

UNIVERSITÀ DELLA CALABRIA

Dipartimento di Matematica ed Informatica

**Dottorato di Ricerca in Informatica**

XXX Ciclo

---

Settore Disciplinare INF/01 INFORMATICA

TESI DI DOTTORATO

DESIGN AND IMPLEMENTATION  
OF A MODERN ASP GROUNDNER

JESSICA ZANGARI

**Supervisor**

Prof. Francesco Calimeri

Prof.ssa Simona Perri

**Coordinatore**

Prof. Nicola Leone

---

A.A. 2016 - 2017



UNIVERSITÀ DELLA CALABRIA

Dipartimento di Matematica ed Informatica

Dottorato di Ricerca in Informatica

XXX Ciclo

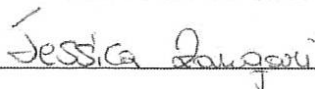
---

Settore Disciplinare INF/01 INFORMATICA

TESI DI DOTTORATO

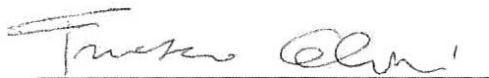
DESIGN AND IMPLEMENTATION  
OF A MODERN ASP GROUNDER

JESSICA ZANGARI

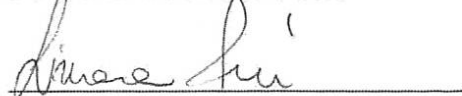
  
\_\_\_\_\_

**Supervisor**

Prof. Francesco Calimeri

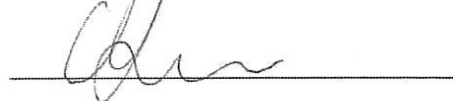
  
\_\_\_\_\_

Prof.ssa Simona Perri

  
\_\_\_\_\_

**Coordinatore**

Ch.mo Prof. Nicola Leone

  
\_\_\_\_\_

---

A.A. 2016 - 2017



# Design and Implementation of a Modern ASP Grounder

**Jessica Zangari**

*Dipartimento di Matematica,  
Università della Calabria  
87036 Rende, Italy  
email : zangari@mat.unical.it*

# Sommario

L'Answer Set Programming (ASP) è un paradigma di programmazione dichiarativa proposto nell'ambito del ragionamento non monotono e della programmazione logica tra la fine degli anni '80 e l'inizio del '90. Grazie al suo potere espressivo ed alla sua capacità di trattare conoscenza incompleta, ASP è stato ampiamente utilizzato nel campo dell'Intelligenza Artificiale e riconosciuto come un potente strumento per la rappresentazione della conoscenza e del ragionamento. D'altra parte, la sua alta espressività comporta un alto costo computazionale che richiede implementazioni affidabili e ad alte prestazioni. Nel corso degli anni, uno sforzo notevole è stato fatto per definire tecniche che garantiscano un calcolo efficiente della semantica ASP. In cambio, la disponibilità di sistemi efficienti ha reso ASP un potente strumento per lo sviluppo di applicazioni avanzate in molte aree di ricerca e in contesti industriali. Inoltre, la comunità scientifica ha significativamente contribuito all'estensione del linguaggio "base" per facilitare ulteriormente la rappresentazione della conoscenza attraverso ASP ed è stata definita una sintassi di input standard, ASP-Core-2, allo scopo di favorire l'interoperabilità tra sistemi ASP.

Sebbene siano stati proposti diversi approcci per la valutazione dei programmi di logica ASP, l'approccio canonico, adottato nei sistemi ASP tradizionali, imita la definizione della semantica "answer sets" basandosi su un modulo di istanziazione (istantiatore), che genera una teoria proposizionale semanticamente equivalente al programma in input, ed un successivo modulo (risolutore), il quale applica tecniche proposizionali per generare gli answer sets.

La prima fase, anche nota come grounding, svolge un ruolo chiave per l'uso effettivo di ASP in contesti reali, dal momento che, in generale, il programma proposizionale prodotto durante la fase di istanziazione è potenzialmente di dimensioni esponenziali rispetto al programma in input, e di conseguenza la fase successiva di risoluzione richiede, nel caso peggiore, tempi esponenziali nella dimensione dell'input. Per mitigare tale problematica, i sistemi ASP moderni impiegano procedure "intelligenti" per ottenere programmi proposizionali di dimensioni significativamente ridotte rispetto all'istanziazione teorica.

Questa tesi si concentra sulla progettazione e sull'implementazione ex-novo di un moderno ed efficiente istanziatore. A tal fine, studiamo una serie di tecniche orientate verso l'ottimizzazione del processo di istanziazione, mettendo in discussione le tecniche utilizzate dai moderni istanziatori allo scopo di migliorare lo stato dell'arte introducendo ulteriori strategie di ottimizzazione, che si prestano all'integrazione in un istanziatore generico di un sistema ASP che segue l'approccio canonico. In particolare, qui presentiamo il nuovo sistema *I-DLV* che incorpora queste tecniche, e fa leva sulla loro sinergia per eseguire un'istanziazione efficiente. Il sistema offre il pieno supporto al linguaggio stan-

dard *ASP-Core-2*, è basato su un'architettura flessibile che facilita l'integrazione di aggiornamenti linguistici e tecniche di ottimizzazione ed è dotato di meccanismi avanzati per la personalizzazione della sua procedura di istanziazione. Inoltre, il suo utilizzo è duplice: oltre ad essere un istanziatore autonomo, esso è anche un sistema di database deduttivo. Infine, insieme al risolutore *wasp*, *I-DLV* è stato integrato nella nuova versione del diffuso sistema *DLV* recentemente rilasciato.

# Abstract

Answer Set Programming (ASP) is a declarative programming paradigm proposed in the area of non-monotonic reasoning and logic programming in the late '80 and early '90. Thanks to its expressivity and capability of dealing with incomplete knowledge, ASP that became widely used in AI and recognized as a powerful tool for Knowledge Representation and Reasoning (KRR). On the other hand, its high expressivity comes at the price of a high computational cost, thus requiring reliable and high-performance implementations. Throughout the years, a significant effort has been spent in order to define techniques for an efficient computation of its semantics. In turn, the availability of efficient ASP systems made ASP a powerful tool for developing advanced applications in many research areas as well as in industrial contexts. Furthermore, a significant amount of work has been carried out in order to extend the “basic” language and ease knowledge representation tasks with ASP, and recently a standard input language, namely ASP-Core-2, has been defined, also with the aim of fostering interoperability among ASP systems.

Although different approaches for the evaluation of ASP logic programs have been proposed, the canonical approach, which is adopted in mainstream ASP systems, mimics the definition of answer set semantics by relying on a grounding module (*grounder*), that generates a propositional theory semantically equivalent to the input program, coupled with a subsequent module (*solver*) that applies propositional techniques for generating its answer sets.

The former phase, called grounding or instantiation, plays a key role for the successful deployment in real-world contexts, as in general the produced ground program is potentially of exponential size with respect to the input program, and therefore the subsequent solving step, in the worst case, takes exponential time in the size of the input. To mitigate these issues, modern grounders employ smart procedures to obtain ground programs significantly smaller than the theoretical instantiation, in general.

This thesis focuses on the ex-novo design and implementation of a new modern and efficient ASP instantiator. To this end, we study a series of techniques geared towards the optimization of the grounding process, questioning the techniques employed by modern grounders with the aim of improving them and introducing further optimization strategies, which lend themselves to the integration into a generic grounder module of a traditional ASP system following a ground & solve approach. In particular, we herein present the novel system *I-DLV* that incorporates all these techniques leveraging on their synergy to perform an efficient instantiation. The system features full support to ASP-Core-2 standard language, advanced flexibility and customizability mechanisms, and is endowed with extensible design that eases the incorporation of language up-



dates and optimization techniques. Moreover, its usage is twofold: besides being a stand-alone grounder, it is also a full-fledged deductive database engine. In addition, along with the solver *wasp* it has been integrated in the new version of the widespread ASP system *DLV* recently released.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Answer Set Programming . . . . .	1
1.2	The New Grounder <i><math>\mathcal{I}</math>-DLV</i> . . . . .	3
1.3	Main Contribution . . . . .	4
1.4	Organization of the Thesis . . . . .	4
<b>I</b>	<b>Context and Foundations</b>	<b>7</b>
<b>2</b>	<b>Answer Set Programming</b>	<b>11</b>
2.1	Syntax . . . . .	11
2.1.1	Terms . . . . .	11
2.1.2	Atoms and Literals . . . . .	12
2.1.3	Rules, Constraints, Queries and Programs . . . . .	13
2.2	Semantics . . . . .	17
2.2.1	Theoretical Instantiation . . . . .	18
2.2.2	Interpretations . . . . .	19
2.2.3	Answer Sets . . . . .	21
2.3	Advanced Constructs . . . . .	23
2.4	Safety Restriction . . . . .	25
<b>3</b>	<b>Knowledge Representation and Reasoning</b>	<b>27</b>
3.1	The GCO Technique . . . . .	27
3.2	Deductive Database Applications . . . . .	33
<b>4</b>	<b>ASP Computation And Implementations</b>	<b>35</b>
4.1	ASP Systems . . . . .	35
4.2	Ground & Solve Approach . . . . .	36
4.2.1	Grounding Phase . . . . .	36
4.2.2	Solving Phase . . . . .	37
<b>II</b>	<b>Designing an Efficient ASP Grounder</b>	<b>39</b>
<b>5</b>	<b>Grounding Process</b>	<b>43</b>
5.1	Computational Complexity . . . . .	43
5.2	Grounding an ASP Program . . . . .	45
5.2.1	Dependency Analysis . . . . .	45
5.2.2	Program Instantiation . . . . .	46

5.2.3	Rule Instantiation . . . . .	48
5.2.4	Dealing with Linguistic Extensions . . . . .	51
5.3	Optimizations . . . . .	55
<b>6</b>	<b>Efficient Retrieval of Ground Instances</b>	<b>57</b>
6.1	Deciding Data Structures . . . . .	57
6.1.1	Generalized Indices . . . . .	58
6.1.2	Single-Double Indices . . . . .	59
6.2	Deciding an Indexing Strategy . . . . .	61
6.2.1	Over-Generation Indexing Strategy . . . . .	61
6.2.2	On-Demand Indexing Strategy . . . . .	61
6.3	Balanced On-Demand Indexing Strategy . . . . .	62
<b>7</b>	<b>Body Ordering</b>	<b>65</b>
7.1	The Basic Strategy . . . . .	65
7.2	Enhancing the Basic Strategy for ASP-Core-2 . . . . .	67
7.2.1	Handling Built-in Atoms . . . . .	68
7.3	Extensions . . . . .	70
7.3.1	Indexing-driven Ordering . . . . .	70
7.3.2	Backjumping-driven Ordering . . . . .	71
7.3.3	Indexing- and Backjumping-driven Ordering . . . . .	73
<b>8</b>	<b>Decomposition Rewriting</b>	<b>75</b>
8.1	Hypergraphs and Tree Decompositions . . . . .	76
8.2	Motivations . . . . .	76
8.3	A Heuristic-driven Decomposition Approach . . . . .	78
8.4	Grounding-based Decomposition Rewriting . . . . .	80
8.4.1	Estimating the Cost of Grounding a Rule . . . . .	86
8.4.2	Estimating the Cost of Grounding a Decomposition . . . . .	87
<b>9</b>	<b>Additional Techniques for Fine-Tuning the Grounding Process</b>	<b>89</b>
9.1	Pushing Down Selections . . . . .	89
9.2	Managing Isolated Variables . . . . .	90
9.2.1	Filtering . . . . .	90
9.2.2	Rewriting . . . . .	91
9.3	Determining the Admissibility of Substitutions . . . . .	92
9.3.1	Aligning Substitutions . . . . .	93
9.3.2	Look-Ahead Technique . . . . .	94
9.4	Anticipating Strong Constraints Evaluation . . . . .	95
9.5	Rewriting Techniques for Handling Different Language Features . . . . .	97
9.5.1	Arithmetic Terms . . . . .	97
9.5.2	Functional Terms . . . . .	99
9.5.3	Aggregate Literals . . . . .	100
9.5.4	Choice Rules . . . . .	102
<b>III</b>	<b>Implementing an Efficient ASP Grounder</b>	<b>107</b>
<b>10</b>	<b>The New Grounder <i>I-DLV</i></b>	<b>111</b>
10.1	Architecture . . . . .	112

10.2 Overall Instantiation Process . . . . .	112
10.3 Customizability and Further Features . . . . .	115
10.3.1 Command-line Customization . . . . .	115
10.3.2 Inline Annotations . . . . .	118
10.3.3 Further Features . . . . .	121
<b>11 Experimental Evaluations of the Optimizations</b>	<b>123</b>
11.1 Indexing Strategies . . . . .	125
11.2 Body Ordering . . . . .	128
11.3 Decomposition Rewriting . . . . .	131
11.4 Pushing Down Selections . . . . .	134
11.5 Managing Isolated Variables . . . . .	135
11.6 Determining the Admissibility of Substitutions . . . . .	137
11.7 Anticipating Strong Constraints Evaluation . . . . .	138
11.8 Syntactic Rewriting Techniques . . . . .	140
<b>12 Experimental Evaluation of <math>\mathcal{I}</math>-DLV</b>	<b>145</b>
12.1 Comparison with the State-of-the-Art . . . . .	145
12.1.1 ASP Grounding Benchmarks . . . . .	145
12.1.2 Deductive Database Benchmarks . . . . .	150
12.2 Impact of Customizability . . . . .	151
12.3 Impact on Solvers . . . . .	152
<b>IV Related Work and Conclusion</b>	<b>157</b>
<b>13 Related Work</b>	<b>161</b>
<b>14 Conclusions</b>	<b>165</b>
<b>Bibliography</b>	<b>166</b>



# List of Figures

2.1	Dependency and Component Graphs. . . . .	17
6.1	An example of <i>generalized index</i> . . . . .	58
6.2	An example of <i>single-double index</i> . . . . .	60
8.1	Decomposing a rule of <i>Permutation Pattern Matching</i> : the associated Hypergraph and a possible tree decomposition . . . . .	84
8.2	Decomposing a rule of <i>Nomystery</i> : the associated Hypergraph . . . . .	85
8.3	Decomposing a rule of <i>Nomystery</i> : two possible tree decompositions . . . . .	85
10.1	$\mathcal{I}$ -DLV Architecture . . . . .	112
11.1	Variation of the indexing strategy via annotations . . . . .	127
11.2	Decomposition Rewriting – 2QBF Benchmarks . . . . .	133
12.1	Sixth ASP Competition Benchmarks – comparison of the old <i>DLV</i> grounder and $\mathcal{I}$ -DLV on <i>DLV</i> syntax . . . . .	147
12.2	Sixth ASP Competition Benchmarks – comparison of <i>gringo</i> and $\mathcal{I}$ -DLV on ASP-Core-2 syntax minus queries . . . . .	147
12.3	OpenRuleBench Benchmarks – grounding comparison . . . . .	149
12.4	OpenRuleBench Benchmarks – query answering comparison . . . . .	151
12.5	Instantiation: number of ground rules produced . . . . .	154
12.6	Instantiation: average body length . . . . .	155



# List of Tables

8.1	Sixth ASP Competition Benchmarks – impact of <i>lpopt</i> on solving	77
8.2	Sixth ASP Competition Benchmarks – impact of <i>lpopt</i> on grounding	78
11.1	Sixth Competition Suite: Problems Description	124
11.2	Fourth Competition Suite: Problems Description	124
11.3	Indexing – 6th Comp. Benchmarks	125
11.4	Indexing – 4th Comp. Benchmarks	126
11.5	Body Ordering – 6th Comp. Benchmarks	129
11.6	Body Ordering – 4th Comp. Benchmarks	130
11.7	Decomposition Rewriting – 6th Comp. Benchmarks	131
11.8	Decomposition Rewriting – 4th Comp. Benchmarks	132
11.9	Decomposition Rewriting – 2QBF Benchmarks – Total Grounded Instances	133
11.10	Pushing Down Selections – 6th Comp. Benchmarks	134
11.11	Pushing Down Selections – 4th Comp. Benchmarks	135
11.12	Managing Isolated Variables – 6th Comp. Benchmarks	136
11.13	Managing Isolated Variables – 4th Comp. Benchmarks	136
11.14	Admissibility of Substitutions – 6th Comp. Benchmarks	137
11.15	Admissibility of Substitutions – 4th Comp. Benchmarks	138
11.16	Anticipation of Strong Constraints Evaluation – 6th Comp. Benchmarks	139
11.17	Anticipation of Strong Constraints Evaluation – 4th Comp. Benchmarks	140
11.18	Anticipation of Strong Constraints Evaluation – Further Benchmarks	140
11.19	Choice Rewriting – 6th Comp. Benchmarks	142
11.20	Choice Rewriting – 4th Comp. Benchmarks	142
11.21	Functional Terms Rewriting – 6th Comp. Benchmarks	143
11.22	Functional Terms Rewriting – 4th Comp. Benchmarks	143
11.23	Arithmetic Terms Rewriting – 6th Comp. Benchmarks	144
11.24	Arithmetic Terms Rewriting – 4th Comp. Benchmarks	144
12.1	Sixth ASP Competition Benchmarks – number of grounded instances and grounding times in seconds	146
12.2	OpenRuleBench Benchmarks – number of grounded instances and grounding times in seconds	148
12.3	CQA Benchmarks – grounding times in seconds	149



12.4 OpenRuleBench Benchmarks – number of solved instances and query answering times in seconds . . . . .	150
12.5 Customizability – number of grounded instances and grounding times in seconds . . . . .	151
12.6 Sixth Competition Solving Benchmarks – number of solved instances and solving times in seconds . . . . .	153
12.7 Sixth Competition Solving Benchmarks – impact of SMARTDECOMPOSITION on solving . . . . .	156

# Chapter 1

## Introduction

### 1.1 Answer Set Programming

The primary intent of Computer Science is problem solving by means of machines. In this context, programming languages permit to obtain the solution(s) of a given problem, enabling the “communication” between humans and machines. Such communication might follow two radically different approaches: imperative or declarative. Imperative languages are *machine-oriented*: they require to model in a formal, machine-oriented wording how a problem should be solved. Determining efficient algorithms to solve complex (yet, tractable) problems often, requires advanced knowledge and quite good programming skills. In addition, since the conceptualization of a problem and its solution(s) are implicitly wired in the code, the imperative approach demonstrates, in general, a low *elaboration tolerance*, that is slight updates to the problem specifications, often, require a significant effort to modify the code accordingly.

An opposite approach is provided by declarative languages, which, instead, are more *human-oriented*, as they permit programmers to concentrate on problem definitions. Consequently, variations in specifications tend to have a much smaller impact on the code, since it explicitly reflects problem specifications.

Around the 1950s, John McCarthy [102] discussed how logic is particularly suited to be a full-fledged declarative programming paradigm, allowing to model problems in a natural and human-oriented fashion, and effectively represent knowledge representation and rational human reasoning. In the same years, a new computer science field was born: Artificial Intelligence or AI, and logic-based languages gained more and more importance and popularity. A breakthrough happens when Alain Colmerauer and its research group introduced Prolog [42] (from the French, *PROgramming en LOGic*), the first logic programming language. However, it emerged that the first-order logic on which Prolog is based is not capable of modelling the commonsense human reasoning, which is *non-monotonic*: we as humans, starting from some premises, may rationally regret them whenever new information become available, while in first-order logic, logical consequences cannot be invalidated since the underlying reasoning is *monotonic*.

Subsequently, new logic formalisms devoted to represent non-monotonic reasoning were introduced, such as Default Logic [115], Autoepistemic Logic [104]

and Circumscription [103]. In the late '80s and early '90s, Michael Gelfond and Vladimir Lifschitz presented the logic formalism Answer Set Programming (ASP) [73, 74] allowing to express non-monotonic reasoning in purely declarative fashion [21, 51, 54, 74, 101, 105].

ASP became widely used in AI and recognized as a powerful tool for Knowledge Representation and Reasoning. The basic construct of ASP is a rule, that has form  $Head \leftarrow Body$ , where the *Body* is a logic conjunction in which negation may appear, and *Head* can be either an atomic formula or a logic disjunction. A rule is interpreted according to common sense principles: roughly, its intuitive semantics corresponds to an implication. More precisely, the semantics of ASP is called *answer set semantics*. In ASP a problem is modeled via a logic program composed by a collection of rules. An ASP system is in charge of determining its solutions by computing its *answer sets*, which correspond one-to-one to a solution of the modeled problem. In case the input program has no answer sets, the encoded problem has no solutions.

Its roots stem from Datalog [40], a popular declarative logic programming language also based on rules. However, differently from Datalog, ASP may admit disjunctive heads and non-stratified negation. These distinguishing features make ASP suitable for modelling non-monotonic reasoning. Theoretically, the introduction of disjunction in rule heads yields to a more expressive paradigm allowing to capture the complexity class  $\Sigma_2^P = NP^{NP}$ .

Throughout the years a significant effort has been spent in order to extend the “basic” language and ease knowledge representation tasks with ASP; it has been proven to be highly versatile, offering several language constructs and reasoning modes. Recently, the community agreed on a standard input language for ASP systems: ASP-Core-2, the official language of the ASP Competition series [34, 68]. Furthermore, ASP has been successfully adopted for developing advanced applications in many research areas, ranging from Artificial Intelligence to Databases and Bioinformatics, as well as in industrial contexts [34, 94, 106, 116, 123].

The “traditional” approach to the evaluation of ASP programs relies on a grounding module (*grounder*), that generates a propositional theory semantically equivalent to the input program, coupled with a subsequent module (*solver*) that applies propositional techniques for generating its answer sets [82]. There have been other attempts deviating from this customary approach [44, 88, 89]; nonetheless, the majority of the current solutions relies on the canonical “ground & solve” strategy, as systems relying on such approach proved to be more reliable and high-performance in the widest range of scenarios.

After more than twenty years of research the theoretical properties of ASP are understood, while the linguistic extensions introduced with ASP-Core-2, their effects on the expressive power of ASP, and the ASP-based applications arising from a broader range of scenarios demand for increasingly high-performance implementations. In addition, while the solving phase has been more largely investigated in literature [96], less emphasis has been placed on the instantiation phase. Nevertheless, grounders solve a complex problem since the produced ground program is potentially of exponential size with respect to the input program [45]. Grounding, hence, may be computationally expensive and has a big impact on the performance of the whole system, as its output is the input for the subsequent solving step, that, in the worst case, takes exponential time in the size of the input [13, 14].

## 1.2 The New Grounder $\mathcal{I}$ -DLV

In this context the objective of this thesis is twofold: we aim at questioning the existing grounding techniques, improving them and introducing novel optimizations, and consequently designing and developing a new instantiator for ASP. In particular, we herein present the system  $\mathcal{I}$ -DLV, recently released [27, 28, 31, 33]; besides being a stand-alone grounder and deductive database engine, it has been integrated as the grounder module of the new version of the popular ASP system *DLV*, namely *DLV2* [2]. Among the most widely used ASP systems, *DLV* has been one of the first solid and reliable; its project dates back a few years after the first definition of answer set semantics [73, 74], and encompassed the development and the continuous enhancements of the system. It is widely used in academy, and, importantly, it is still employed in many relevant industrial applications, significantly contributing in spreading the use of ASP in real-world scenarios.

$\mathcal{I}$ -DLV has been redesigned and re-engineered from scratch. Differently from *DLV*,  $\mathcal{I}$ -DLV natively supports ASP-Core-2 and it is compatible with state-of-the-art technologies. The foremost issue experienced is the high-influence of grounding on solving; in general, simply improving the grounding times does not necessary imply improvements on the solving side, since these heavily depend on the structure and form of the produced instantiation. Thus,  $\mathcal{I}$ -DLV grounding process has been endowed with high flexibility and customizability, thanks to a lightweight modular design that eases the incorporation of optimization techniques and future updates. In particular, one of the novelty is the customizable nature of the grounder, allowing to tailor its produced instantiation to different extents, such as to better conform to solvers needs and to experiment with ASP and its applications for better adapting ASP-based solutions to real-world applications. The novel possibility of *annotating ASP code* with external directives to the grounder is a bold move in this direction, providing a new way for fine-tuning both ASP programs and systems for any specific scenario at hand [30, 32].

Despite being released recently,  $\mathcal{I}$ -DLV performance is promising and comparable with mainstream systems: in the latest ASP Competition [69]  $\mathcal{I}$ -DLV ranked both the first and second positions when combined, respectively, with an automatic solver selector [29] that inductively chooses the best solver depending on some inherent features of the instantiation produced, and with the state-of-the-art solver *clasp* [64]. Moreover,  $\mathcal{I}$ -DLV is an open-source project: its source and binaries are available from the official repository [38].

Eventually, the system is envisioned as core part of a larger project comprising the extension of  $\mathcal{I}$ -DLV towards mechanisms for interoperability with other formalisms and tools [30, 32]. The intent is to foster the usage of ASP, and in general, of logic programming in real-world and complex applications. In order to pursue in this direction, the project has as a spin-off the framework *EmbASP* [25, 61, 26] aiming at easing the integration of artificial intelligence tasks implemented via declarative logic formalisms into external applications, especially into the promising mobile platform.

### 1.3 Main Contribution

This thesis focuses on the study, formalization, design, implementation and experimentation of efficient grounding techniques in order to improve the state-of-the-art in this field.

Firstly, we started by significantly improving some already known techniques having a high impact on the overall instantiation process [27, 28].

Furthermore, based on the long-lasting experience from the ASP competition series [34, 68], given that the same computational problem can be encoded by means of many different ASP programs which are semantically equivalent, we noticed as ASP systems may perform very differently when evaluating each one of them. This issue, in a certain sense, conflicts with the declarative nature of ASP that, as in our original intent, should free the users from the burden of the computational aspects. Therefore, we defined a general algorithm, along with proper heuristic criteria, to automatically transform an ASP program into an equivalent one that can be evaluated more efficiently [31, 33].

In addition, we designed a set of fine-tuning optimizations acting to different extents on the instantiation process, with the general common aim of reducing the search space and improving overall performance [27, 28].

The presented grounding strategies lend themselves to the integration in a generic grounder module of a traditional ASP system following a ground & solve approach. In particular, we herein present the novel system *I-DLV* that incorporates all of them leveraging on their synergy to perform an efficient instantiation. In addition, *I-DLV* is devoted to efficient query answering, and can behave as a full-fledged deductive database system. Moreover, *I-DLV* constitutes the new grounding module of the ASP system *DLV* [2].

The input language of *I-DLV* has been enriched with the support for special comments expressing meta-data information that *I-DLV* can interpret in order to fine-tune its grounding process. Following a widespread term in programming, we named these constructs *annotations*, as they do not change the semantics of input programs, but their impact might be observed just on the performance. In particular, supported annotations belong to the following categories: *grounding annotations* allowing for a fine-grained customization on the grounding process, and *solving annotations* that have been integrated into *DLV2*, and are geared to the customization of the whole computational process [30, 32, 2].

### 1.4 Organization of the Thesis

The present work is divided into four parts:

- The first part presents Answer Set Programming, formalizing its syntax and semantics, providing some example of Knowledge Representation and Reasoning via ASP and discussing the history of the development of ASP systems.
- The second part focuses on the design of techniques for an efficient instantiation. More in detail, firstly we illustrate the typical instantiation work flow, and then we formalize the studied techniques highlighting their intended behaviour and effects.

- The third part is devoted to *I-DLV* and its distinguishing features. Furthermore, we discuss two sets of experiments. Firstly, we assess the performance of the studied techniques and their variants; then, we evaluate *I-DLV* overall performance as both ASP grounder and deductive database engine; eventually, we analyze the quality of its instantiation and its impact on solvers.
- Eventually, the fourth part compares related work and draws conclusions, outlining future and on-going work.



Part I

Context and Foundations





In this part we introduce Answer Set Programming, presenting its theoretical properties and characteristic features. The part is structured as reported below:

- Chapter 2 provides a detailed description of the syntax and semantics of Answer Set Programming complying with **ASP-Core-2**, the recent standard input language.
- Chapter 3 illustrates the powerful modelling capabilities of Answer Set Programming, describing its usage for Knowledge Representation and Reasoning.
- Chapter 4 depicts the history of the development of ASP systems, focusing on the computational process carried out by the majority of current available solutions.



## Chapter 2

# Answer Set Programming

In this chapter we introduce Answer Set Programming (ASP). The core of the language consists in Disjunctive Datalog with nonmonotonic negation under the stable model semantics. Nevertheless, over the years a significant amount of work has been carried out by the scientific community in order to enrich the basic language, and several extensions have been studied and proposed. Recently, the community agreed on a standard input language for ASP systems: ASP-Core-2 [24], the official language of the ASP Competition series [34, 68, 69].

The chapter is structured as follows. Sections 2.1 and 2.2 present a formal definition of the syntax and the semantics of ASP. Section 2.3 describes some syntactic shortcuts. Section 2.4 introduces the concept of *safety*. The herein reported definitions are compliant with ASP-Core-2 v.2.03c (the latest version at the time of writing).

### 2.1 Syntax

Let  $\mathcal{I}$  be a set of *identifiers*. An identifier is a not empty string starting with some lowercase letter and containing only alphanumeric symbols and the symbol “\_” (underscore).

**Example 2.1.1.** Examples of identifiers are:  $a$ ,  $a1\_B$ ,  $a\_ID$ ,  $vertex$

#### 2.1.1 Terms

A term is either a *constant*, a *variable*, an *arithmetic term* or a *functional term*. In particular, constants and variables can be considered as “basic terms”, while arithmetic and functional terms are defined inductively as combinations of terms.

**Definition 2.1.1** (Constant Term). A *constant* is either a *symbolic constant*, if it is an identifier, a *string constant*, if it is a quoted string, or an *integer*.

**Definition 2.1.2** (Variable Term). A *variable* is a not empty string starting with some uppercase letter and containing only alphanumeric symbols and the symbol “\_” (underscore).

Furthermore a special variable, namely *anonymous variable*, is represented by the symbol “\_” (underscore). This syntactic shortcut is intended to indicate a *fresh* variable, that does not appear elsewhere in the context in which it is located.

**Definition 2.1.3** (Arithmetic Term). An *arithmetic term* has form  $-(t)$  or  $(t_1 \diamond t_2)$  for terms  $t_1$  and  $t_2$  with  $\diamond \in \{+, -, *, /\}$ . Parentheses can optionally be omitted, and standard operator precedences apply.

**Definition 2.1.4** (Functional Term). A *functional term* has form  $f(t_1, \dots, t_n)$ , where  $f$  is an identifier, known as *functor*,  $t_1, \dots, t_n$  are terms and  $n > 0$ .

**Example 2.1.2.** Examples of terms are:

- Constants:  $a$ ,  $x$ , “ $http://google.com$ ”,  $0$ ,  $123$
- Variables:  $X$ ,  $X_{134}$ ,  $X2$ ,  $Color$
- Arithmetic terms:  $-X$ ,  $X + Y$ ,  $2 * (-5)$ ,  $X + ab$ ,  $X/3$
- Functional terms:  $f(X)$ ,  $father(aristotle)$ ,  $g(2 * 5, “abc”)$

A term is *ground* (i.e. variable-free) if it does not contain any variable. For, instance in Example 2.1.2 all the constants, the arithmetic term  $2 * (-5)$  and the functional terms  $father(aristotle)$  and  $g(2 * 5, “abc”)$  are ground.

## 2.1.2 Atoms and Literals

**Definition 2.1.5** (Predicate). Given an identifier  $p$  and an integer  $n$  with  $n \geq 0$ , the expression  $p/n$  represents a *predicate*.  $p$  is said *predicate symbol* and  $n$  represents the associated arity.

**Example 2.1.3.** Examples of predicates are:  $a/2$ ,  $p/3$ ,  $predicate\_3/1$ ,  $true/0$ .

In the following, when no ambiguities arise we denote simply as  $p$  a predicate  $p/n$ .

**Definition 2.1.6** (Predicate Atom). A *predicate atom* has form  $p(t_1, \dots, t_n)$ , where  $n \geq 0$ ,  $p/n$  is a predicate with predicate name  $p$  and arity  $n$  and  $t_1, \dots, t_n$  are terms; if  $n = 0$ , parenthesis are omitted and the notation  $p$  is used.

**Definition 2.1.7** (Classical Atom). A *classical atom* is either  $-a$  or  $a$  where  $a$  is a predicate atom and  $-$  denotes the *strong negation* symbol.

**Definition 2.1.8** (Built-in Atom). A *built-in atom* has form  $t_1 \triangleright t_2$  where  $t_1, t_2$  are terms and  $\triangleright \in \{<, <=, =, <>, !=, >, >=\}$ .

**Definition 2.1.9** (Naf-Literal). A *naf-literal* can either be a built-in atom or have form  $a$  or  $not\ a$  where  $a$  is a classical atom, and *not* is the *negation as failure* symbol.

**Example 2.1.4.** Some examples are shown below.

- Predicate Atoms:  $edge(X, Y)$ ,  $atom(f(a, b), c)$ ,  $true$
- Classical Atoms:  $edge(X, Y)$ ,  $atom(f(a, b), c)$ ,  $true$ ,  $-true$

- Built-in Atoms:  $father(aristotle) = nicomachus$ ,  $X! = Y$ ,  $X * 2 = Y$
- Naf-Literals:  $father(aristotle) = nicomachus$ ,  $X! = Y$ ,  $X * 2 = Y$ ,  
 $edge(X, Y)$ ,  $-atom(f(a, b), c)$ ,  $true$ ,  $-true$ ,  $not -true$ ,  $not true$

In addition to the type of atoms above illustrated, *aggregate atoms* have been introduced to permit aggregation operations on multi-sets of terms by means of concise expressions.

**Definition 2.1.10** (Aggregate Element). An *aggregate element* is composed as:  $t_1, \dots, t_m : l_1, \dots, l_n$ , where  $t_1, \dots, t_m$  are terms  $l_1, \dots, l_n$  are naf-literals for  $n \geq 0$ ,  $m \geq 0$ .

**Definition 2.1.11** (Aggregate Atom). An *aggregate atom* has form:

$$af\{e_1, \dots, e_n\} \triangleright t$$

where:

- $af \in \{\#count, \#sum, \#max, \#min\}$
- $e_1, \dots, e_n$  are aggregate elements for  $n \geq 0$
- $\triangleright \in \{<, <=, =, <>, !=, >, >=\}$
- $t$  is a term

**Definition 2.1.12** (Aggregate Literal). An *aggregate literal* is either  $a$  or  $not a$  where  $a$  is an aggregate atom.

**Example 2.1.5.** For instance, the following are aggregate literals:  $not \#max\{X, Y : age(X, Y)\} < 20$ ,  $\#sum\{X, Y : age(X, Y)\} = s(S)$ ,  $\#count\{1 : a(1)\} > 3$ . Moreover, the latter two literals are also aggregate atoms.

An atom is *ground* if it does not contain any variable. A literal is ground if its atom is ground. In Examples 2.1.4 and 2.1.5  $father(aristotle) = nicomachus$ ,  $-atom(f(a, b), c)$ ,  $true$ ,  $-true$ ,  $not -true$ ,  $not true$ ,  $\#count\{1 : a(1)\} > 3$  are ground.

In the following we will refer to classical, built-in and aggregate atoms as *atoms*. Similarly, we will indicate naf and aggregate literals as *literals*. A literal is *negative* if the *not* symbol is present, otherwise it is *positive*.

### 2.1.3 Rules, Constraints, Queries and Programs

After defining the basic constructs, we now describe the main components of an ASP logic program.

**Definition 2.1.13** (Rule). A *rule*  $r$  has the following form:

$$a_1 \mid \dots \mid a_n :- b_1, \dots, b_m.$$

where:

- $a_1, \dots, a_n$  are classical atoms
- $b_1, \dots, b_m$  are literals

–  $n \geq 0, m \geq 0$

The *disjunction*  $a_1 \mid \dots \mid a_n$  is the *head* of  $r$ , while the *conjunction*  $b_1, \dots, b_m$  is the *body* of  $r$ . We denote by  $H(r)$  the set  $\{a_1, \dots, a_n\}$  of the head atoms, and by  $B(r)$  the set  $\{b_1, \dots, b_m\}$  of the body literals.  $B^+(r)$  denotes the set of literals occurring positively in  $B(r)$ ; while  $B^-(r)$  is the set of negative literals in  $B(r)$ . A rule having precisely one head literal (i.e.  $n = 1$ ) is said to be a *normal rule*; if  $n > 1$  the rule is *disjunctive*.

**Example 2.1.6.** Examples of rules are:

$hasUmbrella(X) \mid doesNotHaveUmbrella(X) :- person(X).$

$isRaining \mid -isRaining :- cloudyWeather.$

**Definition 2.1.14** (Fact). A rule  $r$  is a *fact* with  $B(r) = \emptyset$ ,  $|H(r)| = 1$  and  $H(r) = \{a\}$  where  $a$  is a classical ground atom.

**Example 2.1.7.** Examples of facts are:

$cloudyWeather. -isRaining. person(alice). person(bob).$

the  $:-$  sign is usually omitted.

In the following, as it is common, we will adopt the notation reported next to represent in a compact way a set of facts:  $p(m_{1_1}..m_{1_2}, \dots, m_{n_1}..m_{n_2})$ . where  $p/n$  is a predicate of arity  $n$ , and  $m_{i_j}$  with  $i \in \{1, \dots, n\}$  and  $j \in \{1, 2\}$  are terms. For instance,  $a(1..2, f(3..4))$ . defines the facts:  $a(1, f(3)). a(2, f(3)). a(1, f(4)). a(2, f(4))$ .

A predicate  $p/n$  is referred to as an *EDB* predicate if, for each rule  $r$  in which  $p/n$  appears in  $H(r)$ ,  $r$  is a fact; all others predicates are referred to as *IDB* predicates. The set of facts in which *EDB* predicates occur, is called *Extensional Database (EDB)*, the set of all other rules is the *Intensional Database (IDB)*.

**Definition 2.1.15** (Strong (or Integrity) Constraint). A *strong constraint*  $s$  is a rule with  $|H(s)| = \emptyset$ .

**Definition 2.1.16** (Weak Constraint). A *weak constraint*  $c$  is a special type of rule, having form:

$$:\sim b_1, \dots, b_m. [w@l, t_1, \dots, t_n]$$

where:

- $n \geq 0, m \geq 0$
- $b_1, \dots, b_m$  are literals
- $w, l, t_1, \dots, t_n$  are terms;  $w$  and  $l$  are referred to, respectively, as *weight* and *level* for  $c$ ; if  $l = 0$ , the expression  $@0$  can be omitted.

Basically, a weak constraint is like a strong one, where the implication symbol  $:-$  is replaced by  $:\sim$ . The informal meaning of a weak constraint  $:\sim B$ . is “try to falsify  $B$ ,” or “ $B$  should preferably be false”.

For a weak constraint  $c$  we will indicate as *weak specification*, denoted  $W(c)$ , the part within the square brackets.

**Example 2.1.8.** Examples of constraints are:

$$\begin{aligned} & :- \text{isRaining}, \text{not isWetStreet}. \\ & :\sim \text{isRaining}, \text{person}(X), \text{not hasUmbrella}(X). [1] \end{aligned}$$

A rule  $r$  is *ground* if all the atoms in  $H(r)$  are ground and all the literals in  $B(r)$  are ground. A strong constraint  $s$  is ground if all the literals in  $B(s)$  are ground. A weak constraint  $c$  is ground if all the literals in  $B(c)$  are ground, and all the terms in its weak specification  $W(c)$  are ground. In Example 2.1.6 the rule  $\text{isRaining} \mid \text{-isRaining} :- \text{cloudyWeather}$ . is ground, as well as the two constraints in Example 2.1.8.

For a literal  $l$ , let  $\text{var}(l)$  be the set of variables appearing in  $l$ ; if  $l$  is ground  $\text{var}(l) = \emptyset$ . For a conjunction of literals  $C$ ,  $\text{var}(C)$  denotes the set of variables occurring in the literals in  $C$ ; similarly, for a disjunction of atoms  $D$ ,  $\text{var}(D)$  denotes the set of variables in the atoms in  $D$ . Inductively, for a rule  $r$ ,  $\text{var}(r) = \text{var}(H(r)) \cup \text{var}(B(r))$ ; for a strong constraint  $s$ ,  $\text{var}(r) = \text{var}(B(s))$ ; for a weak constraint  $c$ ,  $\text{var}(c) = \text{var}(B(c)) \cup \text{var}(W(c))$ .

Given a rule or weak constraint  $r$ , a variable  $X$  is *global* if it appears outside of an aggregate element in  $r$ ; we denote as  $\text{var}_g(r)$  the set of global variables in  $r$ . Given an aggregate element  $e$  in a rule or weak constraint  $r$ ,  $\text{var}_l(e) = \text{var}(e) \setminus \text{var}(r)$  denotes the set of *local* variables of  $e$ , i.e. the set of variables appearing only in  $e$ , while the set of *global* variables of  $e$  contains variables appearing in both  $r$  and  $e$ , i.e.  $\text{var}_g(e) = \text{var}(r) \cap \text{var}(e)$ . Suppose that  $r$  contains the aggregate elements  $E = \{e_1, \dots, e_n\}$ , then  $\text{var}(r)$  can be also defined as  $\text{var}(r) = \text{var}_g(r) \cup \bigcup_{i=1}^n \{\text{var}_l(e_i) \mid e_i \in E\}$ ; if  $n = 0$  and thus  $E = \emptyset$ , i.e.  $r$  does not contain any aggregate element, then  $\text{var}(r) = \text{var}_g(r)$ .

**Example 2.1.9.** As an example, given the following rule  $r$ :

$$a(X) :- b(X), \text{not } c(X), \#sum\{Y : d(X, Y); Z : f(Z)\}.$$

we can observe that  $\text{var}(r) = \{X, Y, Z\}$ ,  $\text{var}_g(r) = \{X\}$ ,  $\text{var}_l(Y : d(X, Y)) = \{Y\}$ ,  $\text{var}_l(Z : f(Z)) = \{Z\}$ .

In the following, we will denote rules and constraints (weak or strong) simply as *rules*.

**Definition 2.1.17** (Query). A *query* has form:  $a?$  where  $a$  is a classical atom.

**Example 2.1.10.** Examples of queries are:  $\text{-isRaining?}$ ,  $\text{hasUmbrella}(X)?$ .

A query is *ground* if its atom is ground. In Example 2.1.10  $\text{-isRaining?}$  is ground.

**Definition 2.1.18** (Program). A *program* is a finite set of rules, possibly accompanied by a single query.

A program is *ground* if all its rule, constraints, and the possible query are ground. A program containing disjunctive rules is *disjunctive*, otherwise it is *non disjunctive*.



**Example 2.1.11.** The following constitutes a *disjunctive* program:

```

hasUmbrella(X) | doesNotHaveUmbrella(X) :- person(X).
isRaining | -isRaining :- cloudyWeather.
:- isRaining, not isWetStreet.
:~ isRaining, person(X), not hasUmbrella(X). [1]
cloudyWeather. -isRaining.
person(alice). person(bob).
hasUmbrella(bob)?

```

Programs are also classified according to their structural properties, such as dependencies among predicates.

**Definition 2.1.19.** (Dependency Graph) The *Dependency Graph* of  $P$  is a directed graph  $G_P = \langle N, E \rangle$ , where  $N$  is the set of IDB predicates of  $P$ , and  $E$  contains an edge  $(p/n, q/m)$  if there is a rule  $r$  in  $P$  such that  $q/m$  occurs in the head of  $r$  and  $p/n$  occurs in a classical atom of  $B(r)$  or in a classical atom within an aggregate literal of  $B(r)$ .

The graph  $G_P$  induces a partition of  $P$  into subprograms (also called *modules*). For each strongly connected component (SCC)<sup>1</sup>  $C$  of  $G_P$  (a set of predicates), the set of rules defining the predicates in  $C$  is called *module* of  $C$  and is denoted by  $M_C$ . A rule  $r$  occurring in a module  $M_C$  (i.e. containing in its head some predicate  $q/m \in C$ ) is said to be *recursive* if there is a predicate  $p/n \in C$  in the positive body of  $r$ ; otherwise,  $r$  is said to be an *exit rule*. Moreover, we say that  $p/n$  and  $q/m$  are *recursive* predicates. A program containing at least a recursive rule is said *recursive*.

Currently, ASP-Core-2 forbids the usage of recursion inside aggregates, requiring that recursive predicates do not appear within aggregates; nonetheless, the problem has been investigated in literature and several semantics have been proposed for them [83, 72, 47, 107, 109, 108, 118, 119, 59, 58]. However, while previous semantic definitions typically agree in the non-recursive case, the picture is not so clear for recursion. Complying with ASP-Core-2, in this thesis we restrict our attention to non-recursive aggregates.

**Definition 2.1.20.** (Component Graph) The *Component Graph* of a program  $P$  is a directed labelled graph  $G_P^c = \langle N, E, lab \rangle$ , where  $N$  is the set of strongly connected components of  $G_P$ , and  $E$  contains:

- an edge  $(B, A)$  with  $lab((B, A)) = “+”$ , if there is a rule  $r$  in  $P$  such that  $a \in A$  occurs in the head of  $r$  and  $b \in B$  occurs in a classical atom of  $B(r)$  or in a classical atom within an aggregate literal of  $B(r)$ ;
- an edge  $(B, A)$ , with  $lab((B, A)) = “-”$ , if there is a rule  $r$  in  $P$  such that  $a \in A$  occurs in the head of  $r$  and  $b \in B$  occurs in a negative naf-literal of  $B(r)$  or in a negative naf-literal within an aggregate literal of  $B(r)$ , and there is no edge  $e'$  in  $E$ , with  $lab(e') = “+”$ .

A predicate  $p/n$  is *stratified* [5] with respect to negation if it does not occur in cycles in  $G_P^c$  involving negative dependencies (i.e. edges labelled with “-”),

<sup>1</sup>We briefly recall that a strongly connected component of a directed graph is a maximal subset of the vertices, such that every vertex is reachable from every other vertex.

otherwise  $p/n$  is said *unstratified*. Consequently, a program  $P$  is *stratified* with respect to negation if every predicate appearing in it is stratified, or equivalently, if no cycles in  $G_P^c$  involve negative dependencies, otherwise  $P$  is said *unstratified*. A predicate is *solved* if: (i)  $p/n$  is defined solely by non-disjunctive rules (i.e. all rules with  $p/n$  in the head are non-disjunctive), and (ii)  $q$  does not depend (even transitively) on any unstratified predicate or disjunctive predicate (i.e. a predicate defined by a disjunctive rule).

**Example 2.1.12.** As an example let us consider the following program  $P_1$ :

$$\begin{aligned} r_1 &: a(X) :- b(X), \text{not } c(X). \\ r_2 &: b(Y) :- a(Y), Y = X + 1, f(X). \\ r_3 &: c(X) :- d(X), \text{not } a(X). \\ r_4 &: d(X) :- f(X), \text{not } g(X). \end{aligned}$$

The dependency and component graphs are illustrated in Figure 2.1. In the dependency graph  $G_{P_1}$ , there are three components: (1) a first component  $C_1$  is formed by predicate  $a/1$  and  $b/1$ , (2) the predicate  $c/1$  forms another component  $C_2$ , and (3) a third component  $C_3$  is composed by the predicate  $d/1$ . Hence,  $M_{C_1} = \{r_1, r_2\}$ ,  $M_{C_2} = \{r_3\}$ ,  $M_{C_3} = \{r_4\}$ . Moreover, the rules  $r_1$  and  $r_2$  are recursive, thus  $P_1$  is recursive. Finally, in the component graph  $G_{P_1}^c$  there is a cycle involving components  $\{a/1, b/1\}$  and  $\{c/1\}$ , and so  $P_1$  is *unstratified* under negation.

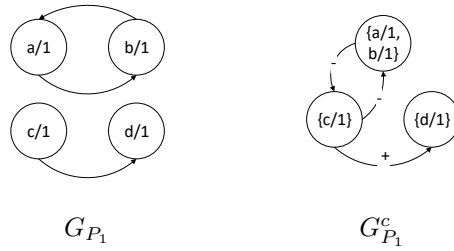


Figure 2.1: Dependency and Component Graphs.

## 2.2 Semantics

The semantics of an ASP program is given by the set of its *answer sets*. Each *answer set* corresponds to a solution for the encoded problem. Notably, ASP is a fully declarative paradigm: the order in which the program is composed by rules, constraints and query, as well as the order of literals and atoms in the rules bodies and heads, have no effect on the semantics.

Furthermore, answer sets are defined for ground programs only. However, for every non-ground program, a semantically equivalent ground program can be defined. The process of producing such a ground program is referred to as *instantiation* or *grounding*. Essentially, for each rule of a non-ground program its variables are considered universally quantified and ranging over the set of ground terms defined by the program Herbrand Universe. Intuitively, variables are just an abstraction to represent ground terms.

In the following, we formalize the semantics of ASP-Core-2, obtained by inheriting the semantics proposed in [74] as a generalization of *stable models* semantics [73], extended to aggregates according to [56, 57].

### 2.2.1 Theoretical Instantiation

Let  $P$  an ASP program.

**Definition 2.2.1** (Herbrand Universe). The *Universe of Herbrand* of  $P$ ,  $U_P$ , is the set of all integers and ground terms constructible from constants and functors appearing in  $P$ . In case no constant appears in  $P$  an arbitrary constant  $c$  is added to  $U_P$ .

**Definition 2.2.2** (Herbrand Base). The *Base of Herbrand* of  $P$ ,  $B_P$ , is the set of all ground classical atoms obtainable by combining predicate names appearing in  $P$  with terms from  $U_P$  as arguments.

**Example 2.2.1.** As running example, let us consider the program  $P_1$ :

$$\begin{aligned} &b(1). b(2). c(1). \\ &a(X) :- b(X), \text{ not } c(X * 1). \\ &d(Y) :- \#count\{X : a(X)\} = Y. \end{aligned}$$

then  $U_{P_1} = \{1, 2\}$  and  $B_{P_1} = \{a(1), a(2), b(1), b(2), c(1), c(2), d(1), d(2)\}$ .

**Definition 2.2.3** (Substitution). Given a Herbrand Universe  $U_P$  of a program  $P$  and a set of variables  $V$ , a *substitution* is total function  $\sigma : V \mapsto U_P$  that maps each variable in  $V$  to an element in  $U_P$ . For some object  $O$  occurring in  $P$  (term, atom, aggregate atom, literal, rule, weak constraint, query, etc.), we denote by  $O\sigma$  the object obtained by replacing each occurrence of a variable  $v \in \text{var}(O)$  by  $\sigma(v)$  in  $O$ .  $\sigma$  is well-formed if the arithmetic evaluation, performed in the standard way, of each arithmetic sub-term  $t$  in  $O\sigma$  is well-defined.

In the following, we will denote a substitution  $\sigma$  also as the set  $\{X = c \mid \sigma(X) = c\}$ .

**Definition 2.2.4** (Global and Local Substitutions). Given a rule or weak constraint  $r$  in  $P$  a substitution is *global* if it involves variables in  $\text{var}_g(r)$ ; for an aggregate element  $e$  in  $r$ , a substitution is *local* if it involves variables in  $\text{var}_l(e)$ .

We remark that for terms, classical atoms, naf-literals and queries a substitution is implicitly global, due to the absence of aggregate elements. In the following for the above mentioned constructs we will indicate substitutions for them as *global* substitutions.

**Example 2.2.2.** Consider the rule  $r_1$  from  $P_1$  and the (global) substitution  $\sigma_1 = \{X = 1\}$ , then  $r_1\sigma_1 = a(1) :- b(1), \text{ not } c(1 * 1)$ . Note that  $\sigma_1$  is well-formed, while for instance, supposing that  $U_P$  contained also the symbolic constant  $abc$ , then a substitution  $\sigma_2 = \{X = abc\}$  would not be well-formed.

Now, consider the rule  $r_2$  from  $P_1$  and the global substitution  $\sigma_3 = \{Y = 1\}$ , then  $r_2\sigma_3 = d(1) :- \#count\{X : a(X)\} = 1$ . If instead, we consider the local substitution  $\sigma_4 = \{X = 1\}$ , then  $r_2\sigma_4 = d(Y) :- \#count\{1 : a(1)\} = Y$ .

The instantiation of an aggregate element  $e$  is obtained by considering well-formed local substitutions for  $e$ ; formally, the instantiation of  $e$  consists in the following set of ground aggregate elements:

$$\text{inst}(e) = \{e\sigma \mid \sigma \text{ is a well-formed local substitution for } e\}$$

Inductively, the instantiation of a series of aggregate elements  $\{e_1, \dots, e_n\}$  is provided by the set of aggregate elements reported below:

$$\text{inst}(\{e_1, \dots, e_n\}) = \bigcup_{i=1}^n \{e_i\sigma \mid \sigma \text{ is a well-formed local substitution for } e_i\}$$

A *ground instance* of a term, classical atom, naf-literal, a rule, weak constraint, or query  $o$  is obtained in two steps: (i), a well-formed global substitution  $\sigma$  for  $o$  is applied to  $o$ ; (ii), for every aggregate atom  $af\{e_1, \dots, e_n\} \triangleright t$  in  $r\sigma$  its aggregate elements  $\{e_1, \dots, e_n\}$  are replaced by  $\text{inst}(\{e_1, \dots, e_n\})$ .

**Example 2.2.3.** Consider the aggregate element  $e = \{X : a(X)\}$  of rule  $r_2$  from  $P_1$ , then the instantiation  $\text{inst}(e)$  of  $e$  consists in  $\text{inst}(e) = \{1 : a(1); 2 : a(2)\}$ .

At this point, a ground instance of  $r_2$  is obtained by applying the substitution  $\sigma_3 = \{Y = 1\}$ , and replacing  $e$  with  $\text{inst}(e)$ :  $d(1) :- \#count\{1 : a(1); 2 : a(2)\} = 1$ .

The arithmetic evaluation of a ground instance  $g$  of some term, classical atom, naf-literal, rule, weak constraint or query is obtained by replacing any maximal arithmetic subterm appearing in  $g$  by its integer value, which is calculated in the standard way.

The ground instantiation of a program  $P$ , denoted by  $\text{grnd}(P)$ , is the set of arithmetically evaluated ground instances of rules, strong and weak constraints in  $P$ .

**Example 2.2.4.** Eventually, let us consider  $P_1$ ,  $\text{grnd}(P_1)$  consists in:

$$\begin{aligned} &b(1). b(2). c(1). \\ &a(1) :- b(1), \text{ not } c(1). \\ &a(2) :- b(2), \text{ not } c(2). \\ &d(1) :- \#count\{1 : a(1); 2 : a(2)\} = 1. \\ &d(2) :- \#count\{1 : a(1); 2 : a(2)\} = 2. \end{aligned}$$

Note that the substitution  $\{X = 1, X = 2\}$  has been applied to  $r_1$ , and the arithmetic terms  $(1 * 1)$  and  $(2 * 1)$  have been evaluated respectively to 1 and 2.

*Remark 2.2.1.* The instantiation of a program is idempotent: for each program  $P$ ,  $\text{ground}(P) = \text{ground}(\text{ground}(P))$ .

## 2.2.2 Interpretations

Once that a ground program is obtained, the truth values of atoms, literals, rules, constraints etc. is properly defined according to interpretations.

**Definition 2.2.5** (Herbrand Interpretation). A (*Herbrand*) *interpretation*  $I$  for  $P$  is a *consistent* subset of  $B_P$ ; to this end, for each predicate atom  $a \in B_P$ ,  $\{a, -a\} \not\subseteq I$  must hold.

Literals can be either true or false w.r.t. an interpretation. To illustrate how their truth values are determined, as a preliminary step, we need to define a proper total order  $\preceq$  on terms in  $U_P$ . Several orderings may be defined, in ASP-Core-2 has been adopted the one reported next.

Let  $t$  and  $u$  be two arithmetically evaluated ground terms, then:

- $t \preceq u$  for integers  $t$  and  $u$  if  $t \leq u$ ,
- $t \preceq u$  if  $t$  is an integer and  $u$  is a symbolic constant,
- $t \preceq u$  for symbolic constants  $t$  and  $u$  with  $t$  lexicographically smaller or equal to  $u$ ,
- $t \preceq u$  if  $t$  is a symbolic constant and  $u$  is a string constant,
- $t \preceq u$  for string constants  $t$  and  $u$  with  $t$  lexicographically smaller or equal to  $u$ ,
- $t \preceq u$  if  $t$  is a string constant and  $u$  is a functional term,
- $t \preceq u$  for functional terms  $t = f(t_1, \dots, t_n)$  and  $u = g(u_1, \dots, u_n)$  if either:
  - $m < n$  or,
  - $m = n$  and  $g \not\prec f$  ( $f$  is lexicographically smaller than  $g$ ) or,
  - $m = n$ ,  $f \preceq g$  and, for any  $1 \leq j \leq m$  such that  $t_j \not\prec u_j$ , there is some  $1 \leq i < j$  such that  $t_i \not\prec u_i$  (i.e. the tuple of terms of  $t$  is smaller than or equal to the arguments of  $u$ ).

At this point, we are ready to properly define literals satisfaction. Let  $I \subseteq B_P$  be a consistent interpretation for  $P$ .

The satisfaction of built-in atoms can be easily defined according to the total order  $\preceq$ , in the intuitive way, as they represent comparisons among terms. A ground classical atom  $a \in B_P$  is *true* w.r.t.  $I$  if  $a \in I$ . A positive ground naf-literal  $a$  is *true* w.r.t.  $I$  if  $a$  is a classical or built-in atom that is true w.r.t.  $I$ ; otherwise,  $a$  is false w.r.t.  $I$ . A negative ground naf-literal *not*  $a$  is true (or false) w.r.t.  $I$  if  $a$  is false (or true) w.r.t.  $I$ .

Given a ground aggregate atom  $af\{e_1, \dots, e_n\} \triangleright t$ , in order to correctly evaluate its semantics according to its aggregate function, the expression  $af\{e_1, \dots, e_n\}$  has to be mapped to a term, say  $u$ . Indeed, aggregate functions can be seen as mappings from set of terms to a term. Let  $T$  be the set of terms in  $\{e_1, \dots, e_n\}$ , then:

- if  $ag = \#count$ , then  $u = |T|$ ;
- if  $ag = \#sum$ , then  $u = \sum_{t_i \in T} t_i$  is an integer;
- if  $ag = \#max$ , then  $u = \max\{t_i | t_i \in T\}$
- if  $ag = \#min$ , then  $u = \min\{t_i | t_i \in T\}$

Essentially,  $\#count$  depends on the cardinality of the set of terms  $T$ ,  $\#sum$  is evaluated as the sum of the integers in  $T$ , while  $\#max$  and  $\#min$  functions strictly rely on the total order  $\preceq$  on terms in  $U_P$ . In case  $T = \emptyset$ , the following

convention is adopted:  $\#max\{\emptyset\} \preceq u$  and  $u \preceq \#min\{\emptyset\}$  for every term  $u \in U_P$ .

Fixed an interpretation some aggregate elements may not contribute to the semantics of an aggregate atom. Intuitively, an interpretation can filter out some aggregate elements according to their truth values w.r.t. the interpretation itself. More formally, the interpretation  $I$  maps a collection  $E$  of aggregate elements to the following set of tuples of ground terms:

$$eval(E, I) = \{(t_1, \dots, t_n) \mid \{t_1, \dots, t_n : l_1, \dots, l_m\} \text{ occurs in } E \text{ and} \\ \{l_1, \dots, l_m\} \text{ are true w.r.t. } I\}$$

Let  $a = af\{e_1, \dots, e_n\} \triangleright t$  be an aggregate atom,  $a$  is true (or false) w.r.t.  $I$  if  $af\{eval(e_1, \dots, e_n, I)\} \triangleright t$ . A positive aggregate literal  $a$  is true (or false) w.r.t.  $I$  if  $a$  is an aggregate atom that is true (or false) w.r.t.  $I$ . A negative aggregate literal  $not\ a$  is true (or false) w.r.t.  $I$  if  $a$  is false (or true) w.r.t.  $I$ .

Let  $r$  be a ground rule in  $grnd(P)$ . The head of  $r$  is *true* w.r.t.  $I$  if  $H(r) \cap I \neq \emptyset$ . The body of  $r$  is *true* w.r.t.  $I$  if all body literals of  $r$  are true w.r.t.  $I$  (i.e.  $B^+(r) \subseteq I$  and  $B^-(r) \cap I = \emptyset$ ) and is *false* w.r.t.  $I$  otherwise. The rule  $r$  is *satisfied* (or *true*) w.r.t.  $I$  if its head is true w.r.t.  $I$  or its body is false w.r.t.  $I$ .

**Example 2.2.5.** Let

$$I_1 = \{b(1), b(2), c(1), a(1), d(1), d(2)\}$$

be an interpretation for  $grnd(P_1)$ , then:

$b(1). b(2). c(1).$	are satisfied w.r.t. $I_1$ .
$a(1) :- b(1), not\ c(1).$	is not satisfied because <i>not</i> $c(1)$ is false w.r.t. $I_1$ .
$a(2) :- b(2), not\ c(2).$	is not satisfied because $a(2)$ is false w.r.t. $I_1$ .
$d(1) :- \#count\{1 : a(1); 2 : a(2)\} = 1.$	is satisfied w.r.t. $I_1$ since $eval(\{1 : a(1); 2 : a(2)\}, I_1) = \{1 : a(1)\}$ , and $\#count\{1 : a(1)\} = 1$ .
$d(2) :- \#count\{1 : a(1); 2 : a(2)\} = 2$	is not satisfied w.r.t. $I_1$ because of the evaluation reported above.

### 2.2.3 Answer Sets

**Definition 2.2.6** (Model). A *model* for  $P$  is an interpretation  $M$  for  $P$  such that every rule  $r \in grnd(P)$  is true w.r.t.  $M$ .

**Definition 2.2.7** (Minimal Model). A model  $M$  for  $P$  is *minimal* if no model  $N$  for  $P$  exists such that  $N$  is a proper subset of  $M$ . The set of all minimal models for  $P$  is denoted by  $MM(P)$ .

**Example 2.2.6.** The interpretation  $I_1$  is not a model for  $P_1$ , while the interpretation

$$I_2 = \{b(1), b(2), c(1), a(2), d(1)\}$$

is a model for  $P_1$ :

$b(1). b(2). c(1).$	are satisfied w.r.t. $I_2$ .
$a(1) :- b(1), \text{ not } c(1).$	is satisfied w.r.t. $I_2$ because both the body and the head are false.
$a(2) :- b(2), \text{ not } c(2).$	is satisfied w.r.t. $I_2$ .
$d(1) :- \#count\{1 : a(1); 2 : a(2)\} = 1.$	is satisfied w.r.t. $I_2$ since $eval(\{1 : a(1); 2 : a(2)\}, I_2) = \{2 : a(2)\}$ , and $\#count\{2 : a(2)\} = 1$ .
$d(2) :- \#count\{1 : a(1); 2 : a(2)\} = 2$	is satisfied w.r.t. $I_2$ because of the evaluation reported above, thus both the body and the head are false.

Moreover,  $I_2$  is also a minimal model for  $P_1$ .

**Definition 2.2.8** (Reduct). Given a ground program  $P$  and an interpretation  $I$ , the *reduct* of  $P$  w.r.t.  $I$  is the subset  $P^I$  of  $P$ , which is obtained from  $P$  by deleting rules in which a body literal is false w.r.t.  $I$ .

It is worthwhile noting that the above definition of reduct, proposed in [56], simplifies the original definition of Gelfond-Lifschitz (GL) transform [74], but is fully equivalent to the GL transform for the definition of answer sets [56].

**Definition 2.2.9** (Answer Set). [112, 74] Let  $I$  be an interpretation for a program  $P$ .  $I$  is an *answer set* for  $P$  if  $I \in MM(P^I)$  (i.e.  $I$  is a minimal model for the program  $P^I$ ). The set of all answer sets for  $P$  is denoted by  $AS(P)$ .

**Example 2.2.7.** Let us consider  $grnd(P_1)$  and  $I_2$ , then the *reduct*  $grnd(P)^{I_2}$  is:

$$\begin{aligned} &b(1). b(2). c(1). \\ &a(2) :- b(2), \text{ not } c(2). \\ &d(1) :- \#count\{2 : a(2)\} = 1. \end{aligned}$$

$I_2$  is a minimal model for  $grnd(P_1)^{I_2}$  since no proper subset of  $I_2$  exists such that is a model for it, and thus  $I_2$  an *answer set* for  $P_1$ .

In particular, since  $P_1$  is a *not disjunctive* and *stratified* program,  $I_2$  is the unique answer set for  $P_1$ , i.e.  $AS(P_1) = \{I_2\}$ . This type of program admits a unique answer set, which corresponds to its *perfect model* [54].

Furthermore, we distinguish *coherent* and *incoherent* programs: *coherent* programs admit at least one answer set, while *incoherent* programs have no answer sets.

In case of weak constraints, answer sets need to be further examined, and classified as *optimal* or not. Intuitively, strong constraints represent conditions that *must* be satisfied, while *weak constraints*, introduced originally in [93, 22], indicate conditions that *should* be satisfied; their semantics involves minimizing the number of violations, thus allowing to easily encode optimization problems.

Optimal answer sets of  $P$  are selected among  $AS(P)$ , according to the following schema. Let  $I$  be an interpretation, then:

$$\begin{aligned} weak(P, I) = & \{(w@l, t_1, \dots, t_m) \\ & : \sim b_1, \dots, b_n[w@l, t_1, \dots, t_m] \text{ occurs in } grnd(P) \\ & \text{and } b_1, \dots, b_n \text{ are true w.r.t. } I\} \end{aligned}$$

For any integer  $l$ , let

$$P_l^I = \sum_{(w@l, t_1, \dots, t_m) \in weak(P, I), w \text{ is an integer}} w$$

denotes the sum of integers  $w$  over tuples with  $w@l$  in  $weak(P, I)$ .

In other words, for each weak constraints satisfied by  $I$  in  $grnd(P)$  we sum the weights per level: these numbers represent a sort of penalty paid by  $I$ : the lower they are, the higher is the possibility for  $I$ , if it represents an answer set, to be optimal.

More formally, we define the notion of *domination* among answer sets as follows. Given an answer set  $A \in AS(P)$ , it is said *dominated* by another answer set  $A'$  if there is some integer  $l$  such that  $P_l^{A'} < P_l^A$  and  $P_{l'}^{A'} < P_{l'}^A$  for all integers  $l' > l$ . An answer set  $A \in AS(P)$  is optimal if there is no  $A' \in AS(P)$  such that  $A$  is dominated by  $A'$ . In general, a coherent program may have one or more optimal answer sets.

**Example 2.2.8.** Let us consider the following ground program  $P_2$ :

$$\begin{aligned} & c(1). c(2). \\ & a(1) \mid b(1) :- c(1). \\ & a(2) \mid b(2) :- c(2). \\ & : \sim a(1). [1@1] \\ & : \sim b(1). [1@2] \\ & : \sim a(2). [2@1] \\ & : \sim b(2). [2@2] \end{aligned}$$

The set  $AS(P_2)$  consists in:

$$\begin{aligned} as_1 : & c(1). c(2). a(1). b(2) \\ as_2 : & c(1). c(2). a(1). a(2) \\ as_3 : & c(1). c(2). b(1). b(2) \\ as_4 : & c(1). c(2). b(1). a(2) \end{aligned}$$

For an answer set  $a$ , we will represent as  $\langle \{w_1, l_1\}, \dots, \{w_n, l_n\} \rangle$  the sum of weights  $w_1, \dots, w_n$  for  $l_1, \dots, l_n$ ,  $n \geq 0$ . Now, for  $as_1$  we obtain  $\langle \{1, 1\}, \{2, 2\} \rangle$ , for  $as_2$   $\langle \{3, 1\}, \{0, 2\} \rangle$ , for  $as_3$   $\langle \{0, 1\}, \{3, 2\} \rangle$  and finally for  $as_4$   $\langle \{2, 1\}, \{1, 2\} \rangle$ . Hence,  $P_2$  admit a unique optimal answer set, namely  $as_2$ , since it is not dominated by any other answer sets.

## 2.3 Advanced Constructs

### Choice Rules

ASP-Core-2 introduces another type of rule, namely *choice rules*; they represent a syntactic shortcut that can be simulated by the rule types previously intro-



duced. However, choice rules, originally proposed in the system *lparse* [122], can greatly ease the task of encoding a computational problem into ASP.

**Definition 2.3.1** (Choice Element). A *choice element* has form:  $a : l_1, \dots, l_m$ , where  $a$  is a *classical atom* and  $l_i$  for  $i \in \{1, \dots, m\}$  is a naf-literal and  $m \geq 0$ .

**Definition 2.3.2** (Choice Atom). A *choice atom* has form:

$$\{e_1; \dots; e_n\} \triangleright u$$

where,

- $n \geq 0$ ,
- $e_i$  for  $i \in \{1, \dots, n\}$  is a choice element,
- $\triangleright \in \{<, <=, =, <>, !=, >, >=\}$ ,
- $u$  is a term.

The operator  $\triangleright$  and the term  $u$  can be omitted whenever  $u = 0$  and  $\triangleright$  corresponds to  $>=$ .

**Definition 2.3.3** (Choice Rule). A *choice rule* consists in a rule with a single choice atom  $a$  in its head, and literals  $b_1, \dots, b_n$  for  $n \geq 0$  in its body:

$$a :- b_1, \dots, b_n.$$

**Example 2.3.1.** An example of choice rule is  $\{a; b; c\} <= 3$ . Intuitively, fixed an interpretation  $I$  if the body is satisfied w.r.t.  $I$ , as in this case since it is empty, it is sufficient that a possibly empty subset of the choice elements is true w.r.t.  $I$  to satisfy the choice atom. Hence, by selecting arbitrarily  $a$ ,  $b$ ,  $c$  as true or false to satisfy the rule. Thus, the rule can be rewritten in the following rules:

$$\begin{array}{l} a \mid -a. \\ b \mid -b. \\ c \mid -c. \end{array}$$

Formally, a choice rule corresponds to the rules, for  $i \in \{1, \dots, n\}$ :

$$a_i \mid \bar{a}_i :- l_1, \dots, l_m, b_1, \dots, b_k$$

and to the constraints:

$$:- b_1, \dots, b_k, \text{not } \# \text{count} \{ \bar{a}_1 : a_1, l_{11}, \dots, l_{m1}; \dots; \alpha_n : a_n, l_{1n}, \dots, l_{mn} \} \triangleright u$$

where, for each classical atom  $s = p(t_1, \dots, t_n)$ ,  $\bar{s} = p'(1, t_1, \dots, t_n)$  and  $\overline{\bar{s}} = p'(0, t_1, \dots, t_n)$ , with  $p'$  an arbitrary predicate associated to  $p$ .

## 2.4 Safety Restriction

In order to instantiate a (non-ground) rule  $r$ , all its variables are considered universally quantified and ranging over the set of all ground terms of the program of which  $r$  is part. However, to ensure the semantics not all ground terms need to be considered, but we can restrict the actual domain for variable substitutions, and in turn to limit the size of the produced instantiation. To this end, as we will mention in Chapter 4, typically, ASP grounders imposed some conditions on the accepted input, such as *lparse*  $\omega$ -restrictedness [121, 122], the  $\lambda$ -restrictedness [70] of the first *gringo* releases (up to version 3.0), and *DLV* safety restriction. In the latest years, *ASP-Core-2* established safety as the standard restriction for modern ASP systems. Essentially, this is a restriction on variables that guarantees that a rule is logically equivalent to the set of its Herbrand instances. Originally, safety has been introduced in the field of databases, in order to ensure that queries over databases are independent from the set of constants considered; similarly, in ASP, safety ensures that programs do not depend on the Universe considered.

Let  $L = \{l_1, \dots, l_n\}$  for  $n \geq 0$  be a set of literals. For a term, literal, rule  $e$  we denote as  $var'(e) \subseteq var(e)$  the set of variables in  $e$  occurring outside of arithmetic terms. The set of safe variables, denoted  $Safe(L) \subseteq var(L)$ , initially corresponding to  $\emptyset$ , is computed inductively according to the following schema:

1. for each classical atom  $a \in L$ ,  $Safe(L) = Safe(L) \cup var'(l)$ ;
2. for each built-in atom  $a \in L$  of form  $v = t$  or  $t = v$ , where  $t$  is a term with  $var(t) \subseteq Safe(L)$  and  $v$  is a variable,  $Safe(L) = Safe(L) \cup v$ ;
3. for each aggregate atom  $a \in L$  of form  $af\{e_1, \dots, e_k\} = v$ , where  $v$  is a variable,  $Safe(L) = Safe(L) \cup v$ .

We will denote aggregate and built-in atoms of types 2 and 3 as *assignment* atoms. Moreover, let  $V$  be the set of variables that an atom  $a$  adds to  $Safe(L)$  we say that  $a$  *binds*  $V$  in  $L$ , or equivalently that  $a$  is a *binder* for  $V$  in  $L$ .

A naf-literal  $l \in L$  is safe if  $var(l) \subseteq Safe(L)$ . An aggregate element  $t_1, \dots, t_p : b_1, \dots, b_q$  appearing in the set  $E$  of aggregate elements of an aggregate literal  $l \in L$  of form  $af\{E\} \triangleright t$  is safe if  $var_g(b_1, \dots, b_q) \subseteq Safe(L)$  and  $var_l(b_1, \dots, b_q) \subseteq Safe(b_1, \dots, b_q)$ ; consequently,  $af\{E\} \triangleright t$  is safe if all its aggregate elements are safe, and  $var(t) \subseteq Safe(L)$ . For a literal  $l \in L$ , let  $VarToSafe(l) = var_g(l) \setminus Safe(l)$ . If  $VarToSafe \neq \emptyset$  we refer to a set of literals  $\{l'_1, \dots, l'_m\} \subseteq L$  ( $m \leq n$ ) binding the set of variables  $VarToSafe(l, l'_1, \dots, l'_m)$  as *saviours* for  $l$ . In general, for the same literal several set of saviours might exist.

A non-choice rule  $r$  is *safe* if every literal in its body is safe and  $var(H(r)) \subseteq Safe(B(r))$ . A choice rule  $r$  is *safe* if every literal in its body is safe and for every choice element of form  $a : N$ , where  $a$  is a classical atom and  $N$  is a set of naf-literals,  $var(a) \subseteq Safe(N) \cup Safe(B(r))$ . A weak constraint  $w$  is *safe* if every literal in its body is safe and  $var(W(w)) \subseteq Safe(B(w))$ . A query  $a?$  is safe if  $var(a) \subseteq Safe(a)$ . A program  $P$  is *safe* if every rule and query composing  $P$  is safe.

**Example 2.4.1.** For instance, the following rules are safe:

$$\begin{aligned}
&a(X) : -b(X), c(X + 1). \\
&a(Y) : -b(X), Y = X + 1. \\
&a(X, Y) : -b(X, Y), \text{not } c(X, Y). \\
&a(Z) : -h(Y), Z = \#count\{X : g(X), \text{not } f(X, Y)\}. \\
&a(X, Y) : -Y = \#sum\{W : g(W)\}, X = \#count\{Z : g(Z), \\
&\quad \text{not } f(Z, Y)\}, \text{not } b(X, Y). \\
&:- \#min\{X, S : b(T, X), S = (2 * T) - X\} = Y. \\
&:\sim \#max\{X : b(X, X + 1)\} = Y. [1@1, f(Y * Y)] \\
&\{a(X, Y) : b(Y); b(X) : c(X * 3)\} : -d(X).
\end{aligned}$$

These other rules are not safe:

$$\begin{aligned}
&a(X) : -c(X + 1). \\
&a(Y) : -Y = X + 1. \\
&a(X, Y) : -\text{not } c(X, Y). \\
&a(Z) : -Z = \#count\{X : g(X), \text{not } f(X, Y)\}. \\
&a(X, Y) : -X = \#count\{Z : g(Z), \text{not } f(Z, Y)\}, \text{not } b(X, Y). \\
&:- \#min\{X, S : b(T, X), S + X = (2 * T)\} = Y. \\
&:\sim \#max\{X : b(X, X + 1)\} = Y. [1@1, f(Y * Z)] \\
&\{a(X, Y) : b(Y); b(X) : c(X * 3)\} : -\text{not } d(X).
\end{aligned}$$

Now, let us consider the following rule  $r_1$ :

$$a(X, Y, Z) : -b(X, Y), c(Y, Z), Z = Y + 1, d(Z + 1), \#count\{T : e(T, X)\} < Y.$$

It is easy to see that  $r_1$  is safe. In addition:

- $b(X, Y)$  binds the variables  $X$  and  $Y$ ;
- $c(Y, Z)$  binds the variables  $Y$  and  $Z$ ;
- $Z = Y + 1$  binds the variable  $Z$ ;
- $d(Z + 1)$  and  $\#count\{T : e(T, X)\} < Y$  do not bind any variable.

For the literals  $b(X, Y)$  and  $c(Y, Z)$   $VarToSafe = \emptyset$ , while for  $Z = Y + 1$   $VarToSafe = Y$ , hence  $b(X, Y)$  is a saviour for it, as well as  $c(Y, Z)$ , because both atoms bind  $Y$ . For  $d(Z + 1)$ ,  $VarToSafe = Z$ , and so there are three possible sets of saviours:  $\{Z = Y + 1, b(X, Y)\}$ ,  $\{Z = Y + 1, c(Y, Z)\}$ , or simply  $\{c(Y, Z)\}$ . Note that even if  $Z = Y + 1$  binds  $Z$  the built-in alone is not a saviour, because it needs a saviour for  $Y$ . Finally, for  $\#count\{T : e(T, X)\} < Y$  possible saviours are:  $\{b(X, Y)\}$ ,  $\{b(X, Y), c(Y, Z)\}$ .

## Chapter 3

# Knowledge Representation and Reasoning

The high knowledge-modeling power of ASP as well as the availability of reliable, high-performance implementations [34, 68] made ASP suitable for solving a variety of complex problems arising in scientific applications [36] from several areas ranging from Artificial Intelligence [6, 10, 11, 60, 106, 20, 62], to Knowledge Management [9, 12] and Databases [98, 19, 90, 15].

As previously anticipated, the ordering of literals and rules is immaterial, thus ASP is intended as a fully declarative language. Moreover, the concept of negation as failure and the *closed world assumption* make it particularly suitable for representing incomplete knowledge and nonmonotonic reasoning. The expressive power is a further key property: the only restriction of ASP to disjunction and negation as failure captures the complexity class  $\Sigma_2^P = \text{NP}^{\text{NP}}$ . More precisely, ASP programs may express, in a precise mathematical sense, every property of finite structures over a function-free first-order structure that is decidable in nondeterministic polynomial time with an oracle in NP [53, 45].

This chapter provides, by means of proper examples, some insights about the capabilities of ASP as a tool for *Knowledge Representation and Reasoning* focusing on the *GCO* programming methodology, then illustrates some Deductive Databases applications.

### 3.1 The GCO Technique

To grasp the power behind ASP properties in this section we introduce one of the most common programming paradigm in ASP, namely the Guess & Check & Optimize (*GCO*) technique.

The key idea of *GCO* is the following: a computational problem is encoded by an ASP program composed by (i) a set of (usually disjunctive or choice) rules, called “guessing part”, used to define the search space; (ii) another (optional) set of rules, called “checking part”, which impose some admissibility constraint; (iii) a further (optional) set of weak constraints to specify preferences over found solutions. Once the program has been expressed in a (non-ground) program, typically referred to as *encoding*, it can be then paired with a set of facts specifying an *instance* of the problem; this separation allows to encode problems in

a uniform way over varying instances.

More in detail, given a set  $F_I$  of facts that specify an instance  $I$  of some problem  $P$ , a *GCO* program  $P$  for  $P$  consists of the following two main parts:

- *Guessing Part.* The guessing part  $G \subseteq P$  of the program defines the search space, such that answer sets of  $G \cup F_I$  represent “solution candidates” for  $I$ .
- *Checking Part.* The (optional) checking part  $C \subseteq P$  of the program filters the solution candidates in such a way that the answer sets of  $G \cup C \cup F_I$  represent the admissible solutions for the problem instance  $I$ .
- *Optimization Part.* The (optional) optimization part  $O \subseteq P$  of the program allows to express a quantitative cost evaluation of solutions by using weak constraints. It implicitly defines an objective function  $f : AS(G \cup C \cup F_I) \rightarrow \mathbb{N}$  mapping the answer sets of  $G \cup C \cup F_I$  to natural numbers. The semantics of  $G \cup C \cup F_I \cup O$  optimizes  $f$  by filtering those answer sets having the minimum value; this way, the optimal (least cost) solutions are computed.

The *GCO* programming methodology has also positive implications from the “Software Engineering” point of view. Indeed, the modular program structure typical of *GCO* allows to develop programs in an incremental fashion, which is helpful to simplify testing and debugging. In particular, this methodology allows to separate the specification the guessing part  $G$ , so that it can be tested that  $G \cup F_I$  correctly defines the search space; then, by means of the checking part it is possible to ensure that the answer sets of  $G \cup C \cup F_I$  actually encode the admissible solutions and via  $G \cup C \cup F_I \cup O$  solutions can be classified.

### Three-colorability

As first example, consider the well-known NP-complete problem *Three-colorability*:

Given an undirected graph  $G = (V, E)$ , assign to each vertex one of three colors such that adjacent vertices always have distinct colors.

Firstly, we can start defining a problem instance via facts. In this case, we need to model an undirected graph  $G = (V, E)$ :

- vertices can be encoded as facts of the form  $vertex(x)$ ;
- for edges, facts of type  $edge(x, y)$  can be used to encode that is there an edge between the vertices  $x$  and  $y$ .

The next step consists in encoding the actual problem into an ASP *GCO* program  $P_{3col}$ . In the guessing part we define the search space, thus we assign to each vertex exactly one color among the three available, say red, green and blue. In the checking part we ensure that no two connected vertices are associated with the same color. For this problem, we do not need an optimize part, indeed,

there are not preferences among solutions to be expressed.

```
% Guessing Part (1 rule)
color(X,red) | color(X,green) | color(X,blue):- vertex(X).

% Checking Part (1 rule)
:- edge(X,Y),color(X,C),color(Y,C).
```

Notably, thanks to the declarative capability of ASP, when designing the encoding the focus is on how to *model* the problem at hand, rather than on how to actually *solve* it.

By coupling  $P_{3col}$  with a set of facts  $F$  for *vertex* and *edge*, if the program  $P_{3col} \cup F$  is coherent, than each answer set represents an admissible solution. For instance, suppose that

$$F = \{vertex(1), vertex(2), vertex(3), edge(1, 2), edge(1, 3), edge(2, 3)\}$$

then the input graph is complete (i.e. every pair of distinct vertices is connected by an edge) and the answer set of  $P_{3col} \cup F$  are:

```
as1 : color(1,blue).   color(2,red).   color(3,green).
as2 : color(1,blue).   color(2,green). color(3,red).
as3 : color(1,red).    color(2,blue).  color(3,green).
as4 : color(1,red).    color(2,green). color(3,blue).
as5 : color(1,green).  color(2,red).   color(3,blue).
as6 : color(1,green).  color(2,blue).  color(3,red).
```

### Hamiltonian Path

As next example, let us consider a classical NP-complete problem in graph theory, namely *Hamiltonian Path*:

Given a directed graph  $G = (V, E)$  and a vertex  $v \in V$  of this graph, does there exist a path in  $G$  starting from  $v$  and passing through each vertex in  $V$  exactly once?

Similarly to Example 3.1 the graph  $G = (V, E)$  can be specified by means of facts over predicates *vertex/1* and *edge/2*. Moreover, we need to define the starting vertex  $v$ : it can be specified by the predicate *start/1*. Consequently, the following program  $P_{hp}$  encodes a solution to the problem:

```
% Guessing Part (3 rules)
{inPath(X,Y)} :- start(X),edge(X,Y).
{inPath(X,Y)} :- reached(X),edge(X,Y).
reached(X) :- inPath(Y,X).

% Checking Part (3 rules)
:- inPath(X,Y),inPath(X,Y1),Y <> Y1.
:- inPath(X,Y),inPath(X1,Y),X <> X1.
:- vertex(X),not reached(X),not start(X).
```

In the guessing part, the two choice rules guess a subset  $S$  of the edges to be in the path, while the rest of the program checks whether  $S$  constitutes a Hamiltonian Path. Here, an auxiliary predicate *reached* is used, which is associated with the guessed predicate *inPath* using the third rule. Note that *reached* is completely determined by the guess for *inPath*, and no further guessing is needed. In turn, through the second rule, the predicate *reached* influences the guess of *inPath*, which is made somehow inductively: initially, a guess on an edge leaving the starting vertex is made by the first rule, followed by repeated guesses of edges leaving from *reached* vertices by the second rule, until all *reached* vertices have been handled.

In the checking part, the first two constraints ensure that the set of edges  $S$  selected by *inPath* meets the following requirements, which every Hamiltonian Path must satisfy: (i) there must not be two edges starting at the same vertex, and (ii) there must not be two edges ending in the same vertex. The third constraint enforces that all vertices in the graph are *reached* from the starting vertex in the subgraph induced by  $S$ .

It is easy to see that any set of edges  $S$  which satisfies all three constraints must contain the edges of a path  $v_0, v_1, \dots, v_k$  in  $G$  that starts at vertex  $v_0 = a$ , and passes through distinct vertices until no further vertex is left, or it arrives at the starting vertex  $a$  again. In the latter case, this means that the path is in fact a Hamiltonian Cycle (from which a Hamiltonian path can be immediately computed, by dropping the last edge).

Thus, given a set of facts  $F$  for *vertex*, *edge*, and *start*, the program  $P_{hp} \cup F$  has an answer set if and only if the corresponding graph has a Hamiltonian Path.

### Ramsey Numbers

We show next another use of the *GCO* programming technique: we build an ASP program whose answer sets witness that a property does not hold, i.e. the property at hand holds if and only if the ASP program has no answer set at all. We next apply the above programming scheme to a problem from the number and graph theories, namely Ramsey Numbers.

The Ramsey number  $R(k, m)$  is the least integer  $n$  such that, no matter how we color the edges of the complete undirected graph (clique) with  $n$  vertices using two colors, say red and blue, there is a red clique with  $k$  vertices (a red  $k$ -clique) or a blue clique with  $m$  vertices (a blue  $m$ -clique) [113].

We next show a program  $P_{ramsey}$  that allows us to decide whether a given integer  $n$  is not the Ramsey Number  $R(3, 4)$ . By varying the input number  $n$ , we can determine  $R(3, 4)$ , as described below. Let  $F$  be the collection of facts for input predicates *vertex* and *edge* encoding a complete graph with  $n$  vertices.

$P_{ramsey}$  is the following program:

```

% Guessing Part (1 rule)
blue(X,Y) | red(X,Y):- edge(X,Y).

% Checking Part (2 rules)
:- red(X,Y),red(X,Z),red(Y,Z).
:- blue(X,Y),blue(X,Z),blue(Y,Z),
   blue(X,W),blue(Y,W),blue(Z,W).

```

Intuitively, the disjunctive rule guesses a color for each edge. The first constraint eliminates the colorings containing a red clique (i.e. a complete graph) with 3 vertices, and the second constraint eliminates the colorings containing a blue clique with 4 vertices. The program  $P_{ramsey} \cup F$  has an answer set if and only if there is a coloring of the edges of the complete graph on  $n$  vertices containing no red clique of size 3 and no blue clique of size 4. Thus, if there is an answer set for a particular  $n$ , then  $n$  is not  $R(3,4)$ , that is,  $n < R(3,4)$ . On the other hand, if  $P_{ramsey} \cup F$  has no answer set, then  $n \geq R(3,4)$ . Thus, the smallest  $n$  such that no answer set is found is the Ramsey number  $R(3,4)$ .

### k-Clique

In this example we focus on the relation between *stratification* negation and *disjunction*. To this end, let us consider the following NP –complete problem in graph theory, referred to as *k-Clique*:

Given an undirected graph  $G = (V, E)$  and an integer  $k$ , a *k-clique* of  $G$  is a complete subgraph of  $G$  with  $k$  vertices.

As in previous examples, a graph can be encoded by means of facts representing vertices and edges and a fact of form  $k(n)$  can be used to represent the integer  $k$ , while the problem can be encoded in the following  $P_{k-clique}$  program:

```

% Guessing Part (2 rules)
clique(X):- vertex(X),not nonClique(X).
nonClique(X):- vertex(X),not clique(X).

% Checking Part (2 rules)
:- #count{X : clique(X)}! = N, k(N).
:- clique(X),clique(Y),X! = Y,not edge(X,Y),not edge(Y,X).

```

The guessing part generates potential cliques. Notably, in these two guessing rules there is a negative dependency between the predicates *clique/1* and *nonClique/1*, thus  $P_{k-clique}$  is unstratified under negation. Interestingly, we are using unstratified negation to simulate disjunction: by replacing the two rules with the rule

$$clique(X) \mid nonClique(X) :- vertex(X).$$

we obtain a semantically equivalent program. Indeed, even if only in some cases, unstratified negation can, somehow, model disjunction. More in detail, unless



the polynomial hierarchy collapses, this correspondence holds for NP problems, as in general disjunction allows to express problems up to  $\Sigma_2^P = \text{NP}^{\text{NP}}$ . In the checking part, the first constraint verifies that only  $k$ -cliques are considered and the second one ensures that vertices belonging to a clique are connected. Given an instance  $I$  fixing  $k$  to  $k_1$ , if  $P_{k\text{-clique}} \cup I$  has no answer set, then no  $k_1$ -clique exists, otherwise each returned answer set will represent a clique of the desired size  $k_1$ .

As final remark, for the guessing part we could instead use a *choice* rule as in the following  $Q_{k\text{-clique}}$  program:

```
% Guessing Part (1 rule)
N = {clique(X) : vertex(X)} :- k(N).

% Checking Part (1 rule)
:- clique(X), clique(Y), X != Y, not edge(X, Y), not edge(Y, X).
```

By doing so, the first constraint can be removed, since the requirement on the clique size can be directly specified in the choice rule.

### Maximal Clique

So far, we considered problems without an optimizing part. As example of complete *GCO* program let us consider a NP-hard problem, namely *Maximal Clique*. It can be considered as a variant of the *k-clique* problem:

Given an undirected graph  $G = (V, E)$  and an integer  $k$ , determine a clique  $C$  of maximal size in  $G$ , i.e. for each other clique  $C'$  in  $G$ , the number of vertices in  $C$  must be larger than or equal to the number of nodes in  $C'$ .

Readopting the same instance representation of Example 3.1, with few modifications to the program  $P_{k\text{-clique}}$  we obtain the following encoding  $P_{\text{max-clique}}$ :

```
% Guessing Part (2 rules)
clique(X) :- vertex(X), not nonClique(X).
nonClique(X) :- vertex(X), not clique(X).

% Checking Part (1 rule)
:- clique(X), clique(Y), X != Y, not edge(X, Y), not edge(Y, X).

% Optimizing Part (1 weak constraint)
:- nonClique(X). [1, X]
```

The guessing part remains unchanged, while in the checking part we removed the constraint on the cliques size. Indeed, in this case we want to find cliques with the maximal size, which is unknown, thus a weak constraint allows us to express this requirement. Each answer set will pay a penalty at level 0 according to how many vertices are not included in the clique it represents, hence the optimal answer set(s) will be the ones with the cliques of the biggest possible size.

## 3.2 Deductive Database Applications

In the following, we present two classical problems from the deductive database field. Notably, both can be encoded by using only positive rules.

### Same Generation

Given a parent-child relationship, represented by acyclic directed graph, we want to find all pairs of persons belonging to the same generation. Two persons are of the same generation if either they are siblings, or they are children of two persons of the same generation. If input is encoded by a relation  $parent(X, Y)$ , where a fact  $parent(a, b)$  states that  $a$  is a parent of  $b$ , the solution can be encoded by the following recursive program, which computes a relation  $samegeneration(X, Y)$  containing all facts such that  $X$  is of the same generation as  $Y$  :

$$\begin{aligned} samegeneration(X, Y) & \\ & :- parent(P, X), parent(P, Y). \\ samegeneration(X, Y) & :- parent(P1, X), \\ & parent(P2, Y), samegeneration(P1, P2). \end{aligned}$$

### Reachability

Given a finite directed graph  $G = (V, E)$ , we want to compute all pairs of nodes  $(a, b) \in V \times V$  such that  $b$  is reachable from  $a$  through a nonempty sequence of edges in  $E$ . In other words, the problem amounts to computing the transitive closure of the relation  $E$ .

In the following ASP encoding, we assume that  $E$  is represented by the binary relation  $edge(X, Y)$ , where a fact  $edge(a, b)$  means that  $G$  contains an edge from  $a$  to  $b$ , i.e.  $(a, b) \in E$ ; the set of nodes  $V$  is not explicitly represented, since the nodes appearing in the transitive closure are implicitly given by these facts. Hence, the following program  $P_{reach}$  computes a relation  $reachable(X, Y)$  containing all facts  $reachable(a, b)$  such that  $b$  is reachable from  $a$  through the edges of the input graph  $G$ :

$$\begin{aligned} reachable(X, Y) & :- edge(X, Y). \\ reachable(X, Y) & :- edge(X, U), reachable(U, Y). \end{aligned}$$



## Chapter 4

# ASP Computation And Implementations

Answer Set Programming has been introduced more than twenty years ago. Throughout the years, a significant effort has been spent in defining techniques for an efficient computation of the answer set semantics, and therefore reliable and high-performance implementations [34, 68] have been released.

This chapter recalls the history of the development of ASP systems. Section 4.2 outlines the main ASP systems developed over the years, while Section 4.2.2 traces the history of the research carried out on the grounding and solving phases.

### 4.1 ASP Systems

The “ground & solve” strategy [82] is currently the commonly adopted strategy in state-of-the-art-systems. This approach mimics the definition of the semantics as given in Section 2.2 by relying on a grounding module (*grounder* or *instantiator*), that generates a propositional theory semantically equivalent to the input program, coupled with a subsequent module (*solver*) that applies proper propositional techniques for generating its answer sets. These phases are usually referred to as *instantiation* or *grounding*, and *solving* or *answer sets search*, respectively.

Throughout the years, systems based on this approach have been released either as monolithic, i.e. internally embedding grounding and solving modules, or as combinations of stand-alone grounders and solvers.

Initially, ASP systems were not able to handle disjunctive heads. In this context, the grounder *lparse* [121] in combination with the solver *smodels* [117] represented one of the first reliable implementation. Later on, *smodels<sub>cc</sub>* [128] was released as an improved version of *smodels*.

The first systems capable of dealing with disjunctive programs have been the monolithic *DLV* [92], and the combinations of the solvers *GnT* [81] and *cmmodels* [75] with an enhanced version of *lparse* supporting disjunction.

The system *DLV* has been fruitfully exploited many relevant industrial applications significantly contributing in fostering the use of ASP in real-world

scenarios. Besides the core system, several extensions have been developed geared, to different extents, towards easing the interoperability of ASP [94].

In the following years, *clingo* [63] represented a novel and efficient ASP system. It combines in a monolithic system the grounder *gringo* [66] and the solver *clasp* [64], which have also been released as stand-alone systems. The recent releases offer advanced capabilities: the series 4 introduces high-level constructs for realizing complex reasoning processes, such as incremental reasoning and much more, while starting from version 5 *clingo* allows to integrate ASP with theory-specific reasoning.

Recently, a new efficient solver have been released, namely *wasp* [3], which is compatible with *gringo*.

A different approach is pursued by the system *LP2SAT* [80] that represents a family of sub-systems relying on a translation of propositional logic programs into logic formulas so that models of the resulting formula are in one-to-one correspondence with the answer sets of the input program.

Eventually, in contraposition with the traditional approach, systems such as *Gasp* [44], *Asperix* [89, 87], *Omega* [46] and *Alpha* [129] are instead tailored on *lazy grounding*, a technique in which the grounding and solving steps are interleaved, and rules are grounded on-demand during solving. These systems try to overcome the so called *grounding bottle-neck*, that occurs on problems for which the instantiation is inherently so huge that the traditional approach is not suitable. Indeed, grounders accomplish an intrinsically tough task, which is in general EXPTIME-hard [45] (see Chapter 5).

## 4.2 Ground & Solve Approach

Hereafter, we spotlight the traditional approach for computing the answer set(s) of an ASP program. In particular, in the following subsections we recall the major grounders and solvers, highlighting the differences in the techniques they adopt.

### 4.2.1 Grounding Phase

The grounding phase acts like a bridge between the logic programming paradigm with variables that ease modelling in ASP and propositional programs which are crucial to devise efficient solving procedures. Instantiators perform a very delicate task, as their grounding process highly influences the subsequent solving phase.

As already mentioned, there are three stable grounders for ASP: *lparse*, the *DLV* instantiator [55], and *gringo*.

*lparse* has been released as a front-end grounder system, whose output encoded in a suitable numeric format, was intended to be given as input to a separated solver. Moreover, *lparse* accepts logic programs respecting its  $\omega$ -restrictedness constraint. This condition enforces each variable in a rule to occur in a positive body literal, called domain literal, whose predicate (*i*) is not mutually recursive with the head, and (*ii*) is neither unstratified nor dependent (also transitively) on an unstratified predicate. To instantiate a rule *r*, *lparse* leverages this restriction, employing a nested loop that iterates on the extensions of the domain predicates occurring in the body of *r*, and generates ground

instances accordingly. Notably, *lparse* originally proposed choice rules, and the aforementioned codification of the computed ground program into a numeric format.

On the other hand, *DLV* was originally intended as a monolithic system, with a tight internal integration of grounding and solving modules. Its distinguishing feature was its ability to handle disjunctive rules. *DLV* imposes the less restrictive condition of *safety*, requiring that each variable in a rule appears in some positive body literal. Furthermore, the *DLV* instantiator introduced the notion of *intelligent* grounding, contributing to the definition of advanced techniques nowadays employed in modern grounders.

Similarly to *lparse*, the first versions of *gringo* were based on the concept of  $\lambda$ -restrictedness [70], consisting of an extension of the *lparse*  $\omega$ -restrictedness. Starting from version 3.0, *gringo* removed domain-based restrictions and instead requires *safety*. Notably, currently *gringo* represents the mainstream ASP grounder thanks to its efficiency and the compliance with ASP-Core-2. *DLV* and *lparse* have been released before the introduction of ASP-Core-2, and their input languages are slightly different; for instance, *DLV* does not handle relevant constructs such as choice rules. Moreover, like *lparse*, *gringo* encodes its output in a numeric format representing the de-facto standard format that mainstream solvers are able to interpret.

### 4.2.2 Solving Phase

The solving algorithms are strictly correlated to satisfiability (SAT) algorithms. The solvers *smodels*, *smodels<sub>cc</sub>* and the solving module of *DLV* employ for their answer set search procedures variations of the classic backtrack-search Davis-Putnam-Logemann-Loveland (DPLL) algorithm, on which many modern SAT solvers rely. The essential steps of the DPLL algorithm are: *decision*, *unit propagation*, and *backtracking*. Essentially, during the decision operation solvers select an *undefined* atom, that is an atom which its truth value is currently unknown, and assume its truth or falsity. The unit propagation step consists in propagating this assumption and deriving logic consequences by checking whether other undefined atoms become true or false. Whenever, an inconsistency is derived a backtracking step is performed. Starting from this basic algorithm, answer set solvers perform specialized operations tailored on answer set search. Notably, *smodels* implements five different propagators, and the solver *smodels<sub>cc</sub>* enhances the *smodels* solving process with conflict-driven backjumping and clause learning. In *DLV* the backtracking step is instead improved by a backjumping machinery empowered with look-back heuristics [100]; in addition, *DLV* differs from the other two solvers, for its capability of dealing with disjunction.

An even stricter correlation with SAT emerged with the solver *cmmodels*, which carries out its answer set search by distinguishing between “tight” and “non-tight” programs. In this context, a predicate appearing in the head of a rule is said to positively depend on the predicates featured by positive literals in the body. A program is tight if there are no cyclic positive dependencies among predicates. For tight programs, it leverages on the observation that this class of programs, answer sets can be found by running a SAT solver on classified programs’ completion, stemming from the theoretical results provided in [41]; while for non-tight programs, the concept of a loop formula has been introduced

in [97, 86], and employed in order to compute answer sets of such programs, in both solvers *cmodels* and *assat* [97].

The solver *clasp* combines both the two aforementioned approaches. Given an input program, similarly to *cmodels* and *assat*, *clasp* initially computes the clausified completion; then, it is endowed with a search procedure that relies on a unit propagator stemming from SAT on the program's completion and a further unfounded propagator stemming from the former class of solvers. Moreover, it relies on advanced techniques such as conflict-driven backjumping and clause learning.

Similar techniques are adopted also in the more recent solver *wasp*, properly extended with custom propagation functions to handle the specific properties of ASP programs.

Further insights about ASP solvers found in [96]. Nowadays, these two latter systems represent the mainstream solvers. As already anticipated, they require that the input program is encoded into a numeric format, allowing them an efficient and faster parsing.

## Part II

# Designing an Efficient ASP Grounder





This part focuses on the first purpose of this thesis, the design of an efficient ASP-Core-2 instantiator. In particular, after outlining the basic instantiation process, we present a series of optimization techniques studied and defined with the aim of reducing both the time and the size of the instantiation. The part is divided as follows:

- in Chapter 5 we present the theoretical aspects and practical issues behind the grounding phase, and then abstract a general and basic instantiation process, for which in the following chapters a broad range of optimizations are introduced.
- Chapter 6 discusses some different indexing strategies, pondering the advantages and disadvantages of each alternative.
- Chapter 7 illustrates different body ordering strategies, outlining the differences among them.
- Chapter 8 presents a decomposition rewriting geared towards automatically determining according to suitable heuristics whether splitting long body rules into multiple smaller ones is a convenient choice. After the definition of an abstract decomposition algorithm, we illustrate a specific version aiming at optimizing the grounding process.
- Finally, in Chapter 9 we define a variegated set of optimizations acting in diverse situations and allowing to further fine-tune performance.

The techniques illustrated in Chapter 6, Chapter 7 and Chapter 9 have been partially reported in [28], which was selected as “best paper” at the conference AI\*IA 2016, the 15th International Conference of the Italian Association for Artificial Intelligence. In addition, the decomposition rewriting presented in Chapter 8 has been reported in [33], and nominated as “student best paper” at PADL 2018, the 20th International Symposium on Practical Aspects of Declarative Languages.



## Chapter 5

# Grounding Process

The chapter is organized as follows. In Section 5.1 we illustrate the major issues of instantiation from a theoretical side. In Section 5.2 we abstract the basic instantiation process. Section 5.3 recalls the major optimization means employed by mainstream grounders.

### 5.1 Computational Complexity

Theoretically, the instantiation of an ASP program  $P$ ,  $grnd(P)$ , can be obtained by systematically replacing variables by all possible ground terms in  $U_P$ . Practically, grounding is much more than a naive replacement of variables: modern ASP grounders are empowered with smart procedures for an efficient and as small as possible instantiation. The need for efficient approaches arises from the intrinsic complexity of the instantiation which has a big impact on the performance of the whole computational process, as its output is the input for a solver module, that, in the worst case, takes exponential time in the size of the input ground program [13, 14]. Indeed, the grounding step may lead a bottleneck and an efficient instantiation is often crucial in real-world applications involving large input data.

In order to understand the challenges of grounding from a theoretical point of view, we provide next some complexity results relevant to this purpose. For more details the reader may refer to [125, 53, 45, 93, 50, 58]. For the sake of simplicity, let us restrict our attention to ASP programs which are stratified under negation, non-disjunctive, constraint-free, function-free, and containing in their rules just positive classical literals. More specifically, we will consider plain Datalog programs.

Recall that a program  $P$  may be split into an encoding  $E$  and into an instance  $I$  (cf. Chapter 3). As Datalog has its roots in relational databases, the concepts of *program* and *database* are more commonly used in place of, respectively, *encoding* and *instance*. Indeed, a *database* is identified with sets of facts, and all facts with the same predicate symbol  $p$  represent a data relation. The set of all predicate symbols occurring in the database together with a possibly infinite domain for the argument constants is called the schema of the database. With each database  $D$ , we associate a finite universe  $U_D$  of constants which encompasses at least all constants appearing in  $D$ , but possibly more.

Let  $P$  be a Datalog program,  $D$  a database and  $A$  a set of ground atoms; there are three main kinds of complexity connected to plain Datalog and its various extensions [125]:

- The *data complexity* is the complexity of checking whether  $D \cup P \models A$  for a fixed Datalog program  $P$  and variable  $D$  and  $A$ .
- The *program complexity* (also called *expression complexity*) is the complexity of checking whether  $D \cup P \models A$  for a fixed database  $D$  and variable  $P$  and  $A$ .
- The *combined complexity* is the complexity of checking whether  $D \cup P \models A$  when  $D$ ,  $P$  and  $A$  are variable.

To accomplish such decision problems, we need to perform the following steps: (i) compute  $grnd(P \cup D)$ , (ii) determine the unique model of  $grnd(P \cup D)$ , say  $M$ , (iii) check whether  $A \subseteq M$ . Indeed, in the restricted setting of plain Datalog  $P \cup D$  admits a unique model that coincides with the unique least Herbrand model [45]; moreover, the step (ii) can be computed in linear time w.r.t. the size of  $grnd(P \cup D)$  [49]. Hence, the most computational expensive step consists in computing the grounding.

Let us analyze firstly data complexity. Grounding a fixed program  $P$  on an input database  $D$  yields polynomially many rules in the size of  $D$ : let  $|P|$  be the number of rules in  $P$ ,  $c$  denotes the number of constants appearing in  $U_D$  and  $v$  denotes the maximum number of variables appearing in any rule of  $P$ ;  $grnd(P \cup U)$  consists in at most  $|P| \cdot c^v$ . When  $P$  is fixed  $|P|$  and  $v$  are known,  $grnd(P \cup D)$  is polynomial in the size of  $D$ . More precisely, it has been proved that:

**Theorem 1** ([45]). *Datalog is data complete for P.*

Regarding program complexity, when  $P$  and  $A$  are variable while  $D$  is fixed, by applying a similar reasoning we obtain that grounding  $P \cup D$  might generate up to  $|P| \cdot c^v$  ground rules. Since  $c$  is fixed while  $|P|$  and  $v$  are not, we obtain:

$$|P| \cdot c^v = 2^{\log_2 |P| \cdot c^v} = 2^{\log_2 |P| + \log_2 c^v} = 2^{\log_2 |P| + v \cdot \log_2 c}$$

Therefore,  $grnd(P \cup U)$  might consist in an exponential number of rules w.r.t. the size of  $P$ ; indeed, it has been proved that:

**Theorem 2** ([45]). *Datalog is program complete for EXPTIME.*

Eventually, concerning the combined complexity, for plain Datalog (as well as for some other relevant Datalog extensions) it is equivalent to the program complexity w.r.t. polynomial-time reductions [45].

**Example 5.1.1.** Let us consider the following program,  $P_1$ :

$$disp(X_1, \dots, X_n) :- obj(X_1), \dots, obj(X_n).$$

along with the database  $D_1$ :

$$obj(0). obj(1).$$

Essentially,  $D_1$  is fixed and contains only two facts while  $P_1$  contains one non-ground rule with a non-fixed number of variables. The instantiation  $ground(P_1 \cup D_1)$  contains  $2^n + 2$  rules:  $2^n$  ground instances for the input rule, and the two input facts. For instance, for  $n = 3$ , we obtain 8 ground instances:

$$\begin{aligned} disp(0, 0, 0) &:- obj(0), obj(0), obj(0). \\ disp(1, 0, 0) &:- obj(1), obj(0), obj(0). \\ disp(0, 1, 0) &:- obj(0), obj(1), obj(0). \\ disp(0, 0, 1) &:- obj(0), obj(0), obj(1). \\ disp(1, 1, 0) &:- obj(1), obj(1), obj(0). \\ disp(1, 0, 1) &:- obj(1), obj(0), obj(1). \\ disp(0, 1, 1) &:- obj(0), obj(1), obj(1). \\ disp(1, 1, 1) &:- obj(1), obj(1), obj(1). \end{aligned}$$

Intuitively,  $P_1$  consists the combinatorial problem of computing the dispositions with repetition of 2 objects in groups of  $n$  corresponding exactly to  $2^n$ .

The above reported results prove that even when the ASP language is restricted to plain Datalog, grounding is a quite complex task. In the general case where an ASP program  $P$  is given in the input, grounders solve a problem which is in general EXPTIME-hard: the size of the grounding  $grnd(P)$  can be up to single exponential in the size of  $P$ . Optimizations intervening in their instantiation process therefore often have a big impact and aim at avoiding the exponential blowup, when possible. In addition, there are classes of programs for which the grounding size can be dramatically reduced, such as programs for which we can assume that the arity of input predicates is bounded [50] or the maximum number of variables allowed in input rules is bounded [126]. Moreover the magic-sets technique [4] may sometimes avoid the exponential space requirements if the input program contains a query by emulating a top-down derivation (we will come back to this technique in Chapter 10).

## 5.2 Grounding an ASP Program

In this section, we outline the basic grounding process consisting in a bottom-up evaluation based on a semi-naive approach of the input program. Over time this approach proved to be particularly reliable and efficient over a huge number of scenarios, especially when domain extensions are very large.

To instantiate a program, according to this strategy, it is firstly divided into sub-programs as illustrated in Section 5.2.1, next each sub-program is instantiated as described in Section 5.2.2. In Section 5.2.3 we provide a description of the core of the whole process, the instantiation of a single rule. For the sake of readability, in Sections 5.2.2 and 5.2.3 we restrict our attention to the basic ASP syntax, and do not take into account linguistic extensions such as choice rules, aggregate literals and built-in atoms; their instantiation is separately outlined in Section 5.2.4.

### 5.2.1 Dependency Analysis

Given a program  $P$ , the Dependency Graph of  $P$ , denoted  $G_P$ , defined in Section 2.1 induces a partition of  $P$  into subprograms (or *modules*) allowing for a

modular evaluation of  $P$ , then the Component Graph of  $P$ , denoted  $G_P^c$ , induces a partial ordering among the SCCs of the Dependency Graph as follows.

**Definition 5.2.1** (Positive and Negative Precedences). For any pair of nodes  $A, B$  of  $G_P^c$ ,  $A$  *positively precedes*  $B$  in  $G_P^c$  (denoted  $A \prec_+ B$ ) if there is a *path* in  $G_P^c$  from  $A$  to  $B$  in which all edges are labeled with “+”;  $A$  *negatively precedes*  $B$  (denoted  $A \prec_- B$ ), if there is a path in  $G_P^c$  from  $A$  to  $B$  in which at least one edge is labeled with “-”.

This ordering induces *admissible component sequences*  $C_1, \dots, C_n$  of SCCs of  $G_P$  such that for each  $i < j$ :

- $C_j \not\prec_+ C_i$
- if  $C_j \prec_- C_i$  then there is a cycle in  $G_P^c$  from  $C_i$  to  $C_j$  (i.e. either  $C_i \prec_+ C_j$  or  $C_i \prec_- C_j$ )

Essentially, it is required that if a module  $A$  positively precedes a module  $B$  then  $A$  must be evaluated before  $B$ , while if  $A$  negatively precedes  $B$  then  $A$  should be possibly evaluated before  $B$ . In general, for a program  $P$  several admissible component sequences may exist.

**Example 5.2.1.** Given the program  $P_1$  of Example 2.1.12 its Dependency and Component graphs are depicted in Figure 2.1. It can be observed that  $\{c/1\} \prec_+ \{d/1\}$ ,  $\{c/1\} \prec_- \{a/1, b/1\}$ , and  $\{a/1, b/1\} \prec_- \{c/1\}$ . Two different admissible component sequences would be  $\{c/1\}, \{d/1\}, \{a/1, b/1\}$ , and  $\{a/1, b/1\}, \{c/1\}, \{d/1\}$ .

An admissible component sequence  $C_1, \dots, C_n$  permits an incremental instantiation of  $P$ , one module at a time, preserving the semantics: the instantiation of  $P$  is performed by iteratively instantiating the modules  $M_{C_1}, \dots, M_{C_n}$ . Furthermore, strong and weak constraints are not represented in the aforementioned graphs due to the missing heads, thus we assume that a further module  $M$  containing these rules is added at the end of the sequence  $M_{C_1}, \dots, M_{C_n}$ .

## 5.2.2 Program Instantiation

Given as input a safe program  $P$ , a ground program  $grnd(P)$  such that  $AS(P) = AS(grnd(P))$  is produced. In detail, the process of instantiating  $P$  consists in the following steps: *i*) it is determined  $G_P$  of  $P$  obtaining a division of  $P$  into modules, *ii*) it is computed  $G_P^c$  on the basis of  $G_P$ , *iii*) analysing  $G_P$  it is selected an admissible component sequence  $(C_1, \dots, C_n)$ , *iv*) these modules are iteratively instantiated one at a time following the ordering  $(C_1, \dots, C_n)$ . During this process, the procedure stores the ground atoms generated into a set  $S \subseteq B_P$ , i.e a subset of the Herbrand Base of  $P$ , initialized to the set of facts in  $P$ ,  $EDB(P)$ . We refer to  $S$  also as the set of *significant* atoms.

In order to ground one module,  $M_{C_i}$  of  $P$  for  $i \in \{1, \dots, n\}$  we distinguish between exit and recursive rules. In particular, each exit rule is instantiated just one time, while recursive rules are grounded multiple times. For each exit rule  $r$ , the ground instances generated of  $r$ , are added to  $grnd(P)$ , whereas their head atoms are added to  $S$ . Then, recursive rules are repeatedly processed on the basis of a semi-naïve evaluation schema [124]. In order to avoid the

generation of the same ground instances multiple times, at each iteration  $n$  only the significant information derived during iteration  $n - 1$  is used.

More in detail, let  $i$  be the current iteration:  $S$  contains atoms produced up to iterations  $i - 2$ , while two additional sets,  $\Delta S$  and  $\mathcal{N}S$ , consist of atoms computed during the iterations  $i - 1$  and  $i$ , respectively. The underlying idea is that the information derived during iteration  $i$  are taken into account for iteration  $i + 1$ . Initially,  $\Delta S$  and  $\mathcal{N}S$  are set to  $\emptyset$ . At each iteration: *i*)  $\Delta S$  is set to  $\mathcal{N}S$ , and  $\mathcal{N}S$  is assigned to  $\emptyset$ , *ii*) each recursive rule  $r$  in  $M_{C_i}$  is grounded by employing the new information in  $\Delta S$  to generate new ground instances, while the head atoms of such instances are added into  $\mathcal{N}S$ , *iii*)  $\Delta S$  is added to  $S$ , since we already dealt with it and so that in the next iteration we will consider the current  $\mathcal{N}S$  as the new  $\Delta S$ . The iterations are stopped as soon as  $\mathcal{N}S = \emptyset$ , thus no new information has been derived.

Intuitively, this instantiation procedure allows to dynamically compute extensions of predicates; head atoms resulting from a rule instantiation immediately become members of the domains for the next iteration, even during the instantiation of the same recursive component.

**Example 5.2.2.** To illustrate how the procedure works, consider the problem *Reachability*, shown in Section 3.2. In particular, the encoding is composed by an exit rule:  $reachable(X, Y) :- edge(X, Y)$ . that states that a vertex  $b$  is directly reachable from a vertex  $a$ , if there is an edge from  $a$  to  $b$ , and a second recursive rule:  $reachable(X, Y) :- edge(X, U), reachable(U, Y)$ . that states that a vertex  $b$  is transitively reachable from a vertex  $a$ , if there is a path from  $a$  to  $b$ .

The program is composed by one component  $C$  containing the only *IDB* predicate  $reachable/2$ , and  $M_C$  contains both rules. The instantiation of  $M_C$  is performed by first evaluating the exit rule on the set  $S$ , initially containing the input edges. Assuming that  $S = \{edge(1, 2), edge(2, 3), edge(3, 4), edge(3, 5)\}$  three ground instances are produced:

$$\begin{aligned} reachable(1, 2) &:- edge(1, 2). \\ reachable(2, 3) &:- edge(2, 3). \\ reachable(3, 4) &:- edge(3, 4). \\ reachable(3, 5) &:- edge(3, 5). \end{aligned}$$

The ground atoms  $reachable(1, 2)$ ,  $reachable(2, 3)$ ,  $reachable(3, 4)$  and  $reachable(3, 5)$  are added to  $S$  and the evaluation of the recursive rule starts with  $\Delta S = \mathcal{N}S = \emptyset$ . The first iteration is performed, using the atoms in  $S$ , and the following rules are produced:

$$\begin{aligned} reachable(1, 3) &:- edge(1, 2), reachable(2, 3). \\ reachable(2, 4) &:- edge(2, 3), reachable(3, 4). \\ reachable(2, 5) &:- edge(2, 3), reachable(3, 5). \end{aligned}$$

Then,  $reachable(1, 3)$ ,  $reachable(2, 4)$  and  $reachable(2, 5)$  are added to  $\mathcal{N}S$ , and  $S$  remains the same since  $\Delta S$  is still  $\emptyset$ . Another iteration starts,  $\Delta S = \mathcal{N}S$  and  $\mathcal{N}S = \emptyset$ . To avoid duplicate rules, for the recursive predicate  $reachable/2$ , only the ground atoms in  $\Delta S$  are used, producing:

$$\begin{aligned} reachable(1, 4) &:- edge(1, 2), reachable(2, 4). \\ reachable(1, 5) &:- edge(1, 2), reachable(2, 5). \end{aligned}$$



Now,  $\mathcal{NS} = \{reachable(1,4), reachable(1,5)\}$  and  $S = S \cup \Delta S$ . Another iteration is performed, again  $\Delta S = \mathcal{NS}$  and  $\mathcal{NS} = \emptyset$ . This time nothing new can be produced, and  $S = S \cup \Delta S$ . The instantiation of the recursive rule terminates as well as the instantiation of the problem, because there are no further components to be examined.

### 5.2.3 Rule Instantiation

The process of determining the ground instances of a rule represents the key point of the whole instantiation. The function `INSTANTIATERULE` of Figure 1 exemplifies the essential steps to be performed to accomplish this process.

Let  $p/n$  be a predicate, we denote as  $I_{p/n}$  the (ground) *extension* of  $p/n$ , containing all its ground instances. Let  $l$  be a literal over the predicate  $p/n$ , with a slight abuse of notation we denoted as  $I_l$ , the (ground) extension of  $l$  as the extension of  $p/n$ , i.e.  $I_l = I_{p/n}$ .

Given a rule  $r$ , a substitution  $\theta : var(r) \mapsto U_P$  is *valid* for  $r$  if for every positive literal  $l$  occurring in  $B^+(r)$ ,  $l\theta \in I_l$  holds. The set of all valid substitutions for each rule  $r$  is logically equivalent to the set of its Herbrand instances [55]. Thus, it is possible to discard a priori any substitution mapping a positive body literal  $l$  to a ground instance of  $l$  which is not in  $I_l$ . Furthermore, since the rule is safe, in our restricted syntax, each variable occurring either in a negative literal or in the head of the rule appears also in some positive body literal. Consequently, the literals in  $B^+(r)$  bind all the variables in  $var(r)$ . Intuitively, the safety condition restricts the set of possible values for variable substitutions in a semantically “valid” way.

The function `INSTANTIATERULE` takes as input a safe rule  $r$  to be instantiated, and the extension of each literal  $l \in B(r)$ , as a set  $I_l$ , and outputs a set of total and valid substitutions,  $S$ , for  $r$ . Since positive literals are in charge of binding variables, we assume that  $B(r)$  is ordered in a way that any negative literal always follows the positive literals binding its variables by means of the sub-procedure `ORDERBODY`. Next, `INSTANTIATERULE` stores the body literals  $l_1, \dots, l_m$  into an ordered list  $B = (null, l_1, \dots, l_n, last)$  and starts the computation of the substitutions for  $r$ . To this end, it maintains a variable  $\theta$ , initially set to  $\emptyset$ , representing, at each step, a partial substitution for  $var(r)$ . For each literal  $l_i \in B(r)$ , after that its body has been reordered, we denote as  $BoundVar(l_i)$  the set of variables occurring in any literal that precedes  $l_i$  in  $B(r)$  (if  $i = 1$ ,  $BoundVar(l_i) = \emptyset$ ), and by  $BindVar(l_i)$  the set of variables that occurs for the first time in  $l_i$ , i.e.  $BindVar(l_i) = var(l_i) \setminus BoundVar(l_i)$ .

At each iteration of the while loop, by using function *Match*, we try to find a match for a literal  $l_i$  with respect to  $\theta$ , in other words, we apply  $\theta$  to  $l_i$  and look for an instantiation of  $l_i\theta$  that matches an atom in  $I_{l_i}$ . More precisely, we look in  $I_{l_i}$  for a ground instance  $g$  which is consistent with the assignments for the variables in  $BoundVar(l_i)$ , and then use  $g$  in order to extend  $\theta$  to the variables in  $BindVar(l_i)$ ; note that, if  $BindVar(l_i) = \emptyset$ , this task simply consists in checking whether  $\theta$  is a valid substitution for  $l_i$ . If no ground atom matches, then we backtrack to the previous literal in the list, otherwise we consider two cases: if there are further literals to be evaluated, then we continue with the next literal in the list; otherwise,  $\theta$  encodes a (total) valid substitution and is thus added to the output set  $S$ . Even in this case, we backtrack to find another solution.

---

The function INSTANTIATERULE

---

```

function INSTANTIATERULE(r : Rule, I1, ..., In : SetOfInstances) : SetOf-
VariableSubstitutions
  var θ : VariableSubstitution, B : ListOfLiterals, l : Literal,
  S : SetOfVariableSubstitutions
  θ = ∅
  /* reorder literals in the body */
  ORDERBODY(r)
  /* return an ordered list of the body literals (null, l1, ..., ln, last) */
  B ← BODYTOLIST(r)
  l ← l1
  S ← ∅
  while l ≠ null do
    if MATCH(l, Ii, θ) then
      if l ≠ last then
        l ← NEXTLITERAL(l)
      else
        /* θ is a total substitution for the variables of r */
        S ← S ∪ {θ}
        /* look for another solution */
        l ← PREVIOUSLITERAL(l)
        θ ← θ |BindVar(l)
      end if
    else
      l ← PREVIOUSLITERAL(l)
      θ ← θ |BindVar(l)
    end if
  end while
  return S
end function

```

---

The function MATCH

---

```

function MATCH(l : Literal, Ii : SetOfInstances, var θ : VariableSubstitu-
tion) : Boolean
  var g : GroundLiteral
  if BindVar(l) = ∅ then
    return ISVALID(θ, l, Ii)
  else
    /* take a ground instance g from Ii, if any */
    while GETINSTANCE(Ii, g) do
      if EXTEND(g, θ) then
        return true
      end if
    return false
  end while
  end if
end function

```

---

Furthermore, typically in modern grounders the set of generated ground instances undergoes to a further simplification step, in which each instance is examined and possibly simplified or even eliminated. In particular, body literals over solved predicates are already known to be true in any answer set, and thus can be safely dropped. Moreover, instance containing in their ground body some negative literal over a solved predicate already known to be false are removed. Intuitively, in this case the rule is trivially satisfied because the conjunction of body literals is always false in any answer set, and thus it does not contribute to the semantics of the ground program.

**Example 5.2.3.** To clarify this process, consider the following rule,  $r_1$ :

$$a(X) \mid b(Z) : - c(X, Z), d(Z, Y), \text{not } e(Y, Z).$$

Assume that the set of extensions are the following:

$$c(1, 2). d(2, 1). d(2, 3). e(1, 1).$$

Initially,  $\theta = \emptyset$ . Suppose that the order of literals in the body is not changed, and thus BODYTOLIST returns this list: (*null*,  $c(X, Z)$ ,  $d(Z, Y)$ , *not*  $e(Y, Z)$ , *last*). Clearly, another correct order would be:  $d(Z, Y)$ ,  $c(X, Z)$ , *not*  $e(Y, Z)$ , while any order in which *not*  $e(Y, Z)$  does not appear after  $c(X, Z)$  and  $d(Z, Y)$  would not be correct.

Next, the function starts by looking for a ground atom matching with  $c(X, Z)$ . Therefore  $c(X, Z)$  is matched with  $c(1, 2)$  and  $\theta = \{X = 1, Z = 2\}$ . Then,  $d(Z, Y)$  is taken into account, so we look for an instance in its extension matching with the partial substitution  $\theta$ , i.e such that  $Z$  corresponds to 2. There are two instances complying with this mapping, suppose we firstly retrieve  $d(2, 1)$ , thus  $\theta = \{X = 1, Z = 2, Y = 1\}$ . We proceed to the next literal *not*  $e(Y, Z)$ . Note that since it is negative, because of the processing ordering imposed and because of safety,  $BindVar(\text{not } e(Y, Z)) = \emptyset$ , moreover by definition,  $\theta$  remains valid (recall that the validity of a substitution depends just on positive literals). At this point, we go to the next literal, reaching *last*, thus  $\theta$  encodes a total valid substitution for  $r_1$ , so it is saved into the set  $S$ . Essentially, we produced the ground rule:

$$a(1) \mid b(2) : - c(1, 2), d(2, 1), \text{not } e(1, 2).$$

Then, we backtrack to  $d(Z, Y)$  and from  $\theta$  we remove  $Y$  which is in  $BindVar(d(Z, Y))$ , so  $\theta$  becomes  $\{X = 1, Z = 2\}$ . Now, we retrieve the other instance satisfying  $\theta$ , which is  $d(2, 3)$ ,  $\theta = \{X = 1, Z = 2, Y = 3\}$  and once again we jump to *not*  $e(Y, Z)$ . Applying the same reasoning above,  $\theta$  is again valid and total, thus we obtained the ground rule:

$$a(1) \mid b(3) : - c(1, 2), d(2, 3), \text{not } e(3, 2).$$

The process goes on, by backtracking again to  $d(2, Y)$ , and then to  $c(X, Z)$ , because there are no more matches for  $d(2, Y)$ . Given that also no further matches are possible for  $c(X, Z)$ , the instantiation of  $r_1$  terminates.

Eventually, supposing that  $r_1$  together with the input facts constitutes a program  $P_1$ , the two instances obtained can be simplified as follows:

$$\begin{aligned} a(1) \mid b(2). \\ a(1) \mid b(3). \end{aligned}$$

Indeed,  $c(1, 2)$ ,  $d(2, 1)$  and  $d(2, 3)$  are facts, and so by definition they are always true in every answer set; while *not*  $e(1, 2)$  and *not*  $e(3, 2)$  are literals over the solved predicate  $e/2$  which are always true. Intuitively, since there is no rule allowing to derive instances for  $e/2$ , i.e with  $e/2$  in head, there is no chance that  $e(1, 2)$  and  $e(3, 2)$  are derived, thus they are always false while their negation is always true.

### 5.2.4 Dealing with Linguistic Extensions

The instantiation process described so far can be adapted to handle the grounding of linguistic extensions.

In particular, it is needed to redefine the behaviour of the sub-functions ORDERBODY and MATCH. Concerning, the function ORDERBODY, given as input rule  $r$ , we require that its body reordered complying with a stronger condition: each literal  $l$  in  $B(r)$  always follows a possible set of saviours for it selected among the literals in  $B(r)$ . Furthermore, the function has to rearrange literals within choice and aggregate elements in  $r$ , if any, by applying the same criterion. More in detail, suppose that  $C$  is a conjunction of literals featured in an aggregate or choice element: each literal  $l'$  in  $C$  is placed after a set of possible saviours for  $l'$  in  $C$ . On the other hand, the function MATCH has to perform specific operations that depend on kind of literal at hand.

#### Arithmetic Terms

Let  $t_1 \diamond t_2$  be an arithmetic term, appearing in a literal  $l$  of a rule  $r$ . Because of the way in which literals are rearranged in the body by the function ORDERBODY, we know that when we have to ground  $t_1 \diamond t_2$ , a mapping for  $var(t_1)$  and  $var(t_2)$  has already been added to  $\theta$ . Thus, we simply instantiate them by applying the current partial substitution  $\theta$  to  $t_1$  and  $t_2$ . Next, as in Section 2.2.1 the arithmetic term is arithmetically evaluated in the standard way.

**Example 5.2.4.** Consider the rule  $r_1$ :

$$a(X) \mid b(Y) :- c(X, Y), d(X + Y, Z), e(Z + 1).$$

Suppose that the input facts are:

$$c(1, 1). d(1, 1). d(2, 1). e(2).$$

There is just one correct ordering for body literals:  $c(X, Y), d(X + Y), e(Z + 1)$ . The instantiation of  $r_1$  starts with  $\theta = \emptyset$ . Then,  $c(X, Y)$  is matched with  $c(1, 1)$ , thus  $\theta = \{X = 1, Y = 1\}$ . At this point, the function jumps to  $d(X + Y, Z)$ : firstly  $\theta(d(X + Y, Z))$  yields to  $d(2, Z)$ , then we look for matching instances in  $I_{d_2}$ . The only instance we can pick is  $d(2, 1)$ , thus  $\theta = \{X = 1, Y = 1, Z = 1\}$ . The function goes to  $e(Z + 1)$ ,  $\theta(e(Z + 1))$  yields to  $e(2)$ . Consequently, we obtained the following ground instance for  $r_1$ :

$$a(1) \mid b(1) :- c(1, 1), d(2, 1), e(2).$$

No further instance can be derived: we first backtrack to  $d(X + Y, Z)$ , then to  $c(X, Y)$  and the instantiation terminates.

### Built-in Atoms

Let  $a$  be a built-in atom of form  $t_1 \triangleright t_2$ , occurring in a rule  $r$ . Similarly to the case of arithmetic terms, because of the required ordering, its instantiation is obtained as  $a\theta$ . Moreover, if it is an assignment built-in and  $\text{var}(t_1) \subseteq \text{BoundVar}(\text{var}(a))$ , then  $a$  binds  $t_2$ , thus after that  $\theta$  has been applied to  $t_1$ ,  $\theta$  is updated by mapping  $t_2$  to  $t_1\theta$ . Similarly, if  $\text{var}(t_2) \subseteq \text{BoundVar}(\text{var}(a))$ , then  $t_1$  is mapped to  $t_2\theta$ .

**Example 5.2.5.** Consider the rule  $r_1$ :

$$a(X, Z) :- b(X, Y), X < Y, Z = X + 1.$$

where, in particular  $Z = X + 1$  is an assignment built-in. Assume that the input facts are:

$$b(1, 1), b(1, 2).$$

Here, for the body there are two possible orderings:  $b(X, Y), X < Y, Z = X + 1$  or  $b(X, Y), Z = X + 1, X < Y$ . Suppose it is selected the first one. The instantiation of  $r_1$  starts with  $\theta = \emptyset$ . Then,  $b(X, Y)$  is matched, and we assume that it is firstly retrieved the instance  $c(1, 1)$ , thus  $\theta = \{X = 1, Y = 1\}$ . At this point, the function jumps to  $X < Y$  obtaining  $1 < 1$ ,  $\theta$  remains the same as there are no bind variables. Next, the function goes to  $Z = X + 1$ ,  $\theta(X + 1)$  yields to  $1 + 1$ , which is arithmetically evaluated to 2, hence  $\theta = \{X = 1, Y = 1, Z = 2\}$ , and the following ground instance is obtained:

$$a(1, 2) :- b(1, 1), 1 < 1, Z = 2.$$

Then, the function goes back to  $X < Y$ , and then to  $b(X, Y)$ . It is retrieved the instance  $b(1, 2)$ , and propagating it as previously showed we obtain the ground instance:

$$a(1, 2) :- b(1, 2), 1 < 2, Z = 2.$$

Going bak again, no further instance is produced and the instantiation of  $r_1$  stops. Moreover, some simplifications may be applied as follows: (i) if a ground rule features a built-in which is trivially not satisfied (like  $1 < 1$  in the first ground rule reported above) the rule can be dropped; (ii) on the other hand, whenever a ground rule contains a built-in always satisfied (like  $1 < 2$  in the second ground rule generated in our example) it can be safely removed from the body.

### Aggregate Literals

Regarding aggregates, the instantiation of a rule containing aggregate literals can be performed by first computing a global substitution, applying it to aggregates and then properly grounding them. More formally, let  $r$  be a rule of form  $H :- B, ag.$ , where  $H$  represents the head of  $r$ ,  $B$  is a possibly empty conjunction of non-aggregate literals, and  $ag$  is an aggregate atom of form  $\#af\{e\} \triangleright t$ , containing an aggregate element  $e$  of form  $\{T : C\}$ , where  $T$  is a sequence of terms and  $C$  is a sequence of naf-literals. The instantiation of  $r$  consists in evaluating the instantiation of the literals in  $B$ , thus in computing a partial and valid substitution  $\theta$  that assigns values to global variables in  $\text{var}_g(r)$ . Then, the conjunction  $C\theta$  is instantiated by using the extensions of predicates appearing

in  $C$ . Under the assumption that aggregates are not recursive, all extensions of these predicates are definitely available. Thus, the following ground aggregate elements are generated:  $\{V\theta_1 : C\theta_1\theta; \dots, V\theta_n : C\theta_n\theta\}$ , where each  $\theta_i$  for  $i \in \{1, \dots, n\}$  is a possible local substitution for the local variables in  $var_l(e)$ .

Let us now extend the case above, supposing that instead  $ag$  contains more than one aggregate element. Essentially, the process illustrated is applied to each of them separately. In detail, suppose that  $ag$  contains the aggregate elements  $\{T_1 : C_1; \dots; T_m : C_m\}$ , then we obtain  $\{V_1\theta_{1_1} : C_1\theta_{1_1}\theta; \dots, V_1\theta_{n_1} : V_1\theta_{n_1}\theta; \dots; V_m\theta_{1_m} : C_m\theta_{1_m}\theta; \dots, V_m\theta_{n_m} : C_m\theta_{n_m}\theta\}$ . Consequently, a ground instance  $gag$  of  $ag$  is obtained as  $\#af\{V\theta_1 : C\theta_1\theta; \dots, V\theta_n : C\theta_n\theta\} \triangleright t\theta$ . In case  $ag$  is a negative literal, the process is the same and  $gag$  will be negative as well.

Furthermore, if  $ag$  is an assignment atom (see Section 2.4),  $t$  is a variable term and  $ag$  is a binder for  $t$ , thus when grounding  $ag$ ,  $\theta$  does not contain a mapping for  $t$ . At this point, the aggregate function  $\#af$  is computed over each possible subset of the set  $\{V_1\theta_{1_1}, \dots, V_1\theta_{n_1}, \dots, V_m\theta_{1_m}, \dots, V_m\theta_{n_m}\}$ . Let  $min$  and  $max$  be the minimum and maximum values, respectively, obtained from this computation. We generate an instance for  $ag$  for each possible value  $i$ , for  $i \in [min, max]$ , updating  $\theta$  by assigning to  $t$  the value  $i$ . Eventually, in case  $r$  contains more aggregates, each of them is instantiated as done for  $ag$ .

**Example 5.2.6.** As an example, consider the rule  $r_1$ :

$$a(Z) : -b(X), \#sum\{Y : c(X, Y), d(Y)\} = Z, e(Z).$$

Suppose that the input facts are:

$$b(1). c(1, 1). d(1). c(1, 2). d(2). e(1).e(2).$$

Firstly, we observe that  $var_g(r_1) = \{X, Z\}$ . The only valid processing ordering is  $(null, b(X), \#sum\{Y : c(X, Y), d(Y)\} = Z, e(Z), last)$ . After matching  $b(X)$ ,  $\theta = \{X = 1\}$ , then the procedure instantiates the aggregate,  $\theta(\#sum\{Y : c(X, Y), d(Y)\} = Z)$  leads to  $\#sum\{Y : c(1, Y), d(Y)\} = Z$ , while, the instantiation of the aggregate element is  $\{1 : c(1, 1), d(1); 2 : c(1, 2), d(2)\}$ . Moreover, the aggregate is an assignment:  $Z$  is bounded by it, so the function generates every possible subset of  $\{1, 2\}$ :  $\emptyset, \{1\}, \{2\}, \{1, 2\}$ , and for each of them computes the sum of its elements. The minimum value is 0 obtained with  $\emptyset$ , while the maximum sum is 3 computed over the elements of the set  $\{1, 2\}$ . So,  $Z$  can assume four values: 0, 1, 2, 3. Each one corresponds to an instance of the aggregate. In particular, the function performs a first match on the aggregate updating  $\theta$  as  $\{X = 1, Z = 0\}$ , then proceeds to  $e(Z)$ , but  $e(0)$  does not exist, thus it jumps back to the aggregate.  $\theta$  is updated as  $\{X = 1, Z = 1\}$ , and this time the function tries to match  $e(1)$ , that exists, so it is generated the instance:

$$a(1) : -b(1), \#sum\{1 : c(1, 1), d(1); 2 : c(1, 2), d(2)\} = 1.$$

Once again, it goes back to the aggregate assigning  $Z$  to 2, and returns to  $e(Z)$  matching the atom with  $e(2)$ , and obtaining:

$$a(1) : -b(1), \#sum\{1 : c(1, 1), d(1); 2 : c(1, 2), d(2)\} = 1.$$

Then, it goes again back to the aggregate,  $Z$  is assigned to 3, but  $e(3)$  does not exist. The function backtracks to the aggregate, and since all possible matches have been performed, it goes back to  $b(X)$ . Also for this atom no further match is possible, and the function stops.

### Choice Atoms

As far as concerns choice atoms, they are instantiated after that the body is grounded. More in detail, let  $r$  be a rule  $\{a : C\} : - B$ , where  $C$  is a conjunction of naf-literals,  $\{a : C\}$  is a choice atom composed by a single choice element, and  $B$  representing the body of  $r$  is a conjunction of literals. Firstly,  $B$  is processed obtaining a partial valid substitution  $\theta$  mapping the variables in  $B(r)$ . Then, the conjunction  $C\theta$  is grounded by using the extensions of predicates appearing in  $C$ . Thus, the head choice atom is grounded as:  $\{a\theta_1\theta : C\theta_1\theta; \dots, a\theta_n\theta : C\theta_n\theta\}$ , where each  $\theta_i$  for  $i \in \{1, \dots, n\}$  is a possible substitution for the variables in  $var(C)$ . In case, the head choice atom of  $r$  be composed of multiple choice elements, each one is grounded exactly as illustrate above.

**Example 5.2.7.** As an example, consider the rule  $r_1$ :

$$\{a(X, Y) : b(Y)\} : - c(X).$$

Suppose that the input facts are:

$$b(1). b(2). c(1). c(2).$$

For the body of  $r_1$  there are two possible valid substitutions:  $\theta_1 = \{X = 1\}$  and  $\theta_2 = \{X = 1\}$ . Each of them leads to an instance for  $r_1$ . Let us firstly consider  $\theta_1$ :  $\theta_1(\{a(X, Y) : b(Y)\})$  yields to  $\{a(1, Y) : b(Y)\}$ . Then,  $b(Y)$  is expanded with matching instances retrieved from  $I_{b_1}$ , obtaining:  $\{a(1, 1) : b(1); a(1, 2) : b(2)\} : - c(1)$ . Thus, an instance for  $r_1$  is:

$$\{a(1, 1) : b(1); a(1, 2) : b(2)\} : - c(1).$$

With a very similar reasoning, with  $\theta_2$  it is generated the other instance:

$$\{a(2, 1) : b(1); a(2, 2) : b(2)\} : - c(2).$$

### Constraints

Concerning strong and weak constraints, as anticipated, their instantiation can be performed after the evaluation of all rules. In addition, instantiating a weak constraints  $w$ , requires that the terms in its weak specification  $W(w)$  are grounded. Since the safety condition imposes that  $var(W(w)) \subseteq Safe(w)$ , the instantiation of the body literals also provides a substitution for  $var(W(w))$ .

**Example 5.2.8.** The instantiation of strong constraint is quite straightforward, so let us illustrate the instantiation of the weak constraint  $r_1$ :

$$:\sim a(X, Y), b(Y, Z). [X@1, Y, Z]$$

Suppose that the input facts are:

$$a(1, 1). a(1, 2). b(1, 2). b(2, 2).$$

The body of  $r_1$  can be instantiated via two different valid substitutions:

- $\theta_1 = \{X = 1, Y = 1, Z = 2\}$ , obtaining:  $a(1, 1), b(1, 2)$ ;
- $\theta_2 = \{X = 1, Y = 2, Z = 2\}$ , obtaining:  $a(1, 2), b(2, 2)$ .

By applying  $\theta_1$  to  $W(r_1)$  we obtain the ground rule:

$$:\sim a(1,1), b(1,2).[1@1, 1, 2]$$

whereas, with  $\theta_2$ , we generate the instance:

$$:\sim a(1,2), b(2,2).[1@1, 2, 2]$$

### 5.3 Optimizations

Over time, the ASP grounders released have introduced effective techniques to improve their performance. Notably, many of them are inherited from the database field.

The *dynamic magic sets* [4] is geared towards the optimization of query answering over logic programs. The aim is to rewrite the input program for identifying a subset of the program instantiation which is sufficient for answering the query. The restriction of the instantiation is obtained by means of additional fresh “magic” predicates, whose extensions represent relevant atoms w.r.t. the query. Extending the original Magic Sets defined for Datalog, the Dynamic Magic Sets technique, specifically conceived for disjunctive programs, inherits the benefits provided by standard magic sets and permits to leverage the information provided by the magic predicates also during the nondeterministic answer set search.

Further techniques have been introduced to specifically optimize the rule instantiation. As we already observed, essentially it requires to evaluate the relational join of the positive body literals, thus the processing ordering of literals in the body is a key issue for the efficiency of the instantiation procedure, just like for join computation in the database field. For instance, the *DLV* instantiator implements a *body reordering* criterion [91]. A thorough discussion of the impact of body ordering on rule instantiation is reported in Chapter 7, where we present a series of body ordering strategies designed on the basis of different heuristics with the intent of maximizing the benefits stemming from optimizations intervening in the rule instantiation task.

Taking inspiration from indexing strategies of databases, indexed data structures have been introduced to optimize the retrieval of ground instances from extensions. In [39] it is illustrated a main-memory indexing technique adopted in the *DLV* instantiator. In particular, this technique relies on indexed data structures, computed during the evaluation and only if they can really be exploited, and such that extensions are indexed on a single argument. In this work, we introduce novel indexing strategies, reported in Chapter 6; which resulted to be more general and powerful than the one adopted in *DLV*, as we will discuss in Chapter 13.

Furthermore, grounders typically do not employ a classical chronological backtracking schema during rule instantiation, but rather they leverage on a *backjumping algorithm* [111]. In particular, given a rule  $r$  to be grounded, this algorithm exploits both the semantical and the structural information about  $r$  to compute the “relevant” ground instances of  $r$ , avoiding the generation of “useless” instances, but fully preserving the semantic of the program. To this end, the algorithm relies on the set of *relevant variables* of  $r$ , consisting of all the variables occurring in literals over unsolved predicates together with the



variables occurring in the head of  $r$ . In this thesis, we studied a set of techniques to fully leverage the benefits deriving from the backjumping strategy, including a body ordering strategy specifically designed for the backjumping strategy (cf. Chapter 7), and ad-hoc further mechanisms (cf. Chapters 9).

## Chapter 6

# Efficient Retrieval of Ground Instances

Optimizing the retrieval of ground instances from predicate extensions is a key point for improving the performance of rule instantiation. In this chapter, we deeply analyze the issues behind the design of an indexing strategy. In detail, in Section 6.1 we formalize two different data structures that can be adopted to boost the retrieval of instances, while in Section 6.2 we describe two diverse indexing strategies based on opposite principles. In Section 6.3 we draw our conclusions and define an indexing strategy that tries to comply with the outlined pros and cons.

### 6.1 Deciding Data Structures

Let  $l$  be a classical atom over a predicate  $p/n$  for  $n > 0$ ,  $I_{p_n}$  is the (ground) extension of  $p/n$  and the set  $\{1, \dots, n\}$  denotes the arguments of  $p/n$ . As shown in Section 5.2.3, the function MATCH is in charge of retrieving instances from  $I_{p_n}$  according to a partial substitution  $\theta$ . In absence of techniques for boosting this task, a naive approach based on a linear search through  $I_{p_n}$  should be employed. Intuitively, the bigger  $I_{p_n}$  is, the more expensive it may be. Moreover, this is even more emphasized because of the high frequency with which this task is performed when backtracking (or backjumping) on literals to instantiate a rule body. Thus, indexing strategies heavily rely on the design of proper data structures stored in main memory and employed to speed up the retrieval task. In the following sections, we define two types of indexing structures relying on *hash maps* in different modalities.

We briefly recall that a *hash map* is a data structure implemented as an associative array, that is a structure that maps keys to values. Indeed, a hash map associates to each stored object a *key*, which can be computed by means of a proper *hash function* directly from the object itself. Elements with the same key are stored in internal data structures, usually referred to as *buckets* or *slots*. If the hash function associate to each element a univocal key (situation known as *perfect hashing*) each bucket contains only one element and thus accessing an object via its key in the worst case is  $O(1)$ . If multiple elements are associated to the same key, then in the average case looking-up for an element is  $O(1)$ ; clearly,

in the worst case all elements are in the same bucket and so the complexity of the look-up is  $O(n)$ , where  $n$  is the total number of elements in the map.

### 6.1.1 Generalized Indices

As first proposal, we present a general and flexible type of data structure.

Initially, the extension of  $I_{p_n}$  is stored in a linear one-dimensional *array-based* structure. An array is an elementary data structure, where objects are stored in continuous memory locations. This choice is motivated by the consideration that this disposition in memory makes the task of iterating through elements faster than in other linear data structure where data might be sparse in memory, such as for instance *linked list* [43].

Then, sparse secondary indices implemented as *hash maps* are associated to  $I_{p_n}$  and effectively employed to gather matching instances. Let  $\langle 1, \dots, k \rangle$  with  $k \leq n$  be an ordered tuple of arguments, that is a tuple such that for each  $a \in \{1, \dots, k\}$  and  $b \in \{1, \dots, k\}$  if  $a < b$ , then  $b$  follows  $a$  in the tuple. Let  $C$  be the set of all the distinct tuples appearing as  $\langle 1, \dots, k \rangle$  arguments of some instance in  $I_{p_n}$ . A *generalized index* for  $p/n$  is a hash map  $M_{p_n}$  that associates to each  $\langle c_1, \dots, c_k \rangle \in C$  (the key of the map) to a collection of instances  $A \subseteq I_{p_n}$  having as  $\langle 1, \dots, k \rangle$  arguments  $\langle c_1, \dots, c_k \rangle$ .

**Example 6.1.1.** As a running example, let us consider the predicate  $a/3$ , whose extensions is composed of the following ground atoms:

$$a(1, 2, 3). a(2, 2, 3). a(1, 2, 4). a(1, 3, 1). a(2, 3, 1). a(3, 4, 5).$$

A *generalized index* for  $a/3$  on its second and third argument can be graphically represented as in Figure 6.1.

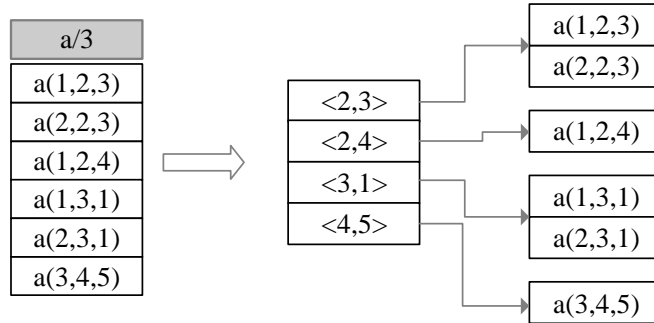


Figure 6.1: An example of *generalized index*

By means of indices, we are now able to efficiently determine matching instances. Let  $r$  be a rule, whose body consists of the literals  $\{l_1, \dots, l_m\}$  with  $m > 0$ . Suppose that the procedure ORDERBODY orders the body as  $l_1, \dots, l_m$ . Recall that for each literal  $l_i$ , we denote by  $BoundVar(l_i)$  the set of variables occurring in any literal that precedes  $l_i$  in the ordered body, that essentially represents the set of variables in  $l$  for which is present a mapping in the current partial

substitution  $\theta$ . An argument  $i$  of  $l$  for  $i \in \{1, \dots, n\}$  is said to be *indexable* if it is either a ground term or a non-ground term  $t$  such that  $\text{var}(t) \subseteq \text{BoundVar}(l)$ . Therefore after the arithmetical evaluation of the indexable arguments of  $l$ , it is selected according to a proper heuristic a tuple of indexable arguments, and an index on such arguments is employed to retrieve matching instances. A basic heuristic consists in selecting all the indexable arguments; more sophisticated heuristics might determine a subset of them considering several factors (cf. Section 6.3). Importantly, since the index is implemented by means of a hash map, the average complexity of the look up operation for a fixed key is constant time.

Moreover, in case that there are no indexable arguments we need to retrieve the whole extension  $I_{p_n}$ , thus we can simply iterate through the array-based structure containing all instances; essentially, this is the reason why we selected a structure on which iteration is fast.

**Example 6.1.2.** As an example, let us consider the following rule  $r_1$ :

$$a(X, Y, Z, W) :- b(X, Y), c(X, Z), d(X, Y, Z + Y, W).$$

To efficiently instantiate  $r_1$ , suppose that the body is processed according to the ordering:  $b(X, Y), c(X, Z), d(X, Y, Z + Y, W)$  we can employ indices according to available indexable arguments. Thus, for the atom  $b(X, Y)$  it is not possible to use any index, because there are no indexable arguments; for the atom  $c(X, Z)$ , the only indexable argument is the first; eventually, the atom  $d(X, Y, Z + Y, W)$  features three indexable arguments, namely the first, the second and the third, and so every combination of these three arguments might be chosen.

### 6.1.2 Single-Double Indices

As a more specific variant of the general index type reported above, we herein propose a further type of index that allows to retrieve matching instances in a twofold modality, according to one indexable argument or two.

More in detail, initially to each ground term  $t$  it is associated a unique integer *identifier*  $id(t)$ ; these mappings are stored into a hash map. Let  $\langle i, j \rangle$  with  $i \leq n$  and  $j \leq n$  be a pair of distinct arguments of  $p/n$ , i.e. such that  $i \in \{1, \dots, n\}$ ,  $j \in \{1, \dots, n\}$  and  $i \neq j$ . Let  $C_i$  (and  $C_j$ ) be the set of all distinct identifiers of the ground terms appearing as  $i$ -th (resp.  $j$ -th) argument of some instance in  $I_{p_n}$ . A *single-double index* for  $p/n$  is a hash map  $SDM_{p_n}$  that associates each  $a \in C_i$  to another hash map mapping each  $b \in C_j$  to the collection of instances  $A \subseteq I_{p_n}$  having as  $i$ -th argument  $a$  and as  $j$ -th argument  $b$ .

The underlying data structure is based on nested hash maps: the intent is to permit the usage of such an index to retrieve instances on the basis of the first argument  $i$  or by considering both arguments  $i$  and  $j$ . In particular, suppose that during the instantiation it is needed to retrieve matching instances for the atom  $l$ , and assume to have at disposal a single-double index on the arguments  $\langle i, j \rangle$ . If only the argument  $i$  is indexable, let  $t$  be ground term featuring as  $i$ -th after that the current partial substitution  $\theta$  has been applied to  $l$ , then by entering in the hash map with the key  $id(t)$ , and iterating through all the instances in the nested map we retrieve all instances having  $t$  as  $i$ -th argument. In case both arguments  $i$  and  $j$  are indexable, in addition let  $u$  be ground term featuring as  $j$ -th argument after that the current partial substitution  $\theta$  has been applied to  $l$ , then by accessing in the external hash map with the key  $id(t)$  and

in the nested hash map with the key  $id(u)$ , we retrieve all instances having as  $i$ -th argument  $t$  and as  $j$ -th argument  $u$ . Notably, by using identifiers as keys, we enable perfect hashing, hence each look-up operation for a fixed key is  $O(1)$  also in the worst case.

We remark that also in this case whenever there are no indexable arguments we iterate through the array-based structure containing all instances without using an index.

**Example 6.1.3.** Recall the extension of the predicate  $a/3$  of Example 6.1.1. A *single-double index* for  $a/3$  on the arguments  $\langle 3, 2 \rangle$  can be seen as depicted in Figure 6.2.

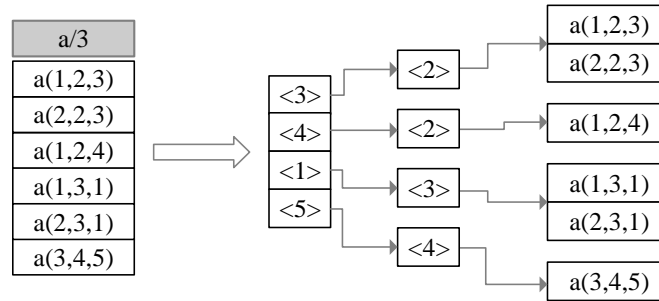


Figure 6.2: An example of *single-double index*

Intuitively, the perfect hashing of this type of index yields to better performance with respect to a generalized index created also on two arguments, and sometimes, may also be better than generalized indices created on more than two arguments, as our experiments will evidence in Section 11.1.

In addition, some consideration that may arise are why two arguments and not just one, i.e. a similar version but without a nested map, or conversely why not enabling further nesting levels, obtaining single-double-triple indices, for instance. We designed single-double indices as efficient data structures to be employed in the majority of practical situations, motivated by the consideration that, typically, predicates with arity greater than three are less frequently employed, and one nesting level is a sufficient compromise. The evaluation of the aforementioned possibilities has been done experimentally, and we evicted that single indices are not enough in several situations, while single-double indices improves performance without particular drawbacks, thanks also to the twofold modality in which they can be used. On the other hand, enabling indices with more nested maps, showed limited advantages and a proportional increase in memory usage w.r.t. the number of arguments employed, and thus the nesting level. Therefore, we suggest a hybrid strategy that relies on single-double indices for the majority of situations and enables generalized indices whenever indices over more than two arguments may be preferable.

## 6.2 Deciding an Indexing Strategy

After the decision of data structures to be employed, a further aspect to consider is when indices have to be created, and on which arguments. In this section, we outline two indexing strategies that create indices at different times during the instantiation process, describing the arising advantages and disadvantages.

### 6.2.1 Over-Generation Indexing Strategy

The *over-generation indexing strategy* is based on the generation of all distinct indices that could be obtained according to the underlying data structure adopted.

In particular, assuming to use the generalized indices of Section 6.1.1, then for each predicate  $p/n$  it is created an index on every possible ordered tuple of arguments  $\langle 1, \dots, k \rangle$  with  $1 \leq k \leq n$ , hence  $((n-1) \cdot n) + 1$  indices.

On the other hand, supposing to employ the single-double-argument indices described in Section 6.1.2, for each predicate  $p/n$  it is required the creation of  $n!/(n-2)!$  indices, since it is created an index on every distinct couple of arguments.

**Example 6.2.1.** For instance, let us come back to the predicate  $a/3$  of Example 6.1.1. According to this approach, when generalized indices are employed we would generate a different index for each of the following tuple of arguments:  $\langle 1 \rangle$ ,  $\langle 2 \rangle$ ,  $\langle 3 \rangle$ ,  $\langle 1, 2 \rangle$ ,  $\langle 1, 3 \rangle$ ,  $\langle 2, 3 \rangle$ ,  $\langle 1, 2, 3 \rangle$ ; for a total of 7 indices. On the other hand, when single-double indices are adopted as underlying data structures, then we would have 6 different indices, one for each of the following pair of arguments:  $\langle 1, 2 \rangle$ ,  $\langle 1, 3 \rangle$ ,  $\langle 2, 1 \rangle$ ,  $\langle 2, 3 \rangle$ ,  $\langle 3, 1 \rangle$ ,  $\langle 3, 2 \rangle$ .

The generation of these indices could be done before that the actual instantiation starts, initially on the basis of EDB. Then, as soon as predicate extensions are computed during the instantiation, indices are updated and filled up with newly generated instances. Intuitively, we can avoid the creation of indices for predicates that do not appear in rule bodies, since this means that it is not required to retrieve instances for them. The major disadvantages is that this strategy might lead to the creation of unnecessary indices, since the instantiation can exploit solely indices on indexable arguments. Moreover, the larger an extension is, the more expensive it will be the cost of creating an index on it; hence, creating a useless index might lead to a not negligible overhead; in addition, for medium/large extension sizes, the risk of running out of memory is quite high. However, by creating all possible indices, the selection of the index to be employed in each situation is eased. Intuitively, when there are several indexable arguments, we may design heuristic criteria that decide the best index to be employed by actively analysing all possible indices and measuring their quality.

### 6.2.2 On-Demand Indexing Strategy

The *on-demand indexing strategy* consists in creating “useful” indices *on-the-fly* directly during the instantiation.

More in detail, let  $r$  be a rule be grounded, and  $\{l_1, \dots, l_m\}$  be the literals in its body with  $m > 0$ . Suppose that the procedure ORDERBODY orders the body

as  $l_1, \dots, l_m$ . At the time in which the function `INSTANTIATERULE` looks for a match on a classical atom  $l_i$  for  $i \in \{1, \dots, m\}$ , if there is at least an indexable argument, it is *on-demand* created an index. In particular, if it is adopted a generalized index, such index is created over a subset of the indexable arguments of  $l_i$ . In case it is employed a single-double index, such index has to be created on a pair of arguments such that the first argument in the pair is indexable, and the second one might be arbitrary selected even if not indexable.

**Example 6.2.2.** Coming back to Example 6.2.1, instead of building all those indices, adopting an on-demand strategy we would generate just the indices actually employed during the instantiation process to retrieve instances. For instance, suppose to have the following rules:

$$\begin{aligned} r_1 &: b(X, Z, W) :- c(X, Y), a(Y, X, W). \\ r_2 &: d(Z, W) :- e(X), a(Z, X, W). \end{aligned}$$

Suppose that the procedure `ORDERBODY` preserves the order of body literals reported above. In this situation, if generalized indices are employed, in order to retrieve instances for the predicate  $a/3$  during the instantiation of  $r_1$ , it may be either created an index on the arguments  $\langle 1, 2 \rangle$ , or on the first argument only, or on the second argument only, depending on the heuristic criterion adopted. For  $r_2$  there is no choice, only the second argument is indexable, thus it is created an index on it. Instead, when single-double indices are used, when the rule  $r_1$  is instantiated it may be created an index on the pairs of arguments  $\langle 1, 2 \rangle$ ,  $\langle 2, 1 \rangle$ ,  $\langle 1, 3 \rangle$  or  $\langle 2, 3 \rangle$ . When grounding  $r_2$  it may be created an index the pairs  $\langle 2, 3 \rangle$  or  $\langle 2, 1 \rangle$ . Notably, if for  $r_1$  it is adopted an index on the arguments  $\langle 2, 1 \rangle$ , it can be reused also when grounding  $r_2$ .

Intuitively, the main advantage of this on-demand strategy is that indices are created only if needed, in contraposition with the over-generating strategy that builds all possible indices. Nevertheless, whenever there are several indexable arguments, the heuristics that select the best index have to be designed in a more blind way by comparing possible employable indices on estimations of their quality.

### 6.3 Balanced On-Demand Indexing Strategy

Summing up all pros and cons that arose we propose an indexing strategy, that tries to optimize all factors considered so far. As anticipated, when multiple arguments are indexable several indices might be employed, designed according to different heuristic criteria. For this choice, we should consider both the effects and the causes. In particular, we have to find a balance between the cost of creating a new index and the benefits deriving from its creation, as creating an index on a large extension might be expensive especially if such an index cannot be effectively exploited. Thus, in such a condition the possibility of creating only exploitable indices and reusing an index more than one time, in case the involved predicate occurs within multiple rules, might be preferable, while using a custom index for each of its occurrence might be unfeasible in practice.

A further important aspect to consider is the strict correlation between the ordering of body literals and indexing strategies. Firstly, body ordering influences the choice of indexing arguments, since only arguments containing bound

variables are actually indexable. Consequently, selecting indices on-demand after body ordering and during the rule instantiation might lead to a greedy creation of indices, and whenever their creation costs are not negligible it can be paid a significant overhead. Hence, the greedy nature of on-demand strategies has to be taken into account by considering the cost of creating an index before to proceed at its creation.

Following an over-generation strategy, an opposed approach might consist in deciding the indices to be employed before the actual instantiation process starts, so that it is possible to minimize the creation of new indices. Hence, with respect to a pure over-generation strategy, here instead of generating all possible indices, we restrict the creation to a subset of them which can be effectively employed. Consequently, during the rule instantiation bodies should be reordered by ensuring the usage of indices previously decided. In other words, the body ordering strategy has to be aware of these constraints, which in turn may not lead to a good ordering, because we are now deciding the indexing strategy without considering the correlation with body ordering. In addition, at the time in which indices are decided, the extensions of IDB predicates are not definitely available. Hence, these decisions have to be made according to a heuristical estimation of what the extensions will actually be.

Essentially, while the former approach gives priority to the body ordering strategy and creates indices greedily, the latter focuses on the indexing strategy and adapts the body ordering strategy accordingly, but has the not avoidable drawback of requiring a precise estimation of predicate extensions to be effective.

To comply with these points, preferring an on-demand approach, hereafter it is proposed an approach that we called *balanced on-demand indexing strategy*, that limits the greedy nature of a pure on-demand strategy by adopting single-double-argument indices as underlying data structures build over a well-motivated subset of indexable arguments.

To this end, for each predicate  $p/n$  and for each argument  $i \in \{1, \dots, n\}$  this strategy stores the set of different values that appear as  $i$ -th argument of some instance in  $I_{p_n}$ , a measure which is also known as *selectivity* of the argument  $i$ . We refer to these sets as *dictionary* of  $p/n$ . The dictionary is dynamically computed during the instantiation process: before that the instantiation starts it is initialized with the values appearing in the EDB, then, as soon as a new ground atom is derived, the dictionary is updated.

Consequently, when multiple arguments are indexable for a predicate  $p/n$ , this strategy selects the two indexable arguments that feature the highest number of different values in  $I_{p_n}$ . If just an argument is indexable, it is selected as first indexing argument, and as second argument it is selected the one with the highest selectivity among all other arguments. In a database oriented wording, these are more likely to represent a primary key for  $I_{p_n}$ , i.e. nearer to designate a small group of instances, ideally just one.

This choice is motivated by the consideration that in general indices created on a smaller number of arguments, are less specific and more versatile, hence reusable multiple times. In addition, in practice benefits stemming from an index created on a larger number of arguments become evident in extreme situations, for instance in case of huge extensions, when grouping them on two arguments leads to large collections, and thus adding further arguments is preferable even if expensive, as we will later on show in Section 11.1.



**Example 6.3.1.** Let us consider the following program  $P_1$ :

$$\begin{aligned} a(W) &: - b(X, Y), c(Z), d(X, Y, Z, W). \\ c &(1..10). \\ b &(1..10, 1..10). \\ d &(1..300, 1..200, 1..100, 1..10). \end{aligned}$$

with a total of 60,000,000 instances. Let us discuss how the dictionary of  $d/4$  is composed. For the first argument, admissible values ranges from 1 to 300; for the second argument, from 1 to 200; for the third argument, from 1 to 100; and, for the fourth argument, from 1 to 10.

Suppose to build a single-double index on the first and second arguments, then for each tuple  $\langle a_1, a_2 \rangle$ , for  $a_1 \in \{1, \dots, 300\}$  and  $a_2 \in \{1, \dots, 200\}$  there are 1,000 corresponding instances in  $I_{d_4}$ . Similarly, by selecting the first and the third arguments, we get 2,000 instances for each tuple  $\langle b_1, b_2 \rangle$  with  $b_1 \in \{1, \dots, 300\}$  and  $b_2 \in \{1, \dots, 100\}$ , and with the second and the third arguments there are 3,000 instances for each tuple  $\langle c_1, c_2 \rangle$  for  $c_1 \in \{1, \dots, 200\}$  and  $c_2 \in \{1, \dots, 100\}$ . Intuitively, by means of the former combination a smaller number of instances is considered each time a match is performed on  $d(X, Y, Z + Y, W)$  and thus according to our criterion, this is the best combination to be chosen. If instead we assume to use a single argument index, the first one should be preferred. Indeed, for each distinct key  $k$  for  $k \in \{1, \dots, 300\}$  there are 200,000 instances in  $I_{d_4}$ .

Eventually, with a generalized index on all the three arguments, we have 10 corresponding instance for each tuple  $\langle e_1, e_2, e_3 \rangle$  with  $e_1 \in \{1, \dots, 300\}$ ,  $e_2 \in \{1, \dots, 200\}$  and  $e_3 \in \{1, \dots, 100\}$ : this set of arguments is nearer to represent a primary key for  $I_{d_4}$ .

In Section 11.1 we will come back to this example, and show an experimental analysis of all possibilities.

## Chapter 7

# Body Ordering

As discussed in Section 5.2.3, the procedure `ORDERBODY` invoked by the `INSTANTIATERULE` function is intended to guarantee the correct instantiation of rule bodies: it rearranges literals according to a basic strategy so that each literal is placed in a “safe” position.

However, more importantly, the procedure `ORDERBODY` may aim at rearranging literals in the rule bodies in order to find an optimal execution ordering for the join operations. Indeed, a good ordering dramatically affects the overall computation time, as will show in Section 11.

In practice, determining all possible ordering and then choosing the optimal one is not feasible, since there may be several possibilities. Therefore, typically, grounders rely on greedy algorithms. Taking inspiration from techniques developed in the database setting, this problem has been originally studied in the *DLV* grounder [91], with the definition of the *Combined* criterion, the body ordering strategy implemented in the first version of *DLV*. Starting from this criterion, we studied different variants to enhance its effectiveness. A first problematic addressed consists in adapting the criterion to *ASP-Core-2*, since the criterion was mainly focused on classical atoms. Secondly, we studied the interaction among body orderings and strictly related optimizations, such as indexing strategies and the backjumping technique. Consequently, by considering these aspects we defined a set of ordering strategies designed on the basis of different heuristics.

In the remainder of the Chapter, after recalling the basic ideas behind the *Combined* criterion (Section 7.1), we describe the new developed criteria. In particular, Section 7.2 presents how the *Combined* criterion has been extended to *ASP-Core-2* describing the *Combined*<sup>+</sup> strategy, while Section 7.3 illustrates some of its variants tailored on the interaction of body ordering strategies with crucial optimizations such as indexing strategies and backjumping techniques.

### 7.1 The Basic Strategy

Given a rule  $r$ , the *Combined* criterion processes  $B(r)$  generating a new version of it, namely  $B'(r)$ , where the literals have been rearranged according to a greedy algorithm.

Essentially, at each step, the “best” literal, which is determined relying on statistics over the involved predicates, is placed, and then the employed statistics

are updated. The “best” literal is the one that minimizes a formula based on two factors: one is a measure of how much the choice of a literal reduces the search space for possible substitutions, and the other takes into account the variables for which the literal is a binder. The intuition behind this choice is that preferring literals with already bound variables might lead to detect possible inconsistencies quickly.

This mechanism is formalized below. Given a classical atom  $l$  over a predicate  $p$ , for each variable  $X \in \text{var}(l)$ , the selectivity of  $X$  in  $l$  (i.e. the number of distinct values), denoted  $V(X, l)$ , corresponds to the number of tuples in the projection of  $X$  over the ground extension of the predicate  $p$ . Moreover, for each variable  $X$  appearing in  $B^+(r)$ , the *active* domain of  $X$  is defined as  $\text{dom}(X) = \max_{l \in B^+(r)} V(X, l)$ . Suppose that  $B'_i$  is the reordered (partial) body at step  $i$ , and  $\text{var}(B'_i)$  is the set of variables within  $B'_i$ , which is also denoted as the *set of bound variables* in  $B'_i$ . For each classical atom  $l \in (B^+(r) \setminus B'_{i-1})$ , its score at step  $i$ , denoted  $s_i(l)$  is estimated as:

$$s_i(l) = \frac{T(B'_{i-1} \bowtie l)}{\prod_{X \in Z} \text{dom}(X)} \cdot \prod_{Y \in (\text{var}(B'_{i-1}) \cap \text{var}(l))} \frac{V(Y, l)}{\text{dom}(Y)^2}$$

where  $Z$  is the set of variables that  $l$  has in common with some other classical atom occurring in  $B^+(r)$ , and the value  $T(B_{i-1} \bowtie l)$  indicates the estimated size of the intermediate join between the literals in  $B_{i-1}$  and  $l$  and is inductively defined under the assumption that values are distributed uniformly over their domains.

In detail, for a classical atom  $c$  composed by a predicate  $q/m$ ,  $T(c)$  denotes the size of the extension of  $q$ , i.e.  $T(c) = |I_{q_m}|$ . Let  $a$  and  $b$  be two classical atoms, then  $T(a \bowtie b)$  represents the size of the join between  $a$  and  $b$  and is defined as:

$$T(a \bowtie b) = T(b) \cdot \prod_{X \in (\text{var}(a) \cap \text{var}(b))} \frac{V(X, a)}{\text{dom}(X)}$$

Consequently, the join  $B_{i-1} \bowtie l$ ,  $T(B_{i-1} \bowtie l)$  is estimated as:

$$T(B'_{i-1} \bowtie l) = T(l) \cdot \prod_{X \in (\text{var}(B'_{i-1}) \cap \text{var}(l))} \frac{V(X, B'_{i-1})}{\text{dom}(X)} \quad (7.1)$$

Let  $l_i$  the atom chosen at step  $i$  then, for each variable  $X \in \text{var}(B'_i)$ , the selectivity of  $X$  in  $B'_i$  is defined as:

$$V(X, B'_i) = V(X, B'_{i-1}) \cdot \frac{V(X, l_i)}{\text{dom}(X)}$$

if  $X \in \text{var}(B'_{i-1})$ ; otherwise when  $X \notin \text{var}(B'_{i-1})$ , then  $V(X, B'_i) = V(X, l_i)$ .

Once that all classical atoms in  $B^+(r)$  are inserted in  $B'(r)$ , the other kind of literals present in  $B(r)$  can be arbitrary placed as well in positions that ensure the correct instantiation. Eventually,  $B(r)$  is set to  $B'(r)$ .

**Example 7.1.1.** Given the following rule  $r_1$ :

$$a(X, Y, Z, T) :- b(X, Y), c(X, Z), d(X, Y, Z, T), X < Z + 1.$$

let us assume that the extensions of the involved predicates are composed as follows:

$$\begin{aligned} &b(1..5, 1..50). \\ &c(1..10, 1..10). \\ &d(1..5, 1..5, 1..2, 1..2). \end{aligned}$$

Hence,  $T(b(X, Y)) = 250$ ,  $T(c(X, Z)) = 100$ ,  $T(d(X, Y, Z, T)) = 100$ ,  $dom(X) = 10$ ,  $dom(Y) = 50$ ,  $dom(Z) = 10$ ,  $dom(T) = 2$ . At step 0, the first atom is selected mainly according to the first factor of the formula reported above because  $B'(r_1)$  is empty and  $var(B'_0) = \emptyset$ , therefore:

$$\begin{aligned} s_0(b(X, Y)) &= 250/(10 \cdot 50) = 0.5 \\ s_0(c(X, Z)) &= 100/(10 \cdot 10) = 1 \\ s_0(d(X, Y, Z, T)) &= 100/(10 \cdot 50 \cdot 10 \cdot 2) = 0.004 \end{aligned}$$

The smallest score is obtained by the atom  $d(X, Y, Z, T)$ , which is inserted as first atom:  $B'_1 = d(X, Y, Z, T)$ . Moreover,  $T(B'_1) = T(d(X, Y, Z, T)) = 100$ ,  $V(X, B'_1) = V(X, d(X, Y, Z, T)) = 5$ ,  $V(Y, B'_1) = V(Y, d(X, Y, Z, T)) = 5$ ,  $V(Z, B'_1) = V(Z, d(X, Y, Z, T)) = 2$  and  $V(T, B'_1) = V(T, d(X, Y, Z, T)) = 2$ . Then,

$$\begin{aligned} s_1(b(X, Y)) &= \frac{250 \cdot \frac{5 \cdot 5}{10 \cdot 50}}{10 \cdot 50} \cdot \frac{5 \cdot 50}{(10 \cdot 50)^2} = 0.000025 \\ s_1(c(X, Z)) &= \frac{100 \cdot \frac{5 \cdot 2}{10 \cdot 10}}{10 \cdot 10} \cdot \frac{10 \cdot 10}{(10 \cdot 10)^2} = 0.01 \end{aligned}$$

Thus,  $B'_2 = d(X, Y, Z, T), b(X, Y)$  and eventually the last remaining atom is placed at the end, obtaining  $B'_3 = d(X, Y, Z, T), b(X, Y), c(X, Z)$ . At this point, the built-in  $X < Z + 1$  may be arbitrary placed. In particular, it can be inserted after that the variables appearing in it are bound, since  $var(X < Z + 1) = \{X, Z\}$  the built-in can be in every position after  $d(X, Y, Z, T)$ .

## 7.2 Enhancing the Basic Strategy for ASP-Core-2

Herein we present the *Combined*<sup>+</sup> criterion, aiming at defining more precise statistics to properly place in the ordered body other kinds of literal, such as negative naf-literals, aggregate literals, choice atoms, and built-in atoms.

In particular, negative classical naf-literals are added as soon as their variables are bound: at each step  $i$ , after that the best classical atom is placed, each remaining negative naf-literal  $n_l \in (B(r) \setminus B'_i)$  is analyzed, and if  $var(l) \subseteq var(B'_i)$ , then  $n_l$  is added to  $B'_i$ . Indeed, negative naf-literals do not increase the set of bound variables, hence placing them as soon as possible permits to faster recover inconsistencies.

As far as regards aggregate literals, contrarily they are placed as late as possible. Indeed, as discussed in Section 5.2.4, their instantiation is, in general, more involved than the instantiation of other types of literal, since it requires the instantiation of the conjunctions of literals inside its aggregate elements. Thus, after that all other literals are placed, each aggregate literal  $a_l \in (B(r) \setminus B'(r))$  is iteratively examined and added to  $B'(r)$ . In detail, let  $a_l = af\{e_1, \dots, e_n\} \triangleright t$  be a positive aggregate literal: if  $\triangleright$  corresponds to  $=$  and  $t$  is a variable term,

$a_l$  can be placed as soon as  $(var_g(a_l) \setminus var(t)) \subseteq var(B')$ ; otherwise,  $a_l$  can be added as soon as  $var_g(a_l) \subseteq var(B')$ . Intuitively, in the former case the aggregate represents an assignment, thus the variable  $t$  is a safe variable. In case  $a_l = not\ af\{e_1, \dots, e_n\} \triangleright t$ , i.e. it is a negative aggregate literal, it is added as soon as  $var_g(a_l) \subseteq var(B')$ .

In addition, after that the rule body has been ordered, the criterion is applied also to literals within choice and aggregate elements in  $r$ , if any. Let  $C$  be a conjunction of literals featured in an aggregate or choice element, the literals in  $C$  are properly reordered in order to optimize the evaluation of the join operations among the literals in  $C$  by inductively applying the *Combined*<sup>+</sup> strategy to the conjunction  $C$ .

### 7.2.1 Handling Built-in Atoms

After defining how to deal with the other type of literals, we need to decide how to deal with built-in atoms, since they requires a more thorough attention: similarly to aggregate literals they might bind variables (i.e. be assignments) or represent comparison operations, however, in general, their instantiation is simpler and so, it is performed faster. Thus, their ordering assumes a more strategic meaning.

**Example 7.2.1.** As a running example, let us consider the rule:

$$a(X, Y, Z) :- b(X), b(Y), b(Z), Z < X, Z < Y.$$

Here the *Combined* criterion has no useful information for the selection of a proper order for literals  $b(X)$ ,  $b(Y)$ ,  $b(Z)$ . Indeed, given that they all have the same predicate and that the considered measures do not take into account the comparisons built-in, each possible permutation of these atoms may be indiscriminately chosen.

However, their ordering can significantly affect performance, since their variables are involved in the comparisons  $Z < X$ , and  $Z < Y$ . In particular,  $Z$  is involved in two comparisons whose meaning is that it should be less than both  $X$  and  $Y$ , hence during the instantiation when substituting  $Z$  with a ground term, it is sufficient to consider just terms satisfying these comparisons.

To overcome such situations, the *Combined*<sup>+</sup> criterion tries to insert built-in atoms as soon as possible, and improves the statistics employed in the *Combined* criterion relying on linear interpolation techniques to determine how much the *actual* search space for variable substitutions is influenced by the presence of built-in atoms. Remarkably, to be compliant with the original criterion, the new criterion preserves the assumption of uniform distribution.

Given a positive literal  $l$  over the predicate  $p$ , and a variable  $X \in var(l)$  which features a domain consisting of integer numbers, we define  $min(X, L)$  as the minimum value,  $max(X, L)$  as the maximum value over such domain. Please notice that the assumption over integer domains is not restrictive, as it is always possible to define a proper mapping from a generic term to a number, according to the *total order* of terms defined in the ASP-Core-2 standard (cf. Section 2.2).

Let us now describe how the *Combined*<sup>+</sup> criterion estimates and updates the measures at each step  $i$ .

For each candidate literal  $l$ , for each variable  $X \in l$ , the criterion makes a pre-estimation  $V_e(X, B_i)$  of  $V(X, B_i)$  if  $X$  is an *evaluable* variable, that is

if the following conditions hold: (i)  $X$  is not bound (i.e.  $X \notin \text{var}(B'_{i-1})$ ), and (ii)  $X$  appears in  $B(r)$  in some comparisons of the form  $X \prec Y$ , where  $\prec \in \{<, <=, >, >=, =, <>\}$  and  $Y$  is either a ground term or a bound variable such that  $Y \in \text{var}(B'_{i-1})$ .

Initially,  $V_e(X, B'_i) = V(X, l)$ ; then, it is recursively updated taking into account each comparison involving  $X$  and satisfying the last condition, as described next.

If  $X$  is involved in a comparison in the form  $X < C$  or  $X <= C$  where  $C$  is a ground term and  $C_{num}$  is the reduction of  $C$  to a numeric value, then:

$$V_e(X, B'_i) = (V_e(X, B'_i) - 1) \cdot \left( \frac{(C_{num} - \min(X, l))}{(\max(X, l) - \min(X, l))} \right).$$

If the comparison is of the form  $X > C$  or  $X >= C$ , then it is equivalent to  $C < X$  or  $C <= X$ , respectively; hence:

$$V_e(X, B'_i) = V_e(X, B'_i) - \left( (V_e(X, B'_i) - 1) \cdot \frac{(C_{num} - \min(X, l))}{(\max(X, l) - \min(X, l))} \right).$$

On the other hand, when the  $X$  is compared to a variable, such as in  $X \prec Y$ , where  $\prec \in \{<, <=, >, >=\}$ , then a constant value  $Y_{num}$  for  $Y$  is first estimated, and then the formulae above are applied; in particular, if  $\prec \in \{<, <=\}$  then  $Y_{num} = \max(Y, l_1)$ , while if  $\prec \in \{>, >=\}$  then  $Y_{num} = \min(Y, l_1)$ , where  $l_1$  is the literal binding  $Y$  in  $B_{i-1}$ .

Finally, if  $X$  appears in a comparison in the form  $X = Z$ , where  $Z$  is either a ground term or a variable, then:

$$V_e(X, B'_i) = \frac{1}{V_e(X, B'_i)}$$

while if the comparison is like  $X \neq Z$  then:

$$V_e(X, B'_i) = V_e(X, B'_i) - \left( \frac{1}{V_e(X, B'_i)} \right).$$

In these last two cases, indeed, the algorithm tries to estimate the probability that the value that will bound the variable  $X$  will actually be equal (or different) to the one estimated. In addition, if  $Z$  is a constant, say  $C$ , it is possible to determine with certainty if the value  $C$  is admissible or not for  $X$ , by computing if  $C$  belongs to the domain of  $X$ ; otherwise, if  $Z$  is a variable, it is still possible to estimate the probability that the values assignable to  $Z$  are admissible by estimating a ground value  $Z_{num}$  as  $1/V(Z, B'_{i-1})$ .

After computing these pre-estimations, the criterion has a better measure of how much the comparisons can influence the variables selectivities, and thus it can properly select the next literal to add in  $B'_{i-1}$ , say  $l'$ , so that  $B'_i = B'_{i-1} \cup l'$ . Eventually, for each variable  $X \in l'$ ,  $V(X, B'_i)$  is updated in an unchanged way with respect to the old criterion, unless  $X$  is *evaluable*: in this case  $V(X, B'_i) = V_e(X, B'_i)$ .

**Example 7.2.2.** Coming back to the running example, let us assume that the available instances for predicate  $b$  currently consists of the set  $b(1..100)$ ; then, the *Combined*<sup>+</sup> algorithm arranges the *rule* as follows:

$$a(X, Y, Z) :- b(X), b(Z), Z < X, b(Y), Z < Y.$$

Initially,  $B'_0 = b(X)$ , and the next chosen literal is then  $b(Z)$ : the algorithm prefers  $b(Z)$  over  $b(Y)$ , as it computes that  $V(Z, B'_1) = 99$  and  $V(Y, B'_1) = 100$ . Intuitively,  $b(X)$  and  $b(Z)$  are here involved in a cross-product operation, thus the algorithm estimates that, in general,  $Z$  ranges over values that are smaller than the ones assigned to  $X$  in 99/100 cases, except for when they have assigned the same value. At this point, we have  $B'_1 = b(X), b(Z)$ , and the comparison built-in  $Z < X$  can be safely selected: indeed, when adding a comparison literal its variables have to be bound in order to correctly evaluate them. Subsequently,  $b(Y)$ , and finally  $Z < Y$ , can be chosen. It is worth noticing that an equally preferable ordering would be, in this case:

$$a(X, Y, Z) :- b(Y), b(Z), Z < Y, b(X), Z < X.$$

Indeed, both  $Z < X$  and  $Z < Y$  have the same influence on the selectivity of  $Z$ .

## 7.3 Extensions

Hereafter, we present three extensions of the *Combined*<sup>+</sup> criterion. Firstly, we define two different criteria based on indexing strategies and the backjumping machinery, then we provide a description of criterion driven by both optimizations that combines these two variants.

### 7.3.1 Indexing-driven Ordering

As anticipated in Section 6, body ordering and indexing strategies are strictly related. To grasp the intuition behind this correlation, let us consider the following example.

**Example 7.3.1.** Consider the rule  $r_1$ :

$$a(X, Y, Z) :- b(X, Y), c(X, Z).$$

Let us assume that the extensions of involved predicates are the following:

$$\begin{aligned} &b(1..5, 1..40, 000). \\ &c(1..2, 500, 1..80). \end{aligned}$$

thus, the extensions of  $I_{b_2}$  and  $I_{c_2}$  consist of 200,000 ground instances each. Moreover, for each tuple  $(i)$  with  $i \in \{1, \dots, 5\}$  there are 40,000 corresponding instances in  $I_{b_2}$ ; while for each tuple  $(j)$  with  $j \in \{1, \dots, 2500\}$  there are 80 instances in  $I_{c_2}$ . Evidently, there are two possible orderings for the rule body: *i*)  $b(X, Y), c(X, Z)$ ; *ii*)  $c(X, Z), b(X, Y)$ . Selecting the ordering *i*) no index can be used on  $b(X, Y)$ , while an index on the first argument can be used on  $c(X, Z)$ ; whereas the situation is opposed for the ordering *ii*). Analyzing the number of

performed matches: in case *i*) is  $400,000 \cdot 5 \cdot 80$ , while in case *ii*), it increases to  $400,000 \cdot 5 \cdot 40,000$ . Therefore, the ordering *i*) is more advantageous. Indeed, even if the size of the extensions coincides, data are more uniformly distributed in  $I_{c_2}$ , thus indexing on the first argument  $c/2$  is preferable than indexing on the same argument  $b/2$ . Notably, by construction, the *Combined* and the *Combined*<sup>+</sup> criteria are completely unaware of the differences between the two orderings.

To comply with these situations, we extended the *Combined*<sup>+</sup> strategy, obtaining an ordering strategy, namely *Combined*<sub>I</sub><sup>+</sup>, that takes into account the influences of body ordering on indexing strategies.

Given a rule  $r$ , at step  $i$  for each literal  $l \in B(r) \setminus B'_{i-1}$  we estimate the effect of adding  $l$  as next literal by considering the quality of indices that might be used for the other remaining literals  $\{l_1, \dots, l_n\} \in B(r) \setminus B'_{i-1}$ .

Formally, let  $V$  be the set of variables in  $\text{var}(l) \cup \text{var}(B'_i)$ . For each literal  $l' \in \{l_1, \dots, l_n\}$ , for which there is at least an indexable argument, we estimate the quality of the best index available for  $l'$  in case  $l$  is added as next literal, denoted  $q_i(l', l)$ , according to the indexing strategy employed, as:

$$q_i(l, l') = 1 - \frac{\prod_{X \in \text{idx}(l')} V(X, l')}{T(l')}$$

where  $\text{idx}(l')$  are the indexing arguments for  $l'$  selected among the arguments featuring ground terms or variables contained in  $V$ .

Essentially,  $q_i(l, l')$  is a value ranging from 0 to 1: the nearer it is to 0, the higher is the quality of the index employable for  $l'$ . Indeed, in case the indexing arguments represent a key,  $\prod_{X \in \text{idx}(l')} V(X, l') = T(l')$ , therefore  $q_i(l, l') = 0$ .

For instance, assuming that each involved predicate is indexed according to the *balanced on-demand indexing strategy* (cf. Chapter 6), supposing that  $a_1$  and  $a_2$  are the two arguments selected, then:

$$q_i(l, l') = 1 - \frac{V(a_1, l') \cdot V(a_2, l')}{T(l)}$$

Let  $q_i(l) = \prod_{l' \in \{l_1, \dots, l_n\}} q_i(l, l')$ . The score for  $l$  at step  $i$  is computed as the  $s_i(l) \cdot q_i(l)$ , where  $s_i(l)$  is the score for  $l$  at step  $i$  computed via the *Combined*<sup>+</sup> criterion.

**Example 7.3.2.** Returning to the example 7.3.1, suppose to be at step 0 and we are choosing to add  $b(X, Y)$  or  $c(X, Z)$  as first literal in a newly reordered body  $B'(r_1)$ . Then  $q_0(b(X, Y)) = q_0(b(X, Y), c(X, Z)) = 1 - 2,500/200,000 = 0.9875$ , while  $q_0(c(X, Z)) = q_0(c(X, Z), b(X, Y)) = 1 - 5/200,000 = 0.999975$ . Moreover,  $s_0(b(X, Y)) = s_0(c(X, Z)) = 200,000$ . Thus, the *Combined*<sub>I</sub><sup>+</sup> criterion decides to insert firstly  $b(X, Y)$ .

### 7.3.2 Backjumping-driven Ordering

The strategy described next has been designed for the backjumping mechanism [111] reported in Section 5.3. As already mentioned, given a (non-ground) rule  $r$ , this technique restricts its instantiation to the set of *relevant* instances. To this end, it relies on the set of *relevant* variables, denoted  $\text{RelVar}(r)$ , composed by variables occurring in literals over unsolved predicates together with the variables occurring in the head of  $r$ .



**Example 7.3.3.** Suppose we are going to instantiate the rule  $r_1$ :

$$a(X, Z) :- b(X, Z), c(X, Y).$$

where  $b/2$  and  $c/2$  are solved predicates, thus  $RelVar(r) = var(H(r)) = \{X, Z\}$ . Before grounding  $r_1$ , we can reorder its body as: *i*)  $b(X, Z), c(X, Y)$  or *ii*)  $c(X, Y), b(X, Z)$ . Notably,  $var(b(X, Z)) \subseteq RelVar(r)$ , while  $c(X, Y)$  contains the variable  $Y$  which is not relevant.

Selecting the order *i*), initially we look for a successful match on  $b(X, Z)$ , next we jump to  $c(X, Y)$  and if the match on it succeed as well, we obtained a valid and relevant substitution for  $r$ . At this point, it is useless to perform further matches on  $c(X, Y)$  since the ground terms mapped to relevant variables in the current substitution will not change: we would just update the mapping for  $Y$ . Rather, as the backjumping technique suggests, we can safely and immediately jump back to  $b(X, Z)$ , to produce a different relevant substitution. Thus, for each instance of  $b(X, Z)$  we need to consider just an instance for  $c(X, Y)$ .

To grasp the intuition behind this decision of the backjumping algorithm, suppose that as input facts we have  $b(1, 1). c(1, 2). c(1, 3). c(1, 4).$ , then according to a classical backtracking algorithm, we would produce:

$$\begin{aligned} a(1, 1) &:- b(1, 1), c(1, 2). \\ a(1, 1) &:- b(1, 1), c(1, 3). \\ a(1, 1) &:- b(1, 1), c(1, 4). \end{aligned}$$

However, after the simplification step, we obtain that these rules are semantically equivalent to the rule:

$$a(1, 1) :- b(1, 1).$$

This happens because  $Y$  is not relevant: essentially, by discerning between relevant and non-relevant variables, the backjumping algorithm is geared towards avoiding to produce the same relevant instance multiple times [111].

On the other hand, adopting the ordering *ii*), initially we look for a successful match on  $c(X, Y)$ , next we jump to  $b(X, Z)$ : at this point after the first successful match, we cannot jump immediately back, because  $b(X, Z)$  binds the relevant variable  $Z$ , thus we need to perform every possible further match. After processing all matching instances for  $b(X, Z)$  we can jump back to  $c(X, Y)$  and consider the next instance for it. Essentially, in this way for each instance of  $c(X, Y)$  we iterate through over all the instances of  $b(X, Z)$ ; therefore, the power of the backjumping algorithm cannot be completely exploited.

Consequently, body ordering might maximize the benefits stemming from the adoption of a backjumping technique. To this end, the  $Combined_B^+$  criterion enhances the  $Combined^+$  with a heuristic allowing to place literals binding relevant variables as soon as possible.

Assume to be at step  $i$  for each literal  $l \in B(r) \setminus B'_{i-1}$ , we associate to  $l$  a score, denoted  $s'_i(l)$ , computed with the following formula:

$$\frac{\prod_{X \in var(l) \cap RelVar(X)} V(X, l)}{\prod_{X \in var(l)} dom(X)^2}$$

Basically, the score belongs to the range  $[0, 1]$  and the smaller is the score the larger is the number of relevant variables that  $l$  binds. However, in general

different literals may bind the same variable, thus the formula tends to prefer literals featuring a smaller selectivity for it.

Eventually, the final score for  $l$  at step  $i$  is computed as the product of  $s'_i(l)$  and  $s_i(l)$  assigned to  $l$  at step  $i$  by the  $Combined^+$  criterion.

**Example 7.3.4.** Let us come back to the example 7.3.3, suppose that as ground extensions for the body predicates, we have:

$$\begin{aligned} &b(1..5, 000, 1..100). \\ &c(1..5, 000, 1..100). \end{aligned}$$

Hence,  $dom(X) = 5,000$ ,  $dom(Y) = 100$  and  $dom(Z) = 100$ ; in addition  $T(b(X, Z)) = T(c(X, Y)) = 500,000$ . Assume to be at step 0:  $s'_0(b(X, Z)) = (5,000 \cdot 100)/(5,000 \cdot 100)^2 = 0.000002$ , while  $s'_0(c(X, Y)) = 5,000/(5,000)^2 = 0.0002$ . Moreover,  $s_0(b(X, Z)) = s_0(c(X, Y)) = 500,000$ . Thus, from the point of view of the  $Combined$ ,  $Combined^+$ , as well as the  $Combined_I^+$  criteria there no evident advantages in preferring the ordering  $i$ , while the  $Combined_B^+$  criterion correctly prefers to add  $b(X, Y)$  as first literal in  $B'(r_1)$ .

### 7.3.3 Indexing- and Backjumping-driven Ordering

To consider the impact of both involved indices and backjumping technique, the  $Combined_{IB}^+$  criterion has been developed. Essentially, it extends the  $Combined^+$  criterion by adding heuristics coming from both the  $Combined_I^+$  and the  $Combined_B^+$  strategies.

More in detail, given a rule  $r$ , at each step  $i$  for each literal  $l \in B(r) \setminus B'_{i-1}$ , the score assigned to it is:  $s_i(l) \cdot q(i_l) \cdot s'_i(l)$ , where  $s_i(l)$  is the score for  $l$  at step  $i$  computed via the  $Combined^+$  criterion,  $q_i(l)$  is the estimation of the quality of available indices in case  $l$  is added at step  $i$  and is calculated as in the  $Combined_I^+$  criterion, while  $s'_i(l)$  is the score assigned as in the  $Combined_B^+$  criterion referring to the binding of relevant variables.

**Example 7.3.5.** It is easy to see that in Example 7.3.1 the  $Combined_B^+$  strategy has no means to select the best ordering between the possible two. The same holds if the  $Combined_I^+$  criterion is applied on the rule showed in Example 7.3.3. Therefore, by combining them the  $Combined_{IB}^+$  strategy is able to make the best choice in both cases.



## Chapter 8

# Decomposition Rewriting

Typically, the same computational problem can be encoded by means of many different ASP programs which are semantically equivalent; however, real ASP systems may perform very differently when evaluating each one of them. This behavior is due, in part, to specific aspects, that strictly depend on the ASP system effectively employed, and, in part, to general “intrinsic” aspects, depending on the program at hand which could feature some characteristics that can make computation easier or harder. Thus, often, to have satisfying performance, expert knowledge may be required in order to select the best encoding. This issue, in a certain sense, conflicts with the declarative nature of ASP that, ideally, should free the users from the burden of the computational aspects. For this reason, ASP systems tend to be endowed with proper pre-processing means aiming at making performance less encoding-dependent; intuitively, such means are of great importance for fostering and easing the usage of ASP in practice.

A proposal in this direction is *lpopt* [16], a pre-processing tool for ASP systems that rewrites rules in input programs by means of *tree decomposition* algorithms. The rationale comes from the fact that, when programs contain rules featuring long bodies, ASP systems performance might benefit from a careful split of such rules into multiple, smaller ones. However, it is worth noting that, while in some cases such decomposition is convenient, in other cases keeping the original rule is preferable; hence, a black-box decomposition like the one of *lpopt*, makes it difficult to predict whether it will lead to benefits or disadvantages.

In this chapter, we start from the *lpopt* idea and propose a method, namely SMARTDECOMPOSITION, that aims at taking full advantage from decompositions, still avoiding performance drawbacks by trying to predict the effects of rewritings. Such method is rather general, as it is intended to be embedded into different ASP systems, and customized accordingly. It analyzes each input rule before the evaluation, and decides whether it could be convenient to decompose it into an equivalent set of smaller rules, or not. Furthermore, as many decompositions may be possible for each rule, further criteria can be defined in order to select a preferred one.

In addition, we define a specific version to be embedded within the grounder we are designing, and in general, adoptable by the grounding module of an ASP system following a traditional “ground & solve” approach. The aim is to improve grounding performance: to this end, heuristic criteria have been tailored to take advantage from information that are available from within the

instantiation process. In particular, we studied a heuristic estimation based on the intrinsic cost of grounding a rule, and propose its usage to estimate the cost of applying a decomposition. Consequently, decompositions can be selected not only considering the non-ground structure of the encoding at hand, but also on the bases of the instance with which it is coupled, so that, combining an encoding  $E$  with an instance  $I$  might not produce the same decompositions if  $E$  is instead paired with a different instance  $I'$ .

The chapter is organized as follows. In Section 8.1 we first recall the notions of *hypergraph* and *tree decomposition*. Section 8.2 outline some motivations behind the introduction of our decomposition approach. Next, in Section 8.3 we provide an abstract description of the SMARTDECOMPOSITION technique, while in Section 8.4 we describe a specific version of SMARTDECOMPOSITION tailored on the instantiation process.

## 8.1 Hypergraphs and Tree Decompositions

The structure of many problems can be described by *graphs* or *hypergraphs*, and *tree decompositions* as well as *hypertree decompositions* are adopted to divide these (hyper)graphs into different parts so that the solution(s) of such problems can be obtained by a polynomial divide-and-conquer algorithm that properly exploits this division [78, 77].

**Definition 8.1.1** (Hypergraph and Graph). A *hypergraph* is a pair  $H = (V(H), E(H))$ , where  $V(H)$  denotes a set of vertices (or nodes) and  $E(H)$  a set of *hyperedges*. A hyperedge  $e \in E(H)$  is itself a set of vertices, with  $e \subseteq V(H)$ . A *graph* is a hypergraph in which all hyperedges consist of two vertices.

**Definition 8.1.2** (Tree Decomposition). A *tree decomposition* of a hypergraph  $H = (V(H), E(H))$  is a tuple  $TD = (T, \chi)$ , where  $T = (V(T), E(T))$  is a tree and  $\chi : V(T) \mapsto 2^{V(H)}$  is a function associating to each vertex (or node)  $t \in V(T)$  a set of vertices  $\chi(t) \subseteq V(H)$ , and the following conditions hold:

- for every  $e \in E(H)$  there exists a vertex  $t \in V(T)$  such that  $e \subseteq \chi(t)$ ,
- for every  $h \in V(H)$  the set  $\{t \in V(T) | h \in \chi(t)\}$  induces a connected subtree of  $T$ .

The *width* of a tree decomposition  $(T, \chi)$  is  $\max\{|\chi(t)| - 1 | t \in V(T)\}$ , i.e. the maximum  $\chi$ -set cardinality over all its vertices, while the *tree-width* of a hypergraph  $H$  is the minimum of the widths of all possible tree decompositions of  $H$ . In the following, w.l.o.g. we assume the tree  $T$  in a tree decomposition to be rooted.

The problem of determining the *tree-width* of a graph  $G$  is NP-hard. However, for each fixed natural number  $k$ , checking whether the *tree-width* of  $G$  is less or equal than  $k$ , can be done in logarithmic space [76].

## 8.2 Motivations

In order to analyse the impact of back-box decompositions on modern ASP systems, we considered two different configurations of the state-of-the-art grounder

*gringo*: (i) *gringo* in its default version, (ii) *lpopt* executed in pipeline *gringo*. These configurations have been paired with the mainstream solvers *clasp* and *wasp*. The latest available versions at the time of writing were launched: *gringo* and *clasp* 5.2.1, and *wasp* 2.1. As benchmarks, we relied the Sixth ASP Competition suite [67], which features 28 problems and 20 different instances per each. Experiments have been performed the same hardware of Chapter 11, and for memory and time limits, we allotted 15 GiB and 600 seconds for each system, per each single run. Table 8.1 reports the number of solved instances and the average running times over solved instances, per problem; “TO” stands for timeouts. The last two lines report the total number of solved instances and the average solving times over them. As for the problems *Consistent Query Answering*, *Reachability*, *Strategic Companies* and *System Synthesis* none of the executed configurations were able to solve any instance because of unsupported syntax or timeouts, we omitted these four problems in Table 8.1.

Table 8.1: Sixth ASP Competition Benchmarks – impact of *lpopt* on solving

Problems	gringo clasp		lpopt gringo clasp		gringo wasp		lpopt gringo wasp	
	#solved	time	#solved	time	#solved	time	#solved	time
Abstract Dialectical Frameworks	20	8.80	20	8.19	12	37.96	12	51.64
Combined Configuration	10	280.62	9	138.02	1	1.18	1	231.42
Complex Optimization	17	137.03	17	122.04	4	104.50	6	111.24
Connected Still Life	6	238.91	6	245.59	12	47.14	12	81.66
Crossing Minimization	6	63.06	6	63.48	19	4.09	19	5.21
Graceful Graphs	9	68.09	9	68.44	4	44.73	6	154.26
Graph Coloring	15	137.57	15	167.20	7	68.77	8	131.21
Incremental Scheduling	13	116.77	14	139.70	5	114.58	8	179.01
Knight Tour With Holes	10	15.05	10	43.58	10	113.87	10	145.06
Labyrinth	13	103.86	11	139.09	10	138.07	10	127.79
Maximal Clique	0	TO	0	TO	8	293.63	8	288.65
MaxSAT	7	43.70	7	50.05	19	100.41	19	104.59
Minimal Diagnosis	20	8.41	20	9.53	20	36.95	20	33.76
Nomistery	7	91.73	9	53.86	8	57.10	10	153.80
Partner Units	14	35.14	14	35.21	10	236.87	8	123.75
Permutation Pattern Matching	11	167.83	17	124.50	20	173.43	6	109.56
Qualitative Spatial Reasoning	19	140.97	19	141.60	16	194.72	16	195.06
Ricochet Robots	8	119.41	10	123.83	6	87.76	8	201.20
Sokoban	9	123.15	9	123.33	9	136.62	10	122.61
Stable Marriage	4	397.52	4	405.56	7	369.93	7	421.11
Steiner Tree	2	52.06	2	52.37	1	249.02	1	249.34
Valves Location Problem	16	22.18	16	40.02	16	68.68	15	41.32
Video Streaming	13	56.89	15	104.66	0	TO	0	-
Visit-all	8	17.21	8	16.98	8	60.97	8	58.44
<b>Total Solved Instances</b>	<b>257/560</b>		<b>267/560</b>		<b>232/560</b>		<b>228/560</b>	
<b>Average Time</b>	<b>93.16</b>		<b>94.11</b>		<b>111.45</b>		<b>119.00</b>	

The experimental results evidence that, in general, decomposing rules is not always convenient in terms of performance, and a black-box decomposition mechanisms by means of *lpopt*, makes it difficult to predict benefits or disadvantages. For instance, completely different outcomes are observed if we consider the two problems *Labyrinth* and *Permutation Pattern Matching* when *lpopt* preprocesses the encodings. Concerning *clasp*, enabling *lpopt* leads to a gain of 6 solved instances with *Permutation Pattern Matching*, while it costs a loss of 2 instances with *Labyrinth*. An opposed situation can be observed for *wasp*. For *Permutation Pattern Matching* without decompositions it is able to solve all the 20 instance, while with *lpopt* it solves just 6 instances. Contrarily, for *Labyrinth* *wasp* solves 10 instances in about the same time independently from the usage of *lpopt*.

Table 8.2 illustrates the impact on the grounding time, reporting grounding times and the number of grounded instances within the same time and memory

limits of *gringo* in its default version, and *lpopt* executed in pipeline *gringo*. Analogously, even if looking at the average grounding times *gringo* seems to benefit from the usage of *lpopt*, in general, the black box decomposition mechanism demonstrates a conflicting impact also on grounding performance: in some cases splitting rules is preferable, in others a notable slowdown emerges. The algorithm presented herein aims at overcoming this uncertainty. It is designed to be integrated into an ASP system, and to smartly exploit information available during the computation to predict, according to proper criteria, whether decomposing will be convenient or not; moreover, while *lpopt* randomly generates for each input rule an admissible decomposition, SMARTDECOMPOSITION can choose the most promising decomposition, among the several possible ones. The algorithm has been designed as abstract and general to allow its customization according to different purposes. In particular, since in our experimental activity conflicting behaviours emerged, suggesting that decompositions possibly should be performed differently from system to system, the aim is to allow such customizations according to the distinguishing modalities in which ASP systems carry out their computational evaluations of input programs.

Table 8.2: Sixth ASP Competition Benchmarks – impact of *lpopt* on grounding

Problem	gringo		lpopt   gringo	
	solved	time	solved	time
Abstract Dialectical Frameworks	20	2.22	20	1.63
Combined Configuration	20	14.66	20	14.78
Complex Optimization	20	8.52	20	12.27
Connected Still Life	20	0.10	20	0.10
Crossing Minimization	20	0.10	20	0.10
Graceful Graphs	20	0.21	20	0.22
Graph Coloring	20	0.10	20	0.10
Incremental Scheduling	20	28.93	20	28.91
Knight Tour With Holes	20	12.78	20	17.79
Labyrinth	20	0.55	20	0.64
Maximal Clique	20	14.23	20	32.64
MaxSAT	20	6.80	20	11.37
Minimal Diagnosis	20	2.99	20	3.35
Nomistry	20	2.41	20	1.16
Partner Units	20	0.39	20	0.41
Permutation Pattern Matching	20	124.49	20	4.67
Qualitative Spatial Reasoning	20	6.07	20	6.06
Ricochet Robots	20	0.19	20	0.21
Sokoban	20	1.17	20	1.18
Stable Marriage	20	116.43	20	119.93
Steiner Tree	20	31.57	20	31.40
System Synthesis	20	1.35	20	1.40
Valves Location Problem	20	3.78	20	3.65
Video Streaming	20	0.10	20	0.10
Visit-all	20	1.04	20	0.39
<b>Total Solved Instances</b>	<b>500/500</b>		<b>500/500</b>	
<b>Average Time</b>	<b>15.25</b>		<b>11.78</b>	

### 8.3 A Heuristic-driven Decomposition Approach

In this section we introduce SMARTDECOMPOSITION, designed to be integrated into an ASP system. In order to decompose input rules we make use of *hyper-graphs* and *tree decompositions* in order optimize the evaluation of the program

---

The algorithm SMARTDECOMPOSITION and the GENERATERULEDECOMPOSITIONS function

---

```

function SMARTDECOMPOSITION( $r$  : Rule) : RuleDecomposition
  var  $e_r$  : number,  $S$  : SetOfRuleDecompositions,  $e_S$  : Number,
   $RD$  : RuleDecomposition
   $e_r \leftarrow$  ESTIMATE( $r$ )
   $S \leftarrow$  GENERATERULEDECOMPOSITIONS( $r$ )
  if  $S \neq \emptyset$  then                                     /*  $r$  is decomposable */
     $RD \leftarrow$  CHOOSEBESTDECOMPOSITION( $S, e_r$ )
     $e_{RD} \leftarrow$  ESTIMATEDECOMPOSITION( $RD$ )
    if DECOMPOSITIONISPREFERABLE( $e_r, e_{RD}$ ) then
      return  $RD$ 
    end if
  end if
  return  $\emptyset$ 
end function

function GENERATERULEDECOMPOSITIONS( $r$  : Rule) : SetOfRuleDecompositions
  var  $HG$  : Hypergraph,  $S$  : SetOfRuleDecompositions,
   $RD$  : RuleDecomposition,  $TD$  : TreeDecomposition
   $TDS$  : SetOfTreeDecompositions
   $HG \leftarrow$  TOHYPERGRAPH( $r$ )
   $TDS \leftarrow$  GENERATETREEDecompositions( $HG$ )
  for each  $TD \in TDS$  do
     $RD \leftarrow$  TORULES( $TD, r$ )
     $S = S \cup RD$ 
  end for
  return  $S$ 
end function

```

---

at hand.

The abstract algorithm SMARTDECOMPOSITION is shown in Figure 3. In the following, we indicate as *tree decomposition* an actual tree-decomposition of a hypergraph, while with *rule decomposition* we denote the conversion of a tree-decomposition into a set of ASP rules. Given as input a (non-ground) rule  $r$ , the algorithm first heuristically computes the impact  $e_r$  of  $r$  on the computation by means of the ESTIMATE function; then, the function GENERATERULEDECOMPOSITIONS computes a set of possible decompositions  $S$ , from which CHOOSEBESTDECOMPOSITION selects the best decomposition  $RD \in S$  according to a criterion to be defined; hence, the impact of  $RD$  is estimated by ESTIMATEDECOMPOSITION, by means of an additional criterion to be defined, and it is compared to  $e_r$  in order to decide if decomposing is convenient.

The definition of the functions ESTIMATE, CHOOSEBESTDECOMPOSITION, ESTIMATEDECOMPOSITION and DECOMPOSITIONISPREFERABLE are left unimplemented, as they are completely customizable and must be defined by properly taking into account features and information available within the specific evaluation procedure of the actual ASP system in which the algorithm is being integrated into.



The function `GENERATERULEDECOMPOSITIONS` is reported in Figure 3. The function `TOHYPERGRAPH` converts the input rule into a hypergraph: such conversion can be carried out in several ways and thus should be customized in the actual implementation in order to meet the computation strategies of the system at hand.

Then,  $HG$  is iteratively analysed in order to produce possible tree decompositions, by means of the function `GENERATETREEDECOMPOSITIONS`. This function might be defined according to different heuristics; since it might take too long and also be useless to enumerate all decompositions in practice, it could be defined in order to compute only a fixed number.

Figure 4 illustrates the function `TORULES`, that, given a tree decomposition  $TD$  and a rule  $r$ , converts  $TD$  into a rule decomposition for  $r$ . At first, it iterates on the nodes of  $TD$  by means of the function `VISITDFS`, which performs a *depth-first search* and returns a list of nodes,  $v_1, \dots, v_n$  such that for  $i \in \{1, \dots, n\}$  and  $j \in \{1, \dots, i\}$ ,  $v_j$  is never a child of  $v_i$ . Each node in the list is analyzed, and a corresponding rule  $r'$  for it is generated via the function `GENERATERULE`. The process is, again, customizable, and should be defined according to the function `TOHYPERGRAPH`.

Eventually, to ensure that the generated rules can be grounded, safety is checked<sup>1</sup>. If  $r'$  is unsafe, a new rule  $r''$  is generated by function `ENSURESAFETY`. If  $UV$  is the set of unsafe variables, an atom  $a$  over a fresh predicate  $p$  containing as terms the variables in  $UV$  is added to both  $H(r'')$  and  $B(r')$ . A set of literals  $L$  binding the variables in  $UV$  is extracted from  $B(r)$  and added to  $B(r'')$ . The choice of the literals to be inserted in  $L$  is also customizable, as, in general, different combinations of literals might bind the same set of variables. It is worthwhile considering that, in principle, one could directly add  $L$  to  $B(r')$  without generating  $r''$ ; however, this might introduce further variables in  $B(r')$ , and alter the original join operations in  $B(r')$ . Eventually, both the rule  $r'$  and the potential rule  $r''$  are added to the rule decomposition representing the output.

## 8.4 Grounding-based Decomposition Rewriting

In this section we present an adaptation of the algorithm to be laid in the new grounder we are designing. The aim is to improve its grounding process, thus the heuristics and all other customizable steps will be defined in order to comply with this purpose.

Let  $r$  be the input rule to be decomposed. We assume that whenever  $r$  contains some aggregate literals or a choice atom, a rewriting process has been previously applied to it, so that these constructs are in a standardized form, in which each internal aggregate or choice element contains at most a literal. The process is better formalized in Section 9.5 and essentially consists in an adaptation of the input rule and in the introduction auxiliary rules. The reasons behind the introduction of this standardization process are summarized

<sup>1</sup>In general, due to the abstract nature of `SMARTDECOMPOSITION` we cannot assume the safety of rules generated by `GENERATERULE`, since this depends on the schemas selected for converting a rule into a hypergraph, and a tree decomposition into a set of rules. For instance, following *lpopt* schemas, at a first stage rules are generated as unsafe, and later it is applied a process to ensure their safety.

---

 Converting a tree decomposition into a rule decomposition
 

---

```

function TORULES( $TD$  : TreeDecomposition,  $r$  : Rule) : RuleDecomposition
  var  $RD$  : RuleDecomposition
   $(v_1, \dots, v_n) \leftarrow \text{VISITDFS}(TD)$ 
  for  $i = 1 \dots n$  do
     $r' \leftarrow \text{GENERATERULE}(v_i, r)$ 
    if  $\text{ISUNSAFE}(r')$  then
       $r'' \leftarrow \text{ENSURESAFETY}(r', B(r))$ 
       $RD = RD \cup r''$ 
    end if
     $RD = RD \cup r'$ 
  end for
  return  $RD$ 
end function

```

---

next. Firstly, from the software engineering point of view, the assumption that these constructs are in a standard form eases the design, and in turn the implementation, of their instantiation mechanisms. In addition, the grounding performance is less influenced by the form in which rules are given in input and we can transform them into format which is more efficient to handle. Lastly, as we will better outline in Section 9.5, by means of these rewriting processes we can almost “freely” apply all optimizations intervening in the rule instantiation function to the instantiation of these constructs. In this context, we do not have to specifically take care of how to split literals inside an aggregate element, and we can directly execute SMARTDECOMPOSITION on the auxiliary rules obtained from this rewriting process to determine whether the internal literals should be split up into multiple rules.

Let us start from the functions TOHYPERGRAPH and GENERATERULE. One of the requirement of our grounder is the support of the ASP-Core-2 language. Thus, these functions must carefully take this into account. For their definition, we rely on a schema analogous to the one adopted in *lpopt*, that we recall next.

The function TOHYPERGRAPH converts the input rule into a hypergraph as specified by the following schema. In detail, a hypergraph of  $r$  consists in a pair  $HG = (V(HG), E(HG))$  such that:

- for each variable  $X \in \text{var}(r)$ , there exists a vertex in  $V(HG)$ ;
- for each literal  $l \in B(r)$ ,
  - if  $l$  is a naf-literal, there exists a hyperedge  $e \in E(HG)$  with  $e = \text{var}(l)$ ;
  - if  $l$  is an aggregate literal, there exists a hyperedge  $e \in E(HG)$  with  $e = \text{var}_g(l)$ ;
- if  $H(r) \neq \emptyset$  ( $r$  is not a constraint), a hyperedge  $h \in E(HG)$  is constituted by the set  $\text{var}(H(r))$ ;
- if  $r$  is a weak constraint, a hyperedge  $h \in E(HG)$  is composed by the set  $\text{var}(W(r))$ .

---

Customized versions of the functions ESTIMATE and ESTIMATEDecomposition

```

function ESTIMATE( $r$  : Rule) : Number
    /* Estimate the cost of grounding a rule according to the formula of
    Section 8.4.1 */
end function

function ESTIMATEDecomposition( $RD$  : RuleDecomposition) : Number
    var  $e_{RD}$  : number
    PREPROCESS( $RD$ )
     $e_{RD} \leftarrow 0$ 
    for each  $r' \in RD$  do
         $e_{RD} = e_{RD} + \text{ESTIMATE}(r')$ 
    end for
    return  $e_{RD}$ 
end function

```

---

The function GENERATERULE is defined as follows. Let  $v_i$  be the input node of a tree decomposition, and  $r$  the input rule to be decomposed. The function outputs a rule  $r'$  such that for the head,  $H(r')$ :

- if  $v_i$  is not the root node, then let  $\text{parent}(v_i)$  be the parent node of  $v_i$ ;  $H(r')$  consists in a single predicate atom  $ha$  composed by a fresh predicate  $hp$  and by the set of variables  $(\chi(v_i) \cup \chi(\text{parent}(v_i)))$  as terms;
- if  $v_i$  is the root node, then if  $r$  is rule and is not a strong constraint,  $H(r') = H(r)$ , i.e.  $r'$  contains in its head the head atoms of the original rule to be decomposed  $r$ ; if  $r$  is a weak constraint  $W(r') = W(r)$ , i.e. the weak specification of  $r$  are added to  $r'$ ;

For the body,  $B(r')$ :

- let  $l_1, \dots, l_h$  be the literals in  $B(r)$ , and  $\text{vars} = \chi(v_i)$ ; for every  $l_i$ ,  $i \in \{1, \dots, h\}$  if  $\text{var}(l_i) \subseteq \chi(v_i)$ , then  $l_i$  is added to  $B(r')$ ;
- let  $c_1, \dots, c_m$  be the child nodes of  $v_i$ , and  $a_{c_1}, \dots, a_{c_m}$  be the head atoms of rules obtained by the processing of  $c_1, \dots, c_m$ , then  $a_{c_1}, \dots, a_{c_m}$  are added to  $B(r')$ .

The policy implemented by the ENSURESAFETY function consists in preferring a small number of literals from  $B(r)$  and classical atoms with small ground extensions. This is a difference w.r.t. the black-box usage of *lpopt* where the choice of saviour literals is less informed, since it is performed before the actual ASP computation process is started.

Figure 5 illustrates an implementation of ESTIMATE and ESTIMATEDecomposition functions. The first heuristically measures the cost of instantiating a rule, and relies on the formula illustrated in Section 8.4.1; the latter, after some pre-processing steps described in Section 8.4.2, computes the cost of the decomposition as the sum of the cost of each rule in a rule decomposition.

The functions DECOMPOSITIONISPREFERABLE and CHOOSEBESTDECOMPOSITION are depicted in Figure 6. The function DECOMPOSITIONISPREFERABLE is in charge of deciding whether the best estimated rule decomposition  $RD$  can be supposed to be preferable with respect to the original rule  $r$  by relying on

---

Customized versions of the functions `DECOMPOSITIONISPREFERABLE` and `CHOOSEBESTDECOMPOSITION`

---

```

function DECOMPOSITIONISPREFERABLE( $e_r$ : number,  $e_{RD}$ : number) :
  boolean
  var threshold: number
  threshold  $\leftarrow$  GETTHRESHOLD()
  if ( $e_r/e_{RD}$ )  $\geq$  threshold then
    return true
  end if
  return false
end function

```

```

function CHOOSEBESTDECOMPOSITION( $S$ : SetOfRuleDecompositions,  $e_r$ :
  number) : RuleDecomposition
  var  $BD$ : RuleDecomposition,  $e_{BD}$ : number,  $e_{RD}$ : number,
   $BD \leftarrow \emptyset$ ,  $e_{BD} \leftarrow e_r$ 
  for each  $RD \in S$  do
     $e_{RD} \leftarrow$  ESTIMATEDecomposition( $RD$ )
    if  $e_{BD} > e_{RD}$  then
       $e_{BD} \leftarrow e_{RD}$ 
       $BD \leftarrow RD$ 
    end if
  end for
  return  $BD$ 
end function

```

---

$e_r$  and  $e_{RD}$ , that are the estimated costs associated to  $r$  and  $RD$ , respectively. In particular, it computes the ratio  $e_r/e_{RD}$ : if  $e_r/e_{RD} \geq threshold$ , the decomposition is applied. Intuitively, when  $e_r < e_{RD}$  one might think that grounding  $r$  is preferable; nevertheless, when  $e_r$  and  $e_{RD}$  are slightly different it may be the case to prefer  $RD$  over  $r$ . Moreover, it is worth remembering that the costs are estimated, and, in particular, as it will be better discussed in Section 8.4.2, the estimate of the cost of a rule decomposition requires to estimate also the extension of some additional predicates introduced by the rewriting, thus possibly making the estimate less accurate. This leads sometimes to cases in which the decomposition is preferable even when  $e_{RD} > e_r$ . One can try to improve the estimations, in the first place; however, an error margin will always be present. For this reason, in order to reduce the impact of such issue, to obtain a more flexible method, their ratio is compared with a *threshold* value.

The function `CHOOSEBESTDECOMPOSITION` estimates the costs of all input decompositions, via the function `ESTIMATEDecomposition`, and returns the one with the smallest cost.

**Example 8.4.1.** To better clarify the process of converting a rule into a hypergraph, and the subsequent conversion of a tree decomposition into a set of rules, let us consider as an example the following rule  $r_1$ , which is part of the encoding of the problem *Permutation Pattern Matching*, belonging to the set of benchmarks of the Sixth ASP Competition [67]:

$$:- \text{pair}(K1, K2), \text{solution}(K1, E1), \text{solution}(K2, E2), E2 < E1.$$

Figure 8.1 depicts the hypergraph associated,  $HG(r_1)$  and a possible tree decomposition of it,  $TD(r_1)$ .

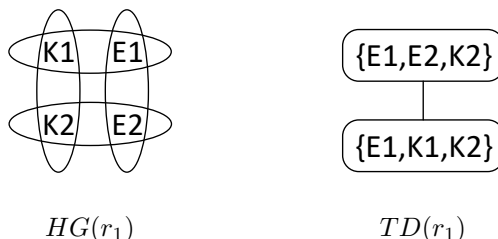


Figure 8.1: Decomposing a rule of *Permutation Pattern Matching*: the associated Hypergraph and a possible tree decomposition

The conversion of  $TD(r_1)$  via the procedure TORULES yields the following rules:

$$\begin{aligned} \text{fresh\_pred\_1}(E2, K1) &:- \text{pair}(K1, K2), \text{solution}(K2, E2). \\ &:- \text{solution}(K1, E1), \text{fresh\_pred\_1}(E2, K1), E2 < E1. \end{aligned}$$

Note that in this case the two rules are immediately obtained as safe after setting their heads and bodies, hence no auxiliary rule has been introduced for safety purposes.

**Example 8.4.2.** To illustrate the mechanism that ensure safety of rules deriving from a decomposition, let us consider the following rule,  $r_2$ :

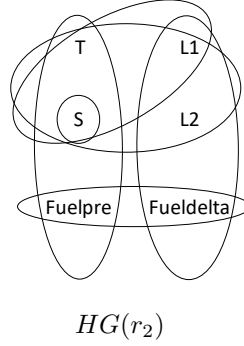
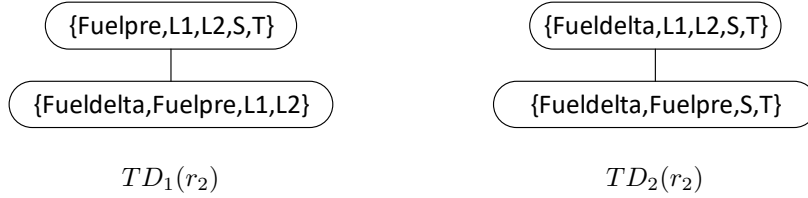
$$\begin{aligned} \text{preconditions}_d(T, L1, L2, S) &:- \text{step}(S), \text{at}(T, L1, S - 1), \\ &\text{fuelcost}(\text{Fueldelta}, L1, L2), \\ &\text{fuel}(T, \text{Fuelpre}, S - 1), \\ &\text{Fuelpre} \geq \text{Fueldelta}. \end{aligned}$$

$r_2$  has been extracted from the encoding of the problem *Nomystery*, part of the Sixth ASP Competition [67].

The conversion of  $r_2$  into a hypergraph,  $HG(r_2)$  leads to the one reported in Figure 8.2. The vertices consist in the variables in  $\text{var}(r_2)$ :  $\text{Fueldelta}$ ,  $\text{Fuelpre}$ ,  $L1$ ,  $L2$ ,  $S$ ,  $T$ . As edges, from the body literals we have:  $\{S\}$ ,  $\{T, L1, S\}$ ,  $\{T, \text{Fuelpre}, S\}$ ,  $\{\text{Fueldelta}, L1, L2\}$ ,  $\{\text{Fuelpre}, \text{Fueldelta}\}$ ; while  $\{T, L1, L2, S\}$  is the vertex stemming from  $\text{var}(H(r))$ .

Figure 8.3 depicts two distinct tree decompositions for  $r_2$ ,  $TD_1(r_2)$  and  $TD_2(r_2)$ . From  $TD_1(r_2)$  we obtain the following set of rules:

$$\begin{aligned} s_1 &: \text{fresh\_pred\_1}(\text{Fuelpre}) :- \text{step}(S), \text{fuel}(\_, \text{Fuelpre}, S - 1). \\ s_2 &: \text{fresh\_pred\_2}(\text{Fuelpre}, L1, L2) :- \text{fuelcost}(\text{Fueldelta}, L1, L2), \\ &\text{Fuelpre} \geq \text{Fueldelta}, \\ &\text{fresh\_pred\_1}(\text{Fuelpre}). \\ s_3 &: \text{preconditions}_d(T, L1, L2, S) :- \text{step}(S), \text{at}(T, L1, S - 1), \\ &\text{fuel}(T, \text{Fuelpre}, S - 1), \\ &\text{fresh\_pred\_2}(\text{Fuelpre}, L1, L2). \end{aligned}$$

Figure 8.2: Decomposing a rule of *Nomystery*: the associated HypergraphFigure 8.3: Decomposing a rule of *Nomystery*: two possible tree decompositions

Notably, to ensure the safety of  $s_2$ , the rule  $s_1$  has been generated. Indeed, the literal  $fresh\_pred\_1(Fuelpre)$  has been added to  $s_2$  to save the variable  $Fuelpre$  appearing in the built-in  $Fuelpre \geq Fuedelta$ . The body of  $s_1$  contains the literal  $fuel(\_, Fuelpre, S - 1)$  binding  $Fuelpre$ , and the literal  $step(S)$  needed to save the arithmetic term  $S - 1$ .

On the other hand, converting  $TD_2(r_2)$  yields the following set of rules:

$$\begin{aligned}
s_4 &: fresh\_pred\_1(Fuedelta) :- fuelcost(Fuedelta, \_, \_). \\
s_5 &: fresh\_pred\_2(Fuedelta, S, T) :- step(S), fuel(T, Fuelpre, S - 1), \\
&\quad Fuelpre \geq Fuedelta, \\
&\quad fresh\_pred\_1(Fuedelta). \\
s_6 &: precondition_s_d(T, L1, L2, S) :- at(T, L1, S - 1), \\
&\quad fuelcost(Fuedelta, L1, L2), \\
&\quad fresh\_pred\_2(Fuedelta, S, T).
\end{aligned}$$

In this case, the rule  $s_4$  is added to guarantee the safety of  $s_5$ . In particular, here the variable  $Fuedelta$  has to be saved, and in the body of  $s_4$  is sufficient to add  $fuelcost(Fuedelta, \_, \_)$ .

### 8.4.1 Estimating the Cost of Grounding a Rule

Estimating the cost of grounding a rule can help to get an intuition of its intrinsic complexity before actually grounding it. In our case, it will be adopted to determine if decomposing a rule could be advantageous.

We present an estimation that relies on statistics over predicates involved in the body, such as their extensions size and their argument selectivities. Before defining the formula described next, we considered several variants, and experimented with different formulae. Some techniques for parallel grounding are studied in [110]; in particular, body literals are distributed to different threads and it is defined a formula employed to decide how to split them. The formula is based on an estimation of the size of the joins of the body literals; similarly, it is introduced a formula to estimate the size of the joins of the body literals in [91], the one adopted for the *Combined* criterion (cf. Chapter 7). In our context, the two estimations resulted to be non suitable, as rather than the size of the join, we need to estimate the cost of instantiating a rule. Therefore, starting from these formulae, we defined a formula better fitting our purposes.

Let  $a = p(t_1, \dots, t_n)$  be a classical atom. As already stated, we denote with  $var(a)$  the set of variables occurring in  $a$ , while  $T(a)$  represents the number of different tuples for  $a$  in the ground extension of  $p$ . Moreover, for each variable  $X \in var(a)$ , we consider as the selectivity of  $X$  in  $a$ , denoted as  $V(X, a)$ , the number of distinct values in the projection of  $X$  over the ground extension of the predicate  $p$ .

Given two classical atoms  $b$  and  $c$ , the cost of evaluating the join between them is estimated as:

$$e_{b \bowtie c} = \frac{T(c)}{\prod_{X \in idx(var(b) \cap var(c))} V(X, c)} \cdot \prod_{X \in var(b) \cap var(c)} \frac{V(X, b)}{\max\{V(X, b), V(X, c)\}}$$

where  $idx(var(b) \cap var(c))$  is the set of the indexing arguments of  $c$ .

Intuitively, the cost of a join is influenced by the actual number of tuples that have to be taken into account when performing the join operation. Since grounders typically employ indexing techniques to retrieve matching instances, the number of tuples for  $c$  effectively considered when performing the join are just the ones matching with the partial substitution of the indexing variables that are in common with  $b$ . In addition, as expectable, this implies that the join  $b \bowtie c$  might not have the same cost of the  $c \bowtie b$ . As a consequence, the cost of grounding a rule tends to be highly influenced from the order of literals in  $B(r)$ .

Let  $r$  be a rule: by repeatedly applying the formula to atoms in the body of  $r$  we obtain an estimate of the cost of grounding  $r$ . Essentially, starting from the first two atoms in the body of  $r$ , and computing their join cost, we iterate the procedure estimating the cost of the join between the relation obtained so far (i.e. the one containing the contribution of the previous considered atoms), and the following atom in the body. The last step eventually gives the final cost of the join.

More in detail, given a rule  $r$ , let  $\langle a_1, \dots, a_m \rangle$  be the ordered list of atoms appearing in  $B(r)$ , for  $m > 1$ . Initially, the cost of grounding  $r$ , denoted by  $e_r$ , is set to  $T(a_1)$ , then the following formula is iteratively applied up to the last atom in the body in order to obtain the total estimated cost for  $r$ . Let us suppose

that we estimated the cost of joining the atoms  $\langle a_1, \dots, a_j \rangle$  for  $j \in \{1, \dots, m\}$ , and consequently we want to estimate the cost of joining the next atom  $a_{j+1}$ ; if we denote by  $A_j$  the relation obtained by joining all  $j$  atoms in  $\langle a_1, \dots, a_j \rangle$ , then:

$$e_{A_j \bowtie a_{j+1}} = \frac{T(a_{j+1})}{\prod_{X \in \text{idx}(\text{var}(A_j) \cap \text{var}(a_{j+1}))} V(X, a_{j+1})} \cdot \prod_{X \in (\text{var}(A_j) \cap \text{var}(a_{j+1}))} \frac{V(X, A_j)}{\text{dom}(X)} \quad (8.1)$$

where  $\text{dom}(X)$  is the maximum selectivity of  $X$  computed among the atoms in  $B(r)$  containing  $X$  as variable, and  $\text{idx}(\text{var}(A_j) \cap \text{var}(a_{j+1}))$  is the set of the indexing arguments of  $a_{j+1}$ . We note that, at each step, once the atom  $a_{j+1}$  has been considered,  $V(X, A_{j+1})$ , representing the selectivity of  $X$  in the virtual relation obtained at step  $j+1$ , has to be estimated in order to be used at next steps: if  $X \in \text{var}(A_j)$ , then  $V(X, A_{j+1}) = V(X, A_j) \cdot (V(X, a_{j+1})/\text{dom}(X))$ , otherwise  $V(X, A_{j+1}) = V(X, a_{j+1})$ . Therefore, the cost of grounding  $r$ , denoted  $e_r$ , corresponds to  $e_{\langle a_1, \dots, a_{m-1} \rangle \bowtie a_m}$ .

### 8.4.2 Estimating the Cost of Grounding a Decomposition

Let  $r$  be a rule and  $RD = \{r_1, \dots, r_n\}$  be a decomposition for  $r$ . In order to estimate the cost of grounding  $RD$ , one must estimate the cost of all rules in  $RD$ . For each  $r_i$  the estimation is performed by means of the formula discussed above. Nevertheless, it is worth noting that each  $r_i$  contains, in general, both predicates originally appearing in  $r$ , denoted as *known predicates*, and *fresh predicates*, that are generated during the decomposition. As for known predicates, thanks to the instantiation order of rules (cf. Section 5.2.2), actual data needed for computing the formula directly come from the instantiation of the previous rules. As for the fresh predicates, since they have been ‘‘locally’’ introduced and do not appear in any of the rules originally in the input program, such data must be estimated.

For such an estimation, the dependencies among the rules in  $RD$  are analysed, and an ordering that guarantees a correct instantiation is determined. This ordering can be determined by means of the same concepts of Section 5.2.1, thus by analysing the dependencies among involved predicates. Intuitively, rules depending only on known predicates can be grounded firstly, while rules depending on fresh predicates can be grounded only once the rules that define them have been instantiated.

Assuming that for the set  $RD$  a correct instantiation order is represented by  $\langle r_1, \dots, r_n \rangle$ , for each  $r'$  in this ordered list, if  $H(r') = p'(t_1, \dots, t_k)$  for  $k \geq 1$  and  $p'$  is a fresh predicate, we estimate the size of the ground extension of  $p'$ , denoted  $IE_{p_k}$ , as the size of the join of the classical atoms in  $B(r')$  by means of the Formula 7.1 described in Section 7.1. In particular, starting from the first two atoms in the body of  $r'$ , and computing their join size, we repeatedly apply the formula to estimate the size of the join between the already considered atoms and the next atom in the body. The last step eventually gives the final size of the join and hence,  $IE_{p_k}$ . In addition, the selectivity of each argument is estimated by assuming a uniform distribution as  $\sqrt[k]{IE_{p_k}}$ .



Therefore, the procedure `PREPROCESS` consists in a preprocessing of the rules in  $RD$  according to a valid grounding order  $\langle r_1, \dots, r_n \rangle$  by means of the aforementioned formula in order to obtain the extensions size and the argument selectivities also for involved fresh predicates. Next, a further preprocessing step consists in reordering rule bodies as they will actually be evaluated, i.e. according to the body ordering strategy employed. Consequently, the `ESTIMATE-EDecomposition` function can effectively estimate the costs of  $RD$ , denoted  $e_{RD}$ , by means of the `ESTIMATE` function.

**Example 8.4.3.** Let us consider again the rule of Example 8.4.2. These slight differences in the two decompositions  $RD_1(r_2)$  and  $RD_2(r_2)$  might have different influences on the ASP computation. Basically, depending on the instance at end, with our approach we estimate the costs of the three possible alternatives: (i) leave the rule as it is, (ii) choose  $RD_1(r_2)$  or (iii)  $RD_2(r_2)$ .

Let us assume that the ground extensions of involved predicates are the following:

$$\begin{aligned} &step(1..5). \\ &fuelcost(1..5, 1..5, 1..5). \\ &fuel(1..5, 1..5, 1..5). \end{aligned}$$

the cost of grounding  $r_2$  is estimated according to Formula (8.1); without reporting all intermediate calculations,  $e_{r_2}$  amounts to 390,625. In order to compute  $e_{RD_1(r_2)}$  we first need to determine a correct evaluation order of the rules in  $RD_1(r_2)$ ; the only valid one is  $\langle s_1, s_2, s_3 \rangle$ . Intuitively,  $s_1$  has only known predicates in its body, thus can be evaluated first; the body of  $s_2$  contains, besides to known predicates,  $fresh\_pred\_1$ , whose estimates will be available just after the evaluation of  $s_1$ ; eventually,  $s_3$  depends on  $fresh\_pred\_2$ , whose estimates will be available right after the evaluation of  $s_2$ . Once the estimates for the fresh predicates  $fresh\_pred\_1$  and  $fresh\_pred\_2$  are obtained by means of the Formula 7.1, they are used for computing  $e_{s_1}$ ,  $e_{s_2}$  and  $e_{s_3}$  with Formula (8.1), and thus  $e_{RD_1(r_2)} = e_{s_1} + e_{s_2} + e_{s_3}$ . Again, without reporting all intermediate calculations,  $e_{RD_1(r_2)}$  amounts to 122,945. Concerning  $RD_2(r_2)$ , the only valid ordering is  $\langle s_4, s_5, s_6 \rangle$  and analogously as done for  $e_{RD_1(r_2)}$ ,  $e_{RD_2(r_2)}$  is computed as 53,075. In this case, it is easy to see that the chosen decomposition is  $RD_2(r_2)$ . Interestingly, with a different input instance, things might change. For instance, if the set of input facts for  $fuelcost$  is changed to  $fuelcost(1..20, 1..20, 1..5)$ , the decomposition  $RD_1(r_2)$  is chosen. Eventually, according to the fixed threshold the decomposition may preferred over the original rule or not.

## Chapter 9

# Additional Techniques for Fine-Tuning the Grounding Process

The optimizations presented so far have a broad impact on whole the grounding process; hereafter, we illustrate further optimizations acting in particular scenarios, which can be handled in a more specific and efficient mode. In particular, this chapter presents a series of optimizations acting to different extent on the instantiation process. They have diverse aims such as decreasing the number of matches performed by the rule instantiation function, recovering as fast as possible inconsistencies in the input program, or syntactically rewriting the input program with the twofold intent of easing the instantiation and improving performance.

This chapter is organized as follows. Section 9.1 presents a technique that anticipates the evaluation of built-in atoms as soon as possible. Section 9.2 describes two techniques oriented to variables not involved in join operations. Section 9.3.1 provides a description of a technique designed to leverage differences in data distributions. In Section 9.4 we illustrate a technique intended to faster determine whether an input program is incoherent. Eventually, Section 9.5 discusses some manipulations of ASP-Core-2 syntactic features.

### 9.1 Pushing Down Selections

The first optimization presented is inspired by the classic relational algebra operation of pushing selections down the execution tree. Similarly to the above technique the aim is to reduce the search space and prevent failings when performing matches, but conversely it is directly embedded in the rule instantiation function.

Let  $r$  be a (non-ground) rule undergoing to the `INSTANTIATERULE` function. Suppose that the function is retrieving ground instances for a classical atom  $l \in B^+(r)$ , possibly by means of an index. Recall that let  $BoundVar(l)$  is the set of variables occurring in  $l$  and in the literals preceding  $l$  within  $B(r)$ . For each built-in atom  $l'$  in  $B(r)$  if  $var(l') \subseteq BoundVar(l)$ , then the selection

operation represented by  $l'$  is directly evaluated and instances not matching with it are skipped.

**Example 9.1.1.** Let us consider, for instance, the following rule  $r_1$ :

$$p(X, Y, V, S) :- t(X, Y, Z), q(Z, V, S), V < S.$$

It is easy to see that, instead of first joining  $t(X, Y, Z)$  with  $q(Z, V, S)$  and then selecting what complies with the comparison  $V < S$ , it is more convenient, in general, to first select, in the extension of  $q(Z, V, S)$ , the instances complying with the comparison, and then to join them with the ones of  $t(X, Y, Z)$ . Notably, this can be obtained by applying a rewriting process before instantiating  $r_1$ , that produces an auxiliary rule intended to filter instances in the extension of  $q/3$  satisfying the selection operation and thus properly rewrite  $r_1$ :

$$\begin{aligned} p_-(Z, V, S) &:- q(Z, V, S), V < S. \\ p(X, Y, V, S) &:- t(X, Y, Z), p_-(Z, V, S). \end{aligned}$$

However, this technique avoids the possible overhead deriving from the introduction of a new rule, and without actually performing the rewriting, simulates it during the instantiation of  $r_1$ : while retrieving the instances of  $q$ , only those complying with the comparison are actually taken into account.

## 9.2 Managing Isolated Variables

Let us consider a non-ground rule  $r$  containing a body atom  $p(X_1, \dots, X_n)$ , with a variable  $X_i$  that does not appear anywhere else in  $r$ . Essentially,  $X_i$  is not involved in any join operation, and we say that  $X_i$  is *isolated*. While instantiating  $r$ , substitutions for  $X_i$  will not affect any failed match for the other body literals, nor the instances obtained for the head literals. However, in order to not lose solutions, in our context where we have to compute all “relevant” solutions, it is needed to distinguish whether  $p/n$  is a solved predicate or not.

The techniques illustrated herein avoid to perform useless matches on these variables in both cases. In particular, in Section 9.2.1 we present a technique studied for isolated variables occurring within solved predicates; while in Section 9.2.2 we propose a more general mechanism to filter out also relevant isolated variables based on a rewriting technique, safely applicable to all predicates, with no distinction between solved and unsolved.

### 9.2.1 Filtering

Coming back to the situation reported above, in case  $p/n$  is a solved predicate,  $X_i$  is also not relevant, and when looking for next matches for  $p/n$ , all instances that differ just because of the substitution of  $X_i$  can be safely ignored. Indeed, according to the backjumping algorithm relevant instances of  $r$  are obtainable by considering substitutions for relevant variables, therefore by ignoring substitutions varying just because of  $X_i$  we are sure to not miss solutions.

To this end, the retrieval of matching instances for  $p/n$  can be empowered with a filter mechanism that suggests only the “relevant” instances. On the other hand, if  $p/n$  is unsolved, instead, all its instances must be considered in

order to preserve semantics of the produced ground program; thus, the use of the described filter mechanism must be prevented.

**Example 9.2.1.** Let us consider the rule  $r_1$  of the examples 7.3.3 and 7.3.4 on which we have focused in the context of the  $Combined_B^+$  criterion:

$$a(X, Z) :- b(X, Z), c(X, Y).$$

where,  $Y$  is an isolated and not relevant variable. By means of this filtering technique even if the rule body is ordered as  $c(X, Y), b(X, Z)$  we iterate through all the instances of  $b(X, Z)$  not for each instance of  $c(X, Y)$ , but we can just consider instances featuring distinct values for  $X$ , ignoring the values for  $Y$ .

Essentially, this filtering mechanism maintains the emphasis on relevant variables working in synergy with the backjumping machinery.

### 9.2.2 Rewriting

Just like the process of pushing down selections, it is possible to deal with isolated variables by means of a rewriting process. However, while in that situation it is preferable to simulate virtually an operation that can be performed as well by means of a rewriting, in this case an approach based on rewriting might be preferable, since it allows to avoid the distinction between solved and not solved predicates.

Let us consider again a non-ground rule  $r$  containing a body atom  $p(X_1, \dots, X_n)$ , with an isolated variable  $X_i$ . We can safely eliminate  $X_i$  by projecting all variables  $X_k, k \neq i$  of  $p$  to an auxiliary predicate  $p'$ . That is, a new (non-ground) rule is added:

$$p'(X_1, \dots, X_{i-1}, X_{i+1}, \dots, X_n) :- p(X_1, \dots, X_n).$$

and  $p(X_1, \dots, X_n)$  is substituted by  $p'(X_1, \dots, X_{i-1}, X_{i+1}, \dots, X_n)$  in the body of  $r$ . By doing so, the generation of ground instances of  $r$  which differ only on the binding of  $X_i$  is avoided. Notably, this rewriting preserves the original semantics of  $r$  if either  $p/n$  is solved or not.

**Example 9.2.2.** As an example let us consider the program,  $P_1$ :

$$\begin{aligned} a(X, Y) &:- b(X), c(Y, Z), \text{not } d(Y). \\ b(1) &| b(2) | b(3). \\ c(1, 1) &| c(1, 2) | c(1, 3). \end{aligned}$$

The variable  $Z$  is isolated because it appears only in  $c(Y, Z)$ ; moreover the predicates  $b/1$  and  $c/2$  are unsolved. By applying to the only non-ground rule in  $P_1$  the above mentioned rewriting we obtain:

$$\begin{aligned} a(X, Y) &:- b(X), c\_ (Y), \text{not } d(Y). \\ c\_ (Y) &:- c(Y, Z). \end{aligned}$$

where the fresh predicate  $c\_$  has been introduced.

Now, let focus on the impact on the instantiation. By grounding  $P_1$  in this original form we obtain:

$$\begin{aligned} a(1, 1) &: -b(1), c(1, 1), \text{not } d(1). \\ a(1, 1) &: -b(1), c(1, 2), \text{not } d(1). \\ a(1, 1) &: -b(1), c(1, 3), \text{not } d(1). \\ a(2, 1) &: -b(2), c(1, 1), \text{not } d(1). \\ a(2, 1) &: -b(2), c(1, 2), \text{not } d(1). \\ a(2, 1) &: -b(2), c(1, 3), \text{not } d(1). \\ a(3, 1) &: -b(3), c(1, 1), \text{not } d(1). \\ a(3, 1) &: -b(3), c(1, 2), \text{not } d(1). \\ a(3, 1) &: -b(3), c(1, 3), \text{not } d(1). \end{aligned}$$

Thus, the atoms  $a(1, 1)$ ,  $a(2, 1)$  and  $a(3, 1)$  are derived three times each, because of the three instances of the predicate  $c$  matching the pattern  $c(1, x)$ , where  $x$  is an arbitrary constant term. On the other hand, by grounding  $P_1$  after that the rewriting has been applied, we have:

$$\begin{aligned} a(1, 1) &: -b(1), c\_ (1), \text{not } d(1). \\ a(2, 1) &: -b(2), c\_ (1), \text{not } d(1). \\ a(3, 1) &: -b(2), c\_ (1), \text{not } d(1). \\ c\_ (1) &: -c(1, 1). \\ c\_ (1) &: -c(1, 2). \\ c\_ (1) &: -c(1, 3). \end{aligned}$$

Supposing that we execute the two versions of  $P_1$  by increasingly adding more facts having 1 as first argument and more facts the predicate for  $b/1$ , while the rest remains the same. Let  $m$  be the number of facts for  $b/1$  and  $n$  be the number of facts of form  $c(1, x)$ , where  $x$  is an arbitrary constant term. Then, with the original version of  $P_1$  we obtain each time  $m \cdot n$  rules, whereas with the rewritten version we generate just  $m + n$  rules.

Indeed, in many cases this rewriting allows to reduce the size of the ground program, and consequently the instantiation time; nevertheless, an overhead is paid, due to the need for copying the projected instances of  $p$  in the extension of  $p'$ ; such overhead, even if negligible in general, might become significant when the benefits of the projection are limited, as will be evidenced in Section 11.5.

### 9.3 Determining the Admissibility of Substitutions

During the rule instantiation function, we intrinsically look for an “agreement” between body literals on variable substitutions, since literals should agree on mappings for each common variable. To further ease this search we developed two different techniques: an ad-hoc mechanism that is geared towards the “alignment” of variable substitutions, presented in Section 9.3.1, and a more flexible look-ahead technique that aims at determining as soon as possible whether during the rule instantiation a partial substitution is not admissible, illustrated in Section 9.3.2.

### 9.3.1 Aligning Substitutions

Given a predicate  $p$  of arity  $n$ , for each argument  $i$  with  $i \in \{0, \dots, n\}$ ,  $set(p_n, i)$  represents the set of possible (distinct) values for the argument  $i$  of  $p$ . Given a rule  $r$ , a variable  $X \in var(B(r))$  is said to be a *join variable* if it occurs in at least two *different* classical atoms in  $B^+(r)$ . In this context, two classical atoms  $l_1$  and  $l_2$  are considered different if:

- $l_1$  and  $l_2$  contains the predicate  $p/n$ ,  $n > 0$ , and a term  $t$  is the  $i$ -th argument of  $l_1$  and the  $j$ -th argument for  $l_2$  with  $i \leq n$ ,  $j \leq n$  and  $i \neq j$ ;
- $l_1$  and  $l_2$  are defined over two different predicates  $p/n$  and  $p/m$ , with  $n \geq 0$ ,  $m \geq 0$  and  $n \neq m$ ;
- $l_1$  and  $l_2$  are defined over two different predicates  $p/n$  and  $q/m$ , with  $n \geq 0$  and  $m \geq 0$ .

Subsequently, before the grounding of  $r$  actually starts, for each join variable  $X$  in  $B^+(r)$  it is computed the intersection, denoted  $I(X)$ , of the sets  $set(q_m, j)$ , where  $X$  is featured by the  $j$ -th argument of the predicate  $q/m$  with  $j \leq m$  in  $B^+(r)$ . At this point,  $r$  can undergo to the instantiation procedure. Let  $\theta$  be a partial variable substitution. Each time is performed a match on a literal  $l \in B^+(r)$  such that  $X \in var(l)$  is a join variable:

- if  $l$  is a classical atom over a predicate  $q/m$  and  $X$  corresponds to its  $k$ -th argument, then it is possible to skip those instances in  $I_{q_m}$  such that as their  $k$ -th argument feature a ground term  $t$  and  $t \notin I(X)$ ;
- if  $l$  is a built-in atom of form  $X = u$ , let  $u'$  be the term obtained after the arithmetical evaluation of  $u\theta$ , then the match fails if  $u' \notin I(X)$ ;
- if  $l$  is an aggregate atom of form  $af\{e_1, \dots, e_n\} = X$  representing an assignment, thus generating suitable mappings for  $X$ , it is possible to skip the matches in which to  $X$  is mapped a ground term  $w$  and  $w \notin I(X)$ .

Essentially, the intent is to faster determine inconsistencies and reduce, in general, the number of possible values for join variables, by skipping those that would not match among distinct variable occurrences. Intuitively, such technique fits best when the involved sets of possible values differ significantly. In particular, in situations in which among the computed intersections, at least one of them is empty, the instantiation of the rule at hand can be safely aborted, as no successful match can be performed.

**Example 9.3.1.** As a first example, let us consider the rule  $r_1$ :

$$a(X, Y, Z) :- b(X, Y), c(Y, X, Z), d(Y, Z), e(X).$$

The variable  $X$  occurring three times in the body of  $r_1$  within literals over different predicates. Let us suppose that  $|set(b_2, 0)| = 300$ ,  $|set(c_3, 1)| = 500$  and  $|set(e_1, 0)| = 10$ . The computed intersection for  $X$  consists here in  $I(X) = set(b_2, 0) \cap set(c_3, 1) \cap set(e_1, 0)$ . Hence, the intersection will just contain the values on which predicates  $b/2$ ,  $c/3$  and  $e/1$  “agree”. Intuitively, due to the presence of  $e(X)$ , at most 10 values can be substituted to  $X$  in order to generate

a significant substitution (i.e. a substitution that allows to obtain a relevant ground instance of  $r_1$ ).

Consequently, when substituting  $X$  with a ground value during the instantiation of  $r$ , each value that is not present in the computed intersection can be safely skipped.

**Example 9.3.2.** Let us now consider the rule  $r_2$ :

$$a(X) :- b(Y), \#count\{Z : c(Z, Y)\} = X, d(J), W = J + 1, e(Y, W), f(W, Y).$$

There are two join variables:  $Y$  and  $W$ ; since they appear in two different classical atoms. Let us assume that  $c/2$  is an unsolved predicate, and that as input facts we have:

$$\begin{aligned} &b(1..500). \\ &c(1..500, 1..500). \\ &d(1..100). \\ &e(1..200, 1..5). \\ &f(1..300, 1..10). \end{aligned}$$

Hence, the intersections are  $I(Y) = \{1, \dots, 10\}$  and  $I(W) = \{1, \dots, 5\}$ . Let us assume also that the body is not reordered and so  $r_2$  goes through the rule instantiation function as it is above. In case the aligning technique is not enabled, during the instantiation of  $r_2$  the number of matches performed on  $\#count\{Z : c(Z, Y)\}$  is 500 because of the 500 instances of  $b/1$ , and for the built-in  $W = J + 1$ , the matches are 100 because of  $d/1$ . However, so far we are considering also not relevant substitutions, and it will evident only when the procedure jumps on the literals  $e(X, W)$  and  $f(W, X)$ , since they reduce the search space for  $X$  and  $W$ ; hence, at this point the procedure has to jump back to recover failing matches.

Conversely, via the technique information about the effective search space for join variables is pre-computed, thus the number of matches performed on the first two literals is reduced to 10, and to 5 for the subsequent two. It is worthwhile noting that a good body ordering might avoid the first scenario showed above, by first placing the literals  $e(X, W)$  and  $f(W, X)$ . However, the alignment technique allows to reduce the search space immediately as soon as a join variable is bounded, independently from the way in which the body is ordered.

### 9.3.2 Look-Ahead Technique

The above presented technique about aligning variable substitutions may cause some slowdown in performance arising from the computation of the involved domain intersections, that might lead also to a significant larger memory consumption in case of particular large extensions.

To overcome these limitations, we defined a more flexible and light-weight technique having a similar purpose. Recall that for a literal  $l$ , we denote as  $BindVar(l)$  the set of variables in  $var(l)$  for which  $l$  is a binder and that for a predicate  $p$  of arity  $n$ , for each argument  $i$  with  $i \in \{0, \dots, n\}$ ,  $set(p_n, i)$  we denote the set of possible (distinct) values for the argument  $i$  of  $p$ .

Let  $r$  be a rule which is undergoing to the rule instantiation function. Let  $l_1, \dots, l_m$  be the classical atoms appearing in its body. Suppose that the body

ordering strategy order them as  $l_1, \dots, l_m$ . During the instantiation, when a match on a literal  $l_i \in \{l_1, \dots, l_m\}$  is performed, a look-ahead step is achieved to verify whether for the other literals featuring some variables in  $BindVar(l_i)$  the mappings assigned to such variables are admissible. More formally, let  $\theta$  be the current partial substitution. For each literal  $l_j$  over a predicate  $p/n$  with  $j \in \{i+1, \dots, m\}$  and for each variable  $X \in BindVar(l_i)$  such that  $X$  appears as  $k$ -th argument in the literal  $l_j$  with  $k \leq n$ , if  $X\theta \notin set(p_n, k)$  then the match on  $l_i$  fails, and another match is performed with a fresh instance for  $l_i$ .

**Example 9.3.3.** Let us consider again the rule  $r_1$  of Example 9.3.1:  $a(X, Y, Z) :- b(X, Y), c(Y, X, Z), d(Y, Z), e(X)$ . Assume to have the following input facts:

$$b(1, 2). b(1, 3). b(1, 4). c(4, 1, 3). \\ c(2, 2, 1). d(2, 1). d(4, 1). e(2). e(3).$$

and that the original body order is maintained. Let  $\theta$  be the current partial substitution. Initially, a match is performed on  $b(X, Y)$ . Suppose that the instance  $b(1, 2)$  is selected, thus  $\theta = \{X = 1, Y = 2\}$ , a look-ahead step is performed by checking whether:

- in  $set(c_3, 1)$  there is the ground term 2, and in  $set(c_3, 2)$  the ground term 1;
- in  $set(d_2, 1)$  there is the ground term 2;
- in  $set(e_1, 1)$  there is the ground term 1.

It is easy to check that the first two conditions holds, while the third one does not. Therefore, another instance is selected, say  $b(1, 3)$ ;  $\theta$  is updated as  $\{X = 1, Y = 3\}$  and since  $3 \notin set(c_3, 1)$  the match fails again. Finally, the last instance is  $b(1, 4)$ , this time the mapping  $\{X = 1, Y = 4\}$  is admissible for all the other literals, as they “agree” on the assigned ground terms to common variables. In absence of such a technique all the three partial substitutions are propagated and the fails are recovered only when the function jumps to a non-matching literal.

Intuitively, we are achieving a similar effect as with the technique illustrated in Section 9.3.1: partial substitutions for which literals do not agree are not propagated, and thus these failing matches are immediately recovered. Furthermore, in general, this technique has a more negligible impact on the performance, mitigating the overhead even in situations in which extensions do not differ significantly, as we will show in Section 11.6. On the other hand, in case involved extensions differ at the point that no instance can be derived, the alignment variable substitutions technique is preferable, since it allow to completely prevent the instantiation.

## 9.4 Anticipating Strong Constraints Evaluation

Due to the missing heads, the instantiation of strong constraints does not affect the creation of predicate extensions: thus, they could be safely processed at the very end, once the instantiation of all program modules is completed, as discussed in Section 5.2.1.



However, due to the simplification mechanism, literals appearing in the body of a ground constraint can be removed, possibly leading to an empty-body constraint. By definition, such constraints are always violated; thus, the input program is incoherent, and the grounding process can be safely aborted. For this reason, their grounding can be anticipated. In particular, a strong constraint can undergoes the rule instantiation function as soon as the extensions of its body predicates are available.

Let  $P$  be an ASP program, and  $C_1, \dots, C_n$  be an *admissible component sequence* for the strongly connected components of its dependency graph  $G_P$ , and let  $S$  be the set of strong constraints in  $P$ . Given a rule  $r$  we denote as  $BP_r$  and  $HP_r$  the set of body and head predicates in  $r$ , respectively. For a component  $C \in \{C_1, \dots, C_n\}$  we denote as  $P_C$  the set of predicates defined by  $C$ , i.e. the union of the set  $HP_r$  for each rule  $r' \in C$ . After the instantiation of a component  $C_i$  for  $i \in \{1, \dots, n\}$ , each strong constraint  $s \in S$  if  $BP_s \subseteq \bigcup_{j=1}^i P_{C_j}$  can be grounded since the extensions of its body predicates have been definitely computed.

We remark that a similar mechanism may be applied to weak constraints, however, even if after the simplification step it is obtained an empty body weak constraint, the instantiation cannot be aborted. Indeed, differently from strong constraints representing integrity conditions, as suggested by their name, weak constraints are “weaker” conditions that should be preferably satisfied (cf. Section 2.2).

This optimization is particularly useful in all that situations in which ASP is used for checking whether some strict conditions hold, allowing to determine as fast as possible the presence of incoherence.

**Example 9.4.1.** As an example, let us consider the following program  $P_1$ , a variant of the program  $P_{3col}$  of the example 3.1:

$$\begin{aligned} r_1 &: color(X, red) \mid color(X, green) \mid color(X, blue) :- vertex(X). \\ r_2 &: :- edge(X, Y), color(X, C), color(Y, C). \\ r_3 &: used\_color(C) :- color(X, C). \end{aligned}$$

Essentially, as in the *Three-colorability* problem, the aim is to assign to each vertex one of three colors such that adjacent vertices always have distinct colors. Moreover, here we want to determine which colors have been effectively employed. To this end, in addition to  $P_{3col}$ ,  $P_1$  features the additional rule  $used\_color(C) :- color(X, C)$ .

Computing the dependency and component graphs of  $P_1$  we obtain that there are two components: a first component formed by the predicate  $color/2$  and a further one composed by  $used\_color/1$ , and there are no negative dependencies. Hence, we can easily conclude that an admissible component sequence is:  $color/2, used\_color/1$ . Let us assume that  $P_1$  is coupled with the following input facts,  $I$ :

$$vertex(1). vertex(2). edge(1, 2). color(1, red). color(2, red).$$

The instantiation of  $P_1 \cup I$  comprises the constraint:

$$:- edge(1, 2), color(1, red), color(2, red).$$

which is always violated because the input facts  $color(1, red)$  and  $color(2, red)$  constrain the assigned color to both vertices 1 and 2 to red. Consequently,

$P_1 \cup I$  has no answer set. Employing a component-wise instantiation of strong constraints,  $r_2$  can be grounded right after the component formed by  $color/2$  has been processed, and the incoherence of  $P_1 \cup I$  becomes immediately evident, hence there is no need to proceed in the instantiation of the next component.

Remarkably, in this specific case the incoherence can be detected without actually grounding the component relative to  $color/2$ , because the ground constraint showed above is obtained directly from input facts. However, in general, grounding a constraint before that the rules defining its body predicates have been processed do not ensure that relevant ground instances for it are missed. For instance, in our example, we would have lost the constraints:

$$\begin{aligned} & :- \text{edge}(1, 2), \text{color}(1, \text{blue}), \text{color}(2, \text{blue}). \\ & :- \text{edge}(1, 2), \text{color}(1, \text{green}), \text{color}(2, \text{green}). \end{aligned}$$

## 9.5 Rewriting Techniques for Handling Different Language Features

In this section we describe some semantically equivalent alternatives to write rules containing particular constructs of ASP-Core-2 syntax such as arithmetic terms (Section 9.5.1), functional terms (Section 9.5.2), aggregate literals (Section 9.5.3) and choice atoms (Section 9.5.4). Each rewriting technique guarantees that the semantics of the original rules is preserved, and has been designed to bring benefits from both the *software engineering* and the performance sides.

### 9.5.1 Arithmetic Terms

Given a literal  $l$  containing an arithmetic term  $u$ , we denote as  $\bar{l}$  the literal obtained by replacing in  $l$  the term  $u$  with a fresh variable  $X$ . Let  $r$  be a (non-ground) rule containing in its body the literals  $l_1, \dots, l_n$ . Let  $l_i$  for  $i \leq n$  be a literal in which it occurs an arithmetic term  $t$ . The body of  $r$  can be rewritten as  $l_1, \dots, l_{i-1}, \bar{l}_i, l_{i+1}, \dots, l_n, X = t$ , where  $X$  is a fresh variable, i.e. a variable that does not appear in  $var(r)$  before that the rewriting is applied, and  $X = t$  is an additional built-in atom.

**Example 9.5.1.** Let us consider the rule  $r_1$ :

$$a(X, Y) :- c(X, Y, Z), c(X + Y * Z).$$

Then,  $r_1$  is rewritten as:

$$a(X, Y) :- c(X, Y, Z), c(FV), FV = X + Y * Z.$$

where  $FV$  is a fresh variable.

The same rewriting can be applied also in case an arithmetic term appears in a classical atom in the head or in a choice atom. Let  $r$  be a (non-ground) rule containing in its head the literals  $l_1 \mid \dots \mid l_n$ . Let  $l_i$  for  $i \leq n$  be a literal in which it occurs an arithmetic term  $t$ . The head of  $r$  can be rewritten as  $l_1 \mid \dots \mid l_{i-1} \mid \bar{l}_i \mid l_{i+1} \mid \dots \mid l_n$ , where  $X$  is a fresh variable and in the body of  $r$  is added the built-in atom  $X = t$ .

Eventually, let  $r$  be a choice rule containing in its head a choice element  $a : l_1, \dots, l_n$ . In case the atom  $a$  contains an arithmetic term  $t$ , the choice element can be rewritten as:  $\bar{a} : l_1, \dots, l_n, X = t$ . On the other hand, supposing that instead an arithmetic term  $t$  appears in a literal  $l_i$  among the literals  $l_1, \dots, l_n$ , the choice element can be rewritten as:  $a : l_1, \dots, l_{i-1}, \bar{l}_i, l_{i+1}, \dots, l_n, X = t$ .

**Example 9.5.2.** Let us consider the rules:

$$\begin{aligned} r_1 &: a(X, Y) \mid b(X + Y) :- c(X, Y). \\ r_2 &: \{a(X + Y) : b(X), b(Z)\} :- c(Y, Z). \\ r_3 &: \{a(X, Y) : b(X), b(X + Y * Z)\} :- c(Y, Z). \end{aligned}$$

Then, according to the rewriting described herein we obtain:

$$\begin{aligned} r_1 &: a(X, Y) \mid b(FV) :- c(X, Y), FV = X + Y. \\ r_2 &: \{a(FV) : b(X), b(Z), FV = X + Y\} :- c(Y, Z). \\ r_3 &: \{a(X, Y) : b(X), b(FV), FV = X * Z\} :- c(Y, Z). \end{aligned}$$

Intuitively, in every situation we are introducing a new built-in atom representing an assignment, since it is the binder of the newly introduced fresh variable. This simple rewriting may bring benefits or not; mostly, its effects depend on the way this rewriting interacts with the optimizations intervening in the instantiation process. In particular, the major influence is observable on the body ordering strategy. To grasp the intuition, let us consider the following example.

**Example 9.5.3.** Let us consider the program  $P_1$ :

$$\begin{aligned} &a(Y) :- b(Y), c(Y - 1). \\ &b(1..10,000,000). \\ &c(1..100). \end{aligned}$$

As already discussed in Chapter 7, the first requirement of body ordering strategies is to guarantee the correct instantiation. This implies that the only possible ordering for the body of the rule reported above is:  $b(Y), c(Y + 1)$ , because to evaluate  $c(Y + 1)$  it is needed that the literal  $b(Y)$  binding  $Y$  occurs before  $c(Y + 1)$ . The number of matches performed to ground  $r_1$  as it is can be roughly estimated as: 10,000,000: for each instance of  $b/1$  we look for a matching instance in the extension of  $c/1$ .

By enabling the aforementioned rewriting the rule is rewritten as:

$$a(X, Y) :- b(Y), c(Z), Z = Y + 1.$$

Consequently, now there are three possible orderings: *i*)  $b(Y), c(Z), Z = Y + 1$ , *ii*)  $c(Z), b(Y), Z = Y + 1$  or *iii*)  $c(Z), Z = Y + 1, b(Y)$ . Selecting the former two orderings we would perform around  $10,000,000 \cdot 100 = 1,000,000,000$  matches, 100 times more than the number of matches performed if the rewriting is not applied. With the latter ordering, the number of matches performed can be reduced to 100. Indeed, in order to ground the rule in this last case, firstly it is needed to iterate through the 100 instances of  $c/1$ , next the built-in  $Z = Y + 1$  can be seen as an assignment for the variable  $Y$  and thus can be evaluated as  $Y = Z - 1$ , so just one match is performed on  $b(Y)$  to check whether the

value assigned to  $Y$  is valid. Notably, the benefit deriving from this ordering arises from the capability of transforming the built-in as an assignment. All the strategies deriving from the *Combined*<sup>+</sup> tries to estimate how much the search space is reduced by built-in atoms, thus they would correctly select the third ordering.

Hence, depending on the body ordering strategy adopted the introduction of built-in atoms in the rule bodies may lead to new possible orderings bringing significant advantages or have a negative influence on performance.

### 9.5.2 Functional Terms

Let us consider a non-ground rule  $r$  containing in the body a literal  $l$  in which a functional term, say  $t$ , occurs. Let  $var(l) = \{X_1, \dots, X_n\}$ , we can introduce an auxiliary rule composed as follows:

$$p'(X_1, \dots, X_n) :- l.$$

where  $p'$  is a fresh predicate, and then replace  $l$  with  $p'(X_1, \dots, X_n)$  in  $r$ .

**Example 9.5.4.** Let us consider the rule  $r_1$ :

$$a(X, Y) :- b(f(X, Y), X, g(h(1, X)), 3), c(X, Y).$$

Then, according to the rewriting described herein we obtain:

$$\begin{aligned} b_-(X, Y) &:- b(f(X, Y), X, g(h(1, X)), 3). \\ a(X, Y) &:- b_-(X, Y), c(X, Y). \end{aligned}$$

Even if, similarly to the rewriting of isolated variables illustrated in Section 9.2.2 an overhead is paid, due to the need for copying a part of the instances in the extension of  $p'$ , in general, the benefit of the rewriting overcomes this drawback allowing an efficient retrieval of ground instances independently from the presence of functional terms.

**Example 9.5.5.** Let us consider the program  $P_1$ :

$$\begin{aligned} a(X, Y) &:- b(X, Y), c(f(X, Z)). \\ b(1..100, 100). \\ c(f(1..1000, 1..1000)). \end{aligned}$$

Supposing that the rewriting is not applied, the number of matches performed when grounding the only rule of  $P_1$ , if the body order reported above is not changed, is around  $10,000 \cdot 1,000,000 = 10,000,000,000$ . Indeed, even if when grounding  $c(f(X, Z))$  the variable  $X$  is bound, the first argument of  $c/1$  is not indexable, and thus we need to iterate through all the extension of  $c/1$ .

To overcome such situations, we may design indexing techniques specifically defined for functional terms, or rewrite the functional term and simply reuse the indexing strategies discussed in Chapter 6.

By means of the rewriting illustrated herein the rule is rewritten as:

$$\begin{aligned} a(X, Y) &:- b(X, Y), c_-(X, Z). \\ c_-(X, Z) &:- c(f(X, Z)) \end{aligned}$$

Therefore, by replacing  $c(f(X, Z))$  with  $c_-(X, Z)$ , the number of matches performed can be reduced to  $10,000 \cdot 1,000 = 10,000,000$ , because we can employ an index on the first argument of  $c_-(X, Z)$ .

### 9.5.3 Aggregate Literals

Let  $r$  be a (non-ground) rule containing in its body an aggregate literal composed by an aggregate element  $e$  of form  $t_1, \dots, t_n : l_1, \dots, l_m$  for  $n > 0$  and  $m > 1$ . We will refer to  $l_1, \dots, l_m$  as *aggregate body* of  $e$ . Let  $\{X_1, \dots, X_k\} = \text{var}(t_1, \dots, t_n) \cup \text{var}_g(l_1, \dots, l_m)$  for  $k > 0$  be the set of variables composed by the variables in the terms  $t_1, \dots, t_n$  and the global variables of  $l_1, \dots, l_m$ . Moreover, let  $s_1, \dots, s_j$  for  $j \geq 0$  be a set of literals in  $B(r)$  corresponding to saviours for the literals  $l_1, \dots, l_m$ , then a rule  $r'$  is obtained as:

$$p'(X_1, \dots, X_k) :- l_1, \dots, l_m, s_1, \dots, s_j.$$

where  $p'$  is a fresh predicate and, we rewrite  $e$  in  $r$  as  $t_1, \dots, t_n : p'(X_1, \dots, X_k)$ .

**Example 9.5.6.** Consider the following rule:

$$a(X) :- g(X, Z), \#max\{Y : b(X, Y + Z), c(Y), not f(Z); W : f(W)\} >= 3.$$

When applying the rewriting to its aggregate elements we obtain:

$$\begin{aligned} a(X) :- g(X, Z), \#max\{Y : p_-(Y, X, Z); W : f(W)\} >= 3. \\ p_-(Y, X, Z) :- b(X, Y + Z), c(Y), not f(Z), g(X, Z). \end{aligned}$$

The only aggregate element that has been rewritten is  $Y : b(X, Y + Z), c(Y), not f(Z)$  because it contains more than one literal. In the body of the auxiliary rule generated, besides its aggregate body, the atom  $g(X, Z)$  has been added because of the atom  $b(X, Y + Z)$ .

Notably, by means of this rewriting, aggregate elements are put in a “standard” form ensuring that each one contains at most a literal. In turn, this standardization eases the instantiation and from the software engineering side the possibility to treat the body of an aggregate element as the body of a rule permits to automatically apply to the instantiation of its aggregate body all the optimizations intervening in the instantiation of the body of a rule.

**Example 9.5.7.** Consider the following program  $P_1$ :

$$\begin{aligned} a(Z) :- f(Z), \#count\{Y, W, T : b(Y), c(W), d(T); U : e(U, Z)\} = Z. \\ b(1) \mid c(1) \mid d(1). \\ b(2) \mid c(2) \mid d(2). \\ f(1) \mid f(2) \mid f(3). \\ e(1, 1) \mid e(1, 2) \mid e(1, 3). \end{aligned}$$

The instantiation of  $P_1$  yields to the following ground program:

$$\begin{aligned} a(9) :- f(1), \#count\{1, 1, 1 : b(1), c(1), d(1); 1, 2, 1 : b(1), c(2), d(1); \\ 1, 1, 2 : b(1), c(1), d(2); 1, 2, 2 : b(1), c(2), d(2); \\ 2, 1, 1 : b(2), c(1), d(1); 2, 2, 1 : b(2), c(2), d(1); \\ 2, 1, 2 : b(2), c(1), d(2); 2, 2, 2 : b(2), c(2), d(2); \\ 1 : e(1, 1)\} = 1. \end{aligned}$$

$$a(9) : - f(2), \#count\{1, 1, 1 : b(1), c(1), d(1); 1, 2, 1 : b(1), c(2), d(1); \\ 1, 1, 2 : b(1), c(1), d(2); 1, 2, 2 : b(1), c(2), d(2); \\ 2, 1, 1 : b(2), c(1), d(1); 2, 2, 1 : b(2), c(2), d(1); \\ 2, 1, 2 : b(2), c(1), d(2); 2, 2, 2 : b(2), c(2), d(2); \\ 1 : e(1, 2)\} = 2.$$

$$a(9) : - f(3), \#count\{1, 1, 1 : b(1), c(1), d(1); 1, 2, 1 : b(1), c(2), d(1); \\ 1, 1, 2 : b(1), c(1), d(2); 1, 2, 2 : b(1), c(2), d(2); \\ 2, 1, 1 : b(2), c(1), d(1); 2, 2, 1 : b(2), c(2), d(1); \\ 2, 1, 2 : b(2), c(1), d(2); 2, 2, 2 : b(2), c(2), d(2); \\ 1 : e(1, 3)\} = 3.$$

As it might be observed, the Cartesian product defined by  $b(Y), c(W), d(T)$  is computed three times, each time a new value for  $Z$  is considered. In our example, for the sake of readability, we considered few input facts, and the result of this product is limited to 8 tuples. However, performance tends to get worse as long as a larger number of tuples is produced due to its repeated instantiation. By applying the rewriting to aggregate elements, we can avoid this drawback, obtaining:

$$a(X) : - \#count\{Y, W, T : p\_ (Y, W, T); U : e(U, Z)\} = X, f(Z). \\ p\_ (Y, W, T) : - b(Y), c(W), d(T).$$

that leads to the following instantiation:

$$p\_ (1, 1, 1) : - b(1), c(1), d(1). \\ p\_ (1, 2, 1) : - b(1), c(2), d(1). \\ p\_ (1, 1, 2) : - b(1), c(1), d(2). \\ p\_ (1, 2, 2) : - b(1), c(2), d(2). \\ p\_ (2, 1, 1) : - b(2), c(1), d(1). \\ p\_ (2, 2, 1) : - b(2), c(2), d(1). \\ p\_ (2, 1, 2) : - b(2), c(1), d(2). \\ p\_ (2, 2, 2) : - b(2), c(2), d(2).$$

$$a(9) : - f(1), \#count\{1, 1, 1 : p\_ (1, 1, 1); 1, 2, 1 : p\_ (1, 2, 1); \\ 1, 1, 2 : p\_ (1, 1, 2); 1, 2, 2 : p\_ (1, 2, 2); \\ 2, 1, 1 : p\_ (2, 1, 1); 2, 2, 1 : p\_ (2, 2, 1); \\ 2, 1, 2 : p\_ (2, 1, 2); 2, 2, 2 : p\_ (2, 2, 2); \\ 1 : e(1, 1)\} = 1.$$

$$a(9) : - f(2), \#count\{1, 1, 1 : p\_ (1, 1, 1); 1, 2, 1 : p\_ (1, 2, 1); \\ 1, 1, 2 : p\_ (1, 1, 2); 1, 2, 2 : p\_ (1, 2, 2); \\ 2, 1, 1 : p\_ (2, 1, 1); 2, 2, 1 : p\_ (2, 2, 1); \\ 2, 1, 2 : p\_ (2, 1, 2); 2, 2, 2 : p\_ (2, 2, 2); \\ 1 : e(1, 2)\} = 2.$$

$$\begin{aligned}
a(9) :- f(3), \#count\{ & 1, 1, 1 : p\_ (1, 1, 1); 1, 2, 1 : p\_ (1, 2, 1); \\
& 1, 1, 2 : p\_ (1, 1, 2); 1, 2, 2 : p\_ (1, 2, 2); \\
& 2, 1, 1 : p\_ (2, 1, 1); 2, 2, 1 : p\_ (2, 2, 1); \\
& 2, 1, 2 : p\_ (2, 1, 2); 2, 2, 2 : p\_ (2, 2, 2); \\
& 1 : e(1, 3)\} = 3.
\end{aligned}$$

Essentially, by means of the auxiliary rule we are able to compute the aforementioned product just once and store it. Thus, the instantiation of the corresponding aggregate element amounts at enumerating the ground extension of the fresh predicate  $p_/3$ . In addition, in this way the aggregate body is grounded as rule body and undergoes to every optimization provided for the rule instantiation process.

### 9.5.4 Choice Rules

As anticipated in Section 2.3 choice rules represent syntactic shortcuts. Intuitively, a choice rule means that, if the body of the rule is true, an arbitrary subset of its choice elements can be chosen as true in order to comply with the involved comparison relation. Herein we describe three alternative rewriting techniques.

Let  $r$  be a (non-ground) rule containing in its head a choice atom:

$$\begin{aligned}
\{a_1(X_{1_1}, \dots, X_{k_1}) : l_{1_1}, \dots, l_{m_1}; \dots; a_n(X_{1_n}, \dots, X_{k_n}) : l_{1_n}, \dots, l_{m_n}\} \triangleright u \\
:- b_1, \dots, b_p.
\end{aligned}$$

For each choice element  $a_i(X_{1_i}, \dots, X_{k_i}) : l_{1_i}, \dots, l_{m_i}$  for  $i \in \{1, \dots, n\}$ , we will refer to  $l_{1_i}, \dots, l_{m_i}$  as its *body*. In each rewriting, if  $p > 1$ , let  $var_g(B(r)) \cap var(H(r)) = \{Y_1, \dots, Y_t\}$ , then we generate the rule  $r'$ :

$$p'(Y_1, \dots, Y_t) :- b_1, \dots, b_p.$$

where  $p'$  is a fresh predicate. In case  $p = 1$ , i.e.  $|B(r)| = 1$ , let  $b_1$  be the only literal in  $B(r)$ . In the following, we generically denote as  $b$  either the atom  $p'(Y_1, \dots, Y_t)$  if  $p > 1$ , or  $b_1$  if  $p = 1$ .

#### First Rewriting

Firstly, we propose a rewriting consisting in the removal of the choice atom, and the introduction of disjunctive rules. In Section 2.3 we already showed that this kind of syntactic manipulation is possible. Next, we define a refined version of such rewriting leading to a more efficient instantiation.

For each choice element,  $a_i(X_{1_i}, \dots, X_{k_i}) : l_{1_i}, \dots, l_{m_i}$  for  $i \in \{1, \dots, n\}$ , let  $\{Z_{1_i}, \dots, Z_{q_i}\} = var_g(B(r)) \cap var(l_{1_i}, \dots, l_{m_i})$ . Then a first auxiliary rule is produced as follows:

$$a_i(X_{1_i}, \dots, X_{k_i}) \mid - a_i(X_{1_i}, \dots, X_{k_i}) :- l_{1_i}, \dots, l_{m_i}, b.$$

In case  $m_i > 1$ , a second auxiliary rule consists in:

$$a'_i(Z_{1_i}, \dots, Z_{q_i}) :- a_i(X_{1_i}, \dots, X_{k_i}), s_1, \dots, s_r.$$

where  $a'_i$  is a fresh predicate and  $\{s_1, \dots, s_r\} \subseteq \{l_{1_i}, \dots, l_{m_i}\}$  is a possible empty list of saviours for the variables  $Z_{1_i}, \dots, Z_{q_i}$ . Moreover, let  $ae$  be a set of aggregate elements obtained by adding:

- an aggregate element  $Z_{1_i}, \dots, Z_{q_i} : a'_i(Z_{1_i}, \dots, Z_{q_i})$  for each choice element  $a_i(X_{1_i}, \dots, X_{k_i}) : l_{1_i}, \dots, l_{m_i}$  with  $m_i > 1$ ;
- an aggregate element  $X_{1_i}, \dots, X_{k_i} : a_i(X_{1_i}, \dots, X_{k_i})$  for each choice element with  $m_i \leq 1$ .

Eventually, the following constraint is added:

$$:- b, \#count\{ae\} \triangleright u.$$

If  $|B(r)| = 0$  in every generated rule, if  $b$  is present it is omitted.

Essentially, by means of the auxiliary rule  $r'$  we are readopting the same trick of Section 9.5.3 of storing the result of a frequent join operation into the ground extension of a fresh predicate. Moreover, the disjunctive rules stemming from each choice elements, as well as the introduced constraint, permit to get rid of the choice atom and ensure that the original comparison relation is respected. Thus, we are able to completely remove choice rules from an input program, with the advantage that, if this rewriting is applied before the instantiation process starts, it has not to be aware of the presence of choice atoms. Hence no specific arrangements have to be employed in the rule instantiation procedure. Furthermore, it is possible to directly apply to the instantiation of a choice element all the optimizations intervening in the instantiation of a rule, since its body is actually moved into a rule.

### Second Rewriting

Hereafter, we describe a second possibility, which is quite similar to the previous rewriting, but instead of removing choice rules, they are split up into multiple smaller choice rules. In particular, for each choice element in a choice atom, an auxiliary choice rule is generated in place of a disjunctive rule.

Formally, for each choice element,  $a_i(X_{1_i}, \dots, X_{k_i}) : l_{1_i}, \dots, l_{m_i}$  for  $i \in \{1, \dots, n\}$ , let  $\{Z_{1_i}, \dots, Z_{q_i}\} = var_g(B(r)) \cap var(l_{1_i}, \dots, l_{m_i})$ . Then a first auxiliary rule is produced as follows:

$$\{a_i(X_{1_i}, \dots, X_{k_i})\} :- l_{1_i}, \dots, l_{m_i}, b.$$

Next, in case  $m_i > 1$ , a second auxiliary rule consists in:

$$a'_i(Z_{1_i}, \dots, Z_{q_i}) :- a_i(X_{1_i}, \dots, X_{k_i}), s_1, \dots, s_r.$$

where  $a'_i$  is a fresh predicate and  $\{s_1, \dots, s_r\} \subseteq \{l_{1_i}, \dots, l_{m_i}\}$  is a possible empty list of saviours for the variables  $Z_{1_i}, \dots, Z_{q_i}$ . Eventually, if  $u \neq 0$  and  $\triangleright$  does not correspond to  $>=$ , the following constraint is added:

$$:- b, \#count\{ae\} \triangleright u.$$

where  $ae$  is a set of aggregate elements obtained as in the first rewriting reported above. If  $|B(r)| = 0$  in every generated rule, if  $b$  is present it is omitted.



As it might be observed, the same constraint used in the first rewriting guarantees the semantics of the comparison relation. Furthermore, similarly to the rewriting of aggregate literals, we are applying a standardization process to choice rules: each generated choice rule features at most a choice element. Notably, we are keeping the advantages stemming from the introduction of the rule  $r'$  by caching the join among the literals in  $B(r)$ , and from the generation of auxiliary rules for choice element bodies.

### Third Rewriting

Finally, we describe a more conservative rewriting, that still tends to standardize the form of choice rules, but without introducing further choice or disjunctive rules.

In detail, for each choice element  $a_i(X_{1_i}, \dots, X_{k_i}) : l_{1_i}, \dots, l_{m_i}$  for  $i \in \{1, \dots, n\}$ , let  $\{Z_{1_i}, \dots, Z_{q_i}\} = \text{var}_g(B(r)) \cap \text{var}(l_{1_i}, \dots, l_{m_i})$ , if  $m_i > 1$  two auxiliary rules are obtained as follows:

$$\begin{aligned} a'_i(Z_{1_i}, \dots, Z_{q_i}) &: -l_{1_i}, \dots, l_{m_i}, b. \\ a''_i(Z_{1_i}, \dots, Z_{q_i}) &: -a'_i(Z_{1_i}, \dots, Z_{q_i}), a_i(X_{1_i}, \dots, X_{k_i}). \end{aligned}$$

where  $a'_i$  and  $a''_i$  are fresh predicates. Next, we build a set of choice elements  $ce$  by adding to it:

- a choice element  $a_i(X_{1_i}, \dots, X_{k_i}) : a'_i(Z_{1_i}, \dots, Z_{q_i})$  for each choice element  $a_i(X_{1_i}, \dots, X_{k_i}) : l_{1_i}, \dots, l_{m_i}$  with  $m_i > 1$ ;
- a choice element  $a_i(X_{1_i}, \dots, X_{k_i}) : l_{1_i}$  for each choice element  $a_i(X_{1_i}, \dots, X_{k_i}) : l_{1_i}$  with  $m_i = 1$ ;
- a choice element  $a_i(X_{1_i}, \dots, X_{k_i}) : l_{1_i}$  for each choice element  $a_i(X_{1_i}, \dots, X_{k_i})$ , i.e.  $m_i = 0$ .

The set  $ce$  is then used to obtain the following choice rule:

$$\{ce\} \triangleright u : -b$$

Moreover, if  $u \neq 0$  and  $\triangleright$  does not correspond to  $\geq$ , let  $ae$  be a set of aggregate elements obtained by adding:

- an aggregate element  $Z_{1_i}, \dots, Z_{q_i} : a''_i(Z_{1_i}, \dots, Z_{q_i})$  for each choice element  $a_i(X_{1_i}, \dots, X_{k_i}) : l_{1_i}, \dots, l_{m_i}$  with  $m_i > 1$ ;
- an aggregate element  $X_{1_i}, \dots, X_{k_i} : a_i(X_{1_i}, \dots, X_{k_i})$  for each choice element with  $m_i \leq 1$ .

This set of aggregate elements is used within the following constraint:

$$: -b, \#count\{ae\} \triangleright u.$$

If  $|B(r)| = 0$  in every generated rule, if  $b$  is present it is omitted.

Basically, this technique applies a different standardization process where each choice element in a choice atom is rewritten in order to contain at most a literal in its body. Thus, additional benefits might derive from the fact that a smaller number of rules is introduced.

**Example 9.5.8.** Let us consider the rule  $r_1$ :

$$\{a(X, Y) : b(X, Y), c(Y, Z); d(W, T) : e(T)\} \geq 1 :- f(Z, W), g((Z + W), X).$$

According to the first rewriting, we obtain:

$$\begin{aligned} & \text{fresh\_pred\_1}(Z, W, X) :- f(Z, W), g((Z + W), X). \\ & a(X, Y) | - a(X, Y) :- b(X, Y), \text{fresh\_pred\_1}(Z, W, X), c(Y, Z). \\ & d(W, T) | - d(W, T) :- e(T), \text{fresh\_pred\_1}(Z, W, X). \\ & \text{fresh\_pred\_2}(Z, X, Y) :- c(Y, Z), a(X, Y). \\ & :- \text{fresh\_pred\_1}(Z, W, X), \\ & \quad \text{not } 1 \leq \#count\{Y, X, Z : \text{fresh\_pred\_2}(Z, X, Y); W, T : d(W, T)\}. \end{aligned}$$

If instead we apply the second proposed rewriting, we have:

$$\begin{aligned} & \text{fresh\_pred\_1}(Z, W, X) :- f(Z, W), g((Z + W), X). \\ & \{a(X, Y)\} :- b(X, Y), \text{fresh\_pred\_1}(Z, W, X), c(Y, Z). \\ & \{d(W, T)\} :- e(T), \text{fresh\_pred\_1}(Z, W, X). \\ & \text{fresh\_pred\_2}(Z, X, Y) :- c(Y, Z), a(X, Y). \\ & :- \text{fresh\_pred\_1}(Z, W, X), \\ & \quad \text{not } 1 \leq \#count\{Y, X, Z : \text{fresh\_pred\_2}(Z, X, Y); W, T : d(W, T)\}. \end{aligned}$$

Finally, with the third rewriting:

$$\begin{aligned} & \text{fresh\_pred\_1}(Z, W, X) :- f(Z, W), g((Z + W), X). \\ & \text{fresh\_pred\_2}(Z, X, Y) :- b(X, Y), c(Y, Z), \text{fresh\_pred\_1}(Z, W, X). \\ & a(X, Y) : \text{fresh\_pred\_2}(Z, X, Y); d(W, T) : e(T) :- \\ & \quad \text{fresh\_pred\_1}(Z, W, X). \\ & \text{fresh\_pred\_3}(X, Y, Z) :- a(X, Y), \text{fresh\_pred\_2}(Z, X, Y). \\ & :- \text{fresh\_pred\_1}(Z, W, X), \\ & \quad \text{not } 1 \leq \#count\{Z, Y, X : \text{fresh\_pred\_3}(X, Y, Z); T, W : d(W, T)\}. \end{aligned}$$



## Part III

# Implementing an Efficient ASP Grounder



In this part we illustrate the final purpose of this thesis, the development of an efficient ASP instantiator. In particular, we present the new released system *I-DLV*, and then we provide an experimental evaluation.

This part is composed of the following chapters:

- Chapter 10 presents *I-DLV*, describing its novel and characterizing features.
- Chapter 11 focuses on an experimental evaluation of the studied and proposed techniques on *I-DLV* performance.
- Eventually, Chapter 12 aims at assessing *I-DLV* overall performance as ASP grounder and deductive database engine. Moreover, we analyze its impact on solvers.

*I-DLV* has been presented in [28], and in the latest ASP Competition [69] *I-DLV* ranked both the first and second positions when combined, respectively, with an automatic solver selector [29] that inductively chooses the best solver depending on some inherent features of the instantiation produced, and with the state-of-the-art solver *clasp* [64].



## Chapter 10

# The New Grounder $\mathcal{I}$ -DLV

The new released grounder  $\mathcal{I}$ -DLV integrates all the techniques designed and introduced in this thesis. The system, besides being a stand-alone grounder and deductive database engine, along with the solver *wasp* [3] constitutes the new version of DLV, namely *DLV2*, recently released [2]. Moreover,  $\mathcal{I}$ -DLV is an open-source project: its source and binaries are available from the official repository [38].

As described in the previous chapters, the most popular grounding techniques reported in literature have been questioned with the aim of improving them, and new techniques have been introduced; to this end,  $\mathcal{I}$ -DLV was initially born as a tool to experiment with grounding techniques. It has been designed and engineered aiming at building a modern and efficient ASP instantiator, natively supporting the ASP-Core-2 standard language, and endowed with a lightweight modular design for easing the incorporation of optimization techniques and future updates. Notably, its grounding process is performed in a highly flexible and customizable way, allowing users to set up directly from the external how it has to be performed. For the actual implementation, we started completely from scratches choosing the C++ language, as it is particularly suitable and commonly adopted in contexts in which efficiency is crucial.

It is marked by dramatically improved performance with respect to the old DLV grounder module, and differently from it,  $\mathcal{I}$ -DLV interoperates with solvers and other ASP systems and tools, thanks to the fully compliance to the ASP-Core-2 standard and the capability to format the output in different ways, including the numeric format required by state-of-the-art solvers *wasp* [3] and *clasp* [65]. Furthermore,  $\mathcal{I}$ -DLV is also a full-fledged deductive database system, maintaining one of the feature that made DLV distinguishing.

In this chapter  $\mathcal{I}$ -DLV is presented highlighting its distinguishing features. Section 10.1 provides a description of the  $\mathcal{I}$ -DLV high-level architecture. Section 10.2 focuses on  $\mathcal{I}$ -DLV typical work flow and on the integration of the optimizations introduced in this thesis. Finally, in Section 10.3 we present some characterizing features of  $\mathcal{I}$ -DLV.



## 10.1 Architecture

The core strategies rely on a bottom-up evaluation based on a semi-naive approach (cf. Section 5.2).

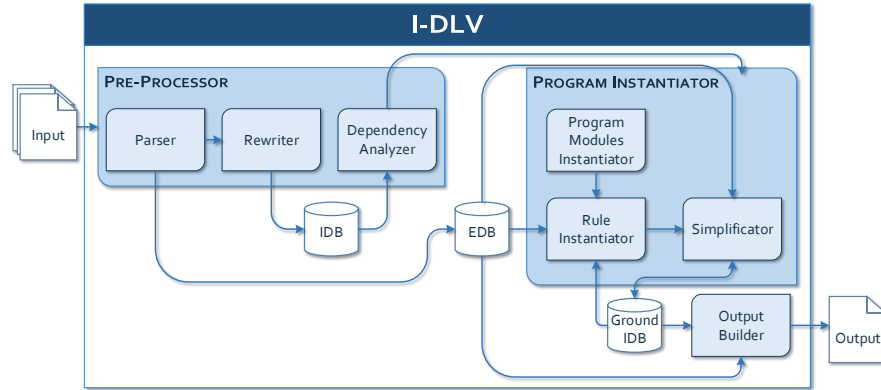


Figure 10.1:  $\mathcal{I}$ -DLV Architecture

Figure 10.1 depicts  $\mathcal{I}$ -DLV high-level architecture. The PRE-PROCESSOR module parses the input program  $P$  and builds the extensional database  $EDB$  from facts in appearing  $P$ ; then, the Rewriter produces the intensional database  $IDB$  from the rules. The Dependency Analyzer examines  $IDB$  rules and predicates, identifying program modules and a proper ordering for incrementally evaluating them according to the definitions described in Section 5.2.1.

The PROGRAM INSTANTIATOR grounds the program; the process is managed by the Program Modules Instantiator, that, applying a semi-naïve schema, evaluates one module at a time according to the order provided by the Dependency Analyzer.

The core of the computation is performed by the Rule Instantiator: given a (non-ground) rule  $r$  and a set of ground atoms  $S$  representing predicate extensions, it generates the ground instances of  $r$ , finding proper substitutions for the variables. The set  $S$  is dynamically computed: initially, it contains only  $EDB$  and from then on it is extended by the ground atoms occurring in the head of the newly generated ground rules. Ground rules are also analysed by the Simplificator, in order to check if some can be simplified or even eliminated, still guaranteeing semantics. Eventually, the output is gathered and properly arranged by the Output Builder. In particular, the Output Builder is able to format the output in different ways, including the numeric format required by state-of-the-art solvers *wasp* and *clasp*. The produced ground program will have the same answer sets of the full theoretical instantiation, yet being possibly smaller.

## 10.2 Overall Instantiation Process

Hereafter, we provide a general overview of the instantiation process of  $\mathcal{I}$ -DLV, focusing on how optimizations have been integrated in  $\mathcal{I}$ -DLV and outlining the order in which they intervene. Their synergy is the key of  $\mathcal{I}$ -DLV efficiency:

each of them aim to different extents at optimizing the instantiation process.

At the very beginning, if the input program contains a query,  $\mathcal{I}$ -DLV employs a rewriting technique geared towards efficient query answering, namely the *magic-sets* technique. It has been originally defined in [8] for non-disjunctive Datalog (i.e. with no function symbols) queries only, and afterwards many generalizations have been proposed. In particular,  $\mathcal{I}$ -DLV integrates an adapted version proposed in [4]. Essentially, the aim of this method is to simulate the top-down evaluation of a query: the original program is modified in order to narrow the computation to what is relevant to answer the query. Intuitively, the goal is to use the constants appearing in the query to reduce the size of the instantiation by eliminating *a priori* a number of ground instances of the rules which cannot contribute to the (possible) derivation of the query goal. Basically, given an input program, the binding information for *IDB* predicates which would be propagated during a top-down computation are materialized by means of proper adornments of the input predicates. Hence, the adorned program is used to generate a set of *magic rules*, which single out the atoms potentially relevant for the input query. The adorned rules are modified by adding magic atoms in the rule bodies: this limits the range of the head variables avoiding the inference of facts which cannot contribute to deriving the query. Eventually, a magic version of the query goal is produced, that will “trigger” the bottom-up generation.

**Example 10.2.1.** To show the effectiveness of the technique, let us consider the program  $P_{reach}$  constituted by the encoding for the *Reachability* problem reported in Section 3.2, along with the ground query  $reachable(1, 5)?$  and the following facts encoding a graph:

$$\begin{aligned} &edge(1, 2). \quad edge(1, 3). \quad edge(3, 4). \quad edge(4, 5). \quad edge(2, 6). \\ &edge(6, 7). \quad edge(3, 6). \quad edge(7, 8). \quad edge(8, 6). \quad edge(2, 8). \end{aligned}$$

In order to answer, without the Magic Sets technique, the system should first complete the instantiation, and then check whether the query atom is actually contained in the unique answer set. Such answer set contains 25 instances of the predicate *reachable*, while not all of them are (possibly) relevant in order to answer the query.

The application of the magic-set technique would produce the following rewritten program, instead:

$$\begin{aligned} &magic\_reachable^{bb}(Z, Y) :- edge(X, Z), \\ &\quad magic\_reachable^{bb}(X, Y). \\ &reachable(X, Y) :- edge(X, Y), \\ &\quad magic\_reachable^{bb}(X, Y). \\ &reachable(X, Y) :- magic\_reachable^{bb}(X, Y), \\ &\quad edge(X, Z), reachable(Z, Y). \\ &magic\_reachable^{bb}(1, 5). \end{aligned}$$

In this rewriting, the extension of  $magic\_reachable^{bb}$  represents the set of start- and end-nodes of all potential sub-paths of paths from 1 to 5. Therefore, when answering the query via a bottom-up computation, only these sub-paths will be actually considered: in this case, just 8 instances of  $magic\_reachable^{bb}$ , leading to the generation of only 3 instances of *reachable*.

Next, rules possibly undergo to the rewriting that removes isolated variables, described in Section 9.2.2. In many cases this reduces the size of the ground program, and consequently the instantiation time; nevertheless, to mitigate the possible overhead that, even if negligible in general, might become significant when the benefits of the projection are limited, such projection rewriting can be disabled on request.

In addition, rules containing aggregate literals are rewritten as discussed in Section 9.5.3, while choice rules are rewritten by applying the second rewriting defined in Section 9.5.4, by default.

As further strategy long-body rules are rewritten by splitting them up into multiple smaller ones, according to the *decomposition rewriting* illustrated in Chapter 8. In particular,  $\mathcal{I}$ -DLV embeds the SMARTDECOMPOSITION version tailored on the improvement of grounding performance illustrated in Section 11.3. For each input rule,  $\mathcal{I}$ -DLV selects the best estimated decomposition among multiple ones; by default  $\mathcal{I}$ -DLV performs five iterations, and stops the generation of tree decompositions after three iterations without improvements. These limits, that can be customized by proper command-line options, have been experimentally obtained by observing that no advantages arise for the grounding performance if larger limits are used. Moreover, by default the threshold value has been set to 0.5. In order to obtain this value, we decided to experimentally test the effects of the choices under several values and found that decomposition is preferable when the threshold is set to 0.5. We plan to further improve the choice of the threshold by taking advantage from automatic and more advanced methods, such as machine learning guided machineries. For the actual implementation, we relied on the open-source C++ library `htd` [1]<sup>1</sup>, an efficient and flexible library allowing to obtain tree decompositions and, importantly to customize them via user-provided fitness function. Moreover, thanks to the fitness-function mechanism we were able to associate to each computed decomposition its cost estimation, and selects the best one accordingly.

The order of literals in the rule bodies is analysed and possibly changed.  $\mathcal{I}$ -DLV implements all the body ordering strategies presented in Chapter 7, as long as a basic strategy ensuring the correct instantiation, and a variant of the *Combined*<sup>+</sup> that push literals with functional terms down in the body, which is intended to be employed whenever the rewriting of functional terms is disabled. By default,  $\mathcal{I}$ -DLV employs the *Combined*<sup>+</sup> criterion, the other strategies can be enabled on request.

Once the non-ground rules have been “adapted”, they undergo the rule instantiation, based on a backjumping search [111]. The process is further optimized by making use of indexing techniques for the retrieval of matching instances from the predicate extensions. By default,  $\mathcal{I}$ -DLV employs the *balanced on-demand indexing strategy* illustrated in Chapter 6. However, this behaviour can be updated by specifying for each atom the indexing structure to employ, as we will see in the following sections.

The search for an “agreement” between body literals on variable substitutions is further eased by means of the *aligning substitutions* and *look ahead* techniques, described in Section 9.3. The benefits are maximized when the sets of substitutions differ significantly, thus they can be enabled on demand.

In addition, during the rule instantiation,  $\mathcal{I}$ -DLV makes use the *pushing*

<sup>1</sup><https://github.com/mabseher/htd>

*down selections* optimization in order to faster determine inconsistencies (cf, Section 9.1).

Moreover, in case the projection rewriting involving isolated variables is disabled, whenever isolated variables occurring in atoms over unsolved predicates, its effects can be virtually simulated as well during the instantiation by means of the *isolated variables filtering* mechanism discussed in Section 9.2.1.

The output of the rule instantiation process is a set of ground instances of the rule at hand. The size of the output is further reduced by examining the produced ground rules and possibly simplifying, or even eliminating them. Indeed, body literals which are already known to be true can be safely dropped. Moreover, once the rule instance has been created, when some negative literal already known to be false occurs in the body, the rule instance is already (trivially) satisfied: it does not contribute to the semantics of the ground program, and it is removed. Notably during this simplification process, *I-DLV* examines all type of literals, trying to reduce the size of the output ground program as much as possible. In case the input program is non-disjunctive and stratified w.r.t. negation, the modular evaluation guided by the PROGRAM MODULES INSTANTIATOR along with the simplification mechanism allows *I-DLV* to completely evaluate the input program: the output consists of a set of facts, that correspond to the unique answer set of the program. In such cases, there is no need to rely on a further solving module.

In addition, *I-DLV* anticipates strong constraints grounding according to the *anticipating strong constraints evaluation* technique depicted in Section 9.4, which coupled with the simplification mechanism, whenever all literals appearing into the body of a ground constraint can be removed may lead to an empty-body constraint. By definition, such constraints are always violated; thus, the input program is incoherent, and the grounding process can be safely aborted.

## 10.3 Customizability and Further Features

As already stated, one of the main goals of the *I-DLV* project is to obtain a novel, flexible tool to experiment with ASP and its applications; to this end, it has been designed in order to allow a fine-grained control over the whole computational process, both via command-line options and inline annotations.

### 10.3.1 Command-line Customization

In what follows, we describe the most relevant options that can be set via command-line in order to customize the behaviour of *I-DLV*.

-- indexing

*I-DLV* allows to control the indexing strategy, thus providing the user with a mean to handle situations where the default behaviour is not satisfactory. In particular, the indexing module can set per each predicate in the program a single- or multiple-index, on the desired arguments. To this end, when more than two arguments are specified, *I-DLV* employs *generalized indices* (cf. Section 6.1.1). For instance with --indexing=p/4=0,1,2;p/2=1 it is set up that for the predicate *p/4* *I-DLV* should use an index on the first three arguments,

while for  $p/2$  an index on the first argument. Before applying the indexing strategy defined by users,  $\mathcal{I}$ -DLV checks if it can be applied, and whenever non indexable arguments are specified  $\mathcal{I}$ -DLV adopts the default indexing strategy.

#### -- ordering

As for body ordering, the user can currently choose among the following alternatives via `-- ordering=n`, where `n` can be:

- 0 : a basic ordering that aims at preserving the original literals positions in the rule, possibly rearranging them in order to guarantee a correct instantiation;
- 1 : a variant of the *DLV Combined* criterion [91], where classical atoms are placed according to the criterion, while the other types of literal tend to be placed down in the body;
- 2 : the *Combined<sup>+</sup>* criterion (enabled by default);
- 3 : an enhanced version of the *Combined<sup>+</sup>* criterion that pushes literals with functional terms down in the body;
- 4 : the *Combined<sub>I</sub><sup>+</sup>* criterion that tries to improve the quality of available indices;
- 5 : the *Combined<sub>B</sub><sup>+</sup>* criterion that works in synergy with backjumping in order to facilitate it;
- 6 : the *Combined<sub>IB</sub><sup>+</sup>* criterion combining the latter two.

#### -- decomp and related options

The decomposition rewriting can be performed in several modes:

- `-- decomp=0` specifies that, for every rule, once that the best decomposition has been determined, it has to be applied even if not estimated as convenient;
- `-- decomp=1` is the default setting, thus enables the “smart” decomposition mechanism described in Chapter 8;
- `-- decomp=2` disable completely the decomposition rewriting.

As already discussed, in general, obtaining a tree decomposition with the optimal tree-width, is intractable. The library `htd`, on which we relied to actual construct tree decompositions, implements several decomposition heuristic approaches [1]. The option `-- decomp-algorithm=n` allows to set the decomposition strategy to be used in order to produce tree decompositions, where the permitted values for `n` are:

- 0 : min-degree (default);
- 1 : max-cardinality;
- 2 : min-fill;

3 : natural-ordering.

A detailed description of tree decomposition techniques is given in the survey [79]. The default threshold can be changed with `--decomp-threshold`. The fixed number of iterations performed by *I-DLV* can be customized with the option `--decomp-iterations`; in addition as soon as are executed a number of iterations without producing a “better” decomposition, the generation process is also stopped: this non-improvement limit can be updated with `--decomp-limit`.

`--no-projection`

The projection rewriting of isolated variables and functional terms, both enabled by default, can be disabled: with `--no-projection=0` they are both disabled, with `--no-projection=1` it is disabled the rewriting of isolated variables and enabled solely the rewriting of functional terms, and viceversa `--no-projection=2` it is disabled only the projection of functional terms.

`--no-isolated-filter`

Furthermore, the filter mechanism for isolated variables appearing within literals over solved predicates, enabled by default, can be disabled, via the option `--no-isolated-filter`.

`--align-substitutions and --look-ahead`

Conversely, with `--align-substitutions` the technique that aligns variable substitutions can be enabled at will: it is disabled by default, given that its benefits strictly depend from the distribution of the input data. Similarly, with `--look-ahead` the look-ahead mechanism can be enabled.

`--choice-rewriting`

Concerning choice rules, the user can choose among all rewriting techniques introduced in Section 9.5.4. In particular, one can ask for the rewriting approach that makes use of disjunction and removes them from the program, with `--choice-rewriting=0`, or the other two more conservative approaches, with `--choice-rewriting=1` or `--choice-rewriting=2`.

`--no-magic-sets and --query`

*I-DLV* is able to process both ground and non-ground queries. By default, it simply produces the instantiation; via the option `--query`, for non-disjunctive and stratified programs, that are completely evaluated by the system, it can directly provide the query answer. The magic-set technique, enabled by default, can be disabled with `--no-magic-sets`.

`--gstats`

For advanced users, insights on the grounding process might be of great interest: they are available via a number of statistics that can be produced on demand. In particular, the following information are provided for each input rule: total instantiation time, number of produced ground instances, number of iterations required for instantiation (in case the rule is recursive); in addition, size of extension and selectivity of all arguments are reported, for each predicate in the rule.

### 10.3.2 Inline Annotations

Besides the command line, system customization and tuning is further eased by a new special feature of *I-DLV*: *annotations* of ASP code. Annotations and meta-data have been applied in different programming paradigms and languages; Java annotations<sup>2</sup>, for instance, have no direct effect on the code they annotate: a typical usage consists in analysing them at runtime in order to change the code behaviour. Some sorts of annotations have been proposed also for declarative paradigms, although to different extents and purposes with respect to our setting; a more detailed discussion is carried out in Chapter 13.

In *I-DLV* annotations allow to give explicit directions on the internal computational process. In particular, supported annotations belong to two categories: *grounding annotations* allowing for a fine-grained customization on the grounding process, and *solving annotations* that have been integrated into *DLV2*, and are geared to the customization of the whole computational process.

Syntactically, all annotations start with the prefix “%@” and end with a dot (“.”). Annotations do not change the semantics of input programs, their impact might be observed just on the performance. For this reason, their notation starts with %, which is used for comments in ASP-Core-2, so that other systems can simply ignore them.

#### Grounding Annotations

These annotations allow customize the *I-DLV* grounding process at a more fine-grained level with respect to the command-line options: they “annotate” the ASP code in a Java-like fashion, while embedded in comments: hence, the resulting programs can still be given as input to other ASP systems that do not support them, without any modification. In particular, our annotations can have two different scopes: at the global level, meaning that they are applied to the whole program, or at the rule level, and hence annotations act just on the rule they precede. In detail, the annotations currently supported are meant for customizing two of the major aspects of the grounding process, such as *body ordering*, *indexing*, as well as optimizations for which enabling or disabling them at rule level might have significant effects on the performance, such as *projection rewriting of isolated variables*, *functional terms rewriting*, *arithmetical terms rewriting*, *aligning substitutions*, and the *look-ahead* technique.

A specific body ordering strategy can be explicitly requested for any rule, simply preceding it with the line:

```
%@rule_ordering(n).
```

---

<sup>2</sup><https://docs.oracle.com/javase/tutorial/java/annotations/>

where  $n$  is a number representing an ordering strategy (cf. Section 10.3.1). In addition, it is possible to specify a particular partial order among body literals of a rule  $r$ , no matter the employed ordering strategy, by means of **before** and **after** directives and according to the following syntax:

$$\begin{aligned} \%@rule\_partial\_order(@before = \{l_1, \dots, l_n\}, \\ @after = \{\bar{l}_1, \dots, \bar{l}_m\}). \end{aligned}$$

where  $\{l_1, \dots, l_n\} \subseteq B(r)$  and  $\{\bar{l}_1, \dots, \bar{l}_m\} \subseteq B(r)$  for  $n, m \geq 0$  and  $\{l_1, \dots, l_n\} \cap \{\bar{l}_1, \dots, \bar{l}_m\} = \emptyset$ , i.e. are disjoint. As it might be expected, these partial orderings are respected only if valid. For instance, if for the rule

$$:- b(X, Y), not\ a(X).$$

an annotation constraining that the literal *not a(X)* is placed before  $b(X, Y)$ , cannot be applied.

**Example 10.3.1.** For instance, assume to have the following situation:

$$\begin{aligned} \%@rule\_partial\_order(@before = \{b(X, Y), X = \#count\{Z : d(Z)\}\}, \\ @after = \{c(X, Y)\}). \\ a(X) :- b(X, Y), c(X, Y), X = \#count\{Z : d(Z)\}. \end{aligned}$$

Then, the rule body is ordered as:  $b(X, Y), X = \#count\{Z : d(Z), c(X, Y)$ .

As for indexing, directives on a per-atom basis can be given for a rule  $r$  as shown next:

$$\%@rule\_atom\_indexed(@atom = \bar{a}, @arguments = \{a_1, \dots, a_k\}).$$

where  $\bar{a}$  is a classical atom occurring in  $B(r)$ , and  $\{a_1, \dots, a_k\}$  a set of arguments. Clearly, supposing that  $\bar{a}$  is an atom over a predicate  $p/n$ , for each argument  $a_i \in \{a_1, \dots, a_k\}$  it should hold that  $0 \leq a_i \leq n - 1$ : we followed a common notation in computer science, so that the first arguments corresponds to the argument 0 and the others follows. In addition, as done for the command-line indexing option (cf. Section 10.3.1), before applying the annotation, *I-DLV* checks that the specified arguments are actually indexable.

**Example 10.3.2.** For instance, assume to have the following situation:

$$\begin{aligned} \%@rule\_atom\_indexed(@atom = c(X, Y, Z, W), @arguments = \{0, 1\}). \\ a(W) :- b(X), b(Y), b(Z), c(X, Y, Z, W). \end{aligned}$$

Then, for the atom  $c(X, Y, Z, W)$  *I-DLV* employs a single-double index created on the first and second arguments.

The projection rewriting can be customized for any rule, by preceding it with the line:

$$\%@rule\_projection(n).$$

where  $n$  can either 0, 1 or 2, as for the command-line projection option described in Section 10.3.1.



The rewriting of arithmetic terms, disabled by default, can be enabled for any rule, by specifying before it the following annotation:

```
%@rule_rewrite_arith().
```

Similarly, the *aligning substitutions* technique disabled by default, can be enabled for a specific rule by preceding it with:

```
%@rule_align_substitutions().
```

while, for enabling the *look-ahead* technique, the annotation to use is the following:

```
%@rule_look_ahead().
```

Multiple preferences can be expressed via different annotations; in case of conflicts, priority is given to the first appearing in the program. In addition, preferences can also be specified at a global scope, by replacing the `rule` directive with the `global` one. Such kind of annotations are applied on the rules, if possible. While a `rule` annotation must precede the intended rule, `global` annotations can appear at any line in the input program. Both `global` and `rule` annotations can be expressed in the same program; in case of overlap on a particular rule/setting, priority is given to the more specific `rule` ones.

### Solving Annotations

Concerning solving side, from industrial and real world applications of ASP it emerged that the possibility to customize the internal computational process of ASP systems by specifying domain-specific heuristics leads to solve hardest instances of problems that are out of reach for state-of-the-art ASP solvers [48]. For this reason, originally, the solver *wasp* has been endowed with a Python interface, allowing users to specify via python scripts domain heuristics acting over sets of ground literals of interests, and thus to define how the solver should act on these literals.

The integration of  $\mathcal{I}$ -DLV and *wasp* within *DLV2* [2] yields to a simpler and more effective specification of such heuristics, since users are free from the burden of specifying ground literals. Rather, in *DLV2*, by means of an annotation, users can simply specify a python file defining a heuristic, along with a non-ground set of literals of interests, where each literal is associated with a tuple of terms, on which the heuristic is intended to act. Then, internally and transparently from the user point of view,  $\mathcal{I}$ -DLV is in charge of grounding these sets and providing them to *wasp*. Further details are reported in [2, 48].

**Example 10.3.3.** For example, a heuristic saved in *heuristic.py* and acting on literals  $p(X)$  can be linked to a program by the following global annotation:

```
%@global_heuristic(@file = "heuristic.py", @elements = {X : p(X)}).
```

### 10.3.3 Further Features

As additional features, *I-DLV* has been endowed with means to ease the interoperability with external sources of knowledge [30].

In particular, *I-DLV* can import relations from a RDBMS by means of an `#import_sql` directive. For example, `#import_sql(DB, "user", "pass", "SELECT * FROM t", p)` is used to access database `DB` and imports all tuples from table `t` into facts with predicate name `p`. Similarly, `#export_sql` directives are used to populate specific tables with the extension of a predicate. In addition, `#import_local_sparql` and `#import_remote_sparql` directives are used to retrieve data from SPARQL data sources.

Furthermore, the input program can be enriched by external atoms of the form:

$$\&p(i_1, \dots, i_n; o_1, \dots, o_m)$$

where `p` is the name of a Python function,  $i_1, \dots, i_n$  and  $o_1, \dots, o_m$  ( $n, m \geq 0$ ) are input and output terms, respectively. For each instantiation  $i'_1, \dots, i'_n$  of the input terms, function `p` is called with arguments  $i'_1, \dots, i'_n$ , and returns a set of instantiations for  $o_1, \dots, o_m$ . For example, a single line of Python:

```
def rev(s): s[::-1]
```

is sufficient to define a function `rev` that reverse strings, and which can be used by a rule of the following form:

$$revWord(Y) :- word(X), \&rev(X; Y).$$



## Chapter 11

# Experimental Evaluations of the Optimizations

In this chapter we report the results of an experimental activity carried out to assess the effects on the performance of  $\mathcal{I}$ -DLV of the grounding techniques studied and introduced in this thesis. In particular, in order to highlight the impact of the diverse techniques we considered two sets of benchmarks: we relied on the whole Sixth and Fourth ASP Competition suites [67]. ASP Competitions are official events [34, 68] assessing ASP systems on challenging benchmarks, in order to promote state of the art techniques and language standards. The latest seventh competition was recently held at the 14th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR 2017), but the benchmarks employed are not publicly available at the time of writing.

The Sixth Competition features 28 problems and 20 different instances per each. As reported in Table 11.1, each problem can be of one following types: (i) *decision*, meaning that the encoding does not contain queries or weak constraints, (ii) *optimization*, i.e. weak constraints are present in the encoding (iii) *query*, the encoding presents a query.

In addition, we selected the benchmarks of the Fourth Competition (see Table 11.2), which features 26 problems and a number of instances varying from 10 to 30 per problem. As it might be observed in Table 11.1 many problems (the ones reported in bold) appear in the both competitions. However, in the latest competitions a large part of the encodings have been optimized; in particular, these problems are the ones marked with a \* symbol in Table 11.1. The choice of this further suite permits to assess performance also on less optimized encodings, on which obtaining good performance is even more challenging and having an efficient grounding process becomes even more crucial. The suites cover the whole syntax of ASP-Core-2 v.2.01c [23]. Moreover, an additional set of benchmarks, freely available and used in [17], has been selected to assess the decomposition rewriting (cf. Chapter 8) on challenging ASP programs with extremely long rules.

Experiments have been performed on a NUMA machine equipped with two 2.8GHz AMD Opteron 6320 processors and 128 GiB of main memory, running Linux Ubuntu 14.04.4 (kernel v.3.19.0-25). Binaries were generated with the GNU C++ compiler v.5.4.0. As for memory and time limits, we allotted 15

Problem	Type	# instances
<b>Abstract Dialectical Frameworks</b>	Optimization	20
Combined Configuration	Decision	20
<b>Complex Optimization*</b>	Decision	20
<b>Connected Still Life</b>	Optimization	20
Consistent Query Answering	Query	20
<b>Crossing Minimization*</b>	Optimization	20
<b>Graceful Graphs</b>	Decision	20
<b>Graph Coloring*</b>	Decision	20
<b>Incremental Scheduling*</b>	Decision	20
<b>Knight Tour with Holes*</b>	Decision	20
<b>Labyrinth</b>	Decision	20
<b>Maximal Clique*</b>	Optimization	20
MaxSAT	Optimization	20
<b>Minimal Diagnosis*</b>	Decision	20
<b>Nomystery*</b>	Decision	20
Partner Units	Decision	20
<b>Permutation Pattern Matching*</b>	Decision	20
<b>Qualitative Spatial Reasoning*</b>	Decision	20
<b>Reachability</b>	Query	20
<b>Ricochet Robots</b>	Decision	20
<b>Sokoban*</b>	Decision	20
<b>Stable Marriage*</b>	Decision	20
Steiner Tree	Optimization	20
<b>Strategic Companies</b>	Query	20
System Synthesis	Optimization	20
Valves Location	Optimization	20
Video Streaming	Optimization	20
<b>Visit-all*</b>	Decision	20

Table 11.1: Sixth Competition Suite: Problems Description

Problem	Type	# instances
Abstract Dialectical Frameworks	Optimization	30
Bottle Filling Problem	Decision	30
Chemical Classification	Decision	30
Complex Optimization	Decision	29
Connected Still Life	Optimization	10
Crossing Minimization	Optimization	30
Graceful Graphs	Decision	30
Graph Coloring	Decision	30
Hanoi Tower	Decision	30
Incremental Scheduling	Decision	30
Knight Tour with Holes	Decision	30
Labyrinth	Decision	30
Maximal Clique	Optimization	30
Minimal Diagnosis	Decision	30
Nomystery	Decision	30
Permutation Pattern Matching	Decision	30
Qualitative Spatial Reasoning	Decision	30
Reachability	Query	30
Ricochet Robots	Decision	30
Sokoban	Decision	30
Solitaire	Decision	27
Stable Marriage	Decision	30
Strategic Companies	Query	30
Valves Location Problem	Optimization	30
Visit-all	Decision	30
Weighted-Sequence Problem	Decision	30

Table 11.2: Fourth Competition Suite: Problems Description

GiB and 600 seconds for each tested version of  $\mathcal{I}$ -DLV, per each single run.

The chapter is structured in several sections. In each section, the impact of a different optimization or a group of optimizations strictly related is analysed.

To this end, for each optimization we consider its impact on the performance of  $\mathcal{I}$ -DLV fixing the setting for the other optimizations and varying the configuration of the technique at hand by enabling, disabling or performing it in different modalities.

## 11.1 Indexing Strategies

In this section we analyse the impact of indexing techniques (see Chapter 6), comparing four different versions of  $\mathcal{I}$ -DLV. In the first version  $\mathcal{I}$ -DLV-*No-I* the grounder does not employ indices: instances are stored array-based structures (C++ STL vectors) and to retrieve them a linear search is performed. The second version  $\mathcal{I}$ -DLV-*GI* is based on an on-demand strategy that employs generalized indices created over all indexable arguments available implemented via C++ STL `unordered_maps`. In the third tested version, denoted  $\mathcal{I}$ -DLV-*SI*,  $\mathcal{I}$ -DLV employs an on-demand indexing strategy based on single indices, i.e. indices over a single argument. In particular, for this version we readapted the same mechanism adopted for single-double indices; thus, we enabled perfect hashing and as underlying data structure we used a single map (without a nested map), by using a C++ STL `unordered_map`. As last version we test the default indexing strategy of  $\mathcal{I}$ -DLV,  $\mathcal{I}$ -DLV-*SDI*, i.e. the balanced on-demand indexing strategy based on single-double indices; technically, we employed a C++ STL `unordered_map` in which it is nested a C++ STL `unordered_multimap`.

Problem	<i>I-DLV-No-I</i>		<i>I-DLV-GI</i>		<i>I-DLV-SI</i>		<i>I-DLV-SDI</i>	
	#grounded	time	#grounded	time	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	20	0.19	20	0.11	20	0.11	20	0.11
Combined Configuration	18	84.10	20	13.70	20	13.10	20	13.53
Complex Optimization	4	27.57	20	68.14	20	66.88	20	66.83
Connected Still Life	20	0.11	20	0.10	20	0.10	20	0.10
Consistent Query Answering	0	TO	20	79.96	20	82.46	20	76.31
Crossing Minimization	20	0.10	20	0.10	20	0.10	20	0.10
Graceful Graphs	20	1.15	20	0.30	20	0.29	20	0.31
Graph Coloring	20	0.10	20	0.10	20	0.10	20	0.10
Incremental Scheduling	14	151.26	20	16.85	20	16.65	20	17.05
Knight Tour With Holes	19	188.96	20	3.23	20	4.02	20	2.36
Labyrinth	20	61.33	20	1.10	20	4.83	20	2.01
Maximal Clique	0	TO	20	5.23	20	5.23	20	4.39
MaxSAT	4	49.38	20	51.13	20	3.89	20	3.96
Minimal Diagnosis	6	304.65	20	4.27	20	5.07	20	5.19
Nomystery	19	41.85	20	3.71	20	4.29	20	4.16
Partner Units	20	2.57	20	0.43	20	0.42	20	0.43
Permutation Pattern Matching	12	141.88	20	138.82	20	139.14	20	135.04
Qualitative Spatial Reasoning	20	60.81	20	5.51	20	6.71	20	5.49
Reachability	0	TO	20	166.69	20	156.93	20	142.64
Ricochet Robots	20	1.73	20	0.33	20	0.39	20	0.37
Sokoban	19	36.25	20	1.49	20	1.29	20	1.23
Stable Marriage	0	TO	20	127.68	20	129.62	20	123.55
Steiner Tree	3	142.71	20	32.00	20	52.82	20	29.83
Strategic Companies	20	0.83	20	0.33	20	0.32	20	0.25
System Synthesis	20	68.49	20	37.58	20	1.29	20	1.12
Valves Location Problem	20	25.23	20	7.97	20	2.57	20	2.58
Video Streaming	20	0.10	20	0.10	20	0.10	20	0.10
Visit-all	20	3.68	20	1.25	20	1.24	20	1.22
<i>Total Grounded Instances</i>	<i>398/560</i>		<i>560/560</i>		<i>560/560</i>		<i>560/560</i>	
<i>Average Time</i>	<i>205.17</i>		<i>27.44</i>		<i>25</i>		<i>22.98</i>	

Table 11.3: Indexing – 6th Comp. Benchmarks

The results are reported in Tables 11.3 and 11.4: the first column reports problem names, the next four pairs of columns show the number of grounded

Problem	<i>I-DLV-No-I</i>		<i>I-DLV-GI</i>		<i>I-DLV-SI</i>		<i>I-DLV-SDI</i>	
	#grounded	time	#grounded	time	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	<b>30</b>	<b>0.21</b>	30	0.11	30	0.11	30	0.11
Bottle Filling Problem	<b>17</b>	<b>31.20</b>	<b>30</b>	<b>9.39</b>	30	4.01	30	4.15
Chemical Classification	<b>9</b>	<b>142.93</b>	<b>26</b>	<b>119.24</b>	<b>26</b>	<b>119.12</b>	<b>26</b>	<b>120.60</b>
Complex Optimization	<b>14</b>	<b>37.67</b>	<b>29</b>	<b>40.61</b>	<b>29</b>	<b>41.69</b>	<b>29</b>	<b>40.16</b>
Connected Still Life	10	0.12	10	0.12	10	0.12	10	0.12
Crossing Minimization	30	0.11	30	0.10	30	0.10	30	0.10
Graceful Graphs	<b>30</b>	<b>1.76</b>	<b>30</b>	<b>0.39</b>	<b>30</b>	<b>0.37</b>	<b>30</b>	<b>0.38</b>
Graph Colouring	30	0.10	30	0.11	30	0.10	30	0.10
Hanoi Tower	<b>30</b>	<b>0.38</b>	<b>30</b>	<b>0.24</b>	<b>30</b>	<b>0.24</b>	<b>30</b>	<b>0.24</b>
Incremental Scheduling	<b>1</b>	<b>439.81</b>	12	307.58	12	301.63	12	296.01
Knight Tour with Holes	<b>0</b>	<b>TO</b>	<b>20</b>	<b>177.75</b>	<b>16</b>	<b>168.41</b>	<b>20</b>	<b>173.50</b>
Labyrinth	<b>30</b>	<b>42.83</b>	<b>30</b>	<b>0.86</b>	<b>30</b>	<b>3.52</b>	<b>30</b>	<b>1.51</b>
Maximal Clique	<b>30</b>	<b>88.45</b>	<b>30</b>	<b>0.33</b>	<b>30</b>	<b>0.33</b>	<b>30</b>	<b>0.33</b>
Minimal Diagnosis	<b>22</b>	<b>213.63</b>	<b>30</b>	<b>2.09</b>	<b>30</b>	<b>2.56</b>	<b>30</b>	<b>2.53</b>
Nomystery	<b>24</b>	<b>67.28</b>	<b>30</b>	<b>36.02</b>	<b>30</b>	<b>41.92</b>	<b>30</b>	<b>43.68</b>
Permutation Pattern Matching	<b>25</b>	<b>36.83</b>	<b>28</b>	<b>59.27</b>	<b>28</b>	<b>59.33</b>	<b>28</b>	<b>58.17</b>
Qualitative Spatial Reasoning	<b>30</b>	<b>29.76</b>	<b>30</b>	<b>3.07</b>	<b>30</b>	<b>3.48</b>	<b>30</b>	<b>2.84</b>
Reachability	<b>0</b>	<b>TO</b>	<b>30</b>	<b>131.35</b>	<b>30</b>	<b>125.67</b>	<b>30</b>	<b>125.65</b>
Ricochet Robots	<b>30</b>	<b>1.05</b>	<b>30</b>	<b>0.24</b>	<b>30</b>	<b>0.28</b>	<b>30</b>	<b>0.25</b>
Sokoban	<b>26</b>	<b>84.26</b>	<b>30</b>	<b>3.77</b>	<b>30</b>	<b>2.85</b>	<b>30</b>	<b>2.66</b>
Solitaire	<b>27</b>	<b>0.44</b>	<b>27</b>	<b>0.12</b>	<b>27</b>	<b>0.21</b>	<b>27</b>	<b>0.13</b>
Stable Marriage	<b>17</b>	<b>480.30</b>	<b>30</b>	<b>29.18</b>	<b>30</b>	<b>28.98</b>	<b>30</b>	<b>27.98</b>
Strategic Companies	<b>30</b>	<b>0.75</b>	<b>30</b>	<b>0.32</b>	<b>30</b>	<b>0.30</b>	<b>30</b>	<b>0.30</b>
Valves Location	<b>30</b>	<b>48.94</b>	<b>30</b>	<b>13.99</b>	<b>30</b>	<b>4.12</b>	<b>30</b>	<b>4.02</b>
Visit-all	<b>30</b>	<b>0.26</b>	<b>30</b>	<b>0.14</b>	<b>30</b>	<b>0.14</b>	<b>30</b>	<b>0.14</b>
Weighted-Sequence Problem	30	2.90	30	2.87	30	2.82	30	2.84
<b>Total Grounded Instances</b>	<b>582/756</b>		<b>722/756</b>		<b>718/756</b>		<b>722/756</b>	
<b>Average Time</b>	<b>46.24</b>		<b>28.01</b>		<b>26.39</b>		<b>27.08</b>	

Table 11.4: Indexing – 4th Comp. Benchmarks

instances and the running time (in seconds) averaged over successfully grounded instances for each of the three ordering criteria. The last lines report the total number of grounded instances and the average running time computed over successfully grounded instances; “TO” stands for time outs. Some benchmarks are reported in bold indicating cases where there is a noticeable difference of at least 10% among the versions tested, and particularly bad results are highlighted in red, while positive cases are reported in green. In the rest of the chapter, we will adopt the same style for the other experimental results; moreover, we will indicate with “MO” memory outs, and with “US” unsupported syntax. We remark that since per each configuration the average running time is computed over successfully grounded instances a small average does not necessarily correspond to good performance as it has to be contextualized w.r.t. the total number of grounded instances.

As it might be expected, in both benchmarks it is evicted that disabling the indexing strategy is not a good choice: *I-DLV-No-I* is systematically worse than the other versions and solves a significant smaller number of instances. Enabling a single index results in a significant improvement: *I-DLV-SI* is significantly better than *I-DLV-No-I*, however, its performance is generally not comparable with the other two versions, especially in the Fourth Competition where *I-DLV-SI* solves a smaller number of instances. *I-DLV-SDI* and *I-DLV-GI* grounded the same number of instances, but *I-DLV-SDI* grounded them in a smaller average time, as it performed the best in most of the problems.

To grasp the intuition behind the differences in performance between single-double and indices, let us coming back to Example 6.1.2 experimenting the impact of variations in the data structures adopted. As already stated, annotations are intended to adapt the grounding process to user’s desiderata. Figure 11.1 shows the results of multiple executions of the following program  $P_1$  :

```

%@rule_ordering(0).
a(W) :- b(X, Y), c(Z), d(X, Y, Z, W).
c(1..10).
b(1..10, 1..10).
d(1..300, 1..200, 1..100, 1..10).

```

The annotation about the body ordering has been added to ensure that the rule body is not reordered, since the aim is to experiment with different indexing structures for  $d(X, Y, Z, W)$ . The labels of the horizontal axis represent the indexing arguments employed for the respective execution. In particular, while the label *def* corresponds the execution of  $P_1$  as reported above, i.e. with the default indexing strategy of  $\mathcal{I}\text{-DLV}$ , in the other cases an annotation of form:

```

%@rule_atom_indexed(@atom = d(X, Y, Z, W), @arguments = {Args})

```

has been added to  $P_1$ . For instance, the label (0) represents that *Args* has been set to 0, thus the atom has been indexed on the first argument only. In particular, for the executions in which a single indexing argument is used, we adopted the same strategy of the version  $\mathcal{I}\text{-DLV}\text{-SI}$ ; whenever there are two arguments, the strategy followed is the same of  $\mathcal{I}\text{-DLV}\text{-SDI}$ ; eventually, for the execution in which three arguments have been used, a generalized index has been employed, as in  $\mathcal{I}\text{-DLV}\text{-GI}$ .

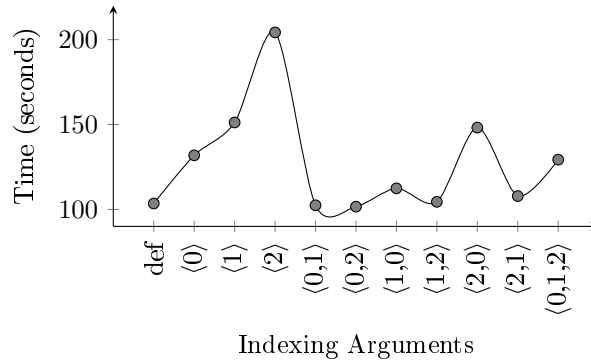


Figure 11.1: Variation of the indexing strategy via annotations

Notably, it emerges that, in general, indexing on three arguments, which are more likely to represent a primary key, is not the best choice. The main reason behind this behaviour is the intrinsic characteristic of the type of data structures employed. While a generalized index is more flexible, the perfect hashing mechanism employed for single and single-double indices tends to have a more positive impact on performance. For this reason,  $\mathcal{I}\text{-DLV}$  by default relies on these perfect-hashing structures, and allows users the possibility to override its default behaviour whenever its performance is not satisfactory.



## 11.2 Body Ordering

The body ordering strategies introduced in Chapter 7 have different impacts on the performance of  $\mathcal{I}$ -DLV. In this section, we compare all the implemented strategies, described in Section 10.3.1: the version  $\mathcal{I}$ -DLV-Ord-0 applies the body strategy 0, the version  $\mathcal{I}$ -DLV-Ord-1 applies the ordering 1, and so on, up to the version  $\mathcal{I}$ -DLV-Ord-6 that applies the strategy 6.

The results are shown in Tables 11.5 and 11.6. First of all, it is evident that selecting a suitable ordering strategy has significant benefits: the version  $\mathcal{I}$ -DLV-Ord-0 that employs a basic ordering with the only aim of ensuring the correct instantiation has the worst performance and solves the smallest number of instances, in general, in both benchmarks.

The  $Combined^+$  criterion ( $\mathcal{I}$ -DLV-Ord-2) performs better or equal than the  $Combined$  variant implemented ( $\mathcal{I}$ -DLV-Ord-1), evidencing no situations in which the  $Combined$  is better. The ordering that pushes down in the body literals with functional terms,  $\mathcal{I}$ -DLV-Ord-3, performs similarly to  $\mathcal{I}$ -DLV-Ord-2; this is because, in these executions the rewriting of functional terms has been enabled, and therefore their behaviours coincide. In Section 11.8 we will consider the impact of this strategy when the rewriting of functional terms is disabled.

The remaining variants of the  $Combined^+$  strategy show interesting results and variegated behaviours. We observe that among the three variants the version  $\mathcal{I}$ -DLV-Ord-5, that orders body by means of the  $Combined_B^+$  criterion and therefore tends to prefer literals binding relevant variables, shows some local improvements, but in both suites it does not ground all instances. Notably, in the Sixth Competition, the version  $\mathcal{I}$ -DLV-Ord-6 corresponding to the  $Combined_{IB}^+$  ordering, that combines the  $Combined_I^+$  and  $Combined_B^+$  strategies, mitigates the drawbacks observed for  $\mathcal{I}$ -DLV-Ord-5, while in the Fourth Competition  $\mathcal{I}$ -DLV-Ord-6 performed worse than  $\mathcal{I}$ -DLV-Ord-5, in general.  $\mathcal{I}$ -DLV-Ord-4, that applies the  $Combined_I^+$  ordering, performed very close to  $\mathcal{I}$ -DLV-Ord-2, especially in the Fourth Competition, where apart from the problem *Qualitative Spatial Reasoning*,  $\mathcal{I}$ -DLV-Ord-4 performs generally better w.r.t.  $\mathcal{I}$ -DLV-Ord-2.

In conclusion, we can observe that  $Combined^+$  tends to be more balanced, performing the best in the majority of situations. As already stated, by default in  $\mathcal{I}$ -DLV the  $Combined^+$  is employed. Its variants and the other orderings might be enabled on request to force that a particular aspect has to be considered, such as the correlation with indexing strategies or with the backjumping machinery. Indeed, as benchmarks evidenced, in some specific situations their ordering policies may be preferable. Thanks to annotations the strategies can be activated at rule level, and custom orderings can also be defined. Indeed, it is worth noting that acting at a global scope, as one could by setting via command-line option, may not have the same effect and not bring the same improvements, as the gain due to the change of strategy over some rules may be overshadowed by corresponding losses over the rest of the program; the flexible customization means featured by  $\mathcal{I}$ -DLV, that allow to configure and fine-tune the grounder as needed, even at a rule level, are exactly aimed at better dealing with such scenarios.

Problem	<i>I-DIV-Ord-0</i>		<i>I-DIV-Ord-1</i>		<i>I-DIV-Ord-2</i>		<i>I-DIV-Ord-3</i>		<i>I-DIV-Ord-4</i>		<i>I-DIV-Ord-5</i>		<i>I-DIV-Ord-6</i>	
	#grounded	time	#grounded	time	#grounded	time	#grounded	time	#grounded	time	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	20	0.14	20	0.11	20	0.11	20	0.11	20	0.11	20	0.11	20	0.12
Combined Configuration	20	28.66	20	28.17	20	13.82	20	13.79	20	13.85	20	13.84	20	13.82
Complex Optimization	4	36.95	20	67.58	20	67.59	20	67.23	20	38.07	20	68.81	20	39.27
Connected Skill Life	20	0.10	20	0.10	20	0.10	20	0.10	20	0.10	20	0.10	20	0.10
Consistent Query Answering	20	71.28	20	76.28	20	76.44	20	76.16	20	76.27	20	76.17	20	76.28
Crossing Minimization	20	0.10	20	0.10	20	0.10	20	0.10	20	0.10	20	0.10	20	0.10
Graphical Graphs	20	0.29	20	0.29	20	0.29	20	0.29	20	0.29	20	0.29	20	0.29
Graph Coloring	20	0.10	20	0.11	20	0.10	20	0.10	20	0.10	20	0.10	20	0.10
Incremental Scheduling	20	17.14	20	16.79	20	17.03	20	16.87	20	18.01	20	18.97	20	17.36
Knight Tour With Holes	20	2.15	20	2.29	20	2.31	20	2.29	20	2.33	20	2.18	20	2.22
Labyrinth	20	2.60	20	1.99	20	1.98	20	1.99	20	1.35	20	3.04	20	2.55
Maximal Clique	20	5.21	20	5.22	20	5.21	20	5.21	20	5.16	20	5.20	20	5.25
MaxSAT	20	3.95	20	4.25	20	4.27	20	4.29	20	4.27	20	4.30	20	4.26
Minimal Diagnosis	20	5.85	20	5.14	20	5.14	20	5.11	20	5.14	20	5.09	20	5.13
Nonptery	20	4.14	20	4.01	20	4.00	20	4.00	20	4.98	20	4.63	20	4.14
Parmer Units	20	0.41	20	0.41	20	0.41	20	0.41	20	0.42	20	0.41	20	0.42
Permutation Pattern Matching	20	132.21	20	132.76	20	134.70	20	133.22	20	138.19	20	162.14	20	142.13
Qualitative Spatial Reasoning	20	5.09	20	5.38	20	5.40	20	5.41	20	6.63	20	5.09	20	6.27
Reachability	20	119.50	20	157.92	20	157.71	20	158.34	20	156.74	20	148.09	20	142.69
Ricochet Robots	20	0.32	20	0.35	20	0.35	20	0.35	20	0.37	20	0.36	20	0.37
Sokoban	20	1.13	20	1.20	20	1.20	20	1.20	20	1.15	20	1.18	20	1.22
Stable Marriage	20	120.95	20	120.72	20	121.14	20	120.47	20	127.66	20	120.04	20	127.67
Steiner Tree	20	29.04	20	28.92	20	28.99	20	28.95	20	29.30	20	27.75	20	27.79
Strategic Companies	7	52.49	20	0.32	20	0.32	20	0.32	20	0.32	20	0.28	20	0.27
System Synthesis	20	1.10	20	1.10	20	1.10	20	1.10	20	1.11	20	1.10	20	1.08
Values Location Problem	20	2.68	20	2.55	20	2.58	20	2.58	20	2.56	20	2.58	20	2.56
Video Streaming	20	0.10	20	0.10	20	0.10	20	0.10	20	0.10	20	0.10	20	0.10
Visit-all	20	1.42	20	1.19	20	1.18	20	1.19	20	1.30	20	1.00	20	1.01
<b>Total Grounded Instances</b>	<b>531/560</b>		<b>560/560</b>		<b>560/560</b>		<b>560/560</b>		<b>560/560</b>		<b>560/560</b>		<b>560/560</b>	
<b>Average Time</b>	<b>21.9</b>		<b>23.76</b>		<b>23.35</b>		<b>23.77</b>		<b>23.21</b>		<b>27.65</b>		<b>22.81</b>	

Table 11.5: Body Ordering – 6th Comp. Benchmarks

Problem	I-DIV-Ord-0		I-DIV-Ord-1		I-DIV-Ord-2		I-DIV-Ord-3		I-DIV-Ord-4		I-DIV-Ord-5		I-DIV-Ord-6	
	#grounded	time	#grounded	time	#grounded	time	#grounded	time	#grounded	time	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	30	0.15	30	0.11	30	0.11	30	0.11	30	0.11	30	0.11	30	0.12
Bottle Filling Problem	30	4.11	30	4.14	30	4.13	30	4.14	30	4.08	30	4.13	30	4.11
Chemical Classification	26	97.43	26	120.54	26	120.62	26	120.31	26	121.93	26	121.98	26	123.79
Complex Optimization	14	40.89	29	40.42	29	40.85	29	39.94	29	17.90	29	40.59	29	18.21
Connected Still Life	10	0.15	10	0.12	10	0.12	10	0.12	10	0.12	10	0.18	10	0.18
Crossing Minimization	30	0.10	30	0.10	30	0.10	30	0.10	30	0.10	30	0.10	30	0.10
Graceful Graphs	30	0.37	30	0.38	30	0.37	30	0.38	30	0.37	30	0.37	30	0.37
Graph Colouring	30	0.10	30	0.10	30	0.10	30	0.10	30	0.10	30	0.10	30	0.10
Hanoi Tower	30	0.24	30	0.24	30	0.24	30	0.24	30	0.24	30	0.24	30	0.24
Incremental Scheduling	12	362.19	12	295.25	12	297.45	12	296.94	12	297.79	0	TO	1	589.02
Knight Tour with Holes	20	180.34	20	173.40	20	173.47	20	174.06	20	178.45	20	174.91	20	178.62
Labyrinth	30	2.00	30	1.50	30	1.51	30	1.51	30	1.07	30	2.32	30	1.97
Maximal Clique	30	0.33	30	0.33	30	0.33	30	0.33	30	0.33	30	0.33	30	0.33
Minimal Diagnosis	30	2.94	30	2.52	30	2.52	30	2.52	30	2.55	30	2.55	30	2.57
Nomystery	30	53.26	30	43.78	30	43.76	30	43.69	30	44.81	30	43.73	30	44.55
Permutation Pattern Matching	26	56.70	28	58.41	28	58.38	28	58.30	28	58.93	28	60.45	26	47.20
Qualitative Spatial Reasoning	30	2.81	30	2.83	30	2.84	30	2.84	30	3.37	30	3.37	30	3.37
Reachability	30	95.54	30	125.80	30	125.48	30	125.36	30	125.92	30	122.93	30	119.21
Ricochet Robots	30	0.24	30	0.25	30	0.25	30	0.25	30	0.25	30	0.25	30	0.25
Sokoban	30	2.54	30	2.65	30	2.67	30	2.66	30	2.49	30	2.57	30	2.61
Solitare	27	0.19	27	0.12	27	0.12	27	0.12	27	0.11	27	0.21	27	0.20
Stable Marriage	30	50.63	30	27.60	30	27.78	30	27.61	30	28.27	30	27.71	30	28.41
Strategic Companies	14	44.81	30	0.30	30	0.31	30	0.30	30	0.31	30	0.26	30	0.26
Valves Location	30	4.21	30	4.04	30	4.06	30	4.06	30	4.05	30	4.07	30	4.04
Visit-all	30	0.14	30	0.14	30	0.14	30	0.14	30	0.14	30	0.14	30	0.14
Weighted-Sequence Problem	30	2.98	30	2.84	30	2.83	30	2.84	30	2.88	30	2.61	30	2.63
<b>Total Grounded Instances</b>	<b>689/756</b>		<b>722/756</b>		<b>722/756</b>		<b>722/756</b>		<b>722/756</b>		<b>710/756</b>		<b>709/756</b>	
<b>Average Time</b>	<b>28.81</b>		<b>27.07</b>		<b>27.13</b>		<b>27.07</b>		<b>26.49</b>		<b>22.63</b>		<b>22.01</b>	

Table 11.6: Body Ordering – 4th Comp. Benchmarks

## 11.3 Decomposition Rewriting

Hereafter, we assess the impact of the customized version of SMARTDECOMPOSITION defined in Chapter 8 on the grounding performance of  $\mathcal{I}$ -DLV.

Three configurations have been compared: (i)  $\mathcal{I}$ -DLV-No-D,  $\mathcal{I}$ -DLV without any decomposition, (ii) *lpopt* (version 2.2, the latest available at the time of writing) combined in pipeline with  $\mathcal{I}$ -DLV (i.e. a black-box usage of *lpopt*), (iii)  $\mathcal{I}$ -DLV-SD, i.e.  $\mathcal{I}$ -DLV empowered with the customized version of SMARTDECOMPOSITION. In order to produce replicable results, the random seed used by *lpopt* for heuristics has been set to 0 for system (ii).

Table 11.7 shows the results on the benchmarks from the Sixth Competition; the dashes indicate that corresponding configurations do not support syntax.

Problem	<i>I</i> -DLV-No-D		<i>lpopt</i>	<i>I</i> -DLV-No-D		<i>I</i> -DLV-SD	
	#grounded	time		#grounded	time	#grounded	time
Abstract Dialectical Frameworks	20	0.11	20	0.11	20	0.13	
Combined Configuration	20	13.53	20	13.52	20	13.33	
<b>Complex Optimization</b>	20	<b>66.83</b>	20	<b>73.41</b>	20	<b>66.82</b>	
Connected Still Life	20	0.10	20	0.10	20	0.10	
<b>Consistent Query Answering</b>	<b>20</b>	<b>76.31</b>	<b>0</b>	<b>US</b>	<b>20</b>	<b>75.35</b>	
Crossing Minimization	20	0.10	20	0.10	20	0.10	
Graceful Graphs	20	0.31	20	0.32	20	0.32	
Graph Coloring	20	0.10	20	0.10	20	0.10	
Incremental Scheduling	20	17.05	20	16.77	20	16.55	
<b>Knight Tour With Holes</b>	20	<b>2.36</b>	20	<b>6.53</b>	20	<b>2.34</b>	
Labyrinth	20	2.01	20	1.83	20	2.02	
<b>Maximal Clique</b>	20	<b>4.39</b>	20	<b>20.70</b>	20	<b>4.31</b>	
MaxSAT	20	<b>3.96</b>	20	<b>8.92</b>	20	<b>3.90</b>	
<b>Minimal Diagnosis</b>	20	<b>5.19</b>	20	<b>4.36</b>	20	<b>4.79</b>	
Nomystery	20	<b>4.16</b>	20	<b>2.50</b>	20	<b>3.46</b>	
Partner Units	20	0.43	20	0.44	20	0.44	
<b>Permutation Pattern Matching</b>	20	<b>135.04</b>	20	<b>4.35</b>	20	<b>4.31</b>	
Qualitative Spatial Reasoning	20	5.49	20	5.47	20	5.48	
<b>Reachability</b>	<b>20</b>	<b>142.64</b>	<b>0</b>	<b>US</b>	<b>20</b>	<b>134.91</b>	
Ricochet Robots	20	0.37	20	0.40	20	0.39	
Sokoban	20	1.23	20	1.25	20	1.25	
<b>Stable Marriage</b>	20	<b>123.55</b>	20	<b>132.33</b>	20	<b>125.27</b>	
Steiner Tree	20	29.83	20	29.90	20	29.73	
<b>Strategic Companies</b>	<b>20</b>	<b>0.25</b>	<b>0</b>	<b>US</b>	<b>20</b>	<b>0.30</b>	
System Synthesis	20	1.12	20	1.13	20	1.11	
Valves Location Problem	20	2.58	20	2.61	20	2.66	
Video Streaming	20	0.10	20	0.10	20	0.10	
<b>Visit-all</b>	20	<b>1.22</b>	20	<b>0.45</b>	20	<b>0.44</b>	
<i>Total Grounded Instances</i>	<i>560/560</i>		<i>500/560</i>		<i>560/560</i>		
<i>Average Time</i>	<i>22.98</i>		<i>13.11</i>		<i>17.86</i>		

Table 11.7: Decomposition Rewriting – 6th Comp. Benchmarks

The results of the “blind usage” of *lpopt* (ii) are conflicting: for instance, it enjoys a great gain w.r.t. the “plain”  $\mathcal{I}$ -DLV while dealing with the *Permutation Pattern Matching* problem, and shows significant losses in other cases. On the other hand, in general, the “smart usage” of decomposition in  $\mathcal{I}$ -DLV-SD allows to avoid negative effects of the black-box decomposition mechanism, still preserving the positive ones, apart for the problems *Labyrinth* and *Nomystery*. The main reason is the interaction of the other rewriting techniques applied in  $\mathcal{I}$ -DLV with the decomposition rewriting. In particular, given a rule, the rewriting technique of isolated variables is currently applied before than the decomposition mechanism, and since these variables are removed from the body, the set of variables appearing in the rule body is changed and differences in the generated decompositions may be observed whenever the rewriting of isolated

variable is enabled, as in these executions. Eventually,  $\mathcal{I}\text{-DLV}\text{-SD}$  clearly outperforms  $\mathcal{I}\text{-DLV}\text{-No}\text{-D}$  and that the average grounding time of  $\mathcal{I}\text{-DLV}\text{-SD}$  over all instances is reduced up to 22%. A similar trend is observable in Table 11.8 about the Fourth Competition. In these benchmarks, the positive impact of SMARTDECOMPOSITION is even more evident, since  $\mathcal{I}\text{-DLV}\text{-SD}$  grounds a larger number of instances in a significant smaller average time. Intuitively, the less an encoding is fine-tuned, the highest may be the benefits stemming from a careful decomposition of input rules.

Problem	$\mathcal{I}\text{-DLV}\text{-No}\text{-D}$		$lpopt \mid \mathcal{I}\text{-DLV}\text{-No}\text{-D}$		$\mathcal{I}\text{-DLV}\text{-SD}$	
	#grounded	time	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	30	0.11	30	0.11	30	0.12
<b>Bottle Filling Problem</b>	<b>30</b>	<b>4.15</b>	<b>30</b>	<b>6.93</b>	<b>30</b>	<b>4.22</b>
<b>Chemical Classification</b>	<b>26</b>	<b>120.60</b>	<b>24</b>	<b>431.64</b>	<b>26</b>	<b>123.21</b>
Complex Optimization	29	40.16	29	43.27	29	41.25
Connected Still Life	10	0.12	10	0.12	10	0.12
Crossing Minimization	30	0.10	30	0.10	30	0.10
Graceful Graphs	30	0.38	30	0.41	30	0.40
Graph Colouring	30	0.10	30	0.10	30	0.10
Hanoi Tower	30	0.24	30	0.24	30	0.24
<b>Incremental Scheduling</b>	<b>12</b>	<b>296.01</b>	<b>17</b>	<b>227.89</b>	<b>21</b>	<b>222.39</b>
Knight Tour with Holes	20	173.50	20	183.68	20	180.36
Labyrinth	30	1.51	30	1.40	30	1.52
<b>Maximal Clique</b>	<b>30</b>	<b>0.33</b>	<b>30</b>	<b>1.12</b>	<b>30</b>	<b>0.31</b>
Minimal Diagnosis	30	2.53	30	2.21	30	2.33
<b>Nomystery</b>	<b>30</b>	<b>43.68</b>	<b>21</b>	<b>102.76</b>	<b>30</b>	<b>41.96</b>
<b>Permutation Pattern Matching</b>	<b>28</b>	<b>58.17</b>	<b>30</b>	<b>3.65</b>	<b>30</b>	<b>3.80</b>
Qualitative Spatial Reasoning	30	2.84	30	2.94	30	2.93
<b>Reachability</b>	<b>30</b>	<b>125.65</b>	<b>0</b>	<b>US</b>	<b>30</b>	<b>109.12</b>
Ricochet Robots	30	0.25	30	0.31	30	0.30
Sokoban	30	2.66	30	2.73	30	2.74
Solitaire	27	0.13	27	0.15	27	0.19
<b>Stable Marriage</b>	<b>30</b>	<b>27.98</b>	<b>30</b>	<b>2.79</b>	<b>30</b>	<b>2.53</b>
<b>Strategic Companies</b>	<b>30</b>	<b>0.30</b>	<b>0</b>	<b>US</b>	<b>30</b>	<b>0.28</b>
Valves Location	30	4.02	30	4.07	30	4.02
Visit-all	30	0.14	30	0.14	30	0.14
<b>Weighted-Sequence Problem</b>	<b>30</b>	<b>2.84</b>	<b>30</b>	<b>9.80</b>	<b>30</b>	<b>3.00</b>
<b>Total Grounded Instances</b>	<b>722/756</b>		<b>658/756</b>		<b>733/756</b>	
<b>Average Time</b>	<b>27.08</b>		<b>34.19</b>		<b>24.57</b>	

Table 11.8: Decomposition Rewriting – 4th Comp. Benchmarks

Eventually, we consider here an additional set of benchmarks in which the decomposition rewriting plays a key role. Such benchmarks have been used in [17] in order to test the efficiency of ASP-solvers paired with  $lpopt$  for QBF solving. To this end, some publicly available QBF instances have been converted to ASP and a novel conversion has been proposed in which the obtained ASP programs feature complex structure and very long rules on which  $lpopt$  proved its effectiveness. We selected the 2-QBF instances adopted in the above mentioned work and converted according to their method with the aim of testing our decomposition rewriting on these challenging programs. Five different configurations have been compared: (i)  $\mathcal{I}\text{-DLV}\text{-No}\text{-D}$ , (ii)  $lpopt \mid \mathcal{I}\text{-DLV}\text{-No}\text{-D}$ , (iii)  $\mathcal{I}\text{-DLV}\text{-SD}$ , along with two additional versions (iv)  $\mathcal{I}\text{-DLV}\text{-SD}\text{-No}\text{-F}\text{-HTD}\text{-new}$  and (v)  $\mathcal{I}\text{-DLV}\text{-SD}\text{-No}\text{-F}\text{-HTD}\text{-old}$ , in which the decomposition mechanism is enabled but differently from  $\mathcal{I}\text{-DLV}\text{-SD}$  the fitness mechanism has been disabled, therefore one possible decomposition is computed and only if it is estimated as convenient (as described in Chapter 8) it is applied. Moreover, these two versions are based on two different releases of the library HTD:  $\mathcal{I}\text{-DLV}\text{-SD}\text{-No}\text{-F}\text{-HTD}\text{-new}$  uses HTD 1.1.0 (the same release has been adopted in  $\mathcal{I}\text{-DLV}\text{-SD}$ ), while  $\mathcal{I}\text{-DLV}\text{-SD}\text{-No}\text{-F}\text{-HTD}\text{-old}$  uses HTD 1.0.0.

*No-F-HTD-old* embeds an older release (HTD 1.1 rc1-bugfix), the same used by the version of *lpopt* tested.

<i>I-DLV-No-D</i>	<i>lpopt</i>   <i>I-DLV-No-D</i>	<i>I-DLV-SD</i>	<i>I-DLV-SD-No-F-HTD-new</i>	<i>I-DLV-SD-No-F-HTD-old</i>
8	82	22	80	84

Table 11.9: Decomposition Rewriting – 2QBF Benchmarks – Total Grounded Instances

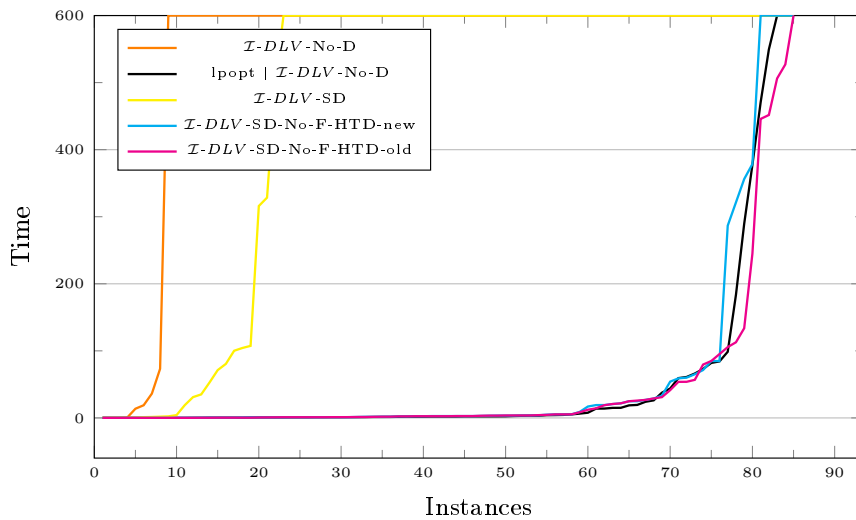


Figure 11.2: Decomposition Rewriting – 2QBF Benchmarks

The benchmarks consist of 200 ASP programs, each one corresponding to a different 2-QBF instance. The results are depicted in Figure 11.2: the number of grounded instances is on the x-axis while running times (in seconds) are on the y-axis. The total number of successfully grounded instances per each tested configuration is reported in Table 11.9. First of all, we observe that *I-DLV-SD* performs better than *I-DLV-No-D* but significantly worse than the other tested versions. This behaviour is due to the fitness mechanism: by default, *I-DLV* generates several possible decompositions for each input rule (cf. Section 10.3.1): in situations in which rules have extremely long bodies performing such iterations resulted to be expensive. Indeed, *I-DLV-SD-No-F-HTD-new* which is a equivalent to *I-DLV-SD* but without the fitness mechanism performed significantly better, but not as good as *lpopt* | *I-DLV-No-D*. The best version resulted to be *I-DLV-SD-No-F-HTD-old*, which is equivalent to *I-DLV-SD-No-F-HTD-new* a part from the version of HTD adopted. Therefore, it is also evicted that the version of the library HTD employed may have a non-negligible impact. In particular, the version of HTD employed in both *I-DLV-SD-No-F-HTD-old* and *lpopt* resulted to be more effective on these benchmarks, in general.

## 11.4 Pushing Down Selections

The technique presented in Section 9.1 aims at anticipating the selection operations as soon as the involved variables are bound, leveraging on these operations in order to suddenly recover failing matches. To assess its effectiveness, we compared two versions of  $\mathcal{I}$ -DLV: (i)  $\mathcal{I}$ -DLV-No-PS, a version in which such optimization is disabled, (ii)  $\mathcal{I}$ -DLV-PS, a version that performs it.

The results are shown in Tables 11.10 and 11.11. This optimization is effective when selection operations are actually present in rule bodies: in the Sixth Competition, its benefit are mainly evident in the problems *Permutation Pattern Matching* and *Maximal Clique*. On the less optimized encodings of the Fourth Competition improvements in many problems; notably on *Incremental Scheduling* this technique allows to ground a larger number of instances. No particular drawbacks or benefits emerge in problems in which the technique is not effective. Intuitively, this happens whenever no selection operations are present or the anticipation of their evaluation implies a negligible reduction of the search space.

Problem	$\mathcal{I}$ -DLV-No-PS		$\mathcal{I}$ -DLV-PS	
	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	20	0.11	20	0.11
Combined Configuration	20	13.62	20	13.53
Complex Optimization	20	65.78	20	66.83
Connected Still Life	20	0.10	20	0.10
Consistent Query Answering	20	78.69	20	76.31
Crossing Minimization	20	0.10	20	0.10
Graceful Graphs	20	0.34	20	0.31
Graph Coloring	20	0.10	20	0.10
Incremental Scheduling	20	16.65	20	17.05
Knight Tour With Holes	20	2.32	20	2.36
Labyrinth	20	2.04	20	2.01
<b>Maximal Clique</b>	20	<b>5.34</b>	20	<b>4.39</b>
MaxSAT	20	4.31	20	3.96
Minimal Diagnosis	20	5.12	20	5.19
Nomystery	20	3.97	20	4.16
Partner Units	20	0.45	20	0.43
<b>Permutation Pattern Matching</b>	20	<b>163.31</b>	20	<b>135.04</b>
Qualitative Spatial Reasoning	20	5.46	20	5.49
Reachability	20	143.43	20	142.64
Ricochet Robots	20	0.36	20	0.37
Sokoban	20	1.20	20	1.23
Stable Marriage	20	116.42	20	123.55
Steiner Tree	20	29.06	20	29.83
Strategic Companies	20	0.32	20	0.25
System Synthesis	20	1.10	20	1.12
Valves Location Problem	20	2.54	20	2.58
Video Streaming	20	0.10	20	0.10
Visit-all	20	1.18	20	1.22
<b>Total Grounded Instances</b>	<b>560/560</b>		<b>560/560</b>	
<b>Average Time</b>	<b>24.2</b>		<b>22.98</b>	

Table 11.10: Pushing Down Selections – 6th Comp. Benchmarks

Problem	$\mathcal{I}$ -DLV-No-PS		$\mathcal{I}$ -DLV-PS	
	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	30	0.11	30	0.11
Bottle Filling Problem	30	4.19	30	4.15
Chemical Classification	26	120.83	26	120.60
Complex Optimization	29	40.92	29	40.16
Connected Still Life	10	0.12	10	0.12
Crossing Minimization	30	0.10	30	0.10
<b>Graceful Graphs</b>	<b>30</b>	<b>0.42</b>	<b>30</b>	<b>0.38</b>
Graph Colouring	30	0.10	30	0.10
Hanoi Tower	30	0.24	30	0.24
<b>Incremental Scheduling</b>	<b>7</b>	<b>475.82</b>	<b>12</b>	<b>296.01</b>
Knight Tour with Holes	20	179.78	20	173.50
Labyrinth	30	1.52	30	1.51
Maximal Clique	30	0.34	30	0.33
Minimal Diagnosis	30	2.53	30	2.53
<b>Nomystery</b>	<b>30</b>	<b>48.29</b>	<b>30</b>	<b>43.68</b>
<b>Permutation Pattern Matching</b>	<b>28</b>	<b>66.00</b>	<b>28</b>	<b>58.17</b>
Qualitative Spatial Reasoning	30	2.90	30	2.84
Reachability	30	126.66	30	125.65
Ricochet Robots	30	0.25	30	0.25
Sokoban	30	2.63	30	2.66
Solitaire	27	0.12	27	0.13
<b>Stable Marriage</b>	<b>30</b>	<b>32.60</b>	<b>30</b>	<b>27.98</b>
Strategic Companies	30	0.30	30	0.30
Valves Location	30	4.01	30	4.02
Visit-all	30	0.14	30	0.14
<b>Weighted-Sequence Problem</b>	<b>30</b>	<b>3.13</b>	<b>30</b>	<b>2.84</b>
<i>Total Grounded Instances</i>	<i>717/756</i>		<i>722/756</i>	
<i>Average Time</i>	<i>28.92</i>		<i>27.08</i>	

Table 11.11: Pushing Down Selections – 4th Comp. Benchmarks

## 11.5 Managing Isolated Variables

In this section we analyze the effects of the optimizations geared towards the management of isolated variables. In particular, we compared three versions of  $\mathcal{I}$ -DLV: (i)  $\mathcal{I}$ -DLV-No-IV, a version in which isolated variables are not managed at all, (ii)  $\mathcal{I}$ -DLV-FIV, a version that applies just the filtering mechanism described in Section 9.2.1; (iii)  $\mathcal{I}$ -DLV-RIV, a version that performs the rewriting process illustrated in Section 9.2.2.

Tables 11.12 and 11.13 depict the obtained results. In the Sixth Competition, an overhead is observable in the *Consistent Query Answering* problem when the rewriting process is applied, the worsening in performance is about 12%; while, in other problems the version (iii) performs better than the other ones. Concerning the Fourth Competition, we observe a worsening for  $\mathcal{I}$ -DLV-FIV and  $\mathcal{I}$ -DLV-RIV in the problems *Bottle Filling Problem* and *Chemical Classification*. Interestingly, the encoding of *Chemical Classification* features rules with lots of isolated variables: there are rules with around 80 isolated variables, and it emerged that the additional operations involving either filtering or projecting out isolated variables yield an overhead. Moreover, apart the aforementioned situations it can be evicted that  $\mathcal{I}$ -DLV-RIV performed the best, in general. In addition, since the rewriting is more general than the filtering mechanism and can be applied in a larger number of scenarios, by default we choose to enable the projection rewriting. Nevertheless, whenever an overhead is observed, both techniques can be disabled.



Problem	$\mathcal{I}$ -DLV-No-IV		$\mathcal{I}$ -DLV-FIV		$\mathcal{I}$ -DLV-RIV	
	#grounded	time	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	20	0.11	20	0.11	20	0.11
Combined Configuration	20	13.91	20	13.74	20	13.53
Complex Optimization	20	74.08	20	73.64	20	69.83
Connected Still Life	20	0.10	20	0.10	20	0.10
<b>Consistent Query Answering</b>	20	<b>67.31</b>	20	<b>67.61</b>	20	<b>76.31</b>
Crossing Minimization	20	0.10	20	0.10	20	0.10
Graceful Graphs	20	0.29	20	0.29	20	0.31
Graph Coloring	20	0.10	20	0.10	20	0.10
Incremental Scheduling	20	17.24	20	17.26	20	17.05
Knight Tour With Holes	20	2.60	20	2.61	20	2.36
Labyrinth	20	1.99	20	2.00	20	2.01
<b>Maximal Clique</b>	20	<b>5.22</b>	20	<b>5.24</b>	20	<b>4.39</b>
MaxSAT	20	4.25	20	4.27	20	3.96
Minimal Diagnosis	20	5.13	20	5.14	20	5.19
<b>Nomystery</b>	20	<b>4.85</b>	20	<b>4.91</b>	20	<b>4.16</b>
<b>Partner Units</b>	20	<b>0.42</b>	20	<b>0.41</b>	20	<b>0.43</b>
Permutation Pattern Matching	20	131.11	20	130.95	20	135.04
Qualitative Spatial Reasoning	20	5.32	20	5.43	20	5.49
Reachability	20	156.12	20	156.01	20	142.64
Ricochet Robots	20	0.35	20	0.35	20	0.37
Sokoban	20	1.19	20	1.20	20	1.23
Stable Marriage	20	119.94	20	124.13	20	123.55
Steiner Tree	20	28.88	20	29.03	20	29.83
<b>Strategic Companies</b>	20	<b>0.32</b>	20	<b>0.32</b>	20	<b>0.25</b>
System Synthesis	20	1.11	20	1.10	20	1.12
Valves Location Problem	20	2.58	20	2.55	20	2.58
Video Streaming	20	0.10	20	0.10	20	0.10
Visit-all	20	1.20	20	1.19	20	1.22
<b>Total Grounded Instances</b>	<b>560/560</b>		<b>560/560</b>		<b>560/560</b>	
<b>Average Time</b>	<b>23.07</b>		<b>23.21</b>		<b>22.98</b>	

Table 11.12: Managing Isolated Variables – 6th Comp. Benchmarks

Problem	$\mathcal{I}$ -DLV-No-IV		$\mathcal{I}$ -DLV-FIV		$\mathcal{I}$ -DLV-RIV	
	#grounded	time	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	30	0.11	30	0.11	30	0.11
<b>Bottle Filling Problem</b>	<b>30</b>	<b>3.85</b>	<b>30</b>	<b>4.70</b>	<b>30</b>	<b>4.15</b>
<b>Chemical Classification</b>	<b>26</b>	<b>91.19</b>	<b>26</b>	<b>118.97</b>	<b>26</b>	<b>120.60</b>
Complex Optimization	29	42.95	29	42.58	29	40.16
Connected Still Life	10	0.12	10	0.12	10	0.12
Crossing Minimization	30	0.10	30	0.10	30	0.10
Graceful Graphs	30	0.37	30	0.37	30	0.38
Graph Colouring	30	0.10	30	0.10	30	0.10
Hanoi Tower	30	0.23	30	0.23	30	0.24
Incremental Scheduling	12	296.31	12	297.10	12	296.01
Knight Tour with Holes	20	171.69	20	172.85	20	173.50
Labyrinth	30	1.51	30	1.51	30	1.51
Maximal Clique	30	0.33	30	0.33	30	0.33
Minimal Diagnosis	30	2.52	30	2.53	30	2.53
<b>Nomystery</b>	<b>29</b>	<b>46.37</b>	<b>29</b>	<b>46.37</b>	<b>30</b>	<b>43.68</b>
Permutation Pattern Matching	28	59.09	28	58.37	28	58.17
Qualitative Spatial Reasoning	30	2.84	30	2.84	30	2.84
Reachability	30	125.71	30	125.38	30	125.65
Ricochet Robots	30	0.24	30	0.24	30	0.25
Sokoban	30	2.63	30	2.67	30	2.66
Solitaire	27	0.12	27	0.12	27	0.13
<b>Stable Marriage</b>	<b>0</b>	<b>MO</b>	<b>30</b>	<b>28.25</b>	<b>30</b>	<b>27.98</b>
Strategic Companies	30	0.30	30	0.30	30	0.30
Valves Location	30	4.04	30	4.04	30	4.02
Visit-all	30	0.14	30	0.14	30	0.14
Weighted-Sequence Problem	30	2.80	30	2.83	30	2.84
<b>Total Grounded Instances</b>	<b>691/756</b>		<b>721/756</b>		<b>722/756</b>	
<b>Average Time</b>	<b>26.11</b>		<b>27.23</b>		<b>27.08</b>	

Table 11.13: Managing Isolated Variables – 4th Comp. Benchmarks

## 11.6 Determining the Admissibility of Substitutions

This section evaluates the impact of the techniques intended to avoid the propagation of non-admissible variable substitutions through literals during the rule instantiation process, such as the aligning substitutions mechanism and the look-ahead technique. Therefore, we compared three different versions of  $\mathcal{I}$ -DLV: (i)  $\mathcal{I}$ -DLV-No-S in which none of the two optimizations is enabled, (ii)  $\mathcal{I}$ -DLV-AS in which only the aligning substitutions mechanism is performed, (iii)  $\mathcal{I}$ -DLV-LA enabling just the look-ahead.

Problem	$\mathcal{I}$ -DLV-No-S		$\mathcal{I}$ -DLV-AS		$\mathcal{I}$ -DLV-LA	
	#grounded	time	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	20	0.11	20	0.11	20	0.11
Combined Configuration	20	13.53	20	13.26	20	13.54
<b>Complex Optimization</b>	20	<b>69.83</b>	20	<b>67.96</b>	20	<b>91.47</b>
Connected Still Life	20	0.10	20	0.10	20	0.10
<b>Consistent Query Answering</b>	20	<b>76.31</b>	20	<b>82.07</b>	20	<b>95.52</b>
Crossing Minimization	20	0.10	20	0.10	20	0.10
Graceful Graphs	20	0.31	20	0.29	20	0.29
Graph Coloring	20	0.10	20	0.10	20	0.10
Incremental Scheduling	20	17.05	20	17.10	20	17.06
Knight Tour With Holes	20	2.36	20	2.35	20	2.33
Labyrinth	20	2.01	20	2.18	20	2.03
Maximal Clique	20	4.39	20	5.29	20	5.31
MaxSAT	20	3.96	20	4.33	20	4.36
Minimal Diagnosis	20	5.19	20	5.30	20	5.18
Nomystery	20	4.16	20	4.19	20	4.16
Partner Units	20	0.43	20	0.41	20	0.41
Permutation Pattern Matching	20	135.04	20	133.83	20	136.68
Qualitative Spatial Reasoning	20	5.49	20	5.48	20	5.48
Reachability	20	142.64	20	159.20	20	157.78
Ricochet Robots	20	0.37	20	0.36	20	0.35
Sokoban	20	1.23	20	1.30	20	1.22
Stable Marriage	20	123.55	20	121.26	20	120.86
Steiner Tree	20	29.83	20	29.33	20	29.33
Strategic Companies	20	0.25	20	0.32	20	0.33
System Synthesis	20	1.12	20	1.11	20	1.10
Valves Location Problem	20	2.58	20	2.58	20	2.55
Video Streaming	20	0.10	20	0.10	20	0.10
Visit-all	20	1.22	20	1.19	20	1.19
<b>Total Grounded Instances</b>	<b>560/560</b>		<b>560/560</b>		<b>560/560</b>	
<b>Average Time</b>	<b>22.98</b>		<b>24.97</b>		<b>23.61</b>	

Table 11.14: Admissibility of Substitutions – 6th Comp. Benchmarks

The results have been collected in Tables 11.14 and 11.15. In particular, we can observe that the version  $\mathcal{I}$ -DLV-AS pays an overhead in the problems *Complex Optimization* and *Consistent Query Answering*. Moreover, even if  $\mathcal{I}$ -DLV-LA and  $\mathcal{I}$ -DLV-No-S performed similarly, in general, it emerges that both  $\mathcal{I}$ -DLV-AS and  $\mathcal{I}$ -DLV-LA experience a minor systematical overhead; as can be easily evicted from the average time computed over all grounded instances. The reason is mostly technical: to implement both techniques further checks have been introduced in the code handling the matching of literals. Intuitively, this piece of code is the one invoked the most during the whole instantiation, thus its efficiency is crucial, and in our case adding these checks implies a slowdown in performance. In particular, the technique about aligning substitutions evidences an additional slowdown due to the pre-computation of intersections. Thus, by

default, we keep these techniques disabled; via command-line options they can be enabled on request, according to the particular situation at hand and the distributions followed by input data.

Problem	<i>I-DLV-No-S</i>		<i>I-DLV-AS</i>		<i>I-DLV-LA</i>	
	#grounded	time	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	30	0.11	30	0.11	30	0.11
Bottle Filling Problem	30	4.15	30	4.33	30	4.48
Chemical Classification	26	120.60	26	123.80	26	122.59
<b>Complex Optimization</b>	29	<b>40.16</b>	29	<b>54.44</b>	29	<b>43.73</b>
Connected Still Life	10	0.12	10	0.12	10	0.12
Crossing Minimization	30	0.10	30	0.10	30	0.10
Graceful Graphs	30	0.38	30	0.38	30	0.37
Graph Colouring	30	0.10	30	0.10	30	0.10
Hanoi Tower	30	0.24	30	0.24	30	0.24
Incremental Scheduling	12	296.01	12	307.84	12	307.00
Knight Tour with Holes	20	173.50	20	175.39	20	177.96
Labyrinth	30	1.51	30	1.58	30	1.61
Maximal Clique	30	0.33	30	0.34	30	0.33
Minimal Diagnosis	30	2.53	30	2.55	30	2.61
Nomystery	30	43.68	30	43.68	30	44.20
Permutation Pattern Matching	28	58.17	28	59.17	28	58.62
Qualitative Spatial Reasoning	30	2.84	30	2.91	30	2.90
Reachability	30	125.65	30	126.65	30	126.83
Ricochet Robots	30	0.25	30	0.25	30	0.25
Sokoban	30	2.66	30	2.71	30	2.88
Solitaire	27	0.13	27	0.12	27	0.12
Stable Marriage	30	27.98	30	29.75	30	31.63
Strategic Companies	30	0.30	30	0.32	30	0.31
Valves Location	30	4.02	30	4.09	30	4.11
Visit-all	30	0.14	30	0.14	30	0.14
Weighted-Sequence Problem	30	2.84	30	2.79	30	2.86
<b>Total Grounded Instances</b>	<b>722/756</b>		<b>722/756</b>		<b>722/756</b>	
<b>Average Time</b>	<b>27.08</b>		<b>28.19</b>		<b>27.88</b>	

Table 11.15: Admissibility of Substitutions – 4th Comp. Benchmarks

## 11.7 Anticipating Strong Constraints Evaluation

The anticipation of evaluation of strong constraints, described in Section 9.4 permits to determine whether an input program is incoherent as soon as possible. To assess its impact on the performance of *I-DLV* we considered two versions the grounder: (i) *I-DLV-No-SC*, a version in which the optimization is not performed, (ii) *I-DLV-SC*, a version in which it is enabled.

The results are shown in Tables 11.16 and 11.17. Since in both suites there are no situations in which input programs are incoherent, no particular impact emerged; therefore, we selected three further problems described next.

**Food.** The problem consists in the generation of plans for repairing faulty workflows. That is, starting from a faulty workflow instance, the goal is to provide a completion of the workflow such that the output of the workflow is correct. Workflows may comprise many activities. Repair actions are compensation, (re)do and replacement of activities. The encoding and the single instance provided (related to a workflow containing 63 predicates, 56 components and 116 rules) have been used in [110].

**Hamiltonian Path.** The problem has been introduced in Section 3. The encoding used is the one already presented. The instances have been adapted

from the ones of the problem *MaxSAT* of the Sixth Competition, obtaining both coherent and incoherent instances.

**Sudoku.** Given a  $N \times N$  tableau, where  $N$  is a square number  $N = n^2$ , this problem consists in filling it with numbers between 1 and  $N$  so that each row, each column, and each of the  $N$   $n \times n$  inner block in which the tableau is divided, contains each of the integers from 1 to  $N$  exactly once. In this generalized setting, the Sudoku problem is known to be NP-complete. We considered instances of varying sizes, with  $N$  ranging from 16 to 64. Both instances and encoding have been retrieved from [35].

Grounding times and number of grounded instances are reported in Table 11.18. In these situations in which input instance may lead to incoherences, it is evident a significant improvement in performance. Moreover, the competition suites evidence no drawbacks on coherent problems. Intuitively, this is not surprising since we are just anticipating apart of job that should be done anyway. For these reasons, in *T-DLV* this optimization is performed by default.

Problem	<i>T-DLV</i> -No-SC		<i>T-DLV</i> -SC	
	#grounded	time	#grounded	time
<b>Sixth Competition</b>				
Abstract Dialectical Frameworks	20	0.11	20	0.11
Combined Configuration	20	13.53	20	13.43
Complex Optimization	20	66.83	20	68.95
Connected Still Life	20	0.10	20	0.10
Consistent Query Answering	20	76.31	20	77.26
Crossing Minimization	20	0.10	20	0.10
Graceful Graphs	20	0.31	20	0.30
Graph Coloring	20	0.10	20	0.10
Incremental Scheduling	20	17.05	20	16.77
Knight Tour With Holes	20	2.36	20	2.31
Labyrinth	20	2.01	20	2.01
Maximal Clique	20	4.39	20	5.25
MaxSAT	20	3.96	20	4.24
Minimal Diagnosis	20	5.19	20	5.25
Nomystery	20	4.16	20	4.08
Partner Units	20	0.43	20	0.41
Permutation Pattern Matching	20	135.04	20	134.64
Qualitative Spatial Reasoning	20	5.49	20	5.45
Reachability	20	142.64	20	143.98
Ricochet Robots	20	0.37	20	0.36
Sokoban	20	1.23	20	1.21
Stable Marriage	20	123.55	20	124.25
Steiner Tree	20	29.83	20	29.18
Strategic Companies	20	0.25	20	0.32
System Synthesis	20	1.12	20	1.11
Valves Location Problem	20	2.58	20	2.55
Video Streaming	20	0.10	20	0.10
Visit-all	20	1.22	20	1.23
<b>Total Grounded Instances</b>	<b>560/560</b>		<b>560/560</b>	
<b>Average Time</b>	<b>23.47</b>		<b>23.23</b>	

Table 11.16: Anticipation of Strong Constraints Evaluation – 6th Comp. Benchmarks

Problem	<i>I-DLV-No-SC</i>		<i>I-DLV-SC</i>	
	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	30	0.11	30	0.11
Bottle Filling Problem	30	4.18	30	4.15
Chemical Classification	26	124.32	26	120.60
Complex Optimization	29	41.62	29	40.16
Connected Still Life	10	0.12	10	0.12
Crossing Minimization	30	0.10	30	0.10
Graceful Graphs	30	0.37	30	0.38
Graph Colouring	30	0.10	30	0.10
Hanoi Tower	30	0.24	30	0.24
Incremental Scheduling	12	295.14	12	296.01
Knight Tour with Holes	20	176.96	20	173.50
Labyrinth	30	1.56	30	1.51
Maximal Clique	30	0.33	30	0.33
Minimal Diagnosis	30	2.61	30	2.53
Nomystery	30	43.30	30	43.68
Permutation Pattern Matching	28	58.64	28	58.17
Qualitative Spatial Reasoning	30	2.87	30	2.84
Reachability	30	125.85	30	125.65
Ricochet Robots	30	0.25	30	0.25
Sokoban	30	2.68	30	2.66
Solitaire	27	0.12	27	0.13
Stable Marriage	30	28.49	30	27.98
Strategic Companies	30	0.31	30	0.30
Valves Location	30	4.08	30	4.02
Visit-all	30	0.14	30	0.14
Weighted-Sequence Problem	30	2.89	30	2.84
<b>Total Grounded Instances</b>	<b>722/756</b>		<b>722/756</b>	
<b>Average Time</b>	<b>27.15</b>		<b>27.08</b>	

Table 11.17: Anticipation of Strong Constraints Evaluation – 4th Comp. Benchmarks

Problem	<i>I-DLV-No-SC</i>		<i>I-DLV-SC</i>	
	#grounded	time	#grounded	time
Food	1	29.35	1	0.10
Hamiltonian Path	17	266.61	40	10.52
Sudoku	70	31.44	70	15.69
<b>Total Grounded Instances</b>	<b>88/111</b>		<b>111/111</b>	
<b>Average Time</b>	<b>76.85</b>		<b>13.69</b>	

Table 11.18: Anticipation of Strong Constraints Evaluation – Further Benchmarks

## 11.8 Syntactic Rewriting Techniques

In this section we consider the impact of syntactically rewriting the input program by applying the techniques illustrated in Section 9.5. In particular, while aggregate literals in *I-DLV* are automatically rewritten and their instantiation has been designed under the assumption that the rewriting described in Section 9.5.3 has been preliminarily applied, for the other syntactic constructs it is possible to change the default adopted strategy.

### Choice Rules

Concerning choice rules, the three distinct rewriting techniques described have been implemented in *I-DLV* (cf. Sections 10.3.1 and 9.5.4). We executed a different version of *I-DLV* per each distinct rewriting: *I-DLV-C1* adopts the strategy that replaces choice rules with disjunctive rules, *I-DLV-C2* makes use of the second rewriting process illustrated in Section 9.5.4, while *I-DLV-C3* employs the third rewriting discussed in the same section. The results are reported in Tables 11.19 and 11.20. The results about the Sixth Competition evidence that native approaches are preferable: *I-DLV-C2* and *I-DLV-C3* perform the best. In particular, a significant advantage can be observed on the problem *Steiner Tree*. However, apart from this problem, the performance of the version *I-DLV-C1* is quite similar to the ones of the other versions. On the other hand, in the Fourth Competition no particular advantages or disadvantages are observable: all the three versions performed similarly.

### Functional Terms

Regarding functional terms, we experimented with three configurations: (i) *I-DLV-No-RFT*, in which the rewriting of functional terms outlined in Section 9.5.2 is disabled, (ii) *I-DLV-No-RFT-Ord3*, in which the rewriting is also disabled but it is employed the body ordering strategy geared towards functional terms, (iii) *I-DLV-RFT* in which the rewriting is enabled.

The results are illustrated in Tables 11.21 and 11.22, and demonstrate that among all possibilities, enabling the rewriting is the best choice, especially on the problems *Abstract Dialectical Frameworks* and *Complex Optimization*. Indeed, in the considered benchmarks it emerged that pushing functional terms down in the body is not a sufficient strategy for handling functional terms efficiently. In *I-DLV* the rewriting is enabled by default; however, since as emerged for the rewriting of isolated variables the introduction of further auxiliary rules may cause an overhead, it can be disabled.

### Arithmetic Terms

Eventually, let us consider the effects of the rewriting of arithmetic terms depicted in Section 9.5.1. We compared two versions of *I-DLV*, *I-DLV-No-RAT* and *I-DLV-RAT*, in which *I-DLV* does not apply or enable such rewriting. Tables 11.23 and Tables 11.24 report the results. Interestingly, some problems can be grounded more efficiently when the rewriting is enabled; for instance, in both suites significant benefits are observable in the problems *Complex Optimization*, and *Nomystery*. On the other hand, some worsening emerges, for instance, in *Ricochet Robots* and *Sokoban*. In general, the effects of such rewriting are not predictable, but may bring benefits depending on the particular encoding at hand; therefore, by default it is disabled.

Problem	<i>T-DLV-C1</i>		<i>T-DLV-C2</i>		<i>T-DLV-C3</i>	
	#grounded	time	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	20	0.11	20	0.11	20	0.11
Combined Configuration	20	13.95	20	13.77	20	13.60
Complex Optimization	20	67.10	20	67.48	20	67.99
Connected Still Life	20	0.10	20	0.10	20	0.10
Consistent Query Answering	20	76.72	20	76.65	20	76.64
Crossing Minimization	20	0.10	20	0.10	20	0.10
Graceful Graphs	20	0.29	20	0.29	20	0.29
Graph Coloring	20	0.10	20	0.10	20	0.10
Incremental Scheduling	20	18.17	20	17.12	20	17.26
Knight Tour With Holes	20	2.30	20	2.31	20	2.31
Labyrinth	20	1.99	20	1.99	20	1.98
Maximal Clique	20	5.23	20	5.22	20	5.21
MaxSAT	20	4.30	20	4.29	20	4.29
Minimal Diagnosis	20	5.15	20	5.11	20	5.12
Nomystery	20	3.93	20	3.93	20	4.02
Partner Units	20	0.73	20	0.41	20	0.41
Permutation Pattern Matching	20	134.36	20	133.78	20	132.27
Qualitative Spatial Reasoning	20	5.43	20	5.44	20	5.39
Reachability	20	156.07	20	156.29	20	156.35
Ricochet Robots	20	0.36	20	0.35	20	0.35
Sokoban	20	1.26	20	1.21	20	1.20
Stable Marriage	20	120.80	20	120.94	20	119.41
Steiner Tree	20	492.17	20	29.09	20	29.34
Strategic Companies	20	0.32	20	0.32	20	0.32
System Synthesis	20	1.19	20	1.11	20	1.10
Valves Location Problem	20	2.58	20	2.58	20	2.55
Video Streaming	20	0.10	20	0.10	20	0.10
Visit-all	20	1.18	20	1.19	20	1.18
<b>Total Grounded Instances</b>	<b>560/560</b>		<b>560/560</b>		<b>560/560</b>	
<b>Average Time</b>	<b>39.86</b>		<b>23.26</b>		<b>23.18</b>	

Table 11.19: Choice Rewriting – 6th Comp. Benchmarks

Problem	<i>T-DLV-C1</i>		<i>T-DLV-C2</i>		<i>T-DLV-C3</i>	
	#grounded	time	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	30	0.11	30	0.11	30	0.11
Bottle Filling Problem	30	4.14	30	4.14	30	4.14
Chemical Classification	26	120.64	26	120.58	26	120.53
Complex Optimization	29	40.61	29	40.53	29	40.52
Connected Still Life	10	0.12	10	0.12	10	0.12
Crossing Minimization	30	0.10	30	0.10	30	0.10
Graceful Graphs	30	0.37	30	0.38	30	0.38
Graph Colouring	30	0.10	30	0.10	30	0.10
Hanoi Tower	30	0.24	30	0.24	30	0.24
Incremental Scheduling	12	295.67	12	297.77	12	300.08
Knight Tour with Holes	20	173.10	20	173.21	20	174.40
Labyrinth	30	1.50	30	1.51	30	1.51
Maximal Clique	30	0.33	30	0.33	30	0.33
Minimal Diagnosis	30	2.52	30	2.52	30	2.53
Nomystery	30	43.63	30	43.66	30	43.68
Permutation Pattern Matching	28	57.93	28	58.06	28	58.05
Qualitative Spatial Reasoning	30	2.83	30	2.88	30	2.84
Reachability	30	125.70	30	125.63	30	125.45
Ricochet Robots	30	0.25	30	0.25	30	0.25
Sokoban	30	2.66	30	2.74	30	2.67
Solitaire	27	0.12	27	0.13	27	0.12
Stable Marriage	30	27.97	30	27.85	30	27.87
Strategic Companies	30	0.30	30	0.31	30	0.31
Valves Location	30	4.06	30	4.13	30	4.02
Visit-all	30	0.14	30	0.14	30	0.14
Weighted-Sequence Problem	30	2.85	30	2.88	30	2.86
<b>Total Grounded Instances</b>	<b>722/756</b>		<b>722/756</b>		<b>722/756</b>	
<b>Average Time</b>	<b>27.07</b>		<b>27.11</b>		<b>27.17</b>	

Table 11.20: Choice Rewriting – 4th Comp. Benchmarks

Problem	I-DLV-No-RFT		I-DLV-No-RFT-Ord3		I-DLV-RFT	
	#grounded	time	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	20	4.45	20	4.50	20	0.11
Combined Configuration	20	14.22	20	28.27	20	13.53
Complex Optimization	20	483.67	20	483.75	20	69.83
Connected Still Life	20	0.10	20	0.10	20	0.10
Consistent Query Answering	20	76.22	20	76.54	20	76.31
Crossing Minimization	20	0.10	20	0.10	20	0.10
Graceful Graphs	20	0.29	20	0.29	20	0.31
Graph Coloring	20	0.10	20	0.10	20	0.10
Incremental Scheduling	20	17.10	20	17.10	20	17.05
Knight Tour With Holes	20	2.32	20	2.30	20	2.36
Labyrinth	20	2.01	20	1.99	20	2.01
Maximal Clique	20	5.21	20	5.22	20	4.39
MaxSAT	20	4.27	20	4.33	20	3.96
Minimal Diagnosis	20	5.13	20	5.15	20	5.19
Nomystery	20	4.00	20	4.03	20	4.16
Partner Units	20	0.41	20	0.41	20	0.43
Permutation Pattern Matching	20	134.07	20	133.41	20	135.04
Qualitative Spatial Reasoning	20	5.45	20	5.43	20	5.49
Reachability	20	155.92	20	156.60	20	142.64
Ricochet Robots	20	0.35	20	0.35	20	0.37
Sokoban	20	1.21	20	1.20	20	1.23
Stable Marriage	20	120.34	20	120.66	20	123.55
Steiner Tree	20	29.04	20	29.10	20	29.83
Strategic Companies	20	0.32	20	0.32	20	0.25
System Synthesis	20	1.10	20	1.12	20	1.12
Valves Location Problem	20	2.44	20	2.46	20	2.58
Video Streaming	20	0.10	20	0.10	20	0.10
Visit-all	20	1.18	20	1.21	20	1.22
<b>Total Grounded Instances</b>	<b>560/560</b>		<b>560/560</b>		<b>560/560</b>	
<b>Average Time</b>	<b>38.25</b>		<b>38.8</b>		<b>22.98</b>	

Table 11.21: Functional Terms Rewriting – 6th Comp. Benchmarks

Problem	I-DLV-No-RFT		I-DLV-No-RFT-Ord3		I-DLV-RFT	
	#grounded	time	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	30	5.13	30	5.07	30	0.11
Bottle Filling Problem	30	4.16	30	4.17	30	4.15
Chemical Classification	26	120.65	26	121.80	26	120.60
Complex Optimization	14	17.49	14	17.38	29	40.16
Connected Still Life	10	0.12	10	0.12	10	0.12
Crossing Minimization	30	0.11	30	0.10	30	0.10
Graceful Graphs	30	0.38	30	0.37	30	0.38
Graph Colouring	30	0.10	30	0.10	30	0.10
Hanoi Tower	30	0.24	30	0.24	30	0.24
Incremental Scheduling	12	302.04	12	296.12	12	296.01
Knight Tour with Holes	20	177.61	20	173.26	20	173.50
Labyrinth	30	1.53	30	1.52	30	1.51
Maximal Clique	30	0.33	30	0.33	30	0.33
Minimal Diagnosis	30	2.53	30	2.55	30	2.53
Nomystery	30	43.85	30	43.46	30	43.68
Permutation Pattern Matching	28	60.04	28	58.38	28	58.17
Qualitative Spatial Reasoning	30	2.84	30	2.85	30	2.84
Reachability	30	125.11	30	125.65	30	125.65
Ricochet Robots	30	0.25	30	0.25	30	0.25
Sokoban	30	2.68	30	2.66	30	2.66
Solitaire	27	0.12	27	0.12	27	0.13
Stable Marriage	30	28.40	30	27.55	30	27.98
Strategic Companies	30	0.30	30	0.31	30	0.30
Valves Location	30	3.90	30	3.89	30	4.02
Visit-all	30	0.14	30	0.14	30	0.14
Weighted-Sequence Problem	30	2.85	30	2.83	30	2.84
<b>Total Grounded Instances</b>	<b>707/756</b>		<b>707/756</b>		<b>722/756</b>	
<b>Average Time</b>	<b>26.86</b>		<b>26.58</b>		<b>27.08</b>	

Table 11.22: Functional Terms Rewriting – 4th Comp. Benchmarks



Problem	<i>T-DLV-No-RAT</i>		<i>T-DLV-RAT</i>	
	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	20	0.11	20	0.11
Combined Configuration	20	13.53	20	13.89
<b>Complex Optimization</b>	20	<b>69.83</b>	20	<b>49.12</b>
Connected Still Life	20	0.10	20	0.10
Consistent Query Answering	20	76.31	20	76.28
Crossing Minimization	20	0.10	20	0.10
Graceful Graphs	20	0.31	20	0.30
Graph Coloring	20	0.10	20	0.10
Incremental Scheduling	20	17.05	20	16.97
Knight Tour With Holes	20	2.36	20	2.31
Labyrinth	20	2.01	20	1.99
<b>Maximal Clique</b>	20	<b>4.39</b>	20	<b>5.26</b>
MaxSAT	20	3.96	20	4.29
Minimal Diagnosis	20	5.19	20	5.12
<b>Nomystery</b>	20	<b>4.16</b>	20	<b>2.30</b>
Partner Units	20	0.43	20	0.41
Permutation Pattern Matching	20	135.04	20	131.97
Qualitative Spatial Reasoning	20	5.49	20	5.38
Reachability	20	142.64	20	156.93
<b>Ricochet Robots</b>	20	<b>0.37</b>	20	<b>0.55</b>
<b>Sokoban</b>	20	<b>1.23</b>	20	<b>1.40</b>
Stable Marriage	20	123.55	20	120.19
<b>Steiner Tree</b>	20	<b>29.83</b>	20	<b>0.10</b>
<b>Strategic Companies</b>	20	<b>0.25</b>	20	<b>0.32</b>
<b>System Synthesis</b>	20	<b>1.12</b>	20	<b>0.10</b>
Valves Location Problem	20	2.58	20	2.58
Video Streaming	20	0.10	20	0.10
Visit-all	20	1.22	20	1.22
<i>Total Grounded Instances</i>	<i>560/560</i>		<i>560/560</i>	
<i>Average Time</i>	<i>22.98</i>		<i>21.41</i>	

Table 11.23: Arithmetic Terms Rewriting – 6th Comp. Benchmarks

Problem	<i>T-DLV-No-RAT</i>		<i>T-DLV-RAT</i>	
	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	30	0.11	30	0.11
Bottle Filling Problem	30	4.15	30	4.21
Chemical Classification	26	120.60	26	120.63
<b>Complex Optimization</b>	29	<b>40.16</b>	29	<b>23.32</b>
Connected Still Life	10	0.12	10	0.12
Crossing Minimization	30	0.10	30	0.10
Graceful Graphs	30	0.38	30	0.38
Graph Colouring	30	0.10	30	0.10
Hanoi Tower	30	0.24	30	0.24
Incremental Scheduling	12	296.01	12	303.25
Knight Tour with Holes	20	173.50	20	176.03
Labyrinth	30	1.51	30	1.52
Maximal Clique	30	0.33	30	0.33
Minimal Diagnosis	30	2.53	30	2.53
<b>Nomystery</b>	30	<b>43.68</b>	30	<b>39.50</b>
Permutation Pattern Matching	28	58.17	28	58.80
Qualitative Spatial Reasoning	30	2.84	30	2.83
Reachability	30	125.65	30	125.12
<b>Ricochet Robots</b>	30	<b>0.25</b>	30	<b>0.44</b>
<b>Sokoban</b>	30	<b>2.66</b>	30	<b>3.10</b>
Solitaire	27	0.13	27	0.11
Stable Marriage	30	27.98	30	28.04
Strategic Companies	30	0.30	30	0.30
Valves Location	30	4.02	30	4.18
Visit-all	30	0.14	30	0.14
<b>Weighted-Sequence Problem</b>	30	<b>2.84</b>	30	<b>2.54</b>
<i>Total Grounded Instances</i>	<i>722/756</i>		<i>722/756</i>	
<i>Average Time</i>	<i>27.08</i>		<i>26.45</i>	

Table 11.24: Arithmetic Terms Rewriting – 4th Comp. Benchmarks

## Chapter 12

# Experimental Evaluation of *I-DLV*

In this chapter we present a more thorough analysis of *I-DLV* performance as both ASP grounder and deductive database system. In Section 12.1 we firstly compare *I-DLV* with current mainstream grounders; next, we assess its deductive database capabilities. In Section 12.2 we report an experimental study on *I-DLV* customization possibilities. Eventually, Section 12.3 examines the impact of *I-DLV* on state-of-the-art solvers evaluating the “quality” of its produced instantiation.

Experiments have been performed the same hardware of Chapter 11, and again for memory and time limits, we allotted 15 GiB and 600 seconds for each system, per each single run.

### 12.1 Comparison with the State-of-the-Art

In this section, we report the results of an experimental activity carried out to assess *I-DLV* performance as both ASP grounder and deductive database system. In order to obtain trustworthy results, we considered tests that have already been largely used and are publicly available. In particular, we relied on:

- the whole Sixth ASP Competition suite [67];
- OpenRuleBench [95], an open (freely available) set of resources comprising a suite of benchmarks for analysing performance and scalability of different rule engines;
- a suite of benchmarks employed in literature to test Consistent Query Answering (CQA) systems on large inconsistent database [84, 99].

#### 12.1.1 ASP Grounding Benchmarks

For this setting we tested *I-DLV* against the two mainstream grounders *gringo* and the (old) intelligent grounder of *DLV*, and in particular the latest available versions at the time of writing: 5.2.1 and 2012-12-17, respectively.

The results about the Sixth ASP Competition suite are reported in Table 12.1: first column shows the name of the problem, while the next three report the average times. Some benchmarks are reported in bold indicating cases where there is a noticeable difference of at least 10% among the systems tested; in addition, bad results are highlighted in red, while good performance in green. In the rest of the chapter, we will adopt the same style for the other experimental results; moreover, we will indicate with “MO” memory outs, and with “US” unsupported syntax.

Problem	<i>gringo</i>		<i>DLV</i>		$\mathcal{I}$ -DLV	
	#grounded	time	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	20	<b>2.22</b>	0	US	20	<b>0.11</b>
Combined Configuration	20	14.76	0	US	20	13.37
<b>Complex Optimization</b>	20	<b>8.56</b>	0	US	20	<b>70.09</b>
Connected Still Life	20	0.10	0	US	20	0.10
<b>Consistent Query Answering</b>	0	US	20	<b>196.99</b>	20	<b>77.50</b>
Crossing Minimization	20	0.10	0	US	20	0.10
<b>Graceful Graphs</b>	20	<b>0.21</b>	0	US	20	<b>0.31</b>
Graph Coloring	20	0.10	0	US	20	0.10
<b>Incremental Scheduling</b>	20	<b>29.21</b>	0	US	20	<b>17.20</b>
<b>Knight Tour With Holes</b>	20	<b>12.85</b>	0	US	20	<b>2.34</b>
<b>Labyrinth</b>	20	<b>0.54</b>	20	<b>15.73</b>	20	<b>2.03</b>
<b>Maximal Clique</b>	20	<b>14.23</b>	0	US	20	<b>4.37</b>
<b>MaxSAT</b>	20	<b>6.79</b>	0	US	20	<b>3.99</b>
<b>Minimal Diagnosis</b>	20	<b>3.00</b>	20	<b>57.92</b>	20	<b>4.74</b>
<b>Nomystery</b>	20	<b>2.37</b>	0	US	20	<b>4.15</b>
Partner Units	20	0.41	0	US	20	0.43
<b>Permutation Pattern Matching</b>	20	<b>124.11</b>	0	US	20	<b>4.31</b>
<b>Qualitative Spatial Reasoning</b>	20	5.93	20	<b>31.78</b>	20	5.48
<b>Reachability</b>	0	US	0	TO	20	<b>141.19</b>
<b>Ricochet Robots</b>	20	<b>0.19</b>	0	US	20	<b>0.38</b>
Sokoban	20	1.17	0	US	20	1.25
Stable Marriage	20	114.75	0	US	20	124.09
Steiner Tree	20	31.44	0	US	20	29.73
<b>Strategic Companies</b>	0	US	20	<b>236.57</b>	20	<b>0.26</b>
<b>System Synthesis</b>	20	<b>1.34</b>	0	US	20	<b>1.12</b>
<b>Valves Location Problem</b>	20	<b>13.02</b>	0	US	20	<b>2.59</b>
Video Streaming	20	0.10	0	US	20	0.10
<b>Visit-all</b>	20	<b>1.06</b>	0	US	20	<b>0.44</b>
<b>Total Grounded Instances</b>	<b>500/560</b>		<b>100/560</b>		<b>560/560</b>	
<b>Average Time</b>	<b>15.25</b>		<b>101.02</b>		<b>17.86</b>	

Table 12.1: Sixth ASP Competition Benchmarks – number of grounded instances and grounding times in seconds

When launched, all systems were able to ground all 20 instances in the allotted time. It is evident, while comparing  $\mathcal{I}$ -DLV against *DLV*, that the new grounder systematically outperforms its predecessor, enjoying performance gains up to 90%. This is immediately evident also in the cactus plot in Figure 12.1 that compares  $\mathcal{I}$ -DLV and its predecessor displaying the grounded instances on problems following *DLV* syntax, whose number increases along the  $x$ -axis, while the runtime is reported on the  $y$ -axis. Also the comparison with *gringo* is encouraging, despite  $\mathcal{I}$ -DLV has been just recently released it proves to be competitive. More in detail, excluding the 3 domains grounded only by  $\mathcal{I}$ -DLV, times are substantially aligned (time differences below 10%) in 10 domains out of 25; as for the remaining domains, *gringo* outperforms  $\mathcal{I}$ -DLV in 6 domains, while  $\mathcal{I}$ -DLV performed better in 9 domains. To have another comparison perspective, Figure 12.2 compares  $\mathcal{I}$ -DLV and *gringo* on problems following ASP-Core-2 syntax but without featuring queries; as done before, the figure plots the number of grounded instances indicated on the  $x$ -axis within running

times given on the  $y$ -axis. We observe that even though the two systems have a very similar performance,  $\mathcal{I}$ -DLV grounded all the instances earlier than *gringo*, in less than 200 seconds.

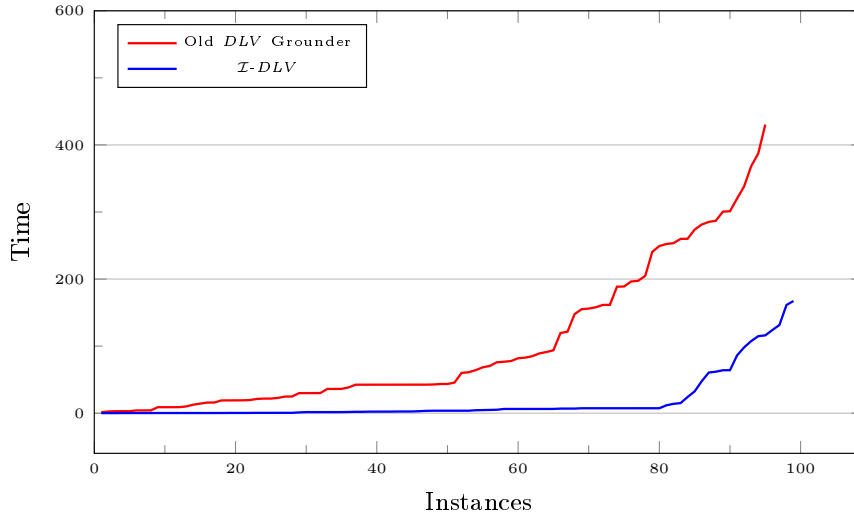


Figure 12.1: Sixth ASP Competition Benchmarks – comparison of the old *DLV* grounder and  $\mathcal{I}$ -DLV on *DLV* syntax

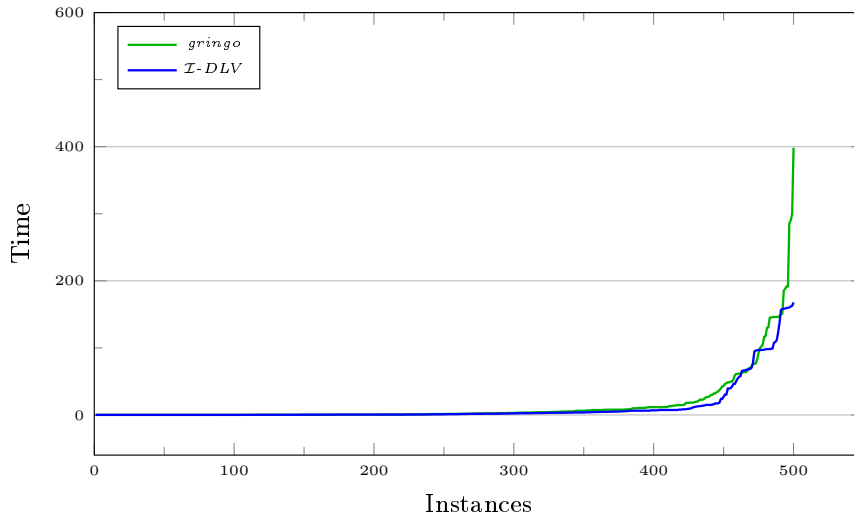


Figure 12.2: Sixth ASP Competition Benchmarks – comparison of *gringo* and  $\mathcal{I}$ -DLV on ASP-Core-2 syntax minus queries

In order to find further comparison settings outside of the ASP Competition series, we took into account the problems appearing in OpenRuleBench and in CQA benchmarks. This is also motivated by the fact that the ASP Competition mainly focuses on problems where solving task is more relevant with respect to the grounding one (indeed, as Figure 12.1 shows, apart from unsupported

syntax issues, all systems completed all instances), while OpenRuleBench and the selected tests from CQA suite demand a more significant work from the grounders.

Regarding OpenRuleBench, since it consists essentially of a query-based set of problems, that *gringo* would not accept “as-is”, we removed the query from the encodings and measured just the grounding times. Obviously, we did the same also for the *DLV* instantiator and  $\mathcal{I}$ -DLV: otherwise, these might have taken advantage from the magic-set technique, thus leading to an unfair test. The results are reported in Table 12.2: after domains names and corresponding number of instances, the next three pairs of columns show the number of grounded instances and the running time averaged over grounded instances. The last line reports the total running times for each system (600 seconds is added for timeout/memout instances, as systems were stopped if unable to finish before). It is evident that *DLV* is outperformed by both *gringo* and  $\mathcal{I}$ -DLV. As for *gringo* and  $\mathcal{I}$ -DLV, both grounded 102 instances out of 108; however, in the majority of domains  $\mathcal{I}$ -DLV appears to enjoy better performance. This is also evidenced in Figure 12.3 that provides another comparison perspective by plotting the running times over grounded instances in a cactus plot, as previously done. In particular, while in the previous benchmarks since not all systems accept the same syntax we separately compared them on commonly supported syntax, here there are no syntax issues, thus we compared all systems in the same plot.

Problem	# inst.	<i>gringo</i>		<i>DLV</i>		$\mathcal{I}$ -DLV	
		#grounded	time	#grounded	time	#grounded	time
Join1 A	3	2	206.03	1	117.57	2	226.02
Join1 B1	3	3	90.38	3	172.92	3	73.61
Join1 B2	3	3	27.41	3	35.34	3	18.78
Join Dupl. A	3	1	171.39	0	TO	1	168.92
Join Dupl. B1	3	2	113.02	2	261.19	2	83.70
Join Dupl. B2	3	3	112.55	3	168.43	3	86.60
Join2	1	1	40.36	1	88.32	1	25.18
Mondial	1	1	2.52	1	2.05	1	1.40
DBLP	1	1	49.49	1	27.06	1	17.67
Lubm1	2	2	55.95	2	31.98	2	20.51
Lubm2	2	2	56.01	2	37.49	2	20.46
Lubm9	2	2	55.95	2	37.02	2	20.51
Same Gen. R.	10	10	65.82	9	146.92	10	54.84
Trans. Closure	10	8	166.92	6	232.74	8	176.47
Wordnet	15	15	7.96	15	12.18	15	5.82
Wine	1	1	13.13	1	28.95	1	11.07
Magic Set	5	5	5.66	5	6.29	5	1.94
Win	10	10	9.91	10	8.02	10	5.29
Same Gen. S.N.	5	5	88.27	4	160.77	5	70.44
Indexing	15	15	5.17	15	2.13	15	1.43
Queens	5	5	0.10	5	0.34	5	0.10
Sixteen Puzzle	5	5	0.10	5	0.37	5	0.10
<b>Total Grounded Instances</b>		<b>102/108</b>		<b>96/108</b>		<b>102/108</b>	
<b>Average Time</b>		<b>43.95</b>		<b>60.64</b>		<b>37.50</b>	
<b>Total Running Time</b>		<b>8,303</b>		<b>13,021</b>		<b>7,613</b>	

Table 12.2: OpenRuleBench Benchmarks – number of grounded instances and grounding times in seconds

As for the CQA benchmarks, we selected a single database among the four available: indeed, they are randomly generated, and from a grounding perspective performance was very similar; moreover, among the available encoding we chose the *Pruning* one introduced in [99], as it was shown to be the one on which most of the execution time is spent on grounding. The executed part of

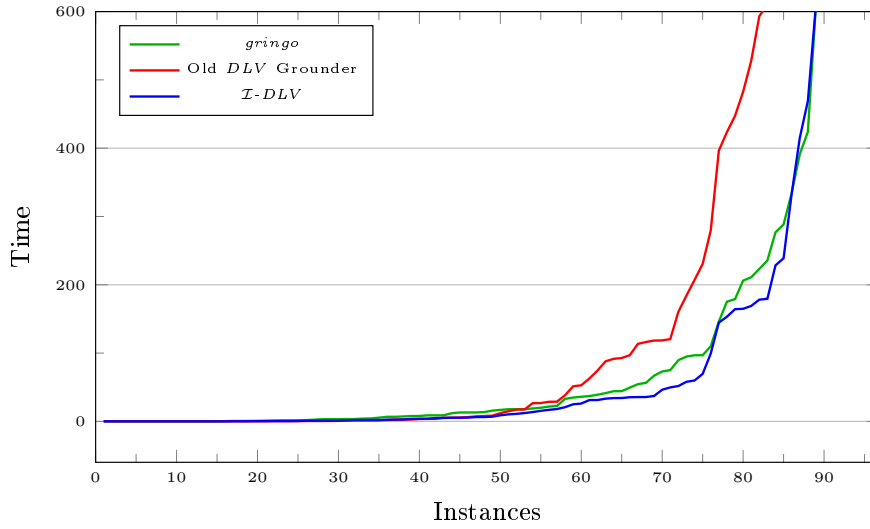


Figure 12.3: OpenRuleBench Benchmarks – grounding comparison

Problem	<i>gringo</i>	<i>I-DLV</i>
Query 1	20.55	14.78
Query 2	25.13	23.18
Query 3	29.90	19.34
Query 4	30.32	18.81
Query 5	12.85	8.23
Query 6	23.57	17.89
Query 7	21.72	19.21
Query 8	21.84	18.02
Query 9	33.55	26.92
Query 10	30.54	24.52
Query 11	31.31	23.19
Query 12	33.93	23.83
Query 13	37.33	29.80
Query 14	20.66	20.75
Query 15	21.99	19.47
Query 16	24.14	19.15
Query 17	27.78	16.52
Query 18	27.42	14.63
Query 19	34.79	25.67
Query 20	37.55	31.08
Query 21	32.82	28.03
Total Grounded Instances	210/210	210/2010
Total Running Time	5,797	4,430
Average Time	27.60	21.09

Table 12.3: CQA Benchmarks – grounding times in seconds

the benchmark consists in 10 instances with increasing sizes, varying from 100k to 1M tuples, and 21 encodings obtained by combining the *Pruning* encoding with 21 different extensions each one simulating a different database query. It is worth noticing that no encoding contains a query explicitly expressed in ASP: queries are simulated via plain normal rules. Table 12.3 shows the average grounding times for each encoding on the 10 instances: *gringo* and *I-DLV* were able to ground the whole suite within the given limits, while *DLV* was not able to execute any problem due to the presence of choice rules, which are not supported by the system. Also in this case, the results confirm the reliability of *I-DLV*, which obtained the best performance on each problem, even if the

special optimization means for query answering could not be employed due to the lack of explicitly expressed queries.

### 12.1.2 Deductive Database Benchmarks

For this setting, the natural choice was the query-based set of problems of the OpenRuleBench initiative. Besides  $DLV$ , we tested  $\mathcal{I}$ -DLV against  $XSB$  [120] (the latest available version, 3.6) which was among the clear winners and is currently one of the most widespread Logic Programming and Deductive Database systems. All systems support query answering, thus, differently from above, queries have not been removed. Moreover,  $DLV$  and  $\mathcal{I}$ -DLV were launched with their default configuration over all domains, and  $XSB$  has been launched with the exact OpenRuleBench settings, where the best configuration was set manually per each problem. The results are reported in Table 12.4: after domain names and corresponding number of instances, the next three pairs of columns show the number of solved instances and the running time averaged over solved instances. Similarly as above, the last line reports the total running times for each system.

Problem	# inst.	$XSB$		$DLV$		$\mathcal{I}$ -DLV	
		#solved	time	#solved	time	#solved	time
Join1 A free-free	3	1	19.76	1	122.84	2	233.98
Join1 A bound-free	3	2	25.04	3	37.09	3	18.16
join1 A free-bound	3	1	8.00	3	178.00	3	94.52
Join1 B1 free-free	3	2	13.39	3	181.62	3	70.96
Join1 B1 bound-free	3	3	2.65	3	3.16	3	1.42
Join1 B1 free-bound	3	2	4.46	3	11.64	3	6.03
Join1 B2 free-free	3	3	8.78	3	38.32	3	18.71
Join1 B2 bound-free	3	3	1.44	3	2.89	3	1.27
Join1 B2 free-bound	3	3	4.14	3	2.90	3	1.28
Join Duplicate A	3	1	93.36	0	TO	1	155.12
Join Duplicate B1	3	2	54.40	2	261.63	2	76.79
Join Duplicate B2	3	3	39.79	3	169.82	3	83.60
Join2	1	1	2.12	1	80.09	1	12.42
DBLP	1	1	92.41	1	23.81	1	15.00
Mondial	1	1	3.29	1	0.77	1	0.51
Same Gen. Recursion free-free	10	10	21.86	9	150.67	10	56.69
Same Gen. Recursion bound-free	10	10	18.32	10	153.06	10	46.27
Same Gen. Recursion free-bound	10	10	26.55	9	150.41	10	53.09
Trans. Closure free-free	10	10	148.81	6	217.35	9	191.18
Trans. Closure bound-free	10	10	93.42	6	207.93	9	174.05
Trans. Closure free-bound	10	10	28.99	10	3.13	10	1.44
Wordnet	15	15	1.70	15	13.99	15	5.08
Wine	1	1	5.89	1	27.76	1	10.62
Indexing	15	10	14.76	15	3.85	15	1.95
<i>Total Solved Instances</i>		<i>115/130</i>		<i>114/130</i>		<i>124/130</i>	
<i>Total Running Time</i>		<i>13,142</i>		<i>19,345</i>		<i>10,204</i>	
<i>Average Time</i>		<i>36.02</i>		<i>85.45</i>		<i>53.79</i>	

Table 12.4: OpenRuleBench Benchmarks – number of solved instances and query answering times in seconds

Also in this setting, results are very encouraging: not only  $\mathcal{I}$ -DLV behaves better than  $DLV$ , but it is definitely competitive against  $XSB$ . Indeed, in spite of a non-negligible variability from a problem to another,  $\mathcal{I}$ -DLV times are, on the overall, comparable with the ones of  $XSB$ ; in addition, it was able to solve even more instances within the allotted time (123 for  $\mathcal{I}$ -DLV, 115 for  $XSB$ ). The cactus plot in Figure 12.4 shows how  $\mathcal{I}$ -DLV outperformed  $DLV$  also on query answering tasks, and evidences a better scalability of  $\mathcal{I}$ -DLV with respect

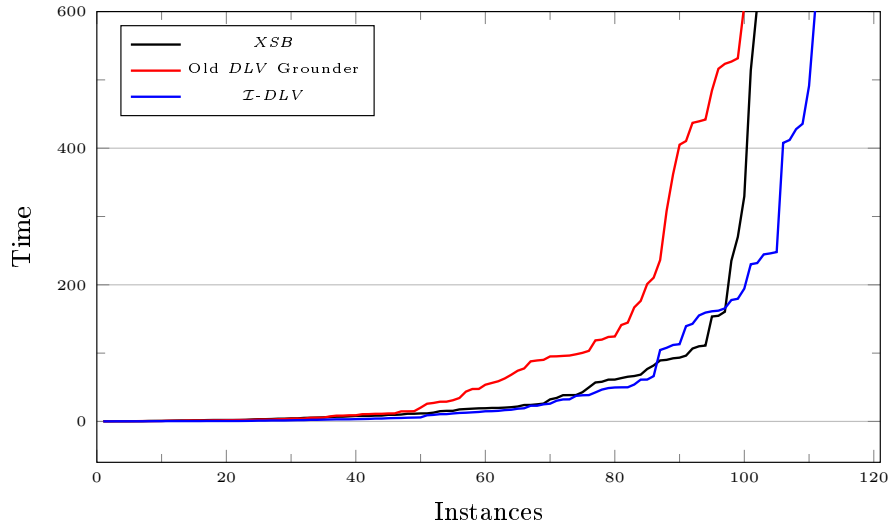


Figure 12.4: OpenRuleBench Benchmarks – query answering comparison

to *XSB*.

## 12.2 Impact of Customizability

In this section we present an experimental analysis on ad-hoc *I-DLV* configurations. Table 12.5 reports the comparison of the default version with a customized versions of *I-DLV* over a set of benchmarks taken from the Third ASP Competition [36], and the already mentioned Forth and Sixth ASP Competitions. For each benchmarking problem the custom configuration has been defined either via command-line options or via annotations. It is easy to see that significant improvements can be obtained by playing with grounding options.

Problem	# inst.	<i>I-DLV</i>		<i>I-DLV-Custom</i>	
		#grounded	time	#grounded	time
3rd Comp. - Grammar Based	10	10	69.29	10	16.10
3rd Comp. - Hydraulic Leaking	10	10	175.20	10	109.88
3th Comp. - Labyrinth	10	10	1.33	10	0.72
4th Comp. - Chemical Classification	30	26	120.60	26	97.43
4th Comp. - Complex Optimization	30	29	41.25	29	17.90
4th Comp. - Bottle Filling	30	30	4.22	30	3.85
4th Comp. - Labyrinth	30	30	1.51	30	0.76
6th Comp. - Complex Optimization	20	20	66.82	20	28.51
6th Comp. - Labyrinth	20	20	2.01	20	0.93
6th Comp. - Nomystery	20	20	3.46	20	2.36
6th Comp. - Steiner Tree	20	20	29.83	20	0.10

Table 12.5: Customizability – number of grounded instances and grounding times in seconds

In order to give an intuition of why this happens, we illustrate an interesting case, namely *Labyrinth*, where performance is significantly affected by the possibility to choose different strategies for the body orderings from rule to rule.



The encoding of such problem has not be updated in the last competitions, thus in all the three competitions the same encoding was employed and the same configuration has been adopted.

In particular, the custom configuration for *Labyrinth* has been obtained by considering this recursive rule:

$$\text{reach}(X, Y, T) :- \text{reach}(XX, YY, T), \text{dneighbor}(D, XX, YY, X, Y), \\ \text{conn}(XX, YY, D, T), \text{conn}(X, Y, E, T), \text{inverse}(D, E), \text{step}(T).$$

and by annotating it with:

```
%@rule_partial_order(@before={inverse(D,E)},
  @after={reach(XX,YY,T), dneighbor(D,XX,YY,X,Y),
  conn(XX,YY,D,T), conn(X,Y,E,T), step(T).}).
```

that corresponds to ask  $\mathcal{I}$ -DLV to select as first literals  $\text{inverse}(D, E)$  in its ordering strategy no matter how the other literals are positioned, and leads to reduce the average grounding time over all instances of around 50%.

Let us give some insights about the reasons behind the performance improvements. In Section 11.2 emerged that the best ordering strategy for *Labyrinth* is the  $\text{Combined}_I^+$  criterion. In particular, for recursive rules,  $\mathcal{I}$ -DLV reorders rule bodies at each iteration, and in the majority of the iterations and of the instances, if the  $\text{Combined}_I^+$  strategy is enabled, the rule body is ordered as follows:

$$\text{reach}(X, Y, T) :- \text{reach}(XX, YY, T), \text{step}(T), \text{conn}(XX, YY, D, T), \\ \text{inverse}(D, E), \text{conn}(X, Y, E, T), \text{dneighbor}(D, XX, YY, X, Y).$$

While the default  $\text{Combined}^+$  criterion tends to choose this other ordering:

$$\text{reach}(X, Y, T) :- \text{reach}(XX, YY, T), \text{step}(T), \text{conn}(XX, YY, D, T), \\ \text{conn}(X, Y, E, T), \text{inverse}(D, E), \text{dneighbor}(D, XX, YY, X, Y).$$

The heuristics, on which the other strategies are based, concern not only the size of the extensions of involved predicates but additional heuristics, as we have already discussed; in particular, since these strategies have as common root the  $\text{Combined}$  criterion, similarly to it, they may tend to prefer literals binding a larger number of variables. Despite being an effective heuristic in general, in this situation it emerged to not be the best choice. Indeed, the extension of the predicate  $\text{inverse}$  is very small in almost all instances, and it is better to add it as soon as possible, possibly at first. Notably, since this rule is recursive the impact of selecting a not best performing ordering is even more emphasized, since the rule is grounded multiple times.

### 12.3 Impact on Solvers

Except for stratified and non disjunctive programs for which, typically, instantiators are directly able to find the unique answer set, for programs that do not belong to this category, the ground program produced represents the input of

solvers which are capable of finding all their answer set(s). In this section, we report the results of an experimental activity aiming at assessing the performance of state-of-the-art solvers *clasp* and *wasp* when coupled with  $\mathcal{I}$ -DLV or *gringo*.

To this end, we executed four different combinations: (i) *gringo* paired with *clasp*, (ii) *gringo* coupled with *wasp*, (iii)  $\mathcal{I}$ -DLV combined with *clasp*, (iv)  $\mathcal{I}$ -DLV together with *wasp*. The latest available versions have been launched: *gringo* and *clasp* 5.2.1, *wasp* 2.1. Table 12.6 shows the experimental results: in red are reported the worst performing combinations, and in green the best ones. In general,  $\mathcal{I}$ -DLV improves the performance of both solvers allowing them to solve a greater number of instances; in particular, the combination  $\mathcal{I}$ -DLV and *clasp* is the best in most of the cases (14 problems out of 28). Excluding the 57 and 31 instances that respectively *clasp* and *wasp* are able to solve only with  $\mathcal{I}$ -DLV due the presence of queries, in the other problems the total number of instances solved by  $\mathcal{I}$ -DLV and *clasp* is 274, while for  $\mathcal{I}$ -DLV and *wasp* it is 243.

Problem	<i>gringo</i>	<i>clasp</i>	<i>gringo</i>	<i>wasp</i>	$\mathcal{I}$ -DLV	<i>clasp</i>	$\mathcal{I}$ -DLV	<i>wasp</i>
	#solved	time	#solved	time	#solved	time	#solved	time
Abstract Dialectical Frameworks	20	8.80	12	37.96	20	6.73	11	32.66
Combined Configuration	10	280.62	1	1.18	10	177.53	0	TO
Complex Optimization	17	137.03	4	104.50	18	158.91	6	159.74
Connected Still Life	6	238.91	12	47.14	6	240.76	12	53.20
Consistent Query Answering	0	US	0	US	20	85.42	18	86.96
Crossing Minimization	6	63.06	19	4.09	7	56.74	19	5.78
Graceful Graphs	9	68.09	4	44.73	9	140.55	5	127.29
Graph Coloring	15	137.57	7	68.77	15	162.01	8	113.37
Incremental Scheduling	13	116.77	5	114.58	12	69.61	7	93.36
Knight Tour With Holes	10	15.05	10	113.87	11	59.06	10	35.28
Labyrinth	13	103.86	10	138.07	12	152.09	10	71.22
Maximal Clique	0	TO	8	293.63	0	TO	9	361.82
MaxSAT	7	43.70	19	100.41	7	39.59	19	92.53
Minimal Diagnosis	20	8.41	20	36.95	20	8.78	20	26.73
Nomystery	7	91.73	8	57.10	9	101.84	9	78.96
Partner Units	14	35.14	10	236.87	14	20.24	9	140.92
Permutation Pattern Matching	11	167.83	20	173.43	20	15.68	20	23.26
Qualitative Spatial Reasoning	19	140.97	16	194.72	20	125.29	13	144.27
Reachability	0	US	0	US	20	145.75	6	141.04
Ricochet Robots	8	119.41	6	87.76	11	158.58	9	135.68
Sokoban	9	123.15	9	136.62	8	76.71	8	79.22
Stable Marriage	4	397.52	7	369.93	5	389.40	6	415.48
Steiner Tree	2	52.06	1	249.02	2	69.68	1	122.69
Strategic Companies	0	US	0	US	17	124.88	7	76.78
System Synthesis	0	TO	0	TO	0	TO	0	TO
Valves Location Problem	16	22.18	16	68.68	16	45.33	15	39.60
Video Streaming	13	56.89	0	TO	13	62.22	9	8.68
Visit-all	8	17.21	8	60.97	8	15.14	8	63.74
<b>Total Solved Instances</b>	<b>257/560</b>		<b>232/560</b>		<b>330/560</b>		<b>274/560</b>	
<b>Average Time</b>	<b>93.16</b>		<b>111.45</b>		<b>93.74</b>		<b>86.85</b>	

Table 12.6: Sixth Competition Solving Benchmarks – number of solved instances and solving times in seconds

Furthermore, we observe that there is no general and evident correlation between grounding and solving performance. In some problems, such as *Abstract Dialectical Framework*, *Complex Optimization* and *Nomystery*, a smaller grounding time does not necessarily imply a benefit on solving performance. Moreover, slight variations in the performance of the grounders lead to significant differences in the performance of the solvers. For instance, on the problem *Crossing Minimization* *clasp* enjoys a better performance with  $\mathcal{I}$ -DLV even if the grounding performance of both grounders are very close; while *wasp* has a

similar behaviour on *Graph Coloring*. Our conclusion is that solvers are mostly influenced by the form in which the ground program is, rather than the grounding performance.

Analysing the produced instantiation, it emerged a correlation between bodies length and number of rules. In particular, differently from *gringo*, in general  $\mathcal{I}$ -DLV tends to produce a smaller number of rules and smaller bodies, as many optimizations of  $\mathcal{I}$ -DLV are geared towards the rewriting of bodies in order to decrease their intrinsic complexity, such as the rewriting of syntactic features and isolated variables, or the decomposition rewriting. This aspects it outlined in the cactus plots reported next, in which restricting the comparison on problems whose syntax is accepted by both grounders we considered the number of rules produced and the average body length over all the instances. In detail, Figure 12.5 displays on the  $x$ -axis the number of instances solved and the respective number of ground rules on the  $y$ -axis, similarly Figure 12.6 plots the average body length on the  $y$ -axis.

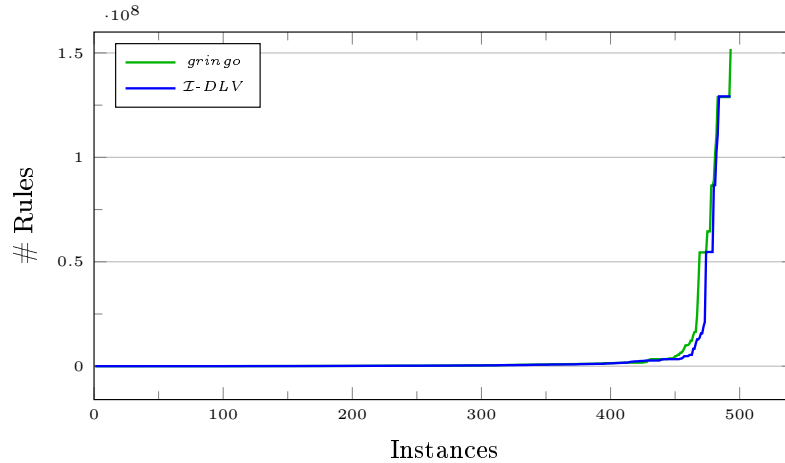


Figure 12.5: Instantiation: number of ground rules produced

Considering the solving performance, we can observe that solvers tend to perform better in cases in which even if they receive in input a greater number of rules, their bodies are significantly smaller. In a sense, it emerges that grounders should find a balance between the number of rules produced and their body length: splitting rules bodies, thus producing a greater number of rule, without sufficiently decreasing the lengths of bodies may not be preferable.

In order to get a more clear picture, we analysed the impact of the decomposition rewriting on solvers, that among all optimizations is the one that mostly acts on the lengths of bodies. We proved in Section 11.3 how a decomposition tailored on grounding costs might significantly increase grounding performance; we combined the same three tested versions of with both *clasp* and *wasp* and launched the resulting configurations over the same set of benchmarks.

Table 12.7 reports the average times and the number of solved instances; the dashes stand for memory outs or time outs. First of all, we observe that in many cases applying our heuristic-based decompositions corresponds to clear improvements for both *clasp* and *wasp*; moreover, sometimes both solvers benefit from the decomposition rewriting even if there is no evidence of improvements

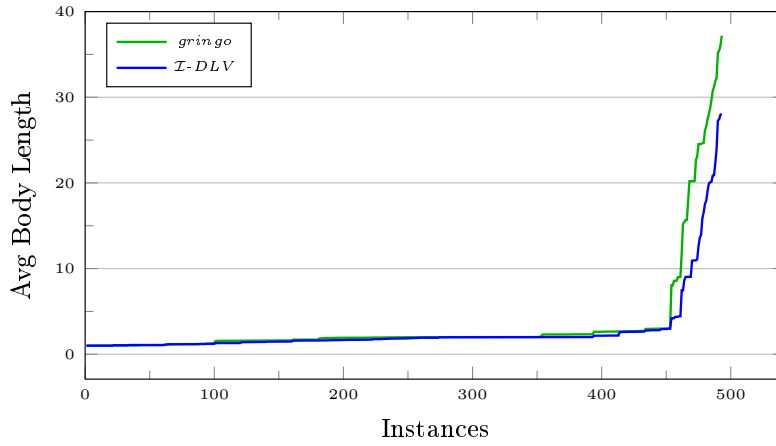


Figure 12.6: Instantiation: average body length

on the grounding times. One can also note that the “blind usage” of *lpopt* leads, in general, to a loss of performance for both solvers: in spite the gain in some cases, the total number of solved instances within the suite is significantly lower. It can be observed also that there are some corner cases in which the black-box approach eventually allows a solver to solve some instance more than both the other two versions; however, the same do not hold for the other solver. This suggests that a deeper analysis is needed, and that one should explicitly tailor the heuristics guiding of the smart decomposition to the given solver at hand; for instance, one can start from the results in [18], where emerged that the performance of modern solvers is influenced by the tree-width of the input program. We believe that tailoring SMARTDECOMPOSITION also on the solving step may yield to important improvements on the whole computational process, even if currently this approach seems to not be easy, since it is not clear whether we should try to find general heuristic criteria which may help all solvers indiscriminately, or intrinsic aspects specific for the solver at hand. In addition, we recently started to deploy an automatic solver selector, based on some machine-learning techniques which are applied to inductively choose the best solver, depending on some inherent features of the instantiation produced by *I-DLV*, which has been preliminarily presented in [29]. Our intent is to better understand how the output of *I-DLV* should be specifically produced to further help the subsequent solving step.

Problem	$\mathcal{I}$ -DLV #solved	clasp time	$\mathcal{I}$ -DLV #solved	clasp time	$\mathcal{I}$ -DLV-SD #solved	clasp time	$\mathcal{I}$ -DLV #solved	wasp time	$\mathcal{I}$ -DLV #solved	wasp time	$\mathcal{I}$ -DLV-SD #solved	wasp time
Abstract Dialectical Frameworks	20	6.80	20	7.35	20	6.73	11	32.58	11	20.85	11	32.57
Combined Configuration	8	138.99	9	174.25	10	177.53	1	342.41	0	TO	0	TO
Complex Optimization	18	158.52	19	174.00	18	158.91	6	160.05	5	97.40	6	159.74
Connected Still Life	6	228.29	6	247.83	6	240.76	12	52.91	12	78.44	12	53.20
Crossing Minimization	7	56.52	6	64.36	7	56.74	19	3.53	19	2.43	19	5.78
Graceful Graphs	9	134.94	10	129.13	9	140.55	6	178.58	4	129.51	5	127.29
Graph Coloring	15	162.29	15	166.51	15	162.01	8	120.76	9	261.49	8	113.37
Incremental Scheduling	12	66.25	12	83.14	12	69.61	8	141.54	6	166.86	7	93.36
Knight Tour With Holes	11	56.85	10	27.12	11	59.06	10	35.51	8	67.01	10	35.28
Labyrinth	12	152.76	11	124.78	12	152.09	10	71.44	9	127.80	10	71.22
Maximal Clique	0	TO	0	TO	0	TOE	9	367.55	9	367.06	9	361.82
MaxSAT	7	39.68	7	46.76	7	39.59	19	93.24	19	97.58	19	92.53
Minimal Diagnosis	20	8.91	20	8.49	20	8.78	20	27.23	20	25.78	20	26.73
Nomystery	8	139.76	8	42.78	9	101.84	8	35.78	9	31.83	9	78.96
Partner Units	14	20.34	14	20.34	14	20.24	5	134.54	9	140.24	9	140.92
Permutation Pattern Matching	11	161.75	16	124.80	20	15.68	20	181.89	8	199.78	20	23.26
Qualitative Spatial Reasoning	20	124.99	20	125.34	20	125.29	13	143.94	13	143.51	13	144.27
Ricochet Robots	9	66.51	12	118.13	11	158.58	7	217.92	8	90.08	9	135.68
Sokoban	8	73.76	9	82.82	8	76.71	8	88.05	9	62.83	8	79.22
Stable Marriage	5	393.02	8	375.71	5	389.40	7	423.62	7	438.05	6	415.48
Steiner Tree	2	69.99	2	70.02	2	69.68	1	122.75	1	122.63	1	122.69
Valves Location Problem	16	42.71	16	25.90	16	45.33	15	40.05	15	38.29	15	39.6
Video Streaming	13	62.40	10	77.52	13	62.22	9	8.64	0	TO	9	8.68
Visit-all	8	16.78	8	15.08	8	15.14	8	65.34	8	63.29	8	63.74
<b>Total Solved Instances</b>	<b>259/560</b>	<b>92.01</b>	<b>268/560</b>	<b>95.42</b>	<b>273/560</b>	<b>88.60</b>	<b>240/560</b>	<b>105.06</b>	<b>218/560</b>	<b>107.82</b>	<b>243/560</b>	<b>85.80</b>
<b>Average Time</b>												

Table 12.7: Sixth Competition Solving Benchmarks – impact of SMARTDECOMPOSITION on solving

## Part IV

# Related Work and Conclusion



In this last part we discuss related work and draw our conclusions. In particular:

- Chapter 13 is devoted to related work.
- Chapter 14 reports conclusions highlighting future work.





# Chapter 13

## Related Work

Some connections to our work can be found with other rule-based engines and deductive database systems; an interesting overview can be found in [95]. Such systems have common roots, even though differ in several aspects, especially with respect to supported languages and evaluation mechanisms; in particular, *XSB* [120], among the most prominent, is indeed a Prolog system which relies on a top-down evaluation. *I-DLV*, as already discussed in this work, is an ASP grounder relying on a bottom-up approach.

As for grounding processes, different approaches are pursued by *lparse* [121], that supports  $\omega$ -restricted [121] programs, and *GIDL* [130], a grounder for  $\text{FO}^+$ . Furthermore, a radically different approach is followed by systems such as *Gasp* [44], *Asperix* [89, 87], *Omega* [46] and *Alpha* [129] tailored on *lazy grounding*, in which, contrarily to the classical ground and solve approach, grounding and solving steps are interleaved, and rules are grounded on-demand during solving.

Stronger connections can be found with other mainstream ASP grounders, such as the grounder module of *DLV* [55], and *gringo* [66]: they all share the basic evaluation approach.

With respect to the old *DLV* instantiator, *I-DLV* features many differences and novelties. From the syntactic point of view, contrarily from its predecessor, *I-DLV* fully supports the ASP-Core-2 standard language, and hence it can interoperate with the state-of-the-art solver solvers. In addition, it is empowered with advanced mechanisms for customizability and interoperability. Moreover, *I-DLV* is the outcome of a completely renewed implementation started from scratch: during both the design and the implementation processes the emphasis has been kept on a lightweight modular architecture that eases the introduction of optimization techniques. Regarding the optimizations intervening in the grounding process of *I-DLV*, apart from a similar backjumping [111] mechanism to process rule instantiation and the magic sets technique for query answering, it incorporates improved and novel optimizations that impact over all phases of the computational machinery. An on-demand indexing strategy was present also in the old *DLV* grounder [39]: it allowed to index each predicate on a single argument, and was based on different data structures which did not permit perfect hashing. The indexing strategies presented herein are more general, more flexible, and, as experiments confirmed, more effective. Concerning body

ordering, *DLV* employed the *Combined* criterion [91]. In its original form such criterion was only geared towards improving the evaluation of join operations in rule bodies. In *I-DLV* such strategy has been enhanced by taking into consideration *ASP-Core-2* syntax, and further variants have been introduced in order to outline correlations between body ordering strategies and other crucial optimizations. Concerning the rewriting of isolated variables, the old *DLV* grounder was endowed with a dedicated rewriting module that implemented, among others, also this functionality. Such a module was monolithic: either all strategies were applied, or none at all. However, every rewriting shows its benefits only in specific scenarios, and can worsen performance in others; thus, the contemporary activation of many is not typically convenient. In addition, since *DLV* supported a restricted syntax, linguistic extensions were handled in different modes. Lastly, as experiments confirm, *I-DLV* enjoys significantly better performance both as instantiator and as deductive database system. Eventually, the decomposition rewriting, the techniques about pushing down selections, managing isolated variables, pre-determining the admissibility of variable substitutions and anticipating strong constraints instantiation were not present in the old *DLV* grounder.

The grounder *gringo* constituted, so far, the only ASP grounder supporting the *ASP-Core-2* standard and hence able to interoperate with solvers, thus becoming the most commonly used ASP grounder, as emerged from the last ASP Competitions; it is also a long-lasting player, part of a larger family of Answer Set Programming tools that already optimize interoperation. Differently from *gringo*, *I-DLV* features novel customization properties, such annotations and special command-line options and *I-DLV* implements specific deductive-database-oriented features, such as magic sets. Furthermore, *gringo* and *I-DLV* incorporates different optimization techniques in order to improve performance. For instance, the rule instantiation process of *gringo* relies on a backjumping algorithm enhanced with the *binder splitting* method, geared toward avoiding the re-generation of some ground instances in situations in which a body literal binds both relevant and non relevant variables [71].

Further correlations can be found with *lpopt* [16]. *I-DLV* inherits from *lpopt* the way in which rules are converted to hypergraphs and decomposed. However, thanks to the embedding of decomposition mechanisms into the grounder, the decomposition rewriting can be better integrated with the other optimizations featured by *I-DLV*, thus fully leveraging on the benefits stemming from their synergic work. Moreover, thanks to this integration *I-DLV* aims at taking full advantage from decompositions, still avoiding performance drawbacks by trying to predict the effects of rewritings. Consequently, in *I-DLV* decompositions can be selected not only considering the non-ground structure of the encoding at hand, but also on the basis of the instance with which it is coupled, so that, combining an encoding  $E$  with an instance  $I$  might not produce the same decompositions if  $E$  is instead paired with a different instance  $I'$ . In addition, *I-DLV* generates multiple decompositions and automatically selects the best one according to the studied heuristic estimation about the cost of grounding a rule. The experiments evidenced the effectiveness of our approach, not only on the grounding side, but also on the solving performance. In addition, we preliminarily defined an abstract heuristic-based decomposition algorithm, allowing to customize decompositions according to different desiderata, such as

the improvement of solving performance.

The usage of annotations has already been proposed in the literature [127, 85, 114]. The work in [127] is one of the most related to our setting, as applies annotations to ASP programs. In particular, it introduces the language *LANA*, that allows one to express meta-information that acts as an external tool for development support (such as documentation, testing, verification, or code completion); therefore, annotations do not have a direct impact on program evaluation. Similarly, the work in [85] refers to annotations as a tool for better documenting Prolog programs, while the one in [114] is oriented towards semantic annotations of ontologies. A different approach is pursued in [52], where some syntactic means for expressing (desirable) properties of ASP HEX programs are introduced. However, the aim of the introduction of annotations in  *$\mathcal{I}$ -DLV* is different from the mentioned mechanisms: as described in Section 10.3.2, they allow an inline customization of the actual grounding machinery: it acts internally, by changing the behaviour of the system that can hence be “trained” in order to follow specific user’s desiderata related to the grounding process.



## Chapter 14

# Conclusions

This thesis focuses on the ex-novo realization of a new modern and efficient ASP instantiator. The initial questions we moved from are: *How can we optimize the state-of-the-art instantiation techniques? Which other techniques could help modern grounders to enjoy better performance?* In order to properly answer, we studied a series of techniques geared towards the optimization of the grounding process that led us to implement and release a new ASP grounder:  *$\mathcal{I}$ -DLV*.

Many techniques employed by modern ASP grounders derive from the database field, like indexing and body ordering strategies. We herein proposed different data structures and indexing strategies, comparing advantages and drawbacks thanks to proper implementations into  *$\mathcal{I}$ -DLV*, and eventually adopted the balanced on-demand indexing strategy proposed herein. On the other hand, we presented a set of ordering strategies modelled on ASP-Core-2 and designed on the basis of different heuristics.

Furthermore, we noticed that real ASP systems are highly influenced by the adopted syntax of input programs and also by the form the programs are written; hence, if one wants to optimize input programs has to tune encodings in a way that conflicts with the intrinsic declarative nature of ASP. Hence, we designed a general technique for fine-tuning an encoding in a more suitable form by automatically decomposing rules. Since systems may benefit in different ways from decompositions, we defined an abstract algorithm, SMARTDECOMPOSITION, whose decisive steps are customizable according to different policies and heuristics. Consequently, we defined a specific version geared towards the optimization of the instantiation, tailored on the cost of grounding a rule and embedded into  *$\mathcal{I}$ -DLV*. Experiments confirmed that the defined version grants significant improvements to the instantiation process of  *$\mathcal{I}$ -DLV*. In the future, we plan to further improve the choice of the threshold by taking advantage from automatic and more advanced methods, such as machine learning guided machineries.

Moreover, interestingly, although our proposal was explicitly tailored to the optimization of grounding times, our experiments evidenced positive effects also on solvers, in general. We hence believe that tailoring SMARTDECOMPOSITION specifically on the solving phase may provide even more significant benefits on the whole computational process. Currently, this seems to be a quite hard task, since different solvers show different behaviours. Therefore, we plan to study both general aspects which may help all solvers indiscriminately, and intrinsic

aspects specific for the solver at hand.

In addition, we proposed several fine-tuning optimizations. Most of them are designed for improving rule instantiation, as it represents the core of the computation, while others relate to rewriting strategies for efficient handling ASP linguistic features; similarly to the decomposition rewriting, these optimizations aim at automatically rewriting encodings in forms which can be handled more efficiently.

These techniques are intended to be profitably integrated into a classical ASP grounder. In particular, *I-DLV* incorporates all the aforementioned optimizations, leveraging on their synergy to perform an efficient grounding process. Some techniques are strictly related and deeply influence each other's effects. Therefore, *I-DLV* has been endowed with a flexible design, that allows to customize its default behaviour, and it features different means for this purpose; we mention here the *annotations* directives, a form of meta-data allowing, to different extents, to guide the grounding process at a fine-grained level. Annotations permit also to tailor *I-DLV* grounding process according to the specific scenario at hand and to behaviours of solvers. The aim is fostering the use of ASP on both real and scientific contexts, providing external means for experimenting with ASP systems. As future work, we plan to further widen the range of aspects over which annotations can intervene.

The system features full support to the ASP-Core-2 standard as well as interoperability with current state-of-the-art ASP solvers, and is also a full-fledged deductive database engine. Despite being released recently, *I-DLV* performance is promising and compatible with mainstream systems, as evicted in our experiments, and in the wins at latest official ASP Competition [69].

As future work, we plan to better study tight integrations with ASP solvers, and equip *I-DLV* with a set of advanced mechanisms and tools for interoperability and integration with other systems and formalisms. In addition to the already defined rewriting techniques, we plan to endow *I-DLV* with further pre-processing steps aiming at making performance less encoding-dependent: we believe that such means are of great importance for fostering and easing the usage of ASP in practice, fully complying with the declarative power of ASP.

*I-DLV* source and binaries are available from the official repository [38]. Preliminary results presented in this thesis as well as complementary projects have been published in [27, 28, 2, 31, 32, 30, 33, 25, 26, 61].

# Bibliography

- [1] Michael Abseher, Nysret Musliu, and Stefan Woltran. htd - A Free, Open-Source Framework for (Customized) Tree Decompositions and Beyond. In Domenico Salvagnin and Michele Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings*, volume 10335 of *Lecture Notes in Computer Science*, pages 376–386. Springer, 2017.
- [2] Mario Alviano, Francesco Calimeri, Carmine Dodaro, Davide Fuscà, Nicola Leone, Simona Perri, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari. The ASP system DLV2. In Balduccini and Janhunen [7], pages 215–221.
- [3] Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. Advances in WASP. In Calimeri et al. [37], pages 40–54.
- [4] Mario Alviano, Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Magic sets for disjunctive datalog programs. *Artificial Intelligence*, 187:156–192, 2012.
- [5] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a Theory of Declarative Knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Inc., Washington DC, 1988.
- [6] Marcello Balduccini, Michael Gelfond, Richard Watson, and Monica Nogueira. The USA-Advisor: A Case Study in Answer Set Planning. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01)*, volume 2173 of *Lecture Notes in Computer Science*, pages 439–442. Springer, 2001.
- [7] Marcello Balduccini and Tomi Janhunen, editors. *Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings*, volume 10377 of *Lecture Notes in Computer Science*. Springer, 2017.
- [8] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–15, Cambridge, Massachusetts, 1986.



- [9] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
- [10] Chitta Baral and Michael Gelfond. Reasoning Agents in Dynamic Domains. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 257–279. Kluwer Academic Publishers, 2000.
- [11] Chitta Baral and Cenk Uyan. Declarative Specification and Solution of Combinatorial Auctions Using Logic Programming. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01)*, volume 2173 of *Lecture Notes in AI (LNAI)*, pages 186–199. Springer Verlag, 2001.
- [12] Victor A. Bardadym. Computer-Aided School and University Timetabling: The New Wave. In Edmund Burke and Peter Ross, editors, *Practice and Theory of Automated Timetabling, First International Conference 1995*, volume 1153 of *Lecture Notes in Computer Science*, pages 22–45. Springer, 1996.
- [13] Rachel Ben-Eliyahu and Rina Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.
- [14] Rachel Ben-Eliyahu and Luigi Palopoli. Reasoning with Minimal Models: Efficient Algorithms and Applications. In *Proceedings Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR-94)*, pages 39–50, 1994.
- [15] Leopoldo E. Bertossi, Anthony Hunter, and Torsten Schaub, editors. *Inconsistency Tolerance*, volume 3300 of *Lecture Notes in Computer Science*. Springer, 2005.
- [16] Manuel Bichler, Michael Morak, and Stefan Woltran. lpopt: A Rule Optimization Tool for Answer Set Programming. In Manuel V. Hermenegildo and Pedro López-García, editors, *Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers*, volume 10184 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2016.
- [17] Manuel Bichler, Michael Morak, and Stefan Woltran. The power of non-ground rules in answer set programming. *TPLP*, 16(5-6):552–569, 2016.
- [18] Bernhard Bliem, Marius Moldovan, Michael Morak, and Stefan Woltran. The impact of treewidth on ASP grounding and solving. In Carles Sierra, editor, *IJCAI*, pages 852–858. ijcai.org, 2017.
- [19] Loreto Bravo and Leopoldo Bertossi. Logic programming for consistently querying data integration systems. In *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico*, pages 10–15, 2003.

- [20] 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*, volume 141 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2006.
- [21] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.
- [22] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.
- [23] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, and Torsten Schaub. ASP-Core-2: 4th ASP Competition Official Input Language Format, 2013. <https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.01c.pdf>.
- [24] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, and Torsten Schaub. Asp-core-2: Input language format, 2015. <https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03c.pdf>.
- [25] Francesco Calimeri, Davide Fuscà, Stefano Germano, Simona Perri, and Jessica Zangari. Embedding ASP in Mobile Systems: Discussion and Preliminary Implementations. In *Proceedings of the Eighth Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2015), workshop of the 31st International Conference on Logic Programming (ICLP 2015)*, 2015.
- [26] Francesco Calimeri, Davide Fuscà, Stefano Germano, Simona Perri, and Jessica Zangari. Boosting the Development of ASP-Based Applications in Mobile and General Scenarios. In *AI\* IA 2016 Advances in Artificial Intelligence: XVth International Conference of the Italian Association for Artificial Intelligence, Genova, Italy, November 29–December 1, 2016, Proceedings*, volume 10037, page 223. Springer, 2016.
- [27] Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari. *I-DLV*: The New Intelligent Grounder of DLV. In Giovanni Adorni, Stefano Cagnoni, Marco Gori, and Marco Maratea, editors, *AI\*IA 2016: Advances in Artificial Intelligence - XVth International Conference of the Italian Association for Artificial Intelligence, Genova, Italy, November 29 - December 1, 2016, Proceedings*, volume 10037 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2016. Winner of the Best Paper Award.
- [28] Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari. *I-DLV*: The New Intelligent Grounder of DLV. *Intelligenza Artificiale*, 11(1):5–20, 2017.

- [29] Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari. *I-DLV+MS*: Preliminary Report on an Automatic ASP Solver Selector. In *24th RCRA International Workshop on “Experimental Evaluation of Algorithms for solving problems with combinatorial explosion”*, 2017. To appear.
- [30] Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari. External Computations and Interoperability in the new DLV Grounder. In *Proceedings of the 16th International Conference of the Italian Association for Artificial Intelligence (AI\*IA 2017)*, Bari, Italy, 2017. To appear.
- [31] Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari. The ASP Instantiator *I-DLV*. In *Proceedings of the First International Workshop on Practical Aspects of Answer Set Programming (PAoASP 2017)*, Espoo, Finland, 2017.
- [32] Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari. The new DLV Grounder: External Computations, Interoperability and Customizability. In *Proceedings of the Fourth International Workshop on Grounding and Transformations for Theories with Variables (GTTV 2017)*, Espoo, Finland, 2017.
- [33] Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari. Optimizing Answer Set Computation via Heuristic-Based Decomposition. In *Practical Aspects of Declarative Languages - 19th International Symposium, PADL 2018, Los Angeles, CA, USA, 8-9 January 2018, Proceedings*, 2018. To Appear, Winner of the Best Paper Award.
- [34] Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca. Design and results of the Fifth Answer Set Programming Competition. *Artificial Intelligence*, 231:151–181, 2016.
- [35] Francesco Calimeri, Giovambattista Ianni, Simona Perri, and Jessica Zangari. The Eternal Battle between Determinism and Nondeterminism: preliminary Studies in the Sudoku Domain. *20th RCRA International Workshop on “Experimental Evaluation of Algorithms for solving problems with combinatorial explosion”*, 2013.
- [36] Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca. The Third Open Answer Set Programming Competition. *Theory and Practice of Logic Programming*, 14(1):117–135, 2014.
- [37] Francesco Calimeri, Giovambattista Ianni, and Mirosław Trzuszczynski, editors. *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, volume 9345 of *Lecture Notes in Computer Science*. Springer, 2015.
- [38] Francesco Calimeri, Simona Perri, Davide Fuscà, and Jessica Zangari. *I-DLV* repository, since 2016. <https://github.com/DeMaCS-UNICAL/I-DLV>.

- [39] Gelsomina Catalano, Nicola Leone, and Simona Perri. On demand Indexing Techniques for the DLV Instantiator. In *Proceedings of the Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2008)*, Udine, Italy, 2008.
- [40] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.
- [41] Keith L. Clark. Negation as Failure. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.
- [42] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *History of programming languages—II*, pages 331–367. ACM, 1996.
- [43] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.
- [44] Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. GASP: Answer Set Programming with Lazy Grounding. *Fundamenta Informaticae*, 96(3):297–322, 2009.
- [45] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3):374–425, 2001.
- [46] Luis Fariñas del Cerro, Andreas Herzig, and Jérôme Mengin, editors. *Logics in Artificial Intelligence - 13th European Conference, JELIA 2012, Toulouse, France, September 26-28, 2012. Proceedings*, volume 7519 of *Lecture Notes in Computer Science*. Springer, 2012.
- [47] Tina Dell’Armi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerald Pfeifer. Aggregate Functions in DLV. In Marina de Vos and Alessandro Provetti, editors, *Proceedings ASP03 - Answer Set Programming: Advances in Theory and Implementation*, pages 274–288, Messina, Italy, September 2003. Online at <http://CEUR-WS.org/Vol-78/>.
- [48] Carmine Dodaro, Philip Gasteiger, Nicola Leone, Benjamin Musitsch, Francesco Ricca, and Konstantin Schekotihin. Combining Answer Set Programming and Domain Heuristics for Solving Hard Industrial Problems (Application Paper). *Theory and Practice of Logic Programming*, 16(5-6):653–669, 2016.
- [49] William F. Dowling and Jean H. Gallier. Linear-time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *Journal of Logic Programming*, 3:267–284, 1984.
- [50] Thomas Eiter, Wolfgang Faber, Michael Fink, and Stefan Woltran. Complexity results for answer set programming with bounded predicate arities and implications. *Annals of Mathematics and Artificial Intelligence*, 51(2–4):123–165, 2007.
- [51] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative Problem-Solving using the DLV System. In *Logic-based artificial intelligence*, pages 79–103. Springer, 2000.

- [52] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. Liberal Safety for Answer Set Programs with External Sources. In Marie desJardins and Michael L. Littman, editors, *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*. AAAI Press, 2013.
- [53] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.
- [54] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer Set Programming: A Primer. In *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School - Tutorial Lectures*, pages 40–110, Brixen-Bressanone, Italy, August-September 2009.
- [55] Wolfgang Faber, Nicola Leone, and Simona Perri. The Intelligent Grounder of DLV. In Esra Erdem, Joohyung Lee, Yuliya Lierler, and David Pearce, editors, *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2012.
- [56] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In José Júlio Alferes and João Leite, editors, *Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004)*, volume 3229 of *Lecture Notes in AI (LNAI)*, pages 200–212. Springer Verlag, September 2004.
- [57] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278–298, 2011. Special Issue: John McCarthy’s Legacy.
- [58] Wolfgang Faber, Gerald Pfeifer, and Nicola Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278 – 298, 2011. John McCarthy’s Legacy.
- [59] Paolo Ferraris. Answer Sets for Propositional Theories. Available via the author’s homepage at <http://www.cs.utexas.edu/users/otto/papers/proptheories.ps>, 2004.
- [60] Gerhard Friedrich and Volodymyr Ivanchenko. Diagnosis from first principles for workflow executions. Technical report, Alpen Adria University, Applied Informatics, Klagenfurt, Austria, 2008. [http://proserver3-iwas.uni-klu.ac.at/download\\_area/Technical-Reports/technical\\_report\\_2008\\_02.pdf](http://proserver3-iwas.uni-klu.ac.at/download_area/Technical-Reports/technical_report_2008_02.pdf).
- [61] Davide Fuscà, Stefano Germano, Jessica Zangari, Marco Anastasio, Francesco Calimeri, and Simona Perri. A framework for easing the development of applications embedding answer set programming. In James Cheney and Germán Vidal, editors, *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pages 38–49. ACM, 2016.
- [62] Alfredo Garro, Luigi Palopoli, and Francesco Ricca. Exploiting agents in e-learning and skills management context. *AI Communications – The European Journal on Artificial Intelligence*, 19(2):137–154, 2006.

- [63] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Clingo = ASP + control*: Preliminary report. In M. Leuschel and T. Schrijvers, editors, *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*, volume arXiv:1405.3694v1, 2014. Theory and Practice of Logic Programming, Online Supplement.
- [64] M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, 2012.
- [65] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Javier Romero, and Torsten Schaub. Progress in clasp series 3. In Calimeri et al. [37], pages 368–383.
- [66] Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. Advances in *gringo* series 3. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, volume 6645 of *Lecture Notes in Computer Science*, pages 345–351. Springer, 2011.
- [67] Martin Gebser, Marco Maratea, and Francesco Ricca. The Design of the Sixth Answer Set Programming Competition. In Calimeri et al. [37], pages 531–544.
- [68] Martin Gebser, Marco Maratea, and Francesco Ricca. What’s Hot in the Answer Set Programming Competition. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the 13th AAI Conference on Artificial Intelligence, Feb 12-17, 2016, Phoenix, Arizona, USA.*, pages 4327–4329. AAI Press, 2016.
- [69] Martin Gebser, Marco Maratea, and Francesco Ricca. The design of the seventh answer set programming competition. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 3–9. Springer, 2017.
- [70] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo : A new grounder for answer set programming. In Chitta Baral, Gerhard Brewka, and John Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning — 9th International Conference, LPNMR'07*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271, Tempe, Arizona, May 2007. Springer Verlag.
- [71] Martin Gebser, Torsten Schaub, and Sven Thiele. GrinGo : A New Grounder for Answer Set Programming. In 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*, pages 266–271. Springer, 2007.
- [72] Michael Gelfond. Representing Knowledge in A-Prolog. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic. Logic Programming and Beyond*, volume 2408 of *Lecture Notes in Computer Science*, pages 413–451. Springer, 2002.

- [73] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, WA, Aug 15-19, 1988 (2 Volumes)*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.
- [74] Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3/4):365–385, 1991.
- [75] Enrico Giunchiglia, Yulia Lierler, and Marco Maratea. Answer Set Programming Based on Propositional Satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, 2006.
- [76] Georg Gottlob, Gianluigi Greco, and Francesco Scarcello. Treewidth and hypertree width. In Lucas Bordeaux, Youssef Hamadi, and Pushmeet Kohli, editors, *Tractability: Practical Approaches to Hard Problems*, pages 3–38. Cambridge University Press, 2014.
- [77] Georg Gottlob, Martin Grohe, Nysret Musliu, Marko Samer, and Francesco Scarcello. Hypertree decompositions: Structure, algorithms, and applications. In Dieter Kratsch, editor, *Graph-Theoretic Concepts in Computer Science, 31st International Workshop, WG 2005, Metz, France, June 23-25, 2005, Revised Selected Papers*, volume 3787 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2005.
- [78] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions: A survey. In *International Symposium on Mathematical Foundations of Computer Science*, pages 37–57. Springer, 2001.
- [79] Thomas Hammerl, Nysret Musliu, and Werner Schafhauser. Metaheuristic algorithms and tree decomposition. In *Springer Handbook of Computational Intelligence*, pages 1255–1270. Springer, 2015.
- [80] Tomi Janhunen. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 16(1-2):35–86, 2006.
- [81] Tomi Janhunen, Ilkka Niemelä, Patrik Simons, and Jia-Huai You. Unfolding Partiality and Disjunctions in Stable Model Semantics. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2000), April 12-15, Breckenridge, Colorado, USA*, pages 411–419. Morgan Kaufmann Publishers, Inc., 2000.
- [82] Benjamin Kaufmann, Nicola Leone, Simona Perri, and Torsten Schaub. Grounding and solving in answer set programming. *AI Magazine*, 37(3):25–32, 2016.
- [83] David B. Kemp and Peter J. Stuckey. Semantics of Logic Programs with Aggregates. In Vijay A. Saraswat and Kazunori Ueda, editors, *Proceedings of the International Symposium on Logic Programming (ISLP'91)*, pages 387–401. MIT Press, 1991.

- [84] Phokion G. Kolaitis, Enela Pema, and Wang-Chiew Tan. Efficient querying of inconsistent databases with binary integer programming. *PVLDB*, 6(6):397–408, 2013.
- [85] Marija Kulas. Debugging prolog using annotations. *Electronic Notes in Theoretical Computer Science*, 30(4):235–255, 1999.
- [86] Joohyung Lee and Vladimir Lifschitz. Loop Formulas for Disjunctive Logic Programs. In *Proceedings of the Nineteenth International Conference on Logic Programming (ICLP-03)*, pages 451–465. Springer Verlag, December 2003.
- [87] Claire Lefèvre, Christopher Béatrix, Igor Stéphan, and Laurent Garcia. ASPeRiX, a first-order forward chaining approach for answer set computing. *Theory and Practice of Logic Programming*, 17(3):266–310, 2017.
- [88] Claire Lefèvre and Pascal Nicolas. A first order forward chaining approach for answer set computing. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*, pages 196–208. Springer, 2009.
- [89] Claire Lefèvre and Pascal Nicolas. The first version of a new asp solver : Asperix. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning — 10th International Conference (LPNMR 2009)*, volume 5753 of *Lecture Notes in Computer Science*, pages 522–527. Springer Verlag, September 2009.
- [90] Nicola Leone, Georg Gottlob, Riccardo Rosati, Thomas Eiter, Wolfgang Faber, Michael Fink, Gianluigi Greco, Giovambattista Ianni, Edyta Kalka, Domenico Lembo, Maurizio Lenzerini, Vincenzino Lio, Bartosz Nowicki, Marco Ruzzi, Witold Staniszki, and Giorgio Terracina. The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In *Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005)*, pages 915–917, Baltimore, Maryland, USA, June 2005. ACM Press.
- [91] Nicola Leone, Simona Perri, and Francesco Scarcello. Improving asp instantiators by join-ordering methods. In *Logic Programming and Nonmonotonic Reasoning*, pages 280–294. Springer, 2001.
- [92] 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.
- [93] 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*, 7(3):499–562, July 2006.



- [94] Nicola Leone and Francesco Ricca. Answer set programming: A tour from the basics to advanced development tools and industrial applications. In Wolfgang Faber and Adrian Paschke, editors, *Reasoning Web. Web Logic Rules - 11th Int'l Summer School 2015, Berlin, Germany, Jul 31 - Aug 4, 2015, Tutorial Lectures*, volume 9203 of *Lecture Notes in Computer Science*, pages 308–326. Springer, 2015.
- [95] Senlin Liang, Paul Fodor, Hui Wan, and Michael Kifer. OpenRuleBench: An Analysis of the Performance of Rule Engines. In *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, pages 601–610. ACM, 2009.
- [96] Yuliya Lierler and Mirosław Truszczyński. Transition systems for model generators - A unifying approach. *Theory and Practice of Logic Programming*, 11(4-5):629–646, 2011.
- [97] Fangzhen Lin and Yuting Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.
- [98] Marco Manna, Francesco Ricca, and Giorgio Terracina. Consistent query answering via ASP from different perspectives: Theory and practice. *Theory and Practice of Logic Programming*, 13(2):277–252, 2013.
- [99] Marco Manna, Francesco Ricca, and Giorgio Terracina. Taming primary key violations to query large inconsistent data via ASP. *Theory and Practice of Logic Programming*, 15(4-5):696–710, 2015.
- [100] Marco Maratea, Francesco Ricca, Wolfgang Faber, and Nicola Leone. Look-back techniques and heuristics in dlv: Implementation, evaluation and comparison to qbf solvers. *Journal of Algorithms in Cognition, Informatics and Logics*, 63(1-3):70–89, 2008.
- [101] Victor W. Marek and Mirosław Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In Krzysztof R. Apt, V. Wiktor Marek, Mirosław Truszczyński, and David S. Warren, editors, *The Logic Programming Paradigm – A 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.
- [102] John McCarthy. Programs with Common Sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91. Her Majesty's Stationery Office, 1959.
- [103] John McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial intelligence*, 13(1):27–39, 1980.
- [104] Robert C Moore. Semantical considerations on nonmonotonic logic. *Artificial intelligence*, 25(1):75–94, 1985.
- [105] Ilkka Niemelä. Logic Programming with Stable Model Semantics as Constraint Programming Paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

- [106] 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, *Practical Aspects of Declarative Languages, Third International Symposium (PADL 2001)*, volume 1990 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2001.
- [107] Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Partial stable models for logic programs with aggregates. In *Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7)*, volume 2923 of *Lecture Notes in AI (LNAI)*, pages 207–219. Springer, 2004.
- [108] Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Well-founded and Stable Semantics of Logic Programs with Aggregates. *Theory and Practice of Logic Programming*, 7(3):301–353, 2007.
- [109] Nikolay Pelov and Mirosław Truszczyński. Semantics of disjunctive programs with monotone aggregates - an operator-based approach. In *Proceedings of the 10th International Workshop on Non-monotonic Reasoning (NMR 2004)*, Whistler, BC, Canada, pages 327–334, 2004.
- [110] Simona Perri, Francesco Ricca, and Marco Sirianni. Parallel Instantiation of ASP Programs: Techniques and Experiments. *Theory and Practice of Logic Programming*, 13(2):253–278, 2013.
- [111] Simona Perri, Francesco Scarcello, Gelsomina Catalano, and Nicola Leone. Enhancing DLV instantiator by backjumping techniques. *Annals of Mathematics and Artificial Intelligence*, 51(2–4):195–228, 2007.
- [112] Teodor C. Przymusiński. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.
- [113] Stanislaw P. Radziszowski. Small Ramsey Numbers. *The Electronic Journal of Combinatorics*, 1, 1994. Revision 9: July 15, 2002.
- [114] Lawrence Reeve and Hyoil Han. Survey of semantic annotation platforms. In *Proceedings of the 2005 ACM Symposium on Applied Computing, SAC '05*, pages 1634–1638, New York, NY, USA, 2005. ACM.
- [115] Raymond Reiter. A logic for default reasoning. *Artificial intelligence*, 13(1-2):81–132, 1980.
- [116] Francesco Ricca, Giovanni Grasso, Mario Alviano, Marco Manna, Vincenzino Lio, Salvatore Iiritano, and Nicola Leone. Team-building with answer set programming in the gioia-tauro seaport. *Theory and Practice of Logic Programming*. Cambridge University Press, 12(3):361–381, 2012.
- [117] Patrik Simons, Ilkka Niemelä, and Timo Soinen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.
- [118] Tran Cao Son and Enrico Pontelli. A Constructive Semantic Characterization of Aggregates in ASP. *Theory and Practice of Logic Programming*, 7:355–375, May 2007.

- [119] Tran Cao Son, Enrico Pontelli, and Islam Elkabani. On Logic Programming with Aggregates. Technical Report NMSU-CS-2005-006, New Mexico State University, 2005.
- [120] Terrance Swift and David Scott Warren. XSB: Extending Prolog with Tabled Logic Programming. *Theory and Practice of Logic Programming*, 12(1-2):157–187, 2012.
- [121] Tommi Syrjänen. Omega-restricted logic programs. In 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*, pages 267–279. Springer, 2001.
- [122] Tommi Syrjänen. Lparse 1.0 User’s Manual, 2002. <http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz>.
- [123] Juha Tiihonen, Timo Soinen, Ilkka Niemelä, and Reijo Sulonen. A practical tool for mass-customising configurable products. In *Proceedings of the 14th International Conference on Engineering Design (ICED’03)*, pages 1290–1299, 2003.
- [124] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1988.
- [125] Moshe Y. Vardi. Complexity of relational query languages. In *Proceedings of the 14th Symposium on Theory of Computation (STOC)*, pages 137–146, 1982.
- [126] Moshe Y. Vardi. On the Complexity of Bounded-Variable Queries. In *Proceedings PODS-95*, pages 266–276, 1995.
- [127] Marina De Vos, Doga Gizem Kisa, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Annotating Answer-Set Programs in LANA. *Theory and Practice of Logic Programming*, 12(4-5):619–637, 2012.
- [128] Jeffrey Ward and John S. Schlipf. Answer Set Programming with Clause Learning. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7)*, volume 2923 of *LNAI*, pages 302–313. Springer, January 2004.
- [129] Antonius Weinzierl. Blending lazy-grounding and CDNL search for answer-set solving. In Balduccini and Janhunen [7], pages 191–204.
- [130] Johan Wittocx and Marc Denecker. GIDL: A Grounder for  $FO^+$ . In *Proceedings of the Twelfth International Workshop on NonMonotonic Reasoning*, pages 189–198, 1998.