# Università della Calabria

Dipartimento di Matematica e Informatica


## Dottorato di Ricerca in Matematica e Informatica

XXXVI ciclo

---

Tesi di Dottorato

# Optimizing Evaluation of Logic Programs: Extended Compilation & Enhanced Rewritings

Settore Scientifico Disciplinare INF/01 – INFORMATICA


**Coordinatore:** Ch.mo Prof. Giorgio Terracina

**Supervisori:** Ch.mo Prof. Francesco Ricca

Prof. Carmine Dodaro

**Dottorando:** Dott. Giuseppe Mazzotta


*Questo lavoro di tesi è dedicato a te, angelo mio, che vegli su di me.*
*Mi hai sempre detto che desideravi vedermi tenere una lezione, presentare un lavoro ad una conferenza, ma questa vita crudele non te lo ha permesso.*
*Ora che sei lassù, spero che tu possa vedermi e che riuscirò a renderti orgoglioso di me, più di quanto già non lo fossi.*
*Ti prometto che darò tutto me stesso per far sì che il nostro sogno diventi realtà, ma tu ora riposa in pace insieme a nonno e gioite insieme a me per questo piccolo traguardo.*

# Contents

# Sommario

La programmazione logica offre un approccio dichiarativo per modellare problemi mediante l'utilizzo di regole logiche. Nell'ambito della programmazione logica, uno dei formalismi più conosciuti e utilizzati è l'Answer Set Programming (ASP) [16, 43]. Grazie alla disponibilità di sistemi efficienti per valutare programmi ASP [2, 38], questo formalismo si è dimostrato una valida alternativa, sia in contesti accademici e sia in contesti industriali, per la modellazione e la risoluzioni di problemi NP-hard.

Tuttavia, è noto che i sistemi ASP tradizionali soffrono di una limitazione intrinseca, nota come *grounding bottleneck*, la quale, in diversi casi di interesse pratico [20, 58], rende non fattibile la valutazione di alcuni programmi. Inoltre, nonostante si possano modellare problemi fino al secondo livello della gerarchia polinomiale (PH), ci sono diversi problemi interessanti al di là di queste classi di complessità che non possono essere modellati usando solo ASP standard.

Partendo da queste considerazioni, per affrontare il problema del *grounding bottleneck*, sono state proposte diverse soluzioni. Tra queste, le tecniche basate sulla compilazione si sono dimostrate molto efficaci nel risolvere istanze di problemi che in precedenza erano irraggiungibili per i sistemi standard [25, 26]. Inoltre, si sono rivelate efficaci anche per la valutazione di programmi positivi e programmi con negazione stratificata [28, 47]. Tuttavia, queste tecniche sono limitate a un frammento relativamente ristretto di programmi logici, e quindi, la domanda "è possibile compilare qualsiasi programma logico?", rimane un problema aperto.

Per aumentare l'espressività di ASP, invece, è stata proposta un'estensione chiamata Answer Set Programming with Quantifier (ASP(Q)) [8]. Questa estensione migliora significativamente le capacità di modellazione di ASP, consentendo di affrontare problemi nell'intera gerarchia polinomiale. Tuttavia, al fine di aumentare l'applicabilità di ASP(Q) in contesti reali, sono

richieste implementazioni sempre più efficienti.

Questa tesi presenta due contributi principali: (i) la progettazione di nuove tecniche basate sulla compilazione per la valutazione di programmi logici secondo le semantica dei modelli stabili e la semantica well-founded; e (ii) l'utilizzo di ottimizzazioni basate su riscritture ad-hoc per la valutazione efficiente di programmi ASP(Q).

Per quanto riguarda il primo contributo, si propogono i seguenti lavori: (a) una nuova tecnica per la compilazione di programmi ASP con aggregati in propagatori, sia eager che lazy, in base alla struttura del programma di input; (b) una tecnica di compilazione e un'architettura per la valutazione di programmi ASP che evita il loro grounding, implementata in un nuovo sistema chiamato ProASP. Questo sistema, estende la classe di programmi ASP che possono essere compilati in un propagatore oltre quella dei programmi che agiscono come vincoli; e (c) un approccio per valutare in modo efficiente programmi logici normali secondo la semantica well-founded. Il sistema proposto genera risolutori ad-hoc per programmi non ground (senza costanti) secondo la semantica well-founded che posso essere utilizzati come tool esterni per valutare più istanze dello stesso problema. Da varie analisi sperimentali condotte in questo lavoro di tesi è possibile vedere vantaggi significativi introdotti dalle tecniche proposte rispetto ai sistemi standard. In particolare, l'efficacia di queste tecniche viene maggiormente evidenziata nella valutazione di problemi dove la fase di grounding risulta essere molto pesante, mentre, in tutti gli altri casi, le tecniche proposte risultato essere competitive con i sistemi esistenti.

Per quanto riguarda il secondo contributo, si presenta PYQASP, una nuova implementazione di ASP(Q) che si basa sulla codifica in QBF proposta in QASP [5] ma incorpora: (a) una codifica ottimizzata basata sulla semantica well-founded per ottenere programmi equivalenti ma più concisi; (b) l'identificazione di un frammento sintattico di programmi ASP(Q) che può essere tradotto direttamente in formule QBF in Conjunctive Normal Form (CNF), eliminando la necessità di costose fasi di normalizzazione per tali programmi; e (c) una metodologia di algorithm selection, ispirata a quella impiegata nel sistema ME-ASP [53], adattata ad ASP(Q). Tutti questi fattori hanno permesso al sistema proposto di ottenere ottime prestazioni su vari domini, spingendo avanti lo stato dell'arte nella valutazione di programmi ASP(Q).

# Abstract

Logic programming offers a declarative approach to model problems using logical rules. A notable branch within logic programming is Answer Set Programming (ASP) [16, 43]. ASP is a widely recognized formalism used for Knowledge Representation and Reasoning, leveraging answer set semantics. With efficient systems available for evaluating ASP programs [38, 2], it is well-suited for modeling and solving, in both academic and industrial contexts, NP-hard problems.

However, it is known that standard ASP systems suffer from an intrinsic limitation, known as the *grounding bottleneck*, that makes the evaluation of ASP programs unfeasible in several cases of practical interest [20, 58]. Moreover, despite being able to model problems up to the second level of the polynomial hierarchy (PH), there are several interesting problems beyond these complexity classes that cannot be modeled using plain ASP.

Starting from these considerations, to tackle the *grounding bottleneck*, different approaches have been proposed in order to mitigate the effects of a such problem. Among possible solutions, compilation-based techniques have proven to be highly effective in solving instances that were previously out of reach for standard systems [26, 25]. Moreover, compilation-based techniques were revealed to be effective also for the evaluation of positive programs and programs with stratified negation [47, 28]. Nevertheless, these techniques are limited to a relatively smaller fragment of logic programs, leaving open the question of whether a compiler can be devised for a larger class of programs.

To increase the expressiveness of ASP, instead, an extension called Answer Set Programming with Quantifiers (ASP(Q)) [8] has been proposed. ASP(Q) significantly enhances the modeling capabilities of ASP, allowing the modeling of problems throughout the entire PH. However, efficient implementations are crucial to make ASP(Q) more applicable to real-world problems.

In this thesis, we propose two main contributions: (i) the design of novel compilation-based techniques for the evaluation of logic programs under answer set and well-founded semantics; and (ii) the enhancement of ASP(Q) evaluation by exploiting ad-hoc rewriting-based optimizations.

Regarding the first contribution, we introduce a novel technique for compiling ASP programs with aggregates into both eager and lazy propagators, according to the input program structure. We also propose a compilation technique and a grounding-less ASP-solving architecture, implemented in a new system named ProASP, which extends the class of ASP programs that can be compiled in a propagator beyond constraints. Furthermore, we introduce an approach for the efficient evaluation of normal logic programs under well-founded semantics. The proposed system generates ad-hoc solvers for non-ground programs (i.e., without constants) that are able to evaluate multiple problem instances. Experimental analysis, conducted to assess the impact of the proposed techniques, highlighted significant performance improvements with respect to standard systems. In particular, the effectiveness is remarkable in problems affected by grounding issues whereas, in problems not presenting such an issue, the proposed techniques obtained comparable performances with respect to state-of-the-art approaches.

Regarding the second contribution, we present PYQASP, a new implementation of ASP(Q) that builds on the concepts of QASP [5]. PYQASP incorporates (i) a more efficient encoding procedure and optimized encodings of ASP(Q) programs into QBF formulas, exploiting the well-founded semantics to achieve more concise yet equivalent programs; (ii) the identification of a natural syntactic fragment of ASP(Q) programs that can be directly translated into QBF in CNF, eliminating the need for costly normalization steps for such programs; and (iii) an algorithm selection methodology, inspired by the competition-winning solver `ME-ASP` [53], imported into the ASP(Q) setting. All these factors empowered PYQASP, which delivered outstanding performances across various problem domains and outperformed QASP, pushing forward the state-of-the-art in ASP(Q) solving.

# Introduction

Our lives have become increasingly intertwined with computer applications, which now automatically handle many everyday problems. Traditional software engineering predominantly adopts imperative programming paradigms, where problems are solved by applying algorithms (i.e., sequence of instructions). Intuitively, programmers need to encode the steps that the machine will execute to compute solutions for a given problem. It is important to note that developing efficient algorithms may require advanced expertise and pose challenges, especially for problems with high computational complexity. One possible alternative to imperative programming is declarative programming, which models problems by describing the features that characterize both the problem and its solutions. Among the declarative formalism, logic programming has proven to be a valid solution for modeling hard combinatorial programs. In logic programming, problems are encoded using logical rules and the solutions of such logic specifications correspond to the solutions of the modeled problem. Logic programs are commonly evaluated by general-purpose systems that implement a given semantics defining the meaning of logical specifications. One prominent branch of logic programming is Answer Set Programming (ASP). ASP [16, 43] is a well-known AI formalism for Knowledge Representation and Reasoning, based on answer set semantics. This formalism has found extensive applications in various AI sub-fields [33], including game theory [7], natural language processing and understanding [27, 56, 61], robotics [35], planning [63], and scheduling [30], among others [33].

The success of ASP can be attributed to the combination of two features: efficient implementations [40], and a highly expressive language capable of modeling hard problems [29] up to the second level of the polynomial hierarchy (PH). Regarding efficient implementations, modern ASP systems can effectively handle problems in both industrial and academic contexts. How-

ever, the emergence of new ASP applications that demand even better performance for hard-to-solve problems highlights the need for the development of more effective and faster ASP systems, making it a crucial and challenging research area [40]. Standard systems suffer of an intrinsic limitation, known as grounding bottleneck, due to the combinatorial blow-up during grounding phase. As a result, there exist classes of programs that cannot be evaluated by standard systems. Regarding the expressiveness of plain ASP language, there are several interesting problems beyond the second level of PH that cannot be modeled using plain ASP. For this reason, ASP has been recently extended to allow for declarative and modular modeling of problems with higher computational complexity, surpassing the limitations of traditional ASP modeling. This extension, known as Answer Set Programming with Quantifiers (ASP(Q)), significantly boosts the expressive capabilities of ASP, enabling users to model problems throughout the entire polynomial hierarchy. Following the introduction of the first solver for ASP(Q) called QASP [5] Despite being very effective in evaluating hard problems, QASP presents some weaknesses due to a highly general algorithm that lacks of specific optimizations that could be crucial in the evaluation of such computationally complex problems.

In this thesis, we propose two main contributions to the aforementioned contexts that are:

(i) The design of novel techniques aimed at generating (compiling), automatically, ad-hoc, and optimized implementations for logic programs, in order to tackle the grounding bottleneck problem (referred to as compilation-based ASP solving techniques).

(ii) The enhancement of ASP(Q) evaluation by exploiting ad-hoc rewriting-based optimizations.

These two contributions will be discussed separately in the first and second part of this thesis.

**Compilation-based approaches for the evaluation of logic programs.** Logic programs are often evaluated by means of general-purpose systems that implement a given semantics [12, 29]. The need for handling with the same algorithm any possible input, in some cases, makes it impossible to apply specific optimizations that would work only for a subclass of programs.

Thus, considerable speedups can be obtained by using ad-hoc evaluation procedures.

State-of-the-art ASP implementations are general-purpose systems that follow the well-known Ground&Solve approach [38, 2]. In this approach, a grounder module computes the propositional counterpart of the input program and then a solver module performs a propositional search for stable models (i.e., the answer sets of the input program). Ground&Solve ASP systems, such as CLINGO [38] and DLV [2], are intrinsically subject to the *grounding bottleneck*. Basically, the grounding step can be computationally expensive and even unfeasible in several cases of practical interest [20, 58].

Numerous efforts have been made to address the grounding bottleneck issue [40]. Recent works demonstrated that the grounding bottleneck can be partially overcome by compiling subprograms in external propagators, that simulate the presence of their ground counterpart during the solving phase [25]. However, these approaches can only compile subprograms acting as constraints without aggregates [26]. Whether it is possible to devise a compiler for rules "generating" answer sets or for advanced constructs like aggregates was left as an open question.

Compilation-based techniques have proven to be effective not only in mitigating the grounding bottleneck in ASP systems but also in the evaluation of Datalog programs. In particular, the system soufflé [47] was demonstrated to be very effective, especially for solving tasks connected with software development, but also other prototypical systems were revealed to be very promising [28]. However, none of the aforementioned systems support the well-founded semantics [65] when no restrictions are posed on the usage of negation.

Starting from these considerations, in this thesis we present the following contributions:

- **Compilation of Aggregates in ASP.** We introduce a novel technique for compiling ASP programs with aggregates into both *eager* and *lazy* propagators, according to the input program structure. We identify the syntactic conditions under which a program with aggregates can be compiled, thus extending the definition of compilable subprogram proposed by Cuteri et al. [25].

- **Compilation of Tight Normal Programs.** We propose a novel compilation technique and a grounding-less ASP-solving architecture,

implemented in a novel system called ProASP, that target tight normal programs [34]. The resulting approach extends the class of ASP programs that can be compiled in a propagator *beyond constraints*.

- **Compilation under Well-founded Semantics.** We introduce an approach for the efficient evaluation of normal logic programs under well-founded semantics. Basically, the proposed system is able to generate ad-hoc solvers for non-ground programs (i.e., constants free) under the well-founded semantics that can be used as external tools for computing the well-founded model for different problem instances.

**Enhancing ASP(Q) evaluation.** Despite being very effective in modeling and solving problems in NP [41] (i.e., the first level of the Polynomial Hierarchy (PH)), ASP is less practical when one has to approach problems beyond NP. The existing programming techniques, such as *saturation* [31, 29], allow for encoding problems that belong to the second level of the PH, but the expressive power of ASP does not span the entire PH. In order to overcome this limitation, Answer Set Programming with Quantifiers (ASP(Q)) has been proposed. ASP(Q) expands ASP with quantifiers over answer sets of ASP programs and allows the programmer to use the standard and natural programming methodology, known as generate-define-test [52], to encode also problems beyond NP. The first solver for ASP(Q), named qasp [5], is based on a translation to Quantified Boolean Formulae (QBF) with the aim of exploiting the well-developed and mature QBF-solving technology [60]. However, the implementation of the QBF encoding employed in qasp is very general and might produce formulas that are hard to evaluate for existing QBF solvers because of the large number of symbols and sub-clauses. Moreover, Amendola et al. [5] observed that the implementation of the translation procedure could – in some specific cases – be so memory-hungry to prevent the production of the QBF formula even when a considerable amount of memory is available.

In this thesis, we present pyqasp, a new implementation of ASP(Q) that expands on the concepts of qasp. pyqasp introduces a more efficient encoding procedure and optimized encodings of ASP(Q) programs into QBF. The key aspects of this contribution are as follows:

- **Optimization based on Well-founded semantics**. We leverage the well-founded model of logic programs to achieve more concise yet equivalent programs. Consequently, the proposed system generates more

compact encodings in QBF, reducing the number of clauses and average clause length.

- **Direct encoding in Conjunctive Normal Form (CNF)**. We define a natural syntactic fragment of ASP(Q) programs that can be directly translated into QBF in CNF. This approach eliminates the need for costly normalization steps, enhancing efficiency.

- **Automatic back-end selection**. We incorporate an algorithm selection methodology, inspired by the competition-winning solver `ME-ASP` [53], into the ASP(Q) setting. This empowers PYQASP to deliver consistent performance across various problem domains.

**Published papers.** The contributions described in the thesis have been subject to scientific publications.

- Giuseppe Mazzotta, Francesco Ricca, and Carmine Dodaro. Compilation of aggregates in ASP systems. In AAAI, pages 5834–5841. AAAI Press, 2022. https://doi.org/10.1609/aaai.v36i5.20527. Contributing to Chapter 3.

- Carmine Dodaro, Giuseppe Mazzotta, and Francesco Ricca. Compilation of tight ASP programs. In 26th European Conference on Artificial Intelligence. https://ebooks.iospress.nl/doi/10.3233/FAIA230316. Contributing to Chapter 4.

- Andrea Cuteri, Giuseppe Mazzotta, and Francesco Ricca. Compilation-based techniques for evaluating normal logic programs under the well-founded semantics. Proceedings of the 38th Italian Conference on Computational Logic, volume 3428 of CEUR Workshop Proceedings. CEUR-WS.org, 2023. Contributing to Chapter 5.

- Wolfgang Faber, Giuseppe Mazzotta and Francesco Ricca. An efficient solver for ASP(Q). Theory and Practice of Logic Programming p. 1–17 (2023). https://doi.org/10.1017/S1471068423000121. Contributing to Chapters 7- 9.

Among these scientific publications, the paper "Compilation of aggregates in ASP systems" obtained the "*Outstanding student paper award honorable mention of AAAI2022*".

**Structure of the thesis.** The remaining part of this thesis is organized as follows:

- Chapter 1 introduces the basic notions of logic programming that are needed to better understand the work proposed in this thesis.
- Part I is focused on the extension of compilation-based approaches that target different classes of programs. In particular:
  - Chapter 2 describes both the standard evaluation of ASP programs and compilation-based ASP solving.
  - Chapter 3 presents a novel technique for compiling aggregates in standard ASP systems.
  - Chapter 4 presents a novel grounding-less architecture for tight normal programs based on well-mature SAT-solving techniques.
  - Chapter 5 proposes a compilation-based approach aimed at generating ad-hoc solvers for evaluating normal logic programs under well-founded semantics.
- Part II is focused on ASP(Q) and we present a novel system named PYQASP that resorts the encoding in QBF implemented in QASP system and implements ad-hoc rewriting-based simplifications aimed at producing more affordable QBF formulas. More precisely:
  - Chapter 7 introduces a more efficient encoding that exploits the well-founded semantics of normal logic programs.
  - Chapter 8 presents a syntactic class of ASP(Q) programs featuring a direct encoding in QBF formulas in Conjunctive Normal Form.
  - Chapter 9 describes an empirical evaluation aimed at assessing the performance of PYQASP on different benchmarks proposed in the literature.
- Related Work describes closely related works to the contributions proposed in this thesis. In particular, we overview possible solutions that tried to mitigate the effects of the grounding bottleneck problem in standard ASP solving but also existing formalisms and implementations for modeling problems beyond the NP class.
- Conclusion sums up the main contributions of this thesis and proposes future directions on this line of research.

# Chapter 1

# Preliminaries on Logic Programming

This chapter provides an introduction to fundamental concepts related to logic programs, answer sets, and well-founded semantics. We will focus on both the syntax and semantics of logic programs, presenting essential notations that will remain consistent throughout this thesis to present the main contributions.

## 1.1 Syntax

In logic programming variables are strings starting with uppercase letters and constants are non-negative integers or strings starting with lowercase letters. A *term* is either a variable or a constant. A *standard atom* is an expression of the form $p(t_1, \ldots, t_n)$, where $p$ is a *predicate* of arity $n$ and $t_1, \ldots, t_n$ are terms. A standard atom $p(t_1, \ldots, t_n)$ is ground if $t_1, \ldots, t_n$ are constants. A standard literal is an atom $p$ or its negation $\sim p$, where $\sim$ denotes negation as failure. A *ground set* is a set of pairs of the form $\langle consts : conj \rangle$, where *consts* is a list of constants and *conj* is a conjunction of ground standard literals. A *symbolic set* is a set specified syntactically as $\{Vars : Conj\}$, where *Vars* is a non-empty list of variables, and *Conj* is a non-empty conjunction of standard literals. An *aggregate function* is of the form $f(S)$, where $S$ is a symbolic set, and $f \in \{\#count, \#sum\}$ is an *aggregate function symbol*. An *aggregate atom* is of the form $f(S) \prec T$, where $f(S)$ is an aggregate function, $\prec \in \{<, \leq, >, \geq, =\}$ is a comparison operator, and $T$ is a term called guard.

An aggregate atom $f(S) \prec T$ is ground if $T$ is a constant and $S$ is a ground set. An *atom* is either a standard atom or an aggregate atom. A *literal* is an atom or its negation. The complement of $l$ is denoted by $\bar{l}$, and it is $\sim a$, if $l = a$, or $a$ if $l = \sim a$. This notation is extended also to sets of literals. For a set of literals $L$, $L^+$, and $L^-$ denote the set of positive and negative literals appearing in $L$, respectively, and $\neg L$ denotes the set of literals $\{\bar{l} \mid l \in L\}$. A program $\pi$ is a set of rules:

$$h \leftarrow b_1, \ldots, b_k, \sim b_{k+1}, \ldots, \sim b_m$$

with $m \geq k \geq 0$, where $h$ is a standard atom referred to as the *head* and it can absent, whereas $b_1, \ldots, b_k, \sim b_{k+1}, \ldots, \sim b_m$ is the *body*, $b_1, \ldots, b_k$ are atoms, and $b_{k+1}, \ldots, b_m$ are standard atoms. A *constraint* is a rule where $h$ is absent, and a *fact* is a rule where $m = 0$. Moreover, for a rule $r$, $H_r$ and $B_r$ are two sets containing the head and the body of a rule $r$, respectively, $B_r^+$ and $B_r^-$ are two sets containing the positive and the negative body of $r$, respectively, $B_r^a$ denotes the set of aggregate atoms appearing in $B_r$, and $Conj^+(B_r^a)$ and $Conj^-(B_r^a)$ denotes the set of positive and negative standard literals appearing in the aggregate atoms of the body, respectively. Given an expression (atom, literal, rule, program, etc.) $\epsilon$ we define the following sets:

- $\mathcal{V}(\epsilon)$ the list of variables appearing in $\epsilon$;
- $\mathcal{T}(\epsilon)$ the list of terms appearing in $\epsilon$;
- $\mathcal{P}(\epsilon)$ the set of predicates appearing in $\epsilon$;
- $\mathcal{H}(\epsilon)$ the set atoms appearing in the head of some rule in $\epsilon$.

Given a program $\pi$, $U_\pi$ is the *Herbrand Universe*, and represents the set of all constants appearing in $\pi$; whereas $B_\pi$ is the *Herbrand Base*, and represents the set of all possible ground standard atoms that can be built using predicates in $\pi$ and constants in $U_\pi$. $\mathcal{B}$ denotes $B_\pi \cup \neg B_\pi$. Given an expression (atom, literal, rule, program, etc.) $\epsilon$ and the Herbrand Universe $U_\pi$, we define $ground(\epsilon)$ as the set of all instantiations of $\epsilon$ built by assigning variables to constant in $U_\pi$. The *dependency graph*, denoted by $G_\pi$, is the directed labeled graph $\langle V, E \rangle$, where $V$ is the set of atoms appearing in $ground(\pi)$ and $E$ contains a positive (resp. negative) edge $(p_1, p_2)$ if there exists a rule $r \in ground(\pi)$ such that $p_2$ appears in $H_r$ and $p_1$ appears in $B_r^+$ (resp. $B_r^-$). The *positive dependency graph*, denoted by $PG_\pi$ is the directed graph $\langle V, E' \rangle$

obtained from $G_\pi$ by filtering out negative edges in $E$. A *Datalog* program (or positive program) is a program containing no negative literals. A *stratified* program $\pi$ is a program such that $G_\pi$ does not contain cycles involving negative edges. A *tight* program $\pi$, is a program such that $PG_\pi$ is acyclic.

## 1.2 Answer Set Semantics

An interpretation $I \subseteq \mathcal{B}$ is a set of literals. A literal $l$ is true w.r.t. $I$ if $l \in I$, $l$ is false w.r.t. $I$ if $\bar{l} \in I$, otherwise it is undefined. In the following, for a program $\pi$, $I^U_\pi$ denotes all literals in $\mathcal{B}$ that are undefined. A conjunction *conj* of literals is true w.r.t $I$ if all literals in *conj* are true w.r.t. $I$, and false if there is at least one literal in *conj* that is false w.r.t. $I$. Let $I(S)$ denote the multiset $[t_1 \mid (t_1, \ldots, t_n) : conj \in S, ground(conj)$ is true w.r.t. $I]$. The evaluation $I(f(S))$ of an aggregate function $f(S)$ w.r.t. $I$ is the result of the application of $f$ on $I(S)$. An interpretation $I$ is *total* if all literals in $\mathcal{B}$ are either true or false, otherwise $I$ is *partial*. An interpretation $I$ is *consistent* if for each literal $l \in I$, $\bar{l} \notin I$, otherwise it is *inconsistent*. A total and consistent interpretation $I$ is a *model* for $\pi$ if for each rule $r \in ground(\pi)$, the head of $r$ is true whenever the body of $r$ is true. A total model $M$ is a (subset-)minimal model if does not exist a total model $M_1$ such that $M_1^+ \subset M^+$. Given a program $\pi$ and an interpretation $I$, the *FLP-reduct* [36] of $\pi$, denoted by $\pi^I$, is defined as the set of rules obtained from $\pi$ by deleting those rules whose body is false w.r.t $I$. A model $I$ is an *answer set* (stable model) of a program $\pi$, if $I$ is a minimal model of $\pi^I$. Given a program $\pi$, the set of answer sets of $\pi$ is denoted by $AS(\pi)$. A program $\pi$ is coherent if it admits at least one answer set (i.e. $AS(\pi) \neq \emptyset$), otherwise, $\pi$ is incoherent.

**Example 1.** *Let $\pi$ be the following program*

$$
\begin{array}{llll}
r_1: & a(X) \leftarrow & d(X), {\sim}b(X) & \qquad r_4: \quad d(1) \leftarrow \\
r_2: & b(X) \leftarrow & d(X), {\sim}a(X) & \qquad r_5: \quad d(2) \leftarrow \\
r_3: & \leftarrow & \#count\{X: \ a(X)\} > 1 &
\end{array}
$$

*The ground instantiation of $\pi$ (i.e. $ground(\pi)$) is the program:*

$$
\begin{array}{llll}
r_1^1: & a(1) \leftarrow & d(1), {\sim}b(1) & \qquad\qquad r_4: \quad d(1) \leftarrow \\
r_1^2: & a(2) \leftarrow & d(2), {\sim}b(2) & \qquad\qquad r_5: \quad d(2) \leftarrow \\
r_2^1: & b(1) \leftarrow & d(1), {\sim}a(1) & \\
r_2^2: & b(2) \leftarrow & d(2), {\sim}a(2) & \\
r_3^1: & \leftarrow & \#count\{\langle 1: \ a(1)\rangle, \langle 2: \ a(2)\rangle\} > 1 &
\end{array}
$$

*Let* $I = \{a(1), b(2), {\sim}b(1), {\sim}a(2), d(1), d(2)\}$*, the FLP-reduct* $\pi^I$ *is:*

$$
\begin{array}{llll}
r_1^1: & a(1) \leftarrow & d(1), {\sim}b(1) & \qquad r_4: \quad d(1) \leftarrow \\
r_2^1: & b(2) \leftarrow & d(2), {\sim}a(2) & \qquad r_5: \quad d(2) \leftarrow
\end{array}
$$

*Since* $I$ *is a minimal model for* $\pi^I$ *then* $I$ *is an answer set of* $\pi$*.*

## 1.3 Well-founded Semantics

Let $\pi$ be a program without aggregates and $I$ be an interpretation, a set of standard atoms $U \subseteq \mathcal{B}$ is an unfounded set of $\pi$ w.r.t. $I$ if for each rule $r \in ground(\pi)$ such that $H_r \subseteq U$, $B_r$ is false w.r.t. $I$ or $B_r \cap U \neq \emptyset$. Intuitively, all rules defining atoms in $U$ have a false body w.r.t. $I$ or depend on atoms in $U$. The greatest unfounded set of $\pi$ w.r.t. $I$, $U(I, \pi)$, is defined as the union of all unfounded sets of $\pi$ w.r.t. $I$. Let $T(I, \pi)$ be the set of standard atoms $a \in \mathcal{B}$ such that there exists a rule $r \in ground(\pi)$ having $a$ in the head and a true body w.r.t. $I$, the well-founded operator, $\mathcal{W}_\pi(I)$, is defined as $T(I, \pi) \cup \neg U(I, \pi)$. Let $I_0 = \emptyset$, $I_{\alpha+1} = \mathcal{W}_\pi(I_\alpha)$, with $\alpha \geq 0$, the (partial) well-founded model is defined as $\mathcal{W} = I_\beta$ where $\beta \geq 0$ is the smallest ordinal such that $I_\beta = I_{\beta+1}$.

**Example 2.** *Let* $\pi$ *be the following program*

$$
\begin{array}{llll}
r_1: & a(X) \leftarrow & d1(X), {\sim}b(X) & \qquad r_4: \quad d1(1) \leftarrow \\
r_2: & b(X) \leftarrow & d2(X), {\sim}a(X) & \qquad r_5: \quad d2(1) \leftarrow \\
r_3: & c(X) \leftarrow & a(X), b(X) & \qquad r_6: \quad d1(2) \leftarrow \\
 & & & \qquad r_7: \quad d2(3) \leftarrow
\end{array}
$$

*The ground instantiation of* $\pi$ *(i.e.* $ground(\pi)$*) is the program:*

$$
\begin{array}{llll}
r_1^1: & a(1) \leftarrow & d1(1), \sim b(1) & r_4: \quad d1(1) \leftarrow \\
r_1^2: & a(2) \leftarrow & d1(2), \sim b(2) & r_5: \quad d2(1) \leftarrow \\
r_1^3: & a(3) \leftarrow & d1(3), \sim b(3) & r_6: \quad d1(2) \leftarrow \\
r_2^1: & b(1) \leftarrow & d2(1), \sim a(1) & r_7: \quad d2(3) \leftarrow \\
r_2^2: & b(2) \leftarrow & d2(2), \sim a(2) & \\
r_2^3: & b(3) \leftarrow & d2(3), \sim a(3) & \\
r_3^1: & c(1) \leftarrow & a(1), b(1) & \\
r_3^2: & c(2) \leftarrow & a(2), b(2) & \\
r_3^3: & c(3) \leftarrow & a(3), b(3) &
\end{array}
$$

*The well-founded model is obtained as follows:*

$$
\begin{array}{rl}
I_0 = & \emptyset \\
T(I_0, \pi) = & \{\boldsymbol{d1(1)}, \boldsymbol{d1(2)}, \boldsymbol{d2(1)}, \boldsymbol{d2(3)}\} \\
U(I_0, \pi) = & \{\boldsymbol{d1(3)}, \boldsymbol{d2(2)}, \boldsymbol{a(3)}, \boldsymbol{b(2)}, \boldsymbol{c(2)}, \boldsymbol{c(3)}\} \\
I_1 = & T(I_0, \pi) \cup \neg U(I_0, \pi) \\
T(I_1, \pi) = & \{d1(1), d1(2), d2(1), d2(3), \boldsymbol{a(2)}, \boldsymbol{b(3)}\} \\
U(I_1, \pi) = & U(I_0, \pi) \\
I_2 = & T(I_1, \pi) \cup \neg U(I_1, \pi) \\
T(I_2, \pi) = & \{d1(1), d1(2), d2(1), d2(3), a(2), b(3)\} \\
U(I_2, \pi) = & U(I_1, \pi) \\
\mathcal{W} = I_3 = & T(I_2, \pi) \cup \neg U(I_2, \pi)
\end{array}
$$

Basically, $\mathcal{W}_\pi$ is the least fixed point $\mathcal{W}_P$ [65]. The well-founded semantics is a robust semantics, since it guarantees the existence of a unique model for each logic program. Moreover, this semantics features interesting properties such that (i) for positive programs and programs with stratified negation, the well-founded model is a total interpretation and coincides with the unique stable model; and (ii) the well-founded model is true in every stable model.

# Part I

# Compilation-based Techniques for the efficient evaluation of normal logic programs

# Chapter 2

# Evaluation of logic programs

In this chapter, we present two strategies for evaluating ASP programs. In particular we will:

(i) describe the standard evaluation of ASP programs that follows the Ground&Solve approach; and

(ii) present compilation-based approaches for ASP proposed so far in the literature.

Additionally, we will introduce the pseudo-code convention, which will be consistently utilized to present compilation-based techniques proposed in this part of the thesis.

## 2.1 Standard evaluation under Answer Set Semantics

Traditional ASP solving follows the well-know, Ground&Solve approach. This approach is built on two components, namely grounder, and solver. The grounder takes as input a program $\pi$ and produces $ground(\pi)$, which is later on processed by the solver to produce answer sets. The solver employs an algorithm (reported as Algorithm 1) that extends the CDCL algorithm [62] with *propagators* specific to ASP [48]. The CDCL is based on a *choose-propagate-learn* pattern, and the idea is to build an answer set step-by-step starting from an empty interpretation $I$. At each step, a literal is heuristically selected and added to $I$ (*choice*). Then, *eager propagators* are used to

**Algorithm 1** CDCL for ASP solving

```
   Input : An ASP Program Π
1  begin
2      M := ∅
3      Loop
4          EagerPropagation(Π, M);
5          if M is inconsistent then
6              Π := Π ∪ Learning(Π, M);
7              M := RestoreConsistency(Π, M);
8              if M is inconstent then return ⊥
9          if M is total then
10             C := LazyPropagation(Π, M);
11             if C ≠ ∅ then Π := Π ∪ C else return M
12         else M := PickBranchingLiteral(Π)
```

extend $I$ with the deterministic consequences of this choice and their *reason*, i.e., literals in $I$ leading to the propagation. In case the propagation leads to an inconsistency in the interpretation $I$ (i.e. $l, \bar{l} \in I$ for some $l \in \mathcal{B}$), the algorithm *learns* a new constraint using the reason of each propagated literal, undoes the choices leading to the inconsistency, and restores the consistency of $I$. This process is repeated until $I$ is a candidate answer set or the consistency of $I$ cannot be restored, showing that no answer sets can be found. If $I$ is a candidate answer set then it is analyzed by *lazy propagators*, which may perform additional checks on $I$ (for example some solvers use them for stability checks when the program is not head cycle free [13, 1]). If $I$ is consistent under those checks, then the algorithm terminates, otherwise new constraints are added to $\mathit{ground}(\pi)$ and the algorithm restarts.

Example 3 describes a possible branch in the execution of Algorithm 1.

**Example 3.** *Let $\mathit{ground}(\pi)$ be the program from Example 1.*

$$
\begin{array}{llll}
r_1^1: & a(1) \leftarrow & d(1), {\sim}b(1) & \qquad\qquad r_4: \quad d(1) \leftarrow \\
r_1^2: & a(2) \leftarrow & d(2), {\sim}b(2) & \qquad\qquad r_5: \quad d(2) \leftarrow \\
r_2^1: & b(1) \leftarrow & d(1), {\sim}a(1) & \\
r_2^2: & b(2) \leftarrow & d(2), {\sim}a(2) & \\
r_3^1: & \leftarrow & \#count\{\langle 1: \ a(1)\rangle, \langle 2: \ a(2)\rangle\} > 1 &
\end{array}
$$

*Algorithm 1 starts initializing $M$ as an empty set. Then, EagerPropagation($\pi$, $M$) is executed. By propagating rule $r_4$ and $r_5$, $d(1)$ and $d(2)$ are derived as true (i.e., they are added to $M$). At this point, $M = \{d(1),\ d(2)\}$ is consistent and not total, and so the solver chooses a branching literal $l$ that is added to $M$, say $\sim b(1)$. Then, EagerPropagation($\pi$, $M$) performs the following propagations:*

- *$a(1)$ is added to $M$ due to $r_1^1$, since both $d(1)$ and $\sim b(1)$ are true (i.e., the reason of the propagation).*
- *$\sim a(2)$ is added to $M$ due to rule $r_3^1$, since $a(1)$ is true.*
- *$b(2)$ is added to $M$ due to rule $r_2^2$, since both $d(2)$ and $\sim a(2)$ are true.*

*Thus, $M = \{d(1)\, d(2),\ \sim b(1),\ a(1),\ \sim a(2),\ b(2)\}$ is a both total and consistent. In this example, LazyPropagation($\pi$, $M$) has no effect, therefore the algorithm terminates returning $M$.*

## 2.2 Compilation-based techniques

The main idea behind compilation-based approaches is to compile the rules of a program $\pi$ into a collection of specialized *propagators* that are able to simulate rules' inferences without grounding them in advance. Such propagators will be further injected into the solver either as lazy or eager propagators. Thus, the standard approach is extended by a third component, namely *compiler*, which translates (possibly non-ground) rules into ad-hoc propagators. More precisely, given a program $P$, it is split into two programs $P'$ and $Q$. The compiler takes as input $Q$ and produces a set of propagators that are nested into the solver module. The grounder module takes as input the program $P'$ and produces the program $\Pi = ground(P')$ that is further given as input to the solver module. At this point, the solver is able to compute the answer sets of $P$ by means of standard propagators of the program $P'$ and ad-hoc propagators that mimic inferences implied by rules in $Q$. Compilation-based techniques in the literature [25, 26] have been proposed for a restricted class of programs. In particular, Cuteri et al. [25] proposed the compilation of a subprogram, under certain syntactic restrictions, into lazy propagators. To this end, they provided the definition of a compilable subprogram $Q \subseteq P$ with respect to the program $P$. Intuitively, $Q$ is compilable w.r.t $P$ if $Q$ is a stratified program and does not define any predicates appearing in $P \setminus Q$.

More precisely, $Q$ can be compiled if the following conditions hold: (i) $Q$ is a stratified program and (ii) $\mathcal{P}(\mathcal{H}(Q)) \cap \mathcal{P}(P \setminus Q) = \emptyset$.

Furthermore, Cuteri et al. [26] proposed the compilation of constraints without aggregates into eager propagators.

Existing compilation-based approaches were revealed to be very effective in tackling the grounding bottleneck problem, but the effectiveness of compilation-based techniques relies heavily on the kind of propagators that will be generated, as noted in [24].

In order to introduce the conventions used for describing the compilation-based techniques proposed in this thesis, we report pseudo-code algorithms that follow the baseline and syntactic conventions described in [26] with some minor differences. In particular, the code enclosed between $\ll\gg$ is printed by the compiler as it is, but the code enclosed in $[\![]\!]_\ell$, is substituted with its run-time value before being printed. In order to better understand the syntactic convention, Example 4 reports a generic algorithm (Algorithm 2) playing the role of the compiler and the code that will be generated by it (Algorithm 3).

**Example 4.** *Let Algorithm 2 be the pseudo-code of the compiler module then Algorithm 3 reports the pseudo-code obtained by executing Algorithm 2. More precisely, Algorithm 2 starts by executing lines 2 and 3 which are printed as they are in Algorithm 3. Then, Algorithm 2 iterates over all values of* $j \in \{1, 2, 3\}$*, and for* $j = 1$*,* $1$ *is substituted in line 6 and generates line 4 of Algorithm 3. For* $j \geq 1$ *the value of* $j$ *is substituted in line 8 and lines 5 and 6 of Algorithm 3 are generated.*

**Algorithm 2** Compiler pseudo code example

```
1  begin
2      ≪def procedure(n,a,b)≫
3      ≪  forall i ∈ {1,...,n} do≫
4      for all j ∈ {1,2,3} do
5          if j < 2 then
6              ≪    a[ [[j]]_ℓ ] = b[ [[j]]_ℓ ] + i≫
7          else
8              ≪    b[ [[j]]_ℓ ] = a[ [[j]]_ℓ ] + i≫
9      ≪  done≫
10     ≪end def≫
```

**Algorithm 3** Generated pseudo code example

```
1 begin
2   def procedure(n,a,b)
3     forall  i ∈ {1,...,n} do
4       a[1] = b[1] + i
5       b[2] = a[2] + i
6       b[3] = a[3] + i
7     done
8   end def
```

# Chapter 3

# Compilation of Aggregates in ASP

In this chapter, we propose a compilation-based approach for the efficient evaluation of aggregates in ASP systems. We implement it on top of a state-of-the-art ASP system WASP, and evaluate the performance on publicly-available benchmarks. Obtained results show that our approach is very effective on grounding-intensive ASP programs and obtained comparable performance with respect to state-of-the-art systems on problems that are not affected by grounding issues.

## 3.1 Conditions for Splitting and Compiling Programs

In this section, we provide the conditions for a sub-program to be compilable under our compilation-based approach using the concept of non-ground dependency graph of an ASP program. We assume without loss of generality that each predicate appears always with the same ariety.

**Definition 1.** *Given an ASP program $\pi$, the non-ground dependency graph of $\pi$, denoted $DG_\pi$, is a labeled graph $(V, E)$ where $V$ is the set of predicate appearing in some head of $\pi$, and $E$ contains (i) $(v_1, v_2, +)$, if there is $r \in \pi \mid v_1 \in \mathcal{P}(B_r^+) \cup \mathcal{P}(Conj^+(B_r^a)), v_2 \in \mathcal{P}(H_r)$; (ii) $(v_1, v_2, -)$, if there is $r \in \pi \mid v_1 \in \mathcal{P}(B_r^-) \cup \mathcal{P}(Conj^-(B_r^a)), v_2 \in \mathcal{P}(H_r)$.*

**Definition 2.** *Given an ASP program $\pi$, an ASP sub-program $\lambda \subseteq \pi$ is* compilable w.r.t. *$\pi$ if the following conditions hold:*

*(i)* *$DG_\lambda$ has no loop in it;*

*(ii)* *for each $p \in \mathcal{P}(\mathcal{H}(\lambda))$, $p \notin \mathcal{P}(\pi \setminus \lambda)$;*

*(iii)* *given two rules $r_1, r_2 \in \lambda, r_1 \neq r_2$, $\mathcal{P}(H_{r_1}) \cap \mathcal{P}(H_{r_2}) = \emptyset$; and*

*(iv)* *for each $r \in \lambda$, $|B_r^a| \leq 1$.*

Intuitively, $\lambda \subseteq \pi$ is compilable w.r.t to $\pi$ if (i) it is both tight and stratified according to $DG_\pi$; (ii) predicates defined in $\lambda$ do not appear in the remaining program $\pi \setminus \lambda$; (iii) no two rules in $\lambda$ share the same predicate in the head; and (iv) each rule of $\lambda$ has at most one aggregate atom in the body.

## 3.2 Normalization of the Input Program

In the following we describe the main preprocessing steps that transform the sub-program to compile. First of all, the sub-program $\lambda$ is analyzed in order to be split in two sub-programs, namely $\lambda_l$ and $\lambda_e$. This analysis consists of labeling predicates that appear in some constraints and recursively labeling predicates that appears in the body of a rule whose head predicate has been already labeled. In this way, the rules whose head predicate has not been labeled could be treated in a lazy way ($\lambda_l$); other rules are in $\lambda_e$. For $\lambda_e$, we perform a rewriting to obtain a normalized form with rules of a specific format, and have a uniform treatment of all the rules to compile.

**Step 1.** Each rule $r \in \lambda_e$, with $|B_r^a| = 1$, $f(\{ Vars : Conj\}) \prec T \in B_r^a$, and $\prec \in \{<, \leq, >, \geq\}$, is replaced by the following rules:

1. $as_r(Vars, \rho) \leftarrow Conj$;
2. $bd_r(\rho, T) \leftarrow B_r \setminus B_r^a$;
3. $aggr_r(\rho, T) \leftarrow dm_r(\rho, T)$, $f(\{ Vars : as_r(Vars, \rho)\}) \geq G$, where $G = T$ if $\prec \in \{\geq, <\}$, and $G = T + 1$ if $\prec \in \{\leq, >\}$;
4. $h \leftarrow bd_r(\rho, T), aggr_r(\rho, T)$ if $\prec \in \{>, \geq\}$ and

$h \leftarrow bd_r(\rho, T), {\sim}aggr_r(\rho, T)$ if $\prec \in \{<, \leq\}$;

where $\rho$ is the list of variables appearing in both $\mathcal{V}(Conj)$ and $\mathcal{V}(B_r \setminus B_r^a)$.

**Example 5.** *Let assume $r$ to be the following rule:*

$$\begin{aligned} a(X, W) \leftarrow \quad & b(X, Y), c(Y, W), \\ & \#sum\{Z : d(X, Z), {\sim}e(Z)\} \geq W. \end{aligned}$$

*Then, $r$ is replaced by the following rules:*

$$\begin{array}{lll} r_1: & as_r(Z, X) & \leftarrow d(X, Z), {\sim}e(Z) \\ r_2: & bd_r(X, W) & \leftarrow b(X, Y), c(Y, W) \\ r_3: & aggr_r(X, W) & \leftarrow dm_r(X, W), \\ & & \#sum\{Z : as_r(Z, X)\} \geq W \\ r_4: & a(X, W) & \leftarrow bd_r(X, W), aggr_r(X, W) \end{array}$$

**Step 2.** Each rule $r \in \lambda_e$, with $|B_r^a| = 1$, $f(\{Vars : Conj\}) \prec T \in B_r^a$, and $\prec \in \{=\}$, is replaced by the rules *1*, *2*, and by the following rules:

5. $aggr_r^1(\rho, T) \leftarrow dm_r(\rho, T),\ f(\{Vars : as_r(Vars, \rho)\}) \geq T$;
6. $aggr_r^2(\rho, T) \leftarrow dm_r(\rho, T),\ f(\{Vars : as_r(Vars, \rho)\}) \geq T + 1$;
7. $h \leftarrow bd_r(\rho, T), aggr_r^1(\rho, T), {\sim}aggr_r^2(\rho, T)$.

**Step 3.** Each rule $r \in \lambda_e$, with $|B_r^a| = 0$, is replaced by the following rules:

8. $h \leftarrow aux_r(\mathcal{V}(B_r^+))$;
9. $\leftarrow aux_r(\mathcal{V}(B_r^+)), \overline{b_i} \qquad \forall i \in \{1, \ldots, m\}$;
10. $\leftarrow B_r, {\sim}aux_r(\mathcal{V}(B_r^+))$.

This step is applied also to rules from steps 1 and 2.

**Example 6.** *Let assume $r$ to be the following rule:*

$$a(Z, X) \leftarrow d(X, Z), {\sim}e(Z).$$

*Then, $r$ is replaced by the following rules:*

$$\begin{array}{lll} r_8: & a(Z, X) & \leftarrow aux_r(X, Z) \\ r_9': & & \leftarrow aux_r(X, Z), {\sim}d(X, Z) \\ r_9'': & & \leftarrow aux_r(X, Z), e(Z) \\ r_{10}: & & \leftarrow d(X, Z), {\sim}e(Z), {\sim}aux_r(X, Z). \end{array}$$

Intuitively, the normalization ensures that aggregate functions are applied to set of atoms, and rules are subject to a form of completion [23]. After applying the normalization step the program contains only rules of the form:

$$h \leftarrow b \tag{3.1}$$

$$h \leftarrow d, \#sum(\{Vars : b\}) \geq g \tag{3.2}$$

$$h \leftarrow d, \#count(\{Vars : b\}) \geq g \tag{3.3}$$

$$\leftarrow c_1, ..., c_n \tag{3.4}$$

Thus, the compiler will only have to produce propagators simulating the above-mentioned four rule types.

Finally, it is important to emphasize that atoms of the form $dm_r(\cdot)$ and $aux_r(\cdot)$ do not appear in the head of any rule in the program, and thus the ASP semantics would make them false in all stable models. Therefore, in our approach, they are treated as *external atoms* [38], whose instantiation and truth values are defined at running time in the propagator when the base $B_{\lambda_e}$ is determined.

## 3.3 Compilation

In this section we present our strategy to compile a subprogram into propagators focusing on the main compilation procedures. In order to present our algorithms we followed the pseudo code convention proposed in Section 2.2.

Given a compilable sub-program $\lambda \subseteq \pi$ in input, $\lambda$ is split in two subprograms, namely $\lambda_l$ and $\lambda_e$, as explained in previous section. Then $\lambda_l$ and $\lambda_e$ are compiled in different ways as described in the following. Recall that rules in $\pi \setminus \lambda$ follow the Ground&Solve pipeline.

### 3.3.1 Eager Propagator

Given a compilable sub-program $\lambda$, $\lambda_e$ is first normalized, obtaining the program $\lambda_e^n$, as described in the previous section. Then, Algorithm 4 compiles $\lambda_e^n$ as an eager propagator. Recall that a propagator is called to compute the deterministic consequences of one literal at time, which is given by the solver [26]. Thus, in the algorithms the variable $l$ printed in output by the compiler is used to store the literal to propagate in the propagator pseudocode. Roughly, the propagator matches ground literals to first order literals

**Algorithm 4** CompileProgram

```
   Input  : A normalized ASP program P
   Output: Prints propagator procedure of P propagating l
1  begin
2      ≪ I_l = ∅≫
3      ≪ switch pred(l)≫
4      for all r ∈ P do
5          if r is of type (3.1) then
6              CompileRule(r)
7          else if r is of type (3.2) then
8              CompileRuleWithSum(r)
9          else if r is of type (3.3) then
10             CompileRuleWithCount(r)
11         else if r is of type (3.4) then
12             for all c ∈ B_r do
13                 CompileConstraintWithStarter(r,c)
14     ≪return I_l≫
```

in the rules of $\lambda_e$ (as in the grounding) and tries to temporarily build the substitution that instantiates that rule, in order to determine where some propagations occur. In case of inconsistency, a reason is computed and passed to the solver for enabling learning [26]. In this process loop unrolling and dead code elimination [57] are performed using the syntactic information on the rule structure. More formally, Algorithm 4 first prints the declaration of an empty implication list, namely $I_l$, where the propagator stores all the propagations caused by the fact that a literal to progagate $l$ is added to the interpretation $I$. Afterward, according to the predicate name of $l$, the propagator evaluates the propagations depending on the kind of the rule in which the predicate appears. In turn, propagators for rule of the form (3.1) are generated by Algorithm 5. To generate such kind of propagators, the algorithm prints two different switch cases for checking if $l$ appears in $h$ (line 3) or in $b$ (line 18). In both cases the algorithm prints the code that initializes a variable substitution (in our code convention $\epsilon$ denotes an empty substitution) from variables in the head (resp. body) of the rule to constants in $l$ (lines 4-6 and 19-21, respectively). Subsequently, true and undefined atoms matching $\sigma(b)$ (resp. $\sigma(h)$) (lines 7-8 and 22-23, respectively)

are stored in two sets, namely $T$ and $U$. Note that $\sigma$ is a variable substitution that replaces variables occurrences with constants they are mapped to, e.g., consider $l = a(X)$, if $\sigma(X) = 1$ then $\sigma(l) = a(1)$. In the first switch case, the generated code evaluates propagations starting from the literal $l$ that can be matched to $h$ via a variable substitution. Thus, if $l$ is positive then $h$ is

**Algorithm 5** CompileRule

**Input** : A rule r of type (3.1), i.e., $h \leftarrow b$
**Output**: Prints propagator procedure for the rule r

```
1  begin
2      ≪σ = ϵ≫ ≪u = ⊥≫
3      ≪case " [[pred(h)]]_ξ ":≫
4      for all i ∈ 1, ..., |trm(h)| do
5          if trm(h)[i] is variable then
6              ≪σ = σ ∪ { [[trm(h)[i]]]_ξ ↦ trm(l)[ [[i]]_ξ ]}≫
7      ≪  T = {p ∈ I+ | match(σ( [[b]]_ξ ), p)}≫
8      ≪  U = {p ∈ (B \ I)+ | match(σ( [[b]]_ξ ), p)}≫
9      ≪  if |T| == 0 ∧ |U| == 1 ∧ l ∈ I+≫
10     ≪     u = p | p ∈ U≫
11     ≪     R = {l} ∪ {p ∈ I− | match(σ( [[b]]_ξ ), p)}≫
12     ≪     I_l = I_l ∪ (u, R)≫
13     ≪  else if |T| = 0 ∧ l ∈ I−≫
14     ≪     forall p ∈ U do≫
15     ≪        u = p≫
16     ≪        I_l = I_l ∪ (ū, {l})≫
17     ≪     done≫
18     ≪case " [[pred(b)]]_ξ ":≫
19     for all i ∈ 1, ..., |trm(b)| do
20         if trm(b)[i] is variable ∧ trm(b)[i] ∈ trm(h) then
21             ≪σ = σ∪{ [[trm(b)[i]]]_ξ ↦ trm(l)[ [[i]]_ξ ]}≫
22     ≪  T = {p ∈ I+ | match(σ( [[b]]_ξ ), p)}≫
23     ≪  U = {p ∈ (B \ I)+ | match(σ( [[b]]_ξ ), p)}≫
24     ≪  if l ∈ I+≫
25     ≪     if σ( [[h]]_ξ ) ∈ (B \ I)+≫
26     ≪        u = σ( [[h]]_ξ )≫
27     ≪        I_l = I_l ∪ (u, {l})≫
28     ≪  else if |T| = 0 ∧ |U| = 0 ∧ σ( [[h]]_ξ ) ∈ (B \ I)+≫
29     ≪     R = {p ∈ I− | match(σ( [[b]]_ξ ), p)}≫
30     ≪     u = σ( [[h]]_ξ )≫
31     ≪     I_l = I_l ∪ (ū, R)≫
```

**Algorithm 6** CompileRuleWithSum

**Input** : A rule r of type (3.2), $h \leftarrow d, \#sum(\{V : b\}) \geq g$
**Output**: Prints propagator code starting for rule r

**begin**
  $k = i$ s.t. $trm(b)[i] = V[0], i \in \{1, ..., |trm(b)|\}$
  $\ll \sigma = \epsilon \gg$
  $\ll$**case** " $[\![pred(h)]\!]_\natural$ "$\gg$
  **for all** $i \in 1, ..., |trm(h)|$ **do**
    **if** $trm(h)[i]$ *is variable* **then**
      $\ll \sigma = \sigma \cup \{ [\![trm(h)[i]]\!]_\natural \mapsto trm(l)[ [\![i]\!]_\natural ]\} \gg$
  $\ll M = \{p \in \mathcal{B} | match(\sigma( [\![b]\!]_\natural ), p)\} \gg$
  $\ll T = (M \cap I^+);\ F = (M \cap I^-);\ U = (M \setminus I)^+ \gg$
  $\ll act, pos = getSum(T, U,\ [\![k]\!]_\natural ) \gg$
  $\ll$ **if** $l \in I^+ \gg$
  $\ll I_l = I_l \cup \{(p, \{l\} \cup F) \mid p \in U \wedge act + pos - trm(p)[ [\![k]\!]_\natural ] < \sigma( [\![g]\!]_\natural )\} \gg$
  $\ll$ **else** $I_l = I_l \cup \{(\overline{p}, \{l\} \cup T) \mid p \in U \wedge\ act + trm(p)[ [\![k]\!]_\natural ] \geq \sigma( [\![g]\!]_\natural )\} \gg$
  $\ll$**case** " $[\![pred(b)]\!]_\natural$ "$\gg$
  **for all** $i \in 1, ..., |trm(b)|$ **do**
    **if** $trm(b)[i]$ *is variable* $\wedge\ trm(b)[i] \in trm(h)$ **then**
      $\ll \sigma = \sigma \cup \{ [\![trm(b)[i]]\!]_\natural \mapsto trm(l)[ [\![i]\!]_\natural ]\} \gg$
  $\ll M = \{p \in \mathcal{B} \mid match(\sigma( [\![h]\!]_\natural ), p)\} \gg$
  $\ll T = (M \cap I^+);\ F = (M \cap I^-);\ U = (M \setminus I)^+ \gg$
  $\ll$ **forall** $h_l \in T \cup U \cup F$ **do**$\gg$
  $\ll \sigma_h = \sigma \gg$
  **for all** $i \in 1, ..., |trm(h)|$ **do**
    **if** $trm(h)[i]$ *is variable* **then**
      $\ll \sigma_h = \sigma_h \cup \{ [\![trm(h)[i]]\!]_\natural \mapsto trm(h_l)[ [\![i]\!]_\natural ]\} \gg$
  $\ll M' = \{p \in \mathcal{B} \mid match(\sigma_h( [\![b]\!]_\natural ), p)\} \gg$
  $\ll T' = (M' \cap I^+);\ F' = (M' \cap I^-);\ U' = (M' \setminus I)^+ \gg$
  $\ll act, pos = getSum(T', U',\ [\![k]\!]_\natural ) \gg$
  $\ll$ **if** $act \geq \sigma_h( [\![g]\!]_\natural ) \gg$
  $\ll I_l = I_l \cup (h_l, T') \gg$
  $\ll$ **else if** $act + pos < \sigma_h( [\![g]\!]_\natural ) \gg$
  $\ll I_l = I_l \cup (\overline{h_l}, F') \gg$
  $\ll$ **else if** $h_l \in T \gg$
  $\ll I_l = I_l \cup \{(p, \{h_l\} \cup F') \mid p \in U' \wedge act + pos - trm(p)[ [\![k]\!]_\natural ] < \sigma_h( [\![g]\!]_\natural )\} \gg$
  $\ll$ **else if** $h_l \in F \gg$
  $\ll I_l = I_l \cup \{(\overline{p}, \{h_l\} \cup T') \mid p \in U' \wedge act + trm(p)[ [\![k]\!]_\natural ] \geq \sigma_h( [\![g]\!]_\natural )\} \gg$
  $\ll$ **done**$\gg$

true w.r.t. $I$ and the propagator has to find a body that can support it (lines 9-12). Otherwise, if $l$ is negative, then $h$ is false w.r.t. $I$, then the propagator

**Algorithm 7** CompileRuleWithCount

**Input** : A rule r of type (3) $h \leftarrow d, \#count\{V : b\} \succ g$
**Output**: Prints propagator code for a rule of type (3)

```
1  begin
2      ≪case [[pred(h)]]_ι ≫
3      ≪   σ = ϵ≫
4      for all i ∈ 1, ..., |trm(h)| do
5          if trm(h)[i] is variable then
6              ≪σ = σ ∪ { [[trm(h)[i]]]_ι ↦ trm(l)[ [[i]]_ι ]}≫
7      ≪   M = {p ∈ B | match(σ( [[b]]_ι ), p)}≫
8      ≪   T = M ∩ I+   F = M ∩ I−   U = (M \ I)+≫
9      ≪   if l ∈ I+ ∧ |T| + |U| = σ(g)≫
10     ≪      I_l = I_l ∪ {(p, ({l} ∪ F)) | p ∈ U}≫
11     ≪   else if l ∈ I− ∧ |T| = σ(g) − 1 ≫
12     ≪      I_l = I_l ∪ {(p̄, ({l} ∪ T)) | p ∈ U}≫
13     ≪case [[pred(b))]]_ι ≫
14     ≪   σ = ϵ≫
15     for all i = 1, ..., |trm(b)| do
16         if trm(b)[i] is variable ∧ trm(b)[i] ∈ trm(H) then
17             ≪σ = σ ∪ { [[trm(b)[i]]]_ι ↦ trm(l)[ [[i]]_ι ]}≫
18     ≪   M = {p ∈ B | match(σ( [[h]]_ι ), p)}≫
19     ≪   T = M ∩ I+   F = M ∩ I−   U = (M \ I)+≫
20     ≪   forall h ∈ T ∪ U ∪ F do≫
21     ≪      σ_h = σ≫
22     for all i = 1, ..., |trm(h)| do
23         if trm(h)[i] is variable then
24             ≪σ_h = σ_h ∪ { [[trm(h)[i]]]_ι ↦ trm(h)[ [[i]]_ι ]}≫
25     ≪      MB = {p ∈ B | match(σ_h( [[b]]_ι ), p)}≫
26     ≪      TB = MB ∩ I+   FB = MB ∩ I−   UB = (MB \ I)+≫
27     ≪      if h_l ∈ T ∧ |TB| + |UB| = σ_h(g) ≫
28     ≪         I_l = I_l ∪ {(p, ({h_l} ∪ FB)) | p ∈ UB}≫
29     ≪      else if h_l ∈ F ∧ |TB| = σ_h(g) − 1 ≫
30     ≪         I_l = I_l ∪ {(p̄, ({h_l} ∪ TB)) | p ∈ UB}≫
31     ≪      else if h_l ∈ U ∧ |TB| ≥ σ_h(g)≫
32     ≪         I_l = I_l ∪ (h_l, TB)≫
33     ≪      else if h_l ∈ U ∧ |TB| + |UB| < σ_h(g) ≫
34     ≪         I_l = I_l ∪ (h̄_l, FB)≫
35     ≪   done≫
```

**Algorithm 8** CompileConstraintWithStarter

**Input** : A constraint C of type (5), a literal c
**Output**: Prints propagator code for the constraint C starting from c

```
1  begin
2      ≪case [[pred(c)]]_ℓ ≫
3      ≪   u = ⊥; σ = ϵ≫
4      for all i ∈ 1, ..., |trm(c)| do
5          if trm(c)[i] is variable then
6              ≪σ = σ ∪ { [[trm(c)[i]]]_ℓ ↦ trm(l)[ [[i]]_ℓ ]}≫
7      B = bodyOrdering(r, c)
8      for all j ∈ 1, ..., |B| do
9          ≪σ _[[j]]_ℓ = σ≫
10         if B[j] is a positive literal then
11             ≪T _[[j]]_ℓ = {p ∈ I^+| match(σ( [[B[j]]]_ℓ ), p)}; U _[[j]]_ℓ = ∅≫
12             ≪if u = ⊥≫
13             ≪   U _[[j]]_ℓ = {p ∈ (B \ I)^+| match(σ( [[B[j]]]_ℓ ), p)}≫
14             ≪forall t _[[j]]_ℓ ∈ T _[[j]]_ℓ ∪ U _[[j]]_ℓ do≫
15             for all k ∈ 1, ..., |trm(B[j])| do
16                 if trm(B[j])[k] is variable then
17                     ≪σ = σ ∪ { [[trm(B[j])[k]]]_ℓ ↦ trm(t _[[j]]_ℓ )[ [[k]]_ℓ ]}≫
18             ≪   if t _[[j]]_ℓ ∈ U≫
19             ≪      u = t _[[j]]_ℓ ≫
20         else
21             ≪t _[[j]]_ℓ = σ( [[B[j]]]_ℓ )≫
22             ≪if t _[[j]]_ℓ ∈ I ∨ (u = ⊥ ∧ t _[[j]]_ℓ ∈ (B \ I) ≫
23             ≪   if t _[[j]]_ℓ ∉ I≫
24             ≪      u = t _[[j]]_ℓ ≫
25     ≪         R = {l}≫
26     for all j ∈ 1, ..., |B| do
27         ≪         R = R ∪ {t _[[j]]_ℓ }≫
28     ≪         R = R \ {u}≫
29     ≪         I_l = I_l ∪ (u, R)≫
30     for all j ∈ 1, ..., |B| do
31         ≪σ_=σ_j≫
32         ≪if u = t _[[j]]_ℓ ≫
33         ≪   u = ⊥≫
34         if B[i] is a positive literal then
35             ≪done≫
```

has to set as false all possible body instantiations (lines 13-16). On the other hand, the second switch case evaluates propagations starting from the literal $l$ that can be matched to $b$ via a variable substitution. Thus, if $l$ is positive, then the body of the rule is true w.r.t. $I$ and then the propagator has to propagate $\sigma(h)$ as true (lines 25-27). Otherwise, if $l$ is negative and all other body instantiations that can support $\sigma(h)$ are false w.r.t. $I$, then $\sigma(h)$ is derived as false (lines 28-31). For rules of type (3.2), i.e., rules of the form $h \leftarrow d, \#sum(\{V : b\}) \geq g$, Algorithm 6 prints the specific propagator code. In particular, it prints two different switch cases for checking if the predicate of $l$ appears in the head of the rule (line 4) or in the aggregate atom (line 14). In both cases the propagator needs to build a variable substitution $\sigma$ to simulate the rule instantiation. This substitution maps the variables occurring in $h$ (resp. $b$) with the constants in $l$ (lines 5-7 and 15-17). In the first switch case the predicate of $l$ appears in $h$ and so, the propagator first collects positive and negative literals matching $\sigma(b)$ that are true w.r.t. $I$ into the set $T$, and $F$ respectively, and undefined atoms matching $\sigma(b)$ into the set $U$ (lines 8-9). Then, the propagator computes actual and possible sums as the sum of the weights associated with literals in $T$ and $U$, respectively (line 10). Now, the propagator has to distinguish two cases. If $l$ is positive (line 11), thus $h$ is true w.r.t. $I$, then the aggregate must be true in order to support $h$. This can be done by checking that the maximal possible sum is always greater than or equal to $g$. Therefore, all literals $p$ whose falsity would make the maximal possible sum less than $g$ are derived as true (line 12). On the other hand, if $l$ is negative, thus $h$ is false w.r.t. $I$, then the aggregate must be false, i.e., the propagator ensures that the actual sum does not exceed $g$. Therefore, all literals $p$ whose truth would make the actual sum greater than or equal to $g$ are derived as false (line 13). The second case, instead, implements the case in which the predicate of $l$ appears in the aggregate. In this scenario, the propagator collects in $T$ (respectively, $F$) all possible positive (respectively, negative) literals that are true w.r.t. $I$ and match the variable substitution applied to $h$, and in $U$ all undefined atoms matching the variable substitution applied to $h$ (lines 18-19). For each literal, say $h_l \in T \cup F \cup U$, the propagator considers three different types of propagations. In particular, if $h_l \in T$ or $h_l \in F$, then the propagator behaves as in the previous case (lines 32-35). Instead, if $h_l \in U$, then the propagator considers two cases. If the actual sum is greater than or equal to the guard (line 28), then the body of the rule is true, and so the head is derived as true (line 29). On the other hand, if the maximal possible sum is less than the guard $g$ (line 30)

then the head is derived as false (line 31). A similar kind of propagators is produced for rules of the form (3.3). More precisely, Algorithm 7 prints the propagator code for a rule $r$ of type (3.3). In the generated code, two different cases of propagation are handled according to the predicate of $l$ (line 2 and 13). In both cases the algorithm prints the code that builds the variable substitution $\sigma$ that maps variables appearing in the head (resp. body) of $r$ to constants in $l$ (lines 3-6 and 14-17). If the predicate of $l$ appears in the head of $r$ then there exists an instantiation of $r$ having $l$ in the head and so, if $l$ is positive (i.e. the head is true) then the aggregate has to be propagated as true, otherwise (i.e. the head is false) the aggregate has to be false. Thus, if $l$ is positive the propagator avoids that the aggregate becomes false and so, if the total number of atoms, matching $\sigma(b)$ that are true or not yet assigned w.r.t. $I$, is equals to guard then those literals not assigned yet must be derived as true (lines 9 to 10). Otherwise if $l$ is negative then the propagator avoids that the aggregate becomes true. Thus, if the number of positive literals, matching $\sigma(b)$ that are true w.r.t. $I$, is equal to the guard minus one then all the literals matching $\sigma(b)$ that are not been assigned yet must be propagated as false (lines 11 to 12). On the other hand, if $l$ appears in $b$, the propagator collects literals that match $\sigma(h)$ and are true w.r.t. $I$ in $T$ and $F$ and positive literals not yet assigned that match $\sigma(h)$ in $U$ (lines 18 to 19). Afterward, for each literal $h_l \in T \cup F$ the propagator code produced behaves as described for the case in which $l$ matches $h$. Instead, for each literal $h_l \in U$ the propagator checks if the number of positive literals that match $\sigma(b)$ and are true w.r.t. $I$ is greater than or equal to the guard (i.e. the aggregate is true) and eventually propagates $h_l$ as true. On the contrary, if the number of positive literals that match $\sigma(b)$ and are true or not yet assigned w.r.t. $I$ is less then the guard (i.e. the aggreagate is false) then $h_l$ is propagated as false (lines 31 - 34). For rules of the form (3.4), instead, our approach resorts the one proposed by [26]. In particular, given a constraint $C$ of type (3.4) and a literal $c \in B_C$, Algorithm 8 prints a switch case to check if the predicate of $l$ (literal that we want to propagate) matches the predicate of $c$. In this case the propagator initializes the undefined literal $u$, that will be propagated, to $\perp$ and builds a variable substitution that maps variables in $c$ with the constant in $l$ (lines 3 - 6). Afterward, the body of $C$ is reordered in a list $B$ in such a way that the literal $c$ is not in $B$ and the positive body will be evaluated before the negative one. For each positive literal $b_i$ the propagator collects positive literals, matching $\sigma(b_i)$, that are true w.r.t. $I$. If the undefined literal $u$ has not been found yet then

the propagator collects also positive literals, matching $\sigma(b_i)$, that are not yet assined. Afterward, the propagator iterates over these literals and updates the variable substitution (lines 10 to 19). For each negative literal $n \in B$, instead, the propagator checks if $\sigma(n)$ belongs to $I$ or if the literal $u$ has not been found yet. If $u = \bot \ \wedge \ \sigma(n) \notin I$ then the propagator assigns $u$ to $\sigma(n)$ (lines 20 - 24). Inside the deepest block, the propagator builds the reason of the propagation and propagates the complement of $u$ in order to prevent constraint violations (lines 25 - 29). Once the propagator for each rule is generated the resulting module represent an eager propagator that is able to simulates the inferences of rules in $\lambda_{eager}$ without grounding them in advance. In order to better understand the idea behind our approach, the following example reports the code generated for a rules of the form (3.2).

**Example 7.** *Let $r$ be the following rule:*

$$a(X,Y) \leftarrow d(X,Y), \#sum\{Z : b(X,Z,W)\} \geq Y.$$

*Then, Algorithm 6 executed on the rule $r$ produces the propagator reported in Algorithm 9, whose execution is reported in the following. For the execution of the propagator, we assume that $\mathcal{B}$ is equal to $\{a(1,2),\ a(2,2),\ b(1,1,1),\ b(1,2,1),\ b(2,2,2),\ b(2,1,2)\}$ and $I = \{d(1,2),\ d(2,2)\}$. Then, propagation is triggered when a literal, say $a(1,2)$, is added to $I$.*

*At this point, the propagator is called with $l = a(1,2)$. Since the predicate of $l$ is in the head of $r$, the propagator enters in the first switch case, updates $\sigma$ mapping $X \mapsto 1$ and $Y \mapsto 2$, and computes the sets $T = F = \emptyset$, and $U = \{b(1,1,1), b(1,2,1)\}$. By considering $T$ and $U$, the actual sum act is equal to 0 whereas the possible sum pos is equal to 3. In this case, $l \in I^+$ and so the propagator iterates the set $U$. Specifically, it first considers $p = b(1,1,1)$, thus the inequality $act + pos - 1 < 2$ does not hold. Then, the propagator considers $p = b(1,2,1)$. At this point, the inequality $act + pos - 2 < 2$ holds, therefore $I_l$ is extended with $b(1,2,1)$ and with the reason leading to its inference (i.e., $I_l = \{b(1,2,1), \{a(1,2)\}\}$). No other atoms in $U$ are available, hence the propagator infers only $b(1,2,1)$ as true with reason $\{a(1,2)\}$.*

*At this point, suppose that $\sim b(2,2,2)$ is added to $I$, then the propagator is called with $l = \sim b(2,2,2)$. Since predicate of $l$ is in the aggregate set occurring in the body the propagator enters in the second switch case, and computes $\sigma = \{(X \mapsto 2)\}$, $T = \emptyset$, $F = \emptyset$, and $U = \{a(2,2)\}$. Then, the propagator iterates over $T$, $U$, and $F$. In this case, there is only one atom, i.e. $h_l = a(2,2)$, therefore, $\sigma_h = \{(X \mapsto 2), (Y \mapsto 2)\}$ is updated and body literals are collected*

*in* $UB = \{b(2,1,2)\}$, $TB = \emptyset$ *and* $FB = \{\sim b(2,2,2)\}$. *Considering* $UB$ *and* $TB$, $act = 0$ *and* $pos = 1$. *Since* $h \in U$ *and* $act + pos < 2$ *holds,* $\sim a(2,2)$ *is derived as true with reason* $\{\sim b(2,2,2)\}$.

**Algorithm 9** ExampleSumPropagator

```
1  begin
2      case "a"
3          σ = ϵ    σ = σ ∪ {X ↦ l[1]}     σ = σ ∪ {Y ↦ l[2]}
4          T = {p ∈ I⁺| match(σ(b(X, Z, W), p)}
5          U = {p ∈ (B \ I)⁺ | match(σ(b(X, Z, W)), p)}
6          act, pos = getSum(T, U, 2)
7          if l ∈ I⁺
8              R = {l} ∪ {p ∈ I⁻| match(σ(b(X, Z, W)), p)}
9              forall p ∈ U do
10                 if act + pos − trm(p)[2] < σ(Y)
11                     I_l = I_l ∪ (p, R)
12             done
13         else
14             forall p ∈ U do
15                 if act + trm(p)[2] ≥ σ(Y)
16                     I_l = I_l ∪ (p̄, {l} ∪ T)
17             done
18     case "b"
19         σ = ϵ
20         σ = σ ∪ {X ↦ trm(l)[1]}
21         T = {p ∈ I⁺| match(σ(a(X, Y)), p)}
22         F = {p ∈ I⁻| match(σ(a(X, Y)), p)}
23         U = {p ∈ (B \ I)⁺| match(σ(a(X, Y)), p)}
24         forall h_l ∈ T ∪ U ∪ F do
25             σ_h = σ    σ_h = σ_h ∪ {X ↦ trm(h_l)[1]}    σ_h = σ_h ∪ {Y ↦ trm(h_l)[2]}
26             UB = {p ∈ (B \ I)⁺| match(σ_h(b(X, Z, W)), h)}
27             FB = {p ∈ I⁻| match(σ_h(b(X, Z, W)), p)}
28             TB = {p ∈ I⁺| match(σ_h(b(X, Z, W)), p)}
29             act, pos = getSum(TB, UB, 2)
30             if h_l ∈ T
31                 forall p ∈ UB do
32                     if act + pos − trm(p)[2] < σ_h(Y)
33                         I_l = I_l ∪ (p, {h_l} ∪ FB)
34                 done
35             else if h_l ∈ F
36                 forall p ∈ UB do
37                     if act + trm(p)[2] ≥ σ_h(Y)
38                         I_l = I_l ∪ (p̄, {h_l} ∪ TB)
39                 done
40             else
41                 if act ≥ σ_h(Y)
42                     I_l = I_l ∪ (h_l, TB)
43                 else if act + pos < σ_h(Y)
44                     I_l = I_l ∪ (h̄_l, FB)
45         done
```

**Algorithm 10** CompileLazyRule

**Input** : A rule r
**Output**: Prints lazy propagator code for a program $P$

```
1  begin
2      h = H_r // head of r
3      B = bodyOrdering(B_r)
4      ≪σ = ϵ≫
5      PrintNestedJoin(B, r, 0)
6      ≪        I = I ∪ {σ(h)}≫
7      for all i ∈ 1, ..., |B| do
8          ≪σ = σ ⟦i,0⟧_t ≫
9          if B[i] is positive literal then
10             ≪done≫
```

### 3.3.2 Lazy Propagator

Lazy propagator instantiates the body of rules by using literals in the model $I$, and extends $I$ with the heads obtained from such body instantiations. In particular, for each rule $r \in \lambda_l$, Algorithm 10 generates an ad-hoc lazy propagator. The algorithm first reorders the body of $r$ in such a way that the positive body is evaluated before the negative one (line 3). Subsequently, the algorithm calls Algorithm 11 on the reordered body ($B$), the rule $r$, and the index 0 as parameters. Then, for each positive literal $b$ in $B$, Algorithm 11 prints the code to iterate over positive literals that are true w.r.t. $I$ and match $\sigma(b)$ (lines 5-9). For each negative literal $n$ in $B$, Algorithm 11 prints the code to check if the negative literal $\sigma(n)$ is true w.r.t. $I$ (lines 31-32). For each aggregate literal, instead, the algorithm recursively calls itself with the list of literals inside the aggregate, the rule $r$, and the index $j$ of the aggregate as parameters (line 15). In this way, the algorithm prints nested join loops and if statements that evaluate the aggregate conjunction. After that, the algorithm prints the code that computes the sum of the weights (or count) of the elements in the aggregate set (lines 16-21). Once nested join loops and if statements for the aggregate conjunction are closed, the algorithm prints the code that either checks if the aggregate relation is verified or maps the aggregate guard to the aggregate value. Then, Algorithm 10 finalizes the procedure, by printing the code that adds $\sigma(H_r)$ to the model $I$ (line 6).

**Algorithm 11** PrintNestedJoin

**Input** : A list of literals B, a rule r, an index i
**Output**: Prints nested join to instantiate B

```
1  begin
2      for all j ∈ 1, ..., |B| do
3          ≪σ ⟦j,i⟧_ι = σ≫
4          if B[j] is a positive literal then
5              ≪T ⟦j,i⟧_ι = {p ∈ I⁺| match(σ( ⟦B[j]⟧_ι ), p)}≫
6              ≪forall t ⟦j,i⟧_ι ∈ T ⟦j,i⟧_ι do≫
7              for all k ∈ 1, ..., |trm(B[i])| do
8                  if trm(B[j,i])[k] is variable then
9                      ≪σ = σ ∪ { ⟦trm(B[j])[k]⟧_ι ↦ trm(t ⟦j,i⟧_ι )[ ⟦k⟧_ι ]}≫
10         else if B[j] is an aggregate literal then
11             B_a = bodyOrdering(conj(B^a_r))
12             V = vars(B^a_r)
13             a_s = ∅
14             a_v = 0
15             PrintNestedJoin(B_a, r, j)
16             ≪            if σ( ⟦V⟧_ι ) ∉ a_s≫
17             ≪               a_s = a_s ∪ σ( ⟦V⟧_ι )≫
18             if B[j] is sum aggregate then
19                 ≪            a_v+ = σ( ⟦V[0]⟧_ι )≫
20             else
21                 ≪            a_v+ = 1≫
22             for k ∈ 1, ..., |conj(B_a)| do
23                 ≪σ = σ ⟦k,j⟧_ι ≫
24                 if conj(B_a)[k] is positive literal then
25                     ≪done≫
26             if B[j] is bound relation then
27                 ≪if a_v ≻ σ( ⟦guard(B^a_r)⟧_ι )≫
28             else
29                 ≪σ = σ ∪ { ⟦guard(B^a_r)⟧_ι ↦ a_v}≫
30         else
31             ≪t ⟦j,i⟧_ι = σ( ⟦B[j]⟧_ι )≫
32             ≪if t ⟦j,i⟧_ι ∈ I ≫
```

## 3.4 Experiments

### 3.4.1 Experiments Setup

In the experiments we considered three different settings.

**Setting (i).** A simple benchmark that we use as a motivating example to show the limits of Ground&Solve:

$$\begin{array}{ll} r_1: & d(1..k) \leftarrow \\ r_2: & \{a(X) : d(X); b(Y) : d(Y)\} \leftarrow \\ r_3: & \leftarrow \#count\{X : a(X)\} > Y, b(Y). \end{array}$$

$r_1$ is a shortcut for $k$ facts of the form $d(i) \leftarrow (i = 1, \ldots, k)$; and $r_2$ contains a *choice rule* [18];

**Setting (ii).** All benchmarks from ASP competitions [20] including at least one rule with aggregates that can be compiled under our conditions. This setting is useful to analyze the performance of our compilation-based technique on benchmarks that do not present a grounding bottleneck problem. The selected benchmarks are the following: Abstract Dialectical Framework (ADF), Bottle Filling Problem (BF), Connected Maximum Density-Still Life (CMD), Crossing Minimization (CM), Incremental Scheduling (IS), Partner Units (PU), Solitaire (S), Weighted Sequence Problem (WS);

**Setting (iii).** Three grounding-intensive benchmarks taken from the literature, in particular Component Assignment Problem (CA) proposed by [4], Dynamic In-Degree Counting (DIC) and Exponential-Save (ES) proposed by [15].

**Hardware and Software.** In all the experiments, the compilation-based approach, reported as WaspProp, has been compared with the plain version of Wasp [3] v. 169e40d and with the state-of-the-art system Clingo v. 5.4.0 [38]. All the tested systems use gringo (included in the binary of Clingo) as grounder. Moreover, for the benchmarks Dynamic In-Degree Counting and Exponential-Save we considered also the system Alpha [66], which is based on lazy-grounding techniques. Alpha cannot be used for other experiments since it does not support some of the language constructs

| k | WaspProp | | Wasp | | Clingo | |
|---|---|---|---|---|---|---|
| | t | mem | t | mem | t | mem |
| 1000 | **0** | 0 | 1.02 | 59.6 | 0.66 | 40.2 |
| 2000 | **0.1** | 18.6 | 4.36 | 286.5 | 3.23 | 303.5 |
| 3000 | **0.24** | 38.5 | 9.98 | 696.3 | 7.68 | 709.5 |
| 4000 | **0.53** | 53.7 | 18.47 | 1168.1 | 13.62 | 1216.8 |
| 5000 | **0.93** | 74.6 | 28.8 | 2215.4 | 22.23 | 1933.3 |
| 6000 | **1.19** | 109.8 | 42.47 | 2807.2 | 32.73 | 2871.1 |
| 7000 | **1.44** | 142.3 | 58.31 | 3402 | 42.88 | 3576.7 |
| 8000 | **1.96** | 152 | - | - | - | - |
| 9000 | **2.89** | 191.6 | - | - | - | - |
| 10000 | **3.87** | 254.7 | - | - | - | - |
| 20000 | **12.52** | 898.5 | - | - | - | - |
| 30000 | **26.43** | 1888.3 | - | - | - | - |
| 40000 | **58.88** | 3319.6 | - | - | - | - |
| 50000 | - | - | - | - | - | - |

Table 3.1: Comparison with wasp and clingo on setting (i).

used in the benchmarks (e.g., *choice rules* with bounds). For each benchmark, we selected the rules to be compiled by looking at the potential grounding issues. Experiments were executed on Xeon(R) Gold 5118 CPUs running Ubuntu Linux (kernel 5.4.0-77-generic), time and memory are limited to 2100 seconds and 4GB, respectively.

### 3.4.2 Results

Obtained results are reported in Table 3.1 and Table 3.2. In the tables we report for each solver the number of solved instances (sol.), the sum of running times (sum t) in seconds, and the average used memory (avg mem) in MB. Moreover, concerning WaspProp, Table 3.2 reports an additional column (comp. t), representing the compile time in seconds. The latter is not included in the sum of the time, since the compilation is done only once for each benchmark (with the exception of Exponential-Save) and, thus, can be done offline. Concerning the setting (i), we observe that Clingo and Wasp cannot solve instances with $k \geq 8000$, whereas WaspProp can efficiently handle instances up to values of $k = 40000$. Concerning the setting (ii), we observe that Clingo obtains the best performance overall, and it

| Bench. | # | WaspProp | | | | Wasp | | | Clingo | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | sol. | sum t | avg m | comp. t | sol. | sum t | avg m | sol. | sum t | avg m |
| ADF | 200 | 117 | 23467.7 | 118.9 | 6.47 | 123 | 24311.1 | 117.2 | **200** | 1244.6 | 26.4 |
| BF | 100 | **100** | 2208.3 | 773.8 | 11.98 | **100** | 545.8 | 761.8 | **100** | **394.0** | 213.4 |
| CMD | 26 | **6** | 2166.8 | 26.4 | 31.8 | **6** | **427.5** | 31.4 | 4 | 85.4 | 11.1 |
| CM. | 85 | **84** | 305.3 | 16.4 | 8.97 | **84** | **257.3** | 14.5 | 51 | 10013.0 | 20.4 |
| IS | 500 | 329 | 25960.7 | 96.6 | 6.94 | 317 | 42282.4 | 210.9 | **345** | 18952.8 | 253.7 |
| PU | 112 | 52 | 35525.7 | 160.9 | 12.51 | 69 | 29612.5 | 247.6 | **80** | 7147.8 | 73.8 |
| S | 27 | **25** | 601.7 | 27.0 | 12.03 | **25** | **140.2** | 18.4 | **25** | 273.1 | 9.8 |
| WS | 65 | **65** | 6663.4 | 36.1 | 8.46 | **65** | 4713.5 | 32.3 | **65** | **569.6** | 14.5 |
| CA. | 302 | **188** | 56137.2 | 814.4 | 20.79 | 70 | 15325.81 | 973.3 | 118 | 36832.0 | 1288.6 |
| DIC. | 80 | **80** | **48.0** | 62.0 | 6.48 | **80** | 1374.6 | 619.4 | **80** | 1282.1 | 551.8 |
| ES | 27 | **21** | 2594.1 | 527.0 | * | 6 | 18.5 | 369.8 | 7 | 24.4 | 365.4 |

Table 3.2: Comparison of the compilation-based approach with WASP and CLINGO on instances of settings (ii) and (iii).

is faster than Wasp and WaspProp on the non-grounding-intensive benchmarks (above the double line in Table 3.2). The impact of the proposed technique can be seen by comparing WaspProp and Wasp. In this case, we observe that the former is competitive with the latter in all the benchmarks but Abstract Dialectical Framework and Partner Units, where Wasp solves 6 and 17 more instances than WaspProp. Nonetheless, WaspProp performs better than Wasp on the benchmark Incremental Scheduling, solving 12 more instances, where the lazy propagator gives a clear advantage. Interestingly, on this benchmark, WaspProp also uses less memory than Wasp and Clingo. Indeed, if only 512 MB are available (as reasonable in some case) WaspProp solves 57 and 53 instances more than Wasp and Clingo, respectively. Concerning the setting (iii), WaspProp outperforms Wasp in all the tested benchmarks solving 133 more instances overall. It is important to observe that each instance of Exponential-Save requires to be compiled, since aggregates to be compiled are part of the instances. Therefore, in this benchmark, the solving time includes also the compilation time. Concerning Dynamic In-Degree Counting, WaspProp and Wasp solve the same number of instances, but WaspProp is much faster. Moreover, we observe that WaspProp solves 84 more instances than Clingo overall. Finally, we observe that WaspProp is competitive also with Alpha, since the latter solves 80 instances of Dynamic In-Degree Counting in 492.0 s using 649.1 MB, and 27 instances of Exponential-Save in 332.9 s using 227.8 MB.

# Chapter 4

# Compilation of Tight Normal Programs

In this chapter, we push forward the idea behind compilation-based approaches and we propose a new system, named ProASP, for compiling tight normal programs into propagators. This represents a significant step forward, by extending the class of ASP programs that can be compiled *beyond constraints*. The proposed system skips entirely the grounding phase and performs solving by injecting custom propagators in the glucose SAT-solver. An experiment, conducted on grounding-intensive ASP benchmarks, shows that ProASP is capable of solving instances that are out of reach for state-of-the-art ASP systems.

## 4.1 The ProASP system

ProASP has two main components, referred to as `Compiler` and `Solver` (see Figure 4.1). The `Compiler` takes as input a non-ground ASP program $P$, and generates the code of an ASP solver, referred to as `Solver`, that can evaluate $P$. `Solver` is made of a CDCL SAT solver augmented with two additional modules: the `Generator` and the `Propagator`. As usual in ASP, the instance of a problem is assumed to be provided without loss of generality as a set of facts [16]. Given the instance in input, `Generator` generates (a subset of) the Herbrand base $B_P$, which is used to fill the CDCL solver data structures with propositional atoms (corresponding to the decision variables in SAT); whereas, the `Propagator` module contains the propagators that,

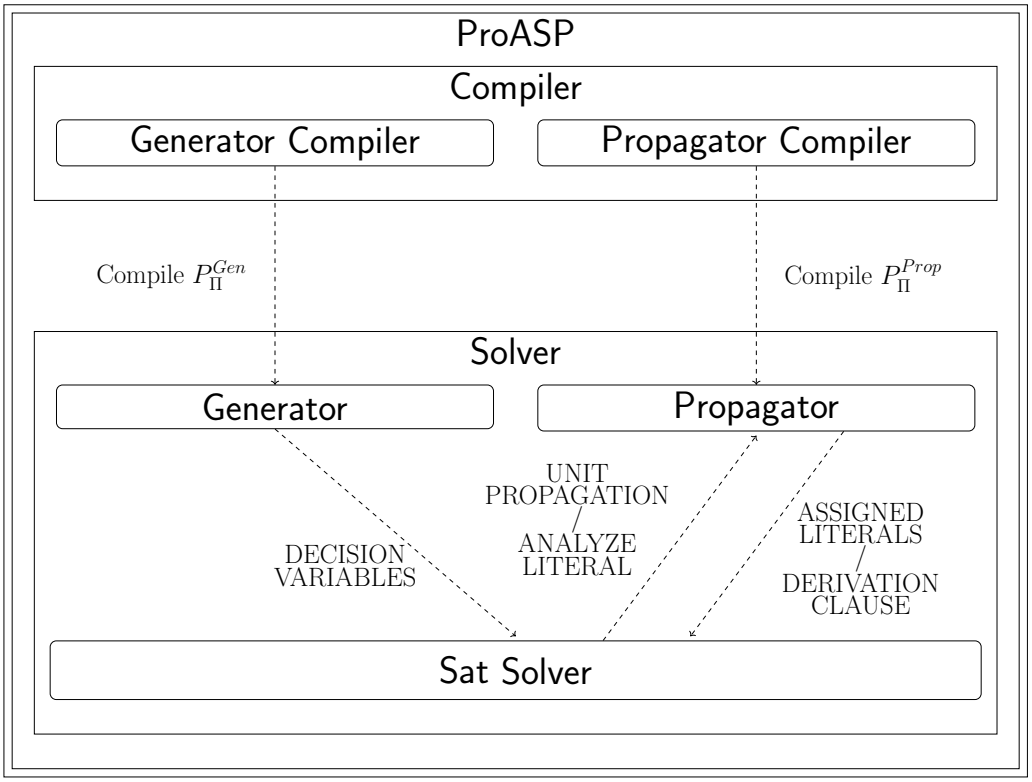


Figure 4.1: ProASP Architecture

very roughly, simulate the grounding of $P$ (as it would have been properly transformed into a SAT formula). `Compiler` and `Solver` are detailed in the following subsections. Comparing ProASP with existing compilation-based ASP solvers (see Section 2.2), we stress that the program in input does not have to be split into sub-programs, and no grounder is needed.

### 4.1.1 ProASP `Compiler`

Let $\Pi$ denote the ASP program in input. We now introduce a rewriting of $\Pi$ that ensures that $(i)$ rules are treated uniformly, and $(ii)$ the propagations, ensuring that atoms are supported, are modeled by constraints. Without loss of generality, we assume there are no two different atoms sharing the same predicate name having different arity.

For a predicate $p$, let $\Pi_p = \{r_1, \ldots, r_n\}$ be the set of rules of $\Pi$ s.t. the

predicate of their head atom is $p$, then if $n > 1$, $unique(\Pi_p)$ is the program:

$$\begin{array}{ll} sup_r(\mathcal{T}(H_r)) \leftarrow B_r & \forall r \in \Pi_p \\ \leftarrow sup_r(\mathcal{T}(H_r)),\ not\ H_r & \forall r \in \Pi_p \\ \leftarrow p(\vec{x}),\ not\ sup_{r_1}(\vec{x}), \ldots, not\ sup_{r_n}(\vec{x}) & \end{array}$$

where $\vec{x}$ is a list of $m$ variables, and $m$ is the arity of $p$; otherwise (i.e., $n = 1$) $unique(\Pi_p) = \Pi_p$. Then, $P_\Pi$ is defined as the union of $unique(\Pi_p)$ for all $p$ occurring in $\Pi$. Note that each predicate name occurs in at most one rule head in $P_\Pi$.

**Example 8.** *Let $\Pi_{ex}$ be the following (sub)program:*

$$\begin{array}{lll} r_1: & a(1,Z) & \leftarrow b(Y),\ c(Y,Z) \\ r_2: & a(X,Z) & \leftarrow d(X,Y),\ c(Y,Z),\ not\ e(Z) \end{array}$$

*In this example $b$, $c$, $d$, and $e$ do not appear in any rule head of $\Pi_{ex}$ and so they are edb predicates (i.e. defined only as facts that will appear in some problem instances), so let us focus on predicate $a$. In this case $\Pi_a = \{r_1, r_2\}$ is rewritten as follows:*

$$\begin{array}{l} sup_{r_1}(1,Z) \leftarrow b(Y),\ c(Y,Z) \\ sup_{r_2}(X,Z) \leftarrow d(X,Y),\ c(Y,Z),\ not\ e(Z) \\ \leftarrow sup_{r_1}(1,Z),\ not\ a(1,Z) \\ \leftarrow sup_{r_2}(X,Z),\ not\ a(X,Z) \\ \leftarrow a(X_1,X_2),\ not\ sup_{r_1}(X_1,X_2),\ not\ sup_{r_2}(X_1,X_2) \end{array}$$

□

We now define $P_\Pi^{Prop}$ as the program obtained from $P_\Pi$ by rewriting each rule $r \in P_\Pi$ that is not a constraint, as follows:

$$\begin{array}{lll} H_r & \leftarrow aux_r(\mathcal{V}(B_r^+)) & \\ & \leftarrow aux_r(\mathcal{V}(B_r^+)), \bar{l} & \forall l \in B_r \\ & \leftarrow l_1,\ \ldots,\ l_n,\ not\ aux_r(\mathcal{V}(B_r^+)) & \end{array}$$

where $B_r = l_1, \ldots,\ l_n$. Basically, $P_\Pi^{Prop}$ models a form of completion [23] by means of additional constraints.

**Example 9.** *Let us consider the rule* $sup_{r_1}(1, Z) \leftarrow b(Y),\ c(Y, Z)$ *from Example 8. It is rewritten as follows:*

$$\begin{aligned} sup_{r_1}(1, Z) \ &\leftarrow aux_r(Y, Z) \\ &\leftarrow aux_r(Y, Z),\ not\ b(Y) \\ &\leftarrow aux_r(Y, Z),\ not\ c(Y, Z) \\ &\leftarrow b(Y),\ c(Y, Z),\ not\ aux_r(Y, Z) \end{aligned}$$

□

Program $P_{\Pi}^{Prop}$ is used to build the Propagator module. We now define program $P_{\Pi}^{Gen}$ that is used to build the Generator module.

Given a rule $r \in \Pi$, let $p$ be the predicate appearing in $H_r$, then $supRule(r)$ is the rule $sup_r(\mathcal{T}(H_r)) \leftarrow B_r$ if $|\Pi_p| > 1$, otherwise $supRule(r)$ is $r$. Moreover, for each rule $r' \in P_{\Pi}$ that is not a constraint, $auxRule(r')$ is the rule $H_{r'} \leftarrow aux_{r'}(\mathcal{V}(B_{r'}^{+}))$. The program $P_{\Pi}^{Gen}$ is obtained by rewriting each rule $r \in \Pi$ as follows:

$$\begin{aligned} &B_{r''} \leftarrow B_{r'} \\ &H_{r''} \leftarrow B_{r''} \\ &H_r \leftarrow H_{r'} \quad if\ r \neq r' \end{aligned}$$

where $r' = supRule(r)$ and $r'' = auxRule(r')$

**Example 10.** *Let us consider the program* $\Pi_{ex}$ *from Example 8, then* $P_{ex}^{Prop}$ *is the following:*

$$\begin{aligned} &\leftarrow sup_{r_1}(1, Z),\ not\ a(1, Z) \\ &\leftarrow sup_{r_2}(X, Z),\ not\ a(X, Z) \\ &\leftarrow a(X_1, X_2),\ not\ sup_{r_1}(X_1, X_2),\ not\ sup_{r_2}(X_1, X_2) \end{aligned}$$

$$\begin{aligned} &sup_{r_1}(1, Z) \leftarrow aux_{r_1}(Y, Z) \\ &\leftarrow aux_{r_1}(Y, Z),\ not\ b(Y) \\ &\leftarrow aux_{r_1}(Y, Z),\ not\ c(Y, Z) \\ &\leftarrow b(Y),\ c(Y, Z),\ not\ aux_{r_1}(Y, Z) \\ &sup_{r_2}(Y, Z) \leftarrow aux_{r_2}(X, Y, Z) \\ &\leftarrow aux_{r_2}(X, Y, Z),\ not\ d(X, Y) \\ &\leftarrow aux_{r_2}(X, Y, Z),\ not\ c(Y, Z) \\ &\leftarrow aux_{r_2}(X, Y, Z),\ e(Z) \\ &\leftarrow d(X, Y),\ c(Y, Z),\ not\ e(Z),\ not\ aux_{r_2}(X, Y, Z) \end{aligned}$$

*and* $P_{ex}^{Gen}$ *is the following:*

$$
\begin{aligned}
&aux_{r_1}(Y,Z) \leftarrow b(Y),\ c(Y,Z)\\
&aux_{r_2}(X,Y,Z) \leftarrow d(X,Y),\ c(Y,Z),\ not\ e(Z)\\
&sup_{r_1}(1,Z) \leftarrow aux_{r_1}(Y,Z)\\
&sup_{r_2}(Y,Z) \leftarrow aux_{r_2}(X,Y,Z)\\
&a(X_1,X_2) \leftarrow sup_{r_1}(X_1,X_2)\\
&a(X_1,X_2) \leftarrow sup_{r_2}(X_1,X_2)
\end{aligned}
$$

□

Once programs $P_{\Pi}^{Prop}$ and $P_{\Pi}^{Gen}$ are computed, they undergo a compilation process. On the one hand, $P_{\Pi}^{Prop}$ is compiled by following the same algorithms described in Chapter 3, and this results in the `Propagator` module of our architecture. On the other hand, $P_{\Pi}^{Gen}$ is also compiled but in a different way. The behavior is similar to a deductive database system that computes all variable substitutions starting from input facts. To this end, the `Compiler` computes the Strongly Connected Components (SCCs) of $P_{\Pi}^{Gen}$, and process subprograms in the topological order induced by the SCCs. In particular, for each component $C_i$, the subprogram $\lambda_i = \{r \in P_{\Pi}^{Gen} : \mathcal{P}(H_r) \subseteq C_i\}$ is compiled as a procedure $proc\_\lambda_i(\cdot)$ that generates all ground atoms of the form $p(\cdot)$ such that $p$ occurs in $C_i$, and $p(\cdot)$ appears in at least one rule head.

In order to give the reader a more direct account of the procedures generated by the compiler, we present in the following an example of compilation.

**Algorithm 12** Sub-procedure generated for the rule $aux_{r_2}(X,Y,Z) \leftarrow d(X,Y),\ c(Y,Z),\ not\ e(Z)$

**Input** : A set of atoms $\mathcal{B}$, a set of atoms $F \subseteq \mathcal{B}$
**Output**: A set of ground atoms matching $aux_{r_2}(X,Y,Z)$

```
1  begin
2      atoms := ∅;
3      forall l1 ∈ {a ∈ B : match(a, d(·,·))} do
4          x := l1[0];        y := l1[1];
5          forall l2 ∈ {a ∈ B : match(a, c(y,·))} do
6              z := l2[1];        l3 := not e(z);
7              if l3 ∉ F
8                  atoms := atoms ∪ {aux_r2(x,y,z)}
9          done
10     done
11     B := B ∪ atoms;
```

**Algorithm 13** Propagator for $\leftarrow aux_{r_2}(X,Y,Z),\ not\ d(X,Y)$

```
   Input  : A literal l, an interpretation M
   Output: A set of literals M_l
1  begin
2      M_l := ∅;
3      if pred(l) = "aux_{r_2}" and l ∈ M^+
4          x := l[0];    y := l[1];    z := l[2];    l_2 := not d(x,y);
5          if l_2 ∉ M^+
6              M_l := M_l ∪ {l_2}
7      else
8          if pred(l) = "d " and l ∈ M^-
9              x := l[0];    y := l[1];
10             T := {a ∈ B : match(a, aux_{r_2}(x,y,·)}
11             forall l_1 ∈ T do
12                 M_l := M_l ∪ {l_1}
13             done
14     return M_l
```

In particular, the sub-procedure generated for the rule $aux_{r_2}(X,Y,Z) \leftarrow d(X,Y),\ c(Y,Z),\ not\ e(Z)$, referred to as $r$ in the following, is reported in pseudo-code as Algorithm 12. Specifically, Algorithm 12 first creates an empty set of ground atoms that will be used to collect all ground atoms of the form $aux_{r_2}(\cdot,\cdot,\cdot)$. Then, Algorithm 12 is made of a different nested block for each literal $l_i$ ($i = 1..|B_r|$) occurring in $B_r$, where the block is a **for loop** for each positive literal and an **if** statement for each negative literal. Indeed, Algorithm 12 contains three nested blocks, i.e. two **for loops** and one **if** statement, generated for $d(X,Y)$, $c(Y,Z)$, and $not\ e(Z)$, respectively. Then, when the scope of the **if** statement is reached, the set of literals $l_1$, $l_2$, and $l_3$ represents a possible instantiation of $r$. Thus, $aux_{r_2}(x,y,z)$ is added to the minimal subset of atoms needed for stable model computation. For instance, suppose that $\mathcal{B} = \{d(1,2), c(2,4), e(3)\}$, then we might have $l_1 = d(1,2)$, $l_2 = c(2,4)$, $l_3 = not\ e(4)$. Therefore, in this case, we can derive $aux_{r_2}(1,2,4)$.

Therefore, the `Generator` module is assembled as a sequence of sub-procedures as the ones described before, according to the SCCs of the program $P_{\Pi}^{Gen}$. The set of all atoms computed by this module becomes the set of decision variables that are given as input to the SAT solver. After the production of the `Generator` module, the `Compiler` creates the `Propagator`

**Algorithm 14** Propagator for $sup_{r_2}(X,Z) \leftarrow aux_{r_2}(X,Y,Z)$

**Input** : A literal $l$, an interpretation $M$
**Output**: A set of literals $M_l$

```
1  begin
2      M_l := ∅;
3      if pred(l) = "sup_{r_2}"
4          x := l[0];      z := l[1];
5          T := {a ∈ B : match(a, aux_{r_2}(x, ·, z)};
6          if l ∈ M^+
7              if T ∩ M = ∅ and |T| = 1
8                  M_l := M_l ∪ T
9          else if l ∈ M^-
10             M_l := M_l ∪ ¬T
11     else if pred(l) = "aux_{r_2}"
12         x := [0];      y := l[1];      z := l[2];
13         if l ∈ M^+
14             M_l := M_l ∪ {sup_{r_2}(x, z)}
15         else if l ∈ M^-
16             T := {a ∈ B : match(a, aux_{r_2}(x, ·, z)};
17             if ¬T ⊆ M^-
18                 M_l := M_l ∪ {not sup_{r_2}(x, z)} return M_l
```

module. Specifically, each rule $r \in P_{\Pi}^{Prop}$ is compiled into a propagator sub-procedure, called $Prop_r$ which evaluates possible propagations of $r$. In particular, two different propagation strategies have been implemented, respectively for constraints and normal rules. We refer the reader to [55] for the details about the generation of the sub-procedures. Here we show an example of the generated sub-procedures, reported as Algorithm 13 and Algorithm 14. In particular, Algorithm 13 is a propagator for the constraint $c : \leftarrow aux_{r_2}(X,Y,Z),\ not\ d(X,Y)$, whereas Algorithm 14 is a propagator for the rule $r : sup_{r_2}(X,Z) \leftarrow aux_{r_2}(X,Y,Z)$. Algorithm 13 evaluates propagations of ground instantiations of $c$ containing a literal $l$ that has been added to the candidate stable model $M$. Thus, if the predicate of $l$ is $aux_{r_2}$ and $l$ is a positive literal then $aux_{r_2}(X,Y,Z)$ is substituted by $l$, and $d(X,Y)$ is propagated. Otherwise, if the predicate of $l$ is $d$ and $l$ is a negative literal then $not\ d(X,Y)$ is replaced by $l$, and all literals of the form $not\ aux_{r_2}(X,Y,\cdot)$ are propagated. Analogously, Algorithm 14 evaluates propagations of ground instantiations of $r$ containing $l$ either in $H_r$ or $B_r$. Thus, if the predicate of $l$ is $sup_{r_2}$ then $sup_{r_2}(X,Z)$ is matched to $l$ by applying a proper variable substitu-

tion. If $l$ is a positive literal, then it means that there exists an instantiation of $r$, $r'$, such that $H_{r'}$ is true w.r.t. $M$. Thus, the propagator ensures that if only one literal is left such that $l' = aux_{r_2}(X, \cdot, Z)$ this is propagated to true. Instead, if $l$ is a negative literal, then there exists an instantiation of $r$, $r'$, such that $H_{r'}$ is false w.r.t. $M$. Thus, the propagator propagates all literals of the form $aux_{r_2}(X, \cdot, Z)$ as false (i.e., *not* $aux_{r_2}(X, \cdot, Z)$ is added to $M$). On the other hand, if the predicate of $l$ is $aux_{r_2}$ then $aux_{r_2}(X, Y, Z)$ is matched to $l$ by applying a proper variable substitution. Thus, if $l$ is a positive literal then it means that there exists an instantiation of $r$, $r'$, where $B_{r'}$ is true w.r.t. $M$. Thus, $H_{r'}$ is propagated as true. Otherwise, if $l$ is a negative literal and $sup_{r_2}(X, Z)$ is true w.r.t. $M$, the propagator ensures that if only one literal is left such that $l' = aux_{r_2}(X, \cdot, Z)$ this is propagated to true.

### 4.1.2 ProASP `Solver`

The `Solver` for an input program $\Pi$ is obtained by compiling (e.g., using g++), the SAT solver, the Generator, and the Propagator together in the same executable. The `Solver` reads as input an instance (set of facts) $F$, and, as the first step, it calls the Generator. The resulting set of ground atoms, say $\mathcal{B}$, is used to initialize the SAT solver data structures. A decision variable is added for each atom in $\mathcal{B}$, and a unit clause is added for each fact in $F$. At the same time, decision variables are mapped to the propagators they can affect. This mapping has a role similar to *watched literals* in standard SAT solvers, i.e., to limit iterations during propagation. For example, the watched literals for the propagator reported in Algorithm 13 are the literals of the form $aux_{r_2}(\cdot, \cdot, \cdot)$ and *not* $d(\cdot, \cdot)$. At this point, the SAT solver is started, and the CDCL search takes place as usual, alternating propagation and decision (cfr. Algorithm: 1). For example, consider the constraint $\leftarrow aux_{r_2}(X, Y, Z),\ not\ d(X, Y)$ and its propagator reported in Algorithm 13, and suppose that $aux_{r_2}(1, 2, 3)$ is added to $M$, then $d(1, 2)$ is derived (and later added to $M$). In case $M$ becomes inconsistent, the Learning procedure is invoked by the CDCL solver. During this process the propagator might be asked to reconstruct the clause that implied $M$, to be used as usual for learning a conflict clause [54]. In the previous example, suppose that *not* $d(1, 2)$ and $d(1, 2)$ are both in $M$, the set of literals causing their propagation, i.e., $aux_{r_2}(1, 2, 3)$ for $d(1, 2)$ is returned by the propagator. (For a detailed description of propagators we refer the reader to Chapter 3)

## 4.2 Experiments

In this section, we present the experimental results evaluating the performance of the ProASP system on grounding-intensive benchmarks. Specifically, we compare the proposed approach with state-of-the-art alternatives on existing benchmarks, and we devise synthetic benchmarks to analyze the strengths and weaknesses of ProASP.

ProASP has been compared with the following ASP systems:

- WaspProp v. cb67c17: the compilation-based approach proposed in the previous chapter where propagators are nested into the solver Wasp [3] and Gringo [39] is used as grounder. Here, due to the syntactical requirements that were necessary for successful compilation, the compilation can only be applied on constraints;
- Wasp v. d87f3f0: The plain version of Wasp solver that uses Gringo as grounder;
- Clingo v. 5.6.2: The plain version of Clingo [38];
- Alpha v. 0.7.0: The state-of-the-art lazy grounding system Alpha [66].

As for the benchmarks, we considered different grounding-intensive benchmarks from the literature, namely: Packing problem (P), Quasi Group (QG), Stable Marriage (SM), Non-Partition Removal Coloring (NPRC), and Weight Assignment Tree (WAT). The problem instances for benchmarks (P), (SM), (NPRC), and (WAT) were taken from previous studies, such as [21, 25, 41]. Moreover, for benchmark (SM), we extended the number of instances by generating novel ones with higher numbers of individuals and varying the percentage of expressed preferences as described in [25]. The (QG) problem consists of placing the numbers from 1 to $n$ into an $n \times n$ matrix, $M$, in such a way that each row and each column does not contain the same number twice. We generated different instances with different values of $n$ (i.e., from 50 to 1000), and for each value of $n$, we generated five instances by randomly initializing the matrix $M$ with a random sample of the set $\{1, \ldots, n\}$.

All experiments were executed on an Intel(R) Xeon(R) CPU E5-4610 v2 @ 2.30GHz running Debian Linux (3.16.0-4-amd64), with memory and CPU time (i.e. user+system) limited of 12GB and 1200 seconds, respectively, and each system was limited to run in a single core. We report that, in

| Benchmark | # | ProASP | | | WaspProp | | | Wasp | | | Clingo | | | Alpha | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | SO | TO | MO | SO | TO | MO | SO | TO | MO | SO | TO | MO | SO | TO | MO |
| (NPRC) | 110 | **110** | 0 | 0 | **110** | 0 | 0 | **110** | 0 | 0 | **110** | 0 | 0 | **110** | 0 | 0 |
| (P) | 50 | **23** | 27 | 0 | 12 | 38 | 0 | 0 | 50 | 0 | 0 | 48 | 2 | 0 | 45 | 5 |
| (QG) | 100 | **20** | 0 | 80 | 15 | 0 | 85 | 12 | 3 | 85 | 5 | 0 | 95 | 5 | 40 | 55 |
| (SM) | 314 | **230** | 84 | 0 | 225 | 89 | 0 | 197 | 117 | 0 | 213 | 4 | 97 | 28 | 286 | 0 |
| (WAT) | 62 | 36 | 14 | 12 | **50** | 0 | 12 | **50** | 0 | 12 | **50** | 0 | 12 | 0 | 62 | 0 |

Table 4.1: Comparison of the different solvers on grounding-intensive benchmarks.

all the considered benchmarks, the compilation time was negligible for both ProASP and WaspProp. Benchmarks and executables are available at `https://doi.org/10.5281/zenodo.8179111`.

### 4.2.1 Results

The results of the experimental evaluation are presented in Table 4.1 and Figures (4.2)–(4.13). Table 4.1 shows the total number of instances for each benchmark (#), and the number of instances solved (SO), the number of instances where the solver exceeded the given time (TO) or memory (MO) limits, for each benchmark and solver. Figures (4.2)–(4.13) include cactus

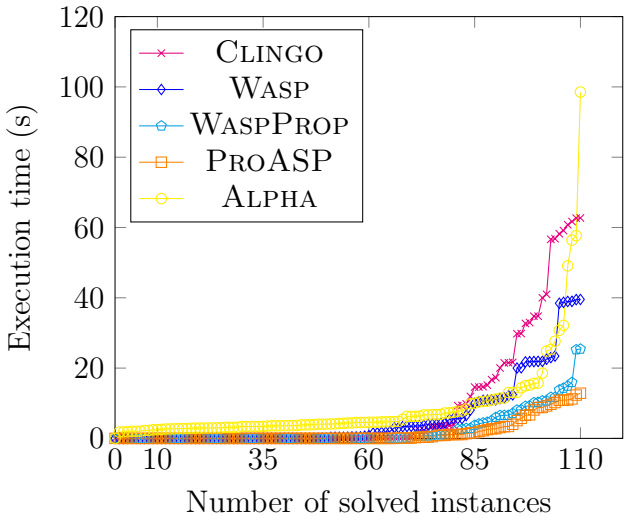


Figure 4.2: Solving time comparison on (NPRC).

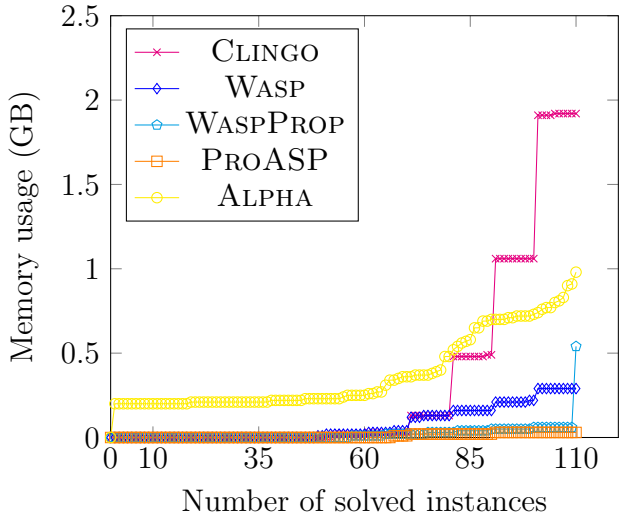


Figure 4.3: Memory usage comparison on (NPRC).

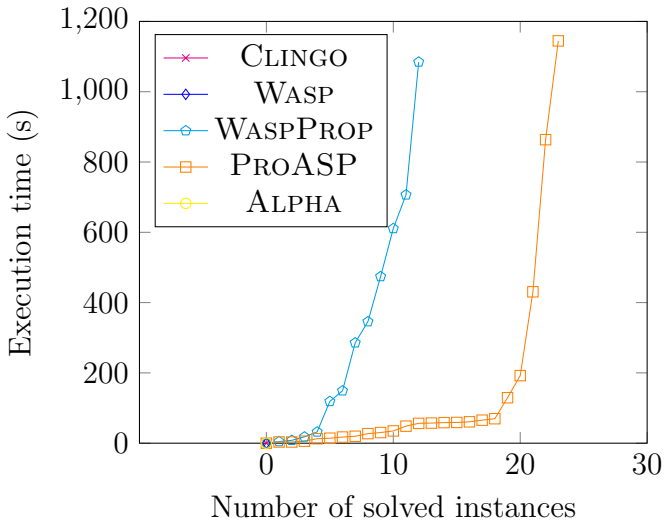


Figure 4.4: Solving time comparison on (P).

plots that depict the time and memory consumption for each benchmark and solver. We recall that, in a cactus plot, instances are sorted by memory (or time) usage, and a point $(i, j)$ indicates that a solver is capable of solving the $i$-th instance with a memory (or time) limit of $j$ gigabytes (or seconds).

In general, the results show that ProASP delivered the best performance

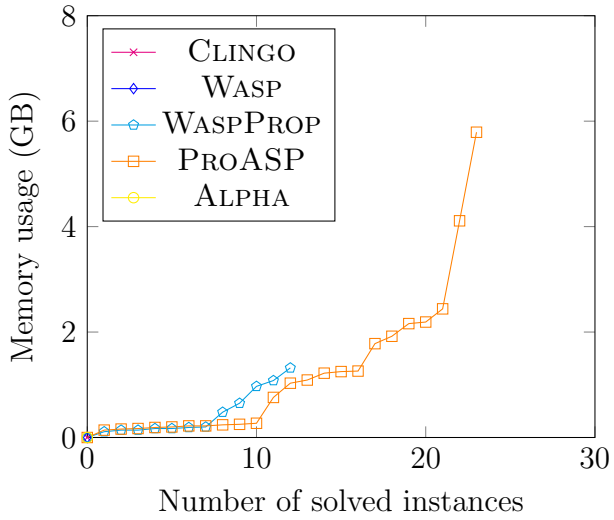


Figure 4.5: Memory usage comparison on (P).

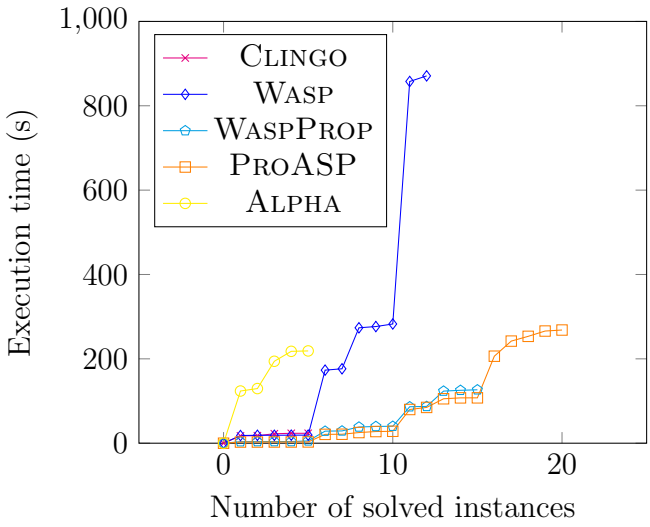


Figure 4.6: Solving time comparison on (QG).

by solving the highest number of instances and reducing memory consumption overall. The effectiveness of compilation-based approaches is evident in the performance of benchmarks (P), (QG), (SM), and (NPRC), where both ProASP and WaspProp outperformed Clingo and Wasp in both solving time (Figures 4.2, 4.4, 4.6, and 4.8) and memory usage (Figures 4.3, 4.5, 4.7,

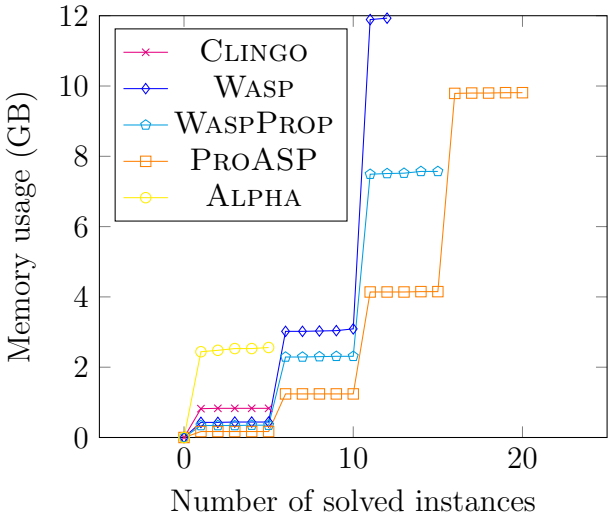


Figure 4.7: Memory usage comparison on (QG).

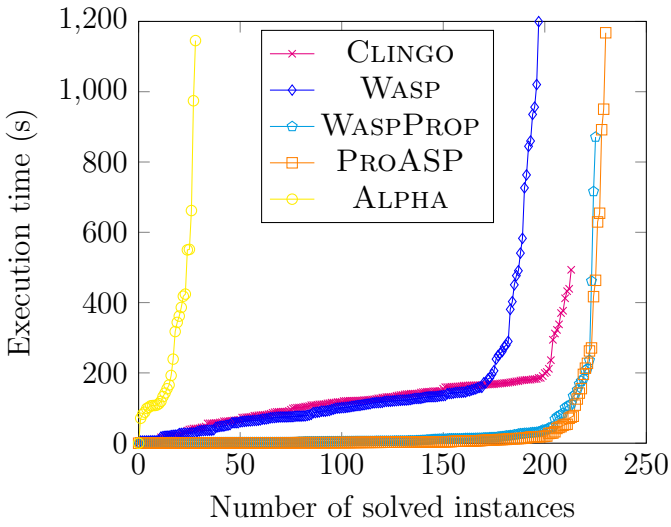


Figure 4.8: Solving time comparison on (SM).

and 4.9). Additionally, the ability to compile the entire program resulted in a significant improvement for ProASP. Specifically, ProASP solved 21 more instances than WaspProp while using less memory in the aforementioned benchmarks. In constrast, the (WAT) domain highlighted some overhead introduced by the usage of propagators. In particular, this is evident when

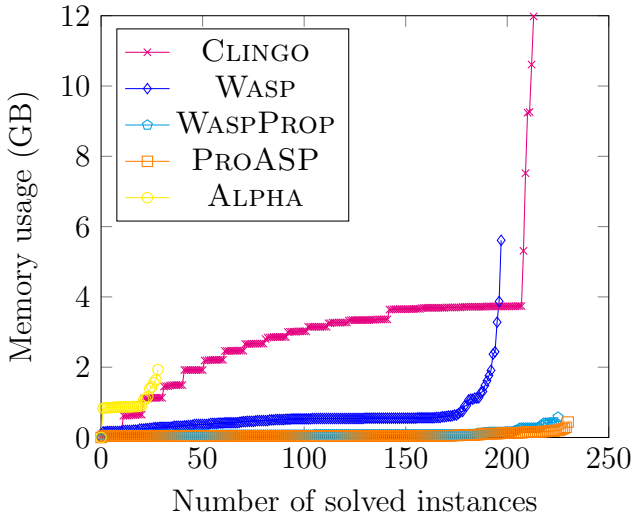


Figure 4.9: Memory usage comparison on (SM).

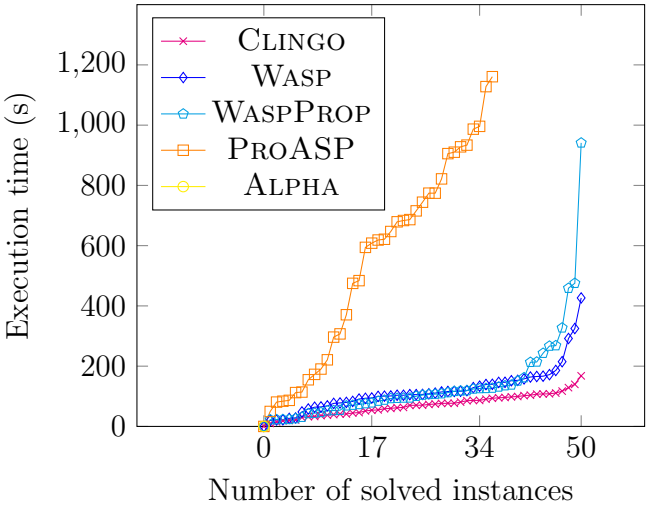


Figure 4.10: Solving time comparison on (WAT).

comparing the performances of Wasp with WaspProp. While the memory consumption is comparable, the execution time starts to increase more rapidly after the 40th instance (as shown in Figure 4.10). This overhead is even more pronounced in ProASP since the entire program is simulated by propagators. This drawback is partly due to a less informed heuristic that

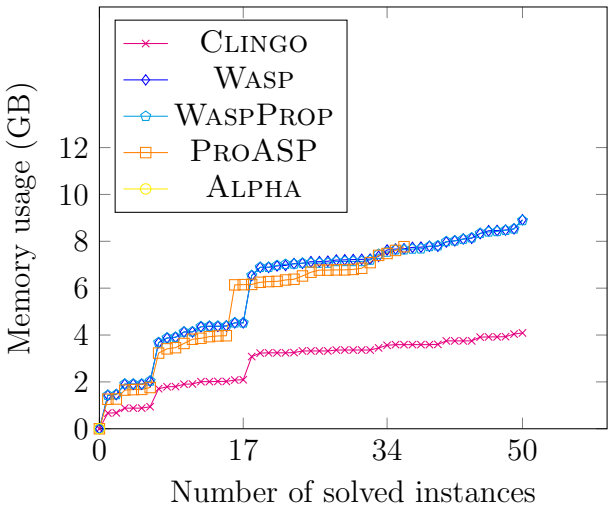


Figure 4.11: Memory usage comparison on (WAT).

guides the CDCL solver and also to a high number of auxiliary atoms introduced by our rewriting to simulate the program's completion. In this case Clingo proved to be the optimal choice for addressing this problem.

We also mention that ProASP outperformed the lazy-solver Alpha by solving more than 276 instances overall and significantly reducing memory consumption on solved instances. However, due to its design strategy, Alpha used less memory on unsolved instances.

**Synthetic benchmark.** To provide a more detailed investigation into the performance of our approach, we conducted an additional experiment on two synthetic benchmarks. This allowed us to highlight both the strengths and weaknesses of ProASP with respect to the system Clingo. The first benchmark, referred to as Synthetic (1), is designed to showcase scenarios where ProASP is expected to have an advantage. It involves encoding a join among six binary relations, denoted as $a_1, a_2, ..., a_6$, and is formulated as

follows:

$$
\begin{array}{ll}
a_1(X,Y) & \leftarrow d(X),\ d(Y),\ \mathit{not}\ na_1(X,Y) \\
na_1(X,Y) & \leftarrow d(X),\ d(Y),\ \mathit{not}\ a_1(X,Y) \\
\ldots & \\
a_6(X,Y) & \leftarrow d(X),\ d(Y),\ \mathit{not}\ na_6(X,Y) \\
na_6(X,Y) & \leftarrow d(X),\ d(Y),\ \mathit{not}\ a_6(X,Y) \\
b(X_1,\ldots,X_7) & \leftarrow a_1(X_1,X_2), a_1(X_2,X_3), \ldots, a_6(X_6,X_7) \\
c(X_1,\ldots,X_7) & \leftarrow a_1(X_1,X_2), a_1(X_2,X_3), \ldots, a_6(X_6,X_7) \\
 & \leftarrow b(\_,\_,\_,\_,X,Y,Z), c(Z,Y,X,\_,\_,\_,\_)
\end{array}
$$

The second benchmark, referred to as Synthetic (2), aims to highlight the limitation of ProASP that was already observed in the (WAT) benchmark. Indeed, it encodes a join between two relations of arity four and includes a projection over three terms:

$$
\begin{array}{ll}
a_1(X,Y,Z,W) & \leftarrow d(X),\ d(Y),\ d(Z),\ d(W), \\
 & \mathit{not}\ na_1(X,Y,Z,W) \\
na_1(X,Y,Z,W) & \leftarrow d(X),\ d(Y),\ d(Z),\ d(W), \\
 & \mathit{not}\ a_1(X,Y,Z,W) \\
a_2(X,Y,Z,W) & \leftarrow d(X),\ d(Y),\ d(Z),\ d(W), \\
 & \mathit{not}\ na_2(X,Y,Z,W) \\
na_2(X,Y,Z,W) & \leftarrow d(X),\ d(Y),\ d(Z),\ d(W), \\
 & \mathit{not}\ a_2(X,Y,Z,W) \\
b(X,Y,Z) & \leftarrow a_1(X,\_,\_,Y), a_2(Y,\_,\_,Z) \\
c(X,Y,Z) & \leftarrow a_1(X,\_,\_,Y), a_2(Y,\_,\_,Z) \\
 & \leftarrow b(X_1,Y,Z), c(Z,Y,X_2)
\end{array}
$$

For both benchmarks we generated 10 instances of increasing size by varying the number of atoms over the predicate $d$, from 3 to 12. Moreover, we increased the memory limit to 24GB, since the grounding phase is highly memory demanding, and we report only the memory consumption since the solving time after grounding is negligible.

As expected, obtained results highlighted a significant improvement introduced by ProASP in benchmark Synthetic (1) (Figure 4.12), where the rate at which memory usage increases in ProASP is significantly slower than that in Clingo, since, in this encoding, there are no projections. Instead, concerning benchmark Synthetic (2) (Figure 4.13), we observe that ProASP has a larger usage of memory, since storing the auxiliary atoms (i.e., tuples of

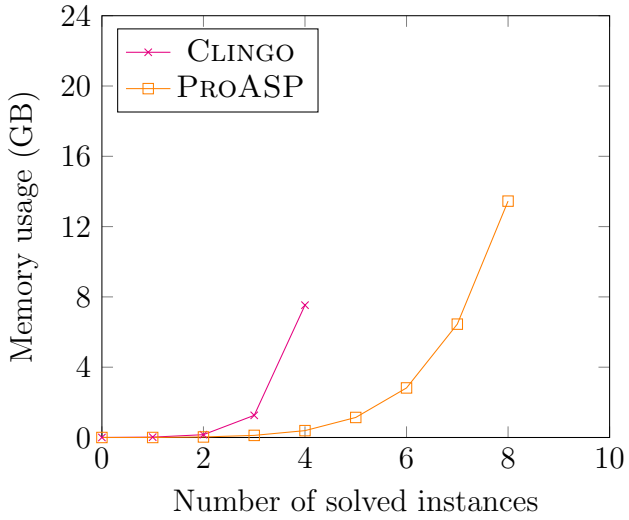


Figure 4.12: Comparison on Synthetic (1).

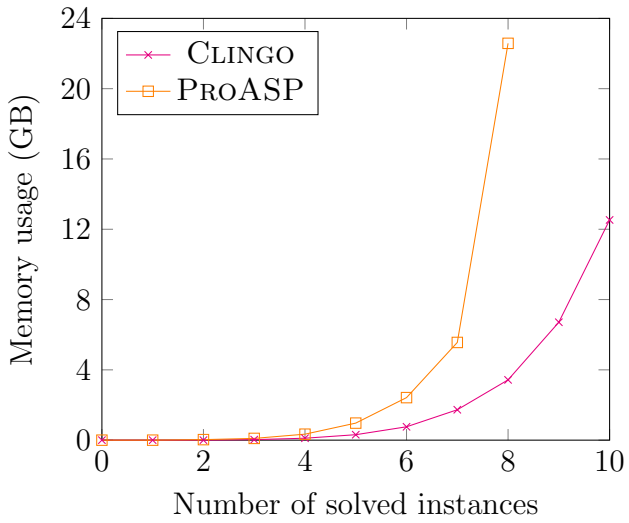


Figure 4.13: Comparison on Synthetic (2).

terms), together with indices used for computing joins, resulted to be heavier than storing ground program (where aux atoms cost as one integer) for large domains of the predicate $d$.

# Chapter 5

# Compilation of Normal Logic Programs Under Well-founded Semantics

In this chapter, we propose a compilation-based approach for the efficient evaluation of logic programs under well-founded semantics. More precisely, the proposed approach compiles non-ground normal logic programs into an ad-hoc solver that takes as input a program instance and computes the (partial) well-founded model. In order to assess the performances of the proposed approach, we conducted an experimental analysis on different classes of programs and obtained results highlight significant improvements with respect to compared systems in the evaluation of non-stratified programs.

## 5.1 Compilation of Well-founded semantics

Given a non ground program $P$, the compiler module (reported as Algorithm 15) generates a procedure that takes as input a set of facts $\mathcal{I}$ and returns the well-founded model of $P \cup \mathcal{I}$. As the first step, the compiler builds a graph $G = \langle V, E \rangle$ where $V = \mathcal{P}(P)$, and $E$ contains an edge $(u, v)$ if there exists a rule $r \in P$ such that $v \in \mathcal{P}(H_r)$ and $u \in \mathcal{P}(B_r)$. Note that $G$ resorts the dependency graph of $P$ but it is built on predicates of $P$. Then, the strongly connected components (SCCs) of $G$, referred to as $C_1, ..., C_n$, are computed. SCCs give us a topological order of $G$ such that no paths exist from $C_j$ to $C_i$ if $i < j$.

By following this order, the compiler produces, for each component $C$, the code that evaluates the rules defining atoms whose predicate belongs

**Algorithm 15** compileProgram

**Input** : A normal program $P$
**Output**: Prints evaluation procedure for $P$

```
1  begin
2      ≪def computeWellFoundedModel(I)≫
3      ≪   I_T = I≫
4      ≪   I_U = ∅≫
5      G = computeG(P)    SCC = computeSCC(G)
6      for all C ∈ SCC do
7          ≪d_stack = I_F = ∅≫
8          for all r ∈ rulesForComponent(P, C) do
9              compileHeadDerivation(B_r, H_r, C)
10         ≪while d_stack ≠ ∅ do≫
11         ≪   starter = d_stack.pop()≫
12         ≪   switch P(starter)≫
13          for all r ∈ recursiveRules(P, C) do
14             for all i ∈ [1, ..., |B_r|] do
15                 if P(B_r[i]) ∈ C ∧ B_r[i] is positive literal then
16                     ≪case [[P(B_r[i])]]_t ≫
17                     ≪   σ = ϵ≫
18                     for all j ∈ 1, ..., |trm(B_r[i])| do
19                         if trm(B_r[i])[j] is variable then
20                             ≪   σ = σ ∪ { [[trm(B_r[i])[j]]]_t ↦
                               trm(starter)[ [[j]]_t ]}≫
21                     compileHeadDerivation(B_r \ {B_r[i]}, H_r, C)
22         ≪done≫
23         ≪do≫
24         ≪   loop = ⊥≫
25         ≪   for starter ∈ B do≫
26         ≪      true = undef = ⊥     starter = s_stack.pop()≫
27         ≪      switch P(starter)≫
28         compileSupportDerivation(recursiveRules(P, C), C)
29         ≪while loop == ⊤≫
30     ≪end def≫
```

**Algorithm 16** compileHeadDerivation

**Input** : A list of literals $B$, an atom $a$, a set of predicates $C$
**Output**: Prints a procedure that instantiates $B$ and derives new atoms matching $a$

```
1 begin
2     compileBody(B, C)
3     compileHead(B, a)
4     for all i ∈ [|B|, ..., 1] do
5         if B[i] is positive literal then
6             ≪  σ = σ ⟦j⟧_i ≫
7             ≪done≫ //closing for loop scope
```

to $C$, referred to as $P_C$. Thus, each rule $r \in P_C$ is compiled into a sub-procedure that iterates over possible instantiations of $B_r$ that are either true or undefined w.r.t. $I_T \cup I_U$ and derives $H_r$ (Algorithm 15 lines 8-9).

Such sub-procedures are generated by Algorithm 16 that starts by calling Algorithm 17 that prints different nested for-loops or if-statements according to literals in $B_r$ (Algorithm 17 lines 3-13). These nested blocks, in the resulting code, will iterate over possible instantiations of $r$. Inside the deepest generated block, the code that collects negative dependencies within the component $C$ into the set $ns$ is printed (Algorithm 17 lines 14-18). Then, Algorithm 16 prints the code that derives new atoms matching $H_r$ by calling Algorithm 18. The code generated by Algorithm 18 checks if the current body (i.e., $b = \{t_1, \cdots, t_n\}$) is true w.r.t. $I_T \cup I_U$. That is, if all positive literals are true (i.e., $b^+ \in I_T$), no negative literals are undefined (i.e., $\neg b^- \cap I_U == \emptyset$), and no negative literals in the same component occur in the current body (i.e., $ns = \emptyset$), then $\sigma(a)$, with $a = H_r$, is derived as true, otherwise it is derived as undefined (code generated by lines 3 and 10 of Algorithm 18). In both cases, derived atoms are collected into the derivation stack in order to be used in the second derivation phase. As the last step of Algorithm 16, for each literal $l \in B_r$, if $l$ is a positive literal then a nested for-loop is closed by restoring the variable substitution $\sigma$. The second derivation step is used to simulate a recursive derivation starting from newly derived atoms. Thus, the code implementing such derivation is generated by looking at recursive rules defining atoms whose predicate belongs to $C$ (i.e., rules comprising atoms over predicate in C both in the head and the positive body). In this case, the generated procedure will consume literals collected into *d_stack* and, for each of them, different sub-procedures are executed according to the predi-

**Algorithm 17** compileBody

**Input** : A list of literals $B$, a set of predicate $C$
**Output**: Prints the instantiation procedure for the rule body $B$

```
1  begin
2      ≪σ = ϵ≫
3      for all j ∈ 1, ..., |B| do
4          ≪σ ⟦j⟧ᵢ = σ≫
5          if B[j] is a positive literal then
6              ≪T ⟦j⟧ᵢ = {p ∈ (I_T ∪ I_U) | match(σ( ⟦B[j]⟧ᵢ), p)}≫
7              ≪forall t ⟦j⟧ᵢ ∈ T ⟦j⟧ᵢ do≫
8              for all k ∈ 1, ..., |trm(B[j])| do
9                  if trm(B[j])[k] is variable then
10                     ≪σ = σ ∪ { ⟦trm(B[j])[k]⟧ᵢ ↦ trm(t ⟦j⟧ᵢ)[ ⟦k⟧ᵢ]}≫
11         else
12             ≪t ⟦j⟧ᵢ = σ( ⟦B[j]⟧ᵢ)≫
13             ≪if not t ⟦j⟧ᵢ ∉ I≫
14     ≪b = ns = ∅≫
15     for all j ∈ 1, ..., |B| do
16         ≪b = b ∪ {t ⟦j⟧ᵢ}≫
17         if B[j] is a negative literal ∧ P(B[j]) ∈ C then
18             ≪ns = ns ∪ {t ⟦j⟧ᵢ}≫
```

**Algorithm 18** compileHead

**Input** : An atom $a$
**Output**: Prints the derivation procedure for new atoms matching $a$

```
1  begin
2      ≪h = σ( ⟦a⟧ᵢ)≫
3      ≪if ns == ∅ ∧ b⁺ ⊆ I_T ∧ (¬b⁻ ∩ I_U) == ∅ ≫
4      ≪   d_stack = d_stack ∪ {h}≫
5      ≪   I_T = I_T ∪ {h}≫
6      ≪   I_U = I_U \ {h}≫
7      ≪else≫
8      ≪   if h ∉ (I_T ∪ I_U)≫
9      ≪      d_stack = d_stack ∪ {h}≫
10     ≪      I_U = I_U ∪ {h}≫
```

cate name of the consumed literal (Algorithm 15 lines 10-22). In particular, the compiler, for each recursive rule $r$, generates different switch-cases for each literal $l \in B_r^+$ whose predicate belongs to $C$ (Algorithm 15 lines 15-21).

Inside each case, a sub-procedure that evaluates $r$ starting from a literal, *starter*, that can matches $l$ via a variable substitution is generated. Each sub-procedure initializes of a variable substitution $\sigma$ from variables in $l$ to constants in *starter* (Algorithm 15 lines 18-20) and contains the code that evaluates the remaining part of $B_r$, generated by Algorithm 16.

In this way, the generated procedure is able to simulate a semi-naive evaluation of recursive rules. Out of the while-loop scope, all those atoms $a \in B_\Pi$ such that $\mathcal{P}(a) \in C$ and $a$ is neither true nor undefined w.r.t. $I_T \cup I_U$ can be derived as false since no rule instantiations that can derive them exists. The next step consists in deriving as false all those atoms $a \in I_U$ such that each rule instantiations with $a$ in the head has a false body w.r.t. $I_T \cup I_F$, and as true all those atoms $a \in I_U$ such that there exists a rule instantiations having $a$ in the head and a true body w.r.t. $I_T \cup I_F$, until a fixed point is reached. To this end, the compiler generates a procedure that iterates atoms $a \in I_U$ and evaluate all those rule instantiations having $a$ in the head and, if it is possible, derives $a$ either as true or false (Algorithm 15 lines 23-29). Such procedure is made of different switch-cases, one for each rule $r$, that are generated by Algorithm 19. Each case contains a sub-procedure that evaluates a rule $r$ starting from an undefined atom *starter* that can be substituted to $H_r$ via a variable substitution. These sub-procedures contains the code that initializes a variable substitution $\sigma$ from variables in $H_r$ to constants in *starter* (generated by Algorithm 19, lines 4-7) and different nested blocks for evaluating $B_r$ that are generated by Algorithm 17. Then, inside the deepest block, the code for the head derivation is printed (Algorithm 19 lines 9-15). If the conjunction $b = t_1, \ldots, t_n$, with $n = |B_r|$, is true w.r.t. $I_T \cup I_F$ (i.e. all positive literals are in $I_T$ and all atoms appearing in some negative literals are not in $I_U$, Algorithm 19 line 10) then *starter* is derived as true, otherwise a flag variable stating that there exists a ground rule with *starter* as head and the body undefined w.r.t. $I_T \cup I_F$ is enabled (Algorithm 19 line 14). Then, nested blocks' scopes are closed (Algorithm 19 lines 16-19). After evaluating all switch-cases, if neither *starter* has not been derived as true nor the $undef$ flag is true then *starter* is derived as false (Algorithm 19 lines 20-21).

## 5.2 Example of compilation

In order to help understanding the outcome of our compiler, in this section we describe the ad-hoc solver generated for the following program Π:

$$\begin{aligned} r1: &\quad a(Y) \leftarrow b(X,Y), c(Y,Z), not\ d(Z) \\ r2: &\quad a(X) \leftarrow f(X),\ not\ g(X) \\ r3: &\quad g(X) \leftarrow e(X),\ not\ a(X) \end{aligned}$$

The compiler computes the graph $G$ reported in figure 5.2. The SCCs of $G$ are $C_0 = \{b\}$, $C_1 = \{c\}$, $C_2 = \{d\}$, $C_3 = \{f\}$, $C_4 = \{e\}$, $C_5 = \{a, g\}$. Since for components $C_i$, with $i$ from 0 to 4, there are no rules defining them, the code produced for such components is empty. Therefore, here we focus on $C_5$. In this case, $P_{C_5}$ is the entire program. Algorithm 15 prints a sub-procedure for each rule of Π by means of Algorithm 16, which produces the code reported as Algorithm 20 when executed with input $r1$.

For evaluating $B_{r1}$, Algorithm 17 have generated an external for-loop that iterates over ground literals $t_1$ that are either true (i.e., literals in $I_T$) or undefined (i.e., literals in $I_U$) and can be substituted to $b(X, Y)$ (Algorithm 20 lines 3-4). For every $t_1$, the variable substitution $\sigma$ is updated mapping $X$ and $Y$ to the first and the second term of $t_1$ respectively (Algorithm 20 line 5). Nested into this for-loop, another for-loop has been printed to iterate over ground literals $t_2$ matching $\sigma(c(Y, Z))$ (Algorithm 20 lines 6-7). Note that the application of $\sigma$ to a literal will replace mapped variables with the value they are mapped to (i.e., $Y \mapsto 1$ then $\sigma(c(Y, Z)) = c(1, Z)$). Inside this for-loop $\sigma$ is updated by mapping $Z$ to the second term of $t_2$ (Algorithm 20 line

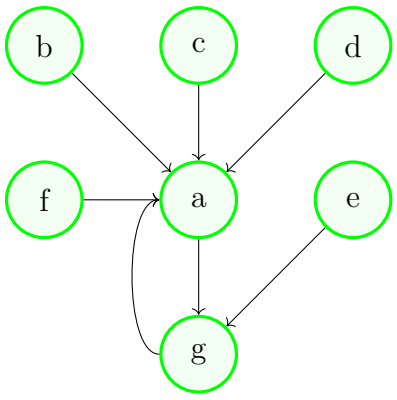


Figure 5.1: Dependency Graph for program Π

**Algorithm 19** compileSupportDerivation

**Input** : A set of rules *recursiveRules*, a set of predicates $C$
**Output**: Prints a procedure that search for rule body that can support a given atom

```
1  begin
2      for all r ∈ recursiveRules do
3          ≪ case [[P(H_r)]]_i ≫
4          ≪     σ = ϵ≫
5          for all j ∈ 1, ..., |trm(H_r)| do
6              if trm(H_r)[j] is variable then
7                  ≪  σ = σ ∪ { [[trm(H_r)[j]]]_i ↦ trm(starter)[ [[j]]_i ]}≫
8          compileBody(B_r, C)
9          ≪          h = σ( [[a]]_i )≫
10         ≪          if b+ ⊆ I_T ∧ (¬b− ∩ I_U) = ∅ ∧ h ∉ I_T ≫
11         ≪             I_T = I_T ∪ {h}≫
12         ≪             I_U = I_U \ {h}≫
13         ≪             true = loop = ⊤≫
14         ≪          else if ((b+ ∪ ¬b−) ∩ I_U) ≠ ∅ ∧ h ∉ I_T ≫
15         ≪             undef = ⊤≫
16         for all i ∈ [|B_r|, ..., 1] do
17             if B_r[i] is positive literal then
18                 ≪  σ = σ [[j]]_i ≫
19                 ≪done≫
20     ≪  if undef = ⊥ ∧ true = ⊥≫
21     ≪     I_U = I_U \ {starter}   F = F ∪ {starter}   loop = ⊤≫
```

8). Then, the last literal to evaluate is *not* $d(Z)$. Since it is a negative literal the last nested block is an if-statement that checks whether $\overline{t_3} = \sigma(d(Z))$ is not true w.r.t. $I_T$ (Algorithm 20 line 10).

Note that the body of each rule is reordered to ensure that negative literals occur after positive ones. Thus, for the safety condition each variable occurring in some negative literal is mapped to a constant by $\sigma$.

So, the conjunction $b = t_1,\ t_2,\ t_3$ is an instantiation of $B_r$ and the generated solver should derive $\sigma(a(Y))$ (Algorithm 20 lines 12-17). Thus, if $b$ is true w.r.t. $I_T \cup I_F$ and the set of negative dependencies $ns$ is empty, then $h = \sigma(a(Y))$ is added to $I_T$, otherwise it is added to $I_U$.

The generated solver contains other two similar procedures also for $r2$ and $r3$. Moreover, since $P_{C_5}$ does not contain recursive rules no switch-cases are generated at all. Therefore, the while-loop over *d_stack* is omitted in the

**Algorithm 20** Example of code generated by Algorithm 16

```
   Input  : {b(X,Y), c(Y,Z), not d(Z)}, a(Y), {'a', 'g'}
 1 begin
 2   σ = ϵ   σ1 = σ
 3   T1 = {p ∈ (IT ∪ IU) | match(σ(b(X,Y)), p)}
 4   forall t1 ∈ T1 do
 5     σ2 = σ   σ = σ ∪ {X ↦ trm(t1)[1]}   σ = σ ∪ {Y ↦ trm(t1)[2]}
 6     T2 = {p ∈ (IT ∪ IU) | match(σ(c(Y,Z)), p)}
 7     forall t2 ∈ T2 do
 8       σ3 = σ   σ = σ ∪ {Y ↦ trm(t2)[1]}   σ = σ ∪ {Z ↦ trm(t2)[2]}
 9       t3 = σ(not d(X))
10       if t̄3 ∉ IT
11         b = ns = ∅   b = b ∪ {t1}   b = b ∪ {t2}   b = b ∪ {t3}
12         h = σ(a(Y))
13         if ns == ∅ ∧ b+ ⊆ IT ∧ (¬b− ∩ IU) = ∅
14           d_stack = d_stack ∪ {h}   IT = IT ∪ {h}   IU = IU \ {h}
15         else
16           if h ∉ (IT ∪ IU)
17             d_stack = d_stack ∪ {h}   IU = IU ∪ {h}
18       σ = σ3
19     done
20     σ = σ2
21   done
```

generated code. The subsequent part of the code contains the procedure for deriving true and false atoms until the fixed point.

As described before, such procedure iterates over each atom $a$ in $I_U$ and, for each of them, evaluate different switch-cases according to the predicate of $a$. An example of switch-cases produced for rules $r_2$ and $r_3$ w.r.t. the component $C_5$ is reported in Algorithm 21.

The first switch-case has been generated for rule $r2$ (Algorithm 21 lines 2-17), and so it contains the code for evaluating $r2$ starting from a ground atom *starter* that can be substituted to $a(X)$ via a variable substitution $\sigma$. Thus, the solver initializes $\sigma$ by mapping $X$ to the first term of *starter* (Algorithm 21 line 3), and then evaluates the instatiations of $B_{r2}$ by means of the nested blocks generated by Algorithm 17 (Algorithm 21 lines 4-8). Inside the last block, the head derivation code has been printed (Algorithm 21 lines

**Algorithm 21** Example of code generated by Algorithm 19

```
   Input  : {g(X) ← e(X), not a(X); a(X) ← f(X), not g(X)}, {'a','g'}
 1 begin
 2     case 'a'
 3       σ = ϵ   σ1 = σ   σ = σ ∪ {X ↦ trm(starter)[1]}
 4       T1 = {p ∈ (I ∪ B) | match(σ(f(X)), p)}
 5       forall t1 ∈ T1 do
 6         σ2 = σ   σ = σ ∪ {X ↦ trm(t1)[1]}
 7         t2 = σ(not g(X))
 8         if t̄2 ∉ I
 9           b = ns = ∅    b = b ∪ {t1}   b = b ∪ {t2}
10           ns = ns ∪ {t2}
11           h = starter
12           if b+ ⊆ I ∧ (¬b− ∩ B) = ∅ ∧ h ∉ I
13             I = I ∪ {h}    B = B \ {h}    true = loop = ⊤
14           else if ((b+ ∪ ¬b−) ∩ B) ≠ ∅ ∧ h ∉ I
15             undef = ⊤
16         σ = σ2
17       done
18     case 'g'
19       σ = ϵ   σ1 = σ    σ = σ ∪ {X ↦ trm(starter)[1]}
20       T1 = {p ∈ (I ∪ B) | match(σ(e(X)), p)}
21       forall t1 ∈ T1 do
22         σ2 = σ
23         σ = σ ∪ {X ↦ trm(t1)[1]}
24         t2 = σ(not a(X))
25         if t̄2 ∉ I
26           b = ns = ∅    b = b ∪ {t1}   b = b ∪ {t2}
27           ns = ns ∪ {t2}
28           h = starter
29           if b+ ⊆ I ∧ (¬b− ∩ B) = ∅ ∧ h ∉ I
30             I = I ∪ {h}    B = B \ {h}    true = loop = ⊤
31           else if ((b+ ∪ ¬b−) ∩ B) ≠ ∅ ∧ h ∉ I
32             undef = ⊤
33         σ = σ2
34       done
35   if undef = ⊥ ∧ true = ⊥
36     B = B \ {starter}    loop = ⊤
```

12-15). If the body instantiation $t_1,\ t_2$ is true w.r.t. $I_T \cup I_F$, then $h = starter$ is derived as true, otherwise the $undef$ flag is enabled. The second switch-case is analogous to the first one but it refers to $r3$ and so the evaluation

starts from a literal, *starter*, matching the atom $b(X)$. When the execution reaches the end of the switch-statement, if no rule instantiations have been found for the atom *starter* then it is derived as false (Algorithm 21 lines 35-36).

## 5.3 Implementation and Experiments

### 5.3.1 Implementation details

The compilation strategy described in the previous section has been entirely implemented in C++ and so both compiler and generated procedures are written in C++. Generated code is built on top of optimized data structures that allow to speed up the whole computation process. In particular, it uses a numerical representation of constant terms that allows a compact and uniform representation. Moreover, different indexing structures are used for each predicate. Indexes are defined on the subset of terms of predicates, for fast retrieval of the list of literals that match a possible tuple. The resulting tool is available online and can be found at `https://github.com/ndria00/HebBaseGen.git`

### 5.3.2 Benchmarks, Systems and Experiments setup

In order to assess the performances of the proposed approach an empirical evaluation has been conducted both on positive programs (i.e., Datalog) and programs with negation. Among positive programs, the following two problems have been considered:

- Large-join problem [50] defined as follows:

$$\begin{aligned} a(X,Y) &\leftarrow b1(X,Z),\ b2(Z,Y).\\ b1(X,Y) &\leftarrow c1(X,Z),\ c2(Z,Y).\\ b2(X,Y) &\leftarrow c3(X,Z),\ c4(Z,Y).\\ c1(X,Y) &\leftarrow d1(X,Z),\ d2(Z,Y). \end{aligned}$$

  where $d1/2$, $d2/2$, $c2/2$, $c3/2$, and $c4/2$ are defined as facts that represent an instance of the problem. For this benchmark, instances of different sizes have been generated in a random fashion. More precisely, for each instance, we set the size (number of facts) roughly from

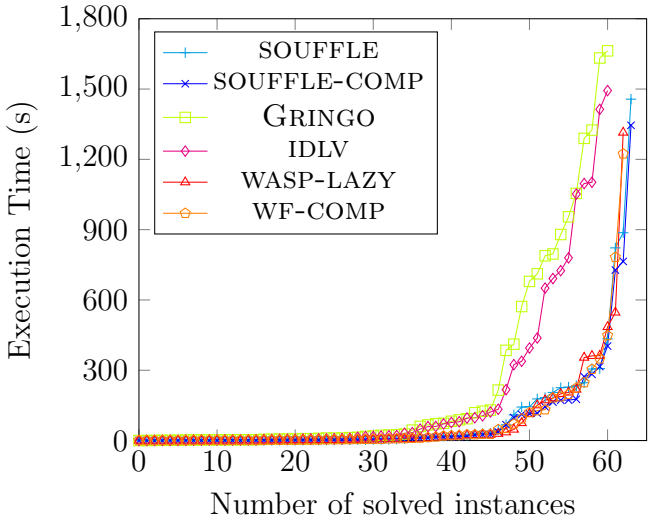


Figure 5.2: Systems comparison on large-join domain

10000 to 10000000 and we randomly divided the number of facts among the previous predicates (around 10-25% for each predicate). Then, for each predicate $p$, we randomly estimated the max value for each term, $n$ and $m$, in such a way that $n * m = s$, where $s$ is the size of the predicate set of $p$.

- Reachability problem defined as follows:

$$reach(X, Y) \leftarrow edge(X, Y).$$
$$reach(X, Y) \leftarrow reach(X, Z),\ edge(Z, Y).$$

  Instances of this problem are directed graphs that have been generated by varying the number of nodes and the density of the edges. In particular, graphs with a number of nodes from 100 to 2000 and density 20, 40, 60, 80, and 100% have been considered.

Among programs with negation, instead, the following three hard benchmarks from asp competitions have been considered: *Knight Tour with Holes*, *Stable Marriage*, and *Graph Colouring* [42].

The proposed approach, labeled WF-COMP, was compared with the following tools:

- General-purpose systems that can evaluate Datalog programs: IDLV [19] and GRINGO [39].

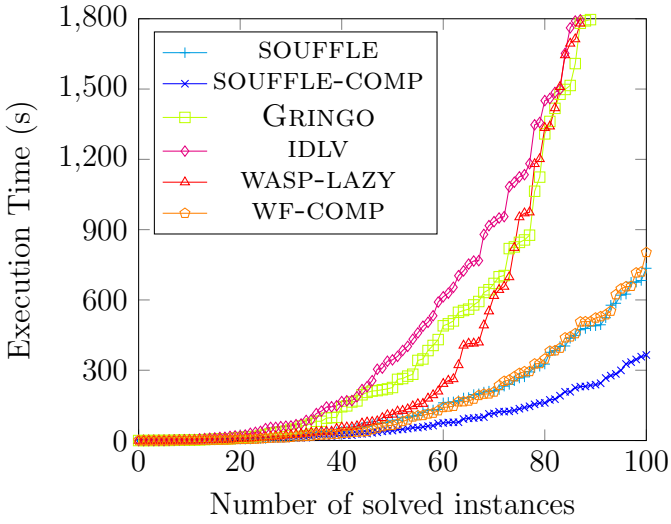


Figure 5.3: Systems comparison on reachability domain

- The soufflé framework [47] for which we have rewritten the encoding to produce a suitable encoding for this framework. In particular, two versions have been considered, the interpreted, SOUFFLE, and compiled, SOUFFLE-COMP, ones.
- Compilation-based approach for stratified normal programs WASP-LAZY [28].
- ASP system DLV2 [2] that can evaluate normal programs under well-founded semantics.

All the experiments were executed on a machine equipped with Xeon(R) Gold 5118 CPUs, running Ubuntu Linux (kernel 5.4.0-77-generic). Time and memory were limited to 1800 seconds and 8GB, respectively. Source code and benchmark suite are available at `https://doi.org/10.5281/zenodo.8212595`.

### 5.3.3 Obtained Results

**Evaluation on positive programs.** In this comparison, we run the systems on instances of large-join and reachability. Obtained results are summarized by the cactus plots in Figures 5.2-5.3. Recall that a cactus plot

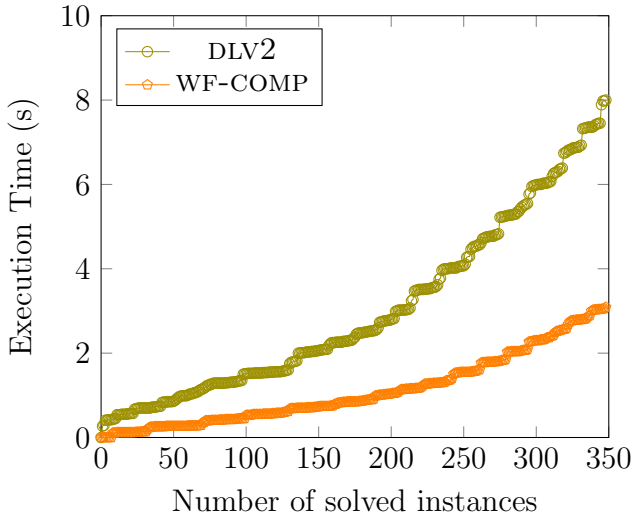


Figure 5.4: Systems comparison on Knight Tour With Holes benchmark

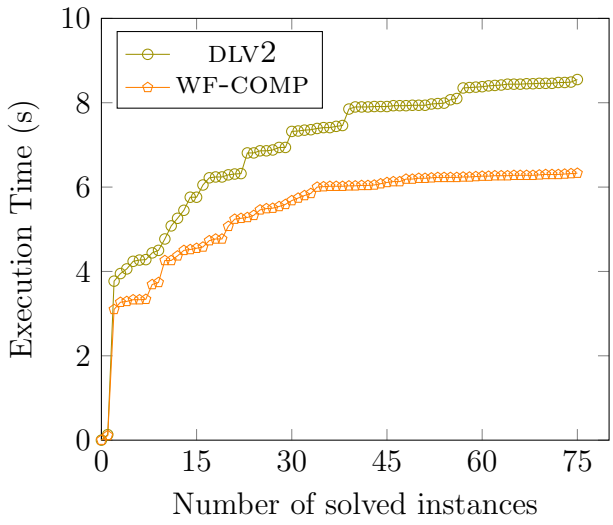


Figure 5.5: Systems comparison on Stable Marriage benchmark

reports a line for each system and each line contains a point $(X, Y)$ if a given system is able to solve $X$ instances within a time limit of $Y$ seconds.

In both cases, WF-COMP outperforms state-of-the-art ASP systems GRINGO and IDLV solving more instances (roughly 15 for reachability and 2 for large-join) and in less time overall. Moreover, WF-COMP outperforms also WASP-

LAZY on reachability problem solving 16 more instances, while it is comparable on the large-join domain with WASP-LAZY. The best method in this comparison is SOUFFLE system. The difference with the proposed tool (SOUFFLE is preferable in complete graphs) is due to different data structures and also to the different input formats. Indeed, SOUFFLE takes as input a numeric format where input facts are organized in files for each input predicate while WF-COMP reads plain text files.

**Evaluation on programs with negation.** In the case of programs with negation, we compare WF-COMP and DLV2 (the other methods do not support unrestricted negation). For each considered benchmark we report a cactus plot see Figures 5.4, 5.5, and 5.6. Obtained results highlight the strength of the proposed approach that outperforms DLV2 on considered benchmarks. In particular, both systems are able to solve all problem instances within time and memory limits, but WF-COMP significantly reduced the total execution time for each benchmark (33.24% on Graph Colouring, 63.26% on Knight Tour With Holes, and 22.37% on Stable Marriage).

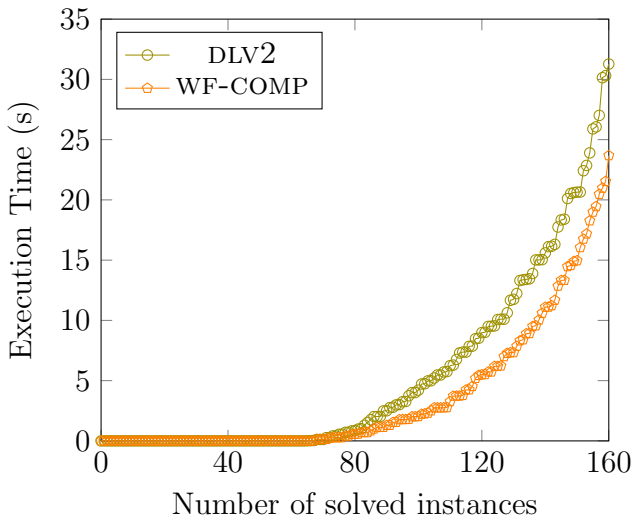


Figure 5.6: Systems comparison on Graph Colouring benchmark

# Part II

# PyQASP: An efficient solver for ASP(Q)

# Chapter 6

# Preliminaries on Quantified Answer Set Programming

In this chapter, we provide the foundational concepts necessary for the presentation of the new solver, which will be described in this part of the thesis. More precisely, we will introduce both syntax and semantics of Quantified Boolean Formulas (QBF) and Answer Set Programming with Quantifiers (ASP(Q)). Additionally, we will describe the encoding of ASP(Q) programs into QBF formula employed by the current implementation of ASP(Q) (i.e. the QASP system) that will be the core of our contribution.

## 6.1 Quantified Boolean Formula

Quantified Boolean formulas expand upon propositional formulas by incorporating universal and existential quantifiers, denoted by $\forall$ and $\exists$ respectibely. This is a natural extension of propositional formulas that allows to model problems of higher computational complexity. For example the problem of logical equivalence of two propositional formulas $\phi_1$ and $\phi_2$, can be modeled as $\forall x_1, \ldots, x_n(\phi_1 \leftrightarrow \phi_2)$. Intuitively, this formula can be read as "for every possible values (i.e., true ($\top$) or false ($\bot$)) of variables $x_1, \ldots, x_n$, $\phi_1$ is satisfied if and only if $\phi_2$ is also satisfied".

**Syntax.** A *Boolean literal* is a propositional variable $x$ or its negation $\neg x$; a *Boolean formula* $\phi$ is obtained by composing Boolean literals by logical connectives ($\wedge$, $\vee$, $\neg$, $\leftarrow$, $\leftrightarrow$). A Boolean formula in conjunctive normal

form (CNF) is a Boolean formula of the form $C_1 \wedge \ldots \wedge C_n$, where each $C_i$, with $1 \leq i \leq n$, is disjunction of Boolean literals (i.e., $C_i = l_1^i \vee \ldots \vee l_{n_i}^i$), referred to as *clause*. The set of Quantified Boolean formulas can be defined inductively as follows:

1. each Boolean formula is also a QBF formula;
2. given a QBF formula $\Phi$ then the formulas $\exists x\ \Phi$ and $\forall x\ \Phi$ are QBF formulas;
3. given a QBF formula $\Phi$ then the formula $\neg\Phi$ is a QBF formula; and
4. given two Quantified Boolean formulas, $\Phi$ and $\Psi$, then the formula $\Phi \circ \Psi$ with $\circ \in \{\wedge, \vee, \neg, \leftarrow, \leftrightarrow\}$ is a QBF formula.

For a QBF formula of the form $\exists x\ \Phi$ (resp. $\forall x\ \Phi$), $\exists x$ (resp. $\forall x$) is referred to as *prefix* whereas $\Phi$ is referred to as *scope*. A QBF formula is in *prenex normal form* if it is of the form $Q_1, \ldots, Q_n\phi$, where each $Q_i$ is a quantifier prefix, and $\phi$ is a Boolean formula. Without losing of generality in what follows we refer to QBF formulas as QBF formulas in prenex normal form.

**Semantics.** Given a QBF formula $\Phi$, $var(\Phi)$ denotes the set of variables appearing in $\Phi$; each variable $x \in var(\Phi)$, is *bound* if it appears in some prefix in $\Phi$, otherwise $x$ is free. Given a QBF formula $\Phi$, $free(\Phi)$ denotes the set of free variables in $\Phi$. A QBF formula $\Phi$ is *closed* if $free(\Phi) = \emptyset$. A *variable assignment* is a mapping from variables to Boolean constants $\top$ (true) or $\bot$ (false). Let $\Phi$ be a QBF formula and $\gamma$ be a variable assignment, we denote by $\Phi[\gamma]$ the formula obtained by substituting variables with constants they are mapped to. For instance, let $\Phi = \forall x\ (x \vee y)$ and $\gamma = \{y \mapsto \top\}$ , $\Phi[\gamma] = \forall x\ (x \vee \top)$. Given a variable assignment $\gamma$, the semantics of QBF formulas is defined inductively:

1. $\gamma(\Phi) = \gamma(x)$, if $\Phi = x$ and $x$ is a propositional variable;
2. $\gamma(\Phi \vee \Psi) = \top$ if and only if $\gamma(\Phi) = \top$ or $\gamma(\Psi) = \top$;
3. $\gamma(\Phi \wedge \Psi) = \top$ if and only if $\gamma(\Phi) = \top$ and $\gamma(\Psi) = \top$;
4. $\gamma(\neg\Phi) = \top$ if and only if $\gamma(\Phi) = \bot$;
5. $\gamma(\exists x\ \Phi) = \top$ if and only if $\gamma(\Phi[x \mapsto \top]) = \top$ or $\gamma(\Phi[x \mapsto \bot]) = \top$;

6. $\gamma(\forall x\ \Phi) = \top$ if and only if $\gamma(\Phi[x \mapsto \top]) = \top$ and $\gamma(\Phi[x \mapsto \bot]) = \top$.

Given a QBF formula $\Phi$, then $\Phi$ is *satifisfiable* if and only if there exists a variable assignment $\gamma : free(\Phi) \mapsto \{\top, \bot\}$ such that $\gamma(\Phi) = \top$, otherwise $\Phi$ is *unsatisfiable*. Let $\Phi$ and $\Psi$ be two QBF formulas: $\Phi \models \Psi$ if for every assignment $\gamma$ such that $\gamma(\Phi) = \top$ then $\gamma(\Psi) = \top$; $\Phi$ is *logically equivalent* to $\Psi$ if $\Phi \models \Psi$ and $\Psi \models \Phi$; $\Phi$ is *satisfiable equivalent* to $\Psi$ if $\Phi$ is satisfiable if and only if $\Psi$ is satisfiable. Two closed (i.e. no free variables) QBF formulas are logically equivalent if and only if they are satisfiable equivalent. In what follows, since we will deal with closed QBF formulas in prenex normal form, without loose of generality we refer to formula equivalence as satisfiability equivalence.

## 6.2 Quantified Answer Set Programming

An *ASP with Quantifiers* (ASP(Q)) program $\Pi$ is of the form [8]:

$$\Box_1 P_1\ \Box_2 P_2\ \cdots\ \Box_n P_n : C, \tag{6.1}$$

where, for each $i = 1, \ldots, n$, $\Box_i \in \{\exists^{st}, \forall^{st}\}$, $P_i$ is a program, and $C$ is a stratified program possibly with constraints. An ASP(Q) program $\Pi$ of the form (6.1) is *existential* if $\Box_1 = \exists^{st}$, otherwise it is *universal*.

Given a program $P$, a total interpretation $I$ over the Herbrand base $B_P$, and an ASP(Q) program $\Pi$ of the form (6.1), we denote by $\mathit{fix}_P(I)$ the set of facts and constraints $\{a \leftarrow |\ a \in I \cap B_P\} \cup \{\leftarrow a \mid a \in B_P \setminus (\neg I^-)\}$, and by $\Pi_{P,I}$ the ASP(Q) program of the form (6.1), where $P_1$ is replaced by $P_1 \cup \mathit{fix}_P(I)$, that is, $\Pi_{P,I} = \Box_1(P_1 \cup \mathit{fix}_P(I))\ \Box_2 P_2\ \cdots \Box_n P_n : C$.

*Coherence* of ASP(Q) programs is defined by induction as follows:

- $\exists^{st} P : C$ is coherent, if there exists $M \in AS(P)$ such that $C \cup \mathit{fix}_P(M)$ is coherent;
- $\forall^{st} P : C$ is coherent, if for every $M \in AS(P)$, $C \cup \mathit{fix}_P(M)$ is coherent;
- $\exists^{st} P\ \Pi$ is coherent, if there exists $M \in AS(P)$ such that $\Pi_{P,M}$ is coherent;
- $\forall^{st} P\ \Pi$ is coherent, if for every $M \in AS(P)$, $\Pi_{P,M}$ is coherent.

"Unwinding" the definition for a quantified program $\Pi$:

$$\exists^{st} P_1 \forall^{st} P_2 \cdots \exists^{st} P_{n-1} \forall^{st} P_n : C$$

yields that $\Pi$ is coherent if there exists an answer set $M_1$ of $P'_1$ such that for each answer set $M_2$ of $P'_2$ there is an answer set $M_3$ of $P'_3, \ldots$, there is an answer set $M_{n-1}$ of $P'_{n-1}$ such that for each answer set $M_n$ of $P'_n$, there is an answer set of $C \cup fix_{P'_n}(M_n)$, where $P'_1 = P_1$, and $P'_i = P_i \cup fix_{P'_{i-1}}(M_{i-1})$, if $i \geq 2$. For an existential ASP(Q) program $\Pi$, $M \in AS(P_1)$ is a *quantified answer set* of $\Pi$, if $(\Box_2 P_2 \cdots \Box_n P_n : C)_{P_1,M}$ is coherent. We denote by $QAS(\Pi)$ the set of all quantified answer sets of $\Pi$.

Given a set of propositional atoms $A$, we denote by $ch(A)$ the program $\{\{a\} | a \in A\}$ made of choice rules over atoms in $A$. We recall that a choice rule $\{a_1; \cdots; a_k\}$ is just a shorthand for rules: $a_i \leftarrow {\sim}na_i$, and $na_i \leftarrow {\sim}a_i$ for $1 \leq i \leq k$, where all $na_i$ are fresh atoms not appearing elsewhere.

For two programs $P$ and $P'$, let $Int(P, P')$ be the set $B_P \cap B_{P'}$ of common atoms. For two programs $P$ and $P'$, the choice interface program $CH(P, P')$ is defined as $ch(Int(P, P'))$. For a propositional formula $\Phi$, $var(\Phi)$ denotes the variables occurring in $\Phi$. For an ASP(Q) program $\Pi$, and an integer $1 \leq i \leq n$, we define the program $P_i^{\leq}$ as the union of program $P_j$ with $1 \leq j \leq i$. Given an ASP(Q) program $\Pi$ of the form (6.1), the intermediate versions $G_i$ of its subprograms, and the QBF $\Phi(\Pi)$ encoding $\Pi$ are:

$$G_i = \begin{cases} P_1 & i = 1 \\ P_i \cup CH(P_{i-1}^{\leq}, P_i) & 1 < i \leq n \\ C \cup CH(P_n^{\leq}, C) & i = n+1 \end{cases}$$

$$\Phi(\Pi) = \boxplus_1 \cdots \boxplus_{n+1} \left( \bigwedge_{i=1}^{n+1} (\phi_i \leftrightarrow CNF(G_i)) \right) \wedge \phi_c$$

where $CNF(P)$ is a CNF formula encoding the program $P$ (such that models of $CNF(P)$ correspond to $AS(P)$); $\phi_1, \ldots, \phi_{n+1}$ are fresh propositional variables; $\boxplus_i = \exists x_i$ if $\Box_i = \exists^{st}$ or $i = n+1$, and $\boxplus_i = \forall x_i$ otherwise, where $x_i = var(\phi_i \leftrightarrow CNF(G_i))$ for $i = 1, \cdots, n+1$, and $\phi_c$ is the formula

$$\phi_c = \phi'_1 \odot_1 (\phi'_2 \odot_2 (\cdots \phi'_n \odot_n (\phi_{n+1}) \cdots))$$

where $\odot_i = \vee$ if $\Box_i = \forall^{st}$, and $\odot_i = \wedge$ otherwise, and $\phi'_i = \neg\phi_i$ if $\Box_i = \forall^{st}$, and $\phi'_i = \phi_i$ otherwise. Intuitively, there is a direct correspondence between

the quantifiers in $\Pi$ and $\Phi(\Pi)$; moreover, in each subprogram of $\Pi$ (i.e, $P_1, \cdots, P_n, C$) the atoms interfacing with preceding subprograms are left open; then the programs are converted into equivalent CNF formulas; finally, the formula $\phi_c$ is built to constrain the variable assignments corresponding to the stable models of each subprogram so that they behave as required by the semantics of ASP(Q). In order to better understand, the following example reports an encoding of an ASP(Q) program into QBF formula.

**Example 11.** *Consider a ASP(Q) program* $\Pi$ *of the form:* $\exists P_1 \forall P_2 : C$, *where*

| $P_1$ | $P_2$ | $C$ |
|---|---|---|
| $\{a; b\} \leftarrow$ | $c \leftarrow \sim a, \sim b$ | $\leftarrow e,\ c$ |
| $\leftarrow f,\ a,\ \sim b$ | $d \leftarrow f$ | $\leftarrow e, \sim d$ |
| $f \leftarrow g$ | $\{e\} \leftarrow$ | |
| $g \leftarrow$ | | |

*The first step of the encoding produces the following programs by adding interface from previous levels:*

| $G_1$ | $G_2$ | $G_3$ |
|---|---|---|
| $\{a; b\} \leftarrow$ | $c \leftarrow \sim a, \sim b$ | $\leftarrow e,\ c$ |
| $\leftarrow f,\ a,\ \sim b$ | $d \leftarrow f$ | $\leftarrow e, \sim d$ |
| $f \leftarrow g$ | $\{e\} \leftarrow$ | $\{e; c; d\}$ |
| $g \leftarrow$ | $\{a; b; f\} \leftarrow$ | |

*The resulting CNF encodings are the following:*

$$
\begin{aligned}
CNF(G_1): \quad & (a \vee aux_1) \wedge (\sim a \vee \sim aux_1) \wedge (b \vee aux_2) \wedge (\sim b \vee \sim aux_2) \wedge \\
& (\sim f \vee \sim a \vee b) \wedge (\sim f \vee g) \wedge (\sim g \vee f) \wedge g \\
CNF(G_2): \quad & (\sim c \vee \sim a) \wedge (\sim c \vee b) \wedge (a \vee \sim b \vee c) \wedge \\
& (\sim d \vee f) \wedge (\sim f \vee d) \wedge (e \vee aux_3) \wedge (\sim e \vee \sim aux_3) \\
& (a \vee aux_4) \wedge (\sim a \vee \sim aux_4) \wedge (b \vee aux_5) \wedge (\sim b \vee \sim aux_5) \wedge \\
& (f \vee aux_6) \wedge (\sim f \vee \sim aux_6) \\
CNF(G_3): \quad & (\sim e \vee \sim c) \wedge (\sim e \vee d) \wedge \\
& (e \vee aux_7) \wedge (\sim e \vee \sim aux_7) \wedge (c \vee aux_8) \wedge (\sim c \vee \sim aux_8) \wedge \\
& (d \vee aux_9) \wedge (\sim d \vee \sim aux_9)
\end{aligned}
$$

*where* $aux_i$ *are hidden atoms are fresh propositional variables introduced by*

*translation The final QBF formula* $\Phi(\Pi)$*:*

$$\exists\ a,\ b,\ f,\ g,\ aux_1,\ aux_2$$
$$\forall\ c,\ d,\ e,\ aux_3,\ aux_4,\ aux_5, aux_6$$
$$\exists\ aux_7,\ aux_8,\ aux_9$$
$$((\phi_1 \leftrightarrow CNF(G_1)) \wedge (\phi_2 \leftrightarrow CNF(G_2)) \wedge (\phi_3 \leftrightarrow CNF(G_3))) \wedge$$
$$(\phi_1 \wedge (\sim\phi_2 \vee \phi_3))$$

**Theorem 1** ([5])**.** *Let* $\Pi$ *be a quantified program. Then* $\Phi(\Pi)$ *is true iff* $\Pi$ *is coherent.*

# Chapter 7

# Simplification based on well-founded semantics

In this chapter, we present an alternative approach that exploits the well-founded semantics in order to obtain a simplified but equivalent ASP(Q) programs featuring a more compact translation into a QBF formula both in terms of number of clauses and average clause length.

**Definition 3.** *Given a program $P$ and its well-founded model $\mathcal{W}$, the residual program $R(P)$ is obtained from $P$ by (i) removing all rules with a false body w.r.t. $\mathcal{W}$, and (ii) removing true literals in $\mathcal{W}$ from the bodies of the remaining ones.*

**Proposition 1.** *Given a program $P$ and its well-founded model $\mathcal{W}$, $AS(P) = AS(R(P))$*

The proof follows from the fact that $\mathcal{W}$ is a subset of any stable model $M$ of $P$ and so the missing rules in $R(P)$ have a false body w.r.t. M and so they are trivially satisfied by $M$. Each rule in $P$ that has not been removed in $R(P)$ is satisfied if and only if the corresponding simplified rule in $R(P)$ is satisfied. So, it can be proved that the reducts of the two programs have the same minimal models for every model $M$.

Let $P$ and $Q$ be two programs and $\mathcal{W}$ the well-founded model of $P$, $CH'(P,Q) = \{\{a\} \mid a \in (Int(P,Q) \setminus (\mathcal{W} \cup \neg\mathcal{W}))\} \cup \{a \leftarrow \mid a \in (Int(P,Q) \cap \mathcal{W})\}$. Given an ASP(Q) program $\Pi$ of the form (6.1), the QBF encoding

$\Phi^{\mathcal{WF}}(\Pi)$ is as follows:

$$G_i^{\mathcal{WF}} = \begin{cases} R(P_1) & i = 1 \\ R(P_i \cup CH'(P_{i-1}^{\leq}, P_i)) & i \in [2..n] \\ R(C \cup CH'(P_n^{\leq}, C)) & i = n+1 \end{cases}$$

$$\Phi^{\mathcal{WF}}(\Pi) = \boxplus_1 \cdots \boxplus_{n+1} \left( \bigwedge_{i=1}^{n+1} (\phi_i^{\mathcal{WF}} \leftrightarrow CNF(G_i^{\mathcal{WF}}) \right) \wedge \phi_c,$$

where $CNF(G_i^{\mathcal{WF}})$ is a CNF formula encoding $G_i^{\mathcal{WF}}$, $\phi_1^{\mathcal{WF}}, \ldots, \phi_{n+1}^{\mathcal{WF}}$ are fresh propositional variables; $\boxplus_i = \exists x_i$ if $\square_i = \exists^{st}$ or $i = n+1$, and $\boxplus_i = \forall x_i$ otherwise, where $x_i = var(\phi_i^{\mathcal{WF}} \leftrightarrow CNF(G_i^{\mathcal{WF}}))$ for $i = 1, \cdots, n+1$, and $\phi_c$ is the formula

$$\phi_c = \phi_1' \odot_1 (\phi_2' \odot_2 (\cdots \phi_n' \odot_n (\phi_{n+1}) \cdots))$$

where $\odot_i = \vee$ if $\square_i = \forall^{st}$, and $\odot_i = \wedge$ otherwise, and $\phi_i' = \neg\phi_i^{\mathcal{WF}}$ if $\square_i = \forall^{st}$, and $\phi_i' = \phi_i^{\mathcal{WF}}$ otherwise. Intuitively, $\Phi^{\mathcal{WF}}(\Pi)$ is constructed by following the encoding proposed in Section 6.2 but each program $P_i$ is replaced by its residual w.r.t. the well-founded model.

**Example 12.** *Consider a ASP(Q) program* $\Pi$ *of the form:* $\exists P_1 \forall P_2 : C$, *where*

| $P_1$ | $P_2$ | $C$ |
|---|---|---|
| $\{a; b\} \leftarrow$ | $c \leftarrow {\sim}a, {\sim}b$ | $\leftarrow e,\ c$ |
| $\leftarrow f,\ a,\ {\sim}b$ | $d \leftarrow f$ | $\leftarrow e, {\sim}d$ |
| $f \leftarrow g$ | $\{e\} \leftarrow$ | |
| $g \leftarrow$ | | |

*The first step of the encoding produces the following residual subprograms by adding the interface* $CH'$ *w.r.t. previous levels:*

| $G_1^{\mathcal{WF}}$ | $G_2^{\mathcal{WF}}$ | $G_3^{\mathcal{WF}}$ |
|---|---|---|
| $\{a; b\} \leftarrow$ | $c \leftarrow {\sim}\ a,\ {\sim}\ b$ | $\leftarrow e,\ c$ |
| $\leftarrow a,\ {\sim}\ b$ | $d \leftarrow$ | $\{c; e\} \leftarrow$ |
| $f \leftarrow$ | $\{e\} \leftarrow$ | $d \leftarrow$ |
| $g \leftarrow$ | $\{a; b\} \leftarrow$ | |
| | $f \leftarrow$ | |

*The resulting CNF encodings are the following:*

$$
\begin{array}{ll}
CNF(G_1^{\mathcal{WF}}): & (b \vee aux_1) \wedge (\sim b \vee \sim aux_1) \wedge (a \vee aux_2) \wedge (\sim a \vee \sim aux_2) \wedge \\
& (\sim a \vee b) \wedge f \wedge g \\
CNF(G_2^{\mathcal{WF}}): & (a \vee aux_3) \wedge (\sim a \vee \sim aux_3) \wedge (b \vee aux_4) \wedge (\sim b \vee \sim aux_4) \wedge \\
& d \wedge f \wedge (e \vee aux_5) \wedge (\sim e \vee \sim aux_5) \\
& (c \vee a \vee b) \wedge (\sim c \vee \sim b) \wedge (\sim c \vee \sim a) \wedge \\
CNF(G_3^{\mathcal{WF}}): & (c \vee aux_6) \wedge (\sim c \vee \sim aux_6) \wedge d \wedge \\
& (e \vee aux_7) \wedge (\sim e \vee \sim aux_7) \wedge (\sim e \vee \sim c)
\end{array}
$$

*where $aux_i$ are hidden atoms are fresh propositional variables introduced by translation. The final QBF formula $\Phi(\Pi)$:*

$$
\begin{array}{l}
\exists\ a,\ b,\ f,\ g,\ aux_1,\ aux_2 \\
\forall\ c,\ d,\ e,\ aux_3,\ aux_4,\ aux_5 \\
\exists\ aux_6,\ aux_7 \\
((\phi_1 \leftrightarrow CNF(G_1^{\mathcal{WF}})) \wedge (\phi_2 \leftrightarrow CNF(G_2^{\mathcal{WF}})) \wedge (\phi_3 \leftrightarrow CNF(G_3^{\mathcal{WF}}))) \wedge \\
(\phi_1 \wedge (\sim \phi_2 \vee \phi_3))
\end{array}
$$

**Theorem 2.** *Let $\Pi$ be an ASP(Q) program, then $\Phi^{\mathcal{WF}}(\Pi)$ is true iff $\Pi$ is coherent.*

*Proof.* By means of $CH'$ possible models of $P_i$ are reduced to those that are coherent with models from previous levels by fixing the truth value of literals that have been determined by the well-founded operator and so $AS(G_i^{\mathcal{WF}})) \subseteq AS(G_i)$. If there exists $M \in AS(G_i)$ such that $M \notin AS(G_i^{\mathcal{WF}})$ then there exists some literal $l \in M$ such that $\sim l$ belongs to the well-founded model of $P_{i-1}$ and so $M$ is not coherent with models of previous levels. So, the program $P_i' = P_i \cup CH'(G_{i-1}^{\mathcal{WF}}, P_i)$ preserves the coherence of $\Pi$. From Proposition 1 we know that $AS(P_i') = AS(R(P_i'))$ and so from Theorem 1 we can conclude that $\Phi^{\mathcal{WF}}(\Pi)$ is true iff $\Pi$ is coherent. □

**Proposition 2.** *Let $\Pi$ be an ASP(Q) program of the form (6.1), if $G_i^{\mathcal{WF}}$ is incoherent then $\phi_i^{\mathcal{WF}}$ can be replaced by $\bot$.*

It is easy to see that if $G_i^{\mathcal{WF}}$ is incoherent then $CNF(G_i^{\mathcal{WF}})$ is unsatisfiable and so $\phi_i^{\mathcal{WF}}$ can be replaced by $\bot$.

**Example 13.** *Let $P_i$ be the program $\{a \leftarrow a;\quad p \leftarrow \sim a,\ \sim p\}$. Since predicates occurring in $P_i$ are $p$ and $a$ and both are defined at level $i$ then*

*$CH'(P_{i-1}^{\leq}, P_i) = \emptyset$ and so, $G_i^{\mathcal{WF}} = R(P_i)$. Since the well-founed model of $P_i$ is $\mathcal{W} = \{\sim a\}$, then $R(P_i) = p \leftarrow\sim p$ that is incoherent, and so $CNF(G_i^{\mathcal{WF}}) = p \wedge \neg p$ is unsatisfiable.*

**Proposition 3.** *Let $\Pi$ be an ASP(Q) program of the form (6.1), if $G_k^{\mathcal{WF}}$ is incoherent then*

$$\Phi^{\mathcal{WF}}(\Pi) \equiv \boxplus_1 \cdots \boxplus_{k-1} \left( \bigwedge_{i=1}^{k-1} (\phi_i^{\mathcal{WF}} \leftrightarrow CNF(G_i^{\mathcal{WF}}) \right) \wedge \phi_c',$$

*where $\phi_1^{\mathcal{WF}}, \ldots, \phi_{k-1}^{\mathcal{WF}}$ are fresh propositional variables; $\boxplus_i = \exists x_i$ if $\square_i = \exists^{st}$, and $\boxplus_i = \forall x_i$ otherwise, $x_i = var(\phi_i \leftrightarrow CNF(G_i^{\mathcal{WF}}))$ for $i = 1, \cdots, k-1$, and $\phi_c'$ is the formula*

$$\phi_c' = \phi_1' \odot_1 (\phi_2' \odot_2 (\cdots \phi_{k-1}' \odot_{k-1} (\phi_k') \cdots))$$

*where $\odot_i = \vee$ if $\square_i = \forall^{st}$, and $\odot_i = \wedge$ otherwise, for $i \in [1, \cdots, k-1]$, $\phi_i' = \neg\phi_i^{\mathcal{WF}}$ if $\square_i = \forall^{st}$, and $\phi_i' = \phi_i^{\mathcal{WF}}$ otherwise, and $\phi_k' = \top$ if $\square_i = \forall^{st}$, and $\phi_k' = \bot$ otherwise.*

From Proposition 3 it follows that if $k = 1$ then $\Phi^{\mathcal{WF}}(\Pi) = \phi_k'$ where $\phi_k' = \top$ if $\square_i = \forall^{st}$, and $\phi_k' = \bot$ otherwise. So, in such cases we can determine the coherence of the ASP(Q) directly in the encoding phase.

**Proposition 4.** *Given an ASP(Q) program $\Pi$, it holds that:*

$$|clauses(\Phi^{\mathcal{WF}}(\Pi))| \leq |clauses(\Phi(\Pi))|$$

We observe that, by Definition 3, residual subprograms are obtained by removing some trivially satisfied rules in every stable model or deleting literals from the rules' body by means of the well-founded operator. This results in a smaller CNF both in terms of the number of clauses, since potentially fewer rules are encoded, and also in average clause length, since each rule is transformed into one or more clauses that have fewer literals. Moreover, by propagating information from the well-founded model of previous levels, stable models of the following levels are restricted to those that are coherent with previous models, if any. If no models exist, then the resulting QBF formula is pruned at the incoherent level. In the worst case scenario, that is $\mathcal{W} = \emptyset$ for every program, $CH'$ produces the same interface program produced by $CH$, $G_i = G_i^{\mathcal{WF}}$ and so $\Phi^{\mathcal{WF}}(\Pi) = \Phi(\Pi)$.

# Chapter 8

# CNF encodings for ASP(Q) programs

In this chapter, we define a syntactical class of ASP(Q) programs that features a direct encoding as QBF formulas in Conjunctive Normal Form (QCNF). Our approach involves defining the class of trivial subprograms and subsequently presenting their corresponding encoding in QCNF. Moreover, to increase the applicability of this direct encoding we propose a rewriting method that takes advantage of the modularity of Guess&Check paradigm to obtain equivalent ASP(Q) programs belonging to the identified class of programs.

## 8.1 Direct Mapping ASP(Q) programs to CNF

Formulas $\Phi(\Pi)$ and $\Phi^{\mathcal{WF}}(\Pi)$ are not in CNF because of the presence of equivalences for each level $i$ and the final formula $\phi_c$ (which is not in CNF either). While this might be seen as a minor issue, the translation of non-CNF formulas into CNF by means of a Tseytin transformation can be a time-consuming procedure that increases the length of the formulas and introduces extra symbols that could slow down QBF solvers.

A natural question, therefore, is whether it is possible to identify classes of ASP(Q) programs such that the resulting QBF formula is in CNF. In the following, we provide some conditions under which this is possible.

Given a program $P$, $facts(P)$ denotes the set of facts in $P$. Given an ASP(Q) program $\Pi$, $Ext_i = \mathcal{H}(P_i) \cap \bigcup_{j>i} Int(P_i, P_j)$ denotes the set of

atoms defined in $P_i$ that belong to the interface of the subsequent levels.

**Definition 4.** *Let $\Pi$ be an ASP(Q) program, a subprogram $P_i$ is trivial if the following conditions hold: (i) $\forall\ 1 \leq j < i : Int(P_i, P_j) \subseteq facts(P_j)$ and (ii) $AS(P_i)\ |_{Ext_i} = 2^{Ext_i}$, where $AS(P_i)\ |_{Ext_i} = \{S \cap Ext_i \mid S \in AS(P_i)\}$ and $2^{Ext_i}$ denotes the power set of $Ext_i$.*

Let $\Pi$ be an ASP(Q) program, $K = \{k \mid P_k$ *is a trivial subprogram* $\wedge\, k \leq n\}$, the QBF encoding $\Phi^K(\Pi)$ is defined as follows:

$$\Phi^K(\Pi) = \boxplus_1 \cdots \boxplus_{n+1} \left( \bigwedge_{\substack{i=1 \\ i \notin K}}^{n+1} (\phi_i \leftrightarrow CNF(G_i)) \right) \wedge \phi_c^K,$$

$\boxplus_i = \exists x_i$ if $\square_i = \exists^{st}$ or $i = n+1$, and $\boxplus_i = \forall x_i$ otherwise, $x_i = var(\phi_i \leftrightarrow CNF(G_i))$ if $i \notin K$, otherwise $x_i = Ext_i$, and $\phi_c^K$ is $\phi_c^K = \phi'_{i_1} \odot_{i_1} (\phi'_{i_2} \odot_{i_2} (\cdots (\phi'_{i_m} \odot_{i_m} (\phi_{n+1})) \cdots))$ where $E = \{1, \cdots, n\} \setminus K = \{i_1, i_2, \cdots, i_m\}$, $i_1 < i_2 < \cdots < i_m$, $\odot_i = \vee$ if $\square_i = \forall^{st}$, and $\odot_i = \wedge$ otherwise, and $\phi'_i = \neg\phi_i$ if $\square_i = \forall^{st}$, and $\phi'_i = \phi_i$ otherwise, with $i \in E$.

**Theorem 3.** *Let $\Pi$ be an ASP(Q) program, and $K = \{k \mid P_k$ is a trivial subprogram $\wedge\ k \leq n\}$, then $\Phi^K(\Pi)$ is satisfiable iff $\Pi$ is coherent.*

*Proof.* From Theorem 1 we know that $\Pi$ is coherent iff $\Phi(\Pi)$ is satisfiable. Now we observe that, by hypothesis, for all $k \in K$ it holds that $P_k$ is trivial, thus $AS(P_k)\ |_{Ext_k} = 2^{Ext_k}$. This implies that for all $k \in K$, $CNF(G_k)$ is satisfiable, and thus $\phi_k \leftrightarrow \top$ holds in $\Phi(\Pi)$. Since $\phi_k \leftrightarrow CNF(G_k) \leftrightarrow \top$ holds, then the k-th conjunct is satisfiable and can be omitted obtaining an equivalent formula:

$$\Phi(\Pi) = \boxplus_1 \cdots \boxplus_n \left( \bigwedge_{\substack{i=1 \\ i \notin K}}^{n+1} (\phi_i \leftrightarrow CNF(G_i)) \right) \wedge \phi_c,$$

$\boxplus_i = \exists x_i$ if $\square_i = \exists^{st}$, and $\boxplus_i = \forall x_i$ otherwise, $x_i = var(\phi_i \leftrightarrow CNF(G_i))$ if $i \notin K$, otherwise $x_i = Ext_i$. Moreover, we observe that for each $k \in K$, $\phi_k \wedge (\phi_{k+1} \odot_{k+1} \cdots)$, can be replaced by $\top \wedge (\phi_{k+1} \odot_{k+1} \cdots)$ which is equivalent to $\phi_{k+1} \odot_{k+1} (\cdots)$ and $\neg\phi_k \vee (\phi_{k+1} \odot_{k+1} \cdots)$, can be replaced

by $\bot \vee (\phi_{k+1} \odot_{k+1} \cdots)$ which is equivalent to $(\phi_{k+1} \odot_{k+1} \cdots)$. So, $\phi_c$ can be simplified by removing $\phi'_k \odot_k$ for each $k \in K$, obtaining an equivalent formula that is $\phi_c^K$. Thus, by construction, $\Phi(\Pi)$ is equivalent to $\Phi^K(\Pi)$ and so $\Phi^K(\Pi)$ is satisfiable iff $\Pi$ is coherent. □

Basically, for any trivial $P_k$ the formula $CNF(G_k)$ is a tautology, allowing for the simplifications that result in $\Phi^K(\Pi)$.

**Proposition 5.** *Let $\Pi$ be an ASP(Q) program, $K = \{k \mid P_k$ is a trivial subprogram $\wedge\ k \leq n\}$. If for each subprogram $P_i$ such that $\square_i = \forall^{st}$, it holds that $i \in K$, then $\Phi^K$ is equivalent to the CNF formula with the same quantifiers: $\Phi^K_{CNF} = \boxplus_1 \cdots \boxplus_{n+1} \bigwedge_{j \in J} CNF(G_j)$, where $J = \{1, \ldots, n+1\} \backslash K$.*

Programs satisfying Proposition 5 have a direct CNF encoding. However, verifying that a program is trivial is hard since Definition 4 requires a co-NP check. There is, however, a very common syntactic class of programs for which this property is trivially satisfied: the ASP(Q) programs of the form (6.1) where each $P_i$ contains only choice rules. An example is the encoding of QBF in ASP(Q) proposed by Amendola et al. [8].

## 8.2 Exploiting Guess&Check

In the following, we identify a larger class of ASP(Q) programs featuring a direct encoding in CNF, the ones that follow the well-known Guess&Check methodology [31].

**Definition 5.** *A program $P$ is Guess&Check if it can be partitioned into two subprograms $G_P$ (i.e. Guess), and $C_P$ (i.e. Check), where $G_P$ contains only choice rules and $C_P$ is the maximal stratified subprogram possibly with constraints of $P$, such that $\mathcal{H}(C_P) \cap B_{G_P} = \emptyset$.*

**Example 14.** *Let $P$ be the program $\{r_1 : \{a; b; c\} \leftarrow,\ r_2 : d \leftarrow a,\ r_3 : d \leftarrow b,\ r_4 :\leftarrow c, d.\}$. $P$ can be partitioned in $G_P = \{r_1\}$ and $C_P = \{r_2, r_3, r_4\}$.*

Guess&Check programs feature a modularity property.

**Proposition 6.** *Let $P$ be a Guess&Check program then $M \in AS(P)$ iff there exists $M' \in AS(G_P)$ such that $M = M' \cup W$ and $W \in AS(C_P \cup fix_{G_P}(M'))$.*

**Definition 6.** *An ASP(Q) program* $\Pi$ *of the form (6.1) is Guess&Check if* $(i)$ *universal and existential quantifiers are alternated, and* $(ii)$ *all* $P_i$ *with* $\square_i = \forall^{st}$ *are Guess&Check subprograms.*

The following definition provides a rewriting for a universal Guess&Check subprogram.

**Definition 7.** *Given a Guess&Check program* $P_1$*, a program* $P_2$*, and a propositional atom* $u$ *such that* $u$ *does not appear neither in* $P_1$ *nor in* $P_2$*, we define*

$$\tau(u, P_1) = \begin{cases} H_r \leftarrow B_r & r \in C_{P_1} \wedge H_r \neq \emptyset \\ u \leftarrow B_r & r \in C_{P_1} \wedge H_r = \emptyset \end{cases}$$

$$\rho(u, P_2) = \{H_r \leftarrow B_r, \sim u \mid r \in P_2\}$$

$$\sigma(u, P_1, P_2) = \tau(u, P_1) \cup \rho(u, P_2)$$

*Given a Guess&Check ASP(Q) program* $\Pi$*, let* $i \in [1, \ldots, n]$ *be such that* $\square_i = \forall^{st}$*:*

$$\Pi^{GC_i} = \begin{cases} \square_1 P_1 \cdots \forall^{st} G_{P_i} : \sigma(u, P_i, C) & i = n \\ \square_1 P_1 \cdots \forall^{st} G_{P_i} \exists^{st} \sigma(u, P_i, P_{i+1}) : \rho(u, C) & i = n-1 \\ \square_1 P_1 \cdots \forall^{st} G_{P_i} \exists^{st} \sigma(u, P_i, P_{i+1}) \forall^{st} P_{i+2} \cup \{\leftarrow u\} \cdots \square_n P_n : C & otherwise \end{cases}$$

**Example 15.** *Consider a ASP(Q) program* $\Pi$ *of the form:* $\exists P_1 \forall P_2 : C$*, where*

| $P_1$ | $P_2$ | $C$ |
|---|---|---|
| $\{a; b\} \leftarrow$ | $c \leftarrow \sim a, \sim b$ | $\leftarrow e,\ c$ |
| $\leftarrow f,\ a,\ \sim b$ | $d \leftarrow f$ | $\leftarrow e, \sim d$ |
| $f \leftarrow g$ | $\leftarrow c,\ e$ | |
| $g \leftarrow$ | $\{e\} \leftarrow$ | |

*The program* $P_2$ *is Guess&Check and can be divided as follows*

| $G_{P_2}$ | $C_{P_2}$ |
|---|---|
| $\{e\} \leftarrow$ | $c \leftarrow \sim a, \sim b$ |
| | $d \leftarrow f$ |
| | $\leftarrow c,\ e$ |

*Thus,* $\Pi$ *can be rewritten as* $\Pi^{GC_2}$*:*

| $P_1$ | $P_2$ | $C$ |
|---|---|---|
| $\{a; b\} \leftarrow$ | $\{e\} \leftarrow$ | $\leftarrow e,\ c,\ {\sim}u$ |
| $\leftarrow f,\ a,\ {\sim}b$ | | $\leftarrow e, {\sim}d,\ {\sim}u$ |
| $f \leftarrow g$ | | $c \leftarrow {\sim}a, {\sim}b$ |
| $g \leftarrow$ | | $d \leftarrow f$ |
| | | $u \leftarrow c,\ e$ |

**Theorem 4.** *Let* $\Pi$ *be a Guess&Check ASP(Q) program, for each* $i \in [1, \ldots, n]$ *such that* $\square_i = \forall^{st}$*,* $\Pi$ *is coherent iff* $\Pi^{GC_i}$ *is coherent.*

Intuitively, this theorem holds because in $\Pi^{GC_i}$ the answer sets of the replaced subprograms are preserved with respect to those in $\Pi$. Interpretations that violate constraints become additional answer sets, that are either invalidated in the next universal subprogram or do not affect the coherence of $\Pi$. A formal proof follows.

*Proof.* (I) First assume that $\Pi$ is incoherent. Obviously, if $\Pi$ is incoherent due to a $P_j$ with $j < i$, then $\Pi^{GC_i}$ is incoherent for the same reason.

If $\Pi$ is incoherent due to $P_i$, in the following we will consider $P_i$ only (as it is the case when $i = 1$), rather than $P_i \cup fix_{P_{i-1}}(N)$ for some answer set $N$ of the previous level (for $i > 1$). All arguments transfer directly to the latter case.

For any $M \in AS(P_i)$ we know from Proposition 6 that $M = M_G \cup M_C$ where $M_G \in AS(G_{P_i})$ and $M_C \in AS(C_{P_i} \cup fix_{G_{P_i}}(M_G))$. If $M$ is the reason for incoherence in $\Pi$, we will show that then $M_G$ is a reason for incoherence in $\Pi^{GC_i}$. We distinguish three cases.

(1) If $i = n$, there is $M \in AS(P_i)$ with $C \cup fix_{P_i}(M)$ incoherent. Here, $M_C$ is also the unique answer set of $\tau(u, P_i) \cup fix_{G_{P_i}}(M_G)$, but since $C \cup fix_{P_i}(M)$ is incoherent, $M_C$ does not satisfy $\rho(u, C)$ either (as $u$ is false in it). It follows that there is $M_G \in AS(G_{P_i})$ such that $\sigma(u, P_i, C) \cup fix_{G_{P_i}}(M_G)$ is incoherent, and hence $\Pi^{GC_i}$ is incoherent.

(2) If $i = n - 1$, there is $M \in AS(P_i)$ such that there is no $M' \in AS(P_n \cup fix_{P_i}(M))$ such that $C \cup fix_{P_n}(M')$ is incoherent. Also here, $M_C$ is also the unique answer set of $\tau(u, P_i) \cup fix_{G_{P_i}}(M_G)$, and each $M' \in AS(P_n \cup fix_{P_i}(M))$ satisfies $\rho(u, P_n)$ as well, because $u$ is false in $M'$, which means that $AS(P_n \cup fix_{P_i}(M)) = AS(\sigma(u, P_i, P_n) \cup fix_{G_{P_i}}(M_G))$. Finally, since $u$ is false in each $M'$, from $C \cup fix_{P_n}(M')$ being incoherent we also get that

$\rho(u, C) \cup fix_{P_n}(M')$ is incoherent. Then we have $M_G \in AS(G_{P_i})$ such that there is no $M' \in AS(\sigma(u, P_i, P_n) \cup fix_{G_{P_i}}(M_G))$ such that $\rho(u, C) \cup fix_{P_n}(M')$ is incoherent, hence $\Pi^{GC_i}$ is incoherent.

(3) If $i < n-1$, there is $M \in AS(P_i)$ such that there is no $M' \in AS(P_{i+1} \cup fix_{P_i}(M))$ such that $\Pi'_{P_{i+1},M'}$ is incoherent, where $\Pi'$ is the suffix of $\Pi$ starting at $P_{i+2}$. Also here, $M_C$ is also the unique answer set of $\tau(u, P_i) \cup fix_{G_{P_i}}(M_G)$, and each $M' \in AS(P_{i+1} \cup fix_{P_i}(M))$ satisfies $\rho(u, P_{i+1})$ as well, because $u$ is false in $M'$, which means that $AS(P_{i+1} \cup fix_{P_i}(M)) = AS(\sigma(u, P_i, P_{i+1}) \cup fix_{G_{P_i}}(M_G))$. Also, observe that since $u$ is false in each of these $M'$, $AS(P_{i+2} \cup fix_{P_{i+1}}(M')) = AS(P_{i+2} \cup \{\leftarrow u\} \cup fix_{P_{i+1}}(M'))$. So there is $M_G \in AS(G_{P_i})$ such that there is no $M' \in AS(\sigma(u, P_i, P_{i+1}) \cup fix_{G_{P_i}}(M_G))$ such that $\Pi''_{P_{i+1},M'}$ (the suffix of $\Pi^{GC_i}$ starting at $P_{i+2} \cup \{\leftarrow u\}$) is incoherent, so $\Pi^{GC_i}$ is incoherent.

(II) Now assume that $\Pi$ is coherent. As above, in the following we will consider $P_i$ only (as it is in the case when $i = 1$), rather than $P_i \cup fix_{P_{i-1}}(N)$ for some answer set $N$ of the previous level (for $i > 1$).

Here we have to show that from coherence for each $M \in AS(P_i)$ in $\Pi$ coherence for each $M_G \in AS(G_{P_i})$ follows. If (case a) there is an $M_C \in AS(C_{P_i} \cup fix_{G_{P_i}}(M_G))$ (the unique answer set), this follows quite easily because $M = M_G \cup M_C$ due to Proposition 6 and $M_C$ is the unique answer set of $\tau(u, P_i) \cup fix_{G_{P_i}}(M_G)$, in which $u$ is false. If (case b) $AS(C_{P_i} \cup fix_{G_{P_i}}(M_G))$ is incoherent, then there is a single answer set $M_u$ of $\tau(u, P_i) \cup fix_{G_{P_i}}(M_G)$, in which $u$ is true. We distinguish three cases.

(1) If $i = n$, for any $M \in AS(P_i)$ the program $C \cup fix_{P_i}(M)$ is coherent; let $X$ be one of its answer sets. In case a, $M_C \cup X$ is an answer set of $\sigma(u, P_i, C) \cup fix_{G_{P_i}}(M_G)$. In case b, $M_u$ is an answer set of $\sigma(u, P_i, C) \cup fix_{G_{P_i}}(M_G)$ (as all rules in $\rho(u, C)$ are satisfied by $M_u$ due to $u$ being true). In both cases, $\Pi^{GC_i}$ is coherent.

(2) If $i = n-1$, for any $M \in AS(P_i)$ there is an $M' \in AS(P_n \cup fix_{P_i}(M))$ such that $C \cup fix_{P_n}(M')$ is coherent with an answer set $X$. In case a, $M_C \cup M'$ is an answer set of $\sigma(u, P_i, P_n) \cup fix_{G_{P_i}}(M_G)$, and $C \cup fix_{P_n}(M_C \cup M')$ is coherent with the answer set $X$. In case b, $M_u$ is an answer set of $\sigma(u, P_i, P_n) \cup fix_{G_{P_i}}(M_G)$ (as all rules in $\rho(u, C)$ are satisfied by $M_u$ due to $u$ being true). But then $M_u$ is also an answer set of $C \cup fix_{P_n}(M_u)$, which is therefore coherent. In both cases, $\Pi^{GC_i}$ is coherent.

(3) If $i < n-1$, for any $M \in AS(P_i)$ there is an $M' \in AS(P_{i+1} \cup fix_{P_i}(M))$ such that $\Pi'_{P_{i+1},M'}$ is coherent, where $\Pi'$ is the suffix of $\Pi$ starting at $P_{i+2}$.

In case a, $M_C \cup M'$ is an answer set of $\sigma(u, P_i, P_{i+1}) \cup fix_{G_{P_i}}(M_G)$, and $\Pi''_{P_{i+1}, M_C \cup M'}$ (the suffix of $\Pi^{GC_i}$ starting at $P_{i+2} \cup \{\leftarrow u\}$) is coherent since $M_C \cup M' = M'$ and $u$ is false in $M'$. In case b, $M_u$ is an answer set of $\sigma(u, P_i, P_{i+1}) \cup fix_{G_{P_i}}(M_G)$ (as all rules in $\rho(u, P_{i+1})$ are satisfied by $M_u$ due to $u$ being true). But then $P_{i+2} \cup \{\leftarrow u\} \cup fix_{P_{i+1}}(M_u)$ has no answer sets (because of $u$), so $\Pi''_{P_{i+1}, M_C \cup M'}$ is trivially coherent. In both cases, $\Pi^{GC_i}$ is coherent. □

We now define a recursive transformation that, given a Guess&Check ASP(Q) program $\Pi$, builds a sequence of ASP(Q) programs $(\Pi_1, \ldots, \Pi_n)$ such that the last program of that sequence is both equivalent to $\Pi$ and features an encoding in CNF.

**Definition 8.** *Let $\Pi$ be a Guess&Check ASP(Q) program, then*

$$\Pi_i = \begin{cases} \Pi & i = 1 \wedge \square_i = \exists^{st} \\ \Pi^{GC_1} & i = 1 \wedge \square_i = \forall^{st} \\ \Pi_{i-1} & i \in [2..n] \wedge \square_i = \exists^{st} \\ (\Pi_{i-1})^{GC_i} & i \in [2..n] \wedge \square_i = \forall^{st} \end{cases}$$

**Theorem 5.** *Let $\Pi$ be a Guess&Check ASP(Q) program, and $K$ be the set of indexes $K = \{k \mid k \in [1, \ldots, n] \wedge \square_k = \forall^{st}\}$ (i.e., s.t. $P_k$ a universally quantified subprogram), then $\Pi$ is coherent iff $\Phi^K_{CNF}(\Pi_n)$ is satisfied.*

*Proof.* Observe that, by definition, $\Pi_n$ is such that all of its universally quantified subprograms are trivial (contain only choice rules). Let $K = \{k \mid k \in [1, \ldots, n] \wedge \square_i = \forall^{st}$ in $\Pi_n\}$, from Theorem 3 it follows that $\Pi_n$ is coherent iff $\Phi^K(\Pi_n)$ is satisfiable. Moreover, from Proposition 5, we have that $\Phi^K(\Pi_n)$ is equivalent to $\Phi^K_{CNF}(\Pi_n)$. From Theorem 4 we have that, $\Pi$ is coherent iff $\Pi_n$ is coherent; since $\Pi_n$ is coherent iff $\Phi^K(\Pi_n)$ is satisfiable, and $\Phi^K(\Pi_n)$ is equivalent to $\Phi^K_{CNF}(\Pi_n)$, the thesis follows. □

Intuitively, here all universal subprograms are replaced by programs that contain only choice rules and are therefore trivial. The result then follows from Theorem 3, Proposition 5, and Theorem 4.

# Chapter 9

# Implementation and Experiments

In this chapter, we describe our implementation and discuss an experimental analysis conducted to:

(*i*) demonstrate empirically the efficacy of the techniques described above;

(*ii*) compare PYQASP with QASP; and

(*iii*) compare PYQASP with a recent implementation of the stable unstable semantics [46].

## 9.1 The pyqasp system

The PYQASP system is an implementation of the transformation techniques described in Chapters 6, 7 and 8. The evaluation of an ASP(Q) program is, basically, done in two steps, namely encoding and solving. In the encoding phase an ASP(Q) program is parsed, identifying ASP programs enclosed under the quantifiers' scope. Note that PYQASP can handle non-propositional inputs, indeed the user can select either GRINGO [39] or IDLV [17] as grounders. Then, each ASP subprogram $P_i$ passes through the following pipeline:

1. **Rewriting Module**. This modules is designed to compute syntactical properties of $P_i$, in order to check whether it is a Guess&Check or trivial subprogram and subsequently apply the appropriate rewriting techniques described in this paper. First of all, $P_i$ is rewritten, taking

into account a previous Guess&Check subprogram $P_j$ with $j < i$, if any exists. Then, if the resulting program is trivial, this module returns atoms defined at the current level with an empty program. Otherwise, if it is a Guess&Check program and it is universally quantified then $P_i$ is split into $G_{P_i}$ and $C_{P_i}$. Moreover, the module computes the result of the transformation $\tau$, introducing a fresh propositional atom $u_i$, that will be used for rewriting the following levels. As a result, it returns the atoms defined in the guess split of the current subprogram with an empty program. In all the other cases this module returns the current program together with symbols defined at the current level.

2. **Well-founded Module**. This module computes the well-founded model together with the residual program by means of `DLV2` as a back-end system, and stores the truth values of literals belonging to the well-founded model.

3. **CNF Encoder Module**. This module takes as input the residual program produced by the well-founded module and encodes it as a CNF formula. In particular, if the residual program is incoherent then it is encoded as the empty clause that is equivalent to $\bot$, and then it breaks the pipeline. Otherwise, if the residual program is empty it is encoded as an empty CNF. In all the other cases, the SAT encoding of the residual program ASP subprograms ($CNF(\cdot)$) is produced using ASPTOOLS [44, 45].

4. **QBF Builder**. This module produces the final QBF formula by associating the symbols produced by **Rewriting Module** with the respective quantifiers and joining the CNFs produced by the previous module in the final conjunction.

The last step of the encoding phase is combining previous CNFs into the formula $\phi_c$. As a result, a QBF formula in `QCIR` format is obtained. The solving step is mainly performed by the solver module, which is a wrapper module for the various QBF solvers. In order to use a solver, the first step in the wrapper is to convert the QCIR formula into an equivalent formula in the solver's input format. Then, the external QBF solver is executed on the converted formula and the final outcome is computed. In the current implementation we provide the following solver wrappers:

- `QuabsWapper`. It uses the QBF solver `quabs` and doesn't require any format conversion since the solver directly accepts QCIR formulas.

- `RareqsWrapper` This wrapper uses the QBF solver `rareqs` whose input format is `gq`. The conversion from `QCIR` to `gq` is implemented by the external module `qcir-conv` provided by (ref to qcir-conv).

- `DepqbfWrapper` It uses the QBF solver `depqbf` equipped with the QBF pre-processor `bloqqer`. This solver takes as input formulas in *QDIMACS* format and so, translation to CNF is required. In particular, if all universally quantified subprograms were Guess&Check then we know that the produced formula is, indeed, in CNF. So a direct mapping into *QDIMACS* format exists, just reporting quantifiers and clauses of intermediate CNFs. Otherwise, the external module `qcir-conv` combined with `fmla` is used in order to translate the input formula into an equivalent *QDIMACS* one. Note that this translation could introduce extra symbols and clauses leading to a bigger formula.

Moreover, PYQASP implements an automatic algorithm selection strategy, devised according to the methodology employed in the `ME-ASP` multi-engine ASP solver proposed by Maratea et al. [53], that selects automatically a suitable back-end for the given input. More precisely, the automatic back-end selection has been realized by exploiting machine learning models that have been trained on dataset reporting syntactical properties of benchmarks proposed for ASP(Q). For this task we extended our system by adding a module (ASPSTATS) that analyzes ground programs during encoding phase and then, a Random Forest Classifier is used to predict the back-end solver to be used. In order to train the employed model, we considered a dataset containing instances from all our benchmarks: Argumentation Coherence, Paracoherent ASP, Minmax Clique, Point of No Return, QBF and 2QBF. In particular, for each instance the features reported in Table 9.1 have been computed by using ASPSTATS module. As required by the `ME-ASP` methodology, training set has been constructed by considering only those instances that have been solved exactly by one back-end solver that indeed is the target label, and considered the best oracles available as labels for multinomial classification. Regarding training phase we used a Random Forest Classifier made of 100 trees that have been trained by using Gini impurity criterion and bootstrap sampling technique. The source code is available at `https://github.com/MazzottaG/PyQASP.git`.

| | |
|---|---|
| $R$ | Rule count |
| $A$ | Number of atoms |
| $(R/A)$ | Ratio between rules count and atoms count |
| $(R/A)^2$ | Squared ratio between rules count and atoms count |
| $(R/A)^3$ | Cube ratio between rules count and atoms count |
| $(A/R)$ | Ratio between atoms count and rules count |
| $(A/R)^2$ | Squared ratio between atoms count and rules count |
| $(A/R)^3$ | Cube ratio between atoms count and rules count |
| $R1$ | Rule with body of length 1 |
| $R2$ | Rule with body of length 2 |
| $R3$ | Rule with body of length 3 |
| $PR$ | Positive rule count |
| $F$ | Normal facts count |
| $DF$ | Disjunctive facts count |
| $NR$ | Normal rule count |
| $NC$ | Constraint count |
| $VF$ | Universal atoms count |
| $VE$ | Existantial atoms count |
| $QF$ | Universial levels count |
| $QE$ | Existantial levels count |
| $QL$ | Quantification levels count |

Table 9.1: ASPSTATS features

## 9.2 ASP(Q) Benchmarks and Hardware Setup

We run a suite of benchmarks that has already been used to assess the performance of ASP(Q) implementations [5]. The suite contains encodings in ASP(Q) and instances of four problems: Quantified Boolean Formulas (QBF); Argumentation Coherence (AC); Minmax Clique (MMC); Paracoherent ASP (PAR). The suite comprises a selection of instances from QBF Lib (`https://www.qbflib.org/`), ICCMA 2019 (`http://argumentationcompetition.org/2019`), ASP Competitions [41], and PAR instances by Amendola et al. [6]. A detailed description of these benchmarks was provided by Amendola et al. [5].

The experiments were run on a system with 2.30GHz Intel(R) Xeon(R) Gold 5118 CPU and 512GB of RAM with Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-137-generic x86_64). Execution time and memory were limited to 800

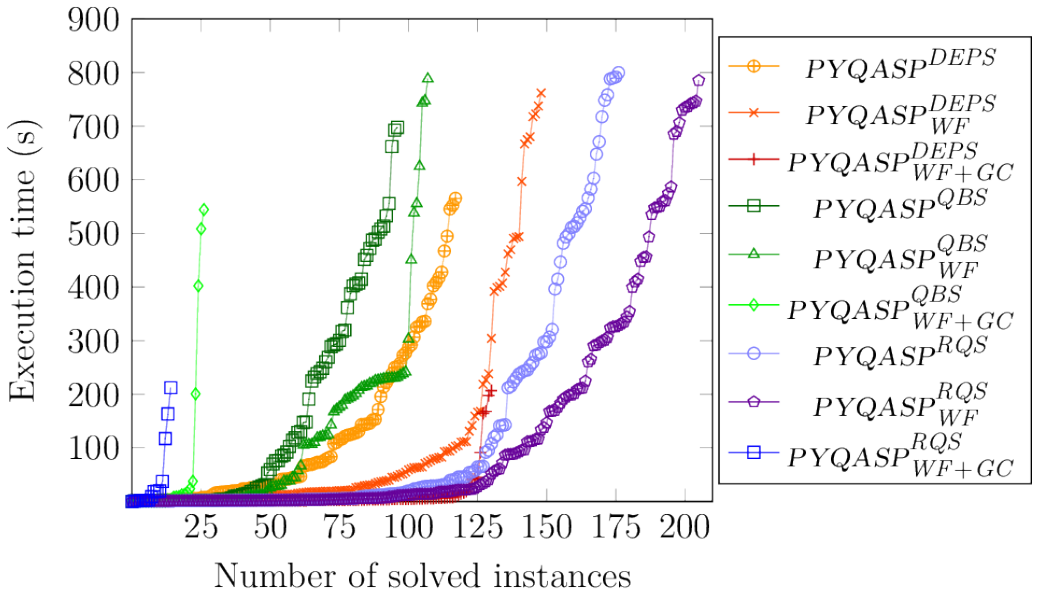


Figure 9.1: Argumentation Coherence (AC).

seconds (of CPU time, i.e., user+system) and 12 GB, respectively. Each system was limited to run in a single core.

## 9.3 Impact of the new techniques

In order to asses the impact of the proposed techniques we compared different version of PYQASP system. More precisely, we run three variants of PYQASP, namely:

- PYQASP: basic encoding with gringo as grounder;
- $\text{PYQASP}_{WF}$: basic encoding with well-founded simplification (IDLV as grounder);
- $\text{PYQASP}_{WF+GC}$: well-founded simplification and direct encoding in CNF (i.e., production of a CNF encoding for *Guess&Check* programs).

These variants were combined with the following three QBF back-ends:

- *RQS*: *qcir-conv.py* (by Klieber - `https://www.wklieber.com/ghostq/qcir-converter.html`) transforms QCIR to the GQ format of *RareQS*

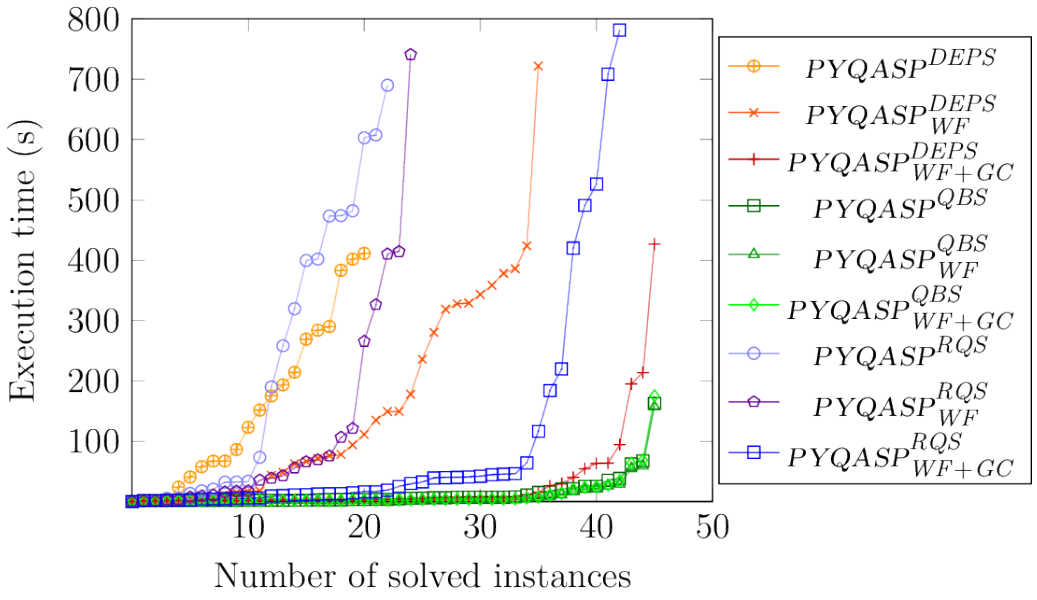


Figure 9.2: Min Max Clique (MMC).

solver (by Janota `http://sat.inesc-id.pt/~mikolas/sw/areqs`), that is called.

- *DEPS*: *qcir-conv.py* and *fmla* convert the formula from QCIR to *QDIMACS*, *bloqqer* (by Biere et al. - `http://fmv.jku.at/bloqqer`) simplifies it, then the QBF solver *DepQBF* (by Lonsin - `https://lonsing.github.io/depqbf`) is called.
- *QBS*: The QBF solver *Quabs* (by Tentrup - `https://github.com/ltentrup/quabs`) is called, with no pre-processor.

All this amounts to running 9 variants of PYQASP. In our naming conventions, the selected back-end is identified by a superscript, and a subscript identifies the optimizations enabled. For example, $\text{PYQASP}^{DEPS}$ indicates PYQASP with back-end *DEPS*, and $\text{PYQASP}^{DEPS}_{WF+GC}$ indicates PYQASP with *DEPS* back-end and all optimizations enabled.

**Results.** Obtained results are summarized in Figures 9.1 to 9.4, which aggregates the performance of each compared method in four cactus plots, one per considered problem. Recall that, a line in a cactus plot contains a

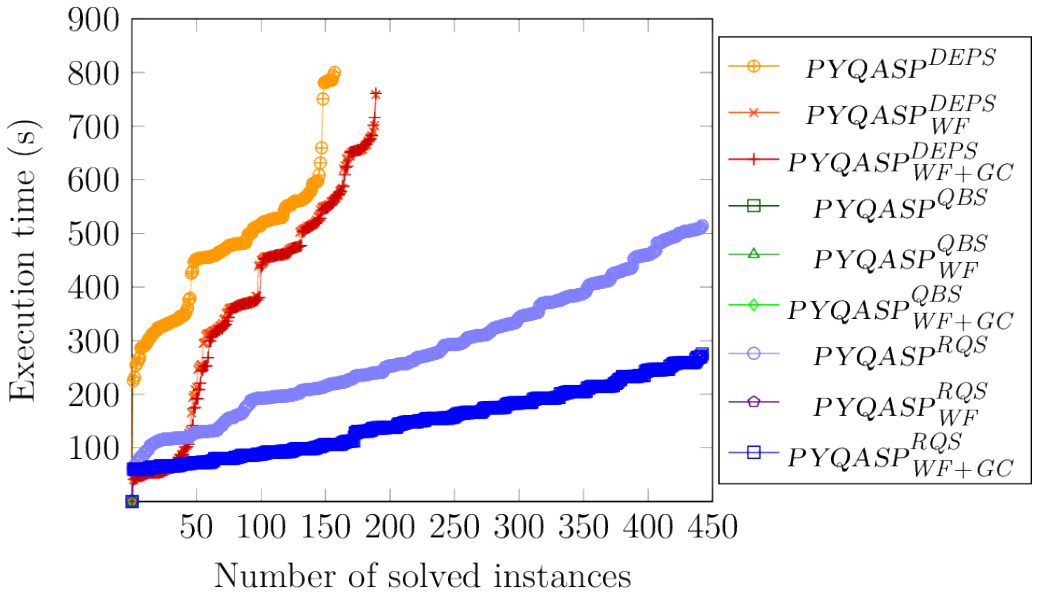


Figure 9.3: Paracoherent ASP (PAR).

point $(x, y)$ whenever the corresponding system solves at most $x$ instances in $y$ seconds.

We first observe that the different back-ends are preferable depending on the benchmark domain. In particular, RQS is the fastest option in AC and PAR (see Figures 9.1-9.3), QBS is the fastest in MMC (see Figure 9.2), and DEPS in QBF (see Figure 9.4). This behavior confirms the findings of Amendola et al. [5].

The well-founded optimization allows to solve more instances and in less time in AC, MMC, and QBF benchmarks, independently of the back-end; whereas, the identification of *Guess&Check* programs pays off in terms of solved instances in QBF and MMC, again independently of the back-end solver. The two techniques combine their positive effects in MMC, PAR, and QBF. In particular, PYQASP$^{DEPS}_{WF+GC}$ solves 25 more instances than PYQASP$^{DEPS}$ in MC, and 73 in QBF; moreover, PYQASP$^{RQS}_{WF+GC}$ solves 20 more instances than PYQASP$^{RQS}$ in MC, and 20 in QBF. However, the application of the *Guess&Check* optimization has a negative effect on AC, since the well-founded operator, applied to the rewritten program, is no longer able to derive some simplifications that instead can be derived from the original program. For this reason, PYQASP$^{RQS}_{WF}$ is the best option in AC, solving 29

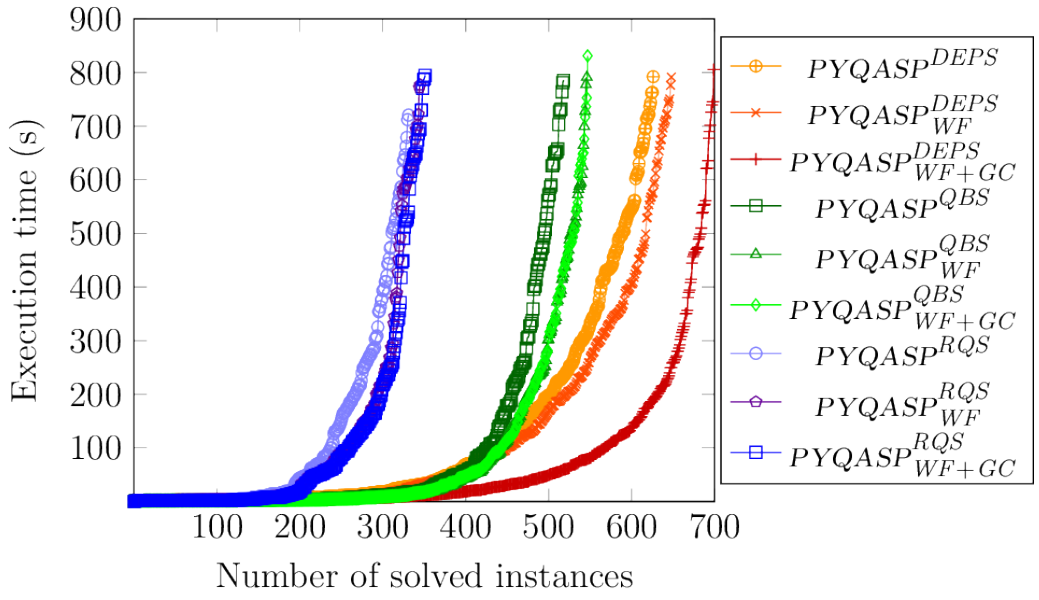


Figure 9.4: Quantified Boolean Formula (QBF).

instances more than PYQASP$^{RQS}$.

All in all, the results summarized in Figures 9.1- 9.4 confirm the efficacy of both well-founded optimization and identification of *Guess&Check* programs.

## 9.4 Comparison with qasp

Starting from the best variants of PYQASP identified in the previous section, we compare them with QASP running the same back-end QBF solvers. As before, the selected back-end is identified by a superscript. In addition, we run a version of PYQASP capable of selecting automatically a suitable back-end solver for each instance, denoted by PYQASP$^{AUTO}$. This latter was obtained by applying to PYQASP the methodology used in the ME-ASP solver [53] for ASP. In particular, we measured some syntactic program features, the ones of ME-ASP augmented with the number of quantifiers, existential (resp. universal) atoms count, and existential (resp. universal) quantifiers to characterize QASP instances. Then, we used the *random forest* classification algorithm. We sampled about 25% of the instances (i.e., 1094 instances uniquely solved) from all benchmark domains, and split in 30% test set (329 instances) and

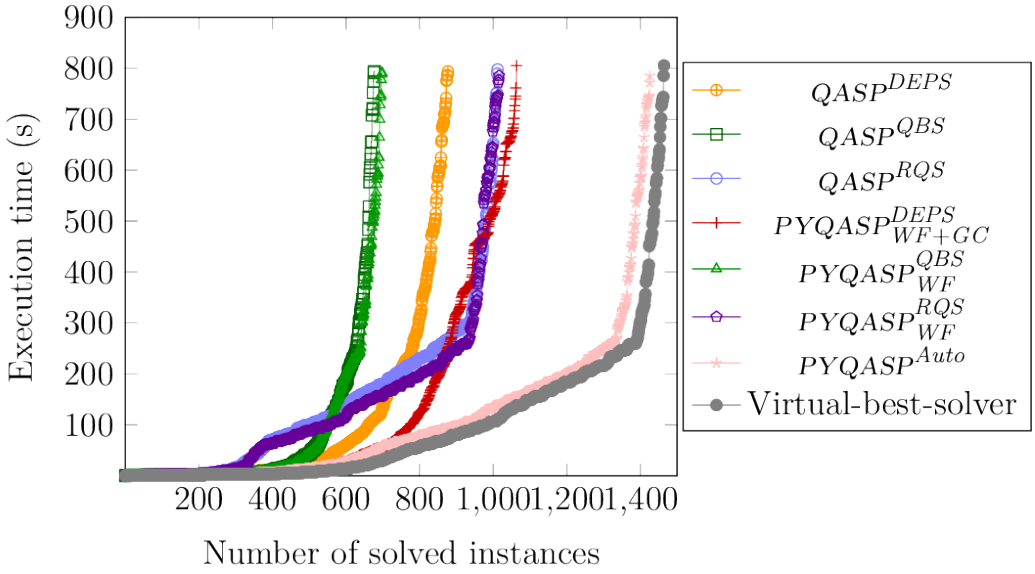


Figure 9.5: Comparison with QASP.

70% training set (765 instances), obtaining: 98% accuracy, 95% recall, 97% of f-measure, which is acceptable. As it is customary in the literature, to assess on the field the efficacy of the algorithm selection strategy, we also computed the Virtual Best Solver (VBS). VBS is the *ideal* system one can obtain by always selecting the best solver for each instance.

Obtained results are reported in the cactus plot of Figure 9.5. First of all, we note that PYQASP is faster and solves more instances than QASP no matter the back-end solver. In particular, $\text{PYQASP}^{DEPS}_{WF+GC}$ solves 186 instances more than $\text{QASP}^{DEPS}$, $\text{PYQASP}^{RQS}_{WF}$ solves 4 instances more than $\text{QASP}^{RQS}$, and $\text{PYQASP}^{QBS}_{WF}$ solves 21 instances more than $\text{QASP}^{QBS}$.

Diving into the details, we observed that PYQASP also uses less memory on average than QASP. Indeed, QASP used more than 12GB in some instances of PAR and AC, whereas PYQASP never exceeded the memory limit in these domains. This is due to a combination of factors. On the one hand, PYQASP never caches the entire program in main memory; on the other hand, the formulas built by PYQASP are smaller than the ones of QASP and this causes the back-end QBF solver to use less memory and be faster during the search.

Finally, as one might expect, the best solving method is $\text{PYQASP}^{AUTO}$. Comparing $\text{PYQASP}^{AUTO}$ with the VBS there is only a small gap (38 instances

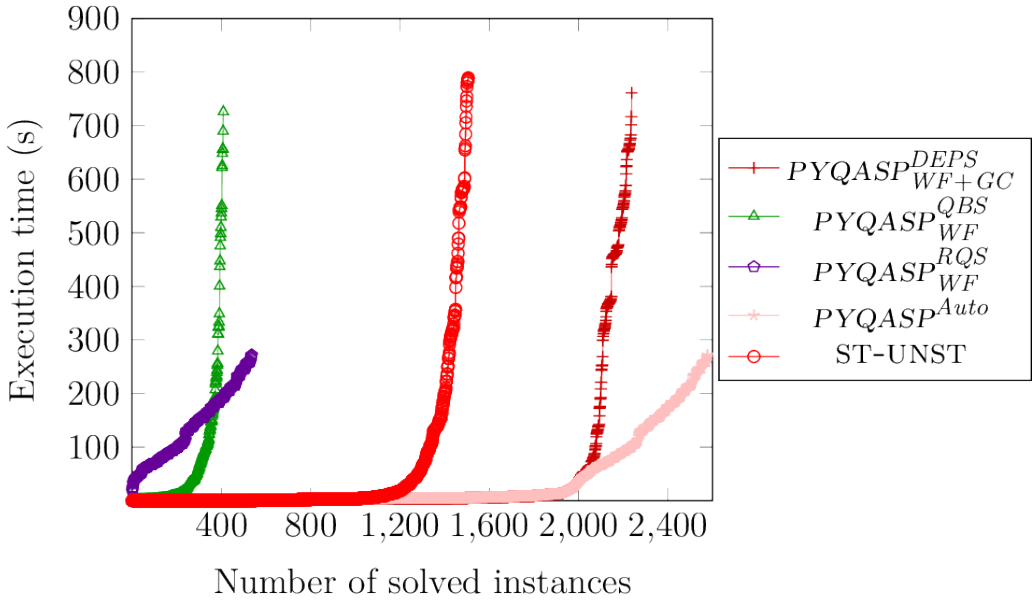


Figure 9.6: Comparison with ST-UNST.

overall). In particular, we observe that, in the majority of cases, the selector is able to pick the best method; it sometimes misses a suitable back-end (especially in MMC which is the smallest and less represented domain in the training set). As a result, PYQASP$^{AUTO}$ is generally effective in combining the strengths of all the back-end solvers. Indeed, PYQASP$^{AUTO}$ solves 363 instances more than PYQASP$^{DEPS}_{WF+GC}$ (i.e., the best variant of PYQASP with fixed back-end) and 414 instances more than QASP$^{RQS}$ (i.e., the best variant of QASP).

## 9.5 Comparison with Stable-unstable

In this section, we compare PYQASP with an efficient implementation of the stable-unstable semantics by Janhunen [46] on common benchmarks. In this comparison, we considered the best fixed back-end variants of PYQASP, PYQASP$^{AUTO}$, with Janhunen's solver [46], which is labeled ST-UNST. It is important to point out that ST-UNST can solve only problems on the second level of the PH (more on this in Related Work) and so, for a fair comparison, we considered the following benchmarks:

- the only problem in our suite having suitable complexity that is Paracoherent ASP (PAR)
- a set of hard 2-QBF instances generated according to the method by Amendola et al. [9]
- *point of no return* (PONR) benchmark introduced by Janhunen [46] to assess ST-UNST

Obtained results are summarized in the cactus plot of Figure 9.6. Analysing the results in each domain, we report that ST-UNST solves 60 instances of PAR, where the best fixed back-end version of PYQASP (namely, $\text{PYQASP}_{WF}^{RQS}$) solves 442. In PONR, ST-UNST solves 30 instances, where $\text{PYQASP}_{WF}^{RQS}$ solves 94. In QBF, ST-UNST solves 1416 instances, where $\text{PYQASP}_{WF+GC}^{DEPS}$ solves 2048. Finally, $\text{PYQASP}^{AUTO}$ is the best method overall, solving a total of 2578 instances, that is 1072 more instances than ST-UNST, which solves 1506 overall.

# Related Work

In this chapter, we discuss closely related works that have been proposed for enhancing the evaluation of logic programs. In particular, we investigate (i) proposed approaches for mitigating the effects of the grounding bottleneck in standard ASP systems; and (ii) compilation-based approaches for the evaluation of Datalog and stratified programs. Then, we shift the attention to solutions for modeling problems of higher computational complexity by focusing on ASP(Q) and possible alternatives.

## Efficient evaluation of logic programs

Previous studies demonstrated that ASP systems based on the Ground&Solve architecture are limited by the grounding bottleneck [15, 26]. Several approaches to overcome this issue have been proposed, which can be divided into three main classes: hybrid approaches, lazy-grounding, and compilation-based.

Hybrid approaches are based on the extension of the base language with additional constructs for connecting ASP solvers with external solvers. These include Constraints Answer Set Programming (CASP) [10, 11, 22, 58, 64], ASP Modulo Theories [38], and HEX programs [32]. While effective, these systems do not address the grounding bottleneck in ASP systems. Instead, they circumvent the issue by shifting the complexity from the logic program to external sources of computation such as constraint solvers and SMT solvers. Lazy-grounding systems perform the grounding of rules during the search for an answer set [15, 49, 51, 59, 66, 67]. In these approaches, a rule is instantiated only when its body is satisfied, thus grounding is done only for rules that are used during the search. The state-of-the-art lazy-grounding system Alpha [66, 67] combines lazy instantiation techniques with learning, conflict-based heuristics, restarts, phase saving, etc. Recently,

Bomanson et al. [15] implemented on top of ALPHA a technique for lazily rewriting aggregates as rules. In the same category as ALPHA is the DualGrounder [51], which performs lazy instantiation resorting to the multi-shot API of CLINGO. ALPHA and DualGrounder can outperform traditional ASP systems on grounding-intensive benchmarks and are comparable in performance (as shown in [51]). Compilation-based approaches, instead, offer the possibility to mitigate the effects of the grounding bottleneck while keeping the benefits of state-of-the-art approaches, but were limited to subprograms without aggregates that behave like constraints. To this end, in the first part of the thesis we proposed WASPPROP and PROASP systems. WASPPROP compiles programs with aggregates behaving like constraints into eager propagators and is built on top of the state-of-the-art ASP solver WASP. Instead, the PROASP system, provides a grounding-less compilation-based architecture, built on top of the GLUCOSE SAT-solver, for tight normal programs, thereby extending the class of compilable programs *beyond constraints*. One of the major differences between ALPHA and compilation techniques proposed so far is that the former discovers the space of propositional atoms during the search, whereas PROASP and WASPPROP compute it at the beginning. This choice might give advantages to ALPHA when the size of (the useful part of) the Herbrand base is already prohibitively large; on the other hand, it gives advantages to PROASP and WASPPROP both in terms of visibility of the search space and complete compatibility with standard systems. In fact, PROASP and WASPPROP are built on top of GLUCOSE and WASP respectively. This allowed us to exploit well-mature and studied techniques implemented in modern ASP and SAT solvers, whereas in ALPHA all these techniques were re-implemented to be blended with lazy-grounding. The experimental evaluation conducted in this part of the thesis revealed that ALPHA compared with WASPPROP shows similar performance on the grounding-intensive benchmarks supported by both systems, whereas, PROASP outperforms ALPHA on tight programs.

Compilation-based techniques have been proposed also for the evaluation of positive programs (i.e., Datalog programs) and programs with stratified negation. Recently, the SOUFFLE system by Jordan et al. [47], and other prototypes have been proposed [28]. For these classes of programs, it has been proved that the well-founded model is a total model, therefore it coincides with the unique stable model of such programs. In the general case (i.e., no restrictions are imposed in the usage of negation) this property does not hold and so these systems cannot compute the well-founded model. In this thesis,

we proposed a compilation-based approach for the evaluation of logic programs under well-founded semantics implemented in the system WF-COMP. This system is able to compile a non-ground normal logic program into an ad-hoc solver implementing the well-founded semantics. Generated solvers can be used for evaluating multiple program instances expressed as a set of facts.

## Modeling problems beyond NP

Among possible solutions for modeling problems beyond the NP class, ASP(Q) was revealed to be a suitable formalism for dealing with problems of such complexity. The first implementation for ASP(Q), namely QASP, was proposed by Amendola et al. [5].

In this thesis, we presented PYQASP, a new system for ASP(Q) that features both a memory-aware implementation in Python and a new optimized translation of ASP(Q) programs in QBF.

First, we observe that both PYQASP and QASP are based on the translation from ASP(Q) to QBF introduced by Amendola et al. [5], both resort to the ASPTOOLS for converting ASP programs to CNF formulas [44, 45], and both can be configured with several back-end QBF solvers. QASP is implemented in Java, whereas PYQASP is implemented in Python, which proved to be a very flexible and handy language to implement the composition of tools needed to develop a QBF-based system for ASP(Q). In this thesis, we presented PYQASP, a new system for ASP(Q) that features both a memory-aware implementation in Python and a new optimized translation of ASP(Q) programs in QBF.

QASP processes the entire ASP(Q) program rewriting in main memory, whereas PYQASP implements a more memory-aware algorithm that keeps at most one subprogram in main memory. This implementation choice was empirically demonstrated to overcome the high memory usage limiting the performance of QASP described by Amendola et al. [5]. QASP uses GRINGO [39] as grounder, whereas PYQASP can be configured to use both GRINGO and IDLV [17], and offers an interface that makes it easier to integrate external QBF solvers. It is worth pointing out that PYQASP supports novel rewriting techniques that result in more efficient encoding in QBF (see Chapters 7 and 8) that are absent in QASP.

Concerning closely-related formalisms that feature an implementation,

we mention the stable-unstable semantics [14], and quantified answer set semantics [37].

The stable-unstable semantics was first supported by a proof of concept prototype [14]; later, Janhunen [46] proposed an implementation based on a rewriting to plain ASP. The implementation proposed by Janhunen [46] employed ASPTOOLS for some pre-processing, but the transformations and the solving techniques are different w.r.t. PYQASP. Stable-unstable semantics can be used to model problems in the second level of the PH, thus our implementation can handle problems of higher complexity. From the usage point of view, we observe that in the system of Janhunen [46] the user is required to define the interface of modules (by means of ASP programs) and to manually combine the tools chain, whereas in PYQASP this is done in a more accessible way.

The quantified answer set semantics [37] was also implemented by resorting to a translation to QBF [37]. However, the difference in the semantics of quantifiers results in a quite different translation to QBF with respect to ASP(Q). Translations from quantified answer set semantics to ASP(Q) and back were proposed by Fandinno et al. [37] but never implemented.

# Conclusion

The goal of this thesis was to enhance the evaluation of logic programs by exploiting promising compilation-based approaches and efficient rewriting techniques for enhancing the evaluation of ASP(Q) programs.

In the first part of the thesis we proposed novel compilation-based approaches for evaluating logic programs both under answer set and well-founded semantics.

Compilation-based techniques have been proposed in the literature for mitigating that effects of grounding bottleneck problem the affects standard ASP systems. These techniques were revealed to be very effective but they were limited to programs behaving as constraints that do not contain aggregates. Starting from this limitation, we investigated possible solutions to tackle the grounding bottleneck on programs containing aggregates. Aggregates are widely used in ASP and, very likely, their grounding may cause a bottleneck. We ended up with a system that, automatically, compiles programs with aggregates into *eager* or *lazy* propagators that are further injected into the WASP solver. The proposed approach introduced significant performance improvements both in terms of solving time and memory consumption. Obtained results motivated our idea to move compilation boundaries beyond constraints. Thus, we studied the challenges in compiling such classes of programs and we proposed a grounding-less ASP-solving architecture that targets tight normal programs [34]. We proposed the PROASP system which is able to generate ad-hoc solvers for non-ground input programs that can be used to solve many instances of the compiled problem. Resulting solvers are, basically, ad-hoc versions of the well-mature SAT-solver GLUCOSE that has been customized with propagators that simulate rule inferences. Experimental evaluations confirmed the effectiveness of PROASP which outperformed state-of-the-art alternatives on grounding-intensive ASP benchmarks.

Moreover, compilation-based approaches have been proposed also for the

evaluation of Datalog programs and programs with stratified negation. For this kind of program answer sets and well-founded semantics coincide, therefore such approaches can evaluate these classes of programs also under the well-founded semantics but they cannot be used in the general case. In this thesis, we proposed a novel system that compiles logic programs in ad-hoc solvers that implements the well-founded semantics. Generated solvers have been empirically evaluated and obtained results demonstrate the effectiveness of the proposed approach both for positive programs and programs with not stratified negation. The improvements are significant on programs with negation while in the evaluation of positive programs, our implementation is among the fastest system in our experiments.

In the second part, instead, we focused on efficient implementations for evaluating ASP(Q) programs. More precisely, we proposed the PYQASP system featuring ad-hoc rewriting-based optimizations based on the syntactic structure of the input program. Such a system exploits the syntactic properties of ASP programs in order to apply ad-hoc optimizations resulting in more compact and affordable QBF encoding. The impact of proposed optimizations has been confirmed by obtained results that also highlighted a strong dependency on the selected back-end solver and the syntactic properties of the input program. To this end, we equipped PYQASP with an automatic back-end selection strategy, that delivered steady performance over varying problem instances. Indeed, PYQASP outperformed QASP pushing forward the state-of-the-art in ASP(Q) solving.

Regarding future works, we planned to:

- Extend the compilation technique to the entire ASP-Core 2 standard, providing ad-hoc solutions for the evaluation of programs both under well-founded and answer set semantics.
- Improve the proposed techniques by adopting more advanced data structures, for the efficient computation of the joins among body literals.
- Identify specific sub-classes of ASP programs that could benefit from ad-hoc compilation strategies.
- Optimize PYQASP encoding strategy by exploiting cautious reasoning.
- Improve PYQASP's algorithm selection model with extended training and a deeper tuning of parameters.

**Publications.** All the contributions described in the thesis have been subject to scientific publications.

- Wolfgang Faber, Giuseppe Mazzotta and Francesco Ricca. An efficient solver for asp(q). Theory and Practice of Logic Programming p. 1–17 (2023). https://doi.org/10.1017/S1471068423000121.

- Giuseppe Mazzotta, Francesco Ricca, and Carmine Dodaro. Compilation of aggregates in ASP systems. In AAAI, pages 5834–5841. AAAI Press, 2022. This contribution obtained the "*Outstanding student paper award honorable mention of AAAI2022*".

- Carmine Dodaro, Giuseppe Mazzotta, and Francesco Ricca. Compilation of tight ASP programs. [Accepted for publication at 26th European Conference on Artificial Intelligence, ECAI2023]

- Andrea Cuteri, Giuseppe Mazzotta, and Francesco Ricca. Compilation-based techniques for evaluating normal logic programs under the well-founded semantics. Proceedings of the 38th Italian Conference on Computational Logic, volume 3428 of CEUR Workshop Proceedings. CEUR-WS.org, 2023.

Moreover, during my Ph.D. other contributions related to the work proposed in this thesis have been published and some others are currently under revision. More precisely, we proposed contributions concerning applications of both ASP and ASP(Q) but also extensions of ASP for unit testing. In particular such contributions are listed below:

- Pierpaolo Bellusci, Giuseppe Mazzotta, and Francesco Ricca. Modelling the outlier detection problem in ASP(Q). In Practical Aspects of Declarative Languages - 24th International Symposium, PADL 2022, Proceedings, volume 13165 of Lecture Notes in Computer Science, pages 15–23. Springer, 2022.

- Wolfgang Faber, Giuseppe Mazzotta, and Francesco Ricca. Enhancing ASP(Q) evaluation (Discussion Paper). [Submitted for the 23rd International Conference of the Italian Association for Artificial Intelligence (AIxIA 2023)]

- Giovanni Amendola, Tobias Berei, Giuseppe Mazzotta, and Francesco Ricca. Unit Testing in ASP: Language and Test-Driven Development

Environment. [Under revision for publication in journal of Theory and Practice of Logic Programming]

- Ignacio Huitzil, Giuseppe Mazzotta, Rafael Peñaloza and Francesco Ricca. ASP-based Axiom Pinpointing for Description Logics. [Accepted for publication at 36th International Workshop on Description Logics, DL2023]

- Carmine Dodaro, Giuseppe Mazzotta, and Francesco Ricca. Compilation of ASP programs: Recent developments (short paper). In Proceedings of the Discussion Papers - 22nd International Conference of the Italian Association for Artificial Intelligence (AIxIA 2022 DP), volume 3419 of CEUR Workshop Proceedings, pages 55–64, 2022

- Giuseppe Mazzotta. Compilation of aggregates in ASP. In Proceedings 37th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2021, 20-27th September 2021, volume 345 of EPTCS, pages 286–295, 2021.

# Bibliography


[1] Mario Alviano, Giovanni Amendola, Carmine Dodaro, Nicola Leone, Marco Maratea, and Francesco Ricca. Evaluation of disjunctive programs in WASP. In Marcello Balduccini, Yuliya Lierler, and Stefan Woltran, editors, *Logic Programming and Nonmonotonic Reasoning - 15th International Conference, LPNMR 2019, Philadelphia, PA, USA, June 3-7, 2019, Proceedings*, volume 11481 of *Lecture Notes in Computer Science*, pages 241–255. Springer, 2019.

[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 *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*, pages 215–221, 2017.

[3] Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. Advances in WASP. In *LPNMR*, volume 9345 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2015.

[4] Mario Alviano, Carmine Dodaro, and Marco Maratea. Shared aggregate sets in answer set programming. *TPLP*, 18(3-4):301–318, 2018.

[5] Giovanni Amendola, Bernardo Cuteri, Francesco Ricca, and Mirek Truszczynski. Solving problems in the PH with ASP(Q). In *Proceedings of LPNMR*, volume 13416 of *LNCS*, pages 373–386. Springer, 2022.

[6] Giovanni Amendola, Carmine Dodaro, Wolfgang Faber, and Francesco Ricca. Paracoherent answer set computation. *Artif. Intell.*, 299:103519, 2021.

[7] Giovanni Amendola, Gianluigi Greco, Nicola Leone, and Pierfrancesco Veltri. Modeling and reasoning about NTU games via answer set programming. In *IJCAI*, pages 38–45. IJCAI/AAAI Press, 2016.

[8] Giovanni Amendola, Francesco Ricca, and Miroslaw Truszczynski. Beyond NP: quantifying over answer sets. *TPLP*, 19(5-6):705–721, 2019.

[9] Giovanni Amendola, Francesco Ricca, and Miroslaw Truszczynski. New models for generating hard random boolean formulas and disjunctive logic programs. *Artif. Intell.*, 279, 2020.

[10] Rehan Abdul Aziz, Geoffrey Chu, and Peter J. Stuckey. Stable model semantics for founded bounds. *TPLP*, 13(4-5):517–532, 2013.

[11] Marcello Balduccini and Yuliya Lierler. Constraint answer set solver EZCSP and why integration schemas matter. *TPLP*, 17(4):462–515, 2017.

[12] Chitta Baral and Michael Gelfond. Logic programming and knowledge representation. *The Journal of Logic Programming*, 19:73–148, 1994.

[13] Rachel Ben-Eliyahu and Rina Dechter. Propositional semantics for disjunctive logic programs. *Ann. Math. Artif. Intell.*, 12(1-2):53–87, 1994.

[14] Bart Bogaerts, Tomi Janhunen, and Shahab Tasharrofi. Stable-unstable semantics: Beyond NP with normal logic programs. *TPLP*, 16(5-6):570–586, 2016.

[15] Jori Bomanson, Tomi Janhunen, and Antonius Weinzierl. Enhancing lazy grounding with lazy normalization in answer-set programming. In *AAAI*, pages 2694–2702. AAAI Press, 2019.

[16] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.

[17] Francesco Calimeri, Carmine Dodaro, Davide Fuscà, Simona Perri, and Jessica Zangari. Efficiently coupling the I-DLV grounder with ASP solvers. *TPLP*, 20(2):205–224, 2020.

[18] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco

Maratea, Francesco Ricca, and Torsten Schaub. Asp-core-2 input language format. *TPLP*, 20(2):294–309, 2020.

[19] Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari. I-DLV: the new intelligent grounder of DLV. *Intelligenza Artificiale*, 11(1):5–20, 2017.

[20] Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca. Design and results of the fifth answer set programming competition. *Artif. Intell.*, 231:151–181, 2016.

[21] Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca. The third open answer set programming competition. *Theory Pract. Log. Program.*, 14(1):117–135, 2014.

[22] Broes De Cat, Marc Denecker, Maurice Bruynooghe, and Peter J. Stuckey. Lazy model expansion: Interleaving grounding with search. *J. Artif. Intell. Res.*, 52:235–286, 2015.

[23] Keith L. Clark. Negation as failure. In *Logic and Data Bases*, Advances in Data Base Theory, pages 293–322, New York, 1977. Plemum Press.

[24] Bernardo Cuteri, Carmine Dodaro, Francesco Ricca, and Peter Schüller. Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis. *Theory Pract. Log. Program.*, 17(5-6):780–799, 2017.

[25] Bernardo Cuteri, Carmine Dodaro, Francesco Ricca, and Peter Schüller. Partial compilation of ASP programs. *Theory Pract. Log. Program.*, 19(5-6):857–873, 2019.

[26] Bernardo Cuteri, Carmine Dodaro, Francesco Ricca, and Peter Schüller. Overcoming the grounding bottleneck due to constraints in ASP solving: Constraints become propagators. In *IJCAI*, pages 1688–1694. ijcai.org, 2020.

[27] Bernardo Cuteri, Kristian Reale, and Francesco Ricca. A logic-based question answering system for cultural heritage. In *JELIA*, volume 11468 of *Lecture Notes in Computer Science*, pages 526–541. Springer, 2019.

[28] Bernardo Cuteri and Francesco Ricca. A compiler for stratified datalog programs: preliminary results. In Sergio Flesca, Sergio Greco, Elio

Masciari, and Domenico Saccà, editors, *Proceedings of the 25th Italian Symposium on Advanced Database Systems, Squillace Lido (Catanzaro), Italy, June 25-29, 2017*, volume 2037 of *CEUR Workshop Proceedings*, page 158. CEUR-WS.org, 2017.

[29] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[30] Carmine Dodaro and Marco Maratea. Nurse scheduling via answer set programming. In *LPNMR*, volume 10377 of *Lecture Notes in Computer Science*, pages 301–307. Springer, 2017.

[31] Thomas Eiter and Georg Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell.*, 15(3-4):289–323, 1995.

[32] Thomas Eiter, Christoph Redl, and Peter Schüller. Problem solving using the HEX family. In *Computational Models of Rationality*, pages 150–174. College Publications, 2016.

[33] Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of answer set programming. *AI Magazine*, 37(3):53–68, 2016.

[34] Esra Erdem and Vladimir Lifschitz. Tight logic programs. *Theory Pract. Log. Program.*, 3(4-5):499–518, 2003.

[35] Esra Erdem and Volkan Patoglu. Applications of ASP in robotics. *Künstliche Intell.*, 32(2-3):143–149, 2018.

[36] Wolfgang Faber, Gerald Pfeifer, and Nicola Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artif. Intell.*, 175(1):278–298, 2011.

[37] Jorge Fandinno, François Laferrière, Javier Romero, Torsten Schaub, and Tran Cao Son. Planning with incomplete information in quantified answer set programming. *TPLP*, 21(5):663–679, 2021.

[38] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In *ICLP (Technical Communications)*, volume 52 of *OASICS*, pages 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[39] Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. Advances in *gringo* series 3. In *LPNMR*, volume 6645 of *Lecture Notes in Computer Science*, pages 345–351. Springer, 2011.

[40] Martin Gebser, Nicola Leone, Marco Maratea, Simona Perri, Francesco Ricca, and Torsten Schaub. Evaluation techniques and systems for answer set programming: a survey. In *IJCAI*, pages 5450–5456. ijcai.org, 2018.

[41] Martin Gebser, Marco Maratea, and Francesco Ricca. The sixth answer set programming competition. *J. Artif. Intell. Res.*, 60:41–95, 2017.

[42] Martin Gebser, Marco Maratea, and Francesco Ricca. The sixth answer set programming competition. *Journal of Artificial Intelligence Research*, 60:41–95, 2017.

[43] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Gener. Comput.*, 9(3/4):365–386, 1991.

[44] Tomi Janhunen. Representing normal programs with clauses. In Ramón López de Mántaras and Lorenza Saitta, editors, *Proceedings of ECAI'2004.*, pages 358–362. IOS Press, 2004.

[45] Tomi Janhunen. Cross-translating answer set programs using the ASP-TOOLS collection. *Künstliche Intell.*, 32(2-3):183–184, 2018.

[46] Tomi Janhunen. Implementing stable-unstable semantics with ASP-TOOLS and clingo. In *PADL 2022, Proceedings*, volume 13165 of *LNCS*, pages 135–153. Springer, 2022.

[47] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 422–430. Springer, 2016.

[48] Benjamin Kaufmann, Nicola Leone, Simona Perri, and Torsten Schaub. Grounding and solving in answer set programming. *AI Mag.*, 37(3):25–32, 2016.

[49] Claire Lefèvre and Pascal Nicolas. The first version of a new ASP solver : Asperix. In *LPNMR*, volume 5753 of *Lecture Notes in Computer Science*, pages 522–527. Springer, 2009.

[50] Senlin Liang, Paul Fodor, Hui Wan, and Michael Kifer. Openrulebench: an analysis of the performance of rule engines. In Juan Quemada, Gonzalo León, Yoëlle S. Maarek, and Wolfgang Nejdl, editors, *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, pages 601–610. ACM, 2009.

[51] Yuliya Lierler and Justin Robbins. Dualgrounder: Lazy instantiation via clingo multi-shot framework. In *JELIA*, volume 12678 of *Lecture Notes in Computer Science*, pages 435–441. Springer, 2021.

[52] Vladimir Lifschitz. Answer set programming and plan generation. *Artif. Intell.*, 138(1-2):39–54, 2002.

[53] Marco Maratea, Luca Pulina, and Francesco Ricca. A multi-engine approach to answer-set programming. *TPLP*, 14(6):841–868, 2014.

[54] João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2021.

[55] Giuseppe Mazzotta, Francesco Ricca, and Carmine Dodaro. Compilation of aggregates in ASP systems. In *AAAI*, pages 5834–5841. AAAI Press, 2022.

[56] Arindam Mitra, Peter Clark, Oyvind Tafjord, and Chitta Baral. Declarative question answering over knowledge bases containing natural language text with answer set programming. In *AAAI*, pages 3003–3010. AAAI Press, 2019.

[57] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[58] Max Ostrowski and Torsten Schaub. ASP modulo CSP: the clingcon system. *TPLP*, 12(4-5):485–503, 2012.

[59] Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. GASP: answer set programming with lazy grounding. *Fundam. Informaticae*, 96(3):297–322, 2009.

[60] Luca Pulina and Martina Seidl. The 2016 and 2017 QBF solvers evaluations (qbfeval'16 and qbfeval'17). *Artif. Intell.*, 274:224–248, 2019.

[61] Peter Schüller. Modeling variations of first-order horn abduction in answer set programming. *Fundam. Informaticae*, 149(1-2):159–207, 2016.

[62] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[63] Tran Cao Son, Enrico Pontelli, Marcello Balduccini, and Torsten Schaub. Answer set planning: A survey. *Theory Pract. Log. Program.*, 23(1):226–298, 2023.

[64] Benjamin Susman and Yuliya Lierler. Smt-based constraint answer set solver EZSMT (system description). In *ICLP (Technical Communications)*, volume 52 of *OASIcs*, pages 1:1–1:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[65] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.

[66] Antonius Weinzierl. Blending lazy-grounding and CDNL search for answer-set solving. In *LPNMR*, volume 10377 of *Lecture Notes in Computer Science*, pages 191–204. Springer, 2017.

[67] Antonius Weinzierl, Richard Taupe, and Gerhard Friedrich. Advancing lazy-grounding ASP solving techniques - restarts, phase saving, heuristics, and more. *Theory Pract. Log. Program.*, 20(5):609–624, 2020.

*In conclusione di questo lavoro di tesi, vorrei esprimere la mia più profonda gratitudine a tutti coloro che mi sono stati vicino in questo bellissimo viaggio che spero non finisca mai. É stato un percorso difficile, pieno di momenti no, ma ho avuto la fortuna di avere al mio fianco due mentori fenomenali, due persone straordinarie come Francesco Ricca e Carmine Dodaro. Grazie alla vostra perseverenza e all'impegno che mettete in tutto ció che fate, ho imparato cosa significa inseguire un obiettivo o affrontare una nuova sfida, il che mi ha permesso di superare limiti per me insormontabili. Non smetteró mai di dirvi grazie, non sarebbe mai abbastanza. Un altro ringraziamento va a tutte le persone che ho conosciuto nel dipartimento che ormai reputo una seconda famiglia. Ho condiviso con voi tutte le mie incertezze, le mie paure ma sopratutto tanti momenti indimenticabili. Infine, vorrei ringraziare chi per me c'è sempre stato e sempre ci sarà, ovunque io sia e qualsiasi cosa io faccia. Ringrazio la mia famiglia e tutti i miei amici per aver sempre creduto in me, anche quando ero io il primo a non farlo.*
***Grazie di cuore a tutti**.*