, (p \leftarrow \bigwedge B) \in \Delta_{nlp}, p \in L, \neg\exists q \; (q \in B \wedge q \in L) \right\}.$$

Then the loop formula $LF_{\Delta_{nlp}}^{\text{Lin-Zhao}}(L)$ is the formula

$$\neg \bigvee R^-(L, \Delta_{nlp}) \supset \bigwedge \overline{L}. \tag{4.4}$$

If $\mathcal{L}^{\mathrm{LZ}}$ is the set of loops in the positive dependency graph, then $LF_{\Delta_{nlp}}^{\text{Lin-Zhao}}$ is defined as $\bigcup_{L \in \mathcal{L}^{\mathrm{LZ}}} LF_{\Delta_{nlp}}^{\text{Lin-Zhao}}(L)$, and $comp'(\Delta_{nlp})$ as $comp(\Delta_{nlp}) \cup \bigwedge \overline{Open(\Delta)}$. Then Lin and Zhao (2004) proved that an interpretation $I$ is a stable model of $\Delta_{nlp}$ iff it is a model of $comp'(\Delta_{nlp}) \cup LF_{\Delta_{nlp}}^{\text{Lin-Zhao}}$.

[3]Our presentation slightly differs from Lin and Zhao's, where $R^+$, $R^-$ were sets of rules rather then rule bodies.

Clearly the positive dependency graph is the restriction of the dependency graph to atoms; a loop in it is a positive—and therefore relevant—loop, i.e., $\mathcal{L}^{\mathrm{LZ}} = \mathcal{L}^{+}$. The difference between $comp'(\Delta_{nlp})$ and $comp(\Delta_{nlp})$ is the difference between stable model semantics, and stable model semantics extending the open symbols, as defined in Definition 4.10. Note that equation (4.4) can be equivalently written as $\bigvee L \supset \bigvee R^{-}(L, \Delta_{nlp})$. Hence, to relate $LF^{\text{Lin-Zhao}}_{\Delta_{nlp}}$ to $LF^{+}_{\Delta}$, we only have to relate $R^{-}(L, \Delta_{nlp})$ to $\mathcal{D}^{\mathrm{ext}}(L)$.

Consider the transformation from normal logic programming form to DefNF as presented in Section 4.2.2: it introduces a new atom and a new rule for each conjunctive body. Let $\Delta$ be the result of applying that transformation on $\Delta_{nlp}$. Then, for any positive loop $L$ in $\Delta_{nlp}$, each $(\bigwedge B) \in R^{+}(L, \Delta_{nlp})$ is represented in $\Delta$ by a new atom $n_i^{+}$, and likewise each $(\bigwedge B) \in R^{-}(L, \Delta_{nlp})$ by a new atom $n_i^{-}$. Let $N^{+}$ be the set of new atoms $n_i^{+}$, and $N^{-}$ the set of new atoms $n_i^{-}$. Then $L' = L \cup N^{+}$ is a positive loop in $\Delta$, and $\mathcal{D}^{\mathrm{ext}}(L') = N^{-}$. In other words, had Lin and Zhao produced their results for DefNF instead of for normal logic programs, their loop formulas would have been identical to $LF^{+}_{\Delta}$.

We easily derive from Lin and Zhao's proof also the following result (recall Definition 4.10 of $\models_{st}$):

**Proposition 4.11.** *Let $\Delta$ be a DefNF definition, $I$ a $\Sigma$-interpretation. Then $I \models_{st} \Delta$ iff $I \models comp(\Delta) \cup LF^{+}_{\Delta}$.*

### 4.4.3 Corollaries and observations

Some interesting questions and observations arise from the above theoretical results.

Firstly, we already had a link between models of total definitions and stable models (cf. Corollary 4.1). Proposition 4.11 offers a new link between stable models and the positive part of the loop formulas; this raises the question whether we can express the concept of totality using the partitioning of relevant loops in positive and mixed loops.

Secondly, since strongly connected components of a graph correspond to the graph's maximal loops, we know that there are no relevant loops encompassing different strongly connected components. We can therefore split a definition into smaller definitions, according to the strongly connected components.

Thirdly, we illustrate the relations between unfounded sets and relevant loops.

Finally, some examples show that some loop formulas may be entailed by other loop formulas; the former can therefore be considered redundant. We try to establish a characterization of loops that are not redundant.

#### Totality

The following result now is an easy consequence:

**Proposition 4.12.** *Let $T$ be a PC(ID) theory. If $\Delta$ is total with respect to $T$, then $T \cup comp(\Delta) \cup LF^{+}_{\Delta} \models LF^{\pm}_{\Delta}$.*

*Proof.* Let $\Delta$ be total with respect to $T$. Let $I$ be an interpretation with $I \models T$. Then $I \models \Delta$ iff there exists a stable $\Delta$-witness for $I$ (by Corollary 4.1), iff $I \models_{st} \Delta$ (by Proposition 4.7), iff $I \models comp(\Delta) \cup LF^{+}_{\Delta}$ (by Proposition 4.11). Hence, any model of $T \cup comp(\Delta) \cup LF^{+}_{\Delta}$ is a model of $\Delta$, and therefore (by Theorem 4.2) of $comp(\Delta) \cup LF^{+}_{\Delta} \cup LF^{\pm}_{\Delta}$. Hence, $T \cup comp(\Delta) \cup LF^{+}_{\Delta}$ entails $LF^{\pm}_{\Delta}$. □

This result offers us a means of establishing non-totality of a definition in a way independent of the definition of the well-founded semantics. It may also have consequences for SAT(ID) solving; it helps to determine what types of propagation to consider. More on this in Chapter 5.

**Strongly connected components**

We investigate whether strongly connected components offer a way to subdivide a definition in several smaller definitions. Let $G$ be the dependency graph of a DefNF definition $\Delta$, restricted to $\widehat{Def(\Delta)}$. For a literal $l$, denote by $\underline{l}$ the atom in $l$; for a set of literals $S$, denote by $\underline{S}$ the set $\{\underline{l} \mid l \in S\}$. We define $\mathcal{SCC}_{\Delta} = \{\underline{S} \mid S \in \mathcal{SCC}^{+}_{G}\}$. Observe that for any $S \in \mathcal{SCC}^{+}_{G}$, also $\overline{S} \in \mathcal{SCC}^{+}_{G}$, since the dependency graph is symmetric with respect to negation; $S$ and $\overline{S}$ are represented by the same set of atoms in $\mathcal{SCC}_{\Delta}$. Since $G$ is restricted to $\widehat{Def(\Delta)}$, singleton sets $\{l\}$ for atoms $l \in Open(\Delta)$ are not represented.

Let $\Delta$ be a definition, and $S \subseteq Def(\Delta)$. Then by $\Delta|_S$ we denote the definition $\{(p \leftarrow \varphi_p) \in \Delta \mid p \in S\}$.

**Proposition 4.13.** *Let $\Delta$ be a DefNF definition, $T$ the PC(ID) theory $\{\Delta|_S \mid S \in \mathcal{SCC}_{\Delta}\}$. Then $\Delta \cong T$.*

*Proof.* The set of loops in $\Delta$ is exactly the union of the sets of loops of each $\Delta|_S$; the set of rules in $\Delta$ is exactly the union of the rules of each $\Delta|_S$. Hence this result follows easily from the loop formula equivalence result, Theorem 4.2. □

This proposition has direct relevance for SAT(ID) solving. In order to have a simple format for PC(ID) theories—to enable easy parsing and simple data structures—we not only want definitions to be in DefNF, but also we want there to be only one definition. This can be obtained using the transformation for merging definitions described in Section 7.2.1. However, an undesired effect might be that there is a bigger search space for the SAT(ID) problem with one big definition than for the equivalent original SAT(ID) problem with many smaller separate definitions. Fortunately this proposition tells us that this is no real concern, because we can again subdivide the search space by simply calculating strongly connected components, and in effect treat each strongly connected component as a separate definition. Note that these separate definitions are at most the size of the original definitions, and possibly smaller.

**Example 4.25.** Let $\Delta = \left\{ \begin{array}{ll} p \leftarrow p, & q \leftarrow q \end{array} \right\}$. The theory obtained by subdividing $\Delta$ is $\left\{ \left\{ \begin{array}{l} p \leftarrow p \end{array} \right\}, \left\{ \begin{array}{l} q \leftarrow q \end{array} \right\} \right\}$, consisting of smaller definitions.

**Unfounded sets**

The concept of *unfounded set* is central to the definition of a well-founded sequence, Definition 2.1, and therefore to the semantics of inductive definitions. Intuitively, an unfounded set is a set of atoms that, when assumed to be all false, are confirmed by their rules in the definition to be false. Since this is a form of circular reasoning, we might expect a strong relation between unfounded sets and relevant loops. The following example, however, shows that the concept of unfounded set is too general for a strong relation to be possible; Propositions 4.14 and 4.15 show our best results relating the two concepts.

**Example 4.26.** Let $\Delta_{4.26} =$

$$\left\{ \begin{array}{ll} p \leftarrow a, & s \leftarrow s \vee t, \\ q \leftarrow q \wedge b, & t \leftarrow u, \\ r \leftarrow t \wedge c, & u \leftarrow t \vee a \end{array} \right\},$$

and let $I$ be the three-valued interpretation with $I(a) = \boldsymbol{f}$, and $I$ unknown in all other atoms. Then $U = \{p, q, r, s, t, u\}$ is an unfounded set with respect to $I$. Indeed, for each $l \in U$, $I[U/\boldsymbol{f}](\varphi_l) = \boldsymbol{f}$.

$\Delta_{4.26}$ contains three relevant loops: $\{q\}$, $\{s\}$ and $\{t, u\}$. Both $\{q\}$ and $\{t, u\}$ are themselves unfounded sets with respect to $I$; $\{s\}$ is not, because it depends disjunctively on $t$. Also $\{p\}$ is on its own an unfounded set with respect to $I$; however, it is not a loop of $\Delta_{4.26}$.

Observe furthermore that the dependency graph of $\Delta_{4.26}$ contains subgraphs that are unconnected, but that nevertheless each have an atom in $U$ (namely, the subgraphs with vertices $\{p\}$, $\{q\}$ and $\{r, s, t, u\}$).

**Proposition 4.14.** *Let $\Delta$ be a DefNF definition, $I$ a three-valued interpretation, and $U$ an unfounded set in $\Delta$ with respect to $I$. Let $G$ be the dependency graph of $\Delta$, restricted to $U$, and let $U'$ be a minimal component in $\mathcal{SCC}^+_G$. Then*
*a) either $U'$ is a relevant loop in $\Delta$ and $I(\bigvee \mathcal{D}^{ext}(U')) = \boldsymbol{f}$, or $U'$ is a singleton $\{l\}$, and $I(\varphi_l) = \boldsymbol{f}$; and*
*b) if the set $U \setminus U'$ is non-empty, it is an unfounded set with respect to $I[U'/\boldsymbol{f}]$.*

This result concerns a *minimal* component in $\mathcal{SCC}^+_G$, but indirectly sheds a light on the structure of $G$ as a whole, through the recursive result of item *b*.

**Example 4.27.** Example 4.26 continued. Both $\{p\}$, $\{q\}$ and $\{t, u\}$ are minimal components in $\mathcal{SCC}^+_G$, where $G$ is the restriction of $\Delta_{4.26}$'s dependency graph to $U$. Consider $U' = \{t, u\}$: it is a relevant loop of $\Delta_{4.26}$, and $I(\bigvee \mathcal{D}^{\text{ext}}(U')) = I(a) = \boldsymbol{f}$. The set $U \setminus U' = \{p, q, r, s\}$ is unfounded with respect to $I[U'/\boldsymbol{f}]$.

*Proof.* If $U'$ is a singleton $\{l\}$, where $l$ does not occur in a loop of $G$, then it follows from the minimality of $U'$ that $\varphi_l \cap U = \emptyset$, and since $U$ is an unfounded set with respect to $I$, it follows from $I[U/\boldsymbol{f}](\varphi_l) = \boldsymbol{f}$ that $I(\varphi_l) = \boldsymbol{f}$. Else, $U'$ is a minimal component in $\mathcal{SCC}_G$. Since $U'$ consists of atoms, it is a relevant loop of $\Delta$. By minimality, we have that $\mathcal{D}^{\text{ext}}(U') \cap U = \emptyset$. Choose an arbitrary $d \in \mathcal{D}^{\text{ext}}(U')$, and let $l \in U' \cap \mathcal{D}_{\text{lits}}$ be the atom with $d \in \varphi_l$. Because $I[U/\boldsymbol{f}](\varphi_l) = \boldsymbol{f}$

and $\varphi_l$ is a disjunction, $I(d) = \boldsymbol{f}$. Hence $I(\bigvee \mathcal{D}^{\text{ext}}(U')) = \boldsymbol{f}$ follows, thereby proving item *a*.

Item *b* follows easily from the definition of unfounded set. □

The following proposition simply confirms the intuitions that we have used to build the concept of relevant loop and of its loop formula.

**Proposition 4.15.** *Let $\Delta$ be a DefNF definition, $I$ a three-valued interpretation. Let $L \subseteq Def(\Delta)$ be a positive (and hence relevant) loop of $\Delta$ with $I(\bigvee \mathcal{D}^{ext}(L)) = \boldsymbol{f}$ and $I(l) = \boldsymbol{u}$ for each $l \in L$. Then $L$ is an unfounded set in $\Delta$ with respect to $I$.*

**Example 4.28.** Example 4.26 continued. Both $\{p\}$ and $\{t, u\}$ are positive loops of $\Delta_{4.26}$, all of whose external disjuncts are false in $I$, and indeed both are unfounded sets with respect to $I$.

*Proof.* Consider arbitrary $l \in L \cap \mathcal{D}_{\text{lits}}$. Then for each $l' \in \varphi_l$, either $l' \in L$ and therefore $I[L/\boldsymbol{f}](l') = \boldsymbol{f}$, or $l' \in \mathcal{D}^{\text{ext}}(L)$, hence from $I(\bigvee \mathcal{D}^{\text{ext}}(L)) = \boldsymbol{f}$ also $I[L/\boldsymbol{f}](l') = \boldsymbol{f}$ follows. Therefore we have $I[L/\boldsymbol{f}](\varphi_l) = \boldsymbol{f}$. For arbitrary $l \in L \cap \mathcal{C}_{\text{lits}}$, from the fact that $L$ is a loop it follows that $\varphi_l$ contains an $l' \in L$. Since $I[L/\boldsymbol{f}](l') = \boldsymbol{f}$, $I[L/\boldsymbol{f}](\varphi_l) = \boldsymbol{f}$ follows. □

### Redundant loops

Gebser and Schaub (2005); Gebser et al. (2006) defined the notion of "elementary loops" for ASP, a subset $\Psi$ of $LF^{\text{Lin-Zhao}}_{\Delta_{nlp}}$ such that $comp(\Delta_{nlp}) \cup \Psi \models LF^{\text{Lin-Zhao}}_{\Delta_{nlp}}$. We derive from their work similar results for our context.

The basic intuition is that some relevant loops may be redundant; we want to have criteria to determine which ones. We look at some examples.

**Example 4.29.** Let $\Delta_{4.29} = \left\{\ p \leftarrow p \wedge q, \quad q \leftarrow p\ \right\}$. Its relevant loops are $\{p\}$ and $\{p, q\}$; both have no external disjuncts. Clearly we have that $comp(\Delta_{4.29}) \cup LF_{\Delta_{4.29}}(\{p\}) \models LF_{\Delta_{4.29}}(\{p, q\})$; in words: $\{p, q\}$ is a redundant loop.

It may seem that the reason for the redundancy of $\{p, q\}$ in previous example is that it has a strict subset which is also a relevant loop. However, the following example shows that this condition is not sufficient.

**Example 4.30.** Let $\Delta_{4.30} = \left\{\ p \leftarrow q \vee r, \quad q \leftarrow p \vee r, \quad r \leftarrow p \vee q\ \right\}$. The relevant loops of $\Delta_{4.30}$ are $\{p, q\}$, $\{p, r\}$, $\{q, r\}$, and $\{p, q, r\}$. Despite the fact that each of the three former loops is a subset of the latter, we have $comp(\Delta_{4.30}) \cup LF_{\Delta}(\{p, q\}) \cup LF_{\Delta}(\{p, r\}) \cup LF_{\Delta}(\{q, r\}) \not\models LF_{\Delta}(\{p, q, r\})$.

Observe the difference between Examples 4.29 and 4.30: in the former, the subloop $L' = \{p\}$ of the redundant loop $L = \{p, q\}$ has $\mathcal{D}^{\text{ext}}(L') \cap L = \emptyset$, while in the latter, each of the subloops $L'$ of $L = \{p, q, r\}$ has $\mathcal{D}^{\text{ext}}(L') \cap L \neq \emptyset$. We refine this intuition.

Let $L$ be a loop of $\Delta$ that does not contain any negative subloop. Then $L$ is relevant, and any subset of $L$ that is a loop is relevant as well. Suppose

$L$ contains a subloop $L' \subsetneq L$ with $\mathcal{D}^{\text{ext}}(L') \cap L = \emptyset$. We show that then $L$ is redundant.

Formally, we prove the following:

**Proposition 4.16.** *Let $L$ be a loop of $\Delta$ that contains a subloop $L' \subsetneq L$ with $\mathcal{D}^{ext}(L') \cap L = \emptyset$. Then $comp(\Delta) \cup \bigcup_{L'' \in \mathcal{L}} LF_\Delta(L'') \models LF_\Delta(L)$, where $\mathcal{L} = \{L'' \subsetneq L \mid L'' \text{ is a loop}\}$.*

*Proof.* Denote $\Psi = comp(\Delta) \cup \bigcup_{L'' \in \mathcal{L}} LF_\Delta(L'')$. We construct a sequence $\langle S_i \rangle_{0 \leq i \leq N}$ of subsets of $L$, and prove the following by induction for each $i$: *a*) if $i > 0$, then $S_i \supsetneq S_{i-1}$, and *b*) $\Psi \models \bigwedge_{l \in S_i} \big(l \supset \bigvee \mathcal{D}^{\text{ext}}(L)\big)$. Since the sequence is strictly growing and limited by $L$, it is finite and ends with $S_N = L$; from the induction hypothesis it then follows that $\bigwedge_{l \in L} \big(l \supset \bigvee \mathcal{D}^{\text{ext}}(L)\big) = LF_\Delta(L)$ is entailed by $\Psi$.

Let $S_0 = L'$. Since $\mathcal{D}^{\text{ext}}(L') \cap L = \emptyset$, we have $\mathcal{D}^{\text{ext}}(L') \subseteq \mathcal{D}^{\text{ext}}(L)$. Therefore $LF_\Delta(L') \models \bigwedge_{l \in L'} \big(l \supset \bigvee \mathcal{D}^{\text{ext}}(L)\big)$, which proves the induction hypothesis for $i = 0$.

Assume the induction hypothesis holds for $i$, and $S_i \subsetneq L$ (otherwise we are done). Because $L$ is a loop, $L \setminus S_i$ contains an $l$ with $\varphi_l \cap S_i \neq \emptyset$. By the induction hypothesis, for any $l' \in \varphi_l \cap S_i$ we have that $\Psi \models l' \supset \bigvee \mathcal{D}^{\text{ext}}(L)$. If $l \in \mathcal{C}_{\text{lits}}$, then $comp(\Delta) \models l \supset l'$; modus ponens on $l \supset l'$ and $l' \supset \bigvee \mathcal{D}^{\text{ext}}(L)$ yields $l \supset \bigvee \mathcal{D}^{\text{ext}}(L)$. If there is no such $l \in \mathcal{C}_{\text{lits}}$, then $l \in \mathcal{D}_{\text{lits}}$, and $comp(\Delta) \models l \supset \bigvee \varphi_l$. Distinguish the following cases:

- $\varphi_l \cap L \subseteq S_i$. Modus ponens on $l \supset \bigvee \varphi_l$ and $l' \supset \bigvee \mathcal{D}^{\text{ext}}(L)$ for every $l' \in \varphi_l \cap S_i$ then yields that $\Psi \models l \supset \bigvee \mathcal{D}^{\text{ext}}(L)$ (since literals $l' \in \varphi_l \setminus L$ are already member of $\mathcal{D}^{\text{ext}}(L)$).

- If there is no such $l$ as in the previous case, then $l$ is part of a loop $L'' \subsetneq (L \setminus S_i)$ with $\mathcal{D}^{\text{ext}}(L'') \cap L \subseteq S_i$. Then resolve $l \supset \bigvee \varphi_l$ with $l' \supset \bigvee \mathcal{D}^{\text{ext}}(L'')$ for each $l' \in \varphi_l \cap L''$. In the resulting formula $l \supset \bigvee X$, $X$ is a subset of $S_i \cup \mathcal{D}^{\text{ext}}(L)$; continued resolution with $l' \supset \bigvee \mathcal{D}^{\text{ext}}(L)$ for each $l' \in X \cap S_i$ yields also in this case $\Psi \models l \supset \bigvee \mathcal{D}^{\text{ext}}(L)$.

Hence, with $S_{i+1} = S_i \cup \{l\}$, the induction hypothesis holds for $i + 1$. □

An *elementary loop* is any relevant loop that is not redundant; i.e. a relevant loop that contains a negative subloop, or a loop that contains neither a negative subloop, nor any strict subloop that has no external disjuncts in the bigger loop. Denote by $\mathcal{E}$ the set of elementary loops in $\Delta$, and by $LF_\Delta^{\mathcal{E}}$ the set of loop formulas $\bigcup_{L \in \mathcal{E}} LF_\Delta(L)$. From the above, we derive the following result:

**Corollary 4.2.** $\Delta \cong comp(\Delta) \cup LF_\Delta^{\mathcal{E}}$.

### 4.4.4 Related work

There is a substantial amount of work on loop formulas for ASP, starting with (Lin and Zhao, 2004). We mention some important ones. Erdem and Lifschitz

(2003) defined the notion of a *tight* Stable logic program $\Delta$ as one for which $\Delta \cong comp(\Delta)$, i.e., no loop formulas are needed. Gebser and Schaub (2005); Gebser et al. (2006) introduced elementary loops for ASP. Lin and Zhao (2004) and Giunchiglia et al. (2004) use loop formulas to build SAT-based ASP solvers: respectively ASSAT and CMODELS. Lee (2005) investigates theoretical properties of loops and loop formulas for ASP; he generalized, amongst others, the formula $LF^{\text{Lin-Zhao}}_{\Delta_{nlp}}(L)$ to arbitrary sets of literals $L$ (not necessarily loops), and showed that still $\Delta_{nlp} \models LF^{\text{Lin-Zhao}}_{\Delta_{nlp}}(L)$ holds. In our setting this property does not hold. Lifschitz and Razborov (2006) show why it is inevitable that the number of loop formulas is, in the worst case, exponential in the theory size. Ferraris et al. (2006) generalize loop formulas to stable model semantics for disjunctive logic programs with nested expressions.

This work is the first to formulate loop formulas for inductive definitions (or for the well-founded semantics). As such, it is also the first vocabulary-preserving reduction of (DefNF) definitions to PC. Pelov and Ternovska (2005) did define a reduction of inductive definitions to propositional logic. Their reduction, however, introduces new symbols to the vocabulary, as well as new formulas based on this new vocabulary to the theory. Whereas our loop formulas can be used (implicitly or explicitly) individually to derive ID-based propagations, the new formulas of Pelov and Ternovska's reduction cannot easily be used for the same purpose, because of the new symbols: these are only meaningful when all new formulas containing them are being used. However, their reduction is polynomial in size, and therefore has very different potential uses then our reduction.

The reduction by Pelov and Ternovska is based on the well-founded operator as defined by Van Gelder et al. (1991): a *level* can be assigned to every defined atom, namely, an integer that denotes the number of times the well-founded operator has to be applied before the atom gets a two-valued truth value. It introduces for every defined atom a number of new atoms that together denote this level. Then the property that the levels are "well behaved" according to the well-founded operator can be encoded in propositional logic, using these new atoms. A significant part of this reduction consists of an encoding of arithmetic properties in propositional logic; it is interesting to note that this could be avoided if a SAT Modulo the theory of arithmetic solver were used. The relation of Pelov and Ternovska's reduction to our loop formulas is unclear.

## 4.5 Conclusions

In this chapter, we have introduced justifications and loop formulas for propositional inductive definitions. We have established that the following statements are equivalent, for a DefNF definition $\Delta$ and a $\Sigma$-interpretation $I$:

1. $I \models \Delta$;

2. there exists a $\Delta$-witness for $I$;

3. $I \models comp(\Delta) \cup LF_\Delta$. This is the first vocabulary-preserving PC characterization of PC(ID) theories.

We have defined several related concepts that will be of further help for constructing SAT(ID) algorithms in Chapter 5:

- a witness for three-valued interpretations;
- the loop-simplification of a definition;
- a stable witness;
- positive versus mixed relevant loops;
- strongly connected components;
- elementary loops.

# Chapter 5

# SAT(ID) algorithms

## 5.1 Introduction

In this chapter, we develop various algorithms that can be used to implement a SAT(ID) solver. Also, we propose various *strategies* for SAT(ID) solving, i.e., various sorts of algorithmic behaviour. We further discuss an implementation of a SAT(ID) solver, MiniSat(ID), and evaluate its behaviour and its performance.

We present our algorithms in a framework that was proposed by Nieuwenhuis et al. (2006) in the context of SAT Modulo Theories (SMT). In the domain of SMT, one studies the satisfiability problem of ground first-order formulas with respect to some given background theory, such as the theory of equality, the theory of the integers, and so on. Different approaches for this problem exist; they vary in how they combine SAT solving of the given ground first-order formula (interpreted as a PC theory) with "theory solving" of the background theory. Nieuwenhuis et al. used their framework to model several of these approaches and describe their properties. The framework can be used to describe the behaviour of model generation algorithms for various extensions of PC, including PC(ID); it separates concerns of correctness and completeness from concerns of how to implement certain propagations.

The framework uses "transition rules" to specify propagations of model generation algorithms. In Section 5.3, we use the two semantical characterizations of PC(ID) from the previous chapter—justification semantics and loop formula semantics—to propose specific transition rules. We discuss different sets of such rules and their resulting behaviour. The focus of this section is primarily in establishing a wide range of possible or conceivable SAT(ID) strategies. For instance, we will introduce the so-called $\Delta$-Propagate transition rule, which derives an arbitrary consequence of the given definition and three-valued interpretation.

In Section 5.4 we then develop algorithms to implement these transition rules. We also give an in-depth discussion of the most important of these algorithms (which is used extensively in MiniSat(ID)) in Section 5.5.

In Section 5.6 we discuss some SAT solving techniques that are used in contemporary state-of-the-art SAT solvers; these techniques can be incorporated in SAT(ID) solvers as well.

We implemented the SAT(ID) solver MiniSat(ID), which we present in Section 5.7. MiniSat(ID) is built as an extension of the SAT solver MiniSat (Eén and Sörensson, 2003). We motivate the algorithmic choices made in MiniSat(ID), discuss its properties and some implementation details, and evaluate its performance. Finally, in Section 5.8 we conclude and discuss related and future work.

## 5.2 Preliminaries

First, we give a brief introduction to SMT in Section 5.2.1, and then present the framework by Nieuwenhuis et al. in Section 5.2.2. In Section 5.2.3 we use this framework to present various sorts of algorithmic behaviour for SMT. Finally, in Section 5.2.4 we discuss a normal form for PC(ID) theories which we will use throughout the rest of the text.

### 5.2.1 SAT Modulo Theories (SMT)

SAT Modulo Theories (SMT) is the problem of deciding whether a given ground first-order formula is satisfiable with respect to some given background theory, such as the theory of equality, the theory of the integers, and so on. The background theory $T$ is potentially a first-order theory. We denote by SMT($T$) the satisfiability problem of ground first-order formulas augmented with $T$.

A ground (i.e. variable-free) first-order formula $\Psi$ is then said to be *$T$-satisfiable* or *$T$-consistent* if $\Psi \wedge T$ is satisfiable. Entailment in $T$, for formulas $\Psi_1$, $\Psi_2$, is denoted $\Psi_1 \models_T \Psi_2$, and means that $\Psi_1 \wedge \neg\Psi_2$ is $T$-inconsistent. A *theory lemma* of $T$ is a propositional clause $\varphi$ such that $\emptyset \models_T \varphi$. A decision procedure for $T$-satisfiability is called a $T$-solver.

**Example 5.1.** Let $T$ be the following theory over the vocabulary with function symbols $f/1$, $g/1$, $h/1$ and the predicate symbol $=/2$:

$$\forall x, y\ x = y \supset f(x) = f(y), \tag{5.1}$$
$$\forall x, y\ x = y \supset g(x) = g(y), \tag{5.2}$$
$$\forall x, y\ x = y \supset h(x) = h(y). \tag{5.3}$$

(This is part of the theory of *Equality with Uninterpreted Functions (EUF)* (Burch and Dill, 1994) over that vocabulary. For simplicity, we have left out other axioms of that theory, which are irrelevant to this example.) Let $\Psi$ be the following ground theory over the same vocabulary, extended with an atom

$p$ and constants $a$, $b$, $c$:

$$a = b \ \vee \ f(a) \neq f(b), \tag{5.4}$$

$$g(a) = g(c) \ \vee \ p, \tag{5.5}$$

$$f(a) = f(b) \ \vee \ \neg p, \tag{5.6}$$

$$h(a) \neq h(b). \tag{5.7}$$

We have that $\Psi \models_T a \neq b$: indeed, (5.7) $\models_T a \neq b$ because $(5.7) \wedge (5.3) \wedge (a = b)$ is inconsistent. From $a \neq b$ and (5.4) we find $f(a) \neq f(b)$, from $f(a) \neq f(b)$ and (5.6) we find $\neg p$, and from $\neg p$ and (5.5) we find $g(a) = g(c)$ (all by normal entailment).

In the following section, we describe the framework by Nieuwenhuis et al. (2006), which can be used to model several approaches to solve SMT problems and their behavioural properties. In Section 5.2.3 we then outline a number of such approaches, and discuss how they can be modelled in the framework.

### 5.2.2 Transition systems

To introduce SAT(ID) algorithms and solving strategies, we use the framework of *transition systems* proposed by Nieuwenhuis et al. (2006). This framework offers a rule-based formalism in which one can easily express model generation algorithms; the authors used it to describe an abstract version of the well-known DPLL algorithm for SAT (Davis and Putnam, 1960; Davis et al., 1962), as well as different generic algorithms for SAT Modulo Theories (SMT). We will later present an instantiation of Nieuwenhuis et al.'s (2006) framework for SAT(ID).

We now introduce the basic concepts of the framework.

A *state sequence* $M$ is a pair consisting of a finite sequence of literals $M_s = \langle l_0 \, l_1 \, \ldots \, l_n \rangle$ such that for $i \neq j$, $l_i \neq l_j$ and $l_i \neq \neg l_j$, and a set $M_d \subseteq \{l_0, \ldots, l_n\}$ called $M$'s set of *decision literals*. In text, we often specify the decision literals by marking them as $l^{\mathsf{d}}$. Thus $\langle l_0^{\mathsf{d}} \, l_1 \, l_2^{\mathsf{d}} \rangle$ denotes the state sequence $(\langle l_0 \, l_1 \, l_2 \rangle, \{l_0, l_2\})$. A state sequence $M = (\langle M_s, M_d)$ determines a (three-valued) interpretation. We denote $M(l) = \boldsymbol{t}$ if $l \in M_s$, $M(l) = \boldsymbol{f}$ if $\neg l \in M_s$, and $M(l) = \boldsymbol{u}$ otherwise. Additionally, we denote by $\bigwedge M$ the conjunction $\bigwedge_{l_i \in M_s} l_i$.

The empty state sequence will be denoted as $\emptyset$. The concatenation of state sequences will be denoted by simple juxtaposition. For instance, if $l$ is a literal and $M, N$ denote the state sequences $(\langle l_0 \, l_1 \, \ldots \, l_m \rangle, M_d)$ and $(\langle l'_0 \, l'_1 \, \ldots \, l'_n \rangle, N_d)$, such that all $l_i$, $l'_j$ and $l$ are literals of different atoms, then $M \, l^{\mathsf{d}} \, N$ denotes the state sequence $(\langle l_0 \, l_1 \, \ldots \, l_m \, l \, l'_0 \, l'_1 \, \ldots \, l'_n \rangle, M_d \cup \{l\} \cup N_d)$.

A *state* is either the special state $FailState$, or a pair of the form $M \parallel F$, where $M$ is a state sequence and $F$ is a logic theory (throughout the chapter, a PC(ID) theory). We may write $M \parallel F, \varphi$ to denote $M \parallel F \cup \{\varphi\}$. A state is used as an abstract representation of an intermediate stage of some inference process, specifying which literals were made true, in what order they were made true, which were the decision literals and what theory $F$ is currently being worked on (note that this may also change throughout the execution).

A *transition relation* $\Longrightarrow$ is a binary relation over states. If $S \Longrightarrow S'$, we say that there is a *transition* (of $\Longrightarrow$) from $S$ to $S'$. A sequence $\langle S_0, \ldots, S_{n-1}, S_n \rangle$ such that $S_i \Longrightarrow S_{i+1}$ for $0 \leq i < n$ is called a *derivation* and is denoted $S_0 \Longrightarrow S_1 \Longrightarrow \ldots \Longrightarrow S_{n-1} \Longrightarrow S_n$. Given such a derivation, $S_n$ is called its current state and each $S_i$ $(i < n)$ one of its previous states. An *application* of a transition relation on a derivation is an extension $S_0 \Longrightarrow \ldots \Longrightarrow S_n \Longrightarrow S_{n+1}$. We denote by $\Longrightarrow^*$ the reflexive-transitive closure of $\Longrightarrow$.

A *transition rule* $\mathsf{R}$ is a rewrite operation on states, and is represented as follows:

$$S \qquad \Longrightarrow S' \qquad \textbf{if} \left\{ \Omega_{\mathsf{R}}[S, S'], \right.$$

where $S$, $S'$ are arbitrary states, and $\Omega_{\mathsf{R}}[S, S']$ denotes some conditions on them. Thus, a transition rule $\mathsf{R}$ specifies a transition relation, denoted $\stackrel{\mathsf{R}}{\Longrightarrow}$, where $S \stackrel{\mathsf{R}}{\Longrightarrow} S'$ if $\Omega_{\mathsf{R}}[S, S']$ is satisfied. A transition rule $\mathsf{R}$ is said to be *applicable* to a state $S$ if there exists a transition of $\stackrel{\mathsf{R}}{\Longrightarrow}$ from $S$. *An algorithm for* a given transition rule $\mathsf{R}$ is an algorithm that, given $S$, finds some $S'$ for which $\Omega_{\mathsf{R}}[S, S']$ holds.

A *transition system* $\mathcal{S}$ is a set of transition rules. Given a transition system $\mathcal{S}$, we define the transition relation $\stackrel{\mathcal{S}}{\Longrightarrow}$ as the union of the transition relations associated to its rules. I.e., $\stackrel{\mathcal{S}}{\Longrightarrow}$ is $\bigcup\{\stackrel{\mathsf{R}}{\Longrightarrow} \mid \mathsf{R} \in \mathcal{S}\}$. A state $S$ is *final* with respect to $\mathcal{S}$ if there is no $S'$ such that $S \stackrel{\mathcal{S}}{\Longrightarrow} S'$.

As an illustration, in Figure 5.1 we define the transition system $\mathcal{S}_{\text{DPLL}}$ consisting of five transition rules: $\{\mathsf{UnitProp}, \mathsf{PureLit}, \mathsf{Decide}, \mathsf{Fail}, \mathsf{Backtrack}\}$. In this figure, $\Psi$ is a CNF theory, $l$ a literal, and $M$, $N$ state sequences. This system can be used to model the classic DPLL algorithm (Davis and Putnam, 1960; Davis et al., 1962), as we will illustrate next.

It was shown by Nieuwenhuis et al. (2006) that

- this system terminates, i.e., there are no infinite derivations; and
- if $\emptyset \parallel \Psi \stackrel{\mathcal{S}_{\text{DPLL}}}{\Longrightarrow^*} S$, where $S$ is final with respect to $\mathcal{S}_{\text{DPLL}}$, then $S$ is $FailState$ iff $\Psi$ is unsatisfiable, and if $S$ is of the form $M \parallel \Psi'$, then $M$ is two-valued and $M \models \Psi$.

The $\mathsf{PureLit}$ rule is satisfiability-preserving, but not equivalence-preserving. For the system $\mathcal{S}_{\text{DPPL}-} = /\mathcal{S}_{\text{DPLL}} \setminus \{\mathsf{PureLit}\}$ also the following completeness result is easy to prove: for any model $M$ of $\Psi$, there is a derivation with final state $M$.

To model the DPLL algorithm, we restrict the applicability of the rules of $\mathcal{S}_{\text{DPLL}}$. In general, such restrictions are called a *strategy* of the given transition system. Naturally, a strategy could be described as a transition system itself (by incorporating the restrictions in the condition $\Omega_{\mathsf{R}}$ of the transition rules). It is often easier, however, to first define a general transition system, and then define

UnitProp:

$$M \parallel \Psi \quad \Longrightarrow M\, l \parallel \Psi \quad \textbf{if} \begin{cases} M(l) = \boldsymbol{u} \\ \text{for some clause } (\varphi \vee l) \in \Psi, M(\varphi) = \boldsymbol{f} \end{cases}$$

PureLit:

$$M \parallel \Psi \quad \Longrightarrow M\, l \parallel \Psi \quad \textbf{if} \begin{cases} M(l) = \boldsymbol{u} \\ \neg l \text{ occurs in no clause of } \Psi \end{cases}$$

Decide:

$$M \parallel \Psi \quad \Longrightarrow M\, l^{\mathsf{d}} \parallel \Psi \quad \textbf{if} \begin{cases} M(l) = \boldsymbol{u} \end{cases}$$

Fail:

$$M \parallel \Psi \quad \Longrightarrow FailState \quad \textbf{if} \begin{cases} \text{for some clause } \varphi \in \Psi, M(\varphi) = \boldsymbol{f} \\ M \text{ contains no decision literals} \end{cases}$$

Backtrack:

$$M\, l^{\mathsf{d}}\, N \parallel \Psi \quad \Longrightarrow M\, \neg l \parallel \Psi \quad \textbf{if} \begin{cases} \text{for some clause } \varphi \in \Psi, (M\, l^{\mathsf{d}}\, N)(\varphi) = \boldsymbol{f} \\ N \text{ contains no decision literals} \end{cases}$$

Figure 5.1: The $D_{\text{DPLL}}$ transition system

different strategies for that system. The DPLL algorithm can be described as the following strategy for the $\mathcal{S}_{\text{DPLL}}$ system:

- apply Backtrack and Fail when they are applicable; otherwise,
- apply UnitProp and PureLit when they are applicable; otherwise,
- apply Decide.

The strategy for the $\mathcal{S}_{\text{DPPL}-}$ system derived from this one by removing the PureLit rule is also called a DPLL strategy.

**Example 5.2.** Let $\Psi = p \vee q, \neg q \vee r, p \vee \neg r, \neg p \vee \neg q$. In the following derivation, we denote by, e.g., *UnitProp on* $p \vee q$, an application of the UnitProp rule, whereby the clause $(\varphi \vee l)$ in the description of UnitProp is instantiated by the clause $p \vee q$. Then the following is a valid DPLL derivation:

$$\emptyset \parallel \Psi \overset{\text{Decide}}{\Longrightarrow} \neg p^{\mathsf{d}} \parallel \Psi \overset{\text{UnitProp (on } p\vee q)}{\Longrightarrow} \neg p^{\mathsf{d}}\, q \parallel \Psi$$
$$\overset{\text{UnitProp (on } p\vee\neg r)}{\Longrightarrow} \neg p^{\mathsf{d}}\, q\, \neg r \parallel \Psi \overset{\text{Backtrack (on } \neg q\vee r)}{\Longrightarrow} p \parallel \Psi$$
$$\overset{\text{UnitProp (on } \neg p\vee\neg q)}{\Longrightarrow} p\, \neg q \parallel \Psi \overset{\text{Decide}}{\Longrightarrow} p\, \neg q\, r^{\mathsf{d}} \parallel \Psi$$

The last state of this derivation is final. Its state sequence $p\,\neg q\,r^{\mathsf{d}}$ corresponds to the two-valued interpretation $\{p \mapsto \boldsymbol{t}, q \mapsto \boldsymbol{f}, r \mapsto \boldsymbol{t}\}$, which is a model of $\Psi$.

One advantage of the transition system framework is that it carries with it a clear separation of concerns: the correctness and completeness of a given system, versus the algorithmic behaviour of a given strategy for the system, versus the algorithms to implement each of the transition rules.

### Generic backtracking rules

Observe that Backtrack and Fail can be seen as derivatives of UnitProp: if, by disregarding the condition $M(l) = \boldsymbol{u}$, the application of the UnitProp rule would lead to a conflicting state sequence, then either Backtrack or Fail is applicable. We use this observation to define a schema for generic Backtrack and Fail rules. In this schema, we will derive from a given transition rule that under certain conditions adds a literal $l$ to the state sequence, a transition rule that backtracks and one that fails when the same conditions hold, except that $l$ is false.

Let $\Omega_{\mathsf{R}}[M, l, \Psi]$ denote some conditions on $M$, $l$ and $\Psi$, and let $\mathsf{R}$ be a transition rule of the form

$$M \parallel \Psi \quad \Longrightarrow M\,l \parallel \Psi \quad \textbf{if} \begin{cases} M(l) = \boldsymbol{u} \\ \Omega_{\mathsf{R}}[M, l, \Psi], \end{cases}$$

then Backtrack(R) is the transition rule

$$M\,k^{\mathsf{d}}\,N \parallel \Psi \quad \Longrightarrow M\,\neg k \parallel \Psi \quad \textbf{if} \begin{cases} (M\,k^{\mathsf{d}}\,N)(l) = \boldsymbol{f} \\ \Omega_{\mathsf{R}}[M\,k^{\mathsf{d}}\,N, l, \Psi] \\ N \text{ contains no decision literals,} \end{cases}$$

and Fail(R) is the transition rule

$$M \parallel \Psi \quad \Longrightarrow \mathit{FailState} \quad \textbf{if} \begin{cases} M(l) = \boldsymbol{f} \\ \Omega_{\mathsf{R}}[M, l, \Psi] \\ M \text{ contains no decision literals.} \end{cases}$$

We denote by $[\mathsf{R}]^{\mathsf{BF}}$ the set of transition rules {Backtrack(R), Fail(R)}, and by $[\mathsf{R}]^{\mathsf{BF}+}$ the set of transition rules {R, Backtrack(R), Fail(R)}.

Note that this schema is generic in the same way algorithms that implement the transition rules of $[\mathsf{R}]^{\mathsf{BF}+}$ are: an algorithm that can be used to find, from a given state $M \parallel \Psi$, literals $l$ for which $M \parallel \Psi \overset{\mathsf{R}}{\Longrightarrow} M\,l \parallel \Psi$, can also be used to find applications of Backtrack(R) and Fail(R)—the main difficulty is in finding literals $l$ for which $\Omega_{\mathsf{R}}[M, l, \Psi]$ holds, rather then in deciding whether $M(l) = \boldsymbol{u}$.

### 5.2.3 SMT solving techniques

In this section, we will discuss several techniques that have been proposed to solve SMT problems. Many of these techniques have been applied in the context of ASP solvers as well. We will indicate how these techniques can be modelled as strategies of an appropriately chosen transition system.

Let $T$ be a given first-order theory (the background theory), and $\Psi$ a ground first-order theory. Let $\mathcal{S}_{\mathrm{SMT}}$ be the transition system obtained by extending the $\mathcal{S}_{\mathrm{DPPL-}}$ system with the following transition rules:

$T$-Propagate:

$$M \parallel \Psi \Longrightarrow M\, l \parallel \Psi \qquad \textbf{if} \begin{cases} M(l) = \boldsymbol{u} \\ \bigwedge M \models_T l \end{cases}$$

$[T\text{-Propagate}]^{\mathsf{BF}}$

$T$-Learn:

$$M \parallel \Psi \Longrightarrow M \parallel \Psi, \varphi \qquad \textbf{if} \begin{cases} \Psi \models_T \varphi \\ \varphi \notin \Psi \end{cases}$$

Restart:

$$M \parallel \Psi \Longrightarrow \emptyset \parallel \Psi \qquad \textbf{if} \begin{cases} - \end{cases}$$

Observe that $\mathcal{S}_{\mathrm{DPPL-}}$ contains the rules $[T\text{-Propagate}]^{\mathsf{BF}}$, i.e., Backtrack($T$-Propagate) and Fail($T$-Propagate). We omit the conditions that define these rules, since they follow the generic backtracking schema. We will do so throughout the rest of the text with other transition rules as well.

We now give a summary of various SMT($T$) solving techniques, and for most of them, illustrate how to model them as a strategy of the $\mathcal{S}_{\mathrm{SMT}}$ transition system. For a comprehensive discussion of their properties, such as correctness and termination, we refer to Nieuwenhuis et al. (2006).

**I. Eager techniques.** These techniques amount to translating (based on the properties of $T$) the given ground theory $\Psi$ to a propositional CNF formula $\Psi'$, and then applying a SAT solver on $\Psi'$. Depending on the purpose, a satisfiability-preserving or a model-preserving transformation should be used.

Though some such transformation approaches can be modelled as strategies of the $\mathcal{S}_{\mathrm{SMT}}$ system (using $T$-Learn), a general strategy is hard to define.

**II. Lazy techniques.** In these techniques, a $T$-solver is used to verify $T$-consistency of SAT models or interpretations. All of the following are lazy techniques.

**Naive approach.** Find a SAT model, and subsequently verify its $T$-consistency. If it is not, add a "theory lemma" that precludes this model to the theory, and restart.

This approach can be modelled in $\mathcal{S}_{\text{SMT}}$ as follows:

- apply the DPLL strategy of the $\mathcal{S}_{\text{DPPL}-}$ system while possible;
- otherwise (i.e. when a SAT model of $\Psi$ has been found), apply $T$-Learn if possible, immediately followed by Restart.

**Incremental approach.** Detect $T$-inconsistencies of state sequences as soon as they are generated, or at regular intervals. This technique is also called "early pruning".

This approach can be modelled in $\mathcal{S}_{\text{SMT}}$ as follows:

- when any of $[T\text{-Propagate}]^{\text{BF}}$ is applicable, then apply $T$-Learn for a theory lemma $\varphi$ such that $M(\neg\varphi) = \boldsymbol{t}$ (such a clause certainly exists when any of $[T\text{-Propagate}]^{\text{BF}}$ is applicable, e.g. a subclause of $\neg \bigwedge M$), and immediately follow this by Restart; otherwise,
- apply any rule of $\mathcal{S}_{\text{DPPL}-}$.

**On-line SAT.** When $T$-inconsistency of a state sequence is found, don't restart the SAT search from scratch, but backtrack only as far as needed to make the resulting state sequence $T$-consistent.

This approach can be modelled in $\mathcal{S}_{\text{SMT}}$ as follows:

- apply a rule of $[T\text{-Propagate}]^{\text{BF}}$ when applicable; otherwise,
- apply any rule of $\mathcal{S}_{\text{DPPL}-}$.

If a given state sequence $M$ is $T$-inconsistent, there also exists a theory lemma that is inconsistent in $M$. An often-used variant of the above strategy can therefore be modelled as follows:

- when a rule of $[T\text{-Propagate}]^{\text{BF}}$ is applicable, apply the $T$-Learn rule to derive a clause that is inconsistent in the current state sequence; otherwise,
- apply any rule of $\mathcal{S}_{\text{DPPL}-}$.

**Theory propagation.** This is also called "forward checking simplification". Instead of verifying $T$-consistency a posteriori, *guide* the SAT search a priori, by providing propagation based on $T$. Observe that this may still include backtracking based on $T$.

This approach can be modelled in $\mathcal{S}_{\text{SMT}}$ as follows:

- apply any rule of $\mathcal{S}_{\text{DPPL}-} \cup [T\text{-Propagate}]^{\text{BF}+}$.

**Exhaustive theory propagation.** Perform all possible theory propagations before applying Decide. This technique ensures that $T$-inconsistent states aren't ever reached.

This approach can be modelled in $\mathcal{S}_{\mathrm{SMT}}$ as follows:

- apply $T$-Propagate when possible; otherwise,
- apply any rule of $\mathcal{S}_{\mathrm{DPPL}-}$.

### 5.2.4 Normal form

In the previous chapter we used a normal form for definitions: a definition is in *DefNF* if each atom defined by it has exactly one rule defining it, and if each rule body is either a conjunction or a disjunction of literals. We now also define a normal form for PC(ID) theories. A PC(ID) theory that has at most one definition, which is in DefNF, and a propositional logic part, which is in conjunctive normal form (CNF), is said to be in *Extended CNF (ECNF)* normal form.[1] Throughout this chapter, we assume a propositional vocabulary $\Sigma$ and an ECNF theory consisting of a CNF formula $\Psi$ and a DefNF definition $\Delta$.

Since this chapter is concerned with *constructing* models, and therefore mostly with *three-valued* interpretations, we will from now on assume interpretations to be three-valued, unless we explicitly state otherwise. Accordingly, by support we mean three-valued support, and by a witness we mean a generalized witness.

We denote an interpretation $I$ either as a (total) function $I : \Sigma \to \{\boldsymbol{t}, \boldsymbol{f}, \boldsymbol{u}\}$, or as a partial function $I : \Sigma \to \{\boldsymbol{t}, \boldsymbol{f}\}$. The notation will be clear from the context. In the second case, we denote by $\mathrm{dom}(I)$ the set of atoms on which $I$ is defined. We say that interpretation $I'$ is an *extension* of $I$ if $\mathrm{dom}(I) \subsetneq \mathrm{dom}(I')$ and $I'|_{\mathrm{dom}(I)} = I$.

## 5.3 SAT(ID) transition rules

The SAT(ID) problem is the problem of deciding the satisfiability of a given ECNF theory $\Psi \cup \{\Delta\}$. Such a problem is viewed here as the SMT($\Delta$) satisfiability problem for $\Psi$. The derived entailment relation $\models_\Delta$ is therefore defined by $\Psi_1 \models_\Delta \Psi_2$ if $\Psi_1 \cup \{\Delta\} \models \Psi_2$. We define the relation $\models_\Delta$ also for interpretations $M$: $M \models_\Delta \Psi$ if $(\bigwedge M) \models_\Delta \Psi$. Note that $M \models_\Delta \Psi$ entails that $M' \models \Psi$ for each model $M'$ of $\Delta$ that extends $M$.

In this section we will propose different transition rules for the SAT(ID) problem, construct various transition systems out of these rules, and investigate their properties.

[1] Other extensions of CNF are added to ECNF in Chapter 6. See Appendix A for a description of the ASCII syntax for ECNF.

| $r$ | $d \leftarrow d_1 \vee \cdots \vee d_N$ | $c \leftarrow c_1 \wedge \cdots \wedge c_N$ |
|---|---|---|
| $B_1(r)$ | $d \vee \neg d_1$ | $\neg c \vee c_1$ |
| $\vdots$ | $\vdots$ | $\vdots$ |
| $B_N(r)$ | $d \vee \neg d_N$ | $\neg c \vee c_N$ |
| $C(r)$ | $\neg d \vee d_1 \vee \cdots \vee d_N$ | $c \vee \neg c_1 \vee \cdots \vee \neg c_N$ |

Figure 5.2: Clauses in the CNF representation of a definition's completion

We start by instantiating the $T$-Propagate and $T$-Learn rules for SAT(ID):

$\Delta$-Propagate:

$$M \parallel \Psi, \Delta \qquad \Longrightarrow M\, l \parallel \Psi, \Delta \qquad \textbf{if} \begin{cases} M(l) = \boldsymbol{u} \\ M \models_\Delta l \end{cases}$$

$\Delta$-Learn:

$$M \parallel \Psi, \Delta \qquad \Longrightarrow M \parallel \Psi, \varphi, \Delta \qquad \textbf{if} \begin{cases} \Psi, \Delta \models \varphi \\ \varphi \notin \Psi. \end{cases}$$

Observe that $\Delta$-Propagate can be considered to model the most general type of propagation of literals based on definitions possible: it derives any arbitrary literal that is a consequence of the current state and the given definition. Likewise, $\Delta$-Learn derives an arbitrary clause that is entailed by the given definition.

While these rules are obviously correct, their conditions do not yield insight into possible implementations. For this purpose we will define less general transition rules with more specific conditions in the following section. In Section 5.3.2 we will then build concrete transition systems from these rules and describe strategies for them. Section 5.4 will be concerned with developing algorithms that implement these transition rules and systems.

### 5.3.1 Specific transition rules

Recall the loop formula equivalence result of Theorem 4.2: for a DefNF definition $\Delta$, $\Delta \equiv comp(\Delta) \cup LF_\Delta$, where $comp(\Delta)$ is the conjunction of all equivalences obtained from rules of $\Delta$ by replacing the definitional implication $\leftarrow$ by $\equiv$, and $LF_\Delta$ is the set of loop formulas $LF_\Delta(L) = \bigvee L \supset \bigvee \mathcal{D}^{\text{ext}}(L)$, for all relevant loops $L$ in $\Delta$.

We represent both $comp(\Delta)$ and $LF_\Delta(L)$ in CNF. Figure 5.2 gives the set of clauses $\{C(r),\ B_1(r),\ \ldots,\ B_N(r)\}$ derived from disjunctive and conjunctive rules $r \in \Delta$; $LF_\Delta(L)$ can be represented as $\bigwedge_{l \in L} (\neg l \vee \bigvee \mathcal{D}^{\text{ext}}(L))$.

In Figure 5.3 we define some transition rules for propagation based on the definition $\Delta$: UnitPropDef, BwLoop and FwLoop. These rules are derived from the loop formula result: they achieve the effect that unit propagation (i.e., the

UnitPropDef:

$$M \parallel \Psi, \Delta \quad \Longrightarrow M\, l \parallel \Psi, \Delta \quad \textbf{if} \begin{cases} M(l) = \boldsymbol{u} \\ (\varphi \vee l) \in comp(\Delta) \text{ with } M(\varphi) = \boldsymbol{f} \end{cases}$$

$[\mathsf{UnitPropDef}]^{\mathsf{BF}}$

BwLoop:

$$M \parallel \Psi, \Delta \quad \Longrightarrow M\, \neg l \parallel \Psi, \Delta \quad \textbf{if} \begin{cases} M(l) = \boldsymbol{u} \\ \Delta \text{ contains a relevant loop } L \\ \text{such that } M(\bigvee \mathcal{D}^{\text{ext}}(L)) = \boldsymbol{f} \\ \text{and } l \in L \end{cases}$$

$[\mathsf{BwLoop}]^{\mathsf{BF}}$

FwLoop:

$$M \parallel \Psi, \Delta \quad \Longrightarrow M\, d \parallel \Psi, \Delta \quad \textbf{if} \begin{cases} M(d) = \boldsymbol{u} \\ \Delta \text{ contains a relevant loop } L \\ \text{such that } M(\bigvee L) = \boldsymbol{t}, \\ d \in \mathcal{D}^{\text{ext}}(L) \text{ and} \\ M(\bigvee \mathcal{D}^{\text{ext}}(L) \setminus \{d\}) = \boldsymbol{f} \end{cases}$$

$[\mathsf{FwLoop}]^{\mathsf{BF}}$

Figure 5.3: Transition rules for definition-based propagations

UnitProp transition rule) would have on the clauses of $comp(\Delta) \cup LF_\Delta$, but without the need to explicitly add these clauses to the theory. Observe that the transition relations $\overset{\mathsf{UnitPropDef}}{\Longrightarrow}$, $\overset{\mathsf{BwLoop}}{\Longrightarrow}$, and $\overset{\mathsf{FwLoop}}{\Longrightarrow}$ specified by these rules are subrelations of $\overset{\Delta\text{-}\mathsf{Propagate}}{\Longrightarrow}$.

**Example 5.3.** Let $\Delta =$

$$\left\{ \begin{array}{l} p \leftarrow q \wedge r, \\ q \leftarrow s \vee a, \\ r \leftarrow r \vee \neg a \vee b, \\ s \leftarrow s \wedge c \end{array} \right\},$$

and consider the state $p \parallel \Delta$. We give a derivation in the transition system {UnitPropDef, BwLoop, FwLoop}:

$$\begin{aligned} p \parallel \Delta \overset{\mathsf{UnitPropDef}}{\Longrightarrow} p\, q \parallel \Delta \overset{\mathsf{UnitPropDef}}{\Longrightarrow} p\, q\, r \parallel \Delta \overset{\mathsf{BwLoop}}{\Longrightarrow} p\, q\, r\, \neg s \parallel \Delta \\ \overset{\mathsf{UnitPropDef}}{\Longrightarrow} p\, q\, r\, \neg s\, a \parallel \Delta \overset{\mathsf{FwLoop}}{\Longrightarrow} p\, q\, r\, \neg s\, a\, b \parallel \Delta. \end{aligned}$$

The first two transitions derive respectively $q$ and $r$ from the completion of the rule $p \leftarrow q \wedge r$, using the clauses $\neg p \vee q$ and $\neg p \vee r$. The third transition derives $\neg s$ from the loop $\{s\}$: its loop formula is $s \supset \bot$. The fourth transition derives $a$ from the completion of the rule $q \leftarrow s \vee a$, using the clause $\neg q \vee s \vee a$. The final transition derives $b$ from the loop $\{r\}$: its loop formula is $r \supset \neg a \vee b$.

Recall Propositions 4.14 and 4.15 to see how the UnitPropDef and BwLoop rules relate to *unfounded sets*. Let $M$ be the current state sequence, and $U$ an unfounded set with respect to $M$, then for any $u \in U$, there is a derivation from the current state that derives $\neg u$ using only UnitPropDef and BwLoop. Vice versa, if $U$ is a positive loop on which BwLoop is applicable, then $U$ is an unfounded set with respect to $M$.

BwLoop and FwLoop are applicable on mixed relevant loops as well; thus, BwLoop could be seen as generalizing the concept of unfounded set to literals.

**Example 5.4.** Let $\Delta =$

$$\left\{ \begin{array}{l} p \leftarrow \neg q \vee a, \\ q \leftarrow \neg p \vee b \end{array} \right\},$$

and consider the state $\neg a \parallel \Delta$. $\Delta$ has two relevant loops, with loop formulas $p \vee \neg q \supset a$ and $\neg p \vee q \supset b$. BwLoop is applicable on the first one: $\neg a \parallel \Delta \overset{\mathsf{BwLoop}}{\Longrightarrow}{}^{*}$ $\neg a\, \neg p\, q \parallel \Delta$. Next, FwLoop is applicable on the second one: $\neg a\, \neg p\, q \parallel \Delta \overset{\mathsf{FwLoop}}{\Longrightarrow}$ $\neg a\, \neg p\, q\, b \parallel \Delta$.

Another important property of the proposed rules is that UnitPropDef, BwLoop and FwLoop together are still less general then $\Delta$-Propagate, as the following example illustrates.

**Example 5.5.** Let $\Delta =$

$$\left\{ \begin{array}{l} p \leftarrow p \vee q \vee b, \\ q \leftarrow q \vee b \end{array} \right\},$$

and consider the state $p \parallel \Delta$. The clauses of $comp(\Delta)$ that contain $p$ or $\neg p$ are $\neg p \vee p \vee q \vee b$, $p \vee \neg p$, $p \vee \neg q$, and $p \vee \neg b$. They are all already satisfied by $\{p \mapsto \boldsymbol{t}\}$; UnitPropDef is not applicable. The loop formulas for $\Delta$ are $p \supset q \vee b$ and $q \supset b$; neither BwLoop nor FwLoop is applicable on either of them. However, $\Delta$ has exactly one model extending $\{p \mapsto \boldsymbol{t}\}$, namely $\{p \mapsto \boldsymbol{t}, q \mapsto \boldsymbol{t}, b \mapsto \boldsymbol{t}\}$. Thus, both $q$ and $b$ can be derived using $\Delta$-Propagate.

As a consequence, the *exhaustive theory propagation* strategy cannot be implemented using only UnitPropDef, BwLoop and FwLoop. Then it cannot be avoided that $\Delta$-inconsistent states are sometimes reached,[2] hence there is a clear need for the backtracking and failure variants of these rules.

In Figure 5.4 we define transition rules for SAT(ID) that modify the theory. The transition rules AddComp and AddLF are also derived from the loop formula

[2] In the exhaustive theory propagation strategy no $\Delta$-inconsistent states are reached, since it gives the $\Delta$-Propagate rule the highest priority.

AddComp:

$$M \parallel \Psi, \Delta \quad \Longrightarrow M \parallel \Psi, \varphi, \Delta \qquad \textbf{if} \begin{cases} \varphi \in comp(\Delta) \\ \varphi \notin \Psi \end{cases}$$

AddLF:

$$M \parallel \Psi, \Delta \quad \Longrightarrow M \parallel \Psi, \Delta, LF \quad \textbf{if} \begin{cases} LF = \neg l \vee \bigvee \mathcal{D}^{\text{ext}}(L) \\ L \text{ is a relevant loop in } \Delta, l \in L \\ LF \notin \Psi \end{cases}$$

Simplify:

$$M \parallel \Psi, \Delta \quad \Longrightarrow M \parallel \Psi, \Delta^{\varnothing[M]} \quad \textbf{if} \begin{cases} \circlearrowleft[M] \text{ is consistent} \end{cases}$$

RemoveDef:

$$M \parallel \Psi, \Delta \quad \Longrightarrow M \parallel \Psi \qquad \textbf{if} \begin{cases} \Psi \models \Delta \end{cases}$$

Figure 5.4: Theory-modifying transition rules

equivalence result; the relations $\overset{\mathsf{AddComp}}{\Longrightarrow}$ and $\overset{\mathsf{AddLF}}{\Longrightarrow}$ are subrelations of $\overset{\Delta\text{-}\mathsf{Learn}}{\Longrightarrow}$. The transition rules Simplify and RemoveDef aim to simplify the given PC(ID) theory. We define the concepts $\circlearrowleft[M]$ and $\Delta^{\varnothing[M]}$ used in the conditions of the Simplify rule later in this section.

Though less general then Δ-Learn, the AddComp and AddLF rules clearly suffice to encode the eager technique, a reduction to SAT. The RemoveDef transition rule is, strictly speaking, redundant, but emphasizes that the purpose of some strategies may be to reduce a definition to SAT.

**Example 5.6.** Consider again $\Delta$ from Example 5.5. $comp(\Delta)$ consists of the clauses $\Psi_1 = \{p \vee \neg q,\ p \vee \neg b,\ q \vee \neg b\}$, plus some trivially satisfied clauses. $LF_\Delta$ consists of the clauses $\Psi_2 = \{\neg p \vee q \vee b,\ \neg q \vee b\}$. We have the following derivation in {AddComp, AddLF, RemoveDef}:

$$\emptyset \parallel \Delta \quad \overset{\mathsf{AddComp}^*}{\Longrightarrow} \quad \emptyset \parallel \Psi_1, \Delta \quad \overset{\mathsf{AddLF}^*}{\Longrightarrow} \quad \emptyset \parallel \Psi_1 \cup \Psi_2, \Delta \quad \overset{\mathsf{RemoveDef}}{\Longrightarrow} \quad \emptyset \parallel \Psi_1 \cup \Psi_2.$$

The final transition is valid because $\Psi_1 \cup \Psi_2 \models \Delta$, by the loop formula result. The theory in the final state is a PC theory, on which normal SAT solving techniques can be applied.

We now explain the Simplify rule. Recall Definition 4.9 of a loop-simplification. We defined the set of *unavoidably false* literals $\circlearrowleft$ as the set of literals $l$ such that any justification contains a non-negative loop reachable from $l$, and the loop-simplification $\Delta^{\varnothing}$ as the definition obtained from $\Delta$ by replacing the

rules that define unavoidably false literals $l$ by rules $l \leftarrow \bot$. If $\circlearrowright$ is inconsistent, $\Delta^{\emptyset}$ is defined as the inconsistent definition $\{p \leftarrow \neg p\}$ (for some arbitrary $p$).

The idea of these definitions is that $\Delta^{\emptyset}$ is a simpler definition than $\Delta$ itself, because many loops have been removed, yet $\Delta \equiv \Delta^{\emptyset}$ (cf. Proposition 4.5). Thus, computations on $\Delta^{\circlearrowright}$ may be easier than on $\Delta$.

We refine the concepts of $\circlearrowright$ and $\Delta^{\emptyset}$. Some literals may be consequences of the CNF theory $\Psi$; when these literals are already known, the set of unavoidably false literals may be larger than when they are not. We therefore redefine the concepts, now *with respect to an interpretation*. For an arbitrary interpretation $M$, we define $\circlearrowright[M]$ as the set of literals $\{l \in \widehat{Def(\Delta)} \mid$ for any justification $J$ for $\Delta$ *that 3-valuedly supports* $M$, $Sub(J,l)$ contains a non-negative loop$\}$. We also define $\Delta^{\emptyset[M]}$, derived from $\Delta$ and $\circlearrowright[M]$ in the same way as $\Delta^{\emptyset}$ is derived from $\Delta$ and $\circlearrowright$.

The Simplify transition rule replaces a definition $\Delta$, in a given state sequence $M$, by $\Delta^{\emptyset[M]}$, on the condition that $\circlearrowright[M]$ is consistent. This replacement is equivalence preserving in $M$. When $\Psi \cup \{\Delta\} \models \bigwedge M$, it is model preserving.

### 5.3.2 Concrete transition systems

We want to find a transition system for SAT(ID) with the following desirable properties: correct, complete, efficient, and efficiently implementable. Naturally, correctness and completeness are of primary importance. Correctness of the transition rules proposed in previous sections has already been established; in this section we propose transition systems that are complete, and discuss strategies for them. We will also compare their behaviour in terms of efficiency; for this purpose, we next introduce the notion of *decide-efficiency*.

#### Decide-efficiency

A common transition rule shared by all SAT(ID) transition systems presented here is Decide (but see Remark 5.1 below in this respect). An important goal of transition systems, or strategies for them, is efficiency in terms of the number of times Decide has to be applied. Let $A$ and $B$ be transition systems or strategies defined over the same set of states. We say $A$ is *more decide-efficient* than $B$, if there exists a state $S$ and state sequence $M$, such that to derive from $S$ some state which has $M$ as its state sequence, the minimal number of applications of the Decide rule (over all possible derivations) that $A$ needs is smaller than the same number of $B$. Under this definition some systems may be mutually more decide-efficient than each other: whereas $A$ may need fewer applications of Decide than $B$ to derive some state sequence $M_1$ from some state $S_1$, there may be another state $S_2$ and state sequence $M_2$ such that $B$ needs fewer applications of Decide than $A$ to derive $M_2$ from $S_2$. We say $A$ is *strictly* more decide-efficient than $B$ if $A$ is more decide-efficient than $B$, and $B$ is not more decide-efficient than $A$. We use the concept of decide-efficiency only for correct and complete SAT(ID) transition systems.

When comparing the behaviour of transition systems or strategies, other measures of efficiency have to be taken into account as well, for at least two reasons: first, making transition rules more general improves decide-efficiency, yet the complexity of algorithms that implement the rules may be worse, and second, in some systems it may be possible to derive certain partial interpretations using fewer applications of Decide, but at the cost of a much larger number of applications of some other rule.

*Remark* 5.1. A correct and complete transition system without a "decision" or "choice" transition rule is conceivable. It contains a rule for reducing a definition to SAT, and a resolution rule.

*Remark* 5.2. A particular strategy for SAT(ID) transition systems might be to derive truth values for defined literals *only* when these values follow from the semantics of the definition and from the current state sequence, i.e., only using rules such as UnitPropDef, BwLoop or FwLoop. In such a strategy, there are two restrictions: *i*) Decide can only be applied to literals in $\widehat{Open(\Delta)}$, not to literals in $\widehat{Def(\Delta)}$; *ii*) UnitProp (on clauses of $\Psi$) can only derive literals of $\widehat{Open(\Delta)}$, not literals of $\widehat{Def(\Delta)}$. One advantage of such a strategy would be that standard algorithms for finding the well-founded model of a definition could be applied (e.g. Van Gelder, 1993; Berman et al., 1995; Lonc and Truszczyński, 2000), in particular when a given interpretation is two-valued on $Open(\Delta)$. However, Järvisalo et al. (2005) investigated restrictions similar to (*i*) in the context of boolean circuits, and concluded that for some theories a proof of unsatisfiability is exponentially longer with than without the restriction; it is likely that their results carry over to PC(ID). Also, a strategy with restriction (*ii*) is clearly strictly less decide-efficient than one without. For these reasons we will not further consider this type of strategy.

### Reduction to SAT

**Definition 5.1** (*Reduce*)**.** We define the *Reduce* transition system as the system consisting of the rules {Decide, AddComp, AddLF, RemoveDef, Restart} $\cup$ $[\text{UnitProp}]^{\text{BF}+}$.

**Proposition 5.1.** Reduce *is complete.*

*Proof.* This follows from the loop formula equivalence result, Theorem 4.2, and the completeness of the $\mathcal{S}_{\text{DPPL}-}$ system. □

Observe that RemoveDef is not essential to the *Reduce* system—the system obtained from *Reduce* by removing this rule is still correct and complete. Naturally, the purpose of RemoveDef is to enable creating a purely propositional theory, so that SAT solvers are applicable to it. Also Restart is not essential; its purpose is to enable a strategy such as *Repetitive SAT*, discussed below.

Also observe that the Restart rule causes non-termination. Special strategies are required to ensure termination. We now discuss some strategies for *Reduce*.

**Reduction to SAT.** This corresponds to the eager technique for SMT. Start by applying AddComp and AddLF until $comp(\Delta) \cup LF_\Delta$ is completely added to $\Psi$. Then apply RemoveDef, and finish by applying the remaining rules as in the DPLL strategy of the $\mathcal{S}_{\text{DPPL}-}$ system. (Do not use Restart.)

Observe that this strategy ensures that when RemoveDef is applied, it is already known that $\Psi \models \Delta$. The problem of deciding whether $\Psi \models \Delta$ is, in general, as hard as the complement of the SAT(ID) problem (since $\Psi \not\models \Delta$ iff $\Delta \cup \neg\Psi$ is satisfiable).

**Repetitive SAT.** This technique is eager in that it adds $comp(\Delta)$ to the CNF theory initially; other than that it corresponds to the naive and on-line lazy techniques for SMT. (Step 1) Start by adding applying $comp(\Delta)$ (using AddComp), and (Step 2) then apply the $\mathcal{S}_{\text{DPPL}-}$ system rules until either a model is found, or $FailState$ is reached. In the former case, (Step 3) apply AddLF for some loop formulas $LF$ that are not satisfied by the model, if such a loop formula exists, and from there go back to Step 2 (either by applying Restart—the naive approach—or by using Backtrack—the on-line approach). If no such loop formula exists, a SAT(ID) model is found.

The naive repetitive SAT strategy corresponds—apart from the different definition of loop formula—to the strategy applied by the ASSAT (Lin and Zhao, 2004) and CMODELS (Lierler, 2005) ASP systems.

We have implemented the on-line repetitive SAT strategy as a possible execution strategy for our solver MiniSat(ID) (cf. Section 5.7).

To implement this strategy, an algorithm is required that can extract from a given definition and two-valued interpretation an unsatisfied loop formula.

The obvious drawback of this strategy is that it neglects important information of the definition until very late in the search, namely, when a SAT model of $\Psi \cup comp(\Delta)$ has been found. This may have the effect that a large fraction of the executed search is superfluous.

**Incremental.** Start by adding $comp(\Delta)$. Then apply the $\mathcal{S}_{\text{DPPL}-}$ system rules, but whenever there are loop formulas not satisfied in the current three-valued interpretation, add one such formula.

This strategy has clear advantages over the previous one: backtracking based on loop formulas is applied as soon as possible. It requires an algorithm that can extract from a given definition and *three-valued* interpretation a set of unsatisfied loop formulas; the disadvantage may be that this algorithm is too expensive to apply for every three-valued interpretation obtained along a derivation.

A further advantage that *Reduction to SAT* and *Repetitive SAT* have over this strategy is that the former two are entirely independent of the SAT solving algorithm. Thus, for these strategies one can make use of off-the-shelf SAT solvers. The *Incremental* strategy, on the other hand, requires a modification of a SAT solver's internal structure.

Adding loop formulas early, as in the case of *Reduction to SAT* and to a much lesser degree also of *Incremental*, has the effect of enabling extra propagations. In strategies where loop formulas are not added eagerly, the same propagations can be achieved only through the Decide rule. Hence we have the following efficiency order: *Reduction to SAT* is strictly more decide-efficient than *Incremental*, which in turn is strictly more decide-efficient than *Repetitive SAT*. However, if we compare efficiency in terms of how often they typically apply the AddLF rule,[3] we have *Incremental* > *Reduction to SAT* and *Repetitive SAT* > *Reduction to SAT*, the reason being that *Reduction to SAT* simply applies AddLF as often as possible.

**Using definition-based propagation techniques**

**Definition 5.2** (*BwLoop*, *FwLoop*, *Loop*)**.** We define the *BwLoop* transition system as $\{\mathsf{Decide}\} \cup [\mathsf{UnitProp}]^{\mathsf{BF+}} \cup [\mathsf{UnitPropDef}]^{\mathsf{BF+}} \cup [\mathsf{BwLoop}]^{\mathsf{BF+}}$, the *FwLoop* transition system as $\{\mathsf{Decide}\} \cup [\mathsf{UnitProp}]^{\mathsf{BF+}} \cup [\mathsf{UnitPropDef}]^{\mathsf{BF+}} \cup [\mathsf{FwLoop}]^{\mathsf{BF+}}$, and the *Loop* transition system as $BwLoop \cup FwLoop$.

**Proposition 5.2.** BwLoop*,* FwLoop *and* Loop *all terminate and are complete.*

*Proof.* Termination is proven just as for the $\mathcal{S}_{\text{DPPL}-}$ system. Completeness again follows from Theorem 4.2 and completeness of the $\mathcal{S}_{\text{DPPL}-}$ system. □

The following example shows that *BwLoop* and *FwLoop* are mutually more decide-efficient. As a consequence, *Loop* is strictly more decide-efficient than both *BwLoop* and *FwLoop*.

**Example 5.7.** Let $\Delta = \{\ p \leftarrow p \vee a\ \}$. It has one loop formula: $p \supset a$. In the state $p \parallel \Delta$, FwLoop is applicable, deriving $a$. To derive $a$ in *BwLoop* from that state, a possible derivation is

$$p \parallel \Delta \overset{\mathsf{Decide}}{\Longrightarrow} p \neg a^{\mathsf{d}} \parallel \Delta \overset{\mathsf{Backtrack(BwLoop)}}{\Longrightarrow} p\, a \parallel \Delta.$$

Conversely, in the state $\neg a \parallel \Delta$, BwLoop is applicable, deriving $\neg p$. To derive $\neg p$ in *FwLoop* from that state, a possible derivation is

$$\neg a \parallel \Delta \overset{\mathsf{Decide}}{\Longrightarrow} \neg a\, p^{\mathsf{d}} \parallel \Delta \overset{\mathsf{Backtrack(FwLoop)}}{\Longrightarrow} \neg a \neg p \parallel \Delta.$$

We now introduce different strategies for *(Bw/Fw)Loop* and compare them.

**Loop-based theory propagation.** This strategy corresponds to the *theory propagation* approach for SMT. We define the strategy for the *Loop* system.

- apply any rule of $[\mathsf{R}]^{\mathsf{BF}}$, for $\mathsf{R} \in \{\mathsf{UnitProp}, \mathsf{UnitPropDef}, \mathsf{BwLoop}, \mathsf{FwLoop}\}$ when possible; otherwise,

[3] In the worst case the three strategies require the same number of applications of the AddLF rule.

- apply any rule of {UnitProp, UnitPropDef, BwLoop, FwLoop} when possible; otherwise,
- apply Decide.

Similar strategies for *BwLoop* and *FwLoop* are derived from this one by removing respectively the $[\mathsf{FwLoop}]^{\mathsf{BF+}}$ or the $[\mathsf{BwLoop}]^{\mathsf{BF+}}$ rules.

**Full loop propagation.** This strategy resembles the previous one, but tries to postpone the transition rules based on relevant loops till the unit propagation rules are exhausted:

- apply any rule of $[\mathsf{UnitProp}]^{\mathsf{BF}}$ or $[\mathsf{UnitPropDef}]^{\mathsf{BF}}$ when possible; otherwise,
- apply any rule of UnitProp or UnitPropDef when possible; otherwise,
- apply any rule of $[\mathsf{BwLoop}]^{\mathsf{BF+}}$ or $[\mathsf{FwLoop}]^{\mathsf{BF+}}$ when possible; otherwise,
- apply Decide.

Again similar strategies, called respectively *Full BwLoop propagation* and *Full FwLoop propagation*, are derived for *BwLoop* and *FwLoop*.

**Stable loop propagation.** This strategy is based on the *Full loop propagation* strategy, but it restricts application of the transition rules based on relevant loops to *positive loops* throughout the search, and allows application of these rules on *mixed relevant loops* only when the state sequence is two-valued.

- apply any rule of $[\mathsf{UnitProp}]^{\mathsf{BF}}$ or $[\mathsf{UnitPropDef}]^{\mathsf{BF}}$ when possible; otherwise,
- apply any rule of UnitProp or UnitPropDef when possible; otherwise,
- apply any rule of $[\mathsf{BwLoop}]^{\mathsf{BF+}}$ or $[\mathsf{FwLoop}]^{\mathsf{BF+}}$ on positive loops when possible; otherwise,
- apply Decide when possible; otherwise,
- apply any rule of $[\mathsf{BwLoop}]^{\mathsf{BF}}$ or $[\mathsf{FwLoop}]^{\mathsf{BF}}$ on mixed relevant loops.

We again derive similar strategies, called respectively *Stable BwLoop propagation* and *Stable FwLoop propagation*, for *BwLoop* and *FwLoop*.

*Loop-based theory propagation* and *Full loop propagation* are equally decide-efficient: they both apply all possible propagation rules before applying Decide. They are both strictly more decide efficient than *Stable loop propagation*, because the latter postpones certain propagations.

The motivation for the *Stable loop propagation* strategy can be found in the observation that theories modelled according to the ID-logic modelling methodology contain only definitions that are total with respect to the theory $T$ in

which they occur—slightly abusing terminology, we call such theories also total. Proposition 4.12 tells us that in such cases, $T \cup comp(\Delta) \cup LF^{+}_{\Delta} \models LF^{\pm}_{\Delta}$. Thus, when no more propagation according to the restricted rules is possible, i.e., when $T \cup comp(\Delta) \cup LF^{+}_{\Delta}$ is satisfied, $LF^{\pm}_{\Delta}$ is also satisfied, so no more rules are applicable.[4] In other words, the final application of $[\mathsf{BwLoop}]^{\mathsf{BF}}$ and/or $[\mathsf{FwLoop}]^{\mathsf{BF}}$ rules for mixed relevant loops is necessary only when theories are not total, something which should not happen according to our methodology.

Nevertheless, even for the class of total theories, *Full loop propagation* is more decide-efficient then *Stable loop propagation*, as the following example illustrates.

**Example 5.8.** Consider the theory $T =$

$$\left\{ \; p \leftarrow \neg p \wedge a \; \right\}, \tag{5.8}$$

$$\neg b \vee \neg a, \tag{5.9}$$

$$\neg c \vee \neg a, \tag{5.10}$$

$$b \vee c, \tag{5.11}$$

and consider the empty state sequence. The only relevant loop for definition (5.8) is mixed: $\{p, \neg p\}$; its loop formula is $p \vee \neg p \supset \neg a$, or simply $\neg a$. Equations (5.9)–(5.10) entail $(b \vee c) \supset \neg a$, and therefore (5.9)–(5.11)$\models \neg a$. Thus, definition (5.8) is total with respect to (5.9)–(5.11). Nevertheless, while the only rule of *Stable loop propagation* applicable to $\emptyset \parallel T$ is $\mathsf{Decide}$, in *Full loop propagation* also $\mathsf{FwLoop}$ is applicable.

However, the *Stable loop propagation* strategy has important algorithmic advantages over the *Full loop propagation* strategy, as we will see in Section 5.4. These advantages stem from the fact that when only positive loops are sought, the search space can be restricted to atoms. (Note that in the context of loop-searching algorithms, by "search space" we always denote a graph of literals.)

The motivation for both the *Stable* and *Full loop propagation* strategies as compared to the *Loop-based theory propagation* strategy is also algorithmic in nature, but can already be illustrated with an example.

**Example 5.9.** Let $\Delta =$

$$\left\{ \begin{array}{l} p_1 \leftarrow p_2 \wedge a, \\ p_2 \leftarrow p_3, \\ \dots \\ p_n \leftarrow p_1 \end{array} \right\},$$

and consider the state $\neg a \parallel \Delta$. Then the *Full loop propagation* strategy would find the following derivation:

$$\neg a \parallel \Delta \stackrel{\mathsf{UnitPropDef}}{\Longrightarrow} \neg a \, \neg p_1 \parallel \Delta \stackrel{\mathsf{UnitPropDef}}{\Longrightarrow} \neg a \, \neg p_1 \, \neg p_n \parallel \Delta$$
$$\stackrel{\mathsf{UnitPropDef}^*}{\Longrightarrow} \neg a \, \neg p_1 \, \neg p_n \dots \neg p_3 \, \neg p_2 \parallel \Delta.$$

[4]Note that we then have that $M \models_{st} \Delta$, where $M$ is the current (two-valued) state sequence, according to Proposition 4.11.

Each of the UnitPropDef applications is a *local* propagation: it is based on one rule of $\Delta_{5.9}$. The above derivation is also valid in *Loop-based theory propagation*, but the latter strategy could also derive, e.g.,

$$\neg a \parallel \Delta \overset{\mathsf{BwLoop}}{\Longrightarrow} \neg a \, \neg p_1 \parallel \Delta \overset{\mathsf{BwLoop}}{\Longrightarrow} \neg a \, \neg p_1 \, \neg p_2 \parallel \Delta$$
$$\overset{\mathsf{BwLoop}^*}{\Longrightarrow} \neg a \, \neg p_1 \, \neg p_2 \ldots \neg p_{n-1} \, \neg p_n \parallel \Delta,$$

using the fact that $\{p_1, p_2, \ldots, p_n\}$ is a loop, and BwLoop is applicable on it. The BwLoop applications, however, are *global*: they require *all* rules of $\Delta_{5.9}$ to be taken into account.[5] Such a global search is particularly time-ineffective if it does not yield any propagations, as would have been the case, e.g., if some rule $p_{i-1} \leftarrow p_i$ had not been in the definition.

Thus, the motivation for preferring the *Stable* and *Full loop propagation* strategies over the *Loop-based theory propagation* strategy is that algorithms for $[\mathsf{UnitProp}]^{\mathsf{BF+}}$ and $[\mathsf{UnitPropDef}]^{\mathsf{BF+}}$ are more time-effective than those for $[\mathsf{BwLoop}]^{\mathsf{BF+}}$ and $[\mathsf{FwLoop}]^{\mathsf{BF+}}$.

Observe also that *Full loop propagation* is equally decide-efficient as *Reduce to SAT*, and strictly more decide-efficient than *Incremental*. Indeed, *Incremental* can use loop formulas only for backtracking, while *Full loop propagation* and *Reduce to SAT* can use them also for propagation.

## 5.4 Algorithms

In this section we propose concrete algorithms for transition rules, taking into account the strategies in which they are to be used.

### 5.4.1 UnitPropDef

An algorithm for the $[\mathsf{UnitPropDef}]^{\mathsf{BF+}}$ transition rules is easy to construct by means of an algorithm for CNF unit propagation (i.e., for the $[\mathsf{UnitProp}]^{\mathsf{BF+}}$ rules). All that is required is to represent $comp(\Delta)$ in CNF form (cf. Figure 5.2; observe in particular that this representation is linear in the size of $\Delta$), and then apply the $[\mathsf{UnitProp}]^{\mathsf{BF+}}$ algorithm. We call the clauses denoted by $C(r)$ in Figure 5.2 *long clauses*.[6] Note that to reconstruct any rule $r$ it suffices to know the long clause $C(r)$, the head atom $h$ of $r$, and whether $h \in \mathcal{D}_{\text{lits}}$ or $h \in \mathcal{C}_{\text{lits}}$.

One particularly effective algorithm for CNF unit propagation is the "two watched literals scheme", introduced by Moskewicz et al. (2001). In any clause $\varphi$ with at least two literals, two different literals $w_1(\varphi)$ and $w_2(\varphi)$ are denoted as "watched". Then the algorithm ensures that the following invariant remains satisfied: $\forall \varphi \in \Psi : I(w_1(\varphi) \wedge w_2(\varphi)) \neq \boldsymbol{f} \vee I(w_1(\varphi) \vee w_2(\varphi)) = \boldsymbol{t}$, where $I$ is

[5] Observe that the BwLoop rule could have been formulated differently, deriving all literals of the loop to be false at once. Our argument here that *all* rules of the definition have to be taken into account for applying the rule remains valid also with this alternative formulation.

[6] If the size of the body is 1, then the "long clause" is actually also binary.

the current interpretation at any point of the search. To implement this, each literal keeps a list of the clauses in which it is watched. If then a literal becomes false, those clauses are visited, and we have the following possibilities for each clause $\varphi$:

- there is a third literal $l \in \varphi$ such that $I(l) = \boldsymbol{t}$: the watch from the false literal can move to $l$;
- there is no such third literal: then the *other* watched literal must be true, if $\varphi$ is to be satisfied. It may already be true. It may also already be false; in that case a conflict is derived. This last case may arise because literals that obtain a two-valued truth value must be processed sequentially: when the two watches of $\varphi$ both become false, one of them will necessarily be processed first.

It is easy to verify that $[\mathsf{UnitProp}]^{\mathsf{BF+}}$ on the clauses of $comp(\Delta)$ implements $[\mathsf{UnitPropDef}]^{\mathsf{BF+}}$.

*Remark* 5.3. Some ASP solvers provide four different propagation mechanisms for rules (from head to body and vice versa, for both disjunctive and conjunctive rules), which together are equivalent to the UnitPropDef rule. For instance, SMODELS implements these four mechanisms in its *AtLeast* procedure (Simons, 2000), using a counter-based algorithm derived from Dowling and Gallier's (1984) algorithm for computing the deductive closure of a set of rules.

### 5.4.2 Relevant loop algorithms

We now investigate algorithms related to relevant loops. Most algorithms proposed here have no direct application in any of the proposed transition rules, but are useful as a reference. They solve the following tasks:

- deciding whether a set of literals is a relevant loop;
- finding a relevant loop;
- finding all relevant loops of a definition.

"Stable" strategies require *positive* relevant loops: for finding these, the algorithms here can be simplified by searching not in the dependency graph of $\Delta$, but in its positive dependency graph (the restriction of the dependency graph to atoms), and by disregarding the search for justifying kernels.

#### Deciding whether a set of literals is a relevant loop

We assume a definition $\Delta$ and a set of literals $L$ to be given. By $G_L$ we denote the dependency graph of $\Delta$, restricted to $L$.

To decide whether $L$ is a relevant loop of $\Delta$, one first has to decide whether $L$ is a loop of $\Delta$. This can be done, e.g., using Tarjan's (1972) algorithm for finding strongly connected components: $L$ is a loop of $\Delta$ iff $G_L$ has a unique strongly connected component, namely the whole graph.

**Algorithm 5.1**: FindJustifyingKernels($L, G_L$)

**Result**: A set of literals: the union of all justifying kernels for $L$.
(Empty if there are none.)

1 $L' :=$ the negative literals in $L$;
2 **foreach** $l \in L' \cap \mathcal{D}_{lits}$ **do** $l.count := \text{card}\{(l,l') \in G_L \mid l' \in L'\}$;
3 $S := \{l \in L' \cap \mathcal{C}_{\text{lits}} \mid \exists (l,l') \in G_L : l' \notin L'\} \cup \{l \in L' \cap \mathcal{D}_{\text{lits}} \mid l.count = 0\}$;
4 $L' := L' \setminus S$;
5 **while** $L' \neq \emptyset \wedge S \neq \emptyset$ **do**
6     Choose $s \in S$; $S := S \setminus \{s\}$;
7     **foreach** $(l, s) \in G_L$ *such that* $l \in L'$ **do**
8         **if** $l \in \mathcal{C}_{lits}$ **then** $S := S \cup \{l\}$; $L' := L \setminus \{l\}$;
9         **if** $l \in \mathcal{D}_{lits}$ **then**
10            $l.count$--;
11            **if** $l.count = 0$ **then** $S := S \cup \{l\}$; $L' := L \setminus \{l\}$;
12 **return** $L'$;

When $L$ is indeed a loop of $\Delta$, Algorithm 5.1 can be used to determine whether or not $L$ contains any justifying kernels, and therefore, whether $L$ is a *relevant* loop.

Algorithm 5.1 assumes that $L$ is a loop in $\Delta$. The algorithm uses the integer data structure $l.count$ for disjunctive literals $l$, and keeps track of two sets, $L'$ and $S$. Both sets are subsets of $L$, and they satisfy $L' \cap S = \emptyset$ throughout the algorithm. Intuitively $S$ contains only literals that are certainly not in any justifying kernel for $L$; $L'$ contains those literals that might be. The algorithm proceeds by trying to remove literals from $L'$ by propagation from literals of $S$. This happens for disjunctively defined literals, when it is shown that none of their disjuncts belongs to $L'$, and for conjunctively defined literals, when one of their conjuncts is shown not to belong to $L'$. When all propagations from a given literal $s$ of $S$ are done, that literal is removed from $S$. The algorithm finishes either when $L'$ is empty (thus, there are no justifying kernels for $L$), or when $S$ is empty (thus, since all propagations are done, $L'$ is a justifying kernel, or a union of different justifying kernels).

This algorithm (as well as Tarjan's algorithm for deciding whether $L$ is a loop) is linear in the size of $L$.

### Finding relevant loops

To *find* a relevant loop $L$, rather than just deciding whether a given set of literals $L$ is one, we can appeal to the following property:

**Proposition 5.3.** *If $\Delta$ has a loop, it has a relevant loop.*

*Proof.* Let $L$ be a loop in $\Delta$. Either $L$ is relevant, or it contains a justifying kernel $L'$. Then $\overline{L'}$ is a relevant loop, since it is a loop, and consists of atoms only. □

Therefore finding a relevant loop can be done by first finding a loop, using Algorithm 5.1 to decide whether it has a justifying kernel, and if so, negating the literals in that kernel. To find a loop, one can use, for instance, Tarjan's (1972) algorithm for finding strongly connected components. And to find a justifying kernel, instead of returning true or false, Algorithm 5.1 can be made to return the union $L'$ of all justifying kernels for a given loop. If $L'$ is non-empty, an individual justifying kernel can easily be extracted by choosing an arbitrary $l \in L'$, and searching all literals in $L'$ that are reachable from $l$. The resulting algorithm is also linear (in the size of $\Delta$).

**Finding all relevant loops**

The *Reduction to SAT* strategy requires finding *all* relevant loops. It was shown by Lifschitz and Razborov (2006) that in the worst case there are exponentially many loop formulas (in the size of the definition). Therefore the usefulness of this strategy has to be doubted.

A simple method is to first find all strongly connected components of the dependency graph, and subsequently test each subset of each component for relevance. Naturally, a lot of redundancy in this method can be avoided with slightly more involved techniques, as well as by taking into account that it suffices to find all *elementary* loops (cf. Section 4.4.3), but the exponential nature can most likely not be avoided.

In contrast, the reduction to SAT proposed by Pelov and Ternovska (2005) introduces new symbols, but does not cause the theory size to explode: the authors show that the reduction of a PC(ID) theory $T$ with one definition $\Delta$ has size $\mathcal{O}(size(T) \times |Def(\Delta)|)$, where $size(T)$ is the total number of literals in $T$ and $|\cdot|$ means cardinality.

### 5.4.3 BwLoop

We recall the BwLoop transition rule:

$$M \parallel \Psi, \Delta \quad \Longrightarrow M\, \neg l \parallel \Psi, \Delta \quad \textbf{if} \begin{cases} M(l) = \boldsymbol{u} \\ \Delta \text{ contains a relevant loop } L \text{ such that} \\ \qquad\qquad M(\bigvee \mathcal{D}^{\text{ext}}(L)) = \boldsymbol{f} \text{ and } l \in L. \end{cases}$$

To implement this rule (and the $[\mathsf{BwLoop}]^{\mathsf{BF}}$ rules), an algorithm is needed that finds not just a relevant loop, but a relevant loop $L$ for which in the current interpretation $M$, $M(\bigvee \mathcal{D}^{\text{ext}}(L)) = \boldsymbol{f}$ and $M(\bigvee L) \neq \boldsymbol{f}$ hold. In other words, a relevant loop whose literals are not (all) false, but whose external disjuncts are. We call such a loop an *unfounded loop in $M$*. Recall that Proposition 4.15 proves that if $M(l) = \boldsymbol{u}$ for each $l \in L$, such a loop is also an unfounded set in $M$.

The algorithm that we propose to search for unfounded loops in $M$ is based on justification semantics. First, to see the relation between relevant loops

and justifications, recall that a loop in a justification for $\Delta$ is also a loop (in the dependency graph) of $\Delta$. Conversely, observe that a loop of $\Delta$ can be constructed as the union of some loops in some justifications for $\Delta$, since $\Delta$'s dependency graph is the union of all justifications for $\Delta$.

The following result establishes to relevance of justification semantics to the search for unfounded loops in $M$.

**Proposition 5.4.** *Let $M$ be an interpretation for which a $\Delta$-witness $J$ exists. Then $\Delta$ contains no unfounded loops in $M$.*

*Proof.* Let $L$ be a relevant loop in $\Delta$ with $M(\bigvee L) \neq \boldsymbol{f}$. Assume towards contradiction that $M(\bigvee \mathcal{D}^{\text{ext}}(L)) = \boldsymbol{f}$. Let $NF = \{l \mid M(l) \neq \boldsymbol{f}\}$, and let $l \in NF$. Because $J$ supports $M$, if $l \in \mathcal{C}_{\text{lits}}$, then all $l' \in \varphi_l$ have $M(l') \neq \boldsymbol{f}$, while if $l \in \mathcal{D}_{\text{lits}}$, $M(d_J(l)) \neq \boldsymbol{f}$. By assumption, this implies that $d_J(l) \notin \mathcal{D}^{\text{ext}}(L)$. We therefore have found that any non-false literal in $L$ has a non-false child in the restriction of $J$ to $L$. Hence this graph contains a minimal strongly connected component $L'$. $L'$ has no children in $NF$. Because $J$ is a witness, $L'$ must be a negative loop. Then since $L$ is a relevant loop, we have that $\mathcal{C}^{\text{ext}}(L') \cap L \neq \emptyset$. Let $c$ be a literal in $\mathcal{C}^{\text{ext}}(L') \cap L$: again because $J$ supports $M$, we find that $M(c) \neq \boldsymbol{f}$. But then $c \in NF$, which contradicts our construction of $L'$. □

Our strategy will be to search for (construct) a $\Delta$-witness; when it is found, then by Proposition 5.4 the $[\mathsf{BwLoop}]^{\mathsf{BF}+}$ rules are not applicable any more. The search will fail if there is an unfounded loop $L$ in $M$: instead of finding a $\Delta$-witness, the search will then return $L$.

We will construct a $\Delta$-witness by making *local* changes to a given justification. We first define a helpful concept.

**Definition 5.3** (Local witness)**.** Let $M$ be a (three-valued) interpretation, $l \in \widehat{Def(\Delta)}$ a defined literal with $M(l) \neq \boldsymbol{f}$, and $J$ a justification for $\Delta$. Then $J$ is a *local* $\Delta$*-witness for* $M$ *in* $l$ if $J$ supports $M$, and $J$ contains no non-negative loops that contain $l$.

A justification $J$ that is a local witness for $M$ in all non-false literals $l \in \widehat{Def(\Delta)}$ is a witness for $M$. It also follows immediately from this definition that if there exists a local witness for $M$ in some literal $l$, then there are no unfounded loops in $M$ that contain $l$. On the other hand, if no local witness exists for $M$ in some literal $l$, then also there exists no witness for $M$.

We will now construct an algorithm for the $[\mathsf{BwLoop}]^{\mathsf{BF}+}$ rules. This algorithm depends on a few assumptions:

1. the rules of $[\mathsf{UnitProp}]^{\mathsf{BF}+}$ and $[\mathsf{UnitPropDef}]^{\mathsf{BF}+}$ are not applicable to $M$;
2. a justification $J_s$ that supports $M$ is given;
3. we search for unfounded loops $L$ in $M$ that contain no false literals at all. This assumption ensures that we can restrict the search space to non-false literals;

4. a literal $cs$ with $M(cs) \neq \boldsymbol{f}$, called a *cycle source*, is given; we search for unfounded loops $L$ in $M$ that contain $cs$.

In Section 5.5 we will motivate and discuss these assumptions, and indicate how $J_s$ and $cs$, which we assume here to be given, can be acquired. Here we limit the discussion to a few remarks about these assumptions.

- Assumption 1. We call a state in which no rule of $[\mathsf{UnitProp}]^{\mathsf{BF+}}$ or $[\mathsf{UnitPropDef}]^{\mathsf{BF+}}$ is applicable a *UP-saturated* state. If we apply the $[\mathsf{BwLoop}]^{\mathsf{BF+}}$ rules in the context of a strategy like *Full* or *Stable loop propagation*, this assumption will hold. We refer to Example 5.9 to motivate the use of those strategies.

- Assumption 2. In a UP-saturated state $M$, there always exists a supporting justification $J_s$.

  Furthermore, it is easy to find one: $J_s$ can simply be constructed by searching, for each $l \in \mathcal{D}_{\text{lits}}$ with $M(l) \neq \boldsymbol{f}$, for a literal $l' \in B_l$ with $M(l') \neq \boldsymbol{f}$; the existence of such a literal is a consequence of the fact that the state is UP-saturated. In MiniSat(ID), we exploit the two watched literals data structure to search for a supporting justification even more efficiently. We explain the details in Section 5.7.3.

- Assumption 3. By restricting the search space to non-false literals, we may miss out on certain unfounded loops $L$ in $M$. However, we will show in Section 5.5 that the falsity of literals in those loops will be derived anyway by a combination of BwLoop on *subloops* of $L$ and UnitPropDef.

- Assumption 4. We will develop the notion of cycle sources in Section 5.5; it suffices here that these are literals that are potentially in an unfounded loop $L$ in $M$.

Our algorithm starts from the given cycle source $cs$, and tries to find a local witness for $M$ in $cs$. It either succeeds, or it finds an unfounded loop in $M$.

We first give a brief intuition behind the algorithm. Starting from $cs$, we first find a set of literals that *might* contain a loop through $cs$. Then we try to "justify" literals of that set by making local changes to $J_s$. A literal is "justified" once it is certain that $J_s$ contains no path from this literal to $cs$. When $cs$ itself is justified, $J_s$ has no loop through it, and because it supports $M$, it is then a local witness for $M$ in $cs$. When on the other hand all possible ways of justifying literals have been tried in vain, all literals that are not justified are in a loop. Potentially that loop is not a relevant loop: we can apply Algorithm 5.1 to test this.

We now give a more comprehensive high-level overview of the algorithm.

- Initially, the algorithm constructs the set $HP$ ("has path") of all literals that have a path (of length at least one) to $cs$ in $J_s$, such that no literal along the path is false in $M$. This is a simple depth-first search in the transpose of $J_s$ (i.e., the symmetric graph of $J_s$). Hence, all literals that

belong to an eventual unfounded loop in $J_s$ are in $HP$, and possibly some more. However, if $cs \notin HP$, then $J_s$ is already a local witness for $M$ in $cs$.

- The algorithm attempts to "justify" literals by making local changes to $J_s$. A literal is justified when it certainly has no path in $J_s$ to $cs$. When a literal gets justified, it is therefore removed from $HP$. The purpose of the algorithm is to justify $cs$.

- The algorithm may make local changes to $J_s$, but all changes retain the invariant that $J_s$ supports $M$. $M$ itself remains unchanged throughout the algorithm. Making a local change to $J_s$ means setting $d_{J_s}(l) := l'$, for some literal $l \in HP \cap \mathcal{D}_{\text{lits}}$ and $l' \in \varphi_l$. The support invariant is retained by choosing $l'$ such that $M(l') \neq \boldsymbol{f}$, since $l \in HP$ and therefore $M(l) \neq \boldsymbol{f}$.

- A literal $l \in \mathcal{D}_{\text{lits}}$ can be justified by setting $d_{J_s}(l)$ to a literal that does not belong to $HP$, and therefore has no path to $cs$ in $J_s$. If $\varphi_l$ contains no such literal, it is possible that it contains a literal $l'$ that can be justified first, after which $l$ can still be justified by setting $d_{J_s}(l) := l'$. A literal $l \in \mathcal{C}_{\text{lits}}$ can be justified by showing that *all* literals in $\varphi_l$ do not belong to $HP$. If there are $HP$ literals in $\varphi_l$, they have to be justified first.

  This top-down process of trying to justify literals is executed by means of a working queue $Q$, initialized with $cs$.[7] It contains literals that we may still try to justify. Literals are pushed on $Q$ at most once. And since only $HP$ literals are considered, $Q$ is therefore eventually emptied, or $cs$ gets justified before that.

- The algorithm maintains a set $L$, initialised with $cs$, of literals that are waiting to be justified. Hence, every literal that is added to $Q$ is added to $L$ as well. But whereas literals are popped from $Q$ as soon as an attempt at their justification is made, they are removed from $L$ only when such an attempt is successful—i.e., "justifying" a literal means removing it both from $HP$ and from $L$. All literals in $L$ are reachable from $cs$ in the dependency graph. When the algorithm terminates with an empty queue $Q$, but a non-empty set $L$, all literals in $L$ are in a loop in *any* justification that supports $M$. Thus, if $L$ does not contain a justifying kernel, it is then an unfounded loop in $M$.

Algorithm 5.2 is the concrete algorithm. The function "Justify($l$)" in it justifies literal $l$, and propagates this justifiedness in a bottom-up way to its parents in $L$.

**Proposition 5.5.** *Let $M$, $J_s$, $cs$ satisfy assumptions 1–4. Then if Algorithm 5.2, called on $cs$, returns a non-empty set $L$, then $L$ is an unfounded loop in $M$ iff*

[7] The use of a queue as data structure, and not a stack, means that the algorithm searches for changes to $J_s$ that may justify $cs$ in a breadth-first way, i.e., as close to $cs$ as possible. This is advantageous for cycle sources that can be justified—they are justified as soon as possible—and not disadvantageous for other cycle sources.

**Algorithm 5.2**: BwLoop($cs, M, J_s, \Delta$)—Justifying a cycle source $cs$

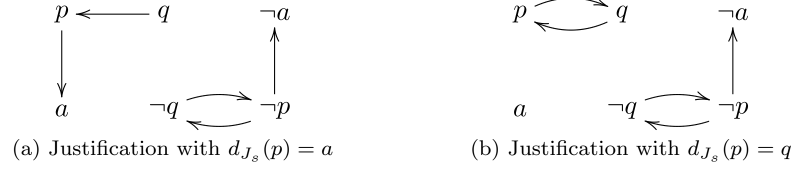
**Result**: A set of literals that is either empty or an unfounded loop in $M$. $J_s$ may change; if result is $\emptyset$, $J_s$ is local $\Delta$-witness for $M$ in $cs$.

1 Let $HP := \{l \mid l$ has a path to $cs$ in $J_s$ through non-false literals$\}$;
2 **if** $cs \notin HP$ **then return** $\emptyset$;
3 Let $L := Q := \{cs\}$;
4 **while** $Q \neq \emptyset$ **do**
5     Pop literal $l$ from $Q$;
6     **if** $l \in \mathcal{D}_{lits}$ **then**
7         **if** $\exists l' \in \varphi_l$ *such that* $l' \notin HP$ *and* $M(l') \neq \boldsymbol{f}$ **then**
8             $d_{J_s}(l) := l'$; `Justify`($l$);
9         **else**
10             **foreach** $l' \in \varphi_l$ *such that* $M(l') \neq \boldsymbol{f}$ *and* $l' \notin L$ **do**
11                 Add $l'$ to $Q$ and to $L$;
12     **else**
13         **if** $\varphi_l \cap HP$ *is empty* **then**
14             `Justify`($l$);
15         **else**
16             **foreach** $l' \in \varphi_l \cap HP \setminus L$ **do** Add $l'$ to $Q$ and to $L$;
17 **return** $L$;
18 **function Justify** $l$
19     Remove $l$ from $HP$, $L$ and $Q$;
20     **if** $l = cs$ **then return** $\emptyset$;
21     **foreach** $l' \in L \cap \mathcal{C}_{lits}$ *with* $l \in \varphi_{l'}$ **do**
22         **if** $\varphi_{l'} \cap HP$ *is empty* **then** Justify($l'$);
23     **foreach** $l' \in L \cap \mathcal{D}_{lits}$ *with* $l \in \varphi_{l'}$ **do**
24         $d_{J_s}(l') := l$; Justify($l'$);
25 **end function**

*$L$ contains no justifying kernel. If instead Algorithm 5.2 returns the empty set, the resulting $J_s$ is a local $\Delta$-witness for $M$ in $cs$.*

*Proof.* (Sketch) The following properties can easily be proven to be invariant.

1. $J_s$ supports $M$.
2. $Q \subseteq L \subseteq HP$.
3. For any literal $l \in L$, the dependency graph contains a path from $cs$ to $l$.
4. Any literal *not* in $HP$ certainly has no path in $J_s$ to $cs$. Any literal $l \in HP$ either has a path in $J_s$ to $cs$, or is waiting to be removed from $HP$ (as may be the case for literals $\in \mathcal{C}_{\text{lits}}$ on which Line 22 has not yet called `Justify`).




(a) Justification with $d_{J_s}(p) = a$ (b) Justification with $d_{J_s}(p) = q$

Figure 5.5: Justifications for $\Delta_{5.10}$

5. (At each start of the while loop.) For any literal $l \in L \setminus Q$, and any path from $l$ to $cs$ in the dependency graph (by 2 and 4 that is at least one path):
   - the path contains a literal in $Q$, or
   - all literals along the path are in $L \setminus Q$, and for each literal $l' \in \mathcal{D}_{\text{lits}}$ along the path, all literals in $\varphi_{l'} \setminus L$ are false in $M$.

When Algorithm 5.2 returns a non-empty set $L$ (Line 17), $Q$ is empty. Then by invariants 3 and 5, each literal $l \in L$ is in a loop in the dependency graph (that also contains $cs$), and all literals in $\mathcal{D}^{\text{ext}}(L)$ are false in $M$. Hence, if $L$ is a relevant loop, i.e., if it contains no justifying kernel, then it is an unfounded loop in $M$.

When Algorithm 5.2 returns the empty set, then by invariants 2, 4 and 5, $J_s$ contains no loop through $cs$; by invariant 1 it is therefore a local witness for $M$ in $cs$. □

Algorithm 5.2 has running time that is linear in the size of $\Delta$, restricted to $HP$.[8] In the worst case, $HP = \widehat{Def(\Delta)}$, but typically, we expect $HP \ll \widehat{Def(\Delta)}$.

**Example 5.10.** Let $\Delta_{5.10} =$

$$\left\{ \begin{array}{l} p \leftarrow q \vee a, \\ q \leftarrow p \end{array} \right\}.$$

- Let $M$ be the empty interpretation, $J_s$ the justification with $d_{J_s}(p) = a$ (see Figure 5.5a), and $cs = p$. Call Algorithm 5.2 on $p$. Then $HP = \{q\}$ in Line 1: $q$ is the only literal that has a non-empty path in $J_s$ to $p$. Since $p \notin \{q\}$, the algorithm returns the empty set; $J_s$ is a local witness for $M$ in $p$ (and, incidentally, a witness for $M$).
- Let $M$ be the empty interpretation as before, $cs = p$, but now $d_{J_s}(p) = q$ (see Figure 5.5b). Call Algorithm 5.2 on $p$. Then $HP = \{p, q\}$ in Line 1. The algorithm continues with setting $L := Q := \{p\}$, entering

[8] The if-test in Line 22 must be implemented using a counter, to avoid that $\varphi_{l'}$ is visited $|\varphi_{l'}|$ times.

the while loop, and popping $p$ from $Q$. It finds that $a \in \varphi_p$, $a \notin HP$ and $M(a) \neq \boldsymbol{f}$, hence it sets $d_{J_s}(p) := a$ and calls `Justify(`$p$`)` in Line 8. In the `Justify(`$p$`)` function, it is immediately found that $p$ is the cycle source; again the algorithm returns the empty set and (the changed) $J_s$ is a local witness for $M$ in $p$.

- Let $M$ be $\{a \mapsto \boldsymbol{f}\}$, $cs = p$, and $d_{J_s}(p) = q$. Call Algorithm 5.2 on $p$. The algorithm commences the same as in previous case, finding $HP = \{p, q\}$ and $L := Q := \{p\}$, and entering the while loop with $p$. Since $M(a) = \boldsymbol{f}$, the if-test fails, and the algorithm reaches the foreach statement in Line 10, where $q$ is added to $Q$ and $L$. We then have $L = \{p, q\}$, $Q = \{q\}$. In the next iteration of the while-loop, $q$ is popped from $Q$. It is $\in \mathcal{C}_{\text{lits}}$, and $\varphi_q$ contains $p$ which is a member of $HP$. The foreach statement of Line 16 does nothing, because $p$ is already in $L$. Then we have $Q = \emptyset$, so the algorithm returns the set $L = \{p, q\}$ in Line 17. Indeed it is an unfounded loop in $M$.

We give a more comprehensive discussion concerning this algorithm in Section 5.5, including an explanation how to construct a witness for $M$ from local witnesses for $M$ in various cycle sources.

### 5.4.4 FwLoop and $\Delta$-Propagate

Recall the FwLoop and $\Delta$-Propagate transition rules:

$$M \parallel \Psi, \Delta \quad \Longrightarrow M\, d \parallel \Psi, \Delta \quad \textbf{if} \begin{cases} M(d) = \boldsymbol{u} \\ \Delta \text{ contains a relevant loop } L \text{ such that} \\ \qquad M(\bigvee L) = \boldsymbol{t},\ d \in \mathcal{D}^{\text{ext}}(L), \text{ and} \\ \qquad\qquad M(\bigvee \mathcal{D}^{\text{ext}}(L) \setminus \{d\}) = \boldsymbol{f}, \end{cases}$$

$$M \parallel \Psi, \Delta \quad \Longrightarrow M\, l \parallel \Psi, \Delta \quad \textbf{if} \begin{cases} M(l) = \boldsymbol{u} \\ M \models_\Delta l \end{cases}$$

To implement the FwLoop rule, an algorithm is needed that finds a relevant loop $L$ containing at least one true literal in $M$, such that all literals but one of $\mathcal{D}^{\text{ext}}(L)$ are false in $M$, and the remaining one is unknown. Note that for the $[\text{FwLoop}]^{\text{BF}}$ rules, the requirement becomes that *all* literals of $\mathcal{D}^{\text{ext}}(L)$ are false in $M$—as for the $[\text{BwLoop}]^{\text{BF}}$ rules. We focus here on FwLoop's distinctive feature: all but one of the literals in $\mathcal{D}^{\text{ext}}(L)$ is false, and the non-false literal is unknown. Therefore we assume that BwLoop is not applicable any more.

A possible implementation is to try candidate disjuncts one by one, verifying for each of them whether it is an FwLoop consequence. The obvious method of verifying this, is temporarily assuming the candidate disjunct to be false, and subsequently applying the BwLoop algorithm, Algorithm 5.2. We show that this method is slightly stronger than FwLoop: it actually implements the $\Delta$-Propagate transition rule.

**Algorithm 5.3**: $\Delta$-Propagate$(M, J_w, \Delta)$

1 Let $S := \{l \mid l \text{ is unknown in } M\}$;
2 **while** $S \neq \emptyset$ **do**
3 Pop $l$ from $S$;
4 Let $CS := \{l' \in \mathcal{D}_{\text{lits}} \mid d_{J_w}(l') = l\}$;
5 **while** $CS \neq \emptyset$ **do**
6 Choose some $cs \in CS$;
7 Let $L :=$ the result of Algorithm 5.2 on $cs$, assuming the interpretation $M \cup \{l \mapsto \boldsymbol{f}\}$;
8 **if** $L \neq \emptyset$ *and* $M(\bigvee L) = \boldsymbol{t}$ *and* $L$ *is relevant* **then return** $l$;
9 **else** $CS := CS \setminus \{cs\}$;
10 Fail;

Let $M$ be an interpretation. A witness for $M$ or for an extension of $M$ is called an *extended witness* for $M$. The intuition of this concept is that some propagations may still be needed before an interpretation is reached that has a witness. For instance, for the definition $\{\ p \leftarrow a\ \}$, the interpretation $\{a \mapsto \boldsymbol{f}\}$ does not have a witness, simply because the propagation to $\neg p$ has not happened yet.

We then have the following derivation from the condition $M \models_\Delta l$ of the $\Delta$-Propagate rule.

$$M \models_\Delta l \text{ iff}$$
$$\text{any model } M' \text{ of } \Delta \text{ extending } M \text{ has } M'(l) = \boldsymbol{t} \text{ iff}$$
$$M\,\neg l \text{ cannot be extended to a model of } \Delta \text{ iff}$$
$$M\,\neg l \text{ cannot be extended to an interpretation that has a } \Delta\text{-witness.}$$

By assuming a candidate disjunct $l$ to be false and then applying Algorithm 5.2, we verify whether $M\,\neg l$ has an extended $\Delta$-witness (cf. Propositions 5.4 and 5.5).

In a UP-saturated state where BwLoop is not applicable any more, there exists a witness $J_w$. In Section 5.5 we illustrate that a witness $J_w$ can be constructed by repetitive application of Algorithm 5.2. Here we assume $J_w$ to be given.

For any given literal $l$, it suffices to run Algorithm 5.2 on the cycle sources $CS := \{l' \in \mathcal{D}_{\text{lits}} \mid d_{J_w}(l') = l\}$. Indeed, any loop $L$ that is unfounded in $M\,\neg l$ certainly contains a literal of $CS$. Algorithm 5.3 implements this procedure of trying to find a $\Delta$-Propagate consequence of $M$. Its complexity is $\mathcal{O}(|Def(\Delta)| \times |\Delta|)$: there are in total $|Def(\Delta)|$ cycle sources on which to call Algorithm 5.2, which takes (in the worst case) $\mathcal{O}(|\Delta|)$ time.

**Example 5.11.** We illustrate Algorithm 5.3. Let $\Delta = \left\{ \begin{array}{l} p \leftarrow q \vee a, \\ q \leftarrow p \wedge b \end{array} \right\}$, $M =$

$\{p \mapsto \boldsymbol{t}\}$, and $J_w$ the justification with $d_{J_w}(p) = a$ and $d_{J_w}(\neg q) = \neg b$. Suppose Line 3 chooses $q$ to assume false. Then $CS = \emptyset$; the inner while-loop is finished trivially. The outer while-loop restarts; suppose Line 3 chooses $a$ to assume false. Then $CS = \{p\}$; Algorithm 5.2 applied on $p$ in interpretation $\{p \mapsto \boldsymbol{t}, a \mapsto \boldsymbol{f}\}$ now yields the loop $\{p, q\}$. Indeed, $\{p, q\}$ is a relevant loop and $a$ is a $\Delta$-Propagate consequence of $M$.

### 5.4.5 Simplify

The Simplify transition rule is intended to find unavoidably false literals:

$$M \parallel \Psi, \Delta \qquad \Longrightarrow M \parallel \Psi, \Delta^{\emptyset[M]} \qquad \textbf{if } \left\{ \circlearrowleft[M] \text{ is consistent} \right.$$

Recall that the set $\circlearrowleft[M]$ was defined as the set of defined literals $l$ such that $J_{sub}(l)$ contains a non-negative loop for any justification $J$ that supports $M$. Thus, the literals of $\circlearrowleft[M]$ are unavoidably false because they are in, or depend on, unavoidable non-negative loops. In this respect, Simplify can be intuitively understood as a global version of BwLoop: it does not just find one unfounded loop in $M$; it finds all of them.

Suppose $M$ is UP-saturated, and $\circlearrowleft[M]$ is consistent. Let $M'$ be the state sequence reached after UP-saturation of $M[\circlearrowleft[M]/\boldsymbol{f}]$. Then it follows from this definition that there exists a witness for $M'$. As such, an algorithm for finding $\circlearrowleft[M]$ can be used both as an algorithm for Simplify, and as an algorithm for finding an initial witness. In the next section, we will need such an initial witness.

On the precondition that $M$ is UP-saturated, Algorithm 5.4 finds $\circlearrowleft[M]$, and therefore $\Delta^{\emptyset[M]}$. It also finds a justification $J_w$ that supports $M$, and such that for any literal $l$ *not* in $\circlearrowleft[M]$, either $M(l) = \boldsymbol{f}$ or $J_{w,sub}(l)$ contains no non-negative loops.

Algorithm 5.4 is intended to be used only initially, i.e. when $M$ is reached through UP-saturation of the empty interpretation. However, it is applicable throughout the search.

The intuition behind Algorithm 5.4 is similar to that behind Algorithm 5.2. In this algorithm, however, we do not start from a cycle source, and the initial set of literals that might contain one or more loops is the set of all defined literals. "Justifying" a literal from the set is done in the same way as before, but now in a bottom-up order—because we want to perform a global search.

We explain Algorithm 5.4 in some more detail. It initially overestimates $\circlearrowleft[M]$ by all non-false defined literals (Line 1), and then iteratively (the repeat-loop) removes literals $l$ for which the constructed justification $J_w$ satisfies the requirements. The first while-loop removes literals $l$ for which the constructed justification $J_w$ contains no loops at all through $l$; the second one finds literals $l$ in negative loops, which are removed in Line 18.

**Algorithm 5.4**: Finding $\circlearrowleft[M]$

1 Let $S^{\circlearrowleft} := \{l \in \widehat{Def(\Delta)} \mid M(l) \neq \boldsymbol{f}\}$;
2 **foreach** $l \in \mathcal{C}_{lits}$ **do** $l.count := \text{card}\{\varphi_l\}$;
3 Let $Ch := \{l \in \widehat{Open(\Delta)} \mid M(l) \neq \boldsymbol{f}\}$;
4 **repeat**
5     **while** $Ch \neq \emptyset$ **do**
6         Let $l \in Ch$; $Ch := Ch \setminus \{l\}$;
7         **foreach** $l' \in \mathcal{D}_{lits} \cap S^{\circlearrowleft}$ *with* $l \in \varphi_{l'}$ **do**
8             $S^{\circlearrowleft} := S^{\circlearrowleft} \setminus \{l'\}$; $Ch := Ch \cup \{l'\}$; $d_{J_w}(l') := l$;
9         **foreach** $l' \in \mathcal{C}_{lits} \cap S^{\circlearrowleft}$ *with* $l \in \varphi_{l'}$ **do**
10             $l'.count$--;
11             **if** $l'.count = 0$ **then** $S^{\circlearrowleft} := S^{\circlearrowleft} \setminus \{l'\}$; $Ch := Ch \cup \{l'\}$;
12     Let $U := \{l \in S^{\circlearrowleft} \mid l$ is a negative literal$\}$; $fixpoint := \boldsymbol{f}$;
13     **while** $\neg fixpoint$ **do**
14         $fixpoint := \boldsymbol{t}$;
15         **foreach** $u \in U$ **do**
16             **if** $u \in \mathcal{D}_{lits} \wedge \varphi_u \cap U = \emptyset$, *or* $u \in \mathcal{C}_{lits} \wedge \varphi_u \cap S^{\circlearrowleft} \not\subseteq U$ **then**
17                 $U := U \setminus \{u\}$; $fixpoint := \boldsymbol{f}$;
18     $S^{\circlearrowleft} := S^{\circlearrowleft} \setminus U$; $Ch := U$;
19     **foreach** $u \in \mathcal{D}_{lits} \cap U$ **do** $d_{J_w}(u) := u'$ for some $u' \in \varphi_u \cap U$;
20 **until** $Ch = \emptyset$ ;
21 **return** $S^{\circlearrowleft}$;

Since justifications have no edges leaving from open literals, all open literals can be used as starting points of the bottom-up propagation; Line 3 further restricts the starting points to non-false literals, so that three-valued support can be preserved when changing $J_w$. When the first while-loop is finished, the remaining literals in the overestimate $S^{\circlearrowleft}$ for $\circlearrowleft$ are all in a loop, or depend on one: for each $l \in S^{\circlearrowleft}$, $\varphi_l \cap S^{\circlearrowleft} \neq \emptyset$. Next, Line 12 initializes a set $U$ to all negative literals in $S^{\circlearrowleft}$, and the second while-loop shrinks this set until it contains only literals $u \in \mathcal{D}_{\text{lits}}$ for which $\varphi_u$ contains a literal in $U$, and literals $u \in \mathcal{C}_{\text{lits}}$ for which $\varphi_u \cap S^{\circlearrowleft} \subseteq U$. Thus, $U$ contains a negative loop in the dependency graph, and after the changes to $J_w$ in Line 19, the literals in $U$ are all in a negative loop in $J_w$, or depend on one—and can therefore also be removed from $S^{\circlearrowleft}$.

This algorithm can be seen as an adaptation of an algorithm for finding the well-founded model by Berman et al. (1995); it behaves as Berman et al.'s algorithm would when all non-false open body literals are replaced with $\boldsymbol{t}$. With an appropriate implementation of the foreach-loop of Line 15, its running time is quadratic in the size of $\Delta$. If $\Delta$ is total (with respect to the empty theory), it is linear.

**Algorithm 5.5**: Finding $\circlearrowright_{st}[M]$

1 Let $S^{\circlearrowright} := \{l \in Def(\Delta) \mid M(l) \neq \boldsymbol{f}\}$;
2 **foreach** $l \in \mathcal{C}_{lits}$ **do** $l.count := \mathrm{card}\{\varphi_l\}$;
3 Let $Ch := \{l \in \widehat{Open(\Delta)} \cup \overline{Def(\Delta)} \mid M(l) \neq \boldsymbol{f}\}$;
4 **while** $Ch \neq \emptyset$ **do**
5 Let $l \in Ch$; $Ch := Ch \setminus \{l\}$;
6 **foreach** $l' \in \mathcal{D}_{lits} \cap S^{\circlearrowright}$ *with* $l \in \varphi_{l'}$ **do**
7 $S^{\circlearrowright} := S^{\circlearrowright} \setminus \{l'\}$; $Ch := Ch \cup \{l'\}$; $d_{J_w}(l') := l$;
8 **foreach** $l' \in \mathcal{C}_{lits} \cap S^{\circlearrowright}$ *with* $l \in \varphi_{l'}$ **do**
9 $l'.count$--;
10 **if** $l'.count = 0$ **then** $S^{\circlearrowright} := S^{\circlearrowright} \setminus \{l'\}$; $Ch := Ch \cup \{l'\}$;
11 **return** $S^{\circlearrowright}$;

**Example 5.12.** Let $\Delta_{5.12} = \left\{ \begin{array}{l} p \leftarrow q \wedge a, \\ q \leftarrow p \end{array} \right\}$, and $M = \{a \mapsto \boldsymbol{t}\}$. Then $M$ is UP-saturated. $\Delta_{5.12}$ has two justifications, which both contain the non-negative loop $L = \{p, q\}$.

We illustrate Algorithm 5.4. Initially, $S^{\circlearrowright} = \{p, \neg p, q, \neg q\}$, $p.count = 2$, and $Ch = \{a\}$. In the first while-loop, $a$ is removed from $Ch$, and $p.count$ set to 1. Then $Ch = \emptyset$, and next $U$ is initialized to $\{\neg p, \neg q\}$. A fixpoint is immediately reached: both $\neg p$ and $\neg q$ have an element of $U$ in their rule body. $S^{\circlearrowright}$ is then reduced to $\{p, q\}$, $\{\neg p, \neg q\}$ is added to $Ch$, and $d_{J_w}(\neg p) := \neg q$ is set. In the next iteration of the repeat-loop, both $\neg p$ and $\neg q$ are removed from $Ch$ without any changes to $S^{\circlearrowright}$. The algorithm terminates with $\circlearrowright[M] = \{p, q\}$.

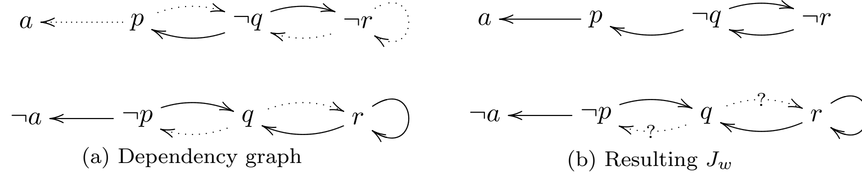
We have also defined the concept of *stable* unavoidably false literals, $\circlearrowright_{st}[M]$. We define a stable variant of the Simplify and SimplifyFail transition rules. Algorithm 5.5 implements this stable variant. Note that it is almost the same as Algorithm 5.4, with removal of the second while-loop. Algorithm 5.5 can therefore be seen as an algorithm to compute the *greatest unfounded set* with respect to $M$; it operates in linear time. While somewhat less sophisticated, it is based on the same principle as SMODELS' Atmost algorithm (Simons, 2000).

Notably, Algorithm 5.5 also finds an initial stable witness. We illustrate the different behaviours of Algorithms 5.4 and 5.5 on an example.

**Example 5.13.** Let $\Delta_{5.13} =$

$$\left\{ \begin{array}{l} p \leftarrow \neg q \vee a, \\ q \leftarrow \neg p \vee r, \\ r \leftarrow r \wedge q \end{array} \right\}.$$

The dependency graph of $\Delta_{5.13}$ is shown in Figure 5.6a. Consider $M = \emptyset$. Applying Algorithm 5.4 yields initial values of $S^{\circlearrowright} = \{p, \neg p, q, \neg q, r, \neg r\}$, $Ch = \{a, \neg a\}$, and *count* values for $\neg p$, $\neg q$ and $r$ of 2. In a first iteration of the repeat-loop, $a$ may be chosen, after which $p$ is removed from $S^{\circlearrowright}$ and $d_{J_w}(p) := a$ is




(a) Dependency graph (b) Resulting $J_w$

Figure 5.6: Dependency graph of and partial justification for $\Delta_{5.13}$.

set. When $\neg a$ is chosen next, $(\neg p).count := 1$ is set, and when $p$ is chosen, $(\neg q).count := 1$ is set, and the first while-loop ends.

Then $S^{\circlearrowleft} = \{\neg p, q, \neg q, r, \neg r\}$, and $U$ is initialized to $\{\neg p, \neg q, \neg r\}$. Subsequently $\neg p$ gets removed from $U$, because $\neg p \in \mathcal{C}_{\text{lits}}$, and $\varphi_{\neg q} \cap S^{\circlearrowleft} = \{q\}$, which is $\not\subseteq U$. The second while-loop ends, and $S^{\circlearrowleft}$ is set to $\{\neg p, q, r\}$, $Ch$ to $\{\neg q, \neg r\}$. Also, $d_{J_w}(\neg r)$ is set to $\neg r$ or $\neg q$, say $\neg q$.

In the next iteration of the repeat-loop, both $\neg q$ and $\neg r$ are removed from $Ch$, without changes to $S^{\circlearrowleft}$. After that $U$ is initialized to $\{\neg p\}$, but again $\neg p$ gets removed from $U$, ending the second while-loop with empty $U$. Then also the repeat-loop ends, and $\circlearrowleft[M] = \{\neg p, q, r\}$ is returned. Indeed, for each literal $l$ in $\circlearrowleft[M]$, $J_{sub}(l)$ contains a non-negative loop for any justification $J$ that three-valuedly supports $M$. The resulting justification, $J_w$, is shown in Figure 5.6b (note that $q \notin \text{dom}(J_w)$); any instance of $J_w$ is a witness for $M[\circlearrowleft[M]/\boldsymbol{f}]$.

Applying instead Algorithm 5.5 (note that now $J_w$ is a *stable* partial justification) yields initially $S^{\circlearrowleft} = \{p, q, r\}$ and $Ch = \{a, \neg a, \neg p, \neg q, \neg r\}$. In the while-loop, $p$ is removed from $S^{\circlearrowleft}$ and $d_{J_w}(p)$ is set to either $\neg q$ or $a$, and $q$ is removed from $S^{\circlearrowleft}$ and $d_{J_w}(q)$ set to $\neg p$. The algorithm terminates with the set $\{r\}$, which is an unfounded set.

## 5.5 Discussion of the BwLoop algorithm

We now focus attention on Algorithm 5.2, which implements the $[\mathsf{BwLoop}]^{\mathsf{BF+}}$ transition rules. We motivate the assumptions that we made in Section 5.4.3.

### 5.5.1 Search space restricted to non-false literals

We made the restriction of searching only for unfounded loops that contain no false literals at all in the given state sequence $M$. We motivate this restriction, but start with an example of an unfounded loop in $M$ that does contain false literals.

**Example 5.14.** Let $\Delta_{5.14} =$

$$\left\{ \begin{array}{l} p \leftarrow q \vee r \vee a, \\ q \leftarrow p, \\ r \leftarrow p \wedge q \end{array} \right\},$$

and consider the state $\neg a \neg r \parallel \Delta_{5.14}$. This state is UP-saturated. $\Delta_{5.14}$ has two relevant loops: $L_1 = \{p, q, r\}$ with $\mathcal{D}^{\mathrm{ext}}(L_1) = \{a\}$, and $L_2 = \{p, q\}$ with $\mathcal{D}^{\mathrm{ext}}(L_2) = \{r, a\}$. With $M = \{a \mapsto \boldsymbol{f}, r \mapsto \boldsymbol{f}\}$ as in the given state, we have both $M(\mathcal{D}^{\mathrm{ext}}(L_1)) = \boldsymbol{f}$ and $M(\mathcal{D}^{\mathrm{ext}}(L_2)) = \boldsymbol{f}$; however, $L_1$ contains a literal that is false in $M$, $L_2$ does not.

The unfounded loop $L_1$ in the previous example contains a subloop that is also an unfounded loop, yet contains no false literals. The following property proves that we can always find such a subloop, and that therefore, we can safely discard negative literals from the search.

**Proposition 5.6.** *Let $M \parallel \Psi, \Delta$ be a UP-saturated state, and $L$ an unfounded loop in $M$ for which $\exists l \in L : M(l) = \boldsymbol{f}$. Then there exists an unfounded loop $L' \subseteq L$ in $M$, such that no literal of $L'$ is $\boldsymbol{f}$ in $M$.*

*Proof.* Recall that $L$ is an unfounded loop in $M$ if it is a relevant loop for which $M(\bigvee \mathcal{D}^{\mathrm{ext}}(L)) = \boldsymbol{f}$, and for which $M(\bigvee L) \neq \boldsymbol{f}$ (not all literals of $L$ are false).

Let $L_{\neq \boldsymbol{f}} = \{l \in L \mid M(l) \neq \boldsymbol{f}\}$. We construct a loop $L' \subseteq L_{\neq \boldsymbol{f}}$.

Let $G_{\neq \boldsymbol{f}}$ be the restriction of the dependency graph to $L_{\neq \boldsymbol{f}}$, and let $l \in L_{\neq \boldsymbol{f}}$. Let $X$ be the set of literals reachable from $l$ in $G_{\neq \boldsymbol{f}}$. By UP-saturatedness, every $l \in L_{\neq \boldsymbol{f}}$ has a child in $G_{\neq \boldsymbol{f}}$ (if $l \in \mathcal{C}_{\mathrm{lits}}$, *all* literals $l' \in \varphi_l$ are children in $G_{\neq \boldsymbol{f}}$). Therefore $X$ contains a loop. We define $L'$ as the minimal strongly connected component of $G_{\neq \boldsymbol{f}}$ restricted to $X$. By definition of $G_{\neq \boldsymbol{f}}$, $L'$ contains no literals that are false in $M$.

We further show that $M(\bigvee \mathcal{D}^{\mathrm{ext}}(L')) = \boldsymbol{f}$. Suppose the contrary: $\exists d \in \mathcal{D}^{\mathrm{ext}}(L')$ with $M(d) \neq \boldsymbol{f}$. Because $L$ is an unfounded loop in $M$, we have $d \notin \mathcal{D}^{\mathrm{ext}}(L)$, and since $L' \subset L$, we find $d \in L$. Since $d$ is by definition a body literal of some literal in $L'$, $d$ is reachable in $G_{\neq \boldsymbol{f}}$ from any literal $x \in L'$. Therefore $d$ should be in $L'$, which contradicts $d \in \mathcal{D}^{\mathrm{ext}}(L')$.

Finally, we show that $L'$ does not contain a justifying kernel. Suppose toward contradiction that $K \subseteq L'$ is a negative loop with $\mathcal{C}^{\mathrm{ext}}(K) \cap L' = \emptyset$. Clearly also $K \subseteq L$, and since $L$ is relevant, $\mathcal{C}^{\mathrm{ext}}(K) \cap L \neq \emptyset$. Hence $\mathcal{C}^{\mathrm{ext}}(K) \cap L \subseteq L \setminus L'$. Let $c$ be a literal of $\mathcal{C}^{\mathrm{ext}}(K) \cap L$: by UP-saturatedness, $M(c) \neq \boldsymbol{f}$, hence $c \in L_{\neq \boldsymbol{f}}$. Thus also $c$ is reachable in $G_{\neq \boldsymbol{f}}$ from any $l' \in L'$, and $c$ should be in $L'$, which contradicts $c \in \mathcal{C}^{\mathrm{ext}}(L')$. □

### 5.5.2 Cycle sources

The search for unfounded loops in $M$ is guided by the concept of *cycle source*: the literal where the search begins.

We want the probability that a given cycle source actually leads to an unfounded loop in $M$ to be as high as possible, in other words, we want as few cycle sources as possible. On the other hand, we also want to have a complete algorithm, hence, for each unfounded loop $L$ in $M$ there should be at least one cycle source $cs \in L$.

We take into account previous searches for unfounded loops in $M$.[9] Indeed, if $M(\bigvee \mathcal{D}^{\text{ext}}(L)) = \boldsymbol{f}$, then some literal $d \in \mathcal{D}^{\text{ext}}(L)$ must have become false *since the last time the same search was done.* Hence, literals $l \in \mathcal{D}_{\text{lits}}$ with a $d \in \varphi_l$ that has become false since the last search are good candidate cycle sources. We elaborate on this idea.

Let $M'$ be an interpretation that has a witness $J_w$, and let $M$ be an interpretation that extends $M'$. Intuitively, $M$ and $M'$ are the state sequences of the current state $S$ and some previous state $S'$. Consider a relevant loop $L$ with no false literal in $M$ ($M(\bigvee L) \neq \boldsymbol{f}$). Then also $M'(\bigvee L) \neq \boldsymbol{f}$, and therefore, by Proposition 5.4, $L$ is not an unfounded loop in $M'$ (because there exists a witness for $M'$). Consider a literal $l \in \mathcal{D}_{\text{lits}} \cap L$. Possibly $d_{J_w}(l) \in \mathcal{D}^{\text{ext}}(L)$. If $M(d_{J_w}(l)) = \boldsymbol{f}$ (i.e., we have found a disjunct that has become false since the last search), then $J_w$ does not support $M$ and hence is not a witness for it. It is therefore possible that no witness for $M$ exists, and that $L$ is an unfounded loop in $M$. As such, $l$ is a good candidate cycle source (in state $S$). If on the other hand $M(d_{J_w}(l)) \neq \boldsymbol{f}$, then $J_w$ may still be a witness for $M$. Possibly $J_w$ violates support of $M$ in another literal $l' \in L$, but in that case $l'$ will be a cycle source for the same reason.

This inspires the following definition:

**Definition 5.4** (Cycle sources)**.** Let $M$, $M'$ be interpretations with $M$ an extension of $M'$, and let $J_w$ be a witness for $M'$. Then the set $CS$ of *cycle sources* (with respect to $M$, $M'$ and $J_w$) is defined as $CS = \{l \in \mathcal{D}_{\text{lits}} \mid M(l) \neq \boldsymbol{f} \wedge M'(d_{J_w}(l)) \neq \boldsymbol{f} \wedge M(d_{J_w}(l)) = \boldsymbol{f}\}$.

Let $M$, $M'$, $J_w$ and $CS$ be as in the definition. Then there exists a justification $J_s$ that supports $M$, such that $d_{J_s}(l) = d_{J_w}(l)$ for each $l \in \mathcal{D}_{\text{lits}} \setminus CS$. For this justification, the set $\{l \in \mathcal{D}_{\text{lits}} \mid d_{J_s}(l) \neq d_{J_w}\}$ is equal to $CS$; for other justifications that support $M$, this set is a superset of $CS$.

The discussion leading up to Definition 5.4 can now be summarized by the following result:

**Proposition 5.7.** *Let $M$, $M'$, $J_w$ and $CS$ be as in Definition 5.4. Then any unfounded loop in $M$ contains a literal $cs \in CS$.*

**Example 5.15.** Let $\Delta_{5.15} =$

$$\left\{ \begin{array}{l} p \leftarrow q \vee a \vee b, \\ q \leftarrow p \end{array} \right\},$$

and consider $M_1 = \emptyset$. The justification $J_w$ with $d_{J_w}(p) = a$ is a witness for $M_1$. Consider $M_2 = M_1[b/\boldsymbol{f}]$; since $a$ is still $\boldsymbol{u}$, $J_w$ is also a witness for $M_2$. There are therefore no cycle sources. Finally, consider $M_3 = M_2[a/\boldsymbol{f}] = \{a \mapsto \boldsymbol{f}, b \mapsto \boldsymbol{f}\}$. Then $CS$ with respect to $M_3$, $M_2$ and $J_w$ is the singleton $\{p\}$. Thus, there may be an unfounded loop $L$ in $M_3$ that contains $p$. That is indeed the case for $L = \{p, q\}$ with $\mathcal{D}^{\text{ext}}(L) = \{a, b\}$.

[9] This also requires an *initial* search: this is performed by Algorithm 5.4 for the Simplify rule.

**Algorithm 5.6**: FindWitness($M, J_s, J_w, \Delta$)

**Result**: An unfounded loop in $M$ if it exists; else, the empty set, and $J_s$ is a witness for $M$.

```
1 Let CS := {l ∈ D_lits | d_{J_s}(l) ≠ d_{J_w}(l)};
2 while CS ≠ ∅ do
3     Choose a cs ∈ CS;
4     Let L := the result of Algorithm 5.2 on cs;
5     if L ≠ ∅ then return L;
      % Else:  CS has become strictly smaller
6 return ∅;
```

**Definition 5.5.** (Witness up to $CS$) A justification $J$ is a witness for interpretation $M$ *up to a set $CS$ of literals*, if $J$ supports $M$, and for any non-negative loop in $J$, either all literals in the loop are false in $M$, or the loop contains a literal of $CS$.

Recall also the concept of local witness in a literal: a justification $J$ is a local witness for $M$ in $cs$ if $J$ supports $M$, and $J$ contains no non-negative loops that contain $cs$. Furthermore, recall that Algorithm 5.2, when applied on $cs$, either finds an unfounded loop containing $cs$, or produces a local witness in $cs$.

Thus, if $J$ is a witness for $M$ up to $CS$, and Algorithm 5.2 applied on $cs \in CS$ (and $J$) produces a local witness for $M$ in $cs$, then afterwards, $J$ is a witness for $M$ up to $CS \setminus \{cs\}$. Therefore, by repetitive application of Algorithm 5.2 on all literals of $CS$, either an unfounded loop in $M$ will be found, or a witness for $M$ up to the empty set, i.e., a witness for $M$. When such a witness has been found, there are no more unfounded loops.

The above reasoning requires that Algorithm 5.2 satisfies an additional invariant: namely that $J$ is a witness up to $CS$, where $J$ is the justification on which Algorithm 5.2 makes its local changes. It can be easily verified that this is indeed the case—the algorithm only changes $d_J(l)$ for literals $l \in HP$, in such a way that if any new loops are created, they certainly already contain a cycle source. Actually, we can even strengthen Algorithm 5.2: when *any* cycle source $cs' \in CS$ is being justified ($HP$ may contain other cycle sources than the one on which the search is started), it can be removed from $CS$, and the invariant is still valid. This requires changing Line 19, which now reads "Remove $l$ from $HP$, $L$ and $Q$", to "Remove $l$ from $HP$, $L$, $CS$ and $Q$". Note that in this way, the original cycle source $cs$ itself is also removed from $CS$ when justified. Using this change to Algorithm 5.2, we can now apply the algorithm repetitively for all cycle sources.

Algorithm 5.6 applies this repetitive procedure. It assumes $J_w$ and $J_s$ to be given, where $J_w$ is a witness for the previous state sequence $M'$, and $J_s$ is a supporting justification for the current state sequence $M$. The application of Algorithm 5.2 may change $J_s$, as well as $CS$ in the way indicated above.

Algorithm 5.6 can be understood as an algorithm for the *Full BwLoop prop-*

*agation* strategy. It gets called when a UP-saturated state has been reached; as soon as a loop has been found, the resulting propagations or backtracks are executed, and the UnitProp and UnitPropDef rules can resume. When the algorithm terminates without finding a loop, Decide is applicable in the *Full BwLoop propagation* strategy.

**Proposition 5.8.** *Let $M$, $J_w$ and $J_s$ be as given above. Then Algorithm 5.6, when called on $CS$, either returns a unfounded loop L in $M$, or returns the empty set. In the second case, the resulting justification $J_s$ is a witness for $M$.*

*Remark* 5.4. When Algorithm 5.6 returns the empty set and a witness, this witness can be used the next time cycle sources are needed. Using the most recent witness for this purpose has the advantage that then the produced set of cycle sources is small, since few literals will have become false in the mean time.

*Remark* 5.5. An interesting property of witnesses is that they remain witnesses after backtracking (until a decision literal). Thus, to integrate the above in a strategy such as *Full BwLoop propagation*, it suffices to always store the last obtained witness, $J_w$.

### 5.5.3 The search space $HP$

The search space of Algorithm 5.2 is given by the rules that define the initial set $HP$. Thus, by modifying the first line of the algorithm, we can restrict (or enlarge) the search space. Notice that the search space not only determines which literals will be visited, but also which literals can be used to justify literals (the ones outside the search space).

Firstly, the initial set $HP$ should be restricted to literals that are in the same strongly connected component as $cs$ in the dependency graph. Other literals may also have a path to the cycle source in $J_s$; however, they cannot be in the same cycle as the cycle source. Recall our observations in Section 4.4.3 in this respect.

*Stop-at-cycle-sources variant.* A possible alternative initialization of $HP$ is the set $\{l \mid l$ has a path to $cs$ in $J_s$ on which all literals are non-false in $M$ and $\notin CS$ and in the same strongly connected component as $cs\}$. In other words, the reachability search in $J_s$'s transpose is "cut off" at cycle sources other than $cs$. The idea in this variant is that literals in a path to $cs$ in $J_s$ that does contain another cycle source will be visited anyway when Algorithm 5.2 is called on that cycle source. This definition makes the search space smaller than in the standard definition.

*Dynamic dependency graph variant.* The *dynamic dependency graph* is the dependency graph restricted to non-false literals. In another initialization of $HP$, we set $HP$ to the strongly connected component of $cs$ in the dynamic dependency graph. The idea here is that the strongly connected components have to be searched once, right before Algorithm 5.6 is called, and after this, the initialization of $HP$ in Algorithm 5.2 can be done in constant time.

We now illustrate the effect that the proposed variants have on the behaviour of Algorithm 5.6.

**Example 5.16.** Let $\Delta_{5.16} =$

$$\left\{ \begin{array}{rcl} p_1 & \leftarrow & p_2 \vee a_1 \vee b_1, \\ & \vdots & \\ p_{N-1} & \leftarrow & p_N \vee a_{N-1} \vee b_{N-1}, \\ p_N & \leftarrow & p_1 \vee a_N \vee b_N \end{array} \right\},$$

and let $M = \{a_1 \mapsto \boldsymbol{f}, \dots, a_N \mapsto \boldsymbol{f}\}$. Suppose $J_w$ and $J_s$ are such that Algorithm 5.6 finds $CS = \{p_1, \dots, p_N\}$. It calls Algorithm 5.2 on each $p_i$, whereby in the standard variant $HP = \{p_1, \dots, p_N\}$ is calculated every time (which costs linear time)—and the rest of the algorithm merely sets $d_{J_s}(p_i) := b_i$ and returns the empty set. Thus, it takes $\mathcal{O}(N^2)$ time to find a witness.

When Algorithm 5.6 instead uses the Dynamic dependency graph variant, the strongly connected component $\{p_1, \dots, p_N\}$ in the dynamic dependency graph is computed only once, in $\mathcal{O}(N)$ time; the initialization of $HP$ to this set is thereafter done in constant time $N$ times. The same witness is therefore found in $\mathcal{O}(N)$ time. When instead the Stop-at-cycle-sources variant is used, $HP$ is each time initialized to the empty set, which also takes constant time, and again the same witness is found in $\mathcal{O}(N)$ time.

The difference in behaviour between the Dynamic dependency graph variant and the Stop-at-cycle-sources variant is illustrated in next example.

**Example 5.17.** Example 5.16 continued. Let $M' = M[\{b_1, \dots, b_N\}/\boldsymbol{f}]$. Again $CS = \{p_1, \dots, p_N\}$. Suppose the Stop-at-cycle-sources variant is used, and the cycle sources are processed in the order as given here. Then for $p_1$, $HP$ is initialized to $\emptyset$ (because $p_N \in CS$); Algorithm 5.2 removes $p_1$ from $CS$. Next, for $p_2$, $HP$ is initialized to $\{p_1\}$; Algorithm 5.2 removes also $p_2$ from $CS$. This continues, until for $p_N$, $HP$ is initialized to $\{p_1, \dots, p_N\}$. Then $p_N \in HP$, and Algorithm 5.2 finds the unfounded loop $\{p_1, \dots, p_N\}$. The whole process takes $\mathcal{O}(N^2)$ time.

Suppose on the other hand that the Dynamic dependency graph variant is used: then (for arbitrary first cycle source), $HP$ is initialized to $\{p_1, \dots, p_N\}$, and the same unfounded loop is found in $\mathcal{O}(N)$ time.

The Dynamic dependency graph variant is the only variant that ensures that Algorithm 5.6 runs in linear time (because of its constant-time initialization of $HP$), as is illustrated by these examples. On the other hand, in this variant the search space of Algorithm 5.2 is much larger than in the other variants.

### 5.5.4 Remarks and related work

There is still much room for variation and optimization in Algorithm 5.2. We gather some detailed remarks regarding such individual ideas in this section, and point to related work.

*Remark* 5.6 (Stable variant, unfounded sets). Algorithm 5.2 can trivially be adapted to become an algorithm for the *Stable loop propagation* strategy. It

should then only find positive loops, which can be achieved by limiting the search to atoms. Again, this is achieved by simply modifying the first line of Algorithm 5.2, where $HP$ is initialized.

The Stable variant can be understood as an algorithm for finding *unfounded sets*. Anger et al. (2006b) published an algorithm that seems closely related to the Stable variant of Algorithm 5.2; their algorithm was used in the CLASP solver by Gebser et al. (2007a). In particular, the following sets have similar functionalities: $\mathsf{Set} \sim L$, $\mathsf{Ext} \sim Q$, $\mathsf{Sink} \sim vocab(\Psi \cup \{\Delta\}) \setminus HP$. The precise relation between $\mathsf{Source}$ and $CS$ is unclear.

SMODELS' Atmost algorithm (Simons, 2000) finds the *greatest* unfounded set with respect to a given three-valued interpretation. It is also a linear-time algorithm (in the size of the largest strongly connect component). This greatest unfounded set may not be a relevant loop itself, but instead contain literals from more than one unfounded loop, as well as literals that depend on such loops. Algorithm 5.4 in Section 5.4.5 also finds the greatest unfounded set.

As also remarked by Anger et al. (2006b), it is often more efficient to find small unfounded sets and let the rest of the computation be handled by unit propagation, than it is to find the greatest unfounded set.

*Remark* 5.7 (Justifying conjunctive literals). For completeness of Algorithm 5.2 (i.e., all possible ways of justifying $cs$ are tried), it is required that all non-false body literals of all $HP$ literals are considered for justification. However, there is no requirement as to when, or in what order, they are considered. For a literal $l \in \mathcal{D}_{\text{lits}}$, any literal in $\varphi_l$ offers a chance to justify $l$, and it is therefore advantageous to consider them all as soon as possible. Therefore all non-false literals in $\varphi_l$ are pushed on $Q$ together (unless one of them is justified). For a literal $l \in \mathcal{C}_{\text{lits}}$ however, even if some literal $l' \in \varphi_l$ gets justified, there may still be some $l'' \in \varphi_l$ not justified yet. It is therefore advantageous to consider them one by one. One way to achieve this is to push one literal $l'$ of $\varphi_l$ on $Q$, and designate it as the *guard* of $l$. When at some later time $l'$ is justified, all literals for which it is guard are pushed on $Q$ for re-evaluation. Using this process, literals $l \in \mathcal{D}_{\text{lits}}$ are pushed on $Q$ at most once, literals $l \in \mathcal{C}_{\text{lits}}$ are pushed on $Q$ at most $|\varphi_l|$ times. This is the version we have implemented in MINISAT(ID) (cf. Section 5.7); the exact algorithm is therefore given in Appendix C.

We illustrate the potential advantage of this system. Consider the definition

$$\left\{ \begin{array}{ll} cs \leftarrow p \vee q, & r_1 \leftarrow cs, \\ p \leftarrow r_1 \wedge \cdots \wedge r_N, & r_2 \leftarrow cs, \\ q \leftarrow s, & \ldots \\ s \leftarrow cs \vee a, & r_N \leftarrow cs \end{array} \right\},$$

and suppose that $cs$ is a cycle source. Let $M = \emptyset$, and let $J_s$ have $d_{J_s}(s) = cs$ and $d_{J_s}(cs) = q$. Algorithm 5.2 first constructs $HP$ in the transpose of $J_s$: it adds $s$, $q$ and $cs$ to $HP$, as well as all $r_i$'s and $p$. It finds $HP = \{cs, p, q, r_1, \ldots, r_N, s\}$. In the first iteration of the while-loop, it pushes $p$ and $q$ on the queue (assume in that order). In the next iteration, $p$ is treated: depending on which variant we have chosen, either all the $r_i$ are pushed on the queue, or one of them, which

is designated the guard of $p$. In the next iteration, $q$ is treated; $s$ is pushed on the queue. Depending on the chosen variant, the queue is now $\langle r_1 \ldots r_N\, s\rangle$, or $\langle r_i\, s\rangle$, so we now have either $N$ iterations, or one iteration, treating $r_i$ atoms (in vain). In the final iteration, $s$ is treated; the algorithm then sets $d_{J_s}(s) = a$, and calls `Justify(`$s$`)`. This in return calls `Justify(`$q$`)`, which sets $d_{J_s}(cs) = q$ and calls `Justify(`$cs$`)`, and terminates.

Suppose on the other hand that $M = \{a \mapsto \boldsymbol{f}\}$. Algorithm 5.2 called on $cs$ then proceeds exactly the same as before, and adds either only one $r_i$, or all of the $r_i$ to $L$, depending on the variant; after $s$ is treated, the queue is empty, and the algorithm returns either $\{cs, p, q, r_i, s\}$, or $\{cs, p, q, r_1, \ldots, r_N, s\}$.

*Remark* 5.8 (Symmetric variant). A general graph technique for finding the strongly connected component containing a node $N$ (and therefore, a superset of all loops containing $N$), is by taking the intersection of all nodes reachable from $N$ with all nodes reachable from $N$ in the transpose of the graph. Algorithm 5.2 applies a similar technique. However, the algorithm is concerned with finding *some* loop, not necessarily the greatest one: consequently it restricts the search by considering not the transpose of the dependency graph, but the transpose of $J_s$: a subgraph of it.

Given this observation, one can see that the symmetric version of Algorithm 5.2 is also correct: initially marking the $HP$ literals in $J_s$ (instead of in its transpose), and then searching in the transpose of the dependency graph.

*Remark* 5.9 (Source pointers). The search is restricted by considering not the transpose of the dependency graph, but the transpose of $J_s$. In this respect, the purpose of $J_s$ is comparable to that of the so-called *source pointers* in SMODELS' Atmost algorithm (Simons, 2000).

*Remark* 5.10 (Loop-*free* justifications). When Algorithm 5.2 returns a loop, $J_s$ certainly contains a cycle through $cs$. However, after BwLoop is applied, $cs$ is false. Hence, three-valued support then remains valid if $d_{J_s}(cs)$ is changed back to $d_{J_w}(cs)$, i.e., the literal that originally became false and caused $cs$ to be a cycle source. The advantage of this is that the resulting justification $J_s$ does not contain a cycle through $cs$. Especially in the case of the stable variant, and when working on a loop-simplified definition, this is a useful way of finding loop-*free* (instead of merely loop-safe) witnesses.

*Remark* 5.11 (AddLF). Note that the AddLF transition rule under the *Incremental* strategy can be implemented using Algorithm 5.2: the only additional requirement is that $M(\bigvee L) = \boldsymbol{t}$. Unless this additional requirement can be exploited to produce a more efficient algorithm, this suggests that BwLoop is more useful than AddLF under the *Incremental* strategy.

## 5.6 Enhancements to DPLL

Contemporary DPLL-based SAT solvers' algorithms are far more sophisticated than the DPLL strategy as outlined in Section 5.2.2. Mitchell (2005) offers an overview of enhancements to DPLL that are implemented in most of today's

Backjump:

$$M\, l^{\mathsf{d}}\, N \parallel \Psi \quad \Longrightarrow M\, l' \parallel \Psi \quad \textbf{if} \begin{cases} M(l') = \boldsymbol{u} \\ \text{There is a clause } (\varphi \vee l') \text{ such that} \\ \qquad \Psi \models (\varphi \vee l') \text{ and } M(\varphi) = \boldsymbol{f} \end{cases}$$

Learn:

$$M \parallel \Psi \quad \Longrightarrow M \parallel \Psi, \varphi \quad \textbf{if} \begin{cases} \Psi \models \varphi \\ \varphi \notin \Psi \end{cases}$$

Forget:

$$M \parallel \Psi, \varphi \quad \Longrightarrow M \parallel \Psi \quad \textbf{if} \begin{cases} \Psi \models \varphi \end{cases}$$

Restart:

$$M\, l^{\mathsf{d}}\, N \parallel \Psi \quad \Longrightarrow M \parallel \Psi \quad \textbf{if} \begin{cases} M \text{ contains no decision literals} \end{cases}$$

Figure 5.7: Enhancements to DPLL

DPLL-based SAT solvers: clause learning, backjumping and restarts, adapted heuristics, and pre-processing techniques. In this section we give a brief account of these enhancements.

### 5.6.1 Clause learning, backjumping and restarts

The idea of *clause learning* and *backjumping* originates in the constraint satisfaction domain (Prosser, 1993), but was first introduced in SAT by Marques-Silva and Sakallah (1999). The idea can be modelled using the Learn and Backjump transition rules given in Figure 5.7: firstly, when a clause $(\varphi \vee l')$ is a consequence of the given theory, it can be added to it; secondly, if the current state sequence falsifies that clause, one can backtrack till an interpretation $M$ for which $M(\varphi) = \boldsymbol{f}$ and $M(l') = \boldsymbol{u}$, and add $l'$ to $M$. Note that the behaviour of the Backjump rule can be described in terms of Learn, Backtrack and UnitProp: if $M\, l^{\mathsf{d}}\, N \parallel \Psi \overset{\mathsf{Backjump}}{\Longrightarrow} M\, l' \parallel \Psi$, then $M\, l^{\mathsf{d}}\, N \parallel \Psi \overset{\mathsf{Learn}}{\Longrightarrow} M\, l^{\mathsf{d}}\, N \parallel \Psi, (\varphi \vee l') \overset{\mathsf{Backtrack}^*}{\Longrightarrow} M \parallel \Psi, (\varphi \vee l') \overset{\mathsf{UnitProp}}{\Longrightarrow} M\, l' \parallel \Psi, (\varphi \vee l')$. In principle, these rules are valid throughout a derivation; in practice, they are invoked after a conflict, and the learned clause is derived using a resolution process that starts from the conflicting clause. For this reason—more precisely: using this strategy—the techniques are often called "conflict-driven" clause learning and backjumping.

Classic DPLL can be described as performing a tree-like resolution process:

by adding clause learning, additional forms of resolution are obtained. Beame et al. (2004) have shown that DPLL enhanced with conflict-driven clause learning is more powerful then regular resolution, which in turn is more powerful than classic DPLL. This gives a theoretical reason for the success of conflict-driven clause learning SAT solvers.

Since the Learn transition rule can add unlimited numbers of clauses, memory management can become problematic. Therefore modern solvers also implement the Forget rule. They deploy heuristics to determine the "usefulness" of a clause: the likelihood that it will contribute to the future search process. Clauses are then removed from the theory when their usefulness drops below a predefined threshold. They also limit the applicability of the Forget rule to clauses that have previously been learned, thus making the requirement $\Psi \models \varphi$ trivial to verify.

Clause learning can also be seen as a way of acquiring new information—and thereby better-informed heuristics—as the search process unfolds. The Restart rule acts on this observation by enabling the search process to simply restart with an empty partial assignment (more precisely, with the state sequence up to the first decision literal), in the hope that the new search will make more useful choices for the Decide rule. Experiments have proven this technique to increase robustness (e.g., Biere, 2008).

Termination and completeness are affected by the addition of these rules. Nieuwenhuis et al. (2006) provides an overview of viable strategies and proofs of termination and completeness.

### 5.6.2 Heuristics

Decision heuristics determine on which literal to apply the Decide rule; the choice of heuristics can make or break an effective SAT solver. Most heuristics that perform well for conflict-driven clause learning solvers are geared towards favouring literals which have appeared in recently derived conflict clauses. Examples include the Variable Move To Front heuristic (Ryan, 2004), the Variable State Independent Decaying Sum heuristic (Moskewicz et al., 2001), and the BerkMin heuristic (Goldberg and Novikov, 2002). Also the Forget and Restart rules are subject to heuristics.

It is an open question whether special-purpose decision heuristics tuned to SAT(ID) can yield improved performance on practical instances (as opposed to those for SAT). An obvious concern is that a good decision heuristic for SAT(ID) should behave on the theory $\Psi, \Delta$ as a good decision heuristic for SAT would on the theory $\Psi, comp(\Delta), LF_\Delta$. This can be achieved if, whenever a conflict related to $\Delta$ is found, the clauses of $comp(\Delta)$ and $LF_\Delta$ that are conflicting are added to the theory, and the chosen SAT heuristic is used on the result.

The SMODELS solver for ASP uses "lookahead" propagation; the information obtained during the lookahead process can be used to calculate the size of the remaining search spaces after individual choices. This is an interesting alternative heuristic; however, the lookahead and clause learning techniques diminish each other's returns and are therefore not often used together.

### 5.6.3 Preprocessing

Preprocessing techniques for SAT try to derive, from a given propositional theory $\Psi$, a *smaller* theory $\Psi'$, such that $\Psi' \models \Psi$. This derivation is done in a preprocessing phase, and can therefore be more elaborate than propagation algorithms are; the idea is that the effort pays off because the SAT solving time on $\Psi'$ is likely to be smaller than that on $\Psi$.

A simple preprocessing technique is the PureLit rule. More elaborate techniques are described a.o. by Eén and Biere (2005) and are based on general resolution; their purpose is both to remove redundant variables and to remove redundant clauses.

In our work, the purpose of a SAT(ID) solver is as a tool to solve model expansion problems for ID-logic theories. This requires theory equivalence instead of satisfiability equivalence; consequently, many standard SAT preprocessing techniques such as pure literal removal are not applicable even to the CNF part of a given ECNF theory.

## 5.7 MiniSat(ID)

We have implemented a SAT(ID) solver on the basis of an existing open source SAT solver, namely MiniSat (Eén and Sörensson, 2003). The resulting solver is called MiniSat(ID) (Mariën et al., 2008) and is available for download (Mariën, 2008). MiniSat(ID) can be considered the successor of MidL (Mariën et al., 2007a,b), our earlier SAT(ID) solver, which operated according to the same strategy.

This section introduces the solving strategy and implementation details of MiniSat(ID), and reports on its performance.

### 5.7.1 Transition system and strategy

MiniSat(ID)'s transition system is given by Figures 5.8–5.9. Its strategy is as follows:

- Initially, UnitProp and UnitPropDef are applied until a UP-saturated state is reached (or Fail was applicable), after which Simplify (or SimplifyFail) is applied.
- Whenever possible, $[\mathsf{UnitProp}]^{\mathsf{BF+}}$ and $[\mathsf{UnitPropDef}]^{\mathsf{BF+}}$ are applied.
- In UP-saturated states, $[\mathsf{BwLoop}]^{\mathsf{BF+}}$ is applied *on positive loops* whenever possible. Also, when a loop on which to apply $[\mathsf{BwLoop}]^{\mathsf{BF+}}$ is found, AddLF is applied for that loop. We call a UP-saturated state where stable $[\mathsf{BwLoop}]^{\mathsf{BF+}}$ is no longer applicable *UPB-saturated*.
- Whenever any of the above leads to a conflict, Learn and Backjump are applied. Also Forget may be applied after Backjump. When a specified (parameterized) number of conflicts has been reached, Restart is applied, and the parameter is increased.

Decide:

$$M \parallel \Psi, \Delta \quad \Longrightarrow M\, l^{\mathsf{d}} \parallel \Psi, \Delta \quad \textbf{if} \begin{cases} M(l) = \boldsymbol{u} \end{cases}$$

UnitProp:

$$M \parallel \Psi, \Delta \quad \Longrightarrow M\, l \parallel \Psi, \Delta \quad \textbf{if} \begin{cases} M(l) = \boldsymbol{u} \\ (\varphi \vee l) \in \Psi \text{ with } M(\varphi) = \boldsymbol{f} \end{cases}$$

[UnitProp]$^{\mathsf{BF}}$

UnitPropDef:

$$M \parallel \Psi, \Delta \quad \Longrightarrow M\, l \parallel \Psi, \Delta \quad \textbf{if} \begin{cases} M(l) = \boldsymbol{u} \\ (\varphi \vee l) \in comp(\Delta) \text{ with } M(\varphi) = \boldsymbol{f} \end{cases}$$

[UnitPropDef]$^{\mathsf{BF}}$

BwLoop:

$$M \parallel \Psi, \Delta \quad \Longrightarrow M\, \neg l \parallel \Psi, \Delta \quad \textbf{if} \begin{cases} M(l) = \boldsymbol{u} \\ \Delta \text{ contains a relevant loop } L \\ \text{such that } M(\bigvee \mathcal{D}^{\text{ext}}(L)) = \boldsymbol{f} \\ \qquad\qquad\qquad \text{and } l \in L \end{cases}$$

[BwLoop]$^{\mathsf{BF}}$

AddLF:

$$M \parallel \Psi, \Delta \quad \Longrightarrow M \parallel \Psi, \Delta, LF \quad \textbf{if} \begin{cases} LF = \neg l \vee \bigvee \mathcal{D}^{\text{ext}}(L) \\ LF \notin \Psi \\ L \text{ is a relevant loop in } \Delta, l \in L \end{cases}$$

Figure 5.8: The MiniSat(ID) transition system, part I

- Decide is applied when none of the above is applicable.
- In UPB-saturated states where none of the above is applicable (i.e., the state sequence is two-valued), [BwLoop]$^{\mathsf{BF+}}$ is applied on mixed loops, if possible.

We offer some observations and explanations about this strategy:

- It is a *stable* strategy. The justifications used during the main search are stable justifications.
- When the given PC(ID) theory is total, there are no final mixed loop propagations, as explained before. Then if the state sequence of a UPB-saturated state is two-valued, it is a model of the theory. For non-total

Simplify:

$$M \parallel \Psi, \Delta \quad \Longrightarrow M \parallel \Psi, \Delta^{\varnothing[M]} \quad \textbf{if} \begin{cases} \circlearrowleft[M] \text{ is consistent} \\ M \text{ contains no decision literals} \end{cases}$$

SimplifyFail:

$$M \parallel \Psi, \Delta \quad \Longrightarrow FailState \quad \textbf{if} \begin{cases} \circlearrowleft[M] \text{ is inconsistent} \\ M \text{ contains no decision literals} \end{cases}$$

Backjump:

$$M\, l^{\mathsf{d}}\, N \parallel \Psi, \Delta \quad \Longrightarrow M\, l' \parallel \Psi, \Delta \quad \textbf{if} \begin{cases} M(l') = \boldsymbol{u} \\ \text{There is a clause } (\varphi \vee l') \\ \text{such that } \Psi, \Delta \models (\varphi \vee l') \\ \qquad \text{and } M(\varphi) = \boldsymbol{f} \end{cases}$$

Learn:

$$M \parallel \Psi, \Delta \quad \Longrightarrow M \parallel \Psi, \varphi, \Delta \quad \textbf{if} \begin{cases} \Psi, \Delta \models \varphi \\ \varphi \notin \Psi \end{cases}$$

Forget:

$$M \parallel \Psi, \varphi, \Delta \quad \Longrightarrow M \parallel \Psi, \Delta \quad \textbf{if} \left\{ \Psi, \Delta \models \varphi \right.$$

Restart:

$$M\, l^{\mathsf{d}}\, N \parallel \Psi, \Delta \quad \Longrightarrow M \parallel \Psi, \Delta \quad \textbf{if} \left\{ M \text{ contains no decision literals} \right.$$

Figure 5.9: The MiniSat(ID) transition system, part II

theories, the final mixed loop propagations are executed using the algorithm for Simplify, Algorithm 5.4. Since in general the solver does not know whether the given theory is total, this final algorithm can be considered as a totality test.[10]

- There exists a stable witness for any UPB-saturated state sequence: MiniSat(ID) stores such a witness. It also ensures that this stable witness is loop-*free* (which is possible since Simplify has been executed).
- Performing AddLF whenever $[\mathsf{BwLoop}]^{\mathsf{BF}+}$ is applicable ensures that a standard SAT algorithm for Learn can be used, such as the "First Unique

[10]If a stable witness could always be *extended* to a general witness by simply adding direct justifications for negative literals in $\mathcal{D}_{\text{lits}}$, then we could simplify Algorithm 5.4. However, this is not the case.

Implication Point" scheme introduced by Zhang et al. (2001), and implemented in MiniSat. Also observe that when a loop formula for loop $L$ is added to the theory, and backtracking occurs, the subsequent unit propagations may do the work of FwLoop for $L$, or they may again do the work of BwLoop for $L$. Note that the smaller the loops that are found, the more likely they are to be used again, either for a loop propagation or for the derivation of a new learned clause.

### 5.7.2 Motivation of this strategy

The motivation of this strategy derives from observations regarding both the types of problems encountered in practice, and the behaviour of some different strategies on such problems. We did a number of comparisons of different strategies, and give a detailed report on them in Section 5.7.4. Here we refer to those comparisons to substantiate our motivation.

**Why SAT(ID) solving at all?** Mitchell and Ternovska (2005) proved that all NP search problems can be cast as parameterized FO model expansion problems (see Section 3.2). It therefore follows that to solve such problems, inductive definitions are strictly speaking not needed.

However, the modelling of problems may have a major impact on how efficient they are solved *in practice*. When ID-logic can be used as modelling language, the "practical expressivity gain" (over FO) may be highly relevant, both in terms of modelling convenience and in terms of solving time. We compare MiniSat solving times of an FO encoding of the Hamiltonian circuit problem with MiniSat(ID) solving times of an ID-logic encoding of it in Comparison 5.1; the former cannot compete with the latter.

Observe that an FO-$MX$ encoding of a problem may require extra vocabulary compared to an ID-logic-$MX$ encoding of the same problem.

**Why not an eager technique?** In a reduction to SAT strategy, one gets all the advantages of the best off-the-shelf SAT solvers for free. This is a strong motivation to try such a strategy.

Observe that Lifschitz and Razborov's (2006) result regarding the number of loop formulas impedes the use of a vocabulary preserving reduction to SAT. To make the eager technique efficient, the reduction therefore has to introduce new vocabulary. This was done for SAT(ID) by Pelov and Ternovska (2005), and for ASP by Janhunen (2004). In Comparison 5.2 we compare an implementation of Pelov and Ternovska's reduction coupled with MiniSat to MiniSat(ID). Again the former cannot compete with the latter.

**Why a stable strategy?** Recall from Example 5.8 that even for the class of total theories, *Full loop propagation* is more decide-efficient then *Stable loop propagation*. Notwithstanding this result, we chose a stable strategy. The motivation is that

1. in a stable strategy, the search space of Algorithms 5.2 and 5.4 alike is smaller than in a full strategy;
2. there seem to be very few practical examples where a propagation as illustrated in Example 5.8 can occur; and
3. even in the examples where such propagation can occur, these extra propagations do not outweigh the extra computational effort due to the larger search spaces (larger graphs in which to search for loops).

**Why no exhaustive Δ-propagation?** The Δ-Propagate transition rule, implemented by Algorithm 5.3, offers exhaustive Δ-propagation. A simple experiment illustrated that the gain in propagations (compared to our strategy of UnitPropDef and BwLoop propagations) is eclipsed by the extra computational effort, cf. Comparison 5.3.

### 5.7.3 Implementation details

One of the purposes of the MiniSat(ID) implementation was to illustrate that SAT(ID) is in a practical sense an extension of SAT, by *extending* an existing SAT solver with inductive definition propagation mechanisms. True to this purpose, the original code of MiniSat has not been modified beyond extra code for command-line parsing, parsing of ECNF theories, and calls to the new propagation methods.

We now present some implementation details of MiniSat(ID).

#### Global structure

MiniSat(ID)'s additions to MiniSat are the following:

1. data structures to represent a definition, as well as two stable justifications, which we will call $J_s$ and $J_w$ (cf. Definition 5.4; $J_w$ represents a witness for an earlier interpretation, $J_s$ a supporting justification for the current interpretation);
2. code to parse an ECNF theory and initialize the above data structures;
3. code to perform the definition-based propagations: Simplify and BwLoop.

The definition $\Delta$ is represented using the CNF encoding of $comp(\Delta)$; recall Figure 5.2 on page 90. Recall that each rule $r \in \Delta$ is represented by a "long clause" $C(r) \in comp(\Delta)$. Additionally, one array says of each defined atom whether it is in $\mathcal{D}_{\text{lits}}$ or in $\mathcal{C}_{\text{lits}}$, and one array maps each defined atom to the clause of $comp(\Delta)$ representing its rule.

An extra array maps each defined atom to its strongly connected component in the dependency graph, where the components are represented by unique numbers.

The justifications $J_s$ and $J_w$ are represented using an array that maps defined atoms in $\mathcal{D}_{\text{lits}}$ to their child. Also their transpose is represented, using an array that maps literals to arrays of defined literals.

MiniSat has a method called `simplify()`, which is called after the initial propagations, and before the first decision literal is picked. In MiniSat(ID), this method contains extra code that implements the Simplify algorithm, and initializes $J_w$.

MiniSat's main propagation method is called `propagate()`; it performs $[\mathsf{UnitProp}]^{\mathsf{BF+}}$. Since $comp(\Delta)$ is present in the CNF theory, this method automatically performs $[\mathsf{UnitPropDef}]^{\mathsf{BF+}}$ as well. In MiniSat(ID), an extra call to a method that implements $[\mathsf{BwLoop}]^{\mathsf{BF+}}$ is added at the end of `propagate()`, in such a way that unit propagations will resume if (as soon as) $[\mathsf{BwLoop}]^{\mathsf{BF+}}$ is able to propagate something. When `propagate()` terminates, a UPB-saturated state has been reached, or a conflict has been found.

We further explain the details of the call to the $[\mathsf{BwLoop}]^{\mathsf{BF+}}$ algorithm.

### MiniSat(ID) implementation of the $[\mathsf{BwLoop}]^{\mathsf{BF+}}$ algorithm

In MiniSat, unit propagation is implemented using the two watched literals technique. The concrete implementation is such that it is easy to retrieve from a clause the two literals in it that are watched: they are placed in front; if one of them is true, it is placed first. Let $w_1(C)$, $w_2(C)$ denote the first and second watched literal of a clause $C$.

In order to construct a supporting stable justification, MiniSat(ID) exploits the two watched literals of the clauses $C(r)$. Consider the following definition of a stable justification $J_s$. For each atom $d \in \mathcal{D}_{\text{lits}}$, defined by rule $r$, let $d_{J_s}(d)$ be $w_1(C(r))$, unless that literal is $\neg d$ (because that is the head literal), then let it be $w_2(C(r))$.[11] It is easy to verify that in a UP-saturated state, i.e., when the two watched literals invariant is satisfied for each clause, this justification $J_s$ three-valuedly supports the state.

The $J_w$ stable justification is constructed to be a stable witness of the last UPB-saturated state. It is loop-free. Even if backtracking occurs to a state before the last UPB-saturated state, $J_w$ is still a witness. When a new decision literal has been chosen, and a new UP-saturated state has been reached, the above technique is used to find $J_s$, and a set of cycle sources $CS$ is derived as the set of atoms $a$ for which $J_w(a) \neq J_s(a)$.

For every cycle source, Algorithm 5.2 is called (it may modify $J_s$); if it returns a loop, the loop formula is added to the theory, the $[\mathsf{BwLoop}]^{\mathsf{BF+}}$ propagations are performed, and the `propagate()` method resumes. When all cycle sources have been processed without returning a loop, the final $J_s$ is copied to $J_w$—it is then a stable witness.

[11]Note that if $\neg d \in \varphi_d$, this definition excludes the body literal $\neg d$ as potential child of $d$ in $J_s$; however, extending such a stable justification to a general one can never yield a witness.

### 5.7.4 Evaluation

**Discussion of the problems and evaluation setup**

We discuss the different types of problems on which we have performed our evaluations. The IDP encodings of these problems are given in Appendix B. The ASP encodings of the same problems are taken from the Asparagus benchmark website;[12] we have tried to ensure that the IDP and ASP encodings are as similar as possible.

*Hamiltonian circuits* We discussed the Hamiltonian circuit problem in Section 3.3.2. The instances used here are

- random Hamiltonian graphs with sizes (number of vertices, number of edges) in the categories: $(100, 200(\pm 50))$, $(200, 600(\pm 100))$, $(300, 1200(\pm 200))$, $(400, 2000(\pm 400))$, $(500, 3000(\pm 500))$;
- non-Hamiltonian graphs with sizes (39, 760), (59, 1740), (79, 3120); these are handcrafted with a random component.

For each category, there are 10 instances. Observe that the problem statement declares that each vertex must be reached: therefore in every instance of this problem, all atoms of $Def(\Delta)$ will be true.

*Sokoban puzzles* The Sokoban puzzle is a planning problem. A maze is given, containing a number of boxes, the same number of goal positions, and a man. The man can move around in the maze and push blocks, with some limitations; the goal is to find a sequence of moves and pushes that results in all boxes being on a goal position.

An ID-logic encoding of this problem may include a non-monotonic inductive definition of *event calculus* (Kowalski and Sergot, 1986). However, this is a definition over the well-founded order of time. As a consequence, the grounding of this definition is non-recursive.

The encoding we use, however, reduces the length of the plan, by representing each sequence of moves followed by a sequence of pushes of one block in a given direction as one step. The reduction in number of timesteps needed to find a solution is considerable, and makes grounding and solving much more efficient. However, this representation requires an inductive definition: in order for a particular push action to be possible at any given timestep, the man should be able to reach the point from which that action is to start.

The propositional definitions resulting from this representation are highly irregular. I.e., the number of literals in rule bodies varies greatly.

All instances of this problem are also taken from the Asparagus benchmark website (all “Dimitri Yorick” and “Duthen” instances).

[12] `http://asparagus.cs.uni-potsdam.de/`

*Hitori puzzles* A Hitori puzzle is combinatorial search problem. A given puzzle is a grid, filled with numbers. A solution is a black or white colouring of all grid squares, such that black squares do not neighbour; each number occurs at most once in the white squares of any row and column; and all the white cells are contiguous. The last constraint is expressed in ID-logic by an inductive definition.

All instances of this problem are handcrafted with a random component and have a grid of size $50 \times 50$.

All instances were chosen so as to be difficult enough, but feasible to solve in a 10-minute timespan. The different Hamiltonian circuit instances vary considerably in size; for the other two problem types all instances have comparable sizes.

Observe that each of these problems contains an inductive definition. On problems without inductive definitions (i.e. SAT problems), MiniSat(ID) has the same behaviour as MiniSat. For the behaviour of MiniSat on SAT problems, we point out that MiniSat was the best performing solver on the last SAT-Race (Sinz, 2008). It also has been top solver many times in various competitions, such as the 2006 SAT-Race and the 2007 SAT competition on industrial and handcrafted instances.

We now discuss the setup of our evaluation, and how the results are to be interpreted.

The platform on which the experiments were run is an Intel Core 2 Duo 3GHz with 2GB RAM running Kubuntu 7.10 Linux. We used a timeout value of 10 minutes per instance. The version of MiniSat(ID) used throughout the comparison here is v1.3b.

In all our comparison figures, each instance corresponds to one mark on the figure. We always compare two algorithms or programs versus one another. The parameter being measured is indicated over the figure. For instance, if the parameter is runtime, then marks that are above the diagonal represent instances that were solved faster by the $x$-axis algorithm (horizontally written) then by the $y$-axis algorithm (vertically written).

In many of the figures, groups of instances are clustered. This means that the value being measured, e.g. runtime, is similar (per algorithm or solver) for each of the instances of such a cluster. This happens most often with Hitori and with Sokoban instances, since these all have comparable sizes.

The indication "TO" in the legend means "timeout"; each instance is marked with a different symbol according to whether neither of the algorithms, only the $x$-axis algorithm, only the $y$-axis algorithm, or both yielded timeout on it.

### Comparisons of different strategies

**Comparison 5.1** (The need for SAT(ID)) In Section 5.7.2 we explained that all problems in NP can be solved by parameterized model expansion for FO, i.e., that inductive definitions are, strictly speaking, not needed.

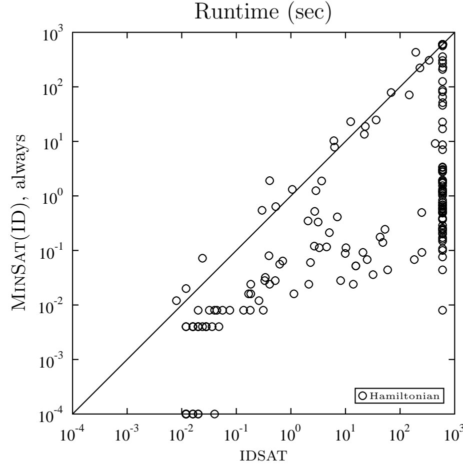



Figure 5.10: Comparison of MiniSat(ID) and idsat

We made an encoding of the Hamiltonian circuit problem in FO-$MX$ (by using a predefined total order on the vertices), and compared it to the ID-logic-$MX$ encoding. To illustrate the importance of using inductive definitions, the comparison on one instance suffices; our observations are typical for most instances. On the smallest graph of our data set (100 edges, 150 vertices), the PC(ID) file resulting from the ID-logic encoding contained a total of 1,176 literal instances; solving the instance with MiniSat(ID) took 0.004 seconds. On the same instance, the PC file resulting from the FO encoding contained $7.9 \times 10^6$ literal instances; MiniSat could not solve the instance within the imposed time limit of 10 minutes.

**Comparison 5.2** (Reduction to SAT vs. SAT(ID)) Pelov and Ternovska (2005) implemented their reduction of PC(ID) to PC in idsat. We compared the efficiency of MiniSat on the resulting files (i.e., idsat's reduction is not included) with that of MiniSat(ID) on the original PC(ID) files for Hamiltonian circuit problems. The result is given in Figure 5.10.[13]

We observe that the majority of instances were solved faster by MiniSat(ID); often by a factor of more then 10.

**Comparison 5.3** (On exhaustive $\Delta$-propagation) We tested the exhaustive $\Delta$-propagation strategy by adding an implementation of Algorithm 5.3 to MiniSat(ID). We compared it experimentally to the standard MiniSat(ID) version.

[13]Version 0.9.5 of idsat was used, and version 2 of MiniSat.

Our comparison was done on Hamiltonian circuit problems: on these problems it is expected that FwLoop is often applicable, hence our test was geared towards showing the exhaustive strategy from its most positive side. This is because all $Def(\Delta)$ atoms are asserted to be true in this problem encoding.

While the number of conflicts needed to find a model often (but not consistently) dropped by a factor of 10, the propagation speed dropped (consistently) by a factor of 100—from 1 to 2 million to a mere 10,000 to 20,000 propagations per second. Accordingly, the total runtime increased from an average of 1.1s to 47s. Recall that Algorithm 5.3 works by repetitive calls to Algorithm 5.2: less then 0.02% of these calls resulted in a propagation. Such propagations made out 0.1–0.2% of the total number of propagations.

**Comparison 5.4** (Normal vs. symmetric version of Algorithm 5.2) We did not implement the symmetric variant of Algorithm 5.2, whereby the search is done in the transpose of the dependency graph (cf. Remark 5.8). However, we measured the size of the initial search space (number of $HP$ literals in Line 1 of the algorithm), and compared it to the search space in the standard version. The search space is the deciding factor regarding the efficiency of the algorithm.

Though the ratio of these two sizes of search spaces greatly varied from one individual run of Algorithm 5.2 to the next, the *average* ratio was very close to one. This suggests that in practice there are no important differences between Algorithm 5.2 and its symmetric version.

**Comparison 5.5** (On the use of *guards*) MiniSat(ID)'s implementation of Algorithm 5.2 deploys a system of *guards* for atoms in $\mathcal{C}_{\text{lits}}$ that are waiting to be justified (cf. Remark 5.7 and Appendix C). We also experimented with an implementation as given in Algorithm 5.2. On some problems, such as Hamiltonian circuit problems, we obtained the exact same behaviour with as without the guards system, since in such problems all atoms in $\mathcal{C}_{\text{lits}}$ have exactly one $HP$ literal in their body. On other problems, we observed a slight advantage for the system with guards.

**Comparison 5.6** (Standard variant vs. Dynamic dependency graph variant) In Section 5.5.3 we explained that the Dynamic dependency graph variant of Algorithm 5.2 is the only variant that ensures linear behaviour when Algorithm 5.2 is called repetitively. On the other hand, it uses a search space that is much larger on average. We implemented this variant and compared it to the standard variant on Hamiltonian circuit problems with 300 vertices. The typical search space of Algorithm 5.2 increased from 150–300 literals for the standard variant to 300–600 literals for the Dynamic dependency graph variant, and the propagation speed dropped from 1–2 million to 0.5–1 million propagations per second. Accordingly, the total runtime rougly doubled.

**Comparison 5.7** (Standard variant vs. Stop-at-cycle-sources variant) In the

Stop-at-cycle-sources variant of Algorithm 5.2, the initialization of the $HP$ literals is the set $\{l \mid l$ has a path to $cs$ in $J_s$ through non-false, non-$CS$ literals$\}$. This set is smaller than the standard one. We implemented both possibilities; the choice of variant can be controlled with MiniSat(ID)'s command line option `-defn_search=`⟨option⟩, where ⟨option⟩ is `include_cs` (the "standard" variant as given in Algorithm 5.2) or `stop_at_cs`. The results of our comparison are shown in Figure 5.11, where the variants are marked "include_cs" and "stop_at_cs" respectively.

We show three comparisons: the average size of the search space of Algorithm 5.2 (in number of literals), the average size of the loops that have been found (also in number of loops), and the runtime. Note that the search space of Algorithm 5.2 may often be empty; since this average is measured over all calls to Algorithm 5.2, while the size of loops is averaged over all actual loops, the average loop size may be bigger than the average search space size.

The expected difference in size of the search space is visible, and consistent. We expect that searching for loops in smaller search spaces may lead to smaller loops being found; also this difference is indeed visible, though not consistently for all instances. The difference is consistent, and rather pronounced, on the Hitori instances.

The choice seems to have no important impact on the runtime required to solve the instances, however. All instances are clustered around the diagonal, and both versions yielded the same number of timeouts (23 out of 366 instances). Throughout the rest of the comparison, we have used the "stop_at_cs" version.

**Comparison 5.8** (Theory propagation vs. lazy on-line) We implemented a *lazy, on-line SAT* strategy for MiniSat(ID). The choice of strategy can be controlled by MiniSat(ID)'s command line option `-defn_strategy=`⟨option⟩, where ⟨option⟩ is `always` for the standard variant, and `lazy` for the lazy on-line SAT variant. Let $T$ be the given PC(ID) theory. Like the standard "always" version, the "lazy" version initially performs a Simplify step. After this, however, it just applies SAT solving on $comp(T)$. When a model of $comp(T)$ has been found, the strategy tries the $[\mathsf{BwLoop}]^{\mathsf{BF}}$ rules: if applicable, backtracking is done, but no further than needed.

We compare "always" and "lazy" in Figure 5.12. While for most instances, the results are clustered around the diagonal, "always" seems to be consistently faster than "lazy" on hitori puzzles. Also, "lazy" yields more timeouts on Hamiltonian circuit instances.

We compared also the total number of loops needed to find a solution. The "lazy" variant found some solutions without any loops at all; because these instances are not representable on a logarithmic scale, we gave them the value $10^{-1}$. Also some instances on which "lazy" timed out have this value; this is the case when a SAT model of $T$ had not been found yet at the moment of the timeout. Note that such instances that were solved by "always" within the

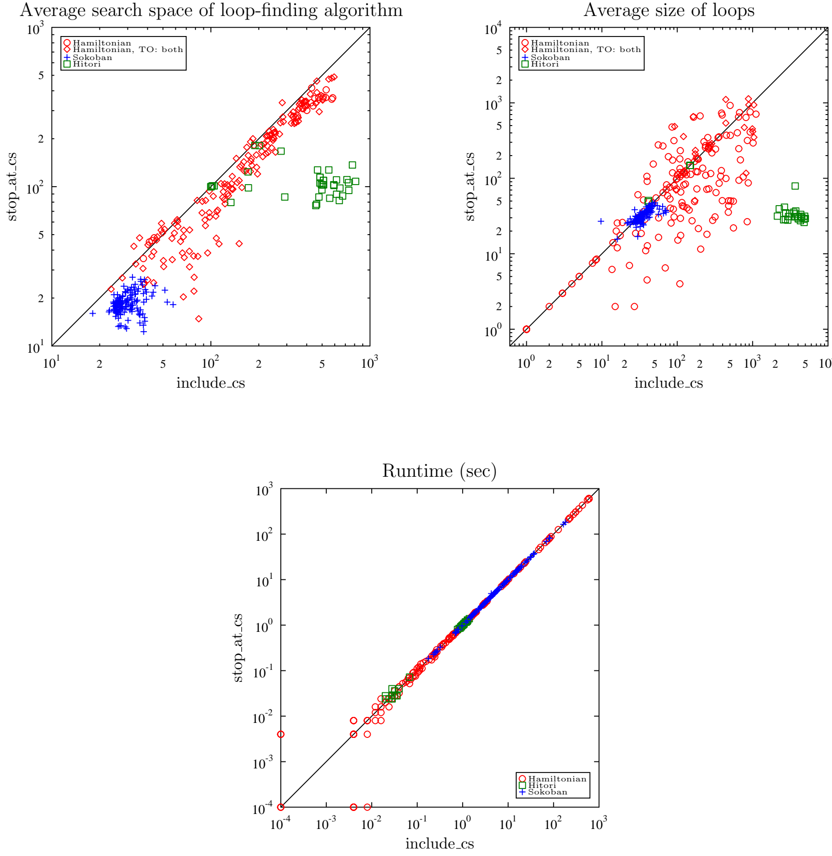



Figure 5.11: Comparison of standard variant and Stop-at-cycle-sources variant of Algorithm 5.2

time limit are a good indication that the extra propagations of Algorithm 5.2 are helpful.

We explain the difference in number of loops of “always” and “lazy”:

**on sokoban puzzles** Sokoban puzzles are planning problems, whereby on each step of the plan a reachability problem must be solved. If $T$ is a PC(ID) theory stemming from a sokoban puzzle, then the “lazy” variant first finds a model of $comp(T)$, i.e., a plan that solves the main goal, but in which potentially some steps are actually unreachable. If so, it then (only then) declares the step as unreachable, and goes back to finding a plan that solves the main goal. The “always” variant, by comparison, may be solving a lot of reachability problems for steps that would not lead to a plan anyway.

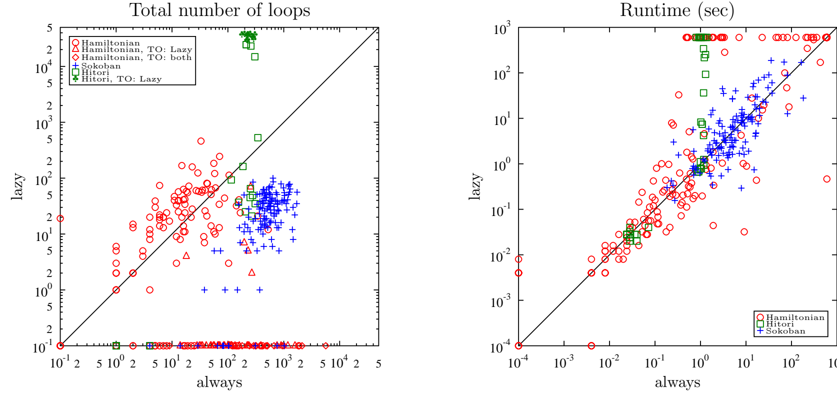



Figure 5.12: Comparison of "always" and "lazy" variants

**on hitori puzzles** The hitori puzzles are constrained such that it is easy to find a solution if the reachability requirement is not imposed. Thus the main difficulty in solving these puzzles comes from the combination of the reachability problem and the other constraints. The "always" variant prevents non-reachable solutions early on in the search, and therefore has a markedly better behaviour than "lazy".

The total number of timeouts of "always" was 23 (out of 366), of "lazy" was 54 (out of 366).

**Comparison 5.9** (Two versions of theory propagation) We also implemented an "adaptive" variant of the standard strategy. It can be called by MINI-SAT(ID)'s command line option `-defn_strategy=adaptive`. The basic strategy of "adaptive" is the same as the standard one, however, there is a heuristic that controls how often the BwLoop propagation phase is performed. More precisely, if this phase did not yield any propagations at a given decision level, then on the next decision level the phase will be skipped. If the decision level after that again does not yield any propagations by BwLoop, then the phase will be skipped on the next *two* decision levels, etc. The number of levels on which the BwLoop phase is skipped thus dynamically depends on the succes of the phase (it decreases again when there were propagations). Note that this is but one possible heuristic; many more options can be tried.

We again compared the total number of loops required to find a solution, and the runtimes. Figure 5.13 compares the "adaptive" strategy to both "always" and "lazy". We observe the same behaviour with respect to runtime as in the case of "always" versus "lazy": most results are clustered around the diagonal, but "adaptive" is slower on hitori puzzles, and on some Hamiltonian circuits.

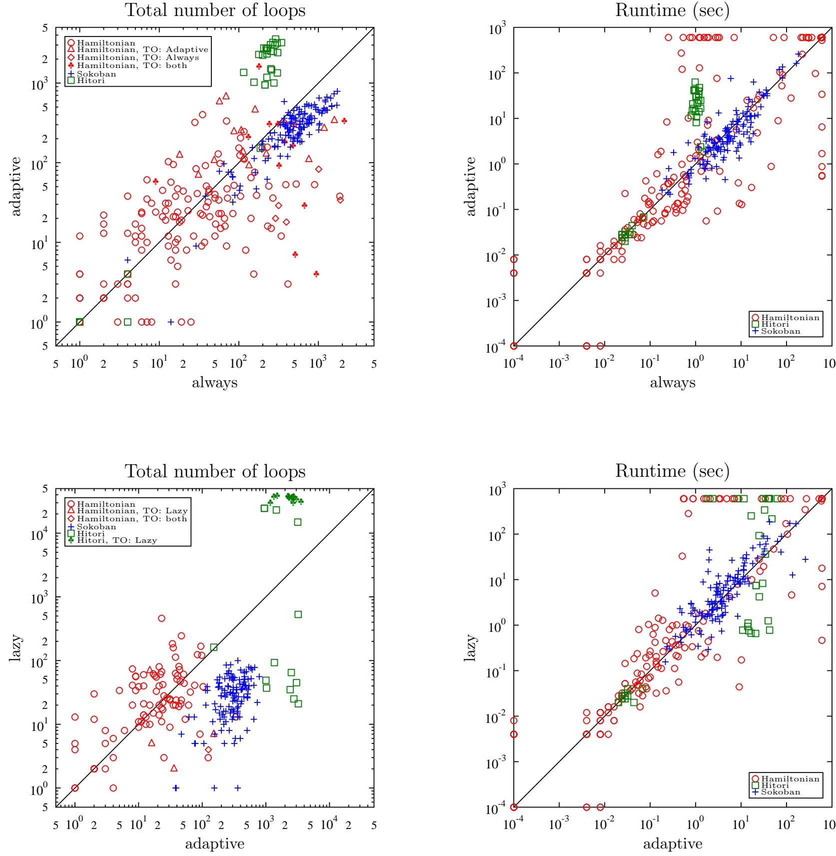



Figure 5.13: Comparison of “adaptive” with “always” and “lazy” variants

However, we observe that “adaptive” consistently requires slightly less loops on sokoban puzzles, and that it consistently requires a lot more loops on hitori puzzles—in both cases the likely explanation is the same as for the “always” versus “lazy” comparison.

The behaviour of “lazy” versus “adaptive” can be explained as a combination of the previous two comparisons, where “adaptive” is a strategy inbetween the other two.

For complete reference, we also show the total number of timeouts (out of 366 instances) of the three different search strategies, per type of instance in Figure 5.14.

| | "always" | "adaptive" | "lazy " |
|---|---|---|---|
| Hamiltonian Circuit (SAT) | 19 | 33 | 50 |
| Hamiltonian Circuit (UNSAT) | 4 | 4 | 4 |
| Sokoban puzzles | 0 | 0 | 0 |
| Hitori puzzles | 0 | 0 | 10 |
| Total | 23 | 37 | 54 |

Figure 5.14: Comparison of number of timeouts ($> 10min$) of different strategies

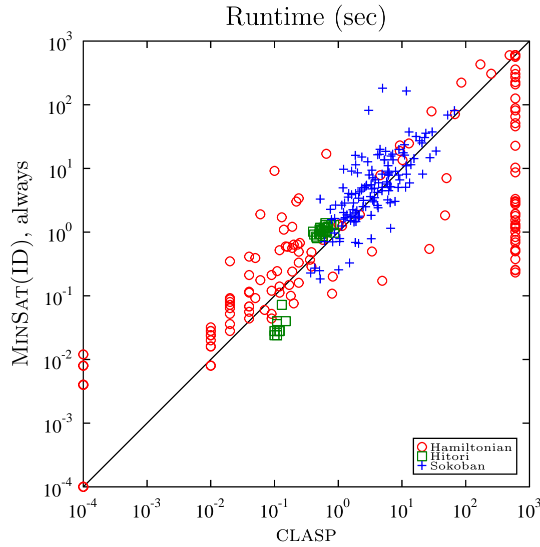



Figure 5.15: Comparison of MiniSat(ID) and clasp

### Comparison with ASP solvers

We have compared the efficiency of MiniSat(ID) to the ASP solvers clasp (version 1.1.2) (Gebser et al., 2007b), SModels$_{cc}$ (version 1.08) (Ward and Schlipf, 2004), and SModels (version 2.33) (Simons, 2000; Simons et al., 2002).[14] The results are given in Figures 5.15, 5.16 and 5.17.

clasp is the solver that has won the most recent ASP competition (Gebser et al., 2007c). MiniSat(ID) and clasp have very comparable results, with a slight edge for clasp on Hamiltonian circuit problems. MiniSat(ID) outperforms SModels$_{cc}$ and SModels.

[14] All solvers were called without (i.e. with default) command-line options, except SModels$_{cc}$, where we used option "nolookahead" as recommended by the authors.

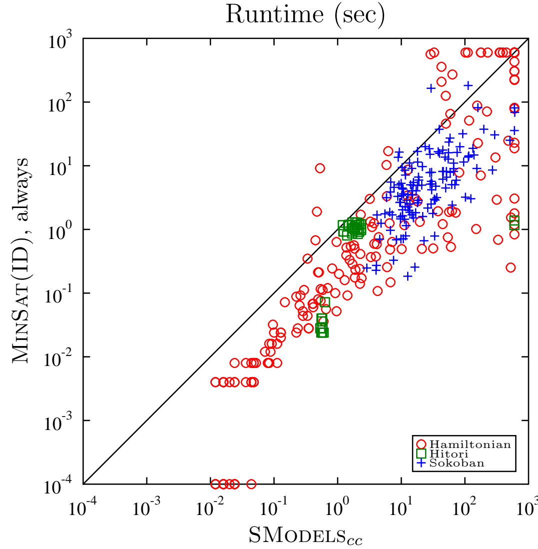



Figure 5.16: Comparison of MiniSat(ID) and SModels$_{cc}$

## 5.8 Conclusions and related work

In this chapter, we studied algorithms and strategies to solve the SAT(ID) problem. We used a framework of *transition rules* by Nieuwenhuis et al. (2006). We have added a generic schema for backtracking and failure transition rules to the framework, and proposed several transition rules for the SAT(ID) problem. In particular,

- the most general rule proposed is $\Delta$-Propagate, whereby *any unit consequence* of the given state sequence and definition is derived;
- the UnitPropDef, BwProp and FwProp rules apply definition-based propagations. These rules are based on the loop formula equivalence result (Theorem 4.2) of the previous chapter;
- the AddComp, AddLF, RemoveDef and Simplify rules apply theory modifications.

We have discussed different strategies for SAT(ID) solving based on these transition rules. Also, we developed algorithms for each of the proposed transition rules. We have implemented one of the strategies in MiniSat(ID), an extension of the SAT solver MiniSat, discussed its solving strategy, and evaluated its performance. Our evaluation included many variants, compared against

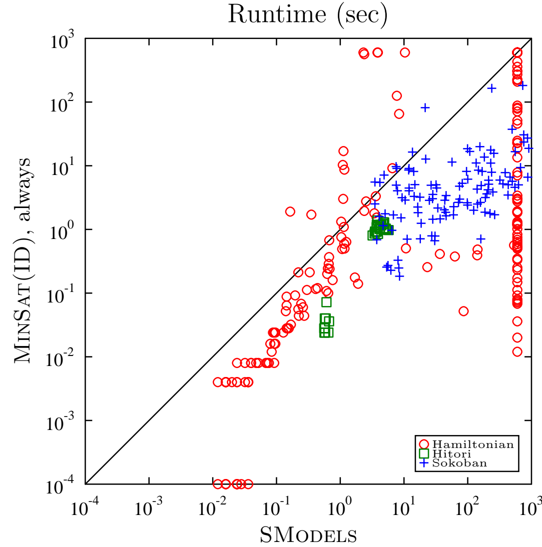



Figure 5.17: Comparison of MiniSat(ID) and SModels

each other, as well as a comparison with several ASP solvers. MiniSat(ID) had favourable results in these experiments.

In the context of SAT(ID), this is the first work to present an overview of possible solving strategies and algorithms. In the context of ASP, a similar work has been done by Gebser and Schaub (2006, 2007), whereby different propagation rules for ASP were presented in a tableau calculus.

For many of the strategies discussed here, a corresponding version in ASP has been developed and implemented. We mention some:

- eager techniques: researched by Janhunen (2004) for ASP, and by Pelov and Ternovska (2005) for SAT(ID);
- naive lazy approach: implemented in assat by Lin and Zhao (2004) and in cmodels by Lierler (2005);
- theory propagation: our own solvers MidL (Mariën et al., 2007a,b) and MiniSat(ID) (Mariën et al., 2008) have implemented some form of theory propagation for SAT(ID); the following solvers have done the same for ASP:
  - SModels (Simons, 2000; Simons et al., 2002). This system additionally performs a *lookahead* search, a type of strategy we have not discussed here. Lookahead search and clause learning diminish each other's returns and are therefore not often used together;

- DLV (Dell'Armi et al., 2001; Leone et al., 2006). This is a system for *disjunctive logic programming*, of which we have not discussed the semantics; however, it is closely related to ASP systems such as SModels;
- SModels$_{cc}$ (Ward and Schlipf, 2004). This system extends the SModels system with clause learning. When used without lookahead search, its strategy is very comparable to that of MidL and MiniSat(ID);
- clasp (Gebser et al., 2007b). This is the system that is most closely related to MidL and MiniSat(ID) both in strategies and in algorithms. It has won the most recent ASP competition (Gebser et al., 2007c).

There are many algorithms for computing the well-founded model of a logic program, e.g., (Van Gelder, 1993; Kemp et al., 1995; Chen and Warren, 1996; Lonc and Truszczyński, 2000; Brass et al., 2001). The main differences with our work are, firstly, that we potentially have to compute *several* well-founded models, namely, one for each interpretation of the open symbols of a definition, and secondly, that our computation has to be integrated with a SAT computation, and therefore defined literals may already have a truth value before there is a justification for that value. As a consequence, we focus on the detection of unfounded loops, and handle the rest of the computation with unit propagation also used in SAT. In contrast, the referenced works all focus on the computation of a *unique* well-founded model, and derive truth values only when justified by the rules of the logic program.

Nevertheless, our method of detecting unfounded loops has important similarities to the unfounded set detection methods of several of these works, in particular (Lonc and Truszczyński, 2000; Brass et al., 2001), although the latter is based on a transformation approach rather than a propagation approach. The similarities are best understood by interpreting defined atoms that are true but have no justification yet to be true as unknown, and by interpreting open literals in the same way as negative literals, namely, literals that can potentially become true later in the computation.

# Chapter 6

# Aggregates

## 6.1 Introduction

Many practical computational problems are best expressed using aggregate expressions. An aggregate expression is a second order expression that ranges over sets (of tuples of domain elements). FO expressions, by comparison, range over individual tuples of domain elements. The *cardinality* aggregate is a typical example: it can be seen as a function that maps a set to an integer, namely the size of the set.

Because many problem domains are most naturally modelled using aggregate expressions, and these cannot easily be expressed in FO or in ID-logic, we will add certain types of aggregate expressions to ID-logic. This not only improves modelling convenience, but often also solving efficiency. Thus, adding aggregate expressions enhances the applicability of model generation for ID-logic. In this chapter we present algorithms for model generation in the presence of aggregate expressions.

In Section 6.2 we illustrate an extension of ID-logic with specific aggregates, and present syntax and semantics for the propositional fragment of this logic. In Section 6.3 we then introduce propositional propagation methods for aggregate expressions, both recursive (i.e., in inductive definitions) and non-recursive. Finally, in Section 6.4 we present an extension of the MiniSat(ID) solver from Section 5.7 with such propagation methods, and evaluate it.

## 6.2 Preliminaries

### 6.2.1 Examples of ID-logic extended with aggregates

ID-logic was extended with arbitrary aggregates by Pelov (2004); Pelov et al. (2007). Here (and in the IDP system) we allow the most frequently occurring aggregates: cardinality, sum, product, minimum and maximum, which we will denote respectively by $Card$, $Sum$, $Prod$, $Min$, and $Max$.

In Section 6.2.2 we present syntax and semantics of *propositional* ID-logic extended with aggregates. In this section we illustrate the use of the first-order extension one some examples, without going into formal specifics concerning syntax and semantics. For these we refer to (Pelov et al., 2007).

**Example 6.1.** A *magic series* of order $n$ is a function $f : \{0,\dots,n-1\} \to \{0,\dots,n-1\}$ with the property that each $f(i)$ is equal to the number of times $i$ equals $f(j)$, for some arbitrary $j$. This is expressed in ID-logic extended with aggregates as:

$$\forall i\ f(i) = Card(\{j \mid f(j) = i\}). \tag{6.1}$$

We illustrate the semantics of $Card$ expressions with this example. Consider the interpretation $I$ with domain $D = \{0,1,2,3\}$, and with $f^I = \{0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 0\}$. Then $I \models (6.1)$, i.e., $f^I$ is a magic series of order 4. Indeed, let $E$ be $Card(\{j \mid f(j) = i\})$, then we have the following:

$$\begin{aligned}
E^{I[i/0]} &= \left|\left\{d_j \in D \mid I[j/d_j] \models f(j) = 0\right\}\right| = \left|\{3\}\right| &&= 1, \text{ and } f(i)^{I[i/0]} = 1;\\
E^{I[i/1]} &= \left|\left\{d_j \in D \mid I[j/d_j] \models f(j) = 1\right\}\right| = \left|\{0,2\}\right| &&= 2, \text{ and } f(i)^{I[i/1]} = 2;\\
E^{I[i/2]} &= \left|\left\{d_j \in D \mid I[j/d_j] \models f(j) = 2\right\}\right| = \left|\{1\}\right| &&= 1, \text{ and } f(i)^{I[i/2]} = 1;\\
E^{I[i/3]} &= \left|\left\{d_j \in D \mid I[j/d_j] \models f(j) = 3\right\}\right| = \left|\emptyset\right| &&= 0, \text{ and } f(i)^{I[i/3]} = 0.
\end{aligned}$$

An alternative syntax (allowed in the IDP system) for cardinality expressions of the form $Card(\{y \mid \Psi[y]\}) \sim n$ is $\exists{\sim}n\ y : \Psi[y]$, for $\sim \in \{=,<,\leq,>,\geq\}$.

With the following example we illustrate the use of $Sum$, and the idea behind the semantics of inductive definitions containing aggregates.

**Example 6.2.** Recall the *company control* problem from Example 1.1, and in specific, definition (1.1).

$$\left\{\begin{array}{ll} \forall x,y\ \big(Controls(x,y) \leftarrow Sum\{\underline{s}, z \mid & (x = z \vee Controls(x,z)) \\ & \wedge Owns(z,y) = s\} > 50\big) \end{array}\right\} [1.1]$$

Let $I_O$ be the $\{Owns/2\}$-interpretation with domain $\{\mathsf{A},\mathsf{B},\mathsf{C}\}$, and with $Owns^{I_O}$ as in Figure 1.1:

$$\left\{\begin{array}{lll} (\mathsf{A},\mathsf{A}) \mapsto 20, & (\mathsf{B},\mathsf{A}) \mapsto 40, & (\mathsf{C},\mathsf{A}) \mapsto 40,\\ (\mathsf{A},\mathsf{B}) \mapsto 26, & (\mathsf{B},\mathsf{B}) \mapsto 49, & (\mathsf{C},\mathsf{B}) \mapsto 25,\\ (\mathsf{A},\mathsf{C}) \mapsto 60, & (\mathsf{B},\mathsf{C}) \mapsto 10, & (\mathsf{C},\mathsf{C}) \mapsto 30 \end{array}\right\}.$$

We compute $\mathrm{wfm}_{(1.1)}(I_O)$ as the limit of a well-founded sequence from $I$, the empty extension of $I_O$ to $\{Controls/2\}$. For this purpose we have to evaluate the $Sum$ expression in three-valued interpretations: this does not yield a precise value, but instead a lower and upper bound on the possible sum values. In the evaluation here we need only the lower bound. Denote by $S[x,y]$ the set $\{\underline{s}, z \mid (x = z \vee Controls(x,z)) \wedge Owns(z,y) = s\}$. We find:

- initially, $I_0 = I$;

- next, $I_1 = I_0[Controls(\mathsf{A},\mathsf{C})/\boldsymbol{t}]$. Indeed, $I_0[(x,y)/(\mathsf{A},\mathsf{C})]\big(Sum(S[x,y]) > 50\big) = \boldsymbol{t}$ because $Sum(S[x,y]^{I_0[(x,y)/(\mathsf{A},\mathsf{C})]}) \geq Sum(\{(\underline{60},\mathsf{A})\}) = 60$;
- next, $I_2 = I_1[Controls(\mathsf{A},\mathsf{B})/\boldsymbol{t}]$. Indeed, $I_1[(x,y)/(\mathsf{A},\mathsf{B})]\big(Sum(S[x,y]) > 50\big) = \boldsymbol{t}$ because $Sum(S[x,y]^{I_1[(x,y)/(\mathsf{A},\mathsf{B})]}) \geq Sum(\{(\underline{26},\mathsf{A}),(\underline{25},\mathsf{C})\}) = 51$;
- next, $I_3 = I_2[Controls(\mathsf{A},\mathsf{A})/\boldsymbol{t}]$. Indeed, we have that $Sum(S[x,y]^{I_2[(x,y)/(\mathsf{A},\mathsf{A})]}) \geq Sum(\{(\underline{20},\mathsf{A}),(\underline{40},\mathsf{B}),(\underline{40},\mathsf{C})\}) = 100$ and hence $I_2[(x,y)/(\mathsf{A},\mathsf{A})]\big(Sum(S[x,y]) > 50\big) = \boldsymbol{t}$;
- finally, $I_4 = I_3[U/\boldsymbol{f}]$, where $U$ is the unfounded set $\{Controls(\mathsf{B},\mathsf{A}), \dots, Controls(\mathsf{C},\mathsf{C})\}$.

With our last example of ID-logic extended with aggregates, we illustrate recursion over a $Min$ aggregate.

**Example 6.3.** Consider the *shortest path* problem: the problem of finding, in a weighted graph, the shortest path between any two edges. Van Gelder (1992) modelled this problem as follows, where $Edge(x,y,w)$ represents an edge $(x,y)$ with weight $w$:

$$\left\{\begin{array}{l} \forall x,y,w\ SP(x,y,w) \leftarrow \\ \qquad w = Min\Big(\Big\{w' \Big| \begin{array}{l} Edge(x,y,w') \lor \\ (\exists w_1,w_2,z\ SP(x,z,w_1) \\ \qquad \land Edge(z,y,w_2) \\ \qquad \land w' = w_1 + w_2)\Big\}\Big) \end{array} \end{array}\right\}. \tag{6.2}$$

Definition (6.2) is a correct representation of the shortest path problem. It uses the knowledge that any shortest path of length $n+1$ must be an extension of a shortest path of length $n$. This fact is the basis of Dijkstra's algorithm.

### 6.2.2 Ground aggregate expressions

#### Syntax

We extend the ECNF normal form introduced in Section 5.2.4 with ground aggregate expressions. In Section 6.2.3 we then illustrate how to obtain such ground aggregate expressions from first-order ones.

A *weighted set* is a set of tuples $(l,w)$, where $l$ is a literal, and $w$ an integer weight. If $S$ is a weighted set, we denote by $S^{\text{lits}}$ the set $\{l \mid (l,w) \in S\}$. We allow only weighted sets for which $|S^{\text{lits}}| = |S|$, i.e., each literal in $S^{\text{lits}}$ has only one occurrence in $S$.

**Definition 6.1** (ECNF)**.** A theory in *ECNF normal form* is a propositional theory that may contain

- a CNF subtheory;
- rules of the form $p \leftarrow \bigvee S$ or $p \leftarrow \bigwedge S$, where $p$ is an atom, and $S$ a set of literals;

- declarations of non-empty sets of literals;
- declarations of non-empty weighted sets;
- a declaration of the form $p \leftarrow l \leq Card(S) \leq u$, where $p$ is an atom, $S$ is a previously declared set of literals, and $l$ and $u$ are integers such that $0 \leq l \leq u \leq |S|$;
- a declaration of the form $p \leftarrow l \leq Aggr(S) \leq u$, where $Aggr$ is one of $\{Sum, Prod, Min, Max\}$, $p$ is an atom, $S$ is a previously declared weighted set, and $l$ and $u$ are integers such that $0 \leq l \leq u \leq \mathbb{A}_{(l,w)\in S}\, w$, where $\mathbb{A}$ is respectively $\sum$, $\prod$, $\min_{\leq}$ or $\max_{\leq}$. If $Aggr$ is $Sum$ or $Prod$, all the weights in $S$ must be $\geq 0$;
- a declaration of the form $AMO(S)$, where $S$ is a set of literals.
- a declaration of the form $EU(S)$, where $S$ is a set of literals.

Each atom $p$ can appear at most once as the head of a rule $p \leftarrow \ldots$.

Declarations of the form $p \leftarrow l \leq Card(S) \leq u$ or $p \leftarrow l \leq Aggr(S) \leq u$ are called *aggregate expressions* in the remainder of the text. Throughout the rest of the text $\Psi$ denotes an ECNF theory, and $\Delta$ an ECNF definition, unless noted otherwise.

For reasons of simplicity, we disallow negative weights in weighted sets that are used in a $Sum$ or $Prod$ aggregate expression. In the algorithms presented in next section, we further assume that no weights of value 0 (zero) are used in $Sum$ or $Prod$ aggregate expressions—this assumption can be met by a simple transformation.

An ECNF theory contains at most one definition, namely the set of all rules $p \leftarrow \ldots$ in the theory. Like for DefNF definitions, each defined atom is the head of exactly one rule. The body of that rule is either a disjunction of literals, conjunction of literals, or the rule is an aggregate expression. We use the symbols $\mathcal{D}_{\text{lits}}$ and $\mathcal{C}_{\text{lits}}$ as before, and we add the symbol $\mathcal{A}_{\text{lits}}$, denoting the atoms that are defined by an aggregate expression and their negations.

### Semantics

We define the semantics of ECNF theories. To do so, we have to define the semantics of definitions anew. Many semantics for aggregate expressions in logic programs have been discussed in the literature, e.g. by Van Gelder (1992); Simons (1999); Pelov et al. (2004). In many cases such extensions were proposed with syntactical restrictions, or only of specific aggregates, e.g. (Mumick et al., 1990; Kemp and Stuckey, 1991; Niemelä et al., 1999; Dell'Armi et al., 2003; Marek et al., 2004).

The approach we take here was developed by Pelov (2004); Pelov et al. (2007): we retain the original definition of the semantics of inductive definitions. This definition makes use of *three-valued* semantics; for this purpose, we extend three-valued semantics to aggregate expressions; in other words, we define the meaning of an aggregate expression in a three-valued interpretation. Once the three-valued satisfaction relation is extended to aggregate expressions, Definition 2.1 of a well-founded sequence is applicable to definitions containing aggregate expressions.

We define a *three-valued set* as a pair $(S^{\boldsymbol{t}}, S^{\boldsymbol{u}})$ of two disjoint sets: a set $S^{\boldsymbol{t}}$ of *certain* elements and a set $S^{\boldsymbol{u}}$ of *possible* elements. Intuitively, such a set is an approximation for all sets $S'$ with $S^{\boldsymbol{t}} \subseteq S' \subseteq S^{\boldsymbol{t}} \cup S^{\boldsymbol{u}}$. For a three-valued set $S = (S^{\boldsymbol{t}}, S^{\boldsymbol{u}})$, we denote by $s^{\boldsymbol{t}} \in S$ that $s \in S^{\boldsymbol{t}}$, by $s^{\boldsymbol{u}} \in S$ that $s \in S^{\boldsymbol{u}}$, and by $s^{\star} \in S$ that $s \in S^{\boldsymbol{t}} \cup S^{\boldsymbol{u}}$.

The *valuation* of a set of literals $S$ in a three-valued interpretation $I$, denoted $S^I$, is the three-valued set $\big(\{l \in S \mid I(l) = \boldsymbol{t}\}, \{l \in S \mid I(l) = \boldsymbol{u}\}\big)$. The valuation of a weighted set $S$ in $I$, also denoted $S^I$, is the three-valued weighted set $\big(\{(l,w) \in S \mid I(l) = \boldsymbol{t}\}, \{(l,w) \in S \mid I(l) = \boldsymbol{u}\}\big)$.

We now define the valuation of expressions $E$ of the form $l \leq Aggr(S) \leq u$ in a three-valued interpretation $I$. We define when $I(E) = \boldsymbol{t}$ and when $I(E) = \boldsymbol{f}$; we derive from this that $I(E) = \boldsymbol{u}$ if $I(E) \neq \boldsymbol{t}$ and $I(E) \neq \boldsymbol{f}$. For $Sum$ and $Prod$, we assume the aggregated terms are strictly positive integers.[1]

- $I\big(lwr \leq Card(S) \leq upr\big) = \boldsymbol{t}$ if $\begin{cases} \big|\{l^{\boldsymbol{t}} \in S^I\}\big| \geq lwr \text{ and} \\ \big|\{l^{\star} \in S^I\}\big| \leq upr; \end{cases}$
- $I\big(lwr \leq Card(S) \leq upr\big) = \boldsymbol{f}$ if $\begin{cases} \big|\{l^{\star} \in S^I\}\big| < lwr \text{ or} \\ \big|\{l^{\boldsymbol{t}} \in S^I\}\big| > upr; \end{cases}$
- $I\big(lwr \leq Min(S) \leq upr\big) = \boldsymbol{t}$ if $\begin{cases} \min_{\leq}(\{w \mid (l,w)^{\star} \in S^I\}) \geq lwr \text{ and} \\ \min_{\leq}(\{w \mid (l,w)^{\boldsymbol{t}} \in S^I\}) \leq upr; \end{cases}$
- $I\big(lwr \leq Min(S) \leq upr\big) = \boldsymbol{f}$ if $\begin{cases} \min_{\leq}(\{w \mid (l,w)^{\boldsymbol{t}} \in S^I\}) < lwr \text{ or} \\ \min_{\leq}(\{w \mid (l,w)^{\star} \in S^I\}) > upr; \end{cases}$
- $I\big(lwr \leq Max(S) \leq upr\big) = \boldsymbol{t}$ if $\begin{cases} \max_{\leq}(\{w \mid (l,w)^{\boldsymbol{t}} \in S^I\}) \geq lwr \text{ and} \\ \max_{\leq}(\{w \mid (l,w)^{\star} \in S^I\}) \leq upr; \end{cases}$
- $I\big(lwr \leq Max(S) \leq upr\big) = \boldsymbol{f}$ if $\begin{cases} \max_{\leq}(\{w \mid (l,w)^{\star} \in S^I\}) < lwr \text{ or} \\ \max_{\leq}(\{w \mid (l,w)^{\boldsymbol{t}} \in S^I\}) > upr; \end{cases}$
- $I\big(lwr \leq Sum(S) \leq upr\big) = \boldsymbol{t}$ if $\begin{cases} \sum_{(l,w)^{\boldsymbol{t}} \in S^I} w \ \geq lwr \text{ and} \\ \sum_{(l,w)^{\star} \in S^I} w \ \leq upr; \end{cases}$
- $I\big(lwr \leq Sum(S) \leq upr\big) = \boldsymbol{t}$ if $\begin{cases} \sum_{(l,w)^{\star} \in S^I} w \ \geq lwr \text{ or} \\ \sum_{(l,w)^{\boldsymbol{t}} \in S^I} w \ \leq upr; \end{cases}$
- $I\big(lwr \leq Prod(S) \leq upr\big) = \boldsymbol{t}$ if $\begin{cases} \prod_{(l,w)^{\boldsymbol{t}} \in S^I} w \ \geq lwr \text{ and} \\ \prod_{(l,w)^{\star} \in S^I} w \ \leq upr; \end{cases}$

[1] It is theoretically straightforward to generalize the semantics for arbitrary integers, but this requires a more involved notation, and is practically largely irrelevant.

- $I\big(lwr \leq Prod(S) \leq upr\big) = \boldsymbol{t}$ if $\begin{cases} \prod_{(l,w)^{\star} \in S^I} w & \geq lwr \text{ or} \\ \prod_{(l,w)^{\boldsymbol{t}} \in S^I} w & \leq upr. \end{cases}$

The semantics for empty sets is standard: $\sum(\emptyset) = 0$, $\prod(\emptyset) = 1$, $\min_{\leq}(\emptyset) = +\infty$, and $\max(\emptyset) = -\infty$. The values $\pm\infty$ never have to be explicitly represented: they only need to be compared to the (finite) lower- and upperbound $lwr$ and $upr$ of the given aggregate expression.

The meaning of an expression of the form $AMO(S)$ in a two-valued interpretation $I$ is $\big|\{l \in S \mid I(l) = \boldsymbol{t}\}\big| \leq 1$, i.e., *at most one* of the literals of $S$ can be true. The meaning of an expression of the form $EU(S)$ in a two-valued interpretation $I$ is $\big|\{l \in S \mid I(l) = \boldsymbol{t}\}\big| = 1$, i.e., there *exists a unique* literal of $S$ that is true. Note that such expressions occur only as sentences of an ECNF theory, and not in definition rules. Therefore it suffices to provide their meaning in two-valued interpretations.

*Remark* 6.1. A specific advantage of the separation of declarations of sets on the one hand, and aggregate expressions over these sets on the other, is that algorithms may exploit the fact that any given set may occur in several expressions.

*Remark* 6.2. For $Sum$ and $Prod$ it is possible to define a more precise semantics, called the *ultimate* semantics. The ultimate semantics takes into account all possible approximated sets. For instance, $lwr \leq Sum(S) \leq upr$ is then true in a three-valued interpretation $I$ only if for each set $S'$ that is actually approximated by $S^I$, $lwr \leq \sum_{(l,w) \in S'} w \leq upr$ holds.

It was proven by Pelov et al. (2007) that the standard satisfaction relation as given above is polynomially computable,[2] while deciding whether the ultimate satisfaction relation holds is in $\mathsf{NP}$, and $\mathsf{NP}$-complete for some classes of aggregate expressions.

### 6.2.3 Grounding of aggregates

In this section we illustrate how to ground first-order ID-logic theories with aggregates and a finite domain into ECNF theories. The purpose of this section is to enable the reader to relate first-order aggregates to propositional aggregate expressions; for details of the grounding process we refer to (Wittocx et al., 2008b).

We introduced a general notion of *predicate introduction* in (Wittocx et al., 2006; Vennekens et al., 2007a). Using predicate introduction, an arbitrary ID-logic theory containing (possibly nested) aggregate expressions can be transformed into an equivalent ID-logic theory (equivalent up to the original vocabulary), where all occurrences of aggregate expressions are in a definition, and of

[2] This result entails that extending ID-logic with aggregate expressions by using the standard satisfaction relation does not increase the complexity of the model expansion task for ID-logic, nor does it increase the class of problems that can be solved with ID-logic model expansion.

the form

$$\forall \overline{y}, n \left( P(\overline{y}, n) \leftarrow Aggr(\{\overline{x} \mid Q[\overline{x}, \overline{y}]\}) \sim n \right) \tag{6.3}$$

or

$$\forall \overline{y} \left( P(\overline{y}) \leftarrow Aggr(\{\overline{x} \mid Q[\overline{x}, \overline{y}]\}) \sim n' \right), \tag{6.4}$$

where $Q$ is a (possibly newly introduced) predicate symbol, $Aggr \in \{Card, Sum, Prod, Min, Max\}$, $\sim \in \{=, <, \leq, >, \geq\}$, and $n'$ is an integer constant.

In ECNF, we also represent all aggregate expressions as rules; we illustrate how first-order rules of the form (6.3) or (6.4) relate to ECNF aggregate expressions.

Let $D$ be a finite domain. For simplicity we assume $D \subset \mathbb{N}$. Assume further $Aggr = Sum$ and $\sim \, = \, \leq$. Then the grounding of (6.3) in $D$ contains the following weighted set declarations: for all $\overline{d} \in D^{|\overline{y}|}$, $S_{\overline{d}} = \big\{(l, x_1) \mid l$ represents the instantiation of $Q[\overline{x}, \overline{y}]$ with $\overline{x} = (\underline{x_1}, x_2, \ldots, x_n)$ and $\overline{y}/\overline{d}$, for all $(x_1, x_2, \ldots, x_n) \in D^n$, where $n = |\overline{x}|\big\}$, and the following aggregate expressions: for all $\overline{d'} \in D^{|\overline{y}|+1}$, $p_{\overline{d'}} \leftarrow 0 \leq Sum(S_{\overline{d}}) \leq u$, where $\overline{d'} = (\overline{d}, u)$ and $p_{\overline{d'}}$ represents the instantiation of $P(\overline{y}, n)$ with $\overline{y} = \overline{d}$, $n = u$. Observe that each declared weighted set may occur in multiple aggregate expressions.

For other aggregate expressions and comparison operators, the grounding is similar. We give an example for $Card$.

**Example 6.4.** Example 6.1 continued. We represent the magic series problem expressed by formula (6.1) in the above normal form:

$$\begin{aligned}
&\forall i, j \; Ts(i,j) \equiv f(i) = j, \\
&\left\{ \; \forall i, j \left( Ts(i,j) \leftarrow Card\big(\{j' \mid f(j') = i\}\big) = j \right) \; \right\}.
\end{aligned}$$

We ground this for a domain of $\{0, 1, 2, 3\}$, i.e., we give the ECNF propositional representation of the magic series problem of order 3. A propositional atom of the form $f_{i,j}$ represents the atom $f(i) = j$; likewise, a propositional atom of the form $Ts_{i,j}$ represents $Ts(i,j)$.

$$\begin{array}{llll}
f_{0,0} \vee \neg Ts_{0,0}, & f_{0,1} \vee \neg Ts_{0,1}, & \ldots, & f_{3,3} \vee \neg Ts_{3,3} \\
\neg f_{0,0} \vee Ts_{0,0}, & \neg f_{0,1} \vee Ts_{0,1}, & \ldots, & \neg f_{3,3} \vee Ts_{3,3}, \\
S_0 = \{f_{0,0}, f_{1,0}, f_{2,0}\}, & & & \\
S_1 = \{f_{0,1}, f_{1,1}, f_{2,1}\}, & & & \\
S_2 = \{f_{0,2}, f_{1,2}, f_{2,2}\}, & & & \\
Ts_{0,0} \leftarrow 0 \leq Card(S_0) \leq 0, & Ts_{0,1} \leftarrow 1 \leq Card(S_0) \leq 1, & & \\
Ts_{0,2} \leftarrow 2 \leq Card(S_0) \leq 2, & Ts_{0,3} \leftarrow 3 \leq Card(S_0) \leq 3, & & \\
Ts_{1,0} \leftarrow 0 \leq Card(S_1) \leq 0, & \ldots, & & \\
\ldots & Ts_{3,3} \leftarrow 3 \leq Card(S_3) \leq 3. & &
\end{array}$$

The semantics for aggregates in definitions generalizes the semantics for aggregates in FO: if $P \neq Q$, then the definition consisting of only the rule (6.3) is equivalent to

$$\forall \overline{y}, n \left( P(\overline{y}, n) \equiv Aggr(\{\overline{x} \mid Q[\overline{x}, \overline{y}]\}) \sim n \right). \tag{6.5}$$

Therefore, in the case of non-recursive aggregate expressions, "←" simply means "≡" in ECNF. In practice, we distinguish between recursive and non-recursive aggregates by searching strongly connected components in the dependency graph: non-recursive aggregates are defined by an atom that forms a singleton component.

#### At-most-one statements

A specific type of aggregate that occurs in many practical examples is an *at-most-one* statement: a cardinality expression of the form $Card(\{x \mid \Psi[x]\}) \leq 1$, or equivalently, $\exists{\leq}1\, x\ \Psi[x]$. The frequent occurrence of such statements and the availability of an efficient propagation algorithm for it justify introduction of the special language construct $AMO(S)$.

The main reason at-most-one statements feature in many practical examples is their occurrence in *exists unique* statements. These can be rewritten as $EU(S) \equiv \bigvee S \wedge AMO(S)$. Exists unique statements, in turn, naturally arise as the result of grounding a first-order sentence that contains the exists unique ($\exists!$) quantifier. This is particularly relevant for representing functions by predicates: a function $F : D^n \rightarrow D$ can be represented by the predicate $P_F(D^{n+1})$, if the axiom that declares $P_F$ to be a function is added to the theory:

$$\forall \overline{x}\ \exists! y\ P_F(\overline{x}, y). \tag{6.6}$$

This can be grounded to a conjunction of ground exists unique statements: $\bigwedge_{\overline{d} \in D^n} EU(\{P_F(\overline{d}, d') \mid d' \in D\})$, which in turn are easily transformed into clauses and $AMO$ statements.

## 6.3 Aggregates: algorithms

### 6.3.1 At-most-one statements

We have the following equivalences for at-most-one statements:

$$\begin{aligned} AMO(S) &\equiv \left( A \leftarrow 0 \leq Card(S) \leq 1 \right) \wedge A \\ &\equiv \bigwedge_{\substack{l_i, l_j \in S, \\ i<j}} (\neg l_i \vee \neg l_j). \end{aligned} \tag{6.7}$$

However, there is a simple and easily implementable propagation rule for such statements, which makes transformations as in (6.7) superfluous.

We define the AtMostOne transition rule and its backtracking and failure variants $[\mathsf{AtMostOne}]^{\mathsf{BF}}$:

$$M \parallel \Psi \quad \Longrightarrow M \neg l \parallel \Psi \quad \textbf{if} \begin{cases} AMO(S) \in \Psi \\ l \in S, M(l) = \boldsymbol{u} \\ \exists l' \in S \text{ with } l' \neq l \text{ and } M(l') = \boldsymbol{t} \end{cases}$$

Observe that exactly the same propagations would be obtained by applying unit propagation on the CNF translation of such expressions (see formula (6.7)).

The following simple algorithm implements the $[\mathsf{AtMostOne}]^{\mathsf{BF+}}$ rules. With each literal $l$, a list is maintained of all expressions $AMO(S)$ for which $l \in S$. Whenever a literal $l$ becomes true, each such expression $AMO(S)$ is visited, whereby all literals in $S \setminus \{l\}$ are made false.

Note that no new data structure has to be introduced to store at-most-one expressions: $AMO(S)$ can just be stored as if it was a clause $\bigvee S$. The only references to this "clause" are from the lists described above (which are new data structures).

*Remark* 6.3. One can define propagation rules for exists unique expressions of the form $EU(S)$; however, they would coincide exactly to the combination of unit propagation on the clause $\bigvee S$, and at-most-one propagation on $AMO(S)$. It therefore suffices to provide only at-most-one propagation rules.

However, one can think of a possible enhancement for exists unique expressions. When a literal $l$ becomes true, apart from $AMO(S)$, also the clause $\bigvee S$ could be visited, and one of its watches set to $l$. In fact, $AMO(S)$ and $\bigvee S$ may even be represented by the same data structure. We tried this enhancement in some brief experiments, and found that it didn't yield any significant speedup.

*Remark* 6.4. While the reduction to SAT given by equation (6.7) is quadratic in size, there exist linear transformations that introduce new literals. They exploit the "divide and conquer" property that

$$AMO(S_1 \cup S_2) \equiv AMO(S_1 \cup \{l\}) \wedge AMO(S_2 \cup \{\neg l\}),$$

where $l$ is a new atom. Using such transformations one can obtain a PC formula equivalent to $AMO(S)$ that is linear in the size of $S$, but also contains a linear number of new atoms. We have not compared our approach to such transformation approaches.

### 6.3.2 Reductions to SAT

For the $Min$ and $Max$ aggregates, the following equivalences translate aggregate expressions into PC theories.

$$l \leq Min(\{(l_1, w_1), \ldots, (l_N, w_N)\}) \leq u \equiv \left(\bigwedge_{i:w_i<l} \neg l_i\right) \wedge \left(\bigvee_{i:l\leq w_i \leq u} l_i\right) \quad (6.8)$$

$$l \leq Max(\{(l_1, w_1), \ldots, (l_N, w_N)\}) \leq u \equiv \left(\bigwedge_{i:w_i>u} \neg l_i\right) \wedge \left(\bigvee_{i:l\leq w_i \leq u} l_i\right) \quad (6.9)$$

For $Card$, $Sum$ and $Prod$, however, similar translations are not as trivial to define. Cardinalities occur most often; translations for them have been defined, e.g., by Sinz (2005); Marques-Silva and Lynce (2007). However, these translations require the cardinality expressions to be given as sentences (constraints) of the theory—occurrences of a cardinality expression as a subformula are excluded. Thus, these transformations do not help to solve a problem like the company control problem of Example 6.2.

The obvious advantage of reductions to SAT is that no special-purpose implementations of propagation for aggregate expressions are required. We list the disadvantages.

- Not all transformations retain full propagation efficiency under CNF unit propagation.
- The generality of aggregate expressions as subformulas is lost, or at the very least, greatly complicates the transformation.
- The ability to exploit multiple occurrences of the same weighted sets is lost, or greatly complicates the transformations.

### 6.3.3 Non-recursive aggregate expressions

We define some different propagations that are possible for aggregate expressions. In the rest of the text, we identify $Card(S)$ with $Sum(S')$, for $S' = \{(l, 1) \mid l \in S\}$—the defined propagations for $Sum$ therefore carry over to $Card$. Furthermore, for the sake of brevity, we will define these propagations for $Min$ and for $Sum$ only: the propagations for $Max$ and $Prod$ expressions can easily be derived from those for $Min$ and $Sum$ expressions. We again assume interpretations to be three-valued, unless we explicitly mention otherwise.

Let $S$ be a weighted set and $M$ an interpretation. We use the following notations:

- $S.min^M_{Aggr} = \underset{(l_i,w_i)\in S \wedge M(l_i)=\boldsymbol{t}}{\mathbb{A}} w_i$; and
- $S.max^M_{Aggr} = \underset{(l_i,w_i)\in S \wedge M(l_i)\neq \boldsymbol{f}}{\mathbb{A}} w_i$,

for $Aggr \in \{Min, Max, Sum, Prod\}$, and accordingly $\mathbb{A} = \min_{\leq}$, $\max_{\leq}$, $\sum$, or $\prod$. Note that $S.max^M_{Min} \leq S.min^M_{Min}$. We call $S.min^M_{Aggr}$ and $S.max^M_{Aggr}$ the

MinBwTrue:

$$M \parallel \Psi \quad \Longrightarrow M\, d \parallel \Psi \quad \textbf{if} \begin{cases} (d \leftarrow l \leq Min(S) \leq u) \in \Psi \\ M(d) = \boldsymbol{u} \\ S.min^M_{Min} \geq l \text{ and } S.max^M_{Min} \leq u \end{cases}$$

[MinBwTrue]$^{\mathsf{BF}}$

MinBwFalse:

$$M \parallel \Psi \quad \Longrightarrow M\, \neg d \parallel \Psi \quad \textbf{if} \begin{cases} (d \leftarrow l \leq Min(S) \leq u) \in \Psi \\ M(d) = \boldsymbol{u} \\ S.min^M_{Min} < l \text{ or } S.max^M_{Min} > u \end{cases}$$

[MinBwFalse]$^{\mathsf{BF}}$

MinFwTT:

$$M \parallel \Psi \quad \Longrightarrow M\, l_i \parallel \Psi \quad \textbf{if} \begin{cases} (d \leftarrow l \leq Min(S) \leq u) \in \Psi \\ M(d) = \boldsymbol{t} \\ \{l_j \mid (l_j, w_j) \in S, M(l_j) \neq \boldsymbol{f}, w_j \leq u\} \\ \qquad = \{l_i\} \end{cases}$$

MinFwTF:

$$M \parallel \Psi \quad \Longrightarrow M\, \neg l_i \parallel \Psi \quad \textbf{if} \begin{cases} (d \leftarrow l \leq Min(S) \leq u) \in \Psi \\ M(d) = \boldsymbol{t} \\ (l_i, w_i) \in S, w_i < l \end{cases}$$

[MinFwTF]$^{\mathsf{BF}}$

MinFwFT:

$$M \parallel \Psi \quad \Longrightarrow M\, l_i \parallel \Psi \quad \textbf{if} \begin{cases} (d \leftarrow l \leq Min(S) \leq u) \in \Psi \\ M(d) = \boldsymbol{f} \\ S.max^M_{Min} \leq u \\ \{l_j \mid (l_j, w_j) \in S, M(l_j) \neq \boldsymbol{f}, w_j < l\} \\ \qquad = \{l_i\} \end{cases}$$

MinFwFF:

$$M \parallel \Psi \quad \Longrightarrow M\, \neg l_i \parallel \Psi \quad \textbf{if} \begin{cases} (d \leftarrow l \leq Min(S) \leq u) \in \Psi \\ M(d) = \boldsymbol{f} \\ S.min^M_{Min} \geq l \\ (l_i, w_i) \in S, w_i \leq u \end{cases}$$

[MinFwFF]$^{\mathsf{BF}}$

Figure 6.1: Transition rules for $Min$ aggregate expressions

SumBwTrue:

$$M \parallel \Psi \quad \Longrightarrow M\, d \parallel \Psi \quad \textbf{if} \begin{cases} (d \leftarrow l \leq Sum(S) \leq u) \in \Psi \\ M(d) = \boldsymbol{u} \\ S.min^M_{Sum} \geq l \text{ and } S.max^M_{Sum} \leq u \end{cases}$$

$[\mathsf{SumBwTrue}]^{\mathsf{BF}}$

SumBwFalse:

$$M \parallel \Psi \quad \Longrightarrow M\, \neg d \parallel \Psi \quad \textbf{if} \begin{cases} (d \leftarrow l \leq Sum(S) \leq u) \in \Psi \\ M(d) = \boldsymbol{u} \\ S.max^M_{Sum} < l \text{ or } S.max^M_{Sum} > u \end{cases}$$

$[\mathsf{SumBwFalse}]^{\mathsf{BF}}$

SumFwTT:

$$M \parallel \Psi \quad \Longrightarrow M\, l_i \parallel \Psi \quad \textbf{if} \begin{cases} (d \leftarrow l \leq Sum(S) \leq u) \in \Psi \\ M(d) = \boldsymbol{t}, (l_i, w_i) \in S, \\ M(l_i) = \boldsymbol{u}, S.min^M_{Sum} + w_i \leq u \\ \qquad \text{and } S.max^M_{Sum} - w_i < l \end{cases}$$

SumFwTF:

$$M \parallel \Psi \quad \Longrightarrow M\, \neg l_i \parallel \Psi \quad \textbf{if} \begin{cases} (d \leftarrow l \leq Sum(S) \leq u) \in \Psi \\ M(d) = \boldsymbol{t}, (l_i, w_i) \in S, \\ M(l_i) = \boldsymbol{u} \text{ and } S.min^M_{Sum} + w_i > u \end{cases}$$

SumFwFT:

$$M \parallel \Psi \quad \Longrightarrow M\, l_i \parallel \Psi \quad \textbf{if} \begin{cases} (d \leftarrow l \leq Sum(S) \leq u) \in \Psi \\ M(d) = \boldsymbol{f} \\ S.min^M_{Sum} \geq l \\ S.max^M_{Sum} - S'.max^M_{Min} \leq u \text{ and} \\ \qquad (l_i, w_i) \in S', \text{ where} \\ \quad S' = \{(l_j, w_j) \in S \mid M(l_j) = \boldsymbol{u}\} \end{cases}$$

SumFwFF:

$$M \parallel \Psi \quad \Longrightarrow M\, \neg l_i \parallel \Psi \quad \textbf{if} \begin{cases} (d \leftarrow l \leq Sum(S) \leq u) \in \Psi \\ M(d) = \boldsymbol{f} \\ S.max^M_{Sum} \leq u \\ S.min^M_{Sum} + S'.max^M_{Min} \geq l \text{ and} \\ \qquad (l_i, w_i) \in S', \text{ where} \\ \quad S' = \{(l_j, w_j) \in S \mid M(l_j) = \boldsymbol{u}\} \end{cases}$$

Figure 6.2: Transition rules for $Sum$ aggregate expressions

*known minimum* and *known maximum*, respectively. When $M$ is two-valued, the known minimum and maximum coincide.

Figure 6.1 defines transition rules for $Min$, and Figure 6.2 for $Sum$. Note that the SumFwFT and SumFwFF transition rules use the known maximum *of a $Min$ aggregate expression*, even though they are used for $Sum$: they use the value $S'.max^{M}_{Min}$, where $S' = \{(l_j, w_j) \in S \mid M(l_j) = \boldsymbol{u}\}$. This is the smallest weight of all unknown literals in $S^{\text{lits}}$.

The transition rules for $Min$ defined in Figure 6.1 have the same propagation efficiency as CNF unit propagation has on the formulas (6.8) from Section 6.3.2. The main advantage of the transition rules is that a weighted set $S$ may occur in multiple $Min$ expressions; evaluating $S.min^{M}_{Min}$ and $S.max^{M}_{Min}$ can be done once for all these expressions.

Observe that other transition rules are imaginable, for instance the following ones for $Sum$:

SumExhFw:

$$M \parallel \Psi \quad \Longrightarrow M\, \neg l_i \parallel \Psi \quad \textbf{if} \begin{cases} (d \leftarrow l \leq Sum(S) \leq u) \in \Psi \\ M(d) = \boldsymbol{t} \\ \text{For any 2-valued } M' \text{ extending } M\, l_i, \\ \qquad S.min^{M'}_{Sum} < l \text{ or } S.min^{M'}_{Sum} > u \end{cases}$$

SumExhBw:

$$M \parallel \Psi \quad \Longrightarrow M\, d \parallel \Psi \quad \textbf{if} \begin{cases} (d \leftarrow l \leq Sum(S) \leq u) \in \Psi \\ M(d) = \boldsymbol{u} \\ \text{For any 2-valued } M' \text{ extending } M, \\ \qquad l \leq S.min^{M'}_{Sum} \leq u. \end{cases}$$

These and other transition rules that do not depend on the known minimum and maximum, but on the actual sum values for two-valued extensions of $M$, are not efficiently implementable. Indeed, the problem of deciding whether SumExhBw (e.g.) is applicable generalizes the subset sum problem, which is NP-complete.

We illustrate the $Sum$ transition rules of Figure 6.2 on an example.

**Example 6.5.** Consider $\Psi_{6.5} =$

$$S = \{(a, 5), (b, 10)\}, \tag{6.10}$$
$$p \leftarrow 1 \leq Sum(S) \leq 6, \tag{6.11}$$
$$q \leftarrow 10 \leq Sum(S) \leq 15. \tag{6.12}$$

Then the following is a valid derivation:

$$p \parallel \Psi_{6.5} \overset{\textsf{SumFwTF}}{\Longrightarrow} p\,\neg b \parallel \Psi_{6.5} \overset{\textsf{SumFwTT}}{\Longrightarrow} p\,\neg b\,a \parallel \Psi_{6.5} \overset{\textsf{SumBwFalse}}{\Longrightarrow} p\,\neg b\,a\,\neg q \parallel \Psi_{6.5}.$$

The initial state sequence is $M_1 = \{p \mapsto \boldsymbol{t}\}$. The first transition applies the SumFwTF rule on (6.11): we have $M_1(p) = \boldsymbol{t}$, $(b, 10) \in S$, $M_1(b) = \boldsymbol{u}$, and $S.min^{M_1}_{Sum} = 0$, and because $0 + 10 > 6$, this transition derives $M_2 = M_1[b/\boldsymbol{f}]$.

The second transition applies the SumFwTT rule on (6.11): we have $M_2(p) = \boldsymbol{t}$, $(a, 5) \in S$, $M_2(a) = \boldsymbol{u}$, $S.min^{M_2}_{Sum} = 0$ and $S.max^{M_2}_{Sum} = 5$, and because both $0 + 5 \leq 6$ and $5 - 5 < 1$, this transition derives $M_3 = [a/\boldsymbol{t}]$.

The final transition applies the SumBwFalse rule on (6.12): we have $M_3(q) = \boldsymbol{u}$, $S.max^{M_3}_{Sum} = 5$, and $5 < 10$.

The algorithms to implement the transition rules of Figures 6.1 and 6.2 are easily derived from the conditions in the rules, and are straightforward to implement, once the appropriate data structures to represent aggregate expressions are in place. Each of these algorithms runs in linear time in the size of the weighted set $S$. The algorithms for MinFwTF, MinFwFF, SumFwTT and SumFwTF may derive a linear number of propagations.

### Data structures

We now present data structures that can be used to implement algorithms for the transition rules of Figures 6.1 and 6.2. We use the following presentation:

**⟨Data structure name⟩**

| | |
|---|---|
| ⟨data type⟩ | ⟨attribute name⟩ |
| ... | (a list of attributes and their corresponding data types) |
| ⟨data type⟩ | ⟨attribute name⟩ |

We represent aggregate expressions using the data structures "Aggregate expression" and "Aggregate set".

**Aggregate expression**

| | |
|---|---|
| int | `lwr`, `upr` |
| Literal | `defn` |

An aggregate expression *ae* denotes a specific expression of the form $\texttt{defn} \leftarrow (\texttt{lwr} \leq Aggr(S) \leq \texttt{upr})$. Both the specific type of aggregate *Aggr* and the set identifier $S$ are not represented by this data structure, and can be retrieved only in combination with the *aggregate set* data structure given below.

### Aggregate set

| | |
|---|---|
| [sum\|prod\|min\|max] | `type` |
| array⟨Literal⟩ | `set` |
| array⟨int⟩ | `weights` |
| array⟨Aggregate expression*⟩ | `exprs` |
| int | `min, max, cmax` |

An aggregate set $as$ denotes a subexpression of an aggregate expression, namely $Aggr(S)$. The specific type of aggregate $Aggr$ is represented by $as$.`type`, the weighted set $S$ is represented by $as$.`set` and $as$.`weights`. It represents by $as$.`exprs` in which aggregate expressions this aggregate set occurs. Finally, $as$.`cmax` has an auxiliary function; it is set equal to $S.max^{\emptyset}_{Aggr}$. It is advantageous to some of the algorithms to store the weighted set—`set` and `weights`—ordered by the weights.

The remaining parameters, $as$.`min` and $as$.`max`, are variable throughout the search, and approximate respectively the known minimum and maximum. For $Sum$ and $Prod$, they hold the actual value, while for $Min$ and $Max$ they hold the index of the *representative literal* of the actual value, which can therefore be found in the `weights` array. However, this index can also go out of bounds when all literals of the set are false: in case of $Min$, the index can be pushed beyond the end of the array, representing a value of $+\infty$; in case of $Max$, the index can be pushed beyond the beginning of the array, representing a value of $-\infty$.

To efficiently determine when and which propagations are possible, the following data structure will be used:

**Aggregate watch**

| | |
|---|---|
| Aggregate set* | `as` |
| Aggregate expression* | `ae` |
| [defn\|pos\|neg] | `type` |
| int | `index` |

Each atom $p$ stores a list of aggregate watches $aw$ that record in which aggregate expressions the atom or its negation occurs. If $aw$.`type`=‘defn’, then $p$ occurs as the defining literal of the aggregate expression $aw$.`ae` in aggregate set $aw$.`as`. Otherwise, $p$ occurs in the aggregate set $aw$.`as` on the $aw$.`index`’th position; positively if $aw$.`type`=‘pos’, negatively if $aw$.`type`=‘neg’.

**Example 6.6.** Example 6.5 continued. We represent $\Psi_{6.5}$ in these data structures:

- There is one aggregate set, $as$; it has `type`=‘sum’, `set`=$[a, b]$, `weights`= $[5, 10]$, `exprs`=$[ae_1, ae_2]$, and `cmax`=15.
- There are two aggregate expressions, $ae_1$ with `lwr`=1, `upr`=6, `defn`=$p$, and $ae_2$ with `lwr`=10, `upr`=15, `defn`=$q$.

- The literals $p$, $q$, $a$, and $b$ all have one aggregate watch: those of $p$ and $q$ have `type`=‘defn’ and have respectively `ae`=$ae_1$ and `ae`=$ae_2$ (and they don’t use `index`); those of $a$ and $b$ have `type`=‘pos’, `as`=$as$, and respectively `index`=0 and `index`=1 (and they don’t use `ae`).

The above data structures suffice to implement the transition rules of Figures 6.1 and 6.2 efficiently. Let $M'$ be some state sequence, and let $M$ be the state sequence $M'\,l$. Then the algorithms proceed by investigating all aggregate watches of the atom $\underline{l}$ in $l$: some of the aggregate expressions involved may derive new propagations from $M$.

### Approximating $S.min^{M}_{Aggr}$ and $S.max^{M}_{Aggr}$

Suppose literal $l$ became true, and its atom $\underline{l}$ has an aggregate watch $aw$ with $aw$.`type`≠‘defn’. Then either $aw$.`as.min` or $aw$.`as.max` can be updated—depending on whether $aw$.`type` is ‘pos’ or ‘neg’ and whether $l$ is an atom or its negation. For instance, in the case of $Sum$, the $aw$.`index`’th element of $aw$.`as.weights` is either added to $aw$.`as.min`, or subtracted from $aw$.`as.max`.

Note that an aggregate set’s `min` and `max` values therefore only *approximate* the theoretical known minimum and maximum values. More precisely, it is possible that some literals of an aggregate set already have a two-valued truth value, which however is not accounted for yet—these literals are still waiting to be propagated. Thus, `min` underestimates the known minimum, and `max` overestimates the known maximum.

Also note that upon backtracking, the reverse updates have to be executed.

For $Sum$ and $Prod$, these updates take constant time; for $Min$ and $Max$, they take time linear in the size of the set (however, by exploiting the fact that the set is ordered according to the weights, it is rarely necessary to visit each literal).

### Actual propagations

There are then two different types of aggregate propagation: backward and forward. The applicability of the backward propagations can easily be tested after either $aw$.`as.min` or $aw$.`as.max` has changed: for each aggregate expression $ae$ in $aw$.`as.exprs`, it is simply evaluated if $aw$.`as.min` $\geq$ $ae$.`lwr` $\wedge$ $aw$.`as.max` $\leq$ $ae$.`upr` (then MinBwTrue respectively SumBwTrue is applicable), or $aw$.`as.min` $>$ $ae$.`upr` $\vee$ $aw$.`as.max` $<$ $ae$.`lwr` (then MinBwFalse respectively SumBwFalse is applicable). Note that for this purpose, it may be beneficial to store two copies of `exprs` in each aggregate set: one ordered by the expression’s `lwr` value, the other by its `upr` value. Each of these propagations takes constant time.

Forward propagations may happen in all of the other cases: either for a literal with $aw$.`type`≠‘defn’, and any aggregate expression $ae$ in $aw$.`as.exprs` for which no backward propagation was possible and $ae$.`defn` is two-valued, or for a literal with $aw$.`type`=‘defn’, and its aggregate expression $aw$.`ae`. It suffices to first test whether $ae$.`defn` is true or false, and depending on that and on $ae$’s `lwr` and `upr` values, make the appropriate literals of $aw$.`as.set` true or false. For

instance, in the case of *Sum* and $ae$.`defn` is false, SumFwFT or SumFwFF may be applicable. First, the smallest weight $w$ of all unknown literals in $aw$.`as.set` is determined. If $aw$.`as.min` $\geq ae$.`lwr` and $aw$.`as.max` $- w \leq ae$.`upr`, then all unknown literals of $aw$.`as.set` are made true. If $aw$.`as.max` $\leq ae$.`upr` and $aw$.`as.min` $+ w \geq ae$.`lwr`, then all unknown literals of $aw$.`as.set` are made false.

Each of the forward propagations takes linear time in the size of the set (often for a linear number of propagations). The ordering of the set can be exploited to limit the search.

### Integration with conflict-driven DPLL solver

To integrate these propagations into a DPLL-based SAT solver that applies conflict-driven clause learning and backjumping (see Section 5.6), a *reason clause* should be associated to each propagation: a clause from which the propagated literal can be derived by unit propagation.

**Example 6.7.** Examples 6.5 and 6.6 continued. A reason clause for the propagation $p \parallel \Psi_{6.5} \overset{\mathsf{SumFwTF}\text{ on (6.11)}}{\Longrightarrow} p\,\neg b \parallel \Psi_{6.5}$ is $\neg b \vee \neg p$. A reason clause for the propagation $p\,\neg b \parallel \Psi_{6.5} \overset{\mathsf{SumFwTT}\text{ on (6.11)}}{\Longrightarrow} p\,\neg b\, a \parallel \Psi_{6.5}$ is $a \vee b \vee \neg p$. Finally, a reason clause for the propagation $p\,\neg b\, a \parallel \Psi_{6.5} \overset{\mathsf{SumBwFalse}\text{ on (6.12)}}{\Longrightarrow} p\,\neg b\, a\, \neg q \parallel \Psi_{6.5}$ is $\neg q \vee b$. Note that $a$ does not appear in this clause.

It is possible to create such reason clauses at the time an aggregate-based propagation is made. However, the majority of such clauses are never used. It is therefore preferable to create reason clauses *on demand*—more specifically, (only) when the clause learning process attempts to apply resolution on such a clause. To enable such on demand reason clause generation, the following data structures can be used:

**Propagation-info**

| | |
|---|---|
| Literal | `lit` |
| int | `weight` |
| [defn\|pos\|neg] | `type` |

**Aggregate reason**

| | |
|---|---|
| Aggregate set* | `as` |
| Aggregate expression* | `ae` |
| [defn\|pos\|neg] | `type` |

The aggregate set data structure is extended with a stack of Propagation-info's, recording the aggregate propagations related to that set, in the order in which they occurred. An aggregate reason represents, for each aggregate propagation, which aggregate expression was used, and what type of propagation

it was. Each two-valued atom that was derived by an aggregate propagation should record its aggregate reason.

Using this combined information, it is possible to reconstruct a reason clause for any literal that has an aggregate reason $ar$. Specifically, the reason clause is built from the empty clause by:

- adding the literal itself;
- if $ar$.`type` $\neq$'defn', adding the literal $ar$.`ae.defn` or its negation (the added literal should be false);
- going over $ar$.`as`'s stack of Propagation-info's $pi$, adding each $pi$.`lit` if it is of the right type, and until the reason clause is complete. The "right type" and "complete" are derived from $ar$.`type`, and from the properties of $ar$.`as`. (It is here that the `cmax` value of an aggregate set is used.)

**Example 6.8.** Examples 6.5–6.7 continued. After the first propagation, from $p$ to $\neg b$, the following aggregate reason $ar_1$ is recorded for $b$: $ar_1$.`as`=$as$ (which points to the set $S = \{(a, 5), (b, 10)\}$), $ar_1$.`ae`=$ae_1$ (which points to the expression with head $p$), and $ar_1$.`type`='pos', because $b$ occurs positively in $S$. Also, a propagation-info $pi_1$ is added to $as$'s list of propagation-infos; $pi_1$.`lit`=$\neg b$, $pi_1$.`weight`=10, and $pi_1$.`type`='neg', because $b$ occurs positively, and $\neg b$ was derived.

The next propagation derives $a$ from $p$ and $\neg b$. We have the aggregate reason $ar_2$ with $ar_2$.`as`=$as$, $ar_2$.`ae`=$ae_1$, and $ar_2$.`type`='pos', and the propagation-info $pi_2$ with $pi_2$.`lit`=$a$, $pi_2$.`weight`=5, and $pi_2$.`type`='pos'.

The final propagation derives $\neg q$ from $\neg b$. We have the aggregate reason $ar_3$ with $ar_3$.`as`=$as$, $ar_3$.`ae`=$ae_2$ (which points to the expression with head $q$), and $ar_3$.`type`='defn', and the propagation-info $pi_3$ with $pi_3$.`lit`=$b$, $pi_3$.`weight` is unused (take, e.g., 0), and $pi_3$.`type`='defn'.

Suppose a reason clause for $\neg q$ is required. The clause certainly contains the literal $\neg q$. $q$'s aggregate reason $ar_3$ is used to construct the rest of the reason clause. Since $ar_3$.`as.type`='sum', $ar_3$.`type`='defn' and $q$ is false, it can be derived that SumBwFalse was used to derive $\neg q$. Because the current known maximum $ar_3$.`as.max` is 5, which is smaller then $ar_3$.`ae.lwr`=10, the algorithm searches for enough false literals in $ar_3$.`as.set` to make the maximum smaller then 10. The initial known maximum is $ar_3$.`as.cmax`=15. It goes over $ar_3$.`as`'s list of propagation-infos, collecting literals with type 'neg'. The first one encountered is $\neg b$; the algorithm adds $b$ to the reason clause. The propagation-info of $\neg b$ has weight 10. We find that after the propagation to $\neg b$, the known maximum was $15 - 10 = 5$, which is already smaller than 10. Therefore, the clause is finished: $\neg q \vee b$.

### 6.3.4 Recursive aggregate expressions

#### Generalizing justification semantics

We now generalize the justification semantics of Section 4.3 to ECNF definitions that may contain aggregate expressions.

We extend Definition 4.1 of a direct justification with a case for atoms that are defined by an aggregate expression. The intuition is as before: a direct justification for a defined literal is a set of literals, such that the truth of all literals is a sufficient condition for the truth of the defined literal. For brevity we again only define the concept for $Sum$ and $Min$.

**Definition 6.2** (Direct justification)**.** Let $\Delta$ be an ECNF definition. Let $l \in \widehat{Def(\Delta)}$. A *direct justification for* $l$ is a set of literals $DJ(l)$ such that:

- if $(l \leftarrow \bigvee \varphi_l) \in \Delta$, $DJ(l)$ is a singleton $\subset \varphi_l$;
- if $(l \leftarrow \bigwedge \varphi_l) \in \Delta$, $DJ(l) = \varphi_l$;
- if $l = \neg d$ and $(d \leftarrow \bigvee \varphi_d) \in \Delta$, $DJ(l) = \overline{\varphi_d}$;
- if $l = \neg d$ and $(d \leftarrow \bigwedge \varphi_d) \in \Delta$, $DJ(l)$ is a singleton $\subset \overline{\varphi_d}$.

If $l \in \mathcal{A}_{\text{lits}}$, then $DJ(l)$ is the union of a *positive* and a *negative direct justification for* $l$, denoted respectively $DJ^+(l)$ and $DJ^-(l)$, such that:

- if $(l \leftarrow lwr \leq Sum(S) \leq upr) \in \Delta$, then
  - $DJ^+(l) \subseteq S^{\text{lits}}$, $DJ^-(l) \subseteq \overline{S^{\text{lits}}}$,
  - $\sum_{(l_i,w_i)\in S,\ l_i \in DJ^+(l)} w_i \quad \geq lwr$, and
  - $\sum_{(l_i,w_i)\in S,\ \neg l_i \notin DJ^-(l)} w_i \quad \leq upr$;
- if $(l \leftarrow lwr \leq Min(S) \leq upr) \in \Delta$, then
  - $DJ^+(l) \subseteq \{l_i \mid (l_i, w_i) \in S, lwr \leq w_i \leq upr\}$, and
  - $DJ^-(l) = \{\neg l_i \mid (l_i, w_i) \in S, w_i < lwr\}$.
- if $l = \neg d$ and $(d \leftarrow lwr \leq Sum(S) \leq upr) \in \Delta$, then either
  - $DJ^-(l) = \emptyset$, $DJ^+(l) \subseteq S^{\text{lits}}$ and $\sum_{(l_i,w_i)\in S,\ l_i \in DJ^+(l)} w_i \ > upr$, or
  - $DJ^+(l) = \emptyset$, $DJ^-(l) \subseteq \overline{S^{\text{lits}}}$ and $\sum_{(l_i,w_i)\in S,\ \neg l_i \notin DJ^-(l)} w_i \ < lwr$;
- if $l = \neg d$ and $(d \leftarrow lwr \leq Min(S) \leq upr) \in \Delta$, then either
  - $DJ^-(l) = \emptyset$ and $DJ^+(l)$ is a singleton $\{l_i\} \subset \varphi_d$ such that $w_i < lwr$, where $(l_i, w_i) \in S$, or
  - $DJ^+(l) = \emptyset$ and $DJ^-(l) = \{\neg l_i \mid (l_i, w_i) \in S, w_i \leq upr\}$.

Observe that if $\Delta$ is an aggregate-free DefNF definition, and $J$ a justification for $\Delta$, then for all literals $l$ in $\mathcal{D}_{\text{lits}}$ or $\mathcal{C}_{\text{lits}}$, the set of literals $\{l' \mid (l, l') \in J\}$ is a direct justification $DJ(l)$ for $l$.

We restate the intuition: a direct justification $DJ(l)$ for a defined literal $l$ is such that if $\bigwedge DJ(l)$ is true, $l$ must be true. One can verify that indeed, whenever $M(\bigwedge DJ(l)) = \boldsymbol{t}$, one of the —Bw— transition rules can be applied to derive $M(l) = \boldsymbol{t}$. Observe that in the case of $Sum$ and $Prod$, a direct justification may contain both a literal and its negation. Such a direct justification can only be a sufficient condition for the truth value unknown.

The notion of justification (Definition 4.2) can now easily be extended:

**Definition 6.3** (Justification, extended)**.** Let $\Delta$ be an ECNF definition over $\Sigma$. Then a *justification for* $\Delta$ is a directed graph $(V, E)$ with $V = \widehat{\Sigma}$ and $E$ such that:

- for each $l \in \widehat{Def(\Delta)}$, the set $\{l' \mid (l, l') \in E\}$ is a direct justification for $l$;
- for each $l \in \widehat{Open(\Delta)}$, there are no edges in $E$ leaving from $l$.

A *stable* justification for $\Delta$ is a directed graph $(V, E)$ with $V = \widehat{\Sigma}$ and $E$ such that:

- for each $l \in Def(\Delta)$, the set $\{l' \mid (l, l') \in E\}$ is a direct justification for $l$;
- for each negative literal $l$ and for each $l \in Open(\Delta)$, there are no edges in $E$ leaving from $l$.

We simply retain the earlier notions of (three-valued) support, loop-safeness, and witness. Three-valued support formalizes the above intuition: whenever $\bigwedge DJ(l)$ is two-valued in the given interpretation, $l$ must have the same truth value.[3] Loop-safeness requires that for any non-negative loop the justification contains, all literals in it are false. A witness of an interpretation is simply a justification that three-valuedly supports the interpretation, and is loop-safe in it.

**Example 6.9.** Example 6.2 continued. We represent definition (1.1), grounded with the given domain, in ECNF. For simplicity of representation, we treat $\top$ as an atom, and we substitute the declared sets in the aggregate expressions. $\Delta_{6.9} =$

$$\left\{\begin{array}{l}
C_{AA} \leftarrow 51 \leq Sum(\{(\top,20),(C_{AB},40),(C_{AC},40)\}) \leq 100,\\
C_{AB} \leftarrow 51 \leq Sum(\{(\top,26),(C_{AB},49),(C_{AC},25)\}) \leq 100,\\
C_{AC} \leftarrow 51 \leq Sum(\{(\top,60),(C_{AB},10),(C_{AC},30)\}) \leq 100,\\
C_{BA} \leftarrow 51 \leq Sum(\{(C_{BA},20),(\top,40),(C_{BC},40)\}) \leq 100,\\
C_{BB} \leftarrow 51 \leq Sum(\{(C_{BA},26),(\top,49),(C_{BC},25)\}) \leq 100,\\
C_{BC} \leftarrow 51 \leq Sum(\{(C_{BA},60),(\top,10),(C_{BC},30)\}) \leq 100,\\
C_{CA} \leftarrow 51 \leq Sum(\{(C_{CA},20),(C_{CB},40),(\top,40)\}) \leq 100,\\
C_{CB} \leftarrow 51 \leq Sum(\{(C_{CA},26),(C_{CB},49),(\top,25)\}) \leq 100,\\
C_{CC} \leftarrow 51 \leq Sum(\{(C_{CA},60),(C_{CB},10),(\top,30)\}) \leq 100
\end{array}\right\}.$$

[3] Recall that if $\bigwedge DJ(l)$ is false, it is not necessarily a sufficient condition for the falsity of $l$—such condition must be given by the truth of $\bigwedge DJ(\neg l)$.

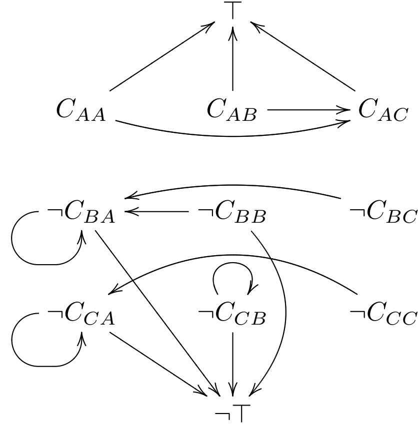



Figure 6.3: A partial justification for $\Delta_{6.9}$.

Figure 6.3 illustrates a partial justification $J$ for $\Delta_{6.9}$ that supports the model $I = \{C_{AA} \mapsto \boldsymbol{t}, C_{AB} \mapsto \boldsymbol{t}, C_{AC} \mapsto \boldsymbol{t}, C_{BA} \mapsto \boldsymbol{f}, C_{BB} \mapsto \boldsymbol{f}, C_{BC} \mapsto \boldsymbol{f}, C_{CA} \mapsto \boldsymbol{f}, C_{CB} \mapsto \boldsymbol{f}, C_{CC} \mapsto \boldsymbol{f}, \top \mapsto \boldsymbol{t}\}$. We have included only the literals that are true.

We give some examples: the direct justification in $J$ of $C_{AC}$ is $DJ(C_{AC}) = \{\top\}$. In the weighted set in $C_{AC}$'s body, $\top$ has weight 60, which is indeed $\geq 51$. This justifies the truth of $C_{AC}$. We have $DJ(C_{AA}) = \{\top, C_{AC}\}$; $\top$ and $C_{AC}$ have weights 20 and 40, totalling 60, which is $\geq 51$, therefore justifying $I(C_{AA}) = \boldsymbol{t}$. We also have $DJ(\neg C_{BC}) = \{\neg C_{BA}\}$; the weights of literals other than $C_{BA}$ in $C_{BC}$'s weighted set are 10 and 30, totalling less than 51. This justifies $I(\neg C_{BC}) = \boldsymbol{t}$, or equivalently, $I(C_{BC}) = \boldsymbol{f}$.

We have the following generalization of Theorem 4.1:

**Theorem 6.1.** *Let $\Delta$ be an ECNF definition over $\Sigma$, $M$ a two-valued $\Sigma$-interpretation. Then $M \models \Delta$ iff there exists a $\Delta$-witness for $M$.*

*Proof.* (Sketch)
The proof is an extension of the proof of Theorem 4.1. The notion of three-valued support for a justification agrees with the three-valued semantics for aggregate expressions defined in Section 6.2.2. Both the definition of the semantics of definitions with aggregate expressions, and the proof of Theorem 4.1 depend on the notion of well-founded sequence, which has retained its original meaning. □

### Generalizing loop formula semantics

We do not generalize all results of Section 4.4 on loop formula semantics to the aggregate case. However, we do define a specific type of generalized loop formulas, namely *generalized stable loop formulas*, and show that they are entailed by

the definition. These loop formulas suffice for the purpose of extending SAT(ID) solvers that use a stable strategy, such as MINISAT(ID), with aggregate expressions.

We start by generalizing the notion of dependency graph.

**Definition 6.4.** The *generalized dependency graph of* $\Delta$ is the graph with literals $\widehat{\Sigma}$ and edges $(l, l')$ for each defined literal $l \in \widehat{Def(\Delta)}$, and for each $l'$ in some direct justification $DJ(l)$ for $l$.

Observe that for literals $l$ in $\mathcal{D}_{\text{lits}}$ and in $\mathcal{C}_{\text{lits}}$, the edges in the dependency graph are simply $(l, l')$, for any $l' \in \varphi_l$. Thus a generalized dependency graph indeed generalizes a dependency graph.

The *loops* of $\Delta$ are, as before, the loops in the dependency graph of $\Delta$. Note that when a literal $l \in \mathcal{A}_{\text{lits}}$ is in a loop $L$ of $\Delta$, at least one positive or negative direct justification for $l$ contains an element of $L$, but not necessarily both. We denote by $All^+(l)$ the set $\{l' \mid l' \in DJ^+(l)$ for some positive direct justification $DJ^+(l)$ for $l\}$, and by $All^-(l)$ the set $\{l' \mid l' \in DJ^-(l)$ for some negative direct justification $DJ^-(l)$ for $l\}$.

We generalize the notion of *external disjuncts* of a loop for positive loops. Recall the intuition that literals in a relevant loop get their justification to be true from the external disjuncts of the loop.

**Definition 6.5** (Generalized external disjuncts)**.** Let $L$ be a positive loop of $\Delta$. The set of *generalized external disjuncts of* $L$, denoted $\mathcal{G}^{\text{ext}}(L)$, is the set of literals $l'$ such that:

- $l' \in \varphi_l \setminus L$ for some $l \in L \cap \mathcal{D}_{\text{lits}}$; or
- $l' \in All^+(l) \setminus L$ for some $l \in L \cap \mathcal{A}_{\text{lits}}$ with $All^+(l) \cap L \neq \emptyset$; or
- $l' \in All^-(l) \setminus L$ for some $l \in L \cap \mathcal{A}_{\text{lits}}$ with $All^-(l) \cap L \neq \emptyset$.

We explain the intuition behind this definition. For some atom $l \in L \cap \mathcal{A}_{\text{lits}}$, the set $All^+(l) \cup All^-(l)$ contains all literals that could possibly be used in some direct justification for $l$. The literals $l' \in All^+(l) \cup All^-(l) \setminus L$ can therefore contribute to the external justification of literals in $L$.

We further explain why we split this definition in two cases, for $All^+(l)$ and for $All^-(l)$. Consider a rule $l \leftarrow lwr \leq Aggr(S) \leq upr$. Intuitively, it contains two subexpressions $lwr \leq Aggr(S)$ and $Aggr(S) \leq upr$, and could be written as $l \leftarrow l_1 \wedge l_2$, $l_1 \leftarrow lwr \leq Aggr(S)$, $l_2 \leftarrow Aggr(S) \leq upr$, where $l_1$ and $l_2$ are new atoms. In this view, and if $Aggr \in \{Sum, Prod, Max\}$, the set $All^+(l)$ contains all literals that can help justify $l_1$, and the set $All^-(l)$ contains all literals that can help justify $l_2$ (and the other way round if $Aggr = Min$). If $All^+(l) \cap L = \emptyset$, therefore, the literals $l' \in All^+(l) \setminus L$ need not be in the generalized external disjuncts, but can instead be considered as generalized external *conjuncts* (since $l_1 \wedge l_2$ is a conjunction); similarly for $All^-(l)$.

Note that for an aggregate-free definition $\Delta$ and a loop $L$ in $\Delta$, $\mathcal{G}^{\text{ext}}(L) = \mathcal{D}^{\text{ext}}(L)$. We retain the original intuition: when all (generalized) external disjuncts of a loop are false, none of the literals in the loop has a justification to be true. A generalized stable loop formula embodies this intuition.

**Definition 6.6** (Generalized stable loop formula)**.** Let $L$ be a positive loop of $\Delta$. Then the *generalized stable loop formula* $GLF_\Delta(L)$ of $L$ is the formula

$$\bigvee L \supset \bigvee \mathcal{G}^{\text{ext}}(L).$$

For an aggregate-free definition $\Delta$ and a positive loop $L$ in $\Delta$, the generalized stable loop formula of $L$ is exactly the loop formula of $L$. Nevertheless, generalized stable loop formulas are in some sense less general than loop formulas for aggregate-free definitions. Indeed, in an interpretation $M$ with $M \models \bigvee L$, $M \models GLF_\Delta(L)$ does not guarantee that for some $l \in L$, *all* literals in the direct justification for $l$ are true.[4] For this reason, and because the formulas are defined for stable loops only, we have no result of the form $comp(\Delta) \cup \Phi \models \Delta$, with $\Phi$ some set of loop formulas.

We do show, however, that a definition entails its generalized stable loop formulas.

**Proposition 6.1.** *Let $L$ be a positive loop of $\Delta$. Then $\Delta \models GLF_\Delta(L)$.*

*Proof.* Assume towards contradiction that $M$ is an interpretation such that $M \models \Delta$ and $M \not\models GLF_\Delta(L)$. Because $M \not\models GLF_\Delta(L)$, we have $M \models \bigvee L$ and $M \models \neg \bigvee \mathcal{G}^{\text{ext}}(L)$.

We show that then there cannot exist a $\Delta$-witness for $M$, which contradicts $M \models \Delta$ by Theorem 6.1. Indeed, consider an atom $l \in L$ with $M(l) = \boldsymbol{t}$, and consider a justification $J$ for $\Delta$ that supports $M$. Then the direct justification $DJ(l)$ of $l$ in $J$ has $M \models \bigwedge DJ(l)$. By our assumption that $M \models \neg \bigvee \mathcal{G}^{\text{ext}}(L)$ and by the definition of generalized external disjuncts, we find that $DJ(l) \subseteq L$. Therefore there exists a positive loop of true atoms in $J$, and thus $J$ is not a witness for $M$. Contradiction. ☐

**Example 6.10.** Example 6.2 and 6.9 continued. Consider again the following two rules of $\Delta_{6.9}$:

$$C_{CA} \leftarrow 51 \leq Sum(\{(C_{CA}, 20), (C_{CB}, 40), (\top, 40)\}) \leq 100,$$
$$C_{CB} \leftarrow 51 \leq Sum(\{(C_{CA}, 26), (C_{CB}, 49), (\top, 25)\}) \leq 100.$$

A positive direct justification for $C_{CA}$ must contain enough literals such that their combined weights is $\geq 51$, and similarly for $C_{CB}$. A negative direct justification for $C_{CA}$ must contain enough negative literals such that the sum of weights of the remaining (non-negated) literals is $\leq 100$. These are all possible positive direct justifications, both for $C_{CA}$ and for $C_{CB}$:

$$\{C_{CA}, C_{CB}\}, \{C_{CA}, \top\}, \{C_{CB}, \top\}, \{C_{CA}, C_{CB}, \top\}.$$

Any subset of $\{\neg C_{CA}, \neg C_{CB}, \neg\top\}$ is a negative direct justification, also both for $C_{CA}$ and for $C_{CB}$. Note in particular that the empty set is a negative direct justification for $C_{CA}$ and for $C_{CB}$.

[4]But the stable loop formulas are most useful in the other direction: $M \models GLF_\Delta(L)$ and $M \models \neg \bigvee \mathcal{G}^{\text{ext}}(L)$ entails $M \models \neg \bigvee L$.

The set $L = \{C_{CA}, C_{CB}\}$ is an example of a positive loop of $\Delta_{6.9}$. It has $\mathcal{G}^{\text{ext}}(L) = \{\top\}$. Note that none of the literals $\neg C_{CA}, \neg C_{CB}, \neg\top$ occurs in any positive direct justification of either $C_{CA}$ or $C_{CB}$. Therefore the generalized stable loop formula of $L$ is $(C_{CA} \vee C_{CB}) \supset \top$.

#### Transition rules for generalized stable loop formulas

Based on the results of previous section, we will now define a generalization of the BwLoop transition rule for positive loops. The BwAggrLoop and its backtracking counterparts $[\mathsf{BwAggrLoop}]^{\mathsf{BF}}$ are defined by

$$M \parallel \Psi, \Delta \quad \Longrightarrow M\,\neg l \parallel \Psi, \Delta \qquad \textbf{if} \begin{cases} M(l) = \boldsymbol{u} \\ \Delta \text{ contains a positive loop } L \text{ such} \\ \text{that } M(\bigvee \mathcal{G}^{\text{ext}}(L)) = \boldsymbol{f} \text{ and } l \in L. \end{cases}$$

We also generalize the AddLF rule to AddAggrLF:

$$M \parallel \Psi, \Delta \quad \Longrightarrow M \parallel \Psi, \Delta, LF \quad \textbf{if} \begin{cases} LF = \neg l \vee \bigvee \mathcal{G}^{\text{ext}}(L) \\ L \text{ is a positive loop in } \Delta, l \in L \\ LF \notin \Psi. \end{cases}$$

Observe that in neither case, the new transition rule fully generalizes the old one, because the conditions exclude mixed loops. In order to obtain a correct system for definitions that contain mixed loops, we add the following transition rule, WellFounded:

$$M \parallel \Psi, \Delta \qquad \Longrightarrow \emptyset \parallel \Psi, \Delta, LF \qquad \textbf{if} \begin{cases} M \text{ is two-valued, and } M \not\models \Delta \\ LF = \bigvee\{\neg l \mid M(l) = \boldsymbol{t}\}. \end{cases}$$

The WellFounded transition rule can be considered as a final check for totality, as was the case for the final application of Simplify in MiniSat(ID)'s transition system outlined in Section 5.7.

#### Algorithms: finding direct justifications

We will develop algorithms for these transition rules. They will be based on the assumption that neither the $[\mathsf{UnitProp}]^{\mathsf{BF}}$ or $[\mathsf{UnitPropDef}]^{\mathsf{BF}}$, nor the transition rules defined in Figures 6.1 and 6.2 are applicable on the current state. We call such a state *UPA-saturated*.

To implement the WellFounded transition rule, we have to generalize Algorithm 5.4 to definitions with aggregate expressions. To implement the AddAggrLF and $[\mathsf{BwAggrLoop}]^{\mathsf{BF+}}$ rules, we have to generalize Algorithm 5.2.

**Algorithm 6.1**: BwAggrLoop—Justifying a cycle source $cs$

```
1  Let HP := {l | l has a path to cs in J_s through non-false atoms};
2  if cs ∉ HP then return ∅;
3  Let L := Q := {cs};
4  while Q ≠ ∅ do
5  |   Pop literal l from Q;
6  |   Find a supporting direct justification for l using only literals ∉ HP;
7  |   if Succeeded then Justify(l);
8  |   else Add HP body literals of l to Q and to L;
9  return L;
10 function Justify l
11 |   Remove l from HP, L and Q;
12 |   if l = cs then return ∅;
13 |   foreach l' ∈ L that has l as a body literal and that can now be
   |   justified do Justify(l');
14 end function
```

Recall the intuition behind Algorithm 5.2: we start from a (non-false) *cycle source*, and find a set of (non-false) literals that might contain a loop through it. Then we try to *justify* literals of that set. A literal is justified once it is certain that the current justification contains no path from this literal to the cycle source. To justify a disjunctively defined literal, we set the direct justification for it to a literal outside of the set; to justify a conjunctively defined literal, we prove that all its body literals are outside of the set.

Also the intuition behind Algorithm 5.4 is based on justifying literals: in that algorithm, we try to justify *all* defined literals. The justification we thereby construct is a directed acyclic graph.

We present a simplified version of Algorithm 5.2 in Algorithm 6.1, where some parts of the algorithm, which we will explain in more detail, have been made generic.

We explain Lines 6 and 8 of Algorithm 6.1. The purpose of Line 6 is to try to justify $l$, in other words, to find a direct justification $DJ(l)$ for $l$ that contains only literals $\notin HP$, and with $M(\bigwedge DJ(l)) = \boldsymbol{u}$ or $\boldsymbol{t}$. In the case of $l \in \mathcal{D}_{\text{lits}}$, this can be done (as in Algorithm 5.2) by finding *one* literal $l'$ in $\varphi_l \setminus HP$ with $M(l') \neq \boldsymbol{f}$; in the case of $l \in \mathcal{C}_{\text{lits}}$, this can be done by showing that *all* literals $l' \in \varphi_l$ are not in $HP$. (Note that by UPA-saturation and by the fact that $M(l) \neq \boldsymbol{f}$, these literals $l'$ all have $M(l') \neq \boldsymbol{f}$.) If such a $DJ(l)$ is found, $l$ can then be removed from $HP$, since it no longer has a path to $cs$ in the current justification. If such a $DJ(l)$ cannot be found, then all literals that need to be justified first before such a $DJ(l)$ could possibly be found need to be added to the queue, $Q$, for future consideration, and to the possible eventual loop, $L$.

Line 13 then applies the same search for a direct justification as in Line 6; the fact that $l$ has just been removed from $HP$ (Line 11) makes that this search

**Algorithm 6.2**: Finding $DJ(l)$ for $l \leftarrow lwr \leq Sum(S) \leq upr$

```
1  min := 0;
2  max := Σ_{(l_i,w_i)∈S} w_i;
3  DJ := ∅;
4  AuxL := ∅;
5  foreach (l_i, w_i) ∈ S do
6      if min ≥ lwr and max ≤ upr then Break;
7      if M(l_i) ≠ f then
8          if l_i ∈ HP then Add l_i to AuxL;
9          else Add l_i to DJ; Let min := min + w_i;
10     if M(l) ≠ t then
11         if ¬l_i ∈ HP then Add ¬l_i to AuxL;
12         else Add ¬l_i to DJ; Let max := max − w_i;
13 if min ≥ lwr and max ≤ upr then
14     DJ(l) := DJ;
15     return true;
16 else
17     Add AuxL \ L to Q and to L;
18     return false;
```

**Algorithm 6.3**: Finding $DJ(l)$ for $l \leftarrow lwr \leq Min(S) \leq upr$

```
1  DJ := {¬l_i | (l_i, w_i) ∈ S, w_i < lwr};
2  AuxL := DJ ∩ HP;
3  FoundPos := false;
4  foreach (l_i, w_i) ∈ S with lwr ≤ w_i ≤ upr do
5      if M(l_i) ≠ f then
6          if l_i ∈ HP then Add l_i to AuxL;
7          else Add l_i to DJ; FoundPos := true; Break;
8  if FoundPos ∧ (DJ ∩ HP = ∅) then DJ(l) := DJ;
9  else Add AuxL \ L to Q and to L;
10 return FoundPos;
```

may now be successful.

To extend Algorithm 5.2 (or equivalently, Algorithm 6.1) to arbitrary ECNF definitions, therefore, we have to extend Line 6 to atoms (recall that we look for positive loops) defined by an aggregate expression.

Algorithms 6.2 and 6.3 perform this task for atoms defined by $Sum$ or $Min$. Similar algorithms for $Prod$ and $Max$ are easy to derive. These algorithms return $true$ iff a direct justification satisfying the above conditions has been found; in the other case, they add the appropriate literals to $Q$ and to $L$ (i.e.,

**Algorithm 6.4**: Finding $DJ(l)$ for $(\neg l) \leftarrow lwr \leq Sum(S) \leq upr$

```
1  min := 0; max := Σ_{(l_i,w_i)∈S} w_i;
2  DJ_min := ∅; DJ_max := ∅;
3  AuxL := ∅;
4  foreach (l_i, w_i) ∈ S do
5      if min > upr or max < lwr then Break;
6      if M(l_i) ≠ f then
7          if l_i ∈ HP then Add l_i to AuxL;
8          else Add l_i to DJ_min; Let min := min + w_i;
9      if M(l_i) ≠ t then
10         if ¬l_i ∈ HP then Add ¬l_i to AuxL;
11         else Add ¬l_i to DJ_max; Let max := max − w_i;
12 if min > upr or max < lwr then
13     if min > upr then DJ(l) := DJ_min;
14     else DJ(l) := DJ_max;
15     return true;
16 else
17     Add AuxL \ L to Q and to L;
18     return false;
```

**Algorithm 6.5**: Finding $DJ(l)$ for $(\neg l) \leftarrow lwr \leq Min(S) \leq upr$

```
1  AuxL := ∅;
2  foreach (l_i, w_i) ∈ S such that w_i < lwr do
3      if M(l_i) ≠ f then
4          if l_i ∈ HP then Add l_i to AuxL;
5          else DJ(l) := {l_i}; return true;
6  DJ := {¬l_i | w_i ≤ upr};
7  if M(⋀DJ) = f or (DJ ∩ HP) ≠ ∅ then
8      Add AuxL \ L to Q and to L;
9      Add (DJ ∩ HP) \ L to Q and to L;
10     return false;
11 DJ(l) := DJ;
12 return true;
```

they perform both Lines 6 and 8). Note that Algorithm 6.3 may exploit the fact that weighted sets are ordered by weight to minimize the work.

These algorithms are linear in the size of the aggregated set. Line 13 of Algorithm 6.1 can call these algorithms $|S^{\text{lits}}|$ times for each atom $l$ defined by an aggregate expression over $S$. Therefore Algorithm 6.1 is quadratic in the size of the definition, restricted to the initial set $HP$.

To extend also Algorithm 5.4, in order to implement the WellFounded transition rule, we need similar algorithms for negative literals defined by an aggregate expression. Algorithms 6.4 and 6.5 provide such algorithms for the case of $Sum$ and $Min$.

*Remark* 6.5. Note that we need a different direct justification for each aggregate *expression*, not for each set over which aggregation is done. Whereas the UPA propagations benefit from the reuse of the values $S.min^M_{Aggr}$ and $S.max^M_{Aggr}$ in different aggregate expressions over the set $S$, no such benefit can be obtained for the $[\mathsf{BwAggrLoop}]^{\mathsf{BF}}$ propagations.

### Cycle sources

We recall also Definition 5.4 of *cycle sources*. For given interpretations $M'$ and $M$, where $M'$ has a witness $J_w$, and $M$ is an extension of $M'$, we defined $CS$ as $\{l \in \mathcal{D}_{\text{lits}} \mid M(l) \neq \boldsymbol{f} \wedge M'(d_{J_w}(l)) \neq \boldsymbol{f} \wedge M(d_{J_w}(l)) = \boldsymbol{f}\}$. The intuition behind this concept was that each eventual loop that could lead to propagation contains a $cs \in CS$.

The assumption behind Algorithm 5.2 was that a justification $J_s$ that supports $M$ is initially given; we remarked that $CS \subseteq \{l \mid J_w(l) \neq J_s(l)\}$. In practice, the tasks of finding a supporting justification $J_s$ and finding the set of cycle sources $CS$ can be combined, since they both require investigating truth values of defined literals, and investigating $J_w$.

We extend both tasks to arbitrary ECNF definitions. Observe that finding a supporting justification $J_s$ requires that the given state is UPA-saturated.

We now generalize the definition: $CS = \{l \in \widehat{Def(\Delta)} \mid M(l) \neq \boldsymbol{f} \wedge M'(\bigwedge_{(l,l')\in J_w} l') \neq \boldsymbol{f} \wedge M(\bigwedge_{(l,l')\in J_w} l') = \boldsymbol{f}\}$. Again, for literals that are not a cycle source, their direct justification can remain as it was in $J_w$.

Let $l \in \widehat{Def(\Delta)}$ be a literal with $M \neq \boldsymbol{f}$. The following simple definitions of specific direct justifications $DJ(l)$ suffice to find a supportive justification $J_s$.

1. if $M(\bigwedge_{(l,l')\in J_w} l') \neq \boldsymbol{f}$, retain this direct justification ($DJ(l) = \{l' \mid (l,l') \in J_w\}$);[5]

2. else,

   - if $l$ is defined by $l \leftarrow lwr \leq Sum(S) \leq upr$, then let $DJ(l)$ be $\{l_i \in S^{\text{lits}} \mid M(l_i) \neq \boldsymbol{f}\} \cup \{\neg l_i \in S^{\text{lits}} \mid M(l_i) \neq \boldsymbol{t}\}$;
   - if $l$ is defined by $(\neg l) \leftarrow lwr \leq Sum(S) \leq upr$, let $DJ(l)$ be either $\{l_i \in S^{\text{lits}} \mid M(l_i) \neq \boldsymbol{f}\}$ (if the corresponding sum of weights is $> upr$) or $DJ(l) = \{\neg l_i \in S^{\text{lits}} \mid M(l_i) \neq \boldsymbol{t}\}$ (if the corresponding sum of weights is $< lwr$);
   - if $l$ is defined by $l \leftarrow lwr \leq Min(S) \leq upr$, let $DJ(l)$ be $\{\neg l_i \mid (l_i, w_i) \in S, M(l_i) \neq \boldsymbol{f}, w_i < lwr\}$;

[5]Note that for non-false literals $\in \mathcal{C}_{\text{lits}}$, this is trivially satisfied in UPA-saturated states.

- if $l$ is defined by $(\neg l) \leftarrow lwr \leq Min(S) \leq upr$, let $DJ(l)$ be either $\{l_i\}$ for some $(l_i, w_i) \in S$ with $M(l_i) \neq \boldsymbol{f}$ and $w_i \leq lwr$, or $\{\neg l_i \mid (l_i, w_i) \in S, w_i \leq upr\}$.

## 6.4 Evaluation

We have implemented the above described algorithms as an extension of the SAT(ID) solver MiniSat(ID) (cf. Section 5.7).

This implementation required additional data structures to be added to MiniSat(ID), as described in Section 6.3.3. It also required a generalization of the justification data structures. In the extended MiniSat(ID), justifications are represented using an array that maps all defined atoms to their direct justification, represented by an array of literals. The transpose of a justification is represented using an array that maps literals to arrays of defined literals.

We did an experimental evaluation by comparing MiniSat(ID) to some ASP solvers. The settings of our experiments are the same as in Section 5.7.4: we used a timeout of 10 minutes; the platform was an Intel Core 2 Duo 3GHz with 2GB RAM running Kubuntu 7.10 Linux; the solver versions are clasp: version 1.1.2, SModels$_{cc}$: version 1.08, SModels: version 2.33. The IDP encodings of the problems used here are given in Appendix B.

In Figure 6.4 we evaluate the efficiency of solving *social golfer* problems of various sizes. These are scheduling problems with three parameters: number of weeks, number of groups, and number of people per group. We have aggregated our results over the last parameter: each row in the figure represents 4 actual instances, for 4 different numbers of people per group.

We observe a great variety in the difficulty of these problems; many instances can be solved in a matter of seconds, whereas many others yield timeout. MiniSat(ID) is the solver with the least number of timeouts.

In Figure 6.5 we evaluate the efficiency of solving magic series problems of various sizes (cf. Examples 6.1 and 6.4). In these problems, the ECNF representation whereby one set is reused in multiple aggregate expressions pays off dramatically. We give the most interesting comparison, whereby grounding times (of lparse 1.0.17 for the ASP solvers, and of GidL 1.5.2 for MiniSat(ID)) are included.

Other comparisons are difficult to perform, because different solvers offer different aggregate expressions, and especially for recursive aggregates, with different semantics. We observe that the performance of MiniSat(ID) on problems with recursive aggregates is lower than on problems with non-recursive aggregates, though: typical propagation speeds are around $5 \times 10^5$ for the recursive case, and around $1 \times 10^6$ for the non-recursive case. The likely explanation is that generalized justification data structures have a bad impact on low-level behaviour (such as caching) of the solver.

| | clasp | | SModels | | SModels$_{cc}$ | | MiniSat(ID) | |
|---|---|---|---|---|---|---|---|---|
| 3 weeks, 3 groups | **0** | 3.18 | **0** | 41.71 | **0** | 4.90 | **0** | 0.47 |
| 3 weeks, 4 groups | **2** | 0.02 | **2** | 0.05 | **2** | 0.09 | **1** | 82.51 |
| 3 weeks, 5 groups | **1** | 0.08 | **1** | 0.19 | **1** | 0.17 | **1** | 0.03 |
| 3 weeks, 6 groups | **0** | 0.13 | **0** | 0.29 | **0** | 0.33 | **0** | 0.06 |
| 4 weeks, 3 groups | **1** | 4.39 | **1** | 28.25 | **1** | 2.14 | **1** | 0.30 |
| 4 weeks, 4 groups | **1** | 1.41 | **1** | 49.25 | **1** | 2.58 | **0** | 102.02 |
| 4 weeks, 5 groups | **1** | 0.22 | **1** | 9.47 | **1** | 0.33 | **1** | 0.04 |
| 4 weeks, 6 groups | **1** | 0.16 | **1** | 6.69 | **1** | 0.60 | **1** | 0.13 |
| 5 weeks, 3 groups | **2** | 7.83 | **3** | 24.53 | **2** | 1.93 | **2** | 16.04 |
| 5 weeks, 4 groups | **1** | 4.26 | **1** | 95.31 | **1** | 6.62 | **1** | 0.77 |
| 5 weeks, 5 groups | **2** | 0.16 | **3** | 0.12 | **2** | 0.39 | **2** | 0.18 |
| 5 weeks, 6 groups | **1** | 0.27 | **1** | 0.61 | **1** | 0.65 | **1** | 0.09 |
| 6 weeks, 3 groups | **2** | 17.02 | **3** | 27.35 | **2** | 3.83 | **2** | 2.51 |
| 6 weeks, 4 groups | **2** | 2.84 | **2** | 170.00 | **2** | 21.45 | **2** | 1.26 |
| 6 weeks, 5 groups | **3** | 0.43 | **3** | 11.74 | **3** | 0.74 | **3** | 45.63 |
| 6 weeks, 6 groups | **2** | 0.25 | **3** | 0.24 | **2** | 0.78 | **2** | 0.10 |
| total | **22** | 42.65 | **26** | 465.80 | **22** | 47.53 | **20** | 252.13 |

Figure 6.4: Timing comparison of different solvers on *social golfer* problems. Per solver: number of timeouts (in bold), and average time (*sec*) of solved instances.

| | clasp | SModels | SModels$_{cc}$ | MiniSat(ID) |
|---|---|---|---|---|
| size 10 | 0.16 | 0.16 | 0.17 | 0.64 |
| size 20 | 2.02 | 2.04 | 2.08 | 0.55 |
| size 30 | 4.19 | 4.25 | 4.38 | 0.59 |
| size 40 | 121.15 | 121.26 | 123.45 | 0.77 |
| size 50 | 112.19 | 113.00 | 116.33 | 12.85 |
| size 60 | >600 | >600 | >600 | 12.53 |
| size 70 | >600 | >600 | >600 | 50.76 |
| size 80 | >600 | >600 | >600 | 27.30 |

Figure 6.5: Comparison of grounding + solving times (*sec*) on *magic series* problems.

## 6.5 Conclusions

We have extended syntax and semantics of ID-logic with aggregate expressions, both in inductive definitions and in FO sentences. The extended semantics covers aggregate expressions of types that frequently occur in practical problem domains, such as $Min$, $Max$, $Card$, $Sum$ and $Prod$.

We use the semantics developed by Pelov (2004); Pelov et al. (2007). This is the first work to implement propositional model generation algorithms for recursive aggregate expressions with this semantics. For other semantics of aggregate expressions in logic programs, we refer to (Van Gelder, 1992; Simons,

1999; Pelov et al., 2004; Mumick et al., 1990; Kemp and Stuckey, 1991; Niemelä et al., 1999; Dell'Armi et al., 2003; Marek et al., 2004).

We have implemented these algorithms by generalizing the justification semantics results of Chapter 4 and some of the algorithms developed in Chapter 5 to definitions containing aggregate expressions. In particular, we have generalized the two most relevant transition rules concerning inductive definitions: BwLoop and AddLF, to the transition rules BwAggrLoop and AddAggrLF, to be used in a stable strategy.

The transition rules presented in Section 6.3.3, and specifically those in Figure 6.2, perform the same propagations as those for logic programs with weight constraints by Simons et al. (2002) (in the *AtLeast* procedure) and disjunctive logic programs with aggregates by Faber et al. (2008) (in the *Deterministic Consequences* operator). In the latter case, also the data structures used are highly similar to ours. The main differences come from the difference in representation (ECNF definitions versus logic programs that may have several rules with the same head).

Liu and Truszczyński (2006) studied logic programs extended with "monotone and convex constraints": abstractions of propositional aggregate expressions. They allowed such constraints to appear in the heads of rules. Liu and Truszczyński also defined loop formulas for logic programs extended with constraints: in the case of monotone constraints in the bodies of rules, their loop formulas correspond to our generalized stable loop formulas in the same way as Lin and Zhao's (2004) loop formulas for normal logic programs correspond to our loop formulas for positive loops (cf. Section 4.4.2).

# Chapter 7

# A comparison of ID-logic and Stable logic programs

## 7.1 Introduction

Answer Set Programming (ASP) is a declarative problem solving paradigm (Niemelä, 1999; Marek and Truszczyński, 1999; Gelfond and Leone, 2002; Baral, 2003). In this paradigm, a problem is solved by computing models ("answer sets") of a specific theory in some logic $\mathcal{L}$. To this end, a human expert should model the theory such that its models have a one-to-one correspondence to the solutions of the problem at hand. The paradigm is associated, in most writings, with a language $\mathcal{L}$ based on the stable model semantics for normal logic programs (Gelfond and Lifschitz, 1988). Depending on the author, this language may include classical negation in addition to negation as failure (Gelfond and Lifschitz, 1991), disjunction in the head of rules (Przymusinski, 1991), weight constraints (Niemelä et al., 1999), choice rules (Simons, 1999), and nested expressions in both body and head of rules (Lifschitz et al., 1999). In this text, we focus on the fragment common to all of these languages: Stable logic programs, as defined in Section 2.3.

However, the paradigm of modelling theories such that their models correspond to problem solutions is applicable to other logics as well. Notably, it is applicable also to ID-logic (as we have illustrated in Chapter 3), or indeed to any other logic whose semantics is defined in a model theoretic way. This paradigm can also be implemented in different ways: the computational task behind it may be Herbrand model generation, or it may be model expansion.

This chapter compares ID-logic to Stable logic programs. There are considerable conceptual differences between these two logics: ID-logic is based on classical logic, whereas Stable logic programs are based on logic programming and Herbrand interpretations. But there are important similarities as well: definitions in ID-logic use the well-founded semantics of logic programming (Van Gelder et al., 1991), which is closely related to the stable model semantics (see,

for instance, Przymusinski, 1990).

In the past decade, the research domain around ASP has thrived. Then, given the similarities of ID-logic and Stable logic programs, the comparison in this chapter is easily motivated.

We begin our comparison by providing transformations from ID-logic theories to Stable logic programs and vice versa, in Section 7.2. Particularly the former transformation yields interesting insights into the different components of both types of theories, and their relation. These results are also useful for direct application: one can use them to implement an ID-logic model generator based on a stable model generator, and vice versa. In Section 7.3 we then investigate the applicability of the ID-logic modelling principles from Section 3.1 in Stable logic programs. Finally, we compare model expansion and Herbrand model generation in Section 7.4. We conclude in Section 7.5.

## 7.2 Transformations

In this section we present equivalence preserving transformations from ID-logic to Stable logic programs and vice versa. These transformations may introduce new symbols to the original vocabulary $\Sigma$: thus the type of equivalence we obtain is *up to* $\Sigma$, denoted $\equiv_\Sigma$. Formally, $T_1 \equiv_\Sigma T_2$, for theories $T_1$ and $T_2$ with vocabularies that contain $\Sigma$, means that for every $vocab(T_1)$-model $M_1$ of $T_1$, $T_2$ has a $vocab(T_2)$-model $M_2$ such that $M_1|_\Sigma = M_2|_\Sigma$, and vice versa.

To simplify the comparison, we restrict ID-logic to Herbrand models, i.e., we assume that the *Unique Names Axioms* (UNA) (Reiter, 1980) and the *Domain Closure Assumption* (DCA) (Reiter, 1982) are implicit. Also, for Stable logic programs we assume that the *Closed World Assumption* (CWA) (Reiter, 1977) is implicit: as such, models of a stable logic program can be compared to other Herbrand models.[1]

### 7.2.1 ID-logic to ASP

We now define an equivalence preserving transformation from ID-logic to Stable logic programs, for a subset of ID-logic. The work in this section is based on (Mariën et al., 2004).

We present a succession of different transformations, most of which reflect the syntactic restrictions of normal logic programs. We begin with a transformation from similarly restricted ID-logic theories to Stable logic programs, and continue with a number of transformations that transform an ID-logic theory into another, equivalent one that has extra syntactic restrictions. This first transformation is based on a transformation from *Abductive logic programs* to Stable logic programs by Satoh and Iwayama (1991).

[1] Gelfond and Lifschitz (1991) note that the meaning of stable models of a logic program depends on the presence of the CWA: if it is not present, models are "belief sets", which cannot be compared to Herbrand models.

### Abductive logic programs

Abductive logic programs are normal logic programs that possibly contain integrity constraints.[2] An *abductive framework* is a tuple of an abductive logic program and a set of *abducibles*, which are atoms that are not defined in the program.

**Definition 7.1** (Generalized stable semantics of abductive frameworks)**.** Let $P$ be a normal logic program with integrity constraints, and $A$ a set of atoms, none of which occurs in the head of any rule in $P$. Then $\langle P, A\rangle$ is an *abductive framework*. Let $E \subseteq A$. A *generalized stable model* $M(E)$ of $\langle P, A\rangle$ is a stable model of $P \cup \{e \texttt{ :- } \top \mid e \in E\}$.

One uses abductive frameworks in *abductive reasoning*: given a proposition $O$ which is observed to be true, find an explanation $E$, a subset of the abducibles, such that the truth of all atoms in $E$ suffices to infer the truth of $O$. Formally, $O$ is said to have an abductive explanation with hypothesis $E$, iff $\langle P, A\rangle$ has a generalized stable model $M(E)$ with $O \in M(E)$. Observe that the explanation atoms $E$ are true in any generalized stable model (since they are added as facts), whereas the other abducibles $A \setminus E$ are false in any generalized stable model (since there are no rules to make them true). Therefore, finding an abductive explanation for some observation $O$ corresponds to finding an appropriate interpretation of the abducibles.

Satoh and Iwayama (1991) showed that an abductive framework $\langle P, A\rangle$ can be transformed into a Stable logic program. For any $a \in A$, a new atom $a'$ is introduced, and two new rules are added to $P$: $a \texttt{ :- } \texttt{not}\, a'$, $a' \texttt{ :- } \texttt{not}\, a$. These two rules have the effect that a truth value for $a$ can be chosen to suit the rest of the theory. If $P'$ is the program resulting from Satoh and Iwayama's transformation on $\langle P, A\rangle$, then, for any $E \subseteq A$, $M(E)$ is a generalized stable model of $\langle P, A\rangle$ iff there is a stable model $M'$ of $P'$ with $M'|_{vocab(P)} = M(E)$.

Though Satoh and Iwayama proved this result for the propositional case, it is immediately generalizable to first-order level.

We now use this result to propose a transformation of a small subclass of ID-logic theories to Stable logic programs. This subclass is defined by a set of constraints, or properties, the ID-logic theories should satisfy; in subsequent sections we will lift most of these properties.

Let $T$ be an ID-logic theory with the following properties:

Property 1. it contains at most one definition, $\Delta$;

Property 2. all definitional rules have a body in the form $(L_1 \wedge \ldots \wedge L_N)$, where the $L_i$ are literals, and $N$ a (possibly infinite) natural number;

Property 3. all FO sentences are in the clausal form $\forall(A_1 \wedge \cdots \wedge A_n \supset B_1 \vee \cdots \vee B_m)$, where $A_i$, $B_j$ are atomic formulas;

[2] Some authors allow the integrity constraints to be arbitrary FO formulas; here we consider them to be normal logic programming constraints.

Property 4. $\Delta$ is total in the rest of the theory.

Then $T$ can be seen as an encoding of an abductive framework: $\Delta$ is the encoding of normal logic program rules, the FO sentences the encoding of integrity constraints, and *Open*($\Delta$) corresponds to the abducibles. We formalize this encoding and translate it to an ASP theory as follows:

Let $ASP(T)$ be the normal logic program obtained from $T$ by:

1. applying the syntactical conversion of the sentences $\forall(A_1 \wedge \cdots \wedge A_n \supset B_1 \vee \cdots \vee B_m)$ to integrity constraints $\texttt{:-}\ A_1, \ldots, A_n, \texttt{not}\, B_1, \ldots, \texttt{not}\, B_m$;

2. applying the syntactical conversion of "$\leftarrow$" to "`:-`", "$\neg$" to "`not`", and "$\wedge$" to ",";

3. applying Satoh and Iwayama's transformation on a first order level: for each predicate $A/n \in Open(\Delta)$, introduce a new predicate $A'/n$, and add the rules $A(\overline{x})\ \texttt{:-}\ \texttt{not}\, A'(\overline{x})$, $A'(\overline{x})\ \texttt{:-}\ \texttt{not}\, A(\overline{x})$. We will refer to rules of this type as *double negation rules*.

**Proposition 7.1.** *Let $T$ be an ID-logic theory that satisfies the above properties. Then $ASP(T) \equiv_{vocab(T)} T$.*

*Proof.* $\Delta$ is total in the rest of the theory, i.e. it has a two-valued well-founded model extending any interpretation of *Open*($\Delta$) that satisfies $T - \Delta$. If a definition has a two-valued well-founded model, that model is also the definition's unique stable model (Van Gelder et al., 1991). Therefore the result from (Satoh and Iwayama, 1991) is applicable. Here our earlier restriction of ID-logic to Herbrand models is used (see beginning of Section 7.2). $\square$

*Remark* 7.1. ID-logic theories often contain *domain declarations* for the predicates that are non-defined: sentences of the form $\forall \overline{x}\ A(\overline{x}) \supset C_A(\overline{x})$, where $A/n \in Open(\Delta)$ is the non-defined predicate, and $C_A(\overline{x})$ a conjunction of literals called the *domain* of $A/n$. For such predicates, we can replace Step 3 of the transformation by a step that adds the double negation rules $A(\overline{x})\ \texttt{:-}\ C_A(\overline{x}), \texttt{not}\, P'(\overline{x}), \quad P'(\overline{x})\ \texttt{:-}\ C_A(\overline{x}), \texttt{not}\, P(\overline{x})$ instead, and removes the domain declaration from $ASP(T)$. Though theoretically equivalent to the above transformation, this transformation produces smaller groundings and therefore leads to more efficient model generation.

**Example 7.1.** Recall from Section 3.3.2 the ID-logic encoding of the Hamiltonian circuit problem. Here we replace the constant symbol *Start* by a unary predicate symbol *Start*/1, because of our restriction of ID-logic to Herbrand models. We also unfold the $\exists!$ notation, and because we use untyped ID-logic here, we explicitly add a predicate symbol $Vertex/1$ to represent the domain.

Let $T_{Ham}$ be

$$\begin{aligned}
&\forall x, y\ Ham(x,y) \supset Edge(x,y),\\
&\forall x, y, z\ Ham(x,y) \land Ham(x,z) \supset y = z,\\
&\forall x, y, z\ Ham(y,x) \land Ham(z,x) \supset y = z,\\
&\left\{\begin{array}{ll} \forall x\ (Reached(x) \leftarrow & Ham(y,x)) \land Start(y),\\ \forall x\ (Reached(x) \leftarrow & \exists y\ Ham(y,x) \land Reached(y)) \end{array}\right\},\\
&\forall x\ Vertex(x) \supset Reached(x).
\end{aligned}$$

We apply the transformation: $ASP(T_{Ham})$ is the Stable logic program

$$Ham(x,y) \texttt{ :- } Edge(x,y), \texttt{not}\, NegHam(x,y) \tag{7.1}$$
$$NegHam(x,y) \texttt{ :- } Edge(x,y), \texttt{not}\, Ham(x,y) \tag{7.2}$$
$$\texttt{:- } Ham(x,y), Ham(x,z), \texttt{not}\, (y = z) \tag{7.3}$$
$$\texttt{:- } Ham(y,x), Ham(y,x), \texttt{not}\, (y = z) \tag{7.4}$$
$$Reached(x) \texttt{ :- } Ham(y,x), Start(y) \tag{7.5}$$
$$Reached(x) \texttt{ :- } Ham(y,x), Reached(y) \tag{7.6}$$
$$\texttt{:- } Vertex(x), \texttt{not}\, Reached(x) \tag{7.7}$$

The double negation rules (7.1)–(7.2) have been formed by taking into account the domain declaration $\forall x, y\ Ham(x,y) \supset Edge(x,y)$ (cf. Remark 7.1).

Interestingly, $ASP(T_{Ham})$ corresponds exactly[3] to the stable logic program presented by Marek and Truszczyński (1999).

In the following sections we provide ID-logic transformations to generalize this result also to theories that do not satisfy Properties 1–3. We cannot easily lift Property 4 though. Example 7.2 shows why this property is required.

**Example 7.2.** Let $T$ be the ID-logic theory consisting only of this definition: $\{P \leftarrow \neg Q, \quad Q \leftarrow \neg P\}$. Since this definition is not total, $T$ is unsatisfiable. However, $ASP(T) = \{P \texttt{ :- not}\, Q, \quad Q \texttt{ :- not}\, P\}$ has two models: $\{P \mapsto \boldsymbol{t}, Q \mapsto \boldsymbol{f}\}$ and $\{P \mapsto \boldsymbol{f}, Q \mapsto \boldsymbol{t}\}$.

*Remark* 7.2. An *abductive reasoning inference system* for a subset of ID-logic was developed by Van Nuffelen (2004). The system offers an implementation of the SLDNFA proof procedure by Denecker and De Schreye (1998).

### Multiple definitions

In this section, we lift Property 1, which states that the theory should contain at most one definition. Let $T$ be a theory satisfying Properties 2–4, but not Property 1. Property 4 should now be satisfied for each separate definition of $T$.

[3]With renaming of symbols. Also, Marek and Truszczyński used explicit constraints of the form $f \texttt{ :- } \varphi, \texttt{not}\, f$, whereas we use $\texttt{:- } \varphi$.

An ID-logic theory containing multiple definitions can be transformed to one that contains exactly one definition. The straightforward approach consists of merging all definitions into one; however this approach may be erroneous in two cases:

- when separate definitions define the same predicate, as in Example 7.3;
- when separate definitions depend on each other, as in Example 7.4.

**Example 7.3.** Consider again the ID-logic theory $T_{3.1}$ from Example 3.1:

$$\left[\begin{array}{l} \{\ \forall x\ (Square(x) \leftarrow Equilateral(x) \land Rectangle(x))\ \}, \\ \{\ \forall x\ (Square(x) \leftarrow Orthogonal(x) \land Rhombus(x))\ \} \end{array}\right].$$

Recall also that because both definitions of $T_{3.1}$ must be satisfied independently, $T_{3.1}$ logically entails the formula $\forall x\ (Equilateral(x) \land Rectangle(x) \equiv Orthogonal(x) \land Rhombus(x))$. If the definitions are merged, this formula is no longer entailed.

To clarify the second case, we need a notion of definitional dependency.

**Definition 7.2** (Definitional dependency graph)**.** Let $T$ be an ID-logic theory, and $\Delta_1, \ldots, \Delta_n$ the definitions in $T$. Then *$T$'s definitional dependency graph* is the graph with nodes $\{\Delta_1, \ldots, \Delta_n\}$ and edges $\{(\Delta_i, \Delta_j) \mid Open(\Delta_i) \cap Def(\Delta_j) \neq \emptyset\}$.

**Example 7.4.** Let $\Delta_1 = \{P \leftarrow Q \lor R,\quad Q \leftarrow P\}$, $\Delta_2 = \{R \leftarrow Q\}$. Then the definitional dependency graph of $T = [\Delta_1, \Delta_2]$ is $\Delta_1 \rightleftarrows \Delta_2$, because $Open(\Delta_1) \cap Def(\Delta_2) = \{R\}$, $Open(\Delta_2) \cap Def(\Delta_1) = \{Q\}$.

Whereas $T$ has 2 models, $M_1 = \{P, Q, R\}$ and $M_2 = \{\neg P, \neg Q, \neg R\}$, the definition resulting from merging $\Delta_1$ and $\Delta_2$, i.e. $\Delta_{1\cup 2} = \{P \leftarrow Q \lor R,\quad Q \leftarrow P,\quad R \leftarrow Q\}$, has only $M_2$ as a model. This difference stems from the fact that $T$'s definitional dependency graph is cyclic.

The solution to both of these problems is the same: renaming defined predicates. Let $\Delta$ be a definition of $T$, and $P/n \in Def(\Delta)$, such that either $T$'s definitional dependency graph has a cycle that passes though $\Delta$, or $T$ contains a definition $\Delta'$ with also $P/n \in Def(\Delta')$. Then introduce a new predicate $P'/n$, replace all occurrences of $P/n$ in $\Delta$ by $P'/n$, and add the sentences $\forall \overline{x}\ P(\overline{x}) \supset P'(\overline{x})$, $\forall \overline{x}\ P'(\overline{x}) \supset P(\overline{x})$ to $T$.[4]

Apply this renaming technique until the resulting definitional dependency graph is acyclic, and each predicate is defined in at most one definition. Then replace all definitions $\Delta_1, \ldots, \Delta_n$ by one definition: $\Delta = \{r \in \Delta_i \mid 1 \leq i \leq n\}$. Let $Lift\text{-}1(T)$ be the resulting ID-logic theory.

**Example 7.5.** Example 7.3 continued. $Lift\text{-}1(T) =$

$$\left[\begin{array}{l} \left\{\begin{array}{l} \forall x\ (Square(x) \leftarrow Equilateral(x) \land Rectangle(x)), \\ \forall x\ (Square'(x) \leftarrow Orthogonal(x) \land Rhombus(x)) \end{array}\right\}, \\ \forall x\ Square(x) \supset Square'(x), \\ \forall x\ Square'(x) \supset Square(x) \end{array}\right].$$

[4] This is equivalent to adding $\forall \overline{x}\ P(\overline{x}) \equiv P'(\overline{x})$, but also complies with Property 3.

**Proposition 7.2.** *Let $T$ be an ID-logic theory such that each definition $\Delta \in T$ is total with respect to $T - \Delta$. Then $Lift$-1$(T) \equiv_{vocab(T)} T$.*

*Proof.* Let $T'$ be the intermediate theory of the transformation: the result of introducing the new predicates and the equivalences, before the merging of the definitions. We first prove that $T' \equiv_{vocab(T)} T$, and then prove that $Lift\text{-}1(T) \equiv T'$.

Let $\Delta$ be a definition of $T$, and $P/n \in Def(\Delta)$, for which a new predicate $P'/n$ has been introduced. Let $\Delta'$ be the result of replacing all occurrences of $P/n$ by $P'/n$. $T'$ does not contain $\Delta$, but instead $\Delta'$, as well as the sentences $\forall \overline{x}\ P(\overline{x}) \supset P'(\overline{x})$, $\forall \overline{x}\ P'(\overline{x}) \supset P(\overline{x})$. Observe that $Open(\Delta) = Open(\Delta')$. Consider an $vocab(T)$-model $I$ of $T$. Let $I_{wfm} = \text{wfm}_\Delta(I|_{Open(\Delta)})$ and $I'_{wfm} = \text{wfm}_{\Delta'}(I|_{Open(\Delta')})$; since $I \models T$, we have that $I$ is an extension of $I_{wfm}$. By definition of $\Delta'$ we have that $\{\overline{d} \in \text{dom}^n(I) \mid \overline{d} \in P^{I_{wfm}}\} = \{\overline{d} \in \text{dom}^n(I) \mid \overline{d} \in P'^{I'_{wfm}}\}$. Hence, both the sentences $\forall \overline{x}\ P(\overline{x}) \supset P'(\overline{x})$ and $\forall \overline{x}\ P'(\overline{x}) \supset P(\overline{x})$, and $\Delta'$, are satisfied in the interpretation $I' = I \cup I'_{wfm}$. Thus $I' \models T'$. Conversely, consider a $vocab(T')$-model $I'$ of $T'$. Then because $I' \models \Delta'$ and $I' \models \forall \overline{x}\ P(\overline{x}) \equiv P'(\overline{x})$, and again by definition of $\Delta'$, we find that $I'|_{vocab(\Delta)} = \text{wfm}_\Delta(I'|_{Open(\Delta)})$. Hence $I'|_{vocab(T)} \models T$.

The second part of the proof, that $Lift\text{-}1(T) \equiv T'$, is by application of the modularity theorem of (Denecker and Ternovska, 2008, Theorem 5.20). There, the notion of a *reduction partition* $\{\Delta_1, \ldots, \Delta_n\}$ of a definition $\Delta$ is introduced; it would take us too far to redefine it here, but suffice it to say that after the renamings in $Lift$-1, the set $\{\Delta_1, \ldots, \Delta_n\}$ is a reduction partition of $\Delta = \{r \in \Delta_i \mid 1 \leq i \leq n\}$. The modularity theorem states for a reduction partition $\{\Delta_1, \ldots, \Delta_n\}$ of a definition $\Delta$, $I \models \Delta$ iff $I \models \Delta_1 \cup \cdots \cup \Delta_n$. □

### Arbitrary rule bodies

In this section we lift Property 2, which requires rules bodies to be conjunctions of literals. This can be done by the following transformation on rule bodies:

1. push negation inside to the level of literals;

2. apply the following replacements until the resulting definition is in normal logic programming form:

   (a) replace $\forall\ (H \leftarrow \varphi_1 \vee \varphi_2)$ by $\forall\ (H \leftarrow \varphi_1)$, $\quad \forall\ (H \leftarrow \varphi_2)$;

   (b) replace $\forall \overline{x}\ (H \leftarrow \varphi_1 \wedge \exists \overline{y}\ \varphi_2)$ by $\forall \overline{x} \forall \overline{y}\ (H \leftarrow \varphi_1 \wedge \varphi_2)$;[5]

   (c) replace $\forall \overline{x}\ (H \leftarrow \forall \overline{y}\ \varphi[\overline{y}])$ by $\forall \overline{x}\ (H \leftarrow \varphi[\overline{y}/\overline{D_1}] \wedge \cdots \wedge \varphi[\overline{y}/\overline{D_n}])$, where $\overline{D_1}, \ldots, \overline{D_n}$ are all tuples of size $|\overline{y}|$ of the domain;[6]

   (d) if none of 2a–2c is applicable, then for any nested subformula $\varphi$, introduce a new predicate $P_\varphi$, replace $\varphi$ by $P_\varphi$, and introduce a new rule $P_\varphi \leftarrow \varphi$.

[5] We assume, without loss of generality, that $\overline{y}$ does not contain free variables of $H$ or $\varphi_1$.

[6] Recall that we have assumed the domain to be the finite Herbrand Universe.

Rules 2a and 2b in this transformation also occur in the Lloyd and Topor (1984) transformation, but the rest of that transformation is not equivalence preserving for ID-logic definitions. Also, it may yield an exponentially bigger theory (because of the distribution of $\vee$ over different rules); rule 2d here prevents a similar explosion. We call the process of replacing a subformula $\varphi$ by a newly defined predicate *Tseitin predicate introduction*, and the new symbols $P_\varphi$ *Tseitin predicates*, after Tseitin's (1968) transformation from arbitrary propositional logic theories to CNF theories. Van Gelder (1993) introduced the same transformation as Tseitin predicate introduction. Note that correct application of the Tseitin predicate introduction rule (Rule 2d) requires that negation is pushed to the literal level first.

Let $Lift\text{-}2(\Delta)$ be the result of applying the above transformation on $\Delta$.

**Proposition 7.3.** *For any ID-logic definition $\Delta$, $Lift\text{-}2(\Delta) \equiv_{vocab(\Delta)} \Delta$.*

*Proof.* Each of the rules 1–2c preserves the behaviour of rules of a well-founded sequence (cf. Definition 2.1) on $\Delta$. The Tseitin predicate introduction rule, Rule 2d, satisfies the conditions of our predicate introduction theorem in (Wittocx et al., 2006, Theorem 2). Therefore each of the transformation rules preserves the well-founded model. □

**Example 7.6.** Consider the following definition of the transitive closure of a relation $R$:

$$\Delta_{7.6} = \left\{ \ \forall x, y \ \big(TC(x,y) \leftarrow R(x,y) \vee (\exists z \ TC(x,z) \wedge R(z,y))\big) \ \right\}.$$

Then $Lift\text{-}2(\Delta_{7.6}) =$

$$\left\{ \begin{array}{rl} \forall x, y & \big(TC(x,y) \leftarrow R(x,y)\big), \\ \forall x, y, z & \big(TC(x,y) \leftarrow TC(x,z) \wedge R(z,y)\big) \end{array} \right\}.$$

Rule 2c replaces a universal quantifier by a conjunction over all tuples of domain elements, i.e., it performs a local grounding. We discuss some alternatives for this rule that avoid such a replacement.

- The Lloyd and Topor transformation replaces $\forall \overline{x} \ (H \leftarrow \forall \overline{y} \ \varphi)$ by $\forall \overline{x} \ (H \leftarrow \neg P_{\neg\varphi})$, $\forall \overline{x} \forall \overline{y} \ (P_{\neg\varphi} \leftarrow \neg\varphi)$, where $P_{\neg\varphi}$ is a new predicate. This is the simplest of transformations, however, it is only correct for ID-logic definitions if there is no recursion over $H$ through $\varphi$. Example 7.7 provides an example where this transformation is not correct.

- Wittocx et al. (2006); Vennekens et al. (2007a) introduced a very general notion of predicate introduction. Based on this notion, we proposed a method of $\forall$ elimination based on a "domain iterator"; this proposal assumes a totally ordered domain to be given.

**Example 7.7.** This is a simplified version of a theory from (Balduccini and Gelfond, 2003); the same example is used also in (Wittocx et al., 2006). In this theory, *causal rules* $r$ are considered; such rules have a certain property $p$ as

effect (represented by $Head(r) = p$), and have other properties $q$ as precondition (represented by $Prec(r, q)$). A causal theory of which properties hold under which circumstances then looks like this:

$$\Delta_1 = \left\{ \; Holds(p) \leftarrow \exists r \; Head(r) = p \land (\forall q \; Prec(r, q) \supset Holds(q)) \; \right\},$$

stating that $p$ holds if it is the effect of some causal rule $r$, the preconditions of which *all* hold. Applying the Lloyd and Topor transformation on $\Delta_1$ yields

$$\Delta_2 = \left\{ \begin{array}{rl} Holds(p) & \leftarrow \exists r \; Head(r) = p \land \neg P_{Prec \supset Holds}(r), \\ P_{Prec \supset Holds}(r) & \leftarrow Prec(r, q) \land \neg Holds(q) \end{array} \right\}.$$

$\Delta_1$ and $\Delta_2$ are not equivalent when there are cyclic causal rules. E.g. in an interpretation $I$ with $Head^I = \{R_1 \mapsto P, R_2 \mapsto Q\}$, $Prec^I = \{(R_1, Q), (R_2, P)\}$, $\Delta_1$ has a model $M$ with $Holds^M = \emptyset$, i.e., neither of the properties $P$ or $Q$ are caused. $\Delta_2$ is non-total, and has no model extending $I$.

**Arbitrary FO sentences**

To lift Property 3, we have to transform arbitrary FO sentences to clausal form, $\forall (A_1 \land \cdots \land A_n \supset B_1 \lor \cdots \lor B_m)$. The technique of Tseitin predicate introduction presented in previous paragraph suffices to this end. In this case, each new Tseitin predicate $P_\varphi$ is defined by a definition with one rule. Note that such a new definition is automatically total, because it is not recursive. Furthermore, there cannot be recursion over a universal quantifier in the body, hence we can apply the Lloyd and Topor transformation in such cases: replace a subformula $\forall \overline{x} \; \varphi$ with free variables $\overline{y}$ by $\neg P_{\neg\varphi}$, and add $\left\{ \; \forall \overline{x} \forall \overline{y} \; (P_{\neg\varphi} \leftarrow \neg\varphi) \; \right\}$. Also note that clausal form includes formulas with $n = 0$, i.e., implications whose antecedent is $\top$.

Let $Lift\text{-}3(T)$ be the theory resulting from this transformation on a theory $T$.

**Proposition 7.4.** *$Lift\text{-}3(T) \equiv_{vocabT} T$.*

**Example 7.8.** Recall from Section 3.1.3 the two ID-logic (FO) encodings of the statement "there is a final timepoint after which no moves are made"; one without and one with a given constant $F$ to represent the final timepoint:

$$\exists t \; \forall t' \; \big(t' \geq t \supset \neg(\exists m \; Move(m, t'))\big), \tag{7.8}$$

$$\forall t \; \big(t \geq F \supset \neg(\exists m \; Move(m, t))\big). \tag{7.9}$$

We slightly rephrase (7.8):

$$\exists t \; \neg\exists t' \; \big(t' \geq t \land (\exists m \; Move(m, t'))\big), \tag{7.10}$$

and apply the transformation on sentence (7.10). First, we apply Tseitin predicate introduction on the subformula $\exists t' \; \big(t' \geq t \land (\exists m \; Move(m, t'))\big)$, introducing the Tseitin predicate $SomeMoveAfter/1$. The resulting formula,

$\exists t\ \neg SomeMoveAfter(t)$, is also not in clausal form, so we replace it by the Tseitin predicate $Final/0$. The resulting ID-logic theory is

$$\begin{array}{l} Final \\ \{\ \forall t\ (Final \leftarrow \neg SomeMoveAfter(t))\ \} \\ \{\ \forall t, t'\ (SomeMoveAfter(t) \leftarrow t' \geq t \wedge Move(m, t'))\ \}. \end{array} \tag{7.11}$$

If we further apply the definition-merging transformation, and the transformation to Stable logic programs, we obtain the logic program[7]

$$\begin{array}{rl} & \texttt{:-}\ \texttt{not}\ Final \\ Final & \texttt{:-}\ \texttt{not}\ SomeMoveAfter(t) \\ SomeMoveAfter(t) & \texttt{:-}\ t' \geq t, Move(m, t'). \end{array} \tag{7.12}$$

Applying the transformation on formula (7.9) yields the following Stable logic program:[7]

$$\begin{array}{rl} & \texttt{:-}\ t \geq F, AnyMove(t) \\ AnyMove(t) & \texttt{:-}\ Move(m, t). \end{array} \tag{7.13}$$

Notice how versions (7.12) and (7.13) look completely different, even though expressions (7.8) and (7.9) in ID-logic have a shared structure.

### Overview

The overall transformation first ($Lift$-3) transforms a theory with arbitrary FO assertions into a theory where all FO sentences are in clausal form. This theory may contain new definitions. Next ($Lift$-2), all definitions in the theory are transformed to definitions with only conjunctions of literals in rule bodies. Then ($Lift$-1) all definitions are merged, whereby possibly some additional clausal sentences are added. Finally ($ASP$) the result is transformed into a Stable logic program, by adding the appropriate double negation rules.

**Theorem 7.1.** *Let $T$ be an ID-logic theory. Let each definition $\Delta$ in $T$ be total in $T - \Delta$. Then $ASP(Lift\text{-}1(Lift\text{-}2(Lift\text{-}3(T)))) \equiv_{vocab(T)} T$.*

*Proof.* By propositions 7.1–7.4. □

*Remark* 7.3. If $T$ contains no rules with recursion over a universal quantifier in the body, then the size of $ASP(Lift\text{-}1(Lift\text{-}2(Lift\text{-}3(T))))$ is linear in the size of $T$.

[7] For simplicity, we leave out the double negation rules for $Move/2$ here.

### 7.2.2 Stable logic programs to ID-logic

The reverse transformation, from Stable logic programs to ID-logic theories, is much simpler; it is due to East and Truszczyński (2006). It is equivalence preserving for all Stable logic programs.

Let $T$ be an arbitrary Stable logic program. Then the ID-logic theory $ID\text{-}logic(T)$ can be obtained from $T$ by:

1. applying the syntactical conversion of "`:-`" to "$\leftarrow$" and "," to "$\wedge$";
2. applying the syntactical conversion of integrity constraints $\texttt{:-}\ A_1, \ldots, A_n, \texttt{not}\ B_1, \ldots, \texttt{not}\ B_m$ to FO sentences $\forall(A_1 \wedge \cdots \wedge A_n \supset B_1 \vee \cdots \vee B_m)$;[8]
3. for every predicate symbol $P/n$, if there is an occurrence of $\texttt{not}\ P$, replacing all those occurrences by a newly introduced symbol $nP/n$, and adding a sentence $\forall \overline{x}\ nP(\overline{x}) \equiv \neg P(\overline{x})$;
4. for every predicate symbol $P/n$ that does not occur in the head of any rule of $T$, add a sentence $\forall \overline{x}\ \neg P(\overline{x})$;
5. grouping all rules together in one ID-logic definition, which is added as a sentence.

**Proposition 7.5** (East and Truszczyński 2006)**.** $T \equiv_{vocab(T)} ID\text{-}logic(T)$.

*Remark* 7.4. This transformation always produces negation-free definitions; negation only occurs in the FO part of the resulting theory.

## 7.3 Applicability of ID-logic modelling principles in Stable logic programs

Recall the methodological principles of modelling in ID-logic from Section 3.1:
- represent definitional knowledge by definitions;
- represent definitions as separate modules;
- represent non-definitional knowledge by FO sentences;
- represent each "case" of an informal definition by one rule in the formal definition;
- use function symbols whenever the represented concept is clearly a function.

In this section we investigate whether, or to what extent, these principles are applicable also to Stable logic programs.

[8] We defined the semantics of integrity constraints $\texttt{:-}\ C$ by a translation to a normal rule $f\ \texttt{:-}\ C, \texttt{not}\ f$, where $f$ represents *false*. Naturally, the transformation to ID-logic theories presented here works also on such translations.

### Represent definitional knowledge by definitions

According to this principle, definitional knowledge should be represented by a definition, i.e., a set of rules.

Observe that mathematically well-defined definitions are total, and that for total definitions the stable model semantics and the well-founded semantics coincide. Hence it is possible to apply the methodological principle of representing definitional knowledge by a set of rules in Stable logic programs.

In Stable logic programs as occurring in the literature, we observe that this methodology is applied in practice. Examples include the Hamiltonian circuit encoding by Marek and Truszczyński (1999), the planning system for decision support for the space shuttle by Nogueira et al. (2001), the block's world encoding by Lifschitz (2002), and the diagnostic reasoning framework by Balduccini and Gelfond (2003), to name just a few.

Recall also that it is part of the ID-logic modelling methodology to use the definition construct *exclusively* for the purpose of representing definitional knowledge. We investigate when, or whether, the rules occurring in Stable logic programs can be considered as definitional rules.

We consider ID-logic theories that are modelled according to our methodological principles from Section 3.1. The Stable logic programs obtained by applying the transformation of Section 7.2.1 on them yield (constraints and) three types of rules: *a*) rules that originate from a definitional rule in ID-logic, *b*) auxiliary rules, used to define subformulas, and *c*) double negation rules.

Rules of type *a* or *b* can clearly be considered definitional rules. Rules of type *c* that express the openness of predicates are in practice often represented using the syntactic sugar of *choice rules* (Simons, 1999).

This naturally raises the question of whether in general, (non-choice) rules in Stable logic programs are always of definitional nature. The answer is no. For instance, Niemelä (1999) describes "a knowledge representation technique based on *rules with exceptions*". Rules with exceptions are neither double negation rules, nor are they of definitional nature. Consider the following example.

**Example 7.9.** Example 3.2 continued. The following Stable logic program encoding of the graph colouring problem is taken from (Niemelä, 1999). The predicates $Vtx/1$ and $Clr/1$ are so-called *domain predicates* and serve the same purpose as types in the ID-logic encoding.

$$Colour(v,c) \ \texttt{:-}\ Vtx(v), Clr(c), \texttt{not}\ OtherColour(v,c) \tag{7.14}$$

$$OtherColour(v,c) \ \texttt{:-}\ Vtx(v), Clr(c), Clr(d), c \neq d, Colour(v,d) \tag{7.15}$$

$$\texttt{:-}\ Edge(v_1,v_2), Clr(c), Colour(v_1,c), Colour(v_2,c) \tag{7.16}$$

Expression (7.16) expresses the basic problem statement (cf. (3.12) in the ID-logic modelling). But rules (7.14) and (7.15) illustrate an interesting practice: they simultaneously declare the predicate symbol $Colour$ to represent a function, and open the search space—i.e., declare $Colour$ to be open. These are *rules with exceptions*: rule (7.14) expresses that vertex $v$ is assigned colour $c$ unless there is some exception, which is specified by rule (7.15).

### Represent definitions as separate modules

According to this principle, one should provide *separate* definitions whenever possible. Stable logic programs offers no syntactical means of doing so, hence this principle is not applicable to it.

However, the desire to obtain in Stable logic programs a modularity similar to that of ID-logic is evident from the work of, amongst others, Oikarinen and Janhunen (2006); Oikarinen (2006). A *logic program module* is defined by Oikarinen (2006) as a tuple $(\Delta, I, O)$, where $\Delta$ is a set of logic program rules, $I$ and $O$ are disjoint sets of predicate symbols, and $Def(\Delta) \cap I = \emptyset$: $I$ and $O$ constitute the *input* and *output* of the module. It is then defined what are stable models of such modules, and how they can be used in Stable logic programs. An example module from (Oikarinen, 2006) is $(\{A \text{ :- } B\}, \{B\}, \{A\})$, which has $\emptyset$ and $\{A, B\}$ as stable models.

It is clear that such modules behave like ID-logic definitions, albeit with stable models instead of well-founded models. When the rules in the module form a total definition, the semantics of the module is the same as the semantics of the corresponding ID-logic definition.

### Represent non-definitional knowledge by FO sentences

According to this principle, non-definitional knowledge should be represented by FO sentences, and such that their declarative reading is as close as possible to the natural language reading of the knowledge being expressed. FO sentences cannot be expressed in Stable logic programs, hence this principle is not directly applicable.

However, *constraints* can be expressed in Stable logic programs; as our transformation in Section 7.2.1 shows, these correspond to FO sentences in clausal form.

As Example 7.8 illustrates, clausal form is not sufficient to express certain statements (e.g. existentially quantified statements) in a declarative way.

### Represent each "case" of an informal definition by one rule in the formal definition

According to this principle, every rule in a definition should correspond to one "if" case in the natural language statement of the definition. Since this sometimes requires more general rule bodies than conjunctions of literals, the principle cannot be applied in Stable logic programs.

Note that the "auxiliary rules" of type $b$ above violate the principle.

### Use function symbols whenever the represented concept is clearly a function

According to this principle, one should represent a concept that clearly is a function by a function symbol. Since Stable logic programs have Herbrand models,

function symbols cannot be interpreted freely in Stable logic programs. Furthermore, theories that contain non-constant function symbols have an infinite Herbrand universe: this severely restricts the applicability of function symbols, and hence of this methodological principle.

## 7.4 A comparison of model expansion and Herbrand model generation

In this section we compare two problem solving tasks that fit in the ASP paradigm: (parameterized) model expansion and Herbrand model generation. Both of these computational tasks are applied to make model generation in the context of a finite domain possible. In Section 7.4.1 we give a general introduction to the comparison; in Section 7.4.2 we provide problem transformations between model expansion and Herbrand model generation problems.

*Remark* 7.5. We reflect on the relation of these computational tasks to Stable logic programs.

Stable logic programs are interwoven with the use of Herbrand models to a considerable extent. For instance, the original definition of the stable model semantics by Gelfond and Lifschitz (1988) is defined only for Herbrand models, and the use of ground theories is inherent to the Gelfond-Lifschitz operator. For these reasons, Stable logic programs are traditionally used only for the task of Herbrand model generation.

In order to use Stable logic programs with model expansion, one must first define *classical* (non-Herbrand) stable models, as in, e.g., (Denecker, 1993).

### 7.4.1 General introduction

Recall from Section 3.2 that parameterized model expansion generalizes finite model generation: finite model generation is model expansion with an empty input vocabulary. In case the Herbrand universe is finite (i.e. the function free case), Herbrand model generation is a special case of finite model generation, hence, is also generalized by model expansion.

The Herbrand universe is finite iff there are no function symbols except constants. However, also for some classes of theories with an infinite Herbrand universe, it is possible to generate Herbrand models. This is the case if all Herbrand models demonstrably have a finite set of true domain atoms. The following simple ID-logic theory is an example of a theory with an infinite Herbrand universe, but whose only Herbrand model has a finite interpretation for its predicate symbols:

$$\left\{ \; P(x) \leftarrow x = F(A) \; \right\}. \tag{7.17}$$

Its only Herbrand model $I$ has domain $\mathrm{dom}(I) = \{A, F(A), F(F(A)), \dots\}$, and interpretation $P^I = \{F(A)\}$.

To ensure that Herbrand models are finite in this sense even when the Herbrand universe is infinite, syntactical restrictions have to be applied. Syrjänen

(2001) developed such a restriction, called *omega restriction*, which is used in ASP. It involves a stratification of the symbols used, and as such may be less then straightforward to use.

The advantages of model expansion as a framework for declarative problem solving have been adequately pointed out by Mitchell and Ternovska (2005). Here we recall two that have an impact on the methodology of modelling.

**Representation of data**

Parameterized $MX$ makes a conceptual difference between a problem description (represented by the parameters $\langle T, \sigma \rangle$: a given problem encoding and input vocabulary) and a problem instance (represented by an input interpretation), i.e., both are independent of one another.

In Herbrand model generation, by contrast, a problem instance (i.e., data) is usually given as facts that are part of the theory of which a Herbrand model is to be found—and that are therefore methodologically indistinguishable of it.

**The use of function symbols**

The $MX$ task requires a finite input interpretation—and therefore a finite domain—to be given. Hence, function symbols will have a finite interpretation in all expansion models. Therefore function symbols can be used freely in theories that are intended for use with the $MX$ task. Note that many typical constraint problems, such as the graph colouring problem, have a function as their solution.

In contrast, the use of Herbrand model generation places a severe restriction on the use of function symbols.

### 7.4.2 Problem transformations

In this section, we show how to transform a specific *instance* of a parameterized ID-logic-$MX$ problem into a Herbrand model generation problem, and vice versa.

Let $T$ be an ID-logic theory, $\sigma \subseteq \mathit{vocab}(T)$ an instance vocabulary, and $I_\sigma$ a finite $\sigma$-interpretation. We construct a new theory $T'$ from $T$ and $I_\sigma$.

- First make the new theory function free by removing all occurrences of function symbols, as in Section 3.5.
- Add a new predicate symbol $D/1$ to $\sigma$, representing the domain $D$ of $I_\sigma$, and interpret it by $D/1^{I_\sigma} = D$. (If the theory contains multiple types, add a new domain predicate for each type.)
- Next, add a new definition $\Delta_{P/n}$ for every $P/n \in \sigma$, and add to it the facts $P(\overline{d}) \leftarrow \top$ for every $\overline{d} \in P/n^{I_\sigma}$.
- Finally, add the UNA and DCA.

There is a one-to-one correspondence between expansion models of $MX_{\langle T,\sigma \rangle}(I_\sigma)$ and Herbrand models of $T'$.

**Example 7.10.** Consider again the graph colouring problem from Example 3.2. Let $T_{7.10}$ be the theory

$$\begin{array}{rl} \textbf{types:} & Vtx, Clr \\ \textbf{input vocabulary } \sigma\textbf{:} & Edge(Vtx, Vtx) \\ \textbf{expansion vocabulary:} & Colour(Vtx) : Clr \end{array}$$
$$\forall v_1 \forall v_2\ Edge(v_1, v_2) \supset (Colour(v_1) \neq Colour(v_2)).$$

Let $I_\sigma$ be as in Example 3.3: the domains are $Vtx = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ and $Clr = \{\mathsf{R}, \mathsf{G}, \mathsf{B}\}$, and $Edge^{I_\sigma}$ is $\{(\mathsf{a}, \mathsf{b}),\ (\mathsf{b}, \mathsf{c}),\ (\mathsf{c}, \mathsf{a})\}$.

We apply the transformation on $MX_{\langle T,\sigma\rangle}(I_\sigma)$. First, we replace the function $Colour(Vtx) : Clr$ by a predicate $Colour(Vtx, Clr)$. We obtain the theory

$$\begin{array}{rl} \textbf{types:} & Vtx, Clr \\ \textbf{input vocabulary } \sigma\textbf{:} & Edge(Vtx, Vtx) \\ \textbf{expansion vocabulary:} & Colour(Vtx, Clr) \end{array}$$
$$\forall v_1 \forall v_2\ Edge(v_1, v_2) \supset \neg(\exists c\ Colour(v_1, c) \wedge Colour(v_2, c)).$$

We further add the domain predicates $Vtx(Vtx)$ and $Clr(Clr)$, and add definitions to represent $Vtx^{I_\sigma}$, $Clr^{I_\sigma}$, and $Edge^{I_\sigma}$, and we obtain $T' =$

$$\begin{array}{rl} \textbf{types:} & Vtx, Clr \\ \textbf{vocabulary:} & Vtx(Vtx), Clr(Clr), Edge(Vtx, Vtx), Colour(Vtx, Clr) \\ & \forall v_1 \forall v_2\ Edge(v_1, v_2) \supset \neg(\exists c\ Colour(v_1, c) \wedge Colour(v_2, c)), \\ & \left\{\ Vtx(\mathsf{a}) \leftarrow \top,\quad Vtx(\mathsf{b}) \leftarrow \top,\quad Vtx(\mathsf{c}) \leftarrow \top\ \right\}, \\ & \left\{\ Clr(\mathsf{R}) \leftarrow \top,\quad Clr(\mathsf{G}) \leftarrow \top,\quad Clr(\mathsf{B}) \leftarrow \top\ \right\}, \\ & \left\{\ Edge(\mathsf{a}, \mathsf{b}) \leftarrow \top,\quad Edge(\mathsf{b}, \mathsf{c}) \leftarrow \top,\quad Edge(\mathsf{c}, \mathsf{a}) \leftarrow \top\ \right\}. \end{array}$$

The UNA, $(\mathsf{a} \neq \mathsf{b}) \wedge (\mathsf{a} \neq \mathsf{c}) \wedge \cdots \wedge (\mathsf{G} \neq \mathsf{B})$, and the DCA, $\forall x\ x = \mathsf{a} \vee x = \mathsf{b} \vee x = \mathsf{c} \vee x = \mathsf{R} \vee x = \mathsf{G} \vee x = \mathsf{B}$, can be left implicit, as is customary in Herbrand model generation.

The reverse transformation simply transforms the Herbrand model generation problem for a theory $T$ into a parameterized $MX$ problem $MX_{\langle T,\sigma\rangle}(I_\sigma)$, where $\sigma = \emptyset$, and $I_\sigma$ is the interpretation with the Herbrand universe as domain. This is only a valid $MX$ problem when $vocab(T)$ contains no function symbols except constants, so that the Herbrand universe is finite.

## 7.5 Conclusion

In this chapter, we have compared Stable logic programs and ID-logic from the viewpoint of the ASP paradigm. It should be noted that both logics may have merits as a useful knowledge representation language also for other paradigms or other computational tasks, such as deductive reasoning. Since ID-logic is based

on FO, the extensive body of work on deductive reasoning (theorem proving) for classical logic can serve as a basis for deductive systems for ID-logic. Hou et al. (2007) introduces foundational work for such a deductive system.

We have provided transformations from ID-logic to Stable logic programs and vice versa. The ID-logic to Stable logic programs transformation consists of a number of inter-ID-logic transformations that simplify the theory to a format closely resembling normal logic programs, followed by a step that introduces double negation rules for non-defined symbols. The reverse transformation, from Stable logic programs to ID-logic, moves all negations outside the scope of inductive definitions and creates a negation-free inductive definition.

We have investigated the applicability of the ID-logic modelling principles from Section 3.1 in Stable logic programs. Finally, we have compared model expansion and Herbrand model generation in Section 7.4.

In conclusion of this comparison, we believe that ID-logic is a better knowledge representation language than Stable logic programs.

# Chapter 8

# Conclusion

This text studied the topic of *model generation for ID-logic* . We summarize its contributions and provide some directions for future work.

## 8.1 Summary of contributions

We have studied methodological principles of modelling in ID-logic, given examples, and presented the IDP system for model expansion for ID-logic.

We have made a semantical study of PC(ID), giving alternative characterizations of the models of propositional inductive definitions (IDs). In particular, we have proven Theorems 4.1 and 4.2, which characterized IDs by means of *justification graphs* and *loop formulas*, respectively. The second result offers the first vocabulary-preserving reduction of IDs to propositional calculus.

We have made an algorithmic study of SAT(ID), presenting various strategies for solving SAT(ID) problems, and providing algorithms for several of these strategies. In particular, Figures 5.3 and 5.4 offer various transition rules for SAT(ID) algorithms, and Algorithms 5.2, 5.3, and 5.4 offer implementations for the most important of these rules. Algorithm 5.3 is the first algorithm to implement the Δ-Propagate transition rule, which can reasonably be called the most general propagation rule for deriving literals possible for IDs. This algorithmic study made heavy use of the semantical characterization presented in earlier sections.

Also, we have implemented the MiniSat(ID) system, a SAT(ID) solver, as an extension of an existing SAT solver, discussed its strategy (cf. Figures 5.8 and 5.9) and implementation, and demonstrated that its performance is state-of-the-art.

We have also made a semantical and algorithmic study of an extension of PC(ID) with aggregate expressions, and its associated satisfiability problem. In particular, Figures 6.1 and 6.2 offer transition rules for non-recursive aggregate expressions, and Algorithms 6.2–6.5 extend Algorithms 5.2 and 5.4 to recursive aggregate expressions. They provide the first implementation of the

well-founded semantics for aggregates developed by Pelov et al. (2007). This study was based on, and extended, the work on justifications presented in earlier sections.

Finally, we have made a comparison of ID-logic and Stable logic programs, by providing transformations between them, by investigating to what extent ID-logic's methodological principles of modelling are applicable in Stable logic programs, and by comparing Herbrand model generation to model expansion.

## 8.2 Future work

There are many ways in which this work could be extended.

First, the semantical analysis of Chapter 4 can be expanded. For instance, it could be studied whether a *minimal* set $\mathcal{L}$ of loops such that $\Delta \equiv comp(\Delta) \cup \bigwedge_{L \in \mathcal{L}} LF_\Delta(L)$ can be characterized. Such a characterization might shed important light on the optimal strategy for a SAT(ID) solver to follow. The characterization of elementary loops in this work is but a first step in that direction.

The characterization of PC(ID) using loop formulas may be used as a basis to define new language extensions. In particular, inductive definitions can be seen as a *least fixpoint* construct, and accordingly, loops containing an atom must get external justification. It seems possible to define a similar *greatest fixpoint* construct, and modify the loop formula characterization: for a greatest fixpoint expression, loops containing a *negative literal* must get external justification. In fact, such a study is being conducted in our research group.

The study of algorithms for the SAT(ID) problem in Chapter 5 can be strengthened in different ways.

First, a more in-depth study of the FwLoop transition rule could be made. Our Algorithm 5.3 implements this rule (and slightly generalizes it), but it is naive in its choices. A thorough investigation may reveal whether more intelligent choices can be made.

Second, the study of repetitive application of the BwLoop transition rule can be deepened. For instance, the order of processing different cycle sources potentially has a big impact on the loops being derived, their sizes, etcetera.

Third, since experimental results indicate that, for best performance, the choice of strategy should be tuned to the type of problem being solved, it could be investigated whether dynamic strategy-choosing techniques may automate this tuning. The *adaptive* strategy implemented in MiniSat(ID) is but the first of wide range of possibilities for such dynamic behaviour.

Fourth, preprocessing techniques for PC(ID) should be researched. Such techniques yield great payoff in SAT; the question is whether model-preserving and semantics-preserving preprocessing techniques may yield the same payoff in model generation for PC(ID).

Finally, an implementation is never ready; details of MiniSat(ID)'s implementation can be improved upon in various ways.

In Chapter 6, the most obvious item of future work is to generalize all results concerning loop formula semantics to definitions with aggregate expressions.

Also here, the algorithms and their implementation may be fine-tuned in various ways.

# Appendix A

# Extended Clausal Normal Form (ECNF)

In this appendix, we define the ASCII syntax for ECNF theories. For the formal definition of the logical syntax of ECNF theories, we refer to Sections 4.2.2, 5.2.4 and 6.2.2.

The syntax is derived from the DIMACS syntax for CNF theories. In this syntax, the following conventions are used:

- atoms are represented by strictly positive integers;
- the negation of an atom $N$ is represented by $-N$;
- clauses are represented by whitespace-separated lists of literals, and are terminated by a 0.

The syntax further allows *comments*: lines that start (first symbol on a new line) by a "`c`". It further contains a notation for a *problem statement*: a line that starts by "`p cnf`", and further contains two positive integers, representing respectively the largest atom that occurs (positively or negatively) in the theory, and the number of clauses in the theory. This problem statement should occur before any clauses (but potentially after some comments). In practice, most contemporary SAT solvers disregard this information, which is furthermore difficult to produce for a grounder.

**Example A.1.** The CNF theory

$$x_1 \lor \neg x_2 \lor x_3,$$
$$\neg x_1 \lor x_2,$$
$$\neg x_3 \lor \neg x_1,$$
$$x_1 \lor x_2 \lor \neg x_3,$$

can be represented in DIMACS CNF as follows:

```
c This is a comment.
p cnf 3 4
1 -2 3 0
-1 2 0
3 -1 0
1 2 -3 0
```

Apart from the comment and problem statement line, a DIMACS CNF file contains only integers and whitespace. In the following, will will implicitly assume any list and any separately mentioned item to be whitespace-separated, and literals to be represented as in DIMACS CNF.

We begin by modifying the problem statement: the problem statement of an ECNF theory starts by "`p ecnf`", and is followed by a (whitespace-separated) list of extensions that are used in the theory.

- if definitions occur, "`def`" must be in the list;
- if aggregate expressions occur, "`aggr`" must be in the list;
- if exists-unique expressions occur, "`eu`" must be in the list;
- if at-most-one expressions occur, "`amo`" must be in the list.

## A.1 Definitions

An ECNF theory may contain one definition that is in DefNF normal form. Recall that a DefNF definition contains exactly one rule for each defined atom, and that each rule body is either a disjunction or a conjunction of literals.

We represent a disjunctively defined rule by "`D`", and a conjunctively defined rule by "`C`". This declaration is then followed first by the atom being defined, next by a list of body literals, and finally, terminated by a 0. Note in particular that both the empty disjunctive body (false) and the empty conjunctive body (true) can be represented, by following the atom being defined by an empty list of body literals.

**Example A.2.** We represent the ECNF theory

$$x_1 \vee \neg x_2 \vee x_3,$$
$$\left\{ \begin{array}{l} x_1 \leftarrow \neg x_2 \vee x_3, \\ x_2 \leftarrow x_1 \wedge x_3 \end{array} \right\}$$

in this format:

```
p ecnf def
1 -2 3 0
D 1 -2 3 0
C 2 1 3 0
```

There is no requirement that a set of rules be represented contiguously; other expressions, such as clauses, may be interspersed.

## A.2 Aggregate expressions

An ECNF theory may contain declarations of sets and weighted sets as follows:

- a set is declared by "`Set`", followed by a positive integer that uniquely denotes the set (note that this integer is of another category than atoms; thus the same number may be used both to denote an atom and a set), followed by a list of literals in the set, terminated by a 0;
- a weighted set is declared by "`WSet`", followed by a positive integer that uniquely denotes the weighted set (whereby the category of sets and weighted sets is the same), followed by a list of pairs of the form $l$=$w$, where $l$ is a literal, and $w$ an integer weight, and terminated by a 0.

Such sets and weighted sets must be non-empty.

An aggregate expression of the form $d \leftarrow lwr \leq Aggr(S) \leq upr$ is represented as follows:

- first, a declaration of the type $Aggr$: respectively "`Card`", "`Sum`", "`Prod`", "`Min`", or "`Max`";
- next, the atom $d$;
- followed by the integer denoting the set or weighted set $S$. This (weighted) set must have been declared before; the type of set (weighted or not) must correspond to the aggregate type (`Card` requires a normal set, the others a weighted set);
- followed by the integers $lwr$ and $upr$, and terminated by a 0.

**Example A.3.** The ECNF theory

$$S_1 = \{(x_1, 5), (\neg x_2, 3)\},$$
$$\left\{ \begin{array}{l} x_3 \leftarrow 1 \leq Sum(S_1) \leq 7, \\ x_4 \leftarrow 4 \leq Sum(S_1) \leq 5 \end{array} \right\}$$

can be represented by

```
Set 1 1=5 2=3 0
Sum 3 1 1 7 0
Sum 4 1 4 5 0
```

## A.3 Exists-unique and at-most-one expressions

An expression of the form $EU(S)$, where $S$ is a set of literals, is represented by "`EU`", followed by a list of the literals in $S$, terminated by 0. Likewise, an expression of the form $AMO(S)$ is represented by "`AMO`", followed by a list of the literals in $S$, terminated by 0. In both cases, $S$ must be a non-empty set.

**Example A.4.** The ECNF theory

$$EU(\{x_1, \neg x_2, x_3\}),$$
$$AMO(\{\neg x_1, x_2, \neg x_3\}),$$
$$x_1 \vee x_2$$

can be represented by

```
EU 1 -2 3 0
AMO -1 2 -3 0
1 2 0
```

# Appendix B

# Problem encodings

We give the IDP encoding of the problems from Sections 3.3, 5.7 and 6.4.

## B.1 Hamiltonian circuit

In IDP, using inductive definitions:

```
Given:
   type Vtx
   Edge(Vtx,Vtx)

Find:
   Hc(Vtx,Vtx)

Satisfying:
   ! x y z : Hc(x,y) & Hc(x,z) => y = z.
   ! x y z : Hc(y,x) & Hc(z,x) => y = z.
   declare Reached(Vtx).
   {  Reached(x) <- Hc(MIN,x).
      Reached(x) <- Reached(y) & Hc(y,x).
   }
   ! x : Reached(x).
```

In IDP, using only inductive definitions that depend on instance vocabulary:

```
Given:
   type Vtx
   Edge(Vtx,Vtx)

Find:
   Hc(Vtx,Vtx)
```

```
Declare:
   Map(Vtx,Vtx)
   Next(Vtx,Vtx)
   Path(Vtx,Vtx)

Satisfying:

    {Next(MAX,MIN).
     Next(u,v) <- SUCC(u,v).
    }

    // Path(x,n) iff there is a path of length exactly n from MIN to x.
    {Path(MIN,MIN).
     Path(y,np1) <- Path(x,n) & SUCC(n,np1) & Edge(x,y).
    }

    // Each vertex gets a unique "mapping number",
    // which must be possible according to Path.
    Map(MIN,MIN).
    ! x: ?1 n: Map(x,n).
    ! n: ?1 x: Map(x,n).
    ! x n: Map(x,n) => Path(x,n).

    // Consecutively mapped vertices should have an existing edge.
    ! u v n1 n2: Map(u,n1) & Map(v,n2) & Next(n1,n2) => Edge(u,v).

    // Now define the Hamiltonian edges as the edges between any
    // two consecutive vertices according to the mapping.
    ! u v n1 n2: Map(u,n1) & Map(v,n2) & Next(n1,n2) => Hc(u,v).
    ! u v: Hc(u,v) => ? n1 n2: Map(u,n1) & Map(v,n2) & Next(n1,n2).
```

## B.2 Sokoban puzzle

```
Given:
   type {
      int Step
      int Row
      int Col
      Block
   }
   Square     (Row, Col)
   Goal       (Row, Col)
   InitManX           : Row
   InitManY           : Col
   InitBlockX (Block) : Row
   InitBlockY (Block) : Col
```

```
   MaxStep : Step

Find:
   Move   (Step) : Block
   MoveTo (Step,Row,Col)

Declare:
   Block     (Step, Block, Row, Col)
   Man       (Step, Row, Col)
   Reachable (Step, Row, Col)
   NoBlock   (Step, Row, Col)
   FreeX     (Step, Row, Row, Col)
   FreeY     (Step, Row, Col, Col)
   NextTo    (Row, Col, Row, Col)

Satisfying:
   // neighbour positions
   {  NextTo(x1,y1,x2,y2) <- abs(x1-x2) + abs(y1-y2) = 1
                             & Square(x1,y1)
                             & Square(x2,y2).
   }

   // free positions at each timestep
   {  NoBlock(t,x,y) <- Square(x,y) & ~(? b : Block(t,b,x,y)).
   }

   // reachable positions at each timestep
   {  Reachable(t,x,y) <- Man(t,x,y).
      Reachable(t,x,y) <- Reachable(t,x1,y1)
                          & NoBlock(t,x,y)
                          & NextTo(x,y,x1,y1).
   }

   // one move per timestep
   ! t x1 y1 x2 y2 : MoveTo(t,x1,y1) & MoveTo(t,x2,y2) =>
                     x1=x2 & y1=y2.
   ! x y : ~MoveTo(MaxStep,x,y).
   ! t : Step(t) & t < MaxStep => ? x y : MoveTo(t,x,y).

   // move only to squares
   ! t x y : MoveTo(t,x,y) => Square(x,y).

   // the block moves over a free line
   ! t b x y : MoveTo(t,x,y) & Move(t)=b =>
               (? y_old : Block(t,b,x,y_old) & FreeY(t,x,y,y_old)) |
               (? x_old : Block(t,b,x_old,y) & FreeX(t,x,x_old,y)).

   // the block to be moved must be reachable
   ! t b x x_old y : MoveTo(t,x,y) & Move(t)=b
                     & Block(t,b,x_old,y) & x<x_old =>
```

```
                  (? z : z = x_old+1 & Reachable(t,z,y)).
! t b x x_old y : MoveTo(t,x,y) & Move(t)=b
                  & Block(t,b,x_old,y) & x>x_old =>
                  (? z : z = x_old-1 & Reachable(t,z,y)).
! t b x y y_old : MoveTo(t,x,y) & Move(t)=b
                  & Block(t,b,x,y_old) & y<y_old =>
                  (? z : z = y_old+1 & Reachable(t,x,z)).
! t b x y y_old : MoveTo(t,x,y) & Move(t)=b
                  & Block(t,b,x,y_old) & y>y_old =>
                  (? z : z = y_old-1 & Reachable(t,x,z)).

// free lines
{  FreeX(t,x_new,x_old,y) <- x_new < x_old
                             & (! x : x_new =< x & x < x_old =>
                                      Square(x,y) & NoBlock(t,x,y)).
   FreeX(t,x_new,x_old,y) <- x_new > x_old
                             & (! x : x_new >= x & x > x_old =>
                                      Square(x,y) & NoBlock(t,x,y)).
}
{  FreeY(t,x,y_new,y_old) <- y_new < y_old
                             & (! y : y_new =< y & y < y_old =>
                                      Square(x,y) & NoBlock(t,x,y)).
   FreeY(t,x,y_new,y_old) <- y_new > y_old
                             & (! y : y_new >= y & y > y_old =>
                                      Square(x,y) & NoBlock(t,x,y)).
}

// position of the man and the blocks
{  Man(0,InitManX,InitManY).
   Man(t+1,x+1,y) <- MoveTo(t,x,y) & Move(t)=b
                     & Block(t,b,x_old,y) & x<x_old.
   Man(t+1,x-1,y) <- MoveTo(t,x,y) & Move(t)=b
                     & Block(t,b,x_old,y) & x>x_old.
   Man(t+1,x,y+1) <- MoveTo(t,x,y) & Move(t)=b
                     & Block(t,b,x,y_old) & y<y_old.
   Man(t+1,x,y-1) <- MoveTo(t,x,y) & Move(t)=b
                     & Block(t,b,x,y_old) & y>y_old.

   Block(0,b,InitBlockX(b),InitBlockY(b)).
   Block(t+1,b,x,y) <- Move(t)~=b & Block(t,b,x,y).
   Block(t+1,b,x,y) <- Move(t)=b & Block(t,b,x,y)
                       & (! xx yy : ~MoveTo(t,xx,yy)).
   Block(t+1,b,x,y) <- Move(t)=b & MoveTo(t,x,y).
}

// goal
! x y b : Block(MaxStep,b,x,y) => Goal(x,y).
```

## B.3 Hitori puzzle

```
Given:
   type {
      int Xpos
      int Ypos
      int Number
   }
   State(Xpos,Ypos,Number)

Find:
   Black(Xpos,Ypos)

Satisfying:
   declare NextTo(Xpos,Ypos,Xpos,Ypos).
   {  NextTo(x1,y1,x2,y2) <- abs(x1 - x2) + abs(y1 - y2) = 1.
   }

   ! x1 y1 x2 y2 : NextTo(x1,y1,x2,y2) => ~(Black(x1,y1) & Black(x2,y2)).

   ! x1 x2 y n : State(x1,y,n) & State(x2,y,n)
               & ~Black(x1,y) & ~Black(x2,y) => x1 = x2.
   ! x y1 y2 n : State(x,y1,n) & State(x,y2,n)
               & ~Black(x,y1) & ~Black(x,y2) => y1 = y2.

   declare Reachable(Xpos,Ypos).
   {  Reachable(1,1) <- ~Black(1,1).
      Reachable(1,2) <- Black(1,1).
      Reachable(x,y) <- NextTo(x,y,rx,ry)
                        & ~Black(x,y)
                        & Reachable(rx,ry).
   }

   ! x y : Xpos(x) & Ypos(y) & ~Black(x,y) => Reachable(x,y).
```

## B.4 Transitive opening

```
Given:
type Vtx
TC(Vtx,Vtx)

Find:
R(Vtx,Vtx)

Satisfying:
    {  TC(x,y) <- R(x,y).
```

```
    TC(x,y) <- TC(x,z) & R(z,y).
 }

 declare RelTC(Vtx,Vtx,Vtx,Vtx).
 {  RelTC(x,y,u,v) <- R(x,y) & ~(x=u & y=v).
    RelTC(x,y,u,v) <- ?z: RelTC(x,z,u,v) & RelTC(z,y,u,v).
 }

 ! x y : R(x,y) => ~RelTC(x,y,x,y).
```

## B.5 Social golfer

```
Given:
   type {
      Players
      Groups
      Weeks
      int Size
   }
   GroupSize : Size

Find:
   Plays(Weeks,Groups,Players)

Satisfying:
   // Each golfer plays in exactly one group every week.
   ! p w : ?1 g : Plays(w,g,p).

   // The number of players in each group is equal to groupsize.
   ! w g : #{ p : Plays(w,g,p) } = GroupSize.

   // Each pair of players meets at most once.
   ! p1 p2 : p1 < p2 => #{ w : ? g : Plays(w,g,p1) & Plays(w,g,p2) } =< 1.
```

## B.6 Magic series

```
Given:
   type int Num

Find:
   El(Num) : Num

Satisfying:
   ! x : El(x) = #{ y :  El(y) = x }.
```

# Appendix C

# Complete version of the BwLoop algorithm

In this appendix, we give the version of the BwLoop algorithm (Algorithm 5.2) as it is implemented in MiniSat(ID). It is given by Algorithm C.1. This algorithm differs from Algorithm 5.2 in two aspects:

- it uses a system of *guards* for conjuctively defined literals;
- it removes other cycle sources than the given cycle source $cs$ from the set of cycle sources $CS$.

We use the same symbols as in Sections 5.4 and 5.5.

**Algorithm C.1**: Justifying a cycle source $cs$

```
1  Let HP := {l | l has a path to cs in J_s through non-false literals};
2  if cs ∉ HP then CS := CS \ {cs}; return ∅;
3  Let L := Q := {cs};
4  while Q ≠ ∅ do
5      Pop literal l from Q;
6      if l ∈ D_lits then
7          if ∃l' ∈ Bd_l such that l' ∉ HP and M(l') ≠ f then
8              d_{J_s}(l) := l'; Justify(l);
9          else
10             foreach l' ∈ Bd_l such that M(l') ≠ f and l' ∉ L do
11                 Add l' to Q and to L;
12     else
13         if ¬∃l' ∈ Bd_l ∩ HP then
14             Justify(l);
15         else
16             if ∃l' ∈ Bd_l ∩ L then guard(l) := l';
17             else Choose some l' ∈ Bd_l ∩ HP; guard(l) := l'; Add l' to Q
                   and to L;
18 return L;
19 function Justify l
20     Remove l from HP, L and Q;
21     if l ∈ CS then CS := CS \ {l};
22     if l = cs then return ∅;
23     foreach l' ∈ L with guard(l') = l do
24         Push l' on Q;
25     foreach l' ∈ L ∩ D_lits with l ∈ Bd_l' do
26         d_{J_s}(l') := l; Justify(l');
27 end function
```

# Bibliography

Christian Anger, Martin Gebser, Thomas Linke, André Neumann, and Torsten Schaub. The *nomore++* approach to answer set solving. In Sutcliffe and Voronkov (2005), pages 95–109. ISBN 3-540-30553-X.

Christian Anger, Martin Gebser, Tomi Janhunen, and Torsten Schaub. What's a head without a body? In Brewka et al. (2006), pages 769–770. ISBN 1-58603-642-4.

Christian Anger, Martin Gebser, and Torsten Schaub. Approaching the core of unfounded sets. In Jürgen Dix and Anthony Hunter, editors, *NMR*, pages 58–66, 2006b.

Marcello Balduccini and Michael Gelfond. Diagnostic reasoning with A-Prolog. *Theory and Practice of Logic Programming (TPLP)*, 3(4-5):425–461, 2003.

Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors. *Logic Programming and Nonmonotonic Reasoning, 8th International Conference, LPNMR 2005, Diamante, Italy, September 5-8, 2005, Proceedings*, volume 3662 of *Lecture Notes in Computer Science*, 2005. Springer. ISBN 3-540-28538-5.

Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors. *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*, 2007. Springer. ISBN 978-3-540-72199-4.

Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research (JAIR)*, 22:319–351, 2004.

Nuel D. Belnap. A useful four-valued logic. In J. Michael Dunn and George Epstein, editors, *Modern Uses of Multiple-Valued Logic*, pages 8–37. Reidel, Dordrecht, 1977. Invited papers from the Fifth International Symposium on Multiple-Valued Logic, held at Indiana University, Bloomington, Indiana, May 13-16, 1975.

Kenneth A. Berman, John S. Schlipf, and John V. Franco. Computing well-founded semantics faster. In Victor W. Marek and Anil Nerode, editors, *LPNMR*, volume 928 of *Lecture Notes in Computer Science*, pages 113–126. Springer, 1995. ISBN 3-540-59487-6.

Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In Kleine Büning and Zhao (2008), pages 28–33. ISBN 978-3-540-79718-0.

Georg Boenn, Martin Brain, Marina De Vos, and John Fitch. Automatic composition of melodic and harmonic music by answer set programming. In María García de la Banda and Enrico Pontelli, editors, *ICLP*, volume 5366 of *Lecture Notes in Computer Science*, pages 160–174. Springer, 2008. ISBN 978-3-540-89981-5.

Stefan Brass, Jürgen Dix, Burkhard Freitag, and Ulrich Zukowski. Transformation-based bottom-up computation of the well-founded model. *Theory and Practice of Logic Programming (TPLP)*, 1(5):497–538, 2001.

Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors. *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, 2006. IOS Press. ISBN 1-58603-642-4.

Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer, 1994. ISBN 3-540-58179-0.

Weidong Chen and David Scott Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.

Keith L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322. Plenum Press, 1978. ISBN 0-306-40060-X.

Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971.

Álvaro Cortés Calabuig. *Towards a logical reconstruction of a theory for locally complete databases.* PhD thesis, K.U.Leuven, Leuven, Belgium, December 2008.

Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Christoph Koch, Nicola Leone, Simona Perri, and Gerald Pfeifer. System description: Dlv. In Eiter et al. (2001), pages 424–428. ISBN 3-540-42593-4.

Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerald Pfeifer. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in DLV. In Georg Gottlob and Toby Walsh, editors, *IJCAI*, pages 847–852. Morgan Kaufmann, 2003.

Marc Denecker. The well-founded semantics is the principle of inductive definition. In Jürgen Dix, Luis Fariñas del Cerro, and Ulrich Furbach, editors, *JELIA*, volume 1489 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1998. ISBN 3-540-65141-1.

Marc Denecker. Extending classical logic with inductive definitions. In Lloyd et al. (2000), pages 703–717. ISBN 3-540-67797-6.

Marc Denecker. *Knowledge representation and reasoning in incomplete logic programming*. PhD thesis, K.U.Leuven, Leuven, Belgium, September 1993.

Marc Denecker and Danny De Schreye. Justification semantics: A unifying framework for the semantics of logic programs. In Luís Moniz Pereira and Anil Nerode, editors, *LPNMR*, pages 365–379. MIT Press, 1993. ISBN 0-262-66083-0.

Marc Denecker and Danny De Schreye. SLDNFA: An abductive procedure for abductive logic programs. *Journal of Logic Programming*, 34(2):111–167, 1998.

Marc Denecker and Eugenia Ternovska. A logic of non-monotone inductive definitions and its modularity properties. In Lifschitz and Niemelä (2004), pages 47–60. ISBN 3-540-20721-X.

Marc Denecker and Eugenia Ternovska. Inductive situation calculus. *Artificial Intelligence*, 171(5-6):332–360, 2007.

Marc Denecker and Eugenia Ternovska. A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic (TOCL)*, 9(2):Article 14, 2008.

Marc Denecker and Joost Vennekens. Well-founded semantics and the algebraic theory of non-monotone inductive definitions. In Baral et al. (2007), pages 84–96. ISBN 978-3-540-72199-4.

William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.

Deborah East and Mirosław Truszczyński. Predicate-calculus-based logics for modeling and solving search problems. *ACM Transactions on Computational Logic (TOCL)*, 7(1):38–83, 2006.

Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005. ISBN 3-540-26276-8.

Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. ISBN 3-540-20851-8.

Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors. *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17-19, 2001, Proceedings*, volume 2173 of *Lecture Notes in Computer Science*, 2001. Springer. ISBN 3-540-42593-4.

Herbert B. Enderton. *A Mathematical Introduction To Logic*. Academic Press, second edition, 2001.

Esra Erdem and Vladimir Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming (TPLP)*, 3(4-5):499–518, 2003.

Esra Erdem and Ferhan Türe. Efficient haplotype inference with answer set programming. In Fox and Gomes (2008), pages 436–441. ISBN 978-1-57735-368-3.

Sandro Etalle and Mirosław Truszczyński, editors. *22nd International Conference on Logic Programming, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4079 of *Lecture Notes in Computer Science*, 2006. Springer. ISBN 3-540-36635-0.

Wolfgang Faber, Gerald Pfeifer, Nicola Leone, Tina Dell'Armi, and Giuseppe Ielpa. Design and implementation of aggregate functions in the DLV system. *Theory and Practice of Logic Programming (TPLP)*, 8(5-6):545–580, 2008.

Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. A generalization of the lin-zhao theorem. *Annals of Mathematics and Artificial Intelligence*, 47(1-2): 79–101, 2006.

Dieter Fox and Carla P. Gomes, editors. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, 2008. AAAI Press. ISBN 978-1-57735-368-3.

Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL($T$): Fast decision procedures. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004. ISBN 3-540-22342-8.

Martin Gebser and Torsten Schaub. Tableau calculi for answer set programming. In Etalle and Truszczyński (2006), pages 11–25. ISBN 3-540-36635-0.

Martin Gebser and Torsten Schaub. Generic tableaux for answer set programming. In Verónica Dahl and Ilkka Niemelä, editors, *ICLP*, volume 4670 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 2007. ISBN 978-3-540-74608-9.

Martin Gebser and Torsten Schaub. Loops: Relevant or redundant? In Baral et al. (2005), pages 53–65. ISBN 3-540-28538-5.

Martin Gebser, Joohyung Lee, and Yuliya Lierler. Elementary sets for logic programs. In *AAAI*. AAAI Press, 2006.

Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In Manuela M. Veloso, editor, *IJCAI*, pages 386–392, 2007a.

Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. *clasp*: A conflict-driven answer set solver. In Baral et al. (2007), pages 260–265. ISBN 978-3-540-72199-4.

Martin Gebser, Lengning Liu, Gayathri Namasivayam, André Neumann, Torsten Schaub, and Mirosław Truszczyński. The first answer set programming system competition. In Baral et al. (2007), pages 3–17. ISBN 978-3-540-72199-4.

Michael Gelfond and Nicola Leone. Logic programming and knowledge representation—the A-Prolog perspective. *Artificial Intelligence*, 138(1-2): 3–38, 2002.

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

Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *ICLP/SLP*, pages 1070–1080. MIT Press, 1988. ISBN 0-262-61056-6.

Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors. *Logic Programming and Nonmonotonic Reasoning, 5th International Conference, LPNMR'99, El Paso, Texas, USA, December 2-4, 1999, Proceedings*, volume 1730 of *Lecture Notes in Computer Science*, 1999. Springer. ISBN 3-540-66749-0.

Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. SAT-based answer set programming. In Deborah L. McGuinness and George Ferguson, editors, *AAAI*, pages 61–66. AAAI Press / The MIT Press, 2004. ISBN 0-262-51183-5.

Eugene Goldberg and Yakov Novikov. BerkMin: A fast and robust SAT-solver. In *DATE*, pages 142–149. IEEE Computer Society, 2002. ISBN 0-7695-1471-5.

Aarti Gupta and Sharad Malik, editors. *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, 2008. Springer. ISBN 978-3-540-70543-7.

Aarti Gupta, Malay K. Ganai, and Chao Wang. SAT-based verification methods and applications in hardware verification. In Marco Bernardo and Alessandro Cimatti, editors, *SFM*, volume 3965 of *Lecture Notes in Computer Science*, pages 108–143. Springer, 2006. ISBN 978-3-540-34304-2.

Keijo Heljanko and Ilkka Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming (TPLP)*, 3(4-5):519–550, 2003.

Ping Hou, Johan Wittocx, and Marc Denecker. A deductive system for PC(ID). In Baral et al. (2007), pages 162–174. ISBN 978-3-540-72199-4.

Tomi Janhunen. Representing normal programs with clauses. In Ramon López de Mántaras and Lorenza Saitta, editors, *ECAI*, pages 358–362. IOS Press, 2004. ISBN 1-58603-452-9.

Matti Järvisalo, Tommi A. Junttila, and Ilkka Niemelä. Unrestricted vs restricted cut in a tableau method for boolean circuits. *Annals of Mathematics and Artificial Intelligence*, 44(4):373–399, 2005.

Matti Järvisalo, Tommi A. Junttila, and Ilkka Niemelä. Justification-based nonclausal local search for SAT. In Malik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikolaos M. Avouris, editors, *ECAI*, pages 535–539. IOS Press, 2008. ISBN 978-1-58603-891-5.

Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

David B. Kemp and Peter J. Stuckey. Semantics of logic programs with aggregates. In Vijay A. Saraswat and Kazunori Ueda, editors, *ISLP*, pages 387–401. MIT Press, 1991. ISBN 0-262-69147-7.

David B. Kemp, Divesh Srivastava, and Peter J. Stuckey. Bottom-up evaluation and query optimization of well-founded models. *Theoretical Computer Science*, 146(1&2):145–184, 1995.

Hans Kleine Büning and Xishun Zhao, editors. *Theory and Applications of Satisfiability Testing – SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, volume 4996 of *Lecture Notes in Computer Science*, 2008. Springer. ISBN 978-3-540-79718-0.

Kathrin Konczak and Ralf Vogel. Abduction and preferences in linguistics. In Baral et al. (2005), pages 384–388. ISBN 3-540-28538-5.

Robert A. Kowalski and Marek J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.

Joohyung Lee. A model-theoretic counterpart of loop formulas. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI*, pages 503–508. Professional Book Center, 2005. ISBN 0-938-07593-4.

Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3):499–562, 2006.

Leonid Libkin. *Elements of Finite Model Theory.* Springer, 2004.

Yuliya Lierler. cmodels - SAT-based disjunctive answer set solver. In Baral et al. (2005), pages 447–451. ISBN 3-540-28538-5.

Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54, 2002.

Vladimir Lifschitz and Ilkka Niemelä, editors. *Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, January 6-8, 2004, Proceedings*, volume 2923 of *Lecture Notes in Computer Science*, 2004. Springer. ISBN 3-540-20721-X.

Vladimir Lifschitz and Alexander A. Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic (TOCL)*, 7(2):261–268, 2006.

Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25(3-4): 369–389, 1999.

Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

Lengning Liu and Mirosław Truszczyński. Properties and applications of programs with monotone and convex constraints. *Journal of Artificial Intelligence Research (JAIR)*, 27:299–334, 2006.

John W. Lloyd and Rodney W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors. *Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings*, volume 1861 of *Lecture Notes in Computer Science*, 2000. Springer. ISBN 3-540-67797-6.

Zbigniew Lonc and Mirosław Truszczyński. On the problem of computing the well-founded semantics. In Lloyd et al. (2000), pages 673–687. ISBN 3-540-67797-6.

Victor W. Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In Krzysztof R. Apt, Victor W. Marek, Mirosław Truszczyński, and David S. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.

Victor W. Marek, Ilkka Niemelä, and Mirosław Truszczyński. Logic programs with monotone cardinality atoms. In Lifschitz and Niemelä (2004), pages 154–166. ISBN 3-540-20721-X.

Maarten Mariën. MiniSat(ID) website. `http://www.cs.kuleuven.be/~dtai/krr/software/minisatid.html`, 2008.

Maarten Mariën, David Gilis, and Marc Denecker. On the relation between ID-Logic and Answer Set Programming. In José Júlio Alferes and João Alexandre Leite, editors, *JELIA*, volume 3229 of *Lecture Notes in Computer Science*, pages 108–120. Springer, 2004. ISBN 3-540-23242-7.

Maarten Mariën, Rudradeb Mitra, Marc Denecker, and Maurice Bruynooghe. Satisfiability checking for PC(ID). In Sutcliffe and Voronkov (2005), pages 565–579. ISBN 3-540-30553-X.

Maarten Mariën, Johan Wittocx, and Marc Denecker. The IDP framework for declarative problem solving. In *Search and Logic: Answer Set Programming and SAT*, pages 19–34, 2006.

Maarten Mariën, Johan Wittocx, and Marc Denecker. MidL: a SAT(ID) solver. In *4th Workshop on Answer Set Programming: Advances in Theory and Implementation*, pages 303–308, 2007a.

Maarten Mariën, Johan Wittocx, and Marc Denecker. Integrating inductive definitions in SAT. In Nachum Derschowitz and Andrei Voronkov, editors, *LPAR*, volume 4790 of *Lecture Notes in Computer Science*, pages 378–392. Springer, 2007b. ISBN 3-540-75558-6.

Maarten Mariën, Johan Wittocx, Marc Denecker, and Maurice Bruynooghe. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In Kleine Büning and Zhao (2008), pages 211–224. ISBN 978-3-540-79718-0.

João P. Marques-Silva and Inês Lynce. Towards robust CNF encodings of cardinality constraints. In Christian Bessiere, editor, *CP*, volume 4741 of *Lecture Notes in Computer Science*, pages 483–497. Springer, 2007. ISBN 978-3-540-74969-1.

João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

David G. Mitchell. A SAT solver primer. *Bulletin of the European Association for Theoretical Computer Science*, 85:112–132, 2005.

David G. Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 430–435. AAAI Press / The MIT Press, 2005. ISBN 1-57735-236-X.

David G. Mitchell, Eugenia Ternovska, Faraz Hach, and Raheleh Mohebali. Model expansion as a framework for modelling and solving search problems. Technical Report TR 2006-24, Simon Fraser University, Canada, 2006.

Raheleh Mohebali. A method for solving NP search based on model expansion and grounding. Master's thesis, Simon Fraser University, Vancouver, Canada, 2007.

Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC'01*, pages 530–535. ACM, 2001. ISBN 1-58113-297-2.

Inderpal S. Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *VLDB*, pages 264–277. Morgan Kaufmann, 1990. ISBN 1-55860-149-X.

Gi-Joon Nam, Karem A. Sakallah, and Rob A. Rutenbar. Satisfiability-based detailed FPGA routing. In *VLSI Design*, pages 574–577. IEEE Computer Society, 1999.

Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

Ilkka Niemelä, Patrik Simons, and Timo Soininen. Stable model semantics of weight constraint rules. In Gelfond et al. (1999), pages 317–331. ISBN 3-540-66749-0.

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL($t$). *Journal of the ACM*, 53(6):937–977, 2006. ISSN 0004-5411.

Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An a-prolog decision support system for the space shuttle. In I. V. Ramakrishnan, editor, *PADL*, volume 1990 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2001. ISBN 3-540-41768-0.

Emilia Oikarinen. Modular answer set programming. Research Report A106, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 2006.

Emilia Oikarinen and Tomi Janhunen. Modular equivalence for normal logic programs. In Brewka et al. (2006), pages 412–416. ISBN 1-58603-642-4.

Nikolay Pelov. *Semantics of Logic Programs with Aggregates.* PhD thesis, K.U.Leuven, Leuven, Belgium, April 2004.

Nikolay Pelov and Eugenia Ternovska. Reducing inductive definitions to propositional satisfiability. In Maurizio Gabbrielli and Gopal Gupta, editors, *ICLP*, volume 3668 of *Lecture Notes in Computer Science*, pages 221–234. Springer, 2005. ISBN 3-540-29208-X.

Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Partial stable models for logic programs with aggregates. In Lifschitz and Niemelä (2004), pages 207–219. ISBN 3-540-20721-X.

Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming (TPLP)*, 7(3):301–353, 2007.

Enrico Pontelli and Tran Cao Son. *Justifications* for logic programs under answer set semantics. In Etalle and Truszczyński (2006), pages 196–210. ISBN 3-540-36635-0.

Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *STTT*, 7(2):156–173, 2005.

Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

Teodor C. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundamenta Informaticae*, 13(4):445–463, 1990.

Teodor C. Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing*, 9(3/4):401–424, 1991.

Raymond Reiter. On closed world data bases. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, 1977. ISBN 0-306-40060-X.

Raymond Reiter. Towards a logical reconstruction of relational database theory. In Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt, editors, *On Conceptual Modelling (Intervale)*, pages 191–233. Springer, 1982. ISBN 3-540-90842-0.

Raymond Reiter. Equality and domain closure in first-order databases. *Journal of the ACM*, 27(2):235–249, 1980.

Abhik Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying proofs using memo tables. In *PPDP*, pages 178–189. ACM Press, 2000. ISBN 1-58113-265-4.

Lawrence Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, Vancouver, Canada, 2004.

Ken Satoh and Noboru Iwayama. Computing abduction by using the TMS. In Koichi Furukawa, editor, *ICLP*, pages 505–518. MIT Press, 1991. ISBN 0-262-56058-5.

Patrik Simons. Extending the stable model semantics with more expressive rules. In Gelfond et al. (1999), pages 305–316. ISBN 3-540-66749-0.

Patrik Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, TCS, Finland, April 2000. Published as Research Report A58.

Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, 2005. ISBN 3-540-29238-1.

Carsten Sinz. SAT-race 2008. Poster presented at Theory and Applications of Satisfiability Testing (SAT), Guangzhou, China, May 12–15, 2008.

Geoff Sutcliffe and Andrei Voronkov, editors. *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings*, volume 3835 of *Lecture Notes in Computer Science*, 2005. Springer. ISBN 3-540-30553-X.

Tommi Syrjänen. Omega-restricted logic programs. In Eiter et al. (2001), pages 267–279. ISBN 3-540-42593-4.

Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

G. S. Tseitin. On the complexity of derivation in the propositional calculus, *Zapiski nauchnykh seminarov. LOMI*, 8:234–259, 1968. 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.

Allen Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47(1):185–221, 1993.

Allen Van Gelder. The well-founded semantics of aggregation. In *PODS*, pages 127–138. ACM Press, 1992. ISBN 0-89791-519-4.

Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

Bert Van Nuffelen. *Abductive Constraint Logic Programming: Implementation and Applications*. PhD thesis, K.U.Leuven, Leuven, Belgium, April 2004.

Joost Vennekens. *Algebraic and logical study of constructive processes in knowledge representation*. PhD thesis, K.U.Leuven, Leuven, Belgium, May 2007.

Joost Vennekens and Marc Denecker. An algebraic account of modularity in ID-logic. In Baral et al. (2005), pages 291–303. ISBN 3-540-28538-5.

Joost Vennekens, Maarten Mariën, Johan Wittocx, and Marc Denecker. Predicate introduction for logics with a fixpoint semantics. Part I: Logic programming. *Fundamenta Informaticae*, 79(1-2):187–208, September 2007a.

Joost Vennekens, Maarten Mariën, Johan Wittocx, and Marc Denecker. Predicate introduction for logics with a fixpoint semantics. Part II: Autoepistemic logic. *Fundamenta Informaticae*, 79(1-2):209–227, September 2007b.

Jeffrey Ward and John S. Schlipf. Answer set programming with clause learning. In Lifschitz and Niemelä (2004), pages 302–313. ISBN 3-540-20721-X.

Johan Wittocx and Maarten Mariën. The IDP system. `http://www.cs.kuleuven.be/~dtai/krr/software/idpmanual.pdf`, 2008.

Johan Wittocx, Joost Vennekens, Maarten Mariën, Marc Denecker, and Maurice Bruynooghe. Predicate introduction under stable and well-founded semantics. In Etalle and Truszczyński (2006). ISBN 3-540-36635-0.

Johan Wittocx, Maarten Mariën, and Marc Denecker. GIDL: A grounder for FO$^+$. In Maurice Pagnucco and Michael Thielscher, editors, *NMR*, pages 189–198. University of New South Wales, 2008a.

Johan Wittocx, Maarten Mariën, and Marc Denecker. Grounding with bounds. In Fox and Gomes (2008), pages 572–577. ISBN 978-1-57735-368-3.

Johan Wittocx, Maarten Mariën, and Marc Denecker. The IDP system: a model expansion system for an extension of classical logic. In *LASH*, pages 153–165, 2008c.

Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

# Publication List

## Articles in international reviewed journals

Joost Vennekens, Maarten Mariën, Johan Wittocx, and Marc Denecker. Predicate introduction for logics with a fixpoint semantics. Part I: Logic programming. *Fundamenta Informaticae*, 79(1-2):187–208, September 2007.

Joost Vennekens, Maarten Mariën, Johan Wittocx, and Marc Denecker. Predicate introduction for logics with a fixpoint semantics. Part II: Autoepistemic logic. *Fundamenta Informaticae*, 79(1-2):209–227, September 2007.

## Contributions at international conferences

Maarten Mariën, David Gilis, and Marc Denecker. On the relation between ID-Logic and Answer Set Programming. In José Júlio Alferes and João Alexandre Leite, editors, *JELIA*, volume 3229 of *Lecture Notes in Computer Science*, pages 108–120. Springer, 2004. ISBN 3-540-23242-7.

Maarten Mariën, Rudradeb Mitra, Marc Denecker, and Maurice Bruynooghe. Satisfiability checking for PC(ID). In Sutcliffe and Voronkov (2005), pages 565–579. ISBN 3-540-30553-X.

Maarten Mariën, Johan Wittocx, and Marc Denecker. Integrating inductive definitions in SAT. In Nachum Derschowitz and Andrei Voronkov, editors, *LPAR*, volume 4790 of *Lecture Notes in Computer Science*, pages 378–392. Springer, 2007. ISBN 3-540-75558-6.

Maarten Mariën, Johan Wittocx, Marc Denecker, and Maurice Bruynooghe. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In Kleine Büning and Zhao (2008), pages 211–224. ISBN 978-3-540-79718-0.

Johan Wittocx, Joost Vennekens, Maarten Mariën, Marc Denecker, and Maurice Bruynooghe. Predicate introduction under stable and well-founded semantics. In Etalle and Truszczyński (2006). ISBN 3-540-36635-0.

Johan Wittocx, Maarten Mariën, and Marc Denecker. Grounding with bounds. In Fox and Gomes (2008), pages 572–577. ISBN 978-1-57735-368-3.

## Contributions at international conferences, published elsewhere

Maarten Mariën, Johan Wittocx, and Marc Denecker. The IDP framework for declarative problem solving. In *Search and Logic: Answer Set Programming and SAT*, pages 19–34, 2006.

Maarten Mariën, Johan Wittocx, and Marc Denecker. MidL: a SAT(ID) solver. In *4th Workshop on Answer Set Programming: Advances in Theory and Implementation*, pages 303–308, 2007.

Johan Wittocx, Maarten Mariën, and Marc Denecker. The IDP system: a model expansion system for an extension of classical logic. In *LASH*, pages 153–165, 2008.

Johan Wittocx, Maarten Mariën, and Marc Denecker. GidL: A grounder for $FO^+$. In Maurice Pagnucco and Michael Thielscher, editors, *NMR*, pages 189–198. University of New South Wales, 2008.

## Technical Reports

Maarten Mariën, Rudradeb Mitra, Marc Denecker, and Maurice Bruynooghe. Satisfiability checking for PC(ID). Technical Report CW 426, Departement of Computer Science, Katholieke Universiteit Leuven, 2005.

# Biography

Maarten Mariën was born on 13 August 1979 in Mechelen, Belgium. After finishing high school at the Koninklijk Atheneum in Heist-op-den-Berg, he started studying engineering at the Katholieke Universiteit Leuven in 1997. In 2003, he received the Master's degree of Science in Engineering in Computer Science (Burgerlijk Ingenieur in de Computerwetenschappen). Since then, he has been working as a PhD student at the research group DTAI of the Department of Computer Science at the K.U. Leuven under the supervision of Prof. M. Denecker. Since January 2005, his research was funded by a personal grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen).