GOAL-DIRECTED ANSWER SET PROGRAMMING

by

Kyle Brandon Marple

APPROVED BY SUPERVISORY COMMITTEE:

Gopal Gupta, Chair

Vibhav Gogate

Kevin W. Hamlen

Haim Schweitzer
GOAL-DIRECTED ANSWER SET PROGRAMMING

by

KYLE BRANDON MARPLE, BS, MS

DISSERTATION
Presented to the Faculty of
The University of Texas at Dallas
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY IN
COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS
August 2014
ACKNOWLEDGMENTS

This dissertation would not exist without the help and support of several wonderful individuals, and I am grateful to them all. Dr. Gopal Gupta, my adviser, introduced me to both logic programming and answer set programming. Without his guidance and support, I would never have made it this far. Dr. Feliks Kluźniak provided me with invaluable feedback and constructive criticism, and told me the truth even when I did not want to hear it. Without him, Galliwasp might still be an incomprehensible mass of C code, and much of my work would have been impossible. I thank Dr. Ajay Bansal and Dr. Richard Min, whose work provided me with the foundation for my own. I would also like to thank those who helped kindle my original interest in computer science: Steve Jackson of Lawton High School, who taught the first programming course I ever took; and Professor Feridoon Moinian and Dr. Chao Zhao of Cameron University, who helped turn my interest into passion. Finally, I would not be where I am today without my father and grandmother, who have supported me through it all.

June 2014
This dissertation was produced in accordance with guidelines which permit the inclusion as part of the dissertation the text of an original paper or papers submitted for publication. The dissertation must still conform to all other requirements explained in the "Guide for the Preparation of Master's Theses and Doctoral Dissertations at The University of Texas at Dallas." It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for this dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts which provide logical bridges between different manuscripts are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the student’s contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the dissertation attest to the accuracy of this statement.
Answer Set Programming (ASP) provides an attractive means for integrating non-monotonic reasoning into logic programming. However, current ASP solvers are implemented using bottom-up execution algorithms. That is, none of the current approaches to implementation are goal-directed. This results in a number of issues which prevent ASP from being adopted on a larger scale, including the need to compute an entire answer set for any query and the ability of minor inconsistencies to invalidate solutions in unrelated areas of a knowledgebase. This dissertation presents a goal-directed method for executing answer set programs in the style of Selective Linear Definite clause resolution (SLD resolution), and discusses its advantages and how it addresses the first of the problems listed above. The implementation of the algorithm in the Galliwasp system is also presented and its performance is compared to that of other popular ASP solvers. Next, optimization techniques developed to improve performance are discussed and their advantages and disadvantages compared, along with the performance of various combinations of the techniques. Finally, Dynamic Consistency Checking (DCC), a technique for querying inconsistent databases which addresses the second problem listed above, is covered.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>PREFACE</td>
<td>v</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xii</td>
</tr>
<tr>
<td>CHAPTER 1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Structure of the Dissertation</td>
<td>3</td>
</tr>
<tr>
<td>CHAPTER 2 BACKGROUND</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Overview</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Answer Set Programming</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Coinductive Logic Programming</td>
<td>9</td>
</tr>
<tr>
<td>CHAPTER 3 GOAL-DIRECTED EXECUTION OF ANSWER SET PROGRAMS</td>
<td>11</td>
</tr>
<tr>
<td>3.1 Acknowledgements</td>
<td>11</td>
</tr>
<tr>
<td>3.2 Overview</td>
<td>11</td>
</tr>
<tr>
<td>3.3 Goal-directed ASP Issues</td>
<td>12</td>
</tr>
<tr>
<td>3.3.1 Ordinary Rules</td>
<td>14</td>
</tr>
<tr>
<td>3.3.2 OLON Rules</td>
<td>16</td>
</tr>
<tr>
<td>3.3.3 Coinductive Success Under ASP</td>
<td>17</td>
</tr>
<tr>
<td>3.3.4 NMR Consistency Check</td>
<td>19</td>
</tr>
<tr>
<td>3.4 Goal-directed Execution of Answer Set Programs</td>
<td>20</td>
</tr>
<tr>
<td>3.4.1 Dual Rules</td>
<td>21</td>
</tr>
<tr>
<td>3.4.2 Goal-directed Method for Computing Answer Sets</td>
<td>21</td>
</tr>
<tr>
<td>3.5 Soundness and Correctness of the Goal-directed Method</td>
<td>24</td>
</tr>
<tr>
<td>3.5.1 Relevance</td>
<td>24</td>
</tr>
</tbody>
</table>
3.5.2 Soundness ................................................. 26  
3.5.3 Completeness ............................................ 27  
3.6 Discussion and Related Work .................................. 29  

CHAPTER 4  GALLIWASP: A GOAL-DIRECTED ANSWER SET SOLVER ........................................ 31  
4.1 Acknowledgements ............................................. 31  
4.2 Overview ...................................................... 31  
4.3 System Architecture of Galliwasp .............................. 32  
4.3.1 Compiler .................................................. 32  
4.3.2 Interpreter ............................................... 37  
4.4 Performance Results .......................................... 38  
4.4.1 Rule and Goal Ordering ................................... 40  
4.5 Related Work .................................................. 42  

CHAPTER 5  IMPROVING THE PERFORMANCE OF GALLIWASP .................................................. 43  
5.1 Overview ...................................................... 43  
5.2 Compiler-based Optimization .................................. 44  
5.2.1 Program Simplification ................................... 44  
5.2.2 Factorization of Common Goals ........................... 46  
5.3 NMR Check Optimization ..................................... 51  
5.4 CHS-based Optimization ...................................... 53  
5.5 Comparative Performance Results ........................... 54  
5.5.1 Compiler Performance ..................................... 54  
5.5.2 Interpreter Performance ................................... 56  
5.6 Related Work .................................................... 60  

CHAPTER 6  DYNAMIC CONSISTENCY CHECKING ......................................................... 61  
6.1 Acknowledgements ............................................. 61  
6.2 Overview ...................................................... 61  
6.3 Dynamic Consistency Checking ............................... 63  
6.3.1 Relevance Under DCC ..................................... 64  
6.3.2 Execution with DCC ....................................... 66
C.1.5 Grounding ................................................................. 91
C.1.6 Basic Usage ............................................................ 92
C.1.7 Optimization Statements ......................................... 94
C.1.8 Dynamic Consistency Checking (DCC) ....................... 94
C.2 Compiler Usage and Options ........................................ 95
C.3 Interpreter Usage and Options .................................... 97
C.4 Compiled Program Format .......................................... 103
  C.4.1 CPF Grammar ...................................................... 104
REFERENCES ........................................................................ 112
VITA
LIST OF FIGURES

4.1 Galliwasp’s system architecture ........................................ 33
LIST OF TABLES

4.1 Comparative performance results ............................................. 39
5.1 Compilation times (in seconds) ................................................. 55
5.2 Compiled program sizes (in bytes) ............................................. 55
5.3 Interpreter performance with various optimization combinations (1 of 2) .... 57
5.4 Interpreter performance with various optimization combinations (2 of 2) .... 58
6.1 DCC versus non-DCC performance results ..................................... 70
CHAPTER 1
INTRODUCTION

1.1 Overview

Answer Set Programming (ASP) incorporates non-monotonic reasoning into logic programming, providing an elegant means of developing non-monotonic reasoning applications. Most logic programming semantics are monotonic, meaning that the addition of a new rule to an existing program will never reduce the set of rules which previously held. Conversely, non-monotonicity means that a new rule can falsify information that would otherwise be true. This aspect of ASP makes it particularly useful for a number of applications. As a result, ASP has gained significant acceptance, particularly within academia, resulting in considerable work to develop the paradigm. It has been used for planning, scheduling, default reasoning, reasoning about actions (Baral 2003) and various other applications. However, there are still three significant obstacles to the adoption of ASP on a larger scale: (i) complete answer sets must be computed for every query, (ii) minor inconsistencies can render an entire knowledgebase useless and (iii) all current ASP solvers are restricted to grounded programs. The first two obstacles are addressed in this dissertation; the third is the subject of ongoing research (Salazar et al. 2014).

Numerous ASP implementations have been developed, including DLV (Leone et al. 2006) and smodels (Niemelä and Simons 1997) as well as SAT-based solvers like cmodels (Giunchiglia et al. 2004) and clasp (Gebser et al. 2007a). However, all of these implementations rely on bottom-up execution methods. Bottom up strategies require the computation of a complete answer set for each query, leading to the first obstacle listed above. When executing a query, a user generally only cares about the solution to that query. However,
extracting this information from a complete answer set can be difficult, particularly in the case of large knowledgebases. Programs may need to be rewritten for each query, adding constraints to remove irrelevant information. In contrast, a top-down, goal-directed execution strategy will allow for the systematic construction of partial answer sets which contain only those additional elements which are used to find a solution. Other attempts to create a goal-directed implementation have been made, but these place restrictions on the programs and/or queries accepted (Bonatti et al., 2008).

The primary contributions of this dissertation are a goal-directed method that accepts ASP programs and queries, techniques developed to improve its performance, and a means of querying inconsistent knowledgebases (dynamic consistency checking). The development of a goal-directed method for execution ASP programs was regarded as difficult, if not impossible (Baral, 2003). However, we not only present such a method, but demonstrate that it can be efficiently implemented using the included techniques, performing comparably to other state-of-the-art ASP solvers. Additionally, dynamic consistency checking extends our method to address the second obstacle listed above, allowing inconsistent, unrelated information in a knowledgebase to be ignored rather than permitting it to invalidate the whole. Thus, our work presents solutions to the first two obstacles listed above. Additionally, our goal-directed method provides a major stepping stone towards addressing the third and final obstacle, the restriction to grounded programs.

In addition to its direct contributions, our goal-directed method also simplifies the integration of ASP with other logic-programming extensions. Desirable extensions of ASP include constraints, probabilities and parallelism, among others. Well-accepted techniques for each of these exist within logic programming: CLP(R) for constraints (Jaffar and Lassez, 1987), ProbLog for probabilities (De Raedt et al., 2007) and Or-parallelism for parallel execution (Gupta et al., 2001). However, all of these were designed with top-down execution in mind. While it might be possible to adapt these techniques to bottom-up ASP, goal-directed
ASP provides a more natural starting point, being much closer to the environment for which they were originally designed.

The techniques we present make goal-directed ASP both possible and practical. Additionally, we address two of the major obstacles to wider adoption of ASP as a whole and provide a foundation for addressing the third. Our work also opens a number of doors for extending ASP with techniques developed for other areas of logic programming. The work presented in this dissertation therefore constitutes a significant contribution to the field of answer set programming.

1.2 Structure of the Dissertation

In this section, we provide the layout of the remaining chapters and a summary of each. The appendixes are also covered briefly.

Chapter 2 provides background information necessary to understand the remainder of the dissertation. We define the syntax and semantics of ASP programs and introduce the Gelfond-Lifschitz method for finding answer sets. We also discuss coinductive logic-programming, upon which our goal-directed method is based.

Chapter 3 introduces our goal-directed method and discusses aspects of ASP that are of particular concern to goal-directed execution. These include the odd loops over negation which produce ASP’s non-monotonicity, the need for consistency checking and restrictions on when success may occur. We examine how our method addresses each of these concerns and prove its soundness and completeness.

Chapter 4 introduces Galliwasp, an ASP solver which implements our goal-directed method. We examine its system architecture, design and implementation. Additionally, we compare its performance to that of several popular ASP solvers.

Chapter 5 discusses the various techniques that we have developed for improving the performance of Galliwasp while leaving our basic goal-directed algorithm largely unchanged.
Particular attention is given to the affects each technique may have on the correctness of our method. Finally, the performance of the various techniques is compared using several metrics.

Chapter 6 covers dynamic consistency checking, our method for querying inconsistent ASP programs. We begin by defining the properties that our method should satisfy, namely that it should remain as close to the original ASP semantics as possible. Our method is developed by examining possible solutions and refining them to satisfy our requirements. We then examine execution using the resulting technique and prove that it is consistent with the ASP semantics for programs which have at least one answer set. We conclude by discussing the advantages of DCC and comparing Galliwasp’s performance with and without it.

Finally, we discuss future work and draw conclusions in chapter 7. Both ongoing and potential work is examined and the contributions of the dissertation are reviewed.

The dissertation also includes three appendixes. Appendix A discusses aspects of ASP not covered in other chapters, as well as our handling of them. Appendix B covers an encoding of the Towers of Hanoi problem which is optimized for goal-directed execution. Finally, appendix C contains the user manual for Galliwasp.
CHAPTER 2
BACKGROUND

2.1 Overview

As our goal-directed method is designed for ASP and based on coinductive logic programming, an understanding of both concepts is needed to understand the remainder of this dissertation. In this chapter, we provide a brief overview of each topic. Answer Set Programming is discussed in section 2.2 and coinductive logic programming in section 2.3.

2.2 Answer Set Programming

Answer Set Programming (ASP) \cite{Gelfond1988, Gelfond2002} is a type of declarative logic programming which incorporates non-monotonic reasoning. Thus, understanding ASP requires basic knowledge of logic programming in general. Logic programming is based on the idea that formal logic, and first order logic in particular, can be used a programming language.

For the purpose of this dissertation, we restrict our discussion of logic programming, and our definition of ASP, to grounded normal logic programs (NLPs). NLPs are used throughout logic programming, providing a common syntax that can be interpreted according to many different semantics. ASP supports additional language features, but they are largely syntactic. These features, and our techniques for handling them, are detailed in appendix A. With this restriction in place, we can describe ASP syntax with the following definitions:

**Definition 1.** In grounded NLPs, an **atom** is a predicate symbol with \( n \) arguments of the form \( p(t_1, \ldots, t_n), n \geq 0 \), where each \( t_i \) is a constant. Constants can be either integers
or strings of letters, numbers and underscores which begin with either an underscore or a lower-case letter. When $n = 0$, the parenthesis are omitted. The negation of an atom is indicated by the keyword “not”, for example, not p. Thus, p, queen(1,12) and not false are all atoms, with not false being a negated atom as well.

**Definition 2.** A basic literal is an atom or a negated atom. A positive literal is an atom while a negative literal is a negated atom. Unless otherwise specified, “literal” refers to a basic literal.

**Definition 3.** A rule or clause in ASP takes one of the following forms:

1. $l_0 :- l_1, \ldots, l_n$.
2. $:- l_0, \ldots, l_n$.

where $n \geq 0$ and each $l_i$ is a literal. A rule has two parts, the head and the body, separated by the consequence operator, $:-$. When interpreting an individual rule, the head succeeds if every literal in the body succeeds. In general, the consequence operator may be read as “if” and commas as “and”. So, the rule:

$$ p :- \text{not} \ q, \ r. $$

may be read as “p if not q and r”. That is, p succeeds if not q and r succeed.

Either the head or the body of a clause may be empty. When the body is empty, the clause is referred to as a fact, and the head must always succeed. ASP interprets rules with an empty head as having the keyword false for their head. This means that execution must fail if the body of a headless rule succeeds and, conversely, the body must fail for execution to succeed. As we will discuss below, this interpretation of headless rules is one source of non-monotonicity in ASP.

**Definition 4.** A normal logic program (NLP) or ASP program consists of a finite set of rules.
Note that the restriction to grounded programs is a limitation of current ASP implementations. Ungrounded ASP programs may be written, but all current ASP solvers require that they be grounded prior to execution. An ungrounded NLP permits atoms to have variables as arguments. Grounding simply removes these variables by enumerating all possible combinations of their bindings. For example, given an ungrounded rule:

\[
p : - a(X), b(Y).
\]

where \(X\) and \(Y\) can each be bound to 1 or 2, grounding will produce the rules:

\[
\begin{align*}
p : & - a(1), b(1). \\
p : & - a(1), b(2). \\
p : & - a(2), b(1). \\
p : & - a(2), b(2). 
\end{align*}
\]

The information above describes what constitutes an ASP program, but it tells us nothing about what a program means. ASP is based on the stable model semantics, which attempts to give a better semantics to negation as failure (NAF), also called default negation (\texttt{not p} holds if we fail to establish \(p\)) (Gelfond and Lifschitz, 1988). Consequently, the semantics of ASP are defined in terms of the Gelfond-Lifschitz Transformation (GL method) for computing stable models (answer sets) (Baral, 2003).

**Definition 5 (Gelfond-Lifschitz Transformation).** For a grounded ASP program \(P\) and candidate answer set \(A\), a residual program \(R\) is created as follows: for each literal \(L \in A\)

1. Remove any rules in \(P\) with \texttt{not} \(L\) in the body.

2. Remove any negative literals from the remaining rules’ bodies.

Let \(F\) be the least fixed point of \(R\). If \(F = A\), then \(A\) is an answer set of \(P\).

The GL method is bottom-up by nature, in that a complete answer set must be guessed and then tested. Consider the application of the GL method to the following simple program:
\[
\begin{align*}
p & : - \text{not } q. \\
q & : - \text{not } p.
\end{align*}
\]

A candidate answer set \(A\) must first be guessed. As answer sets are traditionally restricted to positive literals, there are three possibilities: \(\{p\}\), \(\{q\}\) and \(\{p, q\}\). Once \(A\) has been chosen, the residual program \(R\) is computed. For \(A = \{p\}\), the rule for \(q\) is removed first, as its body contains \text{not } p. The goal \text{not } q is then removed from the body of the remaining rule. The resulting \(R\) consists of a single fact for \(p\). Finally, the least fixed point of \(R\), \(F = \{p\}\), is computed and compared to \(A\). Since \(F = A\), \(\{p\}\) is an answer set of the original program. Repeating the process for \(\{q\}\) will show that it is also a valid answer set. However, for \(A = \{p, q\}\), both rules will be removed when computing \(R\), leaving it empty. Thus \(\{p, q\}\) is not an answer set of the original program.

The final aspect of ASP that must be addressed is its non-monotonicity. Under a monotonic logic programming language, adding a rule to a program will never reduce the set of literals which could previously succeed. However, this is not true of ASP. ASP’s non-monotonicity comes from rules of the form
\[
:- q_1, \ldots, q_n.
\]
or
\[
p : - q_1, \ldots, q_n, \text{not } p.
\]
Consider an example rule:
\[
p : - q, \text{not } p.
\]
Under the GL method above, this rule prevents \(p\) and \(q\) from being in an answer set, unless \(p\) is in the answer set through another rule. If \(p\) succeeds through another rule, the presence of \text{not } p in this rule will lead to its removal from the residual program. However, even though \(q\) may succeed through other rules, the above rule prevents it from being in an answer set unless \(p\) succeeds through another rule, making ASP non-monotonic.
2.3 Coinductive Logic Programming

The foundation of our goal-directed method lies in coinductive logic programming (co-LP) (Gupta et al., 2007). While inductive logic programming deals with the least fixed points (lfp) of logic programs, co-LP deals with their greatest fixed points (gfp).

To explain co-LP, we must examine how it differs from ordinary logic programming. SLD resolution, used in inductive logic programming, systematically computes elements of the least fixed points of logic programs via call expansion and backtracking. Its co-LP counterpart, co-SLD resolution, does the same for elements of greatest fixed points (Simon, 2006; Gupta et al., 2007). Both procedures share a number of common elements:

- Calls are expanded by matching them against the heads of rules in the program and replacing the call with the body of a matching rule.

- When a call matches multiple rules, a choice point is created. A choice point stores the information needed to “roll back” execution to that point so that another matching rule can be selected for expansion.

- Success occurs when each call has been removed by expanding it with a fact.

- Backtracking occurs when a call with no matching rules is encountered. Execution is rolled back to the previous choice point and the next unused rule is expanded. When the last matching rule is selected, the choice point is removed.

- Failure occurs when a call has no matching rules and there are no choice points to allow backtracking.

In addition to the normal success and failure permitted under SLD resolution, co-SLD resolution allows coinductive success to occur when a call unifies with one of its ancestors. This is accomplished using a coinductive hypothesis set (CHS), to which each call is added as it is made (Simon 2006).
Consider the following program defining an infinite list of zeroes and ones:

```prolog
binary_list([]).
binary_list([0|T]) :- binary_list(T).
binary_list([1|T]) :- binary_list(T).
```

Given the query `?- binary_list(X)`, normal SLD resolution will systematically produce all valid, finite solutions, starting with the empty list. However, if we remove the base case, leaving:

```prolog
binary_list([0|T]) :- binary_list(T).
binary_list([1|T]) :- binary_list(T).
```

the program will never terminate. This is because termination under SLD resolution requires that each call eventually fail or reach a base case. If neither of these occurs, execution will loop, continually expanding calls but never succeeding. However, under co-SLD resolution, termination is possible through coinductive success:

```
?- binary_list(X). CHS = {} ; match the first rule (X = [0|X])
?- binary_list([0|X]). CHS = {} ; expand the rule
?- binary_list(X). CHS = {binary_list([0|X])} ; unify with CHS
?- true. CHS = {binary_list([0|X])} ; coinductive success
```

When either of the rules in the modified program is called, the head will be added to the CHS. When the goal in the rule body is called, it will unify with the call in the CHS, allowing it to coinductively succeed. Thus, given the same query and the modified program, co-SLD resolution will succeed in three steps, producing `X = [0|X]`, indicating an infinite list of zeroes. Through backtracking, co-SLD resolution can systematically enumerate all infinite solutions in the same way SLD resolution would produce all finite solutions to the original program. For the same query and the original program, co-SLD resolution can produce all finite and infinite solutions.
CHAPTER 3

GOAL-DIRECTED EXECUTION OF ANSWER SET PROGRAMS

3.1 Acknowledgements

The paper (Marple et al., 2012) which forms the basis for this chapter was co-authored by myself, Dr. Ajay Bansal, Dr. Richard Min and Dr. Gopal Gupta. Much of the original writing was performed by my co-authors prior to my inheriting the project. However, I have significantly rewritten their original paper in order to get it accepted for publication. The proofs and performance results are entirely my own work, the remaining text has been heavily altered and small changes have been made to the underlying goal-directed algorithm. Dr. Gupta provided continual feedback and assistance in modifying the paper.

3.2 Overview

A goal-directed method for executing answer set programs is analogous to top-down, SLD-style resolution for Prolog, while current popular methods for ASP are analogous to bottom-up methods that have been used for evaluating Prolog (and Datalog) programs (Sagonas et al., 1994). A goal-directed execution method for answering queries for an answer set program has several advantages, the main advantage being that it paves the way to lifting the restriction to finitely groundable programs and allowing the realization of ASP with full first-order predicates (Min, 2010).

In this chapter we develop a goal-directed strategy for executing answer set programs that works for any answer set program as well as for any query. We then prove that our

\footnote{The majority of this chapter was originally published as a paper in Proceedings of the 14th symposium on Principles and practice of declarative programming. Modifications have been made to remove content repeated in other chapters. The DOI of the original paper is 10.1145/2370776.2370782 (Marple et al., 2012).}
strategy is sound and complete with respect to the method of Gelfond and Lifschitz discussed in section 2.2. We restrict ourselves to only propositional (grounded) answer set programs; work is in progress to extend our goal-directed method to predicate answer set programs (Min et al., 2009; Min, 2010; Salazar et al., 2014).

Note that the design of a top-down goal-directed execution strategy for answer set programs has been regarded as quite a challenging problem (Baral, 2003). As pointed out in (Dix, 1995), the difficulty in designing a goal-directed method for ASP comes about due to the absence of a relevance property in the stable model semantics, on which answer set programming is based (Dix, 1995; Pereira and Pinto, 2005, 2009). We will introduce a modified relevance property that holds for our goal-directed method and guarantees that partial answer sets computed by our method can be extended to complete answer sets.

The remainder of the chapter is structured as follows. In section 3.3, we look at particular aspects of ASP that our strategy must account for. Next, our goal-directed algorithm is explained in section 3.4. We prove the correctness of our method in section 3.5. Finally, we conclude with discussion and related work in section 3.6.

3.3 Goal-directed ASP Issues

Program 1: Example program.

\[ p :- \neg q. \text{ % Rule a} \]
\[ q :- \neg p. \text{ % Rule b} \]

Any normal logic program can also be viewed as an answer set program. However, ASP adds complexity to a normal logic program in two ways. In addition to the standard Prolog rules, it allows:

1. Cyclical rules which when used to expand a call to a subgoal \( G \) lead to a recursive call to \( G \) through an even (but non-zero) number of negations. For example, given program
ordinary logic programming execution for the query \(- p\). (or \(- q\).) will lead to non-termination. However, ASP will produce two answer sets: \{p, not q\} and \{q, not p\}.\(^2\) Expanding the call \(p\) using rule 1.a in the style of SLD resolution will lead to a recursive call to \(p\) that is in scope of two negations (\(p \rightarrow \text{not } q \rightarrow \text{not not } p\)). Such rules are termed ordinary rules. Rule 1.b is also an ordinary rule, since if used for expanding the call to \(q\), it will lead to a recursive call to \(q\) through two negations. For simplicity of presentation, all non-cyclical rules will also be classified as ordinary rules.

\[
\begin{align*}
p & :- q, \text{not } p, r. \quad \% \text{Rule a} \\
\end{align*}
\]

Program 2: Example program.

2. Cyclical rules which when used to expand a call to subgoal \(G\) lead to a recursive call to \(G\) that is in the scope of an odd number of negations. Such recursive calls are known as odd loops over negation (OLONs). For example, given the program 2, a call to \(p\) using rule 2.a will eventually lead to a call to \(\text{not } p\). Under ordinary logic programming execution, this will lead to non-termination. Under ASP, however, the program consisting of rule 2.a has \{not p, not q, not r\} as its answer set. For brevity, we refer to rules containing OLONs as OLON rules.

\[
\begin{align*}
p & :- q, \text{not } r. \quad \% \text{Rule a} \\
r & :- \text{not } p. \quad \% \text{Rule b} \\
q & :- t, \text{not } p. \quad \% \text{Rule c} \\
\end{align*}
\]

Program 3: Example program.

Note that a rule can be both an ordinary rule and an OLON rule, since given a subgoal \(G\), its expansion can lead to a recursive call to \(G\) through both even and odd numbers of

\(^2\)Note that we will list all literals that are true in a given answer set. Conventionally, an answer set is specified by listing only the positive literals that are true; those not listed in the set are assumed to be false.
negations along different expansion paths. For example in program 3, rule 3.a is both an ordinary rule and an OLON rule.

Our top-down method requires that we properly identify and handle both ordinary and OLON rules. We will look at each type of rule in turn, followed by the steps taken to ensure that our method remains faithful to the GL method.

3.3.1 Ordinary Rules

Ordinary rules such as rules 1.a and 1.b in program 1 above exemplify the cyclical reasoning in ASP. The rules in the example force \( p \) and \( q \) to be mutually exclusive, i.e., either \( p \) is true or \( q \) is true, but not both. One can argue the reasoning presented in such rules is cyclical: If \( p \) is in the answer set, then \( q \) cannot be in the answer set, and if \( q \) is not in the answer set, then \( p \) must be in the answer set.

Given a goal, \( G \), and an answer set program comprised of only ordinary rules, \( G \) can be executed in a top-down manner using coinduction, through the following steps:

- Record each call in the CHS. The recorded calls constitute the coinductive hypothesis set, which is the potential answer set.

- If at the time of the call, the call is already found in the CHS, it succeeds coinductively and finishes.

- If the current call is not in the CHS, then expand it in the style of ordinary SLD resolution (recording the call in the CHS prior to expansion).

- Simplify \( \text{not not } p \) to \( p \), whenever possible, where \( p \) is a proposition occurring in the program.

- If success is achieved with no goals left to expand, then the coinductive hypothesis set contains the (partial) answer set.
The top-down resolution of query \( p \) with program \( \square \) will proceed as follows.

\[
\begin{align*}
\text{:- } & p & \quad \text{CHS} = \{\} \\
& & \quad \text{(expand } p \text{ by rule 1a)} \\
\text{:- } & \neg q & \quad \text{CHS} = \{p\} \\
& & \quad \text{(expand } q \text{ by rule 1b)} \\
\text{:- } & \neg \neg p & \quad \text{CHS} = \{p, \neg q\} \\
& & \quad \text{(simplify } \neg \neg p \rightarrow p) \\
\text{:- } & p & \quad \text{CHS} = \{p, \neg q\} \\
& & \quad \text{(coinductive success: } p \in \text{ CHS)} \\
\text{:- } & \square & \quad \text{success: answer set is } \{p, \neg q\} \\
\end{align*}
\]

Note that the maintenance of the coinductive hypothesis set (CHS) is critical. If a call is encountered that is already in the CHS, it should not be expanded, it should simply (coinductively) succeed. Note that the query \( q \) will produce the other answer set \( \{q, \neg p\} \) in a symmetrical manner. Note also that the query \( \neg q \) will also produce the answer set \( \{p, \neg q\} \) as shown below. Thus, answers to negated queries can also be computed, if we apply the coinductive hypothesis rule to negated goals also, i.e., a call to \( \neg p \) succeeds, if an ancestor call to \( \neg p \) is present:

\[
\begin{align*}
\text{:- } & \neg q & \quad \text{CHS} = \{\} \\
& & \quad \text{(expand } q \text{ by rule 1b)} \\
\text{:- } & \neg \neg p & \quad \text{CHS} = \{\neg q\} \\
& & \quad \text{(not } \neg p \rightarrow p) \\
\text{:- } & p & \quad \text{CHS} = \{p, \neg q\} \\
& & \quad \text{(expand } p \text{ by rule 1a)} \\
\text{:- } & \neg q & \quad \text{CHS} = \{p, \neg q\} \\
& & \quad \text{(coinductive success for } \neg q) \\
\text{:- } & \square & \quad \text{success: answer set is } \{p, \neg q\} \\
\end{align*}
\]
3.3.2 OLON Rules

Our goal-directed procedure based on coinduction must also work with OLON rules. OLON rules are problematic because their influence on answer sets is indirect. Under ASP, rules of the form:

\[ p :- q_1, q_2, \ldots, q_k, \neg p. \]

hold only for those (stable) models in which \( p \) succeeds through other rules in the program or at least one of the \( q_i \)'s is false. Note that a headless rule of the form:

\[ :- q_1, q_2, \ldots, q_n. \]

is another manifestation of an OLON rule, as it is equivalent to the rule:

\[ p :- q_1, q_2, \ldots, q_n, \neg p. \]

where \( p \) is a literal that does not occur anywhere else in the program, in the sense that the stable models for the two rules are identical.

Without loss of generality, consider the simpler rule:

\[ p :- q, \neg p. \]

For an interpretation to be a (stable) model for this rule, either \( p \) must succeed through other rules in the program or \( q \) must be false. Two interesting cases arise: (i). \( p \) is true through other rules in the program. (ii) \( q \) is true through other rules in the program.

Program 4: Example program.

\[
\begin{align*}
p & :- q, \neg p. \quad \% \text{Rule 1} \\
p. & \quad \% \text{Rule 2}
\end{align*}
\]

Program 5: Example program.

\[
\begin{align*}
p & :- q, \neg p. \quad \% \text{Rule 1} \\
q. & \quad \% \text{Rule 2}
\end{align*}
\]

For case (i), if \( p \) is true through other means in the program, then according to the Gelfond-Lifschitz method, it is in the answer set, and the OLON rule is taken out of consideration due to the occurrence of \( \neg p \) in its body. For case (ii), if \( q \) is true through other
means and the rule is still in consideration due to \( p \) not being true through other rules in the program, then there are no answer sets, as \( q \) is both true and false. Thus, the answer set of the program 4 is: \{p, not q\} while there is no answer set for program 5.

Given an OLON rule with \( p \) as its head and the query \( p \), execution based on co-SLD resolution will fail, if we require that the coinductive hypothesis set (CHS) remains consistent at all times. That is, if we encounter the goal \( g \) (resp. not \( g \)) during execution and not \( g \) \( \in \) CHS (resp. \( g \) \( \in \) CHS), then the computation fails and backtracking ensues.

As another example, consider the program containing rule 4.1 (which has \( p \) in its head), but not rule 4.2, and the query \( :- p \). When execution starts, \( p \) will be added to the CHS and then expanded by rule 4.1; if the call to \( q \) fails, then the goal \( p \) also fails. Alternatively, if \( q \) succeeds due to other rules in the program, then upon arriving at the call not \( p \), failure will ensue, since not \( p \) is inconsistent with the current CHS (which equals \{p, q\} prior to the call not \( p \)).

Thus, OLON rules do not pose any problems in top-down execution based on coinduction, however, given an OLON rule with \( p \) as its head, if \( p \) can be inferred by other means (i.e., through ordinary rules) then the query \( p \) should succeed. Likewise, if \( q \) succeeds by other means and \( p \) does not, then we should report a failure (rather, report the absence of an answer set; note that given our conventions, CHS = {} denotes no answer set). We discuss how top-down execution of OLON rules is handled in section 3.3.4.

### 3.3.3 Coinductive Success Under ASP

\[
\begin{align*}
p & :- q. \\
q & :- p.
\end{align*}
\]

**Program 6:** Example program.

While our technique’s use of co-SLD resolution has been outlined above, it requires some additional modification to be faithful to the Gelfond-Lifschitz method. Using normal
coinductive success, our method will compute the *gfp* of the residual program after the GL transform, while the GL method computes the *lfp*. Consider program 6: our method based on coinduction will succeed for queries : - p and : - q producing the answer set \{p, q\} while under the GL method, the answer set for this program is \{not p, not q\}. Our top-down method based on coinduction really computes the *gfp* of the original program. The GL method computes a fixed point of the original program (via the GL transformation and then computation of the *lfp* of the residual program) that is in between the *gfp* and the *lfp* of the original program. In the GL method, *direct cyclical reasoning is not allowed*, however, *cyclical reasoning that goes through at least one negated literal is allowed*. Thus, under the GL method, the answer set of program 6 does not contain a single positive literal, while there are two answer sets for the program 1 given earlier, each with exactly one positive literal, even though both programs 1 and 6 have only cyclical rules.

Our top-down method can be modified so that it produces answer sets consistent with the GL method: *a coinductive recursive call can succeed only if it is in the scope of at least one negation*. In other words, the path from a successful coinductive call to its ancestor call must include a call to *not*.

This restriction disallows inferring *p* from rules such as

\[ p : - p. \]

With this operational restriction in place, the CHS will never contain a positive literal that is in the *gfp* of the residual program obtained after the GLT, but not in its *lfp*. To show this, let us assume that, for some ASP program, a call to *p* will always encounter at least one recursive call to *p* with no intervening negation. In such a case, *p* will never be part of any answer set:

- Under our goal-directed method, any call to *p* will fail when a recursive call is encountered with no intervening negation.
• Under the GL method, $p$ will never be in the lfp of the residual. Even if a rule for $p$ is present in the residual and all other dependencies are satisfied, the rule will still depend on the recursive call to $p$.

3.3.4 NMR Consistency Check

To summarize, the workings of our goal-directed strategy are as follows: given a goal $G$, perform co-SLD resolution while restricting coinductive success as outlined in section 3.3.3. The CHS serves as the potential answer set. A successful answer will be computed only through ordinary rules, as all OLON rules will lead to failure due to the fact that not $h$ will be encountered with proposition $h$ present in the CHS while expanding with an OLON rule whose head is $h$. Once success is achieved, the answer set is the CHS. As discussed later, this answer set may be partial.

The answer set produced by the process above is only a potential answer set. Once a candidate answer set has been generated by co-SLD resolution as outlined above, the set has to be checked to see that it will not be rejected by an OLON rule. Suppose there are $n$ OLON rules in the program of the form:

$$q_i : - B_i,$$

where $1 \leq i \leq n$ and each $B_i$ is a conjunction of goals. Each $B_i$ must contain a direct or indirect call to the respective $q_i$ which is in the scope of odd number of negations in order for $q_i : - B_i$ to qualify as an OLON rule.

If a candidate answer set contains $q_j$ ($1 \leq j \leq n$), then each OLON rule whose head matches $q_j$ must be taken out of consideration (this is because $B_j$ leads to not($q_j$) which will be false for this candidate answer set). For all the other OLON rules whose head proposition $q_k$ ($1 \leq j \leq n$) is not in the candidate answer set, their bodies must evaluate to false w.r.t. the candidate answer set, i.e., for each such rule, $B_k$ must evaluate to false w.r.t. the candidate answer set.
The above restrictions can be restated as follows: a candidate answer set must satisfy the formula \( q_i \lor \neg B_i \) (1 \( \leq \) i \( \leq \) n) for each OLON rule \( q_i : - B_i \). (1 \( \leq \) i \( \leq \) n) in order to be reported as the final answer set. Thus, for each OLON rule, the check

\[
\text{chk}_{q_i} : - q_i.
\]

\[
\text{chk}_{q_i} : - \neg B_i.
\]

is constructed by our method. Furthermore, \( \neg B_i \) will be expanded to produce a \( \text{chk}_{q_i} \) clause for each literal in \( B_i \). For example, if \( B_i \) represented the conjunction of literals \( s, \neg r, t \) in the above example, the check created would be:

\[
\text{chk}_{q_i} : - q_i.
\]

\[
\text{chk}_{q_i} : - \neg s.
\]

\[
\text{chk}_{q_i} : - r.
\]

A candidate answer set must satisfy each of these checks in order to be reported as a solution. This is enforced by rolling the checks into a single call, termed the non-monotonic reasoning check (NMR check):

\[
\text{nmr\_chk} : - \text{chk}_{q_1}, \text{chk}_{q_2}, \ldots \text{chk}_{q_n}.
\]

Now each query \( Q \) is transformed to \( Q, \text{nmr\_chk} \) before it is posed to our goal-directed system. One can think of \( Q \) as the generator of candidate answer sets and \( \text{nmr\_chk} \) as the filter. If \( \text{nmr\_chk} \) fails, then backtracking will take place and \( Q \) will produce another candidate answer set, and so on. Backtracking can also take place within \( Q \) itself when a call to \( p \) (resp. \( \neg p \)) is encountered and \( \neg p \) (resp. \( p \)) is present in the CHS. Note that the CHS must be a part of the execution state, and be restored upon backtracking.

3.4 Goal-directed Execution of Answer Set Programs

We next describe our general goal-directed procedure for computing answer sets.
3.4.1 Dual Rules

For simplicity, we add one more step to the process. Similarly to Alferes et al. [Alferes et al., 2004], for each rule in the program, we introduce its dual. That is, given a proposition \( H \)’s definition (\( B_i \)’s are conjunction of literals):

\[
H :- B_1.
H :- B_2.
\ldots
H :- B_n.
\]

we add the dual rule

\[
\text{not } H :- \text{not } B_1, \text{not } B_2, \ldots, \text{not } B_n.
\]

If a proposition \( q \) appears in the body of a rule but not in any of the rule heads, then the fact

\[
\text{not } q.
\]

is added. Note that adding the dual rules is not necessary; it only makes the exposition of our goal-directed method easier to present and understand.

3.4.2 Goal-directed Method for Computing Answer Sets

Given a propositional query \( :- Q \) and a propositional answer set program \( P \), the goal-directed procedure works as described below. Note that the execution state is a pair \( (G, S) \), where \( G \) is the current goal list, and \( S \) the current CHS.

1. Identify the set of ordinary rules and OLON rules in the program.

2. Assert a \( \text{chk}_q \) rule for every OLON rule with \( q \) as its head and build the \texttt{nmr_check} as described in section 3.3.4

3. For each ordinary rule and \( \text{chk}_q \) rule, construct its dual version.
4. Append the \texttt{nmr\_check} to the query.

5. Set the initial execution state to: \((:- \mathcal{G}_1, \ldots, \mathcal{G}_n, \{\})\).

6. Non-deterministically reduce the execution state using the following rules:

   (a) \textit{Call Expansion}:
   \[
   (:- \mathcal{G}_1, \ldots, \mathcal{G}_i, \ldots, \mathcal{G}_n, S)
   \rightarrow (:- \mathcal{G}_1, \ldots, \mathcal{B}_1, \ldots, \mathcal{B}_m, \ldots, \mathcal{G}_n, \mathcal{S} \cup \{\mathcal{G}_i\})
   \]
   where \(\mathcal{G}_i\) matches the rule \(\mathcal{G}_i \leftarrow \mathcal{B}_1, \ldots, \mathcal{B}_m\) in \(\mathcal{P}\), \(\mathcal{G}_i \notin \mathcal{S}\) and \(\text{not } \mathcal{G}_i \notin \mathcal{S}\).

   (b) \textit{Coinductive Success}:
   \[
   (:- \mathcal{G}_1, \ldots, \mathcal{G}_i-1, \mathcal{G}_i, \mathcal{G}_{i+1}, \ldots, \mathcal{G}_n, S)
   \rightarrow (:- \mathcal{G}_1, \ldots, \mathcal{G}_i-1, \mathcal{G}_{i+1}, \ldots, \mathcal{G}_n, S)
   \]
   if \(\mathcal{G}_i \in \mathcal{S}\) and either:
   
   i. \(\mathcal{G}_i\) is not a recursive call or
   
   ii. \(\mathcal{G}_i\) is a recursive call in the scope of a non-zero number of intervening negations.

   (c) \textit{Inductive Success}:
   \[
   (:- \mathcal{G}_1, \ldots, \mathcal{G}_i-1, \mathcal{G}_i, \mathcal{G}_{i+1}, \ldots, \mathcal{G}_n, S)
   \rightarrow (:- \mathcal{G}_1, \ldots, \mathcal{G}_i-1, \mathcal{G}_{i+1}, \ldots, \mathcal{G}_n, \mathcal{S} \cup \{\mathcal{G}_i\})
   \]
   if \(\mathcal{G}_i\) matches a fact.

   (d) \textit{Coinductive Failure}:
   \[
   (:- \mathcal{G}_1, \ldots, \mathcal{G}_i, \ldots, \mathcal{G}_n, S) \rightarrow \text{(fail, S)}
   \]
   if either:
   
   i. \(\text{not } \mathcal{G}_i \in \mathcal{S}\) or
   
   ii. \(\mathcal{G}_i \in \mathcal{S}\) and \(\mathcal{G}_i\) is a recursive call without any intervening negations.
(e) **Inductive Failure:**

\[
(\text{:- } G_1, \ldots, G_i, \ldots, G_n, S) \rightarrow (\text{fail}, S) \\
\text{if } G_i \text{ has no matching rule in } P.
\]

(f) **Print Answer:**

\[
(\text{:- true}, S) \rightarrow \text{success: } S \text{ is the answer set} \\
\text{where ‘:- true’ } \equiv \text{empty goal list}
\]

Note that when all the goals in the query are exhausted, execution of \texttt{nmr_chk} begins. Upon failure, backtracking ensues, the state is restored and another rule tried. Note that negated calls are expanded using dual rules as in \cite{Alferes2004}, so it is not necessary to check whether the number of intervening negations between a recursive call and its ancestor is even or odd. (See the call expansion rule above). Next we discuss a few important issues:

**Identifying OLON and Ordinary Rules**  Given a propositional answer set program \(P\), OLON rules and ordinary rules can be identified by constructing and traversing the call graph. The complexity of this traversal is \(O(|P| \ast n)\), where \(n\) is the number of propositional symbols occurring in the head of clauses in \(P\) and \(|P|\) is a measure of the program size. Note also that during the execution of a query \(Q\), we need not make a distinction between ordinary and OLON rules; knowledge of OLON rules is needed only for creating the \texttt{nmr_chk}.

| p :- not q.  
| q :- not p.  
| r :- not s.  
| s :- not r.  |

**Program 7:** Example program.

**Partial Answer Set**  Our top-down procedure might not generate the entire answer set. It may generate only the part of the answer set that is needed to evaluate the query. Under
goal-directed execution, the query :- q. for program\textsuperscript{7} will produce only \{q, \text{not } p\} as the answer since the rules defining r and s are completely independent of rules for p and q. One could argue that this is an advantage of a goal-directed execution strategy rather than a disadvantage, as only the relevant part of the program will be explored. In contrast, if the query is :- q, s, then the right answer \{q, \text{not } p, s, \text{not } r\} will be produced by the goal-directed execution method. Thus, the part of the answer set that gets computed depends on the query. Correct maintenance of the CHS throughout the execution is important as it ensures that only consistent and correct answer sets are produced.

### 3.5 Soundness and Correctness of the Goal-directed Method

We will now show the correctness of our goal-directed execution method by showing it to be sound and complete with respect to the GL method. First, we will examine the modified relevance property which holds for our method.

#### 3.5.1 Relevance

As we stated in the introduction, one of the primary problems with developing a goal-directed ASP implementation is the lack of a relevance property in stable model semantics. Dix introduces relevance by stating that, “given any semantics SEM and a program P, it is perfectly reasonable that the truth-value of a literal L, with respect to SEM(P), only depends on the subprogram formed from the relevant rules of P with respect to L” \cite{Dix1995}. He formalizes this using the dependency-graph of P, first establishing that

- “dependencies\textsubscript{of}(X) := \{A : X depends on A\}, and
- \text{rel.rul}(P,X) is the set of relevant rules of P with respect to X, i.e. the set of rules that contain an \(A \in \text{dependencies}\textsubscript{of}(X)\) in their heads” \cite{Dix1995}
and noting that the dependencies and relevant rules of \( \neg X \) are the same as those of \( X \) \cite{Dix1995}. He then defines relevance as, for all literals \( L \):

\[
SEM(P)(L) = SEM(\text{rel}_rul(P, L))(L)
\]

(3.1)

The relevance property is desirable because it would ensure that a partial answer set computed using only relevant rules for each literal could be extended into a complete answer set. However, stable model semantics do not satisfy the definition as given. This is because OLON rules can alter the meaning of a program and the truth values of individual literals without occurring in the set of relevant rules \cite{Dix1995, Pereira2005}. For instance, an irrelevant rule of the form \( p :\neg p. \) when added to an answer set program \( P \), where \( P \) has one or more stable models and \( p \) does not occur in \( P \), results in a program that has no stable models.

Approaches such as \cite{Pereira2005} have addressed the lack of a relevance property by modifying stable model semantics to restore relevance. However, our implementation can be viewed as restoring relevance by expanding the definition of relevant rules to include all OLON rules in a program. Because the NMR check processes every OLON rule, it has the effect of making the truth value of every literal in a program dependent on such rules. That is,

\[
nmr_{\text{rel}_rul}(P, L) = \text{rel}_rul(P, L) \cup O,
\]

(3.2)

\[
O = \{ \text{R: R is an OLON rule in P } \}
\]

Using \( nmr_{\text{rel}_rul}(P, L) \) in place of \( \text{rel}_rul(P, L) \), a modified version of equation (3.1) above holds for our semantics:

\[
SEM(P)(L) = SEM(nmr_{\text{rel}_rul}(P, L))(L)
\]

(3.3)

As a result, any partial model returned by our semantics is guaranteed to be a subset of one or more complete models.
3.5.2 Soundness

**Theorem 1.** For the non-empty set $X$ returned by successful top-down execution of some program $P$, the set of positive literals in $X$ will be an answer set of $R$, the set of rules of $P$ used during top-down execution.

**Proof.** Let us assume that top-down execution of a program $P$ has succeeded for some query $Q$ consisting of a set of literals in $P$, returning a non-empty set of literals $X$. We can observe that $R \subseteq \bigcup_{L \in Q} nmr_{rel \_rul}(P, L)$: for each positive literal in $Q$, one rule with the literal in its head will need to succeed, for each negative literal in $Q$ all rules with the the positive form of the literal in their head will need to fail, and the resulting set must satisfy the NMR check. We will show that $X$ is a valid answer set of $R$ using the GL method. First, because $X$ may contain negative literals and the residual program produced by the GL method is a positive one, let us remove any rules in $R$ containing the positive version of such literals as a goal, and then remove the negated literals from $X$ to obtain $X'$. Because our algorithm allows negative literals to succeed if and only if all rules for the positive form fail or no such rules exist, only rules which failed during execution will be removed by this step. Next, let us apply the GL transformation using $X'$ as the candidate answer set to obtain the residual program $R'$. This will remove rules containing the negation of any literal in $X'$ and remove any negated goals from the remaining rules.

We know that $X'$ will be an answer set of $R$ if and only if $X' = LFP(R')$. Now let us examine the properties of $R'$. As positive literals, we know that each literal in $X'$ must occur as the head of a rule in $R$ which succeeded during execution. Because such rules would have failed if the negation of any goal was present in the CHS, we know that such rules would not have been eliminated from the residual program by the GL transformation, and are thus still present in $R'$ save for the removal of any negated goals. Because any rules containing the negation of a literal in $X$ had to fail during execution, at least one goal in each of these rules
must have failed, resulting in the negation of the goal being added to the CHS. Furthermore, because the NMR check applies the negation of each OLON rule, again the negation of some goal in each such rule must have been added to the CHS. Thus any rule which failed during execution and yet was included in R will have been removed from R'. Finally, because our algorithm allows coinductive success to occur only in the scope of at least one negation, the removal of negated goals from the residual program will ensure that R' contains no loops. Because the remaining rules in R' must have succeeded during execution, their goals must have been added to the CHS, and therefore those goals consisting of positive literals form X'. Thus R' is a positive program with no loops, and each literal in X' must appear as the head of some rule in R' which is either a fact or whose goals consist only of other elements in X'. Therefore the least fixed point of R' must be equal to X', and X' must be an answer set of R.

**Theorem 2.** Our top-down execution algorithm is sound with respect to the GL method. That is, for the non-empty set of literals X returned by successful execution of some program P, the set of positive literals in X is a subset of one or more answer sets of P.

**Proof.** As shown above, the positive literals in the set returned by successful execution of P will be an answer set of \( R \subseteq \bigcup_{L \in Q} nmr_{rel-rul}(P, L) \). Because R will always contain all OLON rules in P, no unused rules in P are capable of affecting the truth values of the literals in X. Thus the modified definition of relevance holds for all literals in X under our semantics and the partial answer set returned by our algorithm is guaranteed to be extensible to a complete one. Thus our algorithm for top-down execution is sound with respect to the GL method.

**3.5.3 Completeness**

**Theorem 3.** Our top-down execution algorithm is complete with respect to the GL method. That is, for a program P, any answer set valid under the GL method will succeed if used
as a query for top-down execution. In addition, the set returned by successful execution will contain no additional positive literals.

Proof. Let X be a valid answer set of P obtained via the GL method. Then there exists a resultant program P′ obtained by removing those rules in P containing the negation of any literal in X and removing any additional negated literals from the goals of the remaining rules. Furthermore, because X is a valid answer set of P, X = LFP(P′). This tells us that for every literal \( L \in X \) there is a rule in P′ with \( L \) as its head, which is either a fact or whose goals consist only of other literals in X.

Let us assume that X is posed as a query for top-down execution of P. As we know that each \( L \in X \) has a rule in P′ with \( L \) as its head and whose positive goals are other literals in X, we know that such a rule also exists in P, with the possible addition of negated literals as goals. However, we know that these negated literals must succeed, that is, all rules with the positive form of such literals in their heads must fail, either by calling the negation of some literal in the answer set or by calling their heads recursively without an intervening negation. Were this not the case, these rules would remain in P′, their heads would be included in LFP(P′) and X would not be a valid answer set of P. Therefore, a combination of rules may be found such that each literal in X appears as the head of at least one rule which will succeed under top-down execution, and whose positive goals are all other literals in X. Furthermore, because each literal in the query must be satisfied and added to the CHS, and any rule with a goal whose negation is present in the CHS will fail, such a combination of rules will eventually be executed by our algorithm. Because such rules would also be present in P′, we know that they cannot add additional positive literals to the CHS, as these would be part of LFP(P′), again rendering X invalid.

This leaves the NMR check, which ensures the set returned by our algorithm satisfies all OLON rules in P. However, we know this is the case, as the subset of positive literals in the CHS is equal to X. Because X is a valid answer set of P, there cannot be any rule in P which
renders $X$ invalid, and thus the NMR check must be satisfiable by a set of literals containing $X$. We also know that the NMR check will not add additional positive literals to the CHS, as any rules able to succeed would be present in $P'$ and thus present in $LFP(P')$.

Therefore any valid answer set $X$ of a program $P$ must succeed if posed as a query for top-down execution of $P$. Thus our top-down algorithm is complete with respect to the GL method.

3.6 Discussion and Related Work

A top-down, goal-directed execution strategy for ASP has been the aim of many researchers in the past. Descriptions of some of these efforts can be found in (Fernández and Lobo, 1993; Bonatti et al., 2008; Bonatti, 2001; Gebser and Schaub, 2006; Kakas and Toni, 1999; Alferes et al., 2004; Shen et al., 2004; Pereira and Pinto, 2005, 2009). The strategy presented in this chapter is based on one presented by several of this chapter’s authors in previous work (Gupta et al., 2007; Min et al., 2009). However, the strategy presented in those works was limited to call-consistent or order-consistent programs. While the possibility of expansion to arbitrary ASP programs was mentioned, it was not expanded upon, and the proofs of soundness and completeness covered only the restricted cases (Min et al., 2009).

A query-driven procedure for computing answer sets via an abductive proof procedure has been explored (Eshghi and Kowalski, 1989; Kakas et al., 1992): a consistency check via integrity constraints is done before a negated literal is added to the answer set. However, “this procedure is not always sound with respect to the above abductive semantics of NAF” (Kakas et al., 1992). (Alferes et al., 2004) have worked in a similar direction, though this is done in the context of abduction and again goal-directedness of ASP is not the main focus. Gebser and Schaub have developed a tableau based method which can be regarded as a step in this direction, however, the motivation for their work is completely different (Gebser and Schaub, 2006).
Bonatti, Pontelli and Tran (Bonatti et al., 2008) have proposed credulous resolution, an extension of earlier work of Bonatti (Bonatti, 2001), that extends SLD resolution for ASP. However, they place restrictions on the type of programs allowed and the type of queries allowed. Their method can be regarded as allowing coinductive success to be inferred only for negated goals. Thus, given query \( \neg p \) and program 1, the execution will look as follows: \( p \rightarrow \neg q \rightarrow \neg \neg p \rightarrow \neg q \rightarrow \text{success} \). Compared to our method, their method performs extra work. For example, if rule 1.1 is changed to \( p :\neg \text{big.goal}, \neg q \), then \text{big.goal} will be executed twice. The main problem in their method is that since it does not take coinduction for positive goals into account, knowing when to succeed inductively and when to succeed coinductively is undecidable. For this reason, their method works correctly only for a limited class of answer set programs (for example, answers to negated queries such \( \neg \neg p \) cannot be computed in a top-down manner). In contrast, our goal-directed method works correctly for all types of answer set programs and all types of queries.

Pereira’s group has done significant work on defining semantics for normal logic programs and implementing them, including implementation in a top-down fashion (Alferes et al., 2004; Pereira and Pinto, 2005, 2009). However, their approach is to modify stable model semantics so that the property of relevance is restored (Pereira and Pinto, 2005). For this modified semantics, goal-directed procedures have been designed (Pereira and Pinto, 2009). In contrast, our goal is to stay faithful to stable model semantics and answer set programming.
CHAPTER 4

GALLIWASP: A GOAL-DIRECTED ANSWER SET SOLVER

4.1 Acknowledgements

The paper \cite{Marple and Gupta, 2013} which forms the basis for this chapter was co-authored by myself and Dr. Gopal Gupta. The underlying research is my own work, along with the majority of the text. Dr. Gupta provided invaluable feedback and guidance, along with minor modifications to the text.

4.2 Overview

The \textit{Galliwasp} system \cite{Marple, 2014}, described in this chapter, is the first efficient implementation of the goal-directed method discussed in chapter 3. As with the algorithm which it implements, \textit{Galliwasp} is consistent with the conventional semantics of ASP: no restrictions are placed on the queries and programs that can be executed. That is, any ASP program can be executed by \textit{Galliwasp}.

The core algorithm of \textit{Galliwasp} may be summarized as follows. Call graphs are used to classify rules as according to the two attributes introduced in section 3.3: (i) if a rule can be called recursively with an odd number of negations between the initial call and its recursive invocation, it is said to contain an \textit{odd loop over negation} (OLON) and referred to as an OLON rule for brevity, and (ii) if a rule has at least one path in the call graph that will not result in such a call, it is called an ordinary rule. Our goal-directed method uses ordinary

\footnote{The majority of this chapter was originally published as a paper in \textit{Volume 7844 of Lecture Notes in Computer Science}. Modifications have been made to remove content repeated in other chapters. The original publication is available at www.springerlink.com. The DOI of the original paper is \url{10.1007/978-3-642-38197-3_9} (Marple and Gupta, 2013).}
rules to generate candidate answer sets (via coinduction extended with negation) and OLON rules to reject invalid candidates. The procedure can be thought of as following the *generate and test* paradigm with ordinary rules being used for generation of candidate answer sets and OLON rules for testing that a candidate answer set is, indeed, a valid answer set.

There are many advantages of a goal-directed execution strategy: (i) ASP can be extended to general predicates (Min, 2010), i.e., to answer set programs that do not admit a finite grounding; (ii) extensions of ASP to constraints (in the style of CLP(R)) (Jaffar and Lassez, 1987), probabilities (in the style of ProbLog (De Raedt et al., 2007)), etc., can be elegantly realized; (iii) Or-parallel implementations of ASP that leverage techniques from parallel Prolog implementations (Gupta et al., 2001) can be developed; (iv) abductive reasoning (Kakas et al., 1992) can be incorporated with greater ease. Work is in progress to extend Galliwasp in these directions.

The remainder of the chapter is structured as follows. In section 4.3 we discuss the design and implementation of Galliwasp. In section 4.4 we compare its performance to that of several other ASP solvers. Finally, we examine related work in section 4.5.

### 4.3 System Architecture of Galliwasp

The Galliwasp system consists of two components: a compiler and an interpreter. The compiler reads in a grounded instance of an ASP program and produces a compiled program, which is then executed by the interpreter to compute answer sets. An overview of the system architecture is shown in figure 4.1.

#### 4.3.1 Compiler

The primary goal of Galliwasp’s design is to maximize the runtime performance of the interpreter. Because Galliwasp computes answer sets for a program based on user-supplied queries, answer sets cannot be computed until a program is actually run by the interpreter.
However, many of the steps in our goal-directed algorithm, as well as parsing, program transformations and static analysis, are independent of the query. The purpose of Galliwasps’s compiler is to perform as much of the query-independent work as possible, leaving only the actual computation of answer sets to the interpreter. Because a single program can be run multiple times with different queries, compilation also eliminates the need to repeat the query-independent steps every time a program is executed. The most important aspects of the compiler are the input language accepted, the parsing and pre-processing performed, the construction of the NMR check, the program transformations applied, and the formatting of the compiled output.

**Input Language**

The Galliwasp compiler’s primary input language is grounded A-Prolog [Gelfond, 2002], extended to allow compatibility with text-mode output of the lparse grounder. This allows grounded program instances to be obtained by invoking lparse with the -t switch, which formats the output as text. The smodels input format is also supported, allowing grounders other than lparse to be used.
When *lpars* encounters a headless rule, it produces a grounded rule with _false as the
head and a compute statement containing the literal not _false. Because the literal _false
is not present in the body of any rule, special handling is required to properly detect such
rules as OLON rules.

The compute statements used by *lpars* are of the form:

```
compute N  Q .
```

where N specifies the number of answer sets to compute and Q is a set of literals that must
be present in a valid answer set. Our system handles these statements by treating them as
optional, hard-coded queries. If a compute statement is present, the interpreter may be run
without user interaction, computing up to N answer sets using Q as the query. When the
interpreter is run interactively, it ignores compute statements and executes queries entered
by the user.

**Parsing and Pre-Processing**

The compiler’s front end and pre-processing stages prepare the input program for easy access
and manipulation during the rest of the compilation process. The front end encompasses
the initial lexical analysis and parsing of the input program, while the pre-processing stage
handles additional formatting and simplification, and substitutes integers for literals.

After lexical analysis of the input program is performed, it is parsed into a list of rules
and statements by a definite clause grammar (DCG). During parsing, a counter is used to
number the rules as they are read, so that the relative order of rules for a given literal can
be maintained.

The list of statements produced by the DCG is next converted into a list of rules and
a single compute statement. Each rule is checked to remove duplicate goals. Any rule
containing its head as a goal with no intervening negation will also be removed at this stage.
The cases covered in this step should not normally occur, but as they are allowed by the
language, they are addressed before moving on.
The next stage of pre-processing is integer substitution. Every propositional symbol \( p \) is mapped to a unique integer \( N_p > 0 \). A positive literal \( p \) is represented by \( N_p \), while a negative literal \( \text{not} \ p \) is represented by \( -N_p \). Since the interpreter must use the original names of propositions when it prints the answer, a table that maps each \( N_p \) to \( p \) is included in the compiled output.

Finally, the list of rules is sorted by head, maintaining the relative order of rules with the same head. This eliminates the need for repeated searching in subsequent stages of compilation. After sorting, compilation moves on to the next stage, the detection of OLON rules and construction of the NMR check.

**Construction of the NMR Check**

Construction of the NMR check begins with the detection of the OLON rules in the ASP program. These rules are then used to construct the individual checks that form the NMR check, as described in section 3.3.4.

Clauses for the sub-checks are treated as any other rule in the program, and subject to program transformation in the next stage of compilation. However, the NMR check itself is not modified by the program transformation stage. Instead, if the modified sub-checks allow for immediate failure, this will be detected at runtime by the interpreter.

**Program Transformation**

The program transformation stage consists of computing dual rules, explained below, and removing rules when it can be trivially determined at compile time that they will never succeed. This stage of compilation improves performance without affecting the correctness of our algorithm.

Dual rules, i.e., rules for the negation of each literal, are computed as follows. For proposition \( p \) defined by the rules:
\[ p :\neg B_1. \]
\[
\ldots
\]
\[ p :\neg B_n. \]

where each \( B_i \) is a conjunction of positive and negative literals, its dual
\[ \neg p :\neg \neg B_1, \ldots, \neg B_n. \]
is added to the program. For any proposition \( q \) for which there are no rules whose head contains \( q \), a fact is added for \( \neg q \).

The addition of dual rules simplifies the design of the interpreter by removing the need to track the scope of negations: all rules can be executed in a uniform fashion, regardless of the number of negations encountered. When a negated calls are encountered, they are expanded using dual rules. While adding these rules may significantly increase the size of the program, this is not a problem: the interpreter performs indexing that allows it to access rules in constant time.

After the dual rules have been computed, the list is checked once more to remove simple cases of rules that can be trivially determined to always fail. This step simply checks to see if a fact exists for the negation of some goal in the rule body and removes the rule if this is the case. If the last rule for a literal is removed, a fact for the negation is added and subsequent rules calling the literal will also be removed.

**Output Formatting**

As with the rest of the compiler, the output produced by the compiler is designed to reduce the amount of work performed by the interpreter. This is done by including space requirements and sorting the rules by their heads before writing them to the output.

As a result of the output formatting, the interpreter is able to read in the input and create the necessary indexes in linear time with respect to the size of the compiled program. After indexing, all that remains is to execute the program, with all other work having been performed during compilation.
4.3.2 Interpreter

While the compiler was designed to perform a variety of time consuming tasks, the interpreter has been designed to maximize run-time performance, finding an answer set or determining failure as quickly as possible. Two modes of operation, interactive and automatic, are supported. When a program is run interactively, the user can input queries and accept or reject answer sets as can be done with answers to a query in Prolog. In automatic mode it executes a compute statement that is included in the program. In either mode, the key operations can be broken up into three categories: program representation and indexing, co-SLD resolution, and dynamic incremental enforcement of the NMR check.

Program Representation and Indexing

One of the keys to Galliwasp’s performance is the indexing performed prior to execution of the program. As a result, look-up operations during execution can be performed in constant time, much as in any implementation of Prolog. As mentioned in section 4.3.1, the format of the compiler’s output allows the indexes to be created in time that is linear with respect to the size of the program.

To allow for the optimizations discussed in future chapters, the query and NMR check are stored and indexed separately from the rest of the program. Whereas the rules of the program, including the sub-checks of the NMR check, are stored in ordinary arrays, the query and NMR check are stored in a linked list. This allows their goals to be reordered in constant time.

Co-SLD Resolution

Once the program has been indexed, answer sets are found by executing the query using coinduction, as described in section 3.4.2. Each call encountered is checked against the CHS. If the call is not present, it is added to the CHS and expanded according to ordinary SLD
resolution. If the call is already in the CHS, immediate success or failure occurs, as explained below.

As mentioned in section 3.4.2 our algorithm requires that a call that unifies with an ancestor cannot succeed coinductively unless it is separated from that ancestor by an even, non-zero number of intervening negations. This is facilitated by our use of dual rules, discussed in section 4.3.1.

When the current call is of the form \texttt{not } p and the CHS contains \( p \) (or vice versa), the current call must fail, lest the CHS become inconsistent. If the current call is identical to one of the elements of the CHS, then the number of intervening negations must have been even. However, it is not clear that it was also non-zero. This additional information is provided by a counter that tracks of the number of negations encountered between the root of the proof tree and the tip of the current branch. When a literal is added to the CHS, it is stored with the current value of the counter. A recursive call to the literal can coinductively succeed only if the counter has increased since the literal was stored.

### 4.4 Performance Results

In this section, we compare the performance of Galliwasp to that of clasp (Gebser et al., 2007a), cmodels (Giunchiglia et al., 2004) and smodels (Niemelä and Simons, 1997). Multiple instances of several problems are compared. The times for Galliwasp in Table 4.1 are for the interpreter reading compiled versions of each problem instance. The times for the remaining solvers are for the solver reading problem instances from files in the smodels input language. A timeout of 600 seconds was used, with the processes being automatically killed after that time and the result being recorded as N/A in the table.

While Galliwasp is outperformed by clasp in most cases, it does achieve better times in all instances of the hanoi problems, as well as pigeon-30x30, pigeon-40x40 and schur-2x13.
<table>
<thead>
<tr>
<th>Problem</th>
<th>Galliwap Time</th>
<th>Clasp Time</th>
<th>Relative</th>
<th>Cmodels Time</th>
<th>Relative</th>
<th>Smodels Time</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>hanoi-4x15</td>
<td>0.005</td>
<td>0.014</td>
<td>280.00</td>
<td>0.022</td>
<td>440.00</td>
<td>0.038</td>
<td>760.00</td>
</tr>
<tr>
<td>hanoi-5x31</td>
<td>0.028</td>
<td>0.243</td>
<td>867.86</td>
<td>0.302</td>
<td>1078.57</td>
<td>0.452</td>
<td>1614.29</td>
</tr>
<tr>
<td>hanoi-4x14</td>
<td>0.005</td>
<td>0.011</td>
<td>220.00</td>
<td>0.015</td>
<td>300.00</td>
<td>0.022</td>
<td>440.00</td>
</tr>
<tr>
<td>hanoi-5x30</td>
<td>0.026</td>
<td>0.197</td>
<td>757.69</td>
<td>0.241</td>
<td>926.92</td>
<td>0.307</td>
<td>1180.77</td>
</tr>
<tr>
<td>queens-12</td>
<td>0.008</td>
<td>0.007</td>
<td>87.50</td>
<td>0.027</td>
<td>337.50</td>
<td>0.049</td>
<td>612.50</td>
</tr>
<tr>
<td>queens-13</td>
<td>0.009</td>
<td>0.009</td>
<td>100.00</td>
<td>0.035</td>
<td>388.89</td>
<td>0.059</td>
<td>655.56</td>
</tr>
<tr>
<td>queens-14</td>
<td>0.023</td>
<td>0.012</td>
<td>52.17</td>
<td>0.050</td>
<td>217.39</td>
<td>0.154</td>
<td>669.57</td>
</tr>
<tr>
<td>queens-15</td>
<td>0.024</td>
<td>0.014</td>
<td>58.33</td>
<td>0.059</td>
<td>245.83</td>
<td>0.261</td>
<td>1087.50</td>
</tr>
<tr>
<td>queens-16</td>
<td>0.087</td>
<td>0.018</td>
<td>20.69</td>
<td>0.066</td>
<td>75.86</td>
<td>0.643</td>
<td>739.08</td>
</tr>
<tr>
<td>queens-17</td>
<td>0.069</td>
<td>0.020</td>
<td>28.99</td>
<td>0.085</td>
<td>123.19</td>
<td>2.206</td>
<td>3197.10</td>
</tr>
<tr>
<td>queens-18</td>
<td>0.449</td>
<td>0.024</td>
<td>5.35</td>
<td>0.108</td>
<td>24.05</td>
<td>4.479</td>
<td>997.55</td>
</tr>
<tr>
<td>queens-19</td>
<td>0.054</td>
<td>0.029</td>
<td>53.70</td>
<td>0.135</td>
<td>250.00</td>
<td>1.419</td>
<td>2627.78</td>
</tr>
<tr>
<td>queens-20</td>
<td>2.548</td>
<td>0.035</td>
<td>1.37</td>
<td>0.150</td>
<td>5.89</td>
<td>23.917</td>
<td>938.66</td>
</tr>
<tr>
<td>queens-21</td>
<td>0.144</td>
<td>0.044</td>
<td>30.56</td>
<td>0.194</td>
<td>134.72</td>
<td>47.098</td>
<td>32706.94</td>
</tr>
<tr>
<td>queens-22</td>
<td>24.466</td>
<td>0.050</td>
<td>0.20</td>
<td>0.234</td>
<td>0.96</td>
<td>381.608</td>
<td>1559.75</td>
</tr>
<tr>
<td>queens-23</td>
<td>0.401</td>
<td>0.054</td>
<td>13.47</td>
<td>0.290</td>
<td>72.32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>queens-24</td>
<td>5.445</td>
<td>0.065</td>
<td>1.19</td>
<td>0.300</td>
<td>5.51</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>mapcolor-4x20</td>
<td>0.002</td>
<td>0.002</td>
<td>100.00</td>
<td>0.005</td>
<td>250.00</td>
<td>0.005</td>
<td>250.00</td>
</tr>
<tr>
<td>mapcolor-4x25</td>
<td>0.003</td>
<td>0.003</td>
<td>100.00</td>
<td>0.006</td>
<td>200.00</td>
<td>0.007</td>
<td>233.33</td>
</tr>
<tr>
<td>mapcolor-4x29</td>
<td>0.003</td>
<td>0.003</td>
<td>100.00</td>
<td>0.007</td>
<td>233.33</td>
<td>0.008</td>
<td>266.67</td>
</tr>
<tr>
<td>mapcolor-4x30</td>
<td>0.004</td>
<td>0.003</td>
<td>75.00</td>
<td>0.007</td>
<td>175.00</td>
<td>0.008</td>
<td>200.00</td>
</tr>
<tr>
<td>mapcolor-3x20</td>
<td>0.002</td>
<td>0.002</td>
<td>100.00</td>
<td>0.004</td>
<td>200.00</td>
<td>0.003</td>
<td>150.00</td>
</tr>
<tr>
<td>mapcolor-3x25</td>
<td>0.012</td>
<td>0.002</td>
<td>16.67</td>
<td>0.004</td>
<td>33.33</td>
<td>0.004</td>
<td>33.33</td>
</tr>
<tr>
<td>mapcolor-3x29</td>
<td>0.094</td>
<td>0.002</td>
<td>2.13</td>
<td>0.005</td>
<td>5.32</td>
<td>0.004</td>
<td>4.26</td>
</tr>
<tr>
<td>mapcolor-3x30</td>
<td>0.081</td>
<td>0.003</td>
<td>3.70</td>
<td>0.005</td>
<td>6.17</td>
<td>0.004</td>
<td>4.94</td>
</tr>
<tr>
<td>pigeon-10x10</td>
<td>0.003</td>
<td>0.003</td>
<td>100.00</td>
<td>0.009</td>
<td>300.00</td>
<td>0.010</td>
<td>333.33</td>
</tr>
<tr>
<td>pigeon-20x20</td>
<td>0.020</td>
<td>0.020</td>
<td>100.00</td>
<td>0.082</td>
<td>410.00</td>
<td>0.251</td>
<td>1255.00</td>
</tr>
<tr>
<td>pigeon-30x30</td>
<td>0.075</td>
<td>0.082</td>
<td>109.33</td>
<td>0.374</td>
<td>498.67</td>
<td>2.945</td>
<td>3926.67</td>
</tr>
<tr>
<td>pigeon-40x40</td>
<td>0.172</td>
<td>0.222</td>
<td>129.07</td>
<td>1.002</td>
<td>582.56</td>
<td>13.589</td>
<td>7900.58</td>
</tr>
<tr>
<td>pigeon-8x7</td>
<td>0.035</td>
<td>0.030</td>
<td>85.71</td>
<td>0.048</td>
<td>137.14</td>
<td>0.249</td>
<td>711.43</td>
</tr>
<tr>
<td>pigeon-9x8</td>
<td>0.309</td>
<td>0.231</td>
<td>74.76</td>
<td>0.322</td>
<td>104.21</td>
<td>2.284</td>
<td>739.16</td>
</tr>
<tr>
<td>pigeon-10x9</td>
<td>3.154</td>
<td>2.020</td>
<td>64.05</td>
<td>1.517</td>
<td>48.10</td>
<td>22.311</td>
<td>707.39</td>
</tr>
<tr>
<td>pigeon-11x10</td>
<td>34.439</td>
<td>16.554</td>
<td>48.07</td>
<td>45.758</td>
<td>132.87</td>
<td>249.785</td>
<td>725.30</td>
</tr>
<tr>
<td>schur-3x13</td>
<td>0.108</td>
<td>0.002</td>
<td>1.85</td>
<td>0.004</td>
<td>3.70</td>
<td>0.003</td>
<td>2.78</td>
</tr>
<tr>
<td>schur-2x13</td>
<td>0.001</td>
<td>0.002</td>
<td>200.00</td>
<td>0.003</td>
<td>300.00</td>
<td>0.002</td>
<td>200.00</td>
</tr>
<tr>
<td>schur-4x44</td>
<td>N/A</td>
<td>0.100</td>
<td>N/A</td>
<td>0.729</td>
<td>N/A</td>
<td>0.160</td>
<td>N/A</td>
</tr>
<tr>
<td>schur-3x44</td>
<td>N/A</td>
<td>0.012</td>
<td>N/A</td>
<td>0.042</td>
<td>N/A</td>
<td>0.033</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Median Relative % 64.05 175.00 711.43

Note: Times in seconds. Each time is averaged over multiple runs. A time limit of 10 minutes per run was imposed. “Relative” indicates performance relative to Galliwap as a percentage, computed as (SolverTime/GalliwapTime) * 100. Timeouts and relative performance entries which rely on a timeout are indicated with N/A. For medians, timeouts were replaced with a value of 10 minutes.
Additionally, it is able to outperform *cmodels* and *smodels* in most cases. Even when outperformed by other solves, most of our results are still comparable. *Galliwasp’s* performance can vary significantly, even between instances of the same problem, depending on the amount of backtracking required. In the case of programs that timed out or took longer than a few seconds, the size and structure of the programs simply resulted in a massive amount of backtracking. Additionally, our *hanoi* encoding is optimized for goal-directed execution, while the other test programs are not. Additional information on *hanoi* can be found in appendix B.

### 4.4.1 Rule and Goal Ordering

As with normal SLD resolution, rules for a given literal are executed in the order given in the program, with the body of each rule executed left to right. With the exception of OLON rules, once the body of a rule whose head matches a given literal has succeeded, the remaining rules do not need to be accessed unless failure and backtracking force them to be selected.

As in top-down implementations of Prolog, this means that the ordering of clauses and goals can directly impact the runtime performance of a program. In the best case scenario, this can allow *Galliwasp* to compute answers much faster than other ASP solvers, but in the worst case scenario *Galliwasp* may end up backtracking significantly more, and as a result take much longer.

One example of this can be found in an instance of the Schur Numbers benchmark used in section 4.4. Consider the following clauses from the grounded instance for 3 partitions and 13 numbers:

```prolog
:- not haspart(1).
:- not haspart(2).
:- not haspart(3).
```
During benchmarking, Galliwasp was able to find an answer set for the program containing the above clauses in 0.2 seconds. However, the same program with the order of only the above clauses reversed could be left running for several minutes without terminating.

Given that Galliwasp is currently limited to programs that have a finite grounding, its performance can be impacted by the grounder program that is used. In the example above, all 13 ground clauses are generated by the grounding of a single rule in the original program. Work is in progress to extend Galliwasp to allow direct execution of datalog-like ASP programs (i.e., those with only constants and variables) without grounding them first. In such a case, the user would have much more control over the order in which rules are tried.

We believe that manually reordering clauses or changing the grounding method will improve performance in most cases. However, the technique makes performance dependent on the query used to determine the optimal ordering. As a result, the technique is not general enough for use in performance comparisons, and no reordering of clauses was used to obtain the results reported here.
4.5 Related Work

Galliwasp’s goal-directed execution method is based on a previously published technique, but has been significantly refined (Gupta et al., 2007; Min et al., 2009). In particular, the original algorithm was limited to ASP programs which were call-consistent or order-consistent (Min et al., 2009). Additionally, the implementation of the previous algorithm was written as a Prolog meta-interpreter, and incapable of providing results comparable to those of existing ASP solvers. Galliwasp, written in C, is the first goal-directed implementation capable of direct comparison with other ASP solvers.

Various other attempts have been made to introduce goal-directed execution to ASP. However, many of these methods rely on modifying the semantics or restricting the programs and queries which can be used, while Galliwasp’s algorithm remains consistent with stable model semantics and works with any arbitrary program or query. For example, the revised Stable Model Semantics (Pereira and Pinto, 2005) allows goal-directed execution (Pereira and Pinto, 2009), but does so by modifying the stable model semantics underlying ASP. SLD resolution has also been extended to ASP through credulous resolution (Bonatti et al., 2008), but with restrictions on the type of programs and queries allowed. Similar work may also be found in the use of ASP with respect to abduction (Alferes et al., 2004), argumentation (Kakas and Toni, 1999) and tableau calculi (Gebser and Schaub, 2006).

The complete source for Galliwasp is available at (Marple, 2014).
CHAPTER 5
IMPROVING THE PERFORMANCE OF GALLIWASP

5.1 Overview

In this chapter, we discuss the various optimization techniques that *Galliwasp* employs to improve the performance of our basic goal-directed execution strategy. The need to develop such techniques became apparent as soon as we applied our strategy to non-trivial programs. While the performance figures in section 4.4 show that *Galliwasp* performs comparably to solvers such as *clasp* (Gebser et al., 2007a), *cmodels* (Giunchiglia et al., 2004) and *smodels* (Niemelä and Simons, 1997), this was not always the case. Many of the same test programs that now terminate in under a second would run for half an hour or longer on early versions of *Galliwasp*.

The primary source of inefficiency in our goal-directed strategy is backtracking. As with Prolog, each time a goal with two or more rules is expanded, a choice point is created. Should failure occur, the interpreter backtracks to the most recent choice point with unused rules, selects the next rule, and continues execution from there. Each rule associated with a choice point creates a branch in the path that execution may follow, and which may eventually result in failure. Because execution will terminate only when a solution is found or every choice point has been exhausted, the potential time a program take to terminate increases with the number of choice points. As a result, our optimization techniques focus on reducing the amount of backtracking, either by reducing the number of choice points or by detecting failure as early as possible.

The remainder of this chapter is structured as follows. First, compiler-oriented techniques are addressed in section 5.2. Next, NMR-check based-techniques are covered in section...
5.3 After that, CHS-based techniques are addressed in section 5.4. Finally, we look at performance results comparing execution with and without the various techniques in section 5.5 before examining related work in section 5.6.

5.2 Compiler-based Optimization

While the optimization techniques employed by the interpreter are intended to speed up execution directly, the purpose of compile-time optimization is to simplify a program before it ever reaches the interpreter. To this end, Galliwasp’s compilation process includes two major optimization steps: program simplification and common goal factorization.

5.2.1 Program Simplification

Program simplification is the first optimization performed during compilation. Reducing the size and complexity of the program early on allows subsequent steps to finish faster, reducing the overall compilation time. As we will show in section 5.5, execution performance is also improved.

Galliwasp’s primary simplification technique is the use of facts for both positive and negative literals to simplify the rules in a program. The process is as follows:

1. Create facts for negated literals: for any literal \( p \) that does not appear in the head of a rule, create a fact for \( \text{not } p \).

2. Examine each rule, checking each goal \( G \) against the list of known facts:
   
   - If a fact for \( G \) exists, remove the goal. If the body of the rule becomes empty, create a fact for the rule head.
   
   - If a fact for the negation of \( G \) exists, remove the clause. If no clauses remain for the rule head, create a fact for the negation of the head.
3. Repeat step 2 until no new facts are created.

Consider the following program:

\[
\begin{align*}
p & :- \text{not } q, s. \\
q & :- \text{not } p, r. \\
r & :- t, \text{not } s. \\
t. \\
\end{align*}
\]

The first step of the process will create a fact for \text{not } s. The first application of the second step will remove the clause for \text{p}, creating a fact for \text{not } p, and reduce the clause for \text{r} to a fact. The second application of step two will use the new facts to reduce the clause for \text{q} to a fact. The process will end after a third application of step two returns no new facts, producing:

\[
\begin{align*}
\text{q}. \\
\text{r}. \\
\text{t}. \\
\text{not } p. \\
\text{not } s. \\
\end{align*}
\]

By construction, the facts created do not alter the meaning of the program. Positive facts are only created for literals that would always succeed and negated facts are only created for literals that would always fail. In the case of OLON rules, removing a rule means the constraint imposed by the rule will always be satisfied, while creating a fact for the rule means that either the head will succeed through other means, or, in the case of headless rules, that no answer set exists. Thus, under normal execution, this process does not affect correctness.

However, when executing with Dynamic Consistency Checking (DCC), discussed in chapter \[\text{[6]}\], simplifying based on facts can alter the outcome in unexpected ways. This is because DCC relies on splitting sets that are created using OLON rules, and sets that contain one or
more common elements are merged. As such, removing goals from OLON rules may break
the link between two splitting sets that would have otherwise been merged into one. For this
reason, the compiler provides the option to disable program simplification, allowing DCC to
function as intended.

5.2.2 Factorization of Common Goals

While not readily apparent, top-down execution of a grounded ASP program can be very
different from a theoretical execution of the ungrounded program. The cause is the grounding
process itself. Simply enumerating the groundings of each clause, as is done by grounders
such as gringo, lparse and others, can introduce a large number of redundant goals. While
relatively harmless to bottom-up ASP systems, can have a major, negative impact on top-
down execution. Common goal factorization was introduced to eliminate these redundancies.
The process is similar to the way left factoring may be carried out on grammar rules, with
the primary difference being that any goal may be factored out, not just the leftmost one.

To understand why common goal factorization is important, it is necessary to examine
how grounding can introduce redundant goals. Consider the rule:

\[ p :- a(X), b(Y), c(Z). \]

Let the domain of X, Y and Z be \{1, 2, 3\}. An ASP grounder will produce something similar
to:

\[ p :- a(1), b(1), c(1). \]
\[ p :- a(1), b(1), c(2). \]
\[ p :- a(1), b(1), c(3). \]
\[ p :- a(1), b(2), c(1). \]
\[ p :- a(1), b(2), c(2). \]
\[ p :- a(1), b(2), c(3). \]
\[ p :- a(1), b(3), c(1). \]
p :- a(1), b(3), c(2).
p :- a(1), b(3), c(3).
p :- a(2), b(1), c(1).
...
p :- a(3), b(3), c(3).

Were the ungrounded rule executed using co-SLD resolution and backtracking, the goal \texttt{a(X)} would be called at most 3 times, once for each binding of \texttt{X}. However, execution of the grounded rule could result in groundings of \texttt{a(X)} being called up to 27 times, 9 times per binding of \texttt{X}. Redundant calls to groundings of \texttt{b(Y)} can add still more work, as can backtracking over the unnecessary calls.

Common goal factorization addresses this issue by merging rules with the same head and common goals. Common goals are factored out, ensuring that they are called no more than necessary. This factorization is accomplished by creating new literals and rules, referred to as \textit{meta-literals} and \textit{meta-rules}, respectively. The basic procedure is as follows:

1. For each literal \texttt{X} in the program, the set of rules for \texttt{X} is examined to determine the goal called by the most rules, \texttt{G}. If no goal is called by more than one rule, we proceed to the next literal, otherwise we continue to the next step.

2. The rules for \texttt{X} are split into two sets: set \texttt{A} contains those rules which call \texttt{G} and set \texttt{B} contains those rules which do not call \texttt{G}.

3. The head of each rule in \texttt{A} is replaced with a new literal, \texttt{Y}.

4. \texttt{G} is removed from the body of each rule in \texttt{A}.

5. A new rule for \texttt{X} is created: \texttt{X :- G, Y}.

6. Sets \texttt{A} and \texttt{B} are processed recursively.
In this manner, the redundant calls are removed, resulting in execution closer to that of the ungrounded rule. Factorizing the rules above yields:

\[
\begin{align*}
p & : a(1), pa. \\
p & : a(2), pa. \\
p & : a(3), pa. \\
p & : b(1), pa1b. \\
p & : b(2), pa1b. \\
p & : b(3), pa1b. \\
p & : b(1), pa2b. \\
p & : b(2), pa2b. \\
p & : b(3), pa2b. \\
p & : b(1), pa3b. \\
p & : b(2), pa3b. \\
p & : b(3), pa3b. \\
p & : c(1). \\
p & : c(2). \\
p & : c(3). \\
\ldots \\
p & : c(3). 
\end{align*}
\]

Note that factorization actually increases the number of literals and rules in a program. However, execution of the factorized version will mimic execution of the ungrounded rule in terms of the number of calls to non-meta literals. Each grounding of \( a(X) \) will be called at most once, each grounding of \( b(Y) \) will be called at most 3 times, and no unnecessary backtracking will occur. Thus, even though the number of rules has increased, the amount of work potentially needed to execute the rules has been greatly reduced.
Correctness of Factorization

The correctness of our goal-directed method is unaffected by factorization. In the case of ordinary rules, this is readily apparent: by construction, a factorized rule will succeed if and only if at least one of the original clauses used to create it would succeed. In the case of OLON rules, factorization may result in any number of OLON rules being combined with any number of ordinary rules. To retain correctness, a sub-check created from such rules must succeed if and only if the set of sub-checks created from the original OLON rules would succeed.

Consider the following rules, where B and C are conjunctions of one or more literals:

\[
\begin{align*}
\text{p} & :\text{-} a, B, \text{not} \ p. \\
\text{p} & :\text{-} a, C.
\end{align*}
\]

Let a be a literal which cannot indirectly call not p. If C can directly or indirectly call not p, then both rules are OLON rules. If C cannot call not p, then the second rule is an ordinary rule. Creating sub-checks for both rules will yield:

\[
\begin{align*}
\text{chk\_p1} & :\text{-} \text{not} \ a. \\
\text{chk\_p1} & :\text{-} \text{not} \ B. \\
\text{chk\_p1} & :\text{-} \ p. \\
\text{chk\_p2} & :\text{-} \text{not} \ a. \\
\text{chk\_p2} & :\text{-} \text{not} \ C. \\
\text{chk\_p2} & :\text{-} \ p.
\end{align*}
\]

Note that if the second rule is an ordinary rule, then the corresponding sub-check will always be satisfied: if the original rule can succeed, then the third clause of the sub-check will succeed, while if the original rule will fail, then one of the first two clauses must succeed.

Next, consider the factorized version of the above rules:
p :- a, p1.
p1 :- B, not p.
p1 :- C.

which will yield the following sub-check (rules for not p1 added for clarity):

chk_p :- not a.
chk_p :- not p1.
chk_p :- p.
not p1 :- negp11, not C.
negp11 :- not B.
negp11 :- p.

The success of either the first or last clauses of the new sub-check will trivially satisfy both of the original sub-checks. Should the second clause succeed, the success of not p1 ensures that both sub-checks are satisfied: negp11 covers the first sub-check and not C covers the second. Correspondingly, if all three clauses of the sub-check fail, then one or both of the original sub-checks is unsatisfiable. Thus the sub-check created from the factorized rule will succeed if and only if both of the original sub-checks would succeed. It is easy to see that this will always be the case: even when more than two rules are involved or multiple goals are factored out, the structure of the factorized rules, and thus the resulting sub-checks, will remain the same. Therefore, factorization will not affect the correctness of our goal-directed method.

It must be noted that while our method of factorization does not affect correctness, it can alter the operational semantics of a program. Because the most common goal is factored out instead of the leftmost goal, the order in which literals are encountered during execution may change. These changes to goal ordering can affect the sequence in which answer sets are produced as well as performance. The effects on performance are particularly hard to quantify, as they depend on the amount of backtracking that results from the goal ordering
alone. However, in cases where the new goal ordering does not result in a large increase in backtracking, factorization will almost certainly result in a net gain in performance.

5.3 NMR Check Optimization

In practice, the vast majority of a program’s execution time is usually spent applying the NMR check, so techniques to speed it up are of obvious interest. The failure of an NMR sub-check can mean backtracking over every previous goal in the NMR check and query. As a result, the potential cost of failure continually rises as execution progresses. As the NMR check for a large program can contain tens of thousands of sub-checks, the point at which a failure is detected can determine whether a program terminates in under a second or takes the better part of an hour. Therefore, it is extremely desirable that failures be detected as soon as possible. To accomplish this, Galliwasp employs runtime simplification and reordering of the NMR check.

The process of simplifying NMR sub-checks is extremely similar to the compile-time procedure described in section 5.2.1 with succeeding goals used in place of facts. When a literal succeeds, any calls to the literal from an NMR sub-check are removed, along with any sub-check clauses calling the literal’s negation. If the last goal of a sub-check clause is removed, the current partial answer set satisfies the sub-check and it can be skipped altogether. Meanwhile, if the last clause of a sub-check is removed, the check cannot be satisfied by the current partial answer set, so immediate failure and backtracking are triggered.

Unlike their compile-time equivalents, runtime simplifications must be reverted upon backtracking to ensure correctness. As long as the succeeding goal is present in the CHS, the removed goals would always succeed and the removed clauses would always fail. However, if backtracking removes the literal from the CHS, this is no longer guaranteed.

While it has the potential to improve performance by itself, the primary role of our runtime simplification procedure is to assist in dynamically reordering the goals within the
NMR check. Every NMR sub-check must be satisfied for a partial answer set to be valid. Therefore, when simplification reduces a sub-check to a single clause, that clause must succeed for the current partial answer set to be valid. To take advantage of this fact, the NMR check goal for such a sub-check is reordered so that the sub-check is executed immediately after any previously reordered sub-checks. This allows for the success or failure of the clause to be determined as quickly as possible, reducing the amount of backtracking that it might trigger.

Factorization, described in section 5.2.2, can make simplification and reordering much more difficult. The problem lies in the conversion of direct calls to indirect calls through meta-literals. Consider the following factorized rule for p and the resulting NMR sub-check, with dual rules added for clarity:

\[
\begin{align*}
p & : - a, p_1. \\
p_1 & : - b, not \ p. \\
p_1 & : - c. \\
chk_p & : - not \ a. \\
chk_p & : - not \ p_1. \\
chk_p & : - p. \\
not \ p & : - not \ a. \\
not \ p & : - not \ p_1. \\
not \ p_1 & : - not \ p_{11}, not \ c. \\
not_p_{11} & : - not \ b. \\
not_p_{11} & : - p.
\end{align*}
\]

The difficulty lies in detecting when not p_1 will succeed or fail, thus allowing the sub-check to be simplified. For example, if not c is already in the CHS and not b succeeds, the interpreter should detect that not_p_{11} is satisfied, which satisfies not p_1, which in turn satisfies the sub-check itself. To accomplish this, the set of rules on which simplification is
performed is expanded to include rules for any meta-literal which can be called by an NMR sub-check with no intervening non-meta literals. Additionally, a recursive step is added to our simplification algorithm: if simplification causes a literal to succeed or fail, the process is recursively applied using that literal (or its negation in the case of failure). Together, these measures allow simplification to occur even if the meta-literals involved are never directly called during execution.

5.4 CHS-based Optimization

CHS-based optimization consists of adding literals that must succeed to the CHS before they have been reached through normal, top-down execution. As performed by Galliwasp, these additions can be broken into two categories: fact priming and goal assumption.

The simplest form of CHS-based optimization is fact priming. Prior to execution, the CHS is ‘primed’ by adding any facts present in the program. This priming can improve performance in two ways. First, a literal may have multiple clauses in addition to a fact. Because clauses are executed in order, first to last, significant computation may be necessary for such a literal to succeed via another clause, or for all clauses prior to the fact to fail. Fact-priming avoids these situations, allowing any literal with a fact to simply succeed coinductively. The other way in which fact priming can improve performance is by aiding in the simplification of the NMR check described in section 5.3. As facts are loaded into the CHS, they are used to simplify NMR sub-checks just as if the corresponding literal had succeeded during normal execution.

Unlike most of the other optimizations employed by Galliwasp, fact priming can affect correctness. Soundness is maintained, as any partial answer set returned will still be a subset of some valid answer set, but completeness is no longer guaranteed when fact priming is enabled. This is because, while a literal may have other clauses in addition to a fact, fact priming will prevent these clauses from being used. Thus any partial answer set which
can only be created using such a clause will be skipped. In cases where this is behavior is undesirable, fact priming can be disabled to restore correctness.

*Galliwasp*’s second form of CHS-based optimization is goal assumption. Goal assumption takes advantage of the fact that, for a clause to succeed, every goal in the clause must also succeed. When a new clause is selected, such as when expanding a literal or during backtracking, each goal in the clause is added to the CHS with a special “assumption” flag set. If the goal eventually succeeds, the assumption flag is removed. To ensure correctness, assumed literals can be used for coinductive failure, but not for coinductive success. Prohibiting coinductive success guarantees that each assumed literal will eventually succeed or fail during normal execution. Coinductive failure is allowed because, if the negation of an assumed goal were to be added to the CHS, it would always lead to failure when the assumed goal was finally encountered during normal execution. Thus allowing coinductive failure based on such goals simply allows for earlier detection of the impending failure.

### 5.5 Comparative Performance Results

In this section, we present performance results for execution with and without the various optimizations we have discussed. The effects of our optimization techniques can be split into two categories: the effects of compile-time optimizations on the compilation process, and the effects of both compile-time and runtime optimizations on interpretation.

#### 5.5.1 Compiler Performance

We have discussed two compiler optimizations, simplification in section 5.2.1 and common goal factorization in section 5.2.2. Both techniques are applied as early as possible in the compilation process, with simplification performed before factorization. As a result, these optimizations can affect not only the final output, but the remainder of the compilation process as well.
### Table 5.1. Compilation times (in seconds)

<table>
<thead>
<tr>
<th>Problem</th>
<th>No Optimization</th>
<th>Simplification</th>
<th>Factorization</th>
<th>Simp + Fact</th>
</tr>
</thead>
<tbody>
<tr>
<td>queens-12</td>
<td>4.423</td>
<td>4.463</td>
<td>2.382</td>
<td>2.485</td>
</tr>
<tr>
<td>queens-20</td>
<td>79.267</td>
<td>79.635</td>
<td>20.113</td>
<td>21.114</td>
</tr>
<tr>
<td>queens-24</td>
<td>227.380</td>
<td>228.376</td>
<td>46.337</td>
<td>48.567</td>
</tr>
<tr>
<td>mapcolor-4x20</td>
<td>0.271</td>
<td>0.272</td>
<td>0.299</td>
<td>0.097</td>
</tr>
<tr>
<td>mapcolor-4x30</td>
<td>0.466</td>
<td>0.472</td>
<td>0.526</td>
<td>0.536</td>
</tr>
<tr>
<td>schur-3x13</td>
<td>0.380</td>
<td>0.276</td>
<td>0.401</td>
<td>0.270</td>
</tr>
<tr>
<td>schur-2x13</td>
<td>0.220</td>
<td>0.169</td>
<td>0.231</td>
<td>0.168</td>
</tr>
<tr>
<td>schur-4x44</td>
<td>22.419</td>
<td>9.189</td>
<td>18.122</td>
<td>6.936</td>
</tr>
<tr>
<td>schur-3x44</td>
<td>12.815</td>
<td>5.427</td>
<td>10.693</td>
<td>4.346</td>
</tr>
<tr>
<td>pigeon-30x30</td>
<td>223.698</td>
<td>225.531</td>
<td>72.531</td>
<td>74.499</td>
</tr>
<tr>
<td>pigeon-40x40</td>
<td>1402.463</td>
<td>1406.677</td>
<td>278.361</td>
<td>288.046</td>
</tr>
<tr>
<td>pigeon-10x9</td>
<td>0.654</td>
<td>0.655</td>
<td>0.609</td>
<td>0.631</td>
</tr>
<tr>
<td>pigeon-11x10</td>
<td>1.017</td>
<td>1.017</td>
<td>0.892</td>
<td>0.896</td>
</tr>
<tr>
<td>hanoi-4x15</td>
<td>10.836</td>
<td>3.944</td>
<td>10.993</td>
<td>3.951</td>
</tr>
<tr>
<td>hanoi-4x14</td>
<td>7.994</td>
<td>3.058</td>
<td>8.046</td>
<td>3.145</td>
</tr>
<tr>
<td>hanoi-5x31</td>
<td>521.511</td>
<td>51.820</td>
<td>525.118</td>
<td>53.912</td>
</tr>
<tr>
<td>hanoi-5x30</td>
<td>452.847</td>
<td>47.658</td>
<td>439.236</td>
<td>46.968</td>
</tr>
</tbody>
</table>

### Table 5.2. Compiled program sizes (in bytes)

<table>
<thead>
<tr>
<th>Problem</th>
<th>No Optimization</th>
<th>Simplification</th>
<th>Factorization</th>
<th>Simp + Fact</th>
</tr>
</thead>
<tbody>
<tr>
<td>queens-12</td>
<td>347152</td>
<td>347152</td>
<td>156952</td>
<td>156952</td>
</tr>
<tr>
<td>queens-20</td>
<td>1813145</td>
<td>1813145</td>
<td>788702</td>
<td>788702</td>
</tr>
<tr>
<td>queens-24</td>
<td>3214423</td>
<td>3214423</td>
<td>1384451</td>
<td>1384451</td>
</tr>
<tr>
<td>mapcolor-4x20</td>
<td>45064</td>
<td>45064</td>
<td>26178</td>
<td>26178</td>
</tr>
<tr>
<td>mapcolor-4x30</td>
<td>73999</td>
<td>73999</td>
<td>43448</td>
<td>43448</td>
</tr>
<tr>
<td>schur-3x13</td>
<td>56826</td>
<td>34751</td>
<td>36702</td>
<td>22475</td>
</tr>
<tr>
<td>schur-2x13</td>
<td>35601</td>
<td>20907</td>
<td>23138</td>
<td>14017</td>
</tr>
<tr>
<td>schur-4x44</td>
<td>862796</td>
<td>487232</td>
<td>539461</td>
<td>302747</td>
</tr>
<tr>
<td>schur-3x44</td>
<td>629395</td>
<td>348406</td>
<td>384032</td>
<td>219829</td>
</tr>
<tr>
<td>pigeon-30x30</td>
<td>3894199</td>
<td>3894199</td>
<td>1710149</td>
<td>1710149</td>
</tr>
<tr>
<td>pigeon-40x40</td>
<td>9721694</td>
<td>9721694</td>
<td>4270119</td>
<td>4270119</td>
</tr>
<tr>
<td>pigeon-10x9</td>
<td>102127</td>
<td>102127</td>
<td>50954</td>
<td>50954</td>
</tr>
<tr>
<td>pigeon-11x10</td>
<td>141894</td>
<td>141894</td>
<td>69639</td>
<td>69639</td>
</tr>
<tr>
<td>hanoi-4x15</td>
<td>648969</td>
<td>103559</td>
<td>648969</td>
<td>103559</td>
</tr>
<tr>
<td>hanoi-4x14</td>
<td>519854</td>
<td>89983</td>
<td>519854</td>
<td>89983</td>
</tr>
<tr>
<td>hanoi-5x31</td>
<td>8797061</td>
<td>539244</td>
<td>8797061</td>
<td>539244</td>
</tr>
<tr>
<td>hanoi-5x30</td>
<td>7877869</td>
<td>503518</td>
<td>7877869</td>
<td>503518</td>
</tr>
</tbody>
</table>
Table 5.1 gives the overall compilation times for several programs using each combination of optimizations, while Table 5.2 gives the size of the compiled output. Taking into account the overhead of each technique, a general correspondence between compilation time and output size is apparent. This is to be expected: the complexity of each compilation step can be defined in terms of either the size of the program or the number of NMR sub-checks. Thus, reducing the size of the program will speed up subsequent steps.

The size reductions resulting from factorization are indirect. In the short term, factorization typically increases the number of rules and goals in a program. However, combining OLON rules can greatly reduce the number of NMR sub-checks, and duals of factorized rules tend to be simpler than those of unfactorized rules. Thus, even though it increases the number of rules and goals when applied, factorization can indirectly lead to a smaller compiled program.

The impact of each technique depends on the structure of the program in question. A program with no facts that can be inferred will not benefit from simplification, while a program with no redundant calls will not benefit from factorization. As the results demonstrate, often only one of the two optimizations will have a significant impact on a program. However, the technique which is most effective varies from program to program. For n-queens, mapcolor and pigeons, factorization makes a major difference, while simplification does nothing, or next to nothing. For hanoi, the situation is reversed. Meanwhile, schur benefits from both techniques and the cumulative result is better than either individual result.

5.5.2 Interpreter Performance

The primary goal of Galliwasp’s design, as well as the goal of every optimization we have presented, is to minimize the time taken to execute a compiled program. The interpreter is designed to maximize runtime performance, while the compiler is designed to reduce the amount of work that the interpreter must perform. Thus, the interpreter’s performance is the true measure of our success.
Table 5.3. Interpreter performance with various optimization combinations (1 of 2)

<table>
<thead>
<tr>
<th>Problem</th>
<th>Compiler Options</th>
<th>Interpreter Options</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>NMR Only</td>
</tr>
<tr>
<td>queens-12</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Simplify</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Factorize</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Simp+Fact</td>
<td></td>
</tr>
<tr>
<td>queens-20</td>
<td>None</td>
<td>12.581</td>
</tr>
<tr>
<td></td>
<td>Simplify</td>
<td>12.718</td>
</tr>
<tr>
<td></td>
<td>Factorize</td>
<td>2.476</td>
</tr>
<tr>
<td></td>
<td>Simp+Fact</td>
<td>2.571</td>
</tr>
<tr>
<td>queens-24</td>
<td>None</td>
<td>41.243</td>
</tr>
<tr>
<td></td>
<td>Simplify</td>
<td>38.837</td>
</tr>
<tr>
<td></td>
<td>Factorize</td>
<td>5.325</td>
</tr>
<tr>
<td></td>
<td>Simp+Fact</td>
<td>5.279</td>
</tr>
<tr>
<td>mapcolor-4x20</td>
<td>None</td>
<td>.002</td>
</tr>
<tr>
<td></td>
<td>Simplify</td>
<td>.002</td>
</tr>
<tr>
<td></td>
<td>Factorize</td>
<td>.002</td>
</tr>
<tr>
<td></td>
<td>Simp+Fact</td>
<td>.002</td>
</tr>
<tr>
<td>mapcolor-4x30</td>
<td>None</td>
<td>.007</td>
</tr>
<tr>
<td></td>
<td>Simplify</td>
<td>.007</td>
</tr>
<tr>
<td></td>
<td>Factorize</td>
<td>.004</td>
</tr>
<tr>
<td></td>
<td>Simp+Fact</td>
<td>.004</td>
</tr>
<tr>
<td>schur-3x13</td>
<td>None</td>
<td>.003</td>
</tr>
<tr>
<td></td>
<td>Simplify</td>
<td>.002</td>
</tr>
<tr>
<td></td>
<td>Factorize</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Simp+Fact</td>
<td>.104</td>
</tr>
<tr>
<td>schur-2x13</td>
<td>None</td>
<td>.002</td>
</tr>
<tr>
<td></td>
<td>Simplify</td>
<td>.001</td>
</tr>
<tr>
<td></td>
<td>Factorize</td>
<td>.001</td>
</tr>
<tr>
<td></td>
<td>Simp+Fact</td>
<td>.001</td>
</tr>
<tr>
<td>schur-4x44</td>
<td>None</td>
<td>74.518</td>
</tr>
<tr>
<td></td>
<td>Simplify</td>
<td>39.325</td>
</tr>
<tr>
<td></td>
<td>Factorize</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Simp+Fact</td>
<td>N/A</td>
</tr>
<tr>
<td>schur-3x44</td>
<td>None</td>
<td>.174</td>
</tr>
<tr>
<td></td>
<td>Simplify</td>
<td>.084</td>
</tr>
<tr>
<td></td>
<td>Factorize</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Simp+Fact</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Note: Times in seconds. Each time is averaged over multiple runs. A time limit of 10 minutes per run was imposed, with N/A indicating that a timeout occurred. Abbreviations: NMR = NMR check optimization, FP = fact priming, GA = goal assumption.
<table>
<thead>
<tr>
<th>Problem</th>
<th>Compiler Options</th>
<th>Interpreter Options</th>
<th>NMR Only</th>
<th>NMR+FP</th>
<th>NMR+GA</th>
<th>NMR+FP+GA</th>
</tr>
</thead>
<tbody>
<tr>
<td>pigeon-30x30</td>
<td>None</td>
<td></td>
<td>.347</td>
<td>.133</td>
<td>.134</td>
<td>.133</td>
</tr>
<tr>
<td></td>
<td>Simplify</td>
<td></td>
<td>.138</td>
<td>.148</td>
<td>.139</td>
<td>.137</td>
</tr>
<tr>
<td></td>
<td>Factorize</td>
<td></td>
<td>.068</td>
<td>.066</td>
<td>.065</td>
<td>.065</td>
</tr>
<tr>
<td></td>
<td>Simp+Fact</td>
<td></td>
<td>.068</td>
<td>.067</td>
<td>.066</td>
<td>.066</td>
</tr>
<tr>
<td>pigeon-40x40</td>
<td>None</td>
<td></td>
<td>.522</td>
<td>.327</td>
<td>.330</td>
<td>.327</td>
</tr>
<tr>
<td></td>
<td>Simplify</td>
<td></td>
<td>.387</td>
<td>.323</td>
<td>.325</td>
<td>.326</td>
</tr>
<tr>
<td></td>
<td>Factorize</td>
<td></td>
<td>.155</td>
<td>.155</td>
<td>.153</td>
<td>.154</td>
</tr>
<tr>
<td></td>
<td>Simp+Fact</td>
<td></td>
<td>.158</td>
<td>.154</td>
<td>.154</td>
<td>.158</td>
</tr>
<tr>
<td>pigeon-10x9</td>
<td>None</td>
<td></td>
<td>5.422</td>
<td>5.441</td>
<td>5.669</td>
<td>5.683</td>
</tr>
<tr>
<td></td>
<td>Simplify</td>
<td></td>
<td>5.368</td>
<td>5.407</td>
<td>5.639</td>
<td>5.751</td>
</tr>
<tr>
<td></td>
<td>Factorize</td>
<td></td>
<td>2.938</td>
<td>2.913</td>
<td>3.155</td>
<td>3.163</td>
</tr>
<tr>
<td></td>
<td>Simp+Fact</td>
<td></td>
<td>3.027</td>
<td>2.916</td>
<td>3.145</td>
<td>3.127</td>
</tr>
<tr>
<td>pigeon-11x10</td>
<td>None</td>
<td></td>
<td>64.147</td>
<td>63.540</td>
<td>66.357</td>
<td>66.214</td>
</tr>
<tr>
<td></td>
<td>Simplify</td>
<td></td>
<td>63.852</td>
<td>64.964</td>
<td>66.936</td>
<td>66.501</td>
</tr>
<tr>
<td></td>
<td>Factorize</td>
<td></td>
<td>32.211</td>
<td>32.102</td>
<td>35.409</td>
<td>35.015</td>
</tr>
<tr>
<td></td>
<td>Simp+Fact</td>
<td></td>
<td>32.085</td>
<td>32.144</td>
<td>34.560</td>
<td>34.462</td>
</tr>
<tr>
<td>hanoi-4x15</td>
<td>None</td>
<td></td>
<td>.024</td>
<td>.023</td>
<td>.024</td>
<td>.024</td>
</tr>
<tr>
<td></td>
<td>Simplify</td>
<td></td>
<td>.006</td>
<td>.006</td>
<td>.006</td>
<td>.006</td>
</tr>
<tr>
<td></td>
<td>Factorize</td>
<td></td>
<td>.024</td>
<td>.024</td>
<td>.024</td>
<td>.023</td>
</tr>
<tr>
<td></td>
<td>Simp+Fact</td>
<td></td>
<td>.006</td>
<td>.006</td>
<td>.006</td>
<td>.006</td>
</tr>
<tr>
<td>hanoi-4x14</td>
<td>None</td>
<td></td>
<td>.019</td>
<td>.019</td>
<td>.019</td>
<td>.019</td>
</tr>
<tr>
<td></td>
<td>Simplify</td>
<td></td>
<td>.005</td>
<td>.005</td>
<td>.005</td>
<td>.005</td>
</tr>
<tr>
<td></td>
<td>Factorize</td>
<td></td>
<td>.049</td>
<td>.019</td>
<td>.019</td>
<td>.019</td>
</tr>
<tr>
<td></td>
<td>Simp+Fact</td>
<td></td>
<td>.005</td>
<td>.005</td>
<td>.005</td>
<td>.005</td>
</tr>
<tr>
<td>hanoi-5x31</td>
<td>None</td>
<td></td>
<td>.288</td>
<td>.273</td>
<td>.275</td>
<td>.277</td>
</tr>
<tr>
<td></td>
<td>Simplify</td>
<td></td>
<td>.027</td>
<td>.028</td>
<td>.028</td>
<td>.028</td>
</tr>
<tr>
<td></td>
<td>Factorize</td>
<td></td>
<td>.323</td>
<td>.285</td>
<td>.274</td>
<td>.272</td>
</tr>
<tr>
<td></td>
<td>Simp+Fact</td>
<td></td>
<td>.028</td>
<td>.028</td>
<td>.027</td>
<td>.028</td>
</tr>
<tr>
<td>hanoi-5x30</td>
<td>None</td>
<td></td>
<td>.259</td>
<td>.250</td>
<td>.260</td>
<td>.251</td>
</tr>
<tr>
<td></td>
<td>Simplify</td>
<td></td>
<td>.026</td>
<td>.025</td>
<td>.025</td>
<td>.026</td>
</tr>
<tr>
<td></td>
<td>Factorize</td>
<td></td>
<td>.313</td>
<td>.254</td>
<td>.252</td>
<td>.267</td>
</tr>
<tr>
<td></td>
<td>Simp+Fact</td>
<td></td>
<td>.025</td>
<td>.025</td>
<td>.026</td>
<td>.026</td>
</tr>
</tbody>
</table>

*Note:* Times in seconds. Each time is averaged over multiple runs. A time limit of 10 minutes per run was imposed, with N/A indicating that a timeout occurred. Abbreviations: NMR = NMR check optimization, FP = fact priming, GA = goal assumption.
Tables 5.3 and 5.4 show the execution times for various combinations of both compile-time and runtime optimizations. It is important to note that only combinations which include NMR check optimization, described in section 5.3, are shown. NMR check optimization can easily be singled out as having the most significant impact among all of the techniques presented. With it disabled, only queens-12 and the various hanoi instances will succeed within the time limit, with queens-12 taking more than 9.5 minutes. Only the hanoi instances are unaffected by NMR check optimization; our Towers of Hanoi encoding, discussed in detail in appendix B contains only one OLON rule.

After NMR check optimization, the compile-time optimizations have the most significant impact on runtime performance. The most effective technique for each instance is generally the same for both compilation and runtime: n-queens, mapcolor and pigeons benefit more from factorization, while hanoi benefits more from simplification. However, in the case of schur, the situation is somewhat more complex. Cases without factorization show significant benefits from simplification, but almost all cases with factorization simply time out. This is because the schur instances are particularly sensitive to changes in goal ordering. While factorization still reduces the overall complexity of the program, the resulting changes to goal ordering have an extremely negative effect on these instances.

The effects of the remaining runtime optimizations vary greatly from instance to instance. Very few cases show significant benefits from fact priming simply because the programs tested contain few facts. Applications with a large number of facts, such as databases, could expect significantly greater benefits from this technique. Benefits from goal assumption are similarly rare, with the added caveat that the resulting overhead can negatively affect performance in many cases. This is primarily due to the structure of our test programs: with the exception of hanoi, constraints are encoded as OLON rules, shifting them to the NMR check. Our NMR check optimizations handle early failure detection within NMR sub-checks, leaving goal assumption with little to detect. In the case of hanoi, no selected move will lead to failure.
later on, so again goal assumption has little impact. On the other hand, non-deterministic programs which integrate constraints into normal rules would be far more likely to benefit.

As the results indicate, each of the optimizations discussed may be effective, ineffective, or even harmful under different circumstances. To achieve optimal performance, a user must be familiar with a program’s structure and select the appropriate techniques. Even then, the results can be difficult to predict, sometimes requiring trial and error to determine the most effective settings.

5.6 Related Work

Numerous techniques exist to improve the performance bottom-up solvers, including backjumping (Leone et al., 2004) and conflict-driven clause learning (Gebser et al., 2007b). However, these techniques were developed for SAT-based solvers, and adapting Galliwasp to them would require significant modification of our core goal-directed algorithm. This is therefore something we leave for future work. Instead, we developed techniques to improve Galliwasp’s performance while leaving our core execution strategy largely unchanged.

The compiler-based techniques presented are derived from existing techniques used in other areas. Our program simplification technique is based on the application of the $T_P$ operator used in defining the fixpoint semantics of logic programming (Van Emden and Kowalski, 1976). Meanwhile, common goal factorization is based on left factoring, a similar transformation technique applied to grammars in top-down parsing (Aho et al., 2007). The primary difference in our technique is that the goal which occurs the most often is factored out rather than the leftmost.
6.1 Acknowledgements

The paper which forms the basis for this chapter was co-authored by myself and Dr. Gopal Gupta. The underlying research is my own work, along with the majority of the text. A section authored primarily by Dr. Gupta was removed during the editing of the chapter and may be found in the original paper. Additionally, Dr. Gupta provided invaluable feedback and guidance, along with minor modifications to the remaining text.

6.2 Overview

One of the problems which prevents ASP from being adopted on a larger scale is the ability of a minor inconsistency to render an entire knowledgebase useless. Currently, most popular ASP solvers rely on SAT solvers (Giunchiglia et al., 2004; Gebser et al., 2007a) which can’t simply disregard inconsistencies that are unrelated to a query. Because complete answer sets are computed, the underlying program must be consistent. Thus much of the existing work in querying inconsistent knowledgebases has focused on repairing programs to restore consistency (Arenas et al., 2003). In contrast, our goal in this chapter is to be able to work with the consistent part of the knowledgebase, i.e., as long as a query does not invoke clauses from the part of the knowledgebase that is inconsistent, we should be able to execute it and produce an answer set, if one exists. Thus, we do deviate from standard ASP semantics,\footnote{This chapter has been accepted for publication as a separate paper. To appear in Theory and Practice of Logic Programming (TPLP).}
as under ASP semantics, there are no answer sets in the presence of inconsistencies in the knowledgebase.

In this chapter, we introduce *dynamic consistency checking* (DCC), a method for querying inconsistent databases that requires no modification of the underlying programs or queries. Instead, DCC takes advantage of goal-directed answer set programming to ignore inconsistencies that are unrelated to the current query. Additionally, because DCC reduces the number of consistency checks that a partial answer set must satisfy, it can significantly improve the performance of goal-directed execution.

At the core of the problem is the issue of relevance. Because ASP and the underlying stable model semantics lack a *relevance property*, the truth value of an atom can depend on other, totally unrelated rules and atoms \cite{Dix1995}. Because such rules may not be encountered during normal top-down execution, any goal-directed execution strategy for ASP must either alter the semantics or employ some form of consistency checking to ensure correctness. In designing our goal-directed method we chose the latter route, employing consistency checks to ensure that constraints imposed by these rules are satisfied.

DCC employs splitting sets \cite{Lifschitz1994} to reduce the number of consistency checks that must be satisfied while retaining strong guarantees regarding correctness. Execution using DCC employs a modified relevance criteria to determine which consistency checks are relevant to the current partial answer set, and only those checks are enforced.

DCC has been implemented as an extension of the *Galliwasp* system \cite{Marple2013}, which makes use of our original goal-directed method. As we will demonstrate, DCC has several advantages over other potential strategies based on ignoring unrelated inconsistencies. We will show that, if a program has at least one consistent answer set, then a query will succeed using DCC if and only if the partial answer set returned is a subset of some consistent answer set. If no consistent answer set exists, then DCC can allow partial answer sets to be found for a consistent subset of the program. We will also demonstrate
that DCC can improve the performance of goal-directed execution and that partial answer sets produced using DCC can provide more targeted results than either full answer sets or partial answer sets with comprehensive consistency checking.

The remainder of the chapter is structured as follows. In section 6.3 we introduce our technique for dynamic consistency checking using splitting sets and prove several interesting properties. In section 6.4 we examine advantages of DCC and compare the results of Galliwasp with and without dynamic consistency checking.

6.3 Dynamic Consistency Checking

Dynamic Consistency Checking (DCC) began as an attempt to improve the performance of goal-directed execution. While we have developed various other techniques to reduce the performance impact of consistency checking, none of them reduce the actual number of checks that must be satisfied, as this is impossible to do while guaranteeing full compliance with the ASP semantics. DCC was our attempt to reduce the number of checks performed while staying as close to the original ASP semantics as possible.

As any reduction in the number of consistency checks will result in non-compliance with the ASP semantics, selecting which checks to enforce depends on the properties desired from the modified semantics. In the case of DCC, these properties also make the technique useful for querying inconsistent knowledgebases.

Definition 6. For a program P, the desired properties of DCC are:

1. Execution shall always be consistent with the ASP semantics of the sub-program of P (further defined in section 6.3.1).

2. If P has at least one consistent answer set, execution shall be consistent with the ASP semantics of P.

In this section, we discuss the relevance property employed by DCC before moving on to the algorithm itself. Finally, we provide proofs that DCC satisfies the above properties.
6.3.1 Relevance Under DCC

While our original relevance property, given in section 3.5.1 as formula 3.3, makes every consistency check relevant to every literal, DCC selects only those checks necessary to enforce our desired properties from Definition 6. Relevant checks are dynamically selected based on the literals in the partial answer set.

![Program 8](image)

At first glance, it might seem sufficient to select only those checks which directly call literals in the partial answer set (or their negations). However, this can lead to incorrect results. Consider the program in program 8. The program has one consistent answer set: \{c, not a, not b, not p, not q\}. However, given a query \(?- a\), selecting only those checks which directly call some literal in the partial answer set will yield \{a, b, not c\}, thus violating our desired properties.

Clearly, our properties require that we select at least those checks which can potentially reach a literal in the partial answer set. However, this can lead to behavior that is difficult to predict. Consider the program in program 9 with the query \(?- not p\). The presence of either q or not q in each OLON rule might seem to indicate that both consistency checks will be activated and cause the query to fail. However, only chk_1 will be activated. Because
:- p, q.
q :- not r, not q.

chk_1 :- not p.
chk_1 :- not q.
chk_2 :- r.
chk_2 :- q.
nmr_check :- chk_1, chk_2.

Program 9: Example program (consistency checks added).

the first clause will succeed, neither q nor its negation will be added to the partial answer set, and
the query will succeed.

To achieve more predictable behavior, DCC selects relevant checks using specially con-
stucted splitting sets. A splitting set for a program is any set of literals such that if the head
of a rule is in the set, then every literal in the body of the rule must also be in the set
[1994]. The rules in a program P can then be divided relative to a
splitting set U into the bottom, b_U(P), containing those rules whose head is in U, and the
top, P \ b_U(P).

The splitting sets used to determine relevant NMR sub-checks are created by constructing
splitting sets for each NMR sub-check and merging sets whose intersection is non-empty. The
result is a set of disjoint splitting sets U_i such that for an NMR sub-check C, if C ∈ U_i, then
for every literal L reachable by C, L ∈ U_i. This allows us to define the sub-checks relevant
to a literal as those whose heads are in the same splitting set:

dcc_rel_rul(P, L) = rel_rul(P, L) ∪ Oلون(P, L),

Oلون(P, L) = \{R : R ∈ Oلون(P) \ b_{U_i}(P) ∧ L ∈ U_i\}

(6.1)

where Oلون(P, L) is the set of Oلون rules relevant to L. This leads us to DCC’s relevance
property, which defines the semantics of P with respect to L in terms of the new set of relevant
rules:

SEM(P)(L) = SEM(dcc_rel_rule(P, L))(L)

(6.2)
This definition allows for more predictable behavior than simply selecting the checks reachable by a given literal. In the case of programs 8 and 9, only one splitting set will be created, resulting in behavior that is identical to normal goal-directed ASP. Indeed, as we will prove in section 6.3.3, execution will be consistent with ASP whenever a program has at least one answer set.

6.3.2 Execution with DCC

Given DCC’s relevance property in formula 6.2, our goal-directed execution strategy must be modified to enforce it. A query should succeed if and only if every OLON rule relevant to a literal in the partial answer set is satisfied. In addition to creating the associated splitting sets, the application of the relevant NMR sub-checks also becomes more complex.

The creation of the necessary splitting sets can be accomplished by examining a program’s call graph after the NMR sub-checks have been added. A simple depth-first search is sufficient to construct the splitting set for an individual sub-check, after which overlapping sets can be merged. For added efficiency, constructing and merging the sets can be performed simultaneously: whenever a literal is encountered that has already been added to another set, that set is merged with the current one. This eliminates the need to traverse any branch in the call graph more than once. The overhead of searching the sets themselves can be minimized with proper indexing.

To apply the NMR check when executing a query with DCC, it must also be dynamically constructed. The NMR check should consist of those sub-checks which are relevant to a literal in the partial answer set. However, because the sub-checks themselves may add literals to the partial answer set, simply executing the query and then selecting the relevant checks once is insufficient. Instead, each time a literal succeeds, the relevant sub-checks are added to the NMR check. Similarly, the state of the NMR check is restored when backtracking occurs. In this manner, the NMR check will always remain consistent with the current partial answer set.
6.3.3 Correctness of DCC

Now that we have established DCC’s algorithm, we can prove that it satisfies the property it was designed to enforce. That is:

**Theorem 4.** If a program $P$ has at least one consistent answer set, then a query will succeed under DCC if and only if the partial answer set is a subset of some consistent answer set of $P$.

**Proof.** Observe that, if a DCC query succeeds, the partial answer set will be $X = A \cup B$ where

- $A$ is a partial answer set of the splitting set $U$ formed by the union of the splitting sets containing relevant NMR sub-checks
- $B$ is the set of succeeding literals which are not reachable by any NMR sub-check

Per the Splitting Set Theorem ([Lifschitz and Turner, 1994](#)), a set $X'$ is an answer set of $P$ if and only if $X' = A' \cup B'$ where $A'$ is an answer set of $b_U(P)$, $B'$ is an answer set of $e_U(P \setminus b_U(P), A')$, and $A' \cup B'$ is consistent\(^2\) Thus our theory will hold if $A \subseteq A'$, $B \subseteq B'$ and $A' \cup B'$ is consistent.

Because every NMR sub-check relevant to some literal in $A$ will be activated and must succeed for the DCC query to succeed, $A$ will always be a subset of some consistent answer set of $b_U(P)$. Furthermore, such an answer set must exist for the DCC query to succeed. Thus, for any succeeding DCC query, there exists an answer set $A'$ of $b_U(P)$ such that $A \subseteq A'$.\(^3\)

\(^2\)For a set $X$ of positive literals in $U$, $e_U(P \setminus b_U(P), X)$ is a partial evaluation of the top of $P$ with respect to $X$. The partial evaluation is constructed by first dropping rules whose bodies contain the negation of a literal in $X$ and then removing calls to literals in $X$ from the bodies of the remaining rules.

\(^3\)If no literals in the query are reachable by any NMR sub-checks, $U$ will be empty and both $A'$ and $A$ will be the empty set.
Because only OLON rules can lead to inconsistency in an ASP program\footnote{While rules involving classical negation and disjunction can lead to inconsistency, \textit{Galliwasp} handles these by converting them to a set of equivalent normal rules, including OLON rules.}, the set $B$ will always be a subset of some consistent answer set of $e_U(P \setminus b_U(P), A')$, if one exists. Therefore, if at least one consistent answer set exists for $P$, we can select $B'$ such that $B'$ is an answer set of $e_U(P \setminus b_U(P), A)$ such that $B \subseteq B'$.

Finally, because $A'$ contains every NMR sub-check relevant to any literal in $A$, $A'$ will always be consistent with $B'$. Thus, if $P$ has at least one answer set, a query will succeed under DCC if and only the partial answer set is a subset of some consistent answer set of $P$.

\hfill $\Box$

### 6.4 Advantages of DCC

Execution with DCC offers several advantages over normal goal-directed ASP. The three primary advantages are partial answer sets of inconsistent programs, output that is relevant to the query, and improved performance.

#### 6.4.1 Answer Sets of Inconsistent Programs

One disadvantage of ASP is the way in which it handles inconsistency in a knowledgebase. Any inconsistency, no matter how small, renders the entire program inconsistent, and thus no answer set will exist. This behavior can be particularly inconvenient in large knowledgebases where an inconsistency may be completely unrelated to a particular query. Given a large, perfectly consistent database implemented in ASP, adding the rule :- not c. where c is a unique literal, will cause any query to the database to fail.

With DCC, if a query succeeds prior to adding the rule above, then it will continue to succeed even after the rule is added.
6.4.2 Query-relevant Output

One advantage of goal-directed ASP is the ability to compute partial answer sets using a query. Ideally, partial answer sets will contain only literals which are related to the query. However, the execution of the NMR check can force the addition of literals which are unrelated to the current query. By omitting unnecessary NMR checks, DCC can limit this irrelevant output.

Consider the case where two consistent ASP programs, A and B, are concatenated to form a new program C. Assume that A and B have no literals in common and that each contains one or more OLON rules. A full answer set of C will obviously contain literals from both of the sub-programs. As a result of the OLON rules, any partial answer set obtained using goal-directed ASP will also contain literals from both sub-programs. However, using DCC, a succeeding query which targets only one sub-program will only contain literals from that sub-program.

Exploiting this behavior does require care on the part of the programmer. For example, many ASP programs use OLON rules in place of queries. However, such rules will often force all or most of a program’s literals into a single splitting set. As a result, every OLON rule will always be deemed relevant, and DCC will function no differently than normal goal-directed ASP. We will see this behavior in some of the sub-programs examined in the next section.

6.4.3 Performance Compared to Normal Consistency Checking

In this section we compare Galliwasp’s performance on several programs, with and without DCC. As the results in Table 6.1 demonstrate, programs that take advantage of DCC can see a massive improvement in performance. Additionally, even when a program does not take advantage of DCC, the overhead remains minimal.

To simulate programs which take advantage of DCC, the following three programs were concatenated together in various combinations:
Table 6.1. DCC versus non-DCC performance results

<table>
<thead>
<tr>
<th>Problem</th>
<th>Splitting Sets</th>
<th>Query</th>
<th>Execution Times (s)</th>
<th>Original</th>
<th>w/ DCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>hanoi-5x15</td>
<td>0</td>
<td>solveh</td>
<td>.276</td>
<td>.274</td>
<td></td>
</tr>
<tr>
<td>pigeons-30x30</td>
<td>1</td>
<td>solvep</td>
<td>.065</td>
<td>.065</td>
<td></td>
</tr>
<tr>
<td>schur-3x13</td>
<td>1</td>
<td>solves</td>
<td>.105</td>
<td>.105</td>
<td></td>
</tr>
<tr>
<td>hanoi-schur</td>
<td>1</td>
<td>solveh</td>
<td>.134</td>
<td>.028</td>
<td></td>
</tr>
<tr>
<td>hanoi-schur</td>
<td>1</td>
<td>solves</td>
<td>.134</td>
<td>.134</td>
<td></td>
</tr>
<tr>
<td>hanoi-pigeons</td>
<td>1</td>
<td>solveh</td>
<td>.346</td>
<td>.341</td>
<td></td>
</tr>
<tr>
<td>hanoi-pigeons</td>
<td>1</td>
<td>solvep</td>
<td>.343</td>
<td>.342</td>
<td></td>
</tr>
<tr>
<td>pigeons-schur</td>
<td>2</td>
<td>solvep</td>
<td>9.958</td>
<td>.672</td>
<td></td>
</tr>
<tr>
<td>pigeons-schur</td>
<td>2</td>
<td>solves</td>
<td>9.745</td>
<td>.172</td>
<td></td>
</tr>
<tr>
<td>han-sch-pigs</td>
<td>2</td>
<td>solveh</td>
<td>9.817</td>
<td>.093</td>
<td></td>
</tr>
<tr>
<td>han-sch-pigs</td>
<td>2</td>
<td>solvep</td>
<td>9.780</td>
<td>.094</td>
<td></td>
</tr>
<tr>
<td>han-sch-pigs</td>
<td>2</td>
<td>solves</td>
<td>9.942</td>
<td>.201</td>
<td></td>
</tr>
</tbody>
</table>

Note: Each time is averaged over multiple runs. A time limit of 10 minutes per run was imposed, with N/A indicating that a timeout occurred.

- **hanoi-5x15** is a 5 ring, 15 move instance of the Towers of Hanoi. The query `?- solveh` will return a partial answer set containing the solution.

- **pigeons-30x30** is an instance of the MxN-Pigeons problem. The query `?- solvep` will find a complete answer set.

- **schur-3x13** is a 3 partition, 13 number instance of the Schur Numbers problem. The query `?- solves` finds a complete answer set.

Each of the three base programs, and thus each combination, has at least one consistent answer set. The Towers of Hanoi instance contains no OLON rules, and consequently no splitting sets. The other two programs contain OLON rules that force the computation of a complete answer set, and thus have one splitting set each. As a result, a DCC query containing only `solveh` will not activate any NMR sub-checks, while queries containing `solvep` or `solves` will activate every NMR sub-check for their respective problems. Thus DCC execution of `solveh` will not access any splitting sets, while `solvep` and `solves` will access one set each.
In general, the fewer splitting sets accessed by a DCC query relative to the total, the better it will perform compared to a non-DCC query. This is exemplified by the cases with two splitting sets in Table 6.1. In the programs tested, each splitting set represents a large number of OLON rules. As the non-DCC results indicate, the negative impact of increasing the number of OLON rules can be immense. DCC is able to avoid this by satisfying only those rules relevant to the current query.

6.5 Related Work

DCC is an extension of goal-directed ASP (Marple et al., 2012) and has been implemented using the Galliwas system (Marple and Gupta, 2013). The technique relies heavily on the properties of splitting sets, and the Splitting Set Theorem in particular (Lifschitz and Turner, 1994).

Numerous other methods for querying inconsistent databases have been developed. The problem of Consistent Query Answering is defined in terms of minimal database repairs in (Arenas et al., 1999), which develops a technique based on query modification that is built upon in several subsequent works (Celle and Bertossi, 2000; Arenas et al., 2003). However, these techniques require that database inconsistencies be identified and accounted for. Because DCC relies on a goal-directed technique for computing answer sets, our method allows inconsistent information to simply be ignored unless it directly relates to the current query.
CHAPTER 7
FUTURE WORK AND CONCLUSION

7.1 Future Work

Our work in goal-directed ASP has opened several avenues for future work. Chief among these are ungrounded ASP and the integration of ASP with other logic programming techniques.

The primary advantage of goal-directed ASP is that it paves the way to lifting ASP’s restriction to grounded programs, opening the door to ASP with predicates. This is the subject of our current research. The first step, extending our method to Datalog ASP, is already in progress (Salazar et al., 2014). Datalog ASP programs allow only constants and variables in the predicates they contain, among other restrictions (Min, 2010). From there, we hope to extend our system to full predicate ASP. If successful, this will allow the use of ASP with programs that have an infinite or prohibitively large grounding.

Goal-directed execution also allows more natural integration with other logic programming extensions, such as constraint programming, probabilistic reasoning and parallelism. These extensions will in turn allow for additional applications, such as planning with real-time constraints (timed planning) (Bansal, 2007). Many of these extensions have been developed for Prolog, and thus designed with top-down, goal-directed execution in mind. Thus, when adapting these techniques to ASP, goal-directed execution should provide a more natural starting point than bottom-up techniques. Specific possibilities for extension include CLP(R) (Jaffar and Lassez, 1987) for constraints and OR-parallelism (Gupta et al., 2001).
7.2 Conclusions

In this dissertation, we presented a top-down, goal-directed method of executing answer set programs. While other attempts at goal-directed ASP either alter the semantics or restrict the programs and queries which can be executed, our method accepts arbitrary ASP programs and queries while remaining consistent with the ASP semantics. In addition to the algorithm itself, proofs of correctness were provided.

We also discussed the Galliwasp system, an implementation of our goal-directed algorithm, which performs comparably with other state-of-the-art ASP solvers. We detailed the design and implementation of the system and compared its performance results to those of other popular ASP solvers. Additionally, we looked at various techniques for improving Galliwasp’s performance and provided detailed results comparing combinations of these techniques.

Finally, we presented dynamic consistency checking, a technique for querying inconsistent ASP programs. Under normal ASP, minor inconsistencies can render an entire knowledge-base useless, DCC is able to ignore inconsistencies which are unrelated to the current query. This allows partial answer sets to be found for consistent portions of an inconsistent program. We have also proven that DCC is consistent with the ASP semantics for any program which has at least one answer set.

Our work thus addresses two of the three major obstacles to wider adoption of ASP, presented in the introduction: our goal-directed method removes the need to compute a complete answer set for every query, while DCC allows for successful execution in the presence of unrelated inconsistencies. Addressing the third obstacle, the restriction of ASP to grounded programs, is the subject of current and future work.

In conclusion, the techniques we have presented make goal-directed ASP both possible and practical. In addition, we have addressed two of the major obstacles to wider adoption of ASP as a whole and provided a foundation for addressing the third. Our work also opens
a number of doors for extending ASP with techniques developed for other areas of logic programming. The work presented in this dissertation therefore constitutes a significant contribution to the field of answer set programming.
APPENDIX A
HANDLING SPECIAL LITERALS AND RULES

A.1 Overview

Throughout this dissertation, we have limited our discussion of ASP to normal rules and basic literals, as discussed in section 2.2. While we have omitted them from our discussion until now, ASP supports a number of additional literal and rule types. In this appendix, we discuss Galliwasp’s handling of these additional types. As elsewhere, we restrict our discussion to components of grounded ASP programs; constructs removed during the grounding process are omitted.

The reason these types have not been covered elsewhere is that they are primarily syntactic constructs: with the exception of optimization statements, Galliwasp is able to convert these additional types to equivalent basic literals and normal rules. Additionally, ASP grounders may apply such transformations before Galliwasp receives a program; most of Galliwasp’s own transformations are based on those used by lparse (Syrjänen 2000).

A.2 Special Literals

In addition to basic literals, ASP supports a number of additional literal types. These include constraint literals, weight literals and classically negated literals.

A.2.1 Constraint Literals

Constraint literals allow a programmer to place bounds on the number of literals in a set which must succeed. They take the form:

\[
\text{lower} \{ \text{B} \} \text{upper}
\]
where lower and upper are integer bounds and B, the “body” of the constraint literal, is a set of basic literals. A constraint literal succeeds if at least lower and at most upper literals in the body succeed. Each bound is optional, with 0 used in place of a missing lower bound and the number of literals in B used in place of a missing upper bound.

Recall from section 4.3.1 that Galliwasp accepts grounded programs in two formats, lparse’s text output and the smodels input format. In both cases, constraint literals are simplified somewhat during grounding: upper bounds are removed and additional rules are created such that rules with constraint literals have no other goals. To accomplish this, a rule such as

\[ p :- \text{lower} \{ B \} \text{upper}, Q. \]

where B and Q are conjunctions of literals, is replaced by three new rules:

\[ p :- p1, \text{not} p2, Q. \]

\[ p1 :- \text{lower} \{ B \}. \]

\[ p2 :- \text{upper+1} \{ B \}. \]

where p1 and p2 do not occur anywhere else in the program.

This simplifies the matter of handling constraint literals to ensuring that a rule whose only goal is a constraint literal will succeed if and only if the lower bound is satisfied. Galliwasp handles this by computing the power set of the constraint literal’s body and creating a normal rule for each subset whose size is greater than or equal to the lower bound. That is, given a rule

\[ p :- 2 \{ q, r, s \}. \]

the power set of the body is \{\{\}, \{q\}, \{r\}, \{s\}, \{q, r\}, \{q, s\}, \{r, s\}, \{q, r, s\}\}, so the rule will be replaced with the following normal rules:

\[ p :- q, r. \]

\[ p :- q, s. \]

\[ p :- r, s. \]

\[ p :- q, r, s. \]
In cases where the lower bound is zero, meaning the literal should always succeed, a fact is created for the empty set. To improve performance, the new rules are ordered by number of goals, least to greatest, so that larger subsets will be tested last. One possible optimization is to fail once all sets with the smallest cardinality have failed, as larger subsets must contain at least one of the smaller subsets. However, for completeness, the larger subsets should be tested when backtracking to find additional solutions.

A.2.2 Weight Literals

Weight literals are syntactically similar to constraint literals, except that each literal in the body is assigned a weight and square brackets are used instead of braces:

\[
\text{lower}\ [ q_1=w_1, \ldots, q_n=w_n ] \text{ upper}
\]

Rather than count the number of succeeding literals in the body, the bounds are tested against the sum of the succeeding literals’ weights. As with constraint literals, both bounds are optional and missing bounds are replaced with values that will always be satisfied. Weights are integers, and in grounded programs they are always positive; negative weights are permitted in ungrounded programs, but they are converted to positive values during grounding. The transformations applied during grounding are otherwise identical, with new rules being created to remove upper bounds and ensure that rules calling weight literals have no other goals.

*Galliwasp*’s handling of weight literals and constraint literals is almost identical. However, when computing the power set of a weight literal’s body, each subset is associated with the sum of the weights of the literals in the subset. Normal ruler are then created for each subset whose associated weight is greater than or equal to the lower bound. For example, given the rule

\[
p \ :- \ 4 \ [ q=1, \ r=3, \ s=2 ].
\]

the subsets whose weights satisfy the lower bound are \{q, r\}, \{r, s\} and \{q, r, s\}, so the following rules would be created:
\[
\begin{align*}
p & :- q, r. \\
p & :- r, s. \\
p & :- q, r, s.
\end{align*}
\]
As with constraint literals, rules are ordered by number of goals and a fact is created if the empty set satisfies the lower bound.

### A.2.3 Classically Negated Literals

In addition to negation as failure, indicated by “not”, ASP supports classical negation, indicated by prefixing a basic literal with a minus sign. For example, \(-p\) and \(-\text{queens}(1)\) are classically negated literals.

Classical negation simply imposes the constraint that a literal and its classical negation cannot be members of the same answer set. Thus, handling of such literals is extremely simple: for each classically negated literal in a program, an OLON rule is added to enforce the necessary constraint. For example, given the literal \(-p\), the following rule would be added:

\[
:- p, -p.
\]
In all other respects, Galliwasp treats classically negated literals as basic literals; the minus sign is viewed as part of the literal’s string representation rather than as an operator.

### A.3 Special Rule Types

As with literals, ASP support other rule types in addition to the normal rules on which we have focused. The two which may appear in grounded programs are choice rules and optimization statements.

#### A.3.1 Choice Rules

A choice rule is a rule with a constraint or weight literal as its head:
lower \{ p_1, \ldots, p_n \} upper :- B.

or

lower [ p_1=w_1, \ldots, p_n=w_n ] upper :- B.

where B is a conjunctions of zero or more literals.

The meanings of the weight and constraint literals are unchanged: both bounds are optional and a literal is satisfied when its bounds are satisfied. However, the transformations applied during grounding are somewhat different. The \textit{smodels} input format, used by most ASP solvers, does not allow bounds in the head of a rule, so they must be removed. This is accomplished by converting each bounded choice rule into a boundless choice rule with an OLON rule for each bound. For example, the weight rule

\[\text{lower} \[\begin{array}{c}
 p_1=w_1, \ldots, p_n=w_n
\end{array}\]\text{upper} :- B.\]

will be transformed into the rules

\[\begin{array}{c}
\{ p_1, \ldots, p_n \} :- B.
\end{array}\]

\[\begin{array}{c}
:- n - \text{lower} + 1 \[\begin{array}{c}
 p_1=w_1, \ldots, p_n=w_n
\end{array}\], B.
\end{array}\]

\[\begin{array}{c}
\text{upper} + 1 \[\begin{array}{c}
 p_1=w_1, \ldots, p_n=w_n
\end{array}\], B.
\end{array}\]

After removing the bounds from the choice rule, the resulting OLON rules are processed like other rules containing weight or constraint literals, isolating the special literals in their own rules. When \textit{Galliwasp} encounters them, they are processed just like any other weight or constraint literals. Meanwhile, the boundless choice rule requires special handling. Such rules mean that, if the body of the rule is satisfied, any number of literals in the head may or may not be in the answer set. To retain this optional nature, \textit{Galliwasp} creates three rules for each member of the head, two of which form a simple even loop. For example, the rule

\[\begin{array}{c}
\{ p \} :- q.
\end{array}\]

will become

\[\begin{array}{c}
p :- p_1, q.
p_1 :- \text{not negp1}.
negp1 :- \text{not p1}.
\end{array}\]
where $p_1$ and $\neg p_1$ do not appear anywhere else in the program. This encodes that, when $q$ is true, $p$ may or may not be true.

**Disjunctive Rules** Disjunctive rules take the form

$$p_1 \mid \ldots \mid p_n : - B.$$ 

where $B$ is a conjunction of zero or more literals. Such rules are a special form of choice rule, equivalent to

$$1 \{ p_1 \mid \ldots \mid p_n \} 1 : - B.$$ 

*Galliwasp* handles these rules by converting them to the equivalent choice rule and processing them accordingly.

### A.3.2 Optimization Statements

Optimization statements are the only ASP rules which *Galliwasp* cannot convert to equivalent normal rules, and must thus handle in both the compiler and interpreter. These statements consist of a keyword, `minimize` or `maximize`, followed by a boundless weight or constraint literal. For example,

`minimize [ p_1=w_1, \ldots, p_n=w_n ]`. 

and

`maximize { p_1, \ldots, p_n }`. 

are both optimization statements. Unlike other ASP rules, such statements are intended to impact answer sets directly, indicating that an answer set should minimize or maximize the number of succeeding literals (or their weights) in the body of the statement. However, unlike OLON rules, these constraints never lead to failure; they simply encode preference when multiple answer sets are possible. If a program has only one answer set, every optimization statement is trivially satisfied.

Because finding an answer set that is optimal with respect to multiple optimization statements could require finding and comparing every valid answer set for a program, *Galliwasp*
only guarantees optimality with respect to the first optimization statement encountered. The remaining optimization statements yield the most optimal solution \textit{allowed by the current partial answer set}.

During compilation, the body of each optimization statement is expanded as with weight and constraint literals, creating a rule for each subset. Because the bodies have no bounds, every subset, including the empty set, is used. Afterwards, each rule body is modified to add the negation of each literal in the optimization statement but not in the rule body. Then, rules for a single optimization statement are assigned a unique head and sorted by weight or succeeding literals according to desirability: lowest to highest for \texttt{minimize} statements and highest to lowest for \texttt{maximize} statements. For example, the statement

\begin{verbatim}
maximize \{ p, q, r \}.
\end{verbatim}

would produce the rules

\begin{verbatim}
opt :- p, q, r.
opt :- p, q, not r.
opt :- p, not q, r.
opt :- not p, q, r.
opt :- p, not q, not r.
opt :- not p, q, not r.
opt :- not p, not q, r.
opt :- not p, not q, not r.
\end{verbatim}

At runtime, the heads of each optimization statement are inserted as goals in front of the query, in the order in which they occur in the original program. Additionally, unlike other query and NMR check goals, optimization statements are never reordered, and no goals can be placed ahead of them. This ensures that the first partial answer set returned will be optimal with respect to at least the first optimization statement, as the corresponding clauses will have been executed first and in order of optimality. Optimality cannot be guaranteed
for subsequent optimization statements, as each is restricted by the current partial answer set.

The downside to this method of handling optimization statements is that, by placing choicepoints at the very beginning of the query, it can severely impact runtime performance. As we have yet to find another method of handling optimization statements in a goal-directed manner, we simply recommend avoiding them whenever possible.
APPENDIX B

HANOI: TAKING ADVANTAGE OF PARTIAL ANSWER SETS

B.1 Our Towers of Hanoi Encoding

While Galliwasp may sometimes perform well on arbitrary programs, its true potential lies in programs written with top-down execution in mind. Throughout this dissertation, our performance figures include results for hanoi instances on which our implementation performs noticeably better than every other solver. These instances are of particular interest because our Towers of Hanoi encoding, shown in program 10, is designed to take advantage of Galliwasp’s ability to compute partial answer sets, while our other sample programs are not.

Our hanoi encoding is similar to the way the program would be written in Prolog, but it lacks constraints that other solvers need in order to eliminate unwanted moves from the answer set. As a result, Galliwasp’s partial answer set will contain just the moves which constitute a solution, but a complete answer set from another solver will contain every valid move. Additionally, extracting the moves which constitute the desired solution is trivial with Galliwasp’s output but impossible with the full answer set, as the desired moves are indistinguishable from the other legal moves.

To ensure that a bottom-up solver such as clasp will produce only the correct moves, additional constraints, shown in program 11, would need to be added. However, adding constraints increases the size of the grounded program and forces Galliwasp to perform additional, unnecessary work. It also means more work for the programmer, who must spend time reducing the output to something useful.
disk(1..d).
ppeg(1..3).
time(0..t).

% move T: move disk from P1 to P2.
% Any move may or may not be selected.
move(T,P1,P2) :-
    time(T), peg(P1), peg(P2),
    P1 != P2,
    not negmove(T, P1, P2).

negmove(T,P1,P2) :-
    time(T), peg(P1), peg(P2),
    P1 != P2,
    not move(T, P1, P2).

% Move D disks from peg A to peg B using peg C.
% Assign sequential move numbers.
moven(D,TI,TO,A,B,C) :-
    peg(A), peg(B), peg(C),
    time(TI), time(TO), time(T2),time(T3),
    disk(D),disk(M),
    A != B, A != C, B != C,
    TO > T3, T2 > TI,
    D > 1,
    M = D - 1,
    moven(M,TI,T2,A,C,B),
    T3 = T2 + 1,
    move(T3,A,B),
    moven(M,T3,TO,C,B,A).
moven(1,TI,TO,A,B,C) :-
    peg(A), peg(B), peg(C),
    time(TI), time(TO),
    A != B, A != C, B != C,
    TO = TI + 1,
    move(TO,A,B).

% A solution moves all disks in t moves.
:- not moven(d,0,t,1,2,3).

---

Program 10: Our hanoi encoding (ungrounded).
% Only one move per time unit.
:- time(T), peg(P1), peg(P2), peg(P3), peg(P4),
    move(T, P1, P2),
    move(T, P3, P4),
    P1 != P3.
:- time(T), peg(P1), peg(P2), peg(P3), peg(P4),
    move(T, P1, P2),
    move(T, P3, P4),
    P2 != P4.

% No move for time 0.
:- peg(P1), peg(P2),
    move(0,P1,P2).

Program 11: hanoi constraints needed by bottom-up solvers.

B.2 Output Comparison

For the purpose of comparison, this section contains output for hanoi-4x15, the hanoi instance for 4 disks and 15 moves, as produced by Galliwasp and clasp. Galliwasp's partial answer set contains 53 literals, while clasp's complete answer set contains 298 literals. Minor differences in output formatting are preserved: Galliwasp sorts literals alphabetically, wraps its output in braces and inserts commas between literals while clasp simply prints the literals in the answer set.

B.2.1 Galliwasp's Output for hanoi-4x15

{ disk(1), disk(2), disk(3), disk(4), move(1,1,3), move(10,3,1),
  move(11,2,1), move(12,3,2), move(13,1,3), move(14,1,2), move(15,3,2),
  move(2,1,2), move(3,3,2), move(4,1,3), move(5,2,1), move(6,2,3),
  move(7,1,3), move(8,1,2), move(9,3,2), move(1,0,1,1,3,2),
  move(1,10,11,2,1,3), move(1,12,13,1,3,2), move(1,14,15,3,2,1),
  move(1,2,3,3,2,1), move(1,4,5,2,1,3), move(1,6,7,1,3,2),
  move(1,8,9,3,2,1), move(2,0,3,1,2,3), move(2,12,15,1,2,3),


moven(2, 4, 7, 2, 3, 1), moven(2, 8, 11, 3, 1, 2), moven(3, 0, 7, 1, 3, 2),
moven(3, 8, 15, 3, 2, 1), moven(4, 0, 15, 1, 2, 3), peg(1), peg(2), peg(3),
time(0), time(1), time(10), time(11), time(12), time(13), time(14),
time(15), time(2), time(3), time(4), time(5), time(6), time(7), time(8),
time(9) }

B.2.2 Clasp's Output for hanoi-4x15

moven(4, 0, 15, 1, 2, 3) negmove(0, 2, 1) move(1, 2, 1) move(2, 2, 1) move(3, 2, 1)
moven(4, 2, 1) move(5, 2, 1) move(6, 2, 1) move(7, 2, 1) move(8, 2, 1) move(9, 2, 1)
moven(10, 2, 1) move(11, 2, 1) move(12, 2, 1) move(13, 2, 1) move(14, 2, 1)
moven(15, 2, 1) negmove(0, 3, 1) negmove(1, 3, 1) move(2, 3, 1) move(3, 3, 1)
negmove(4, 3, 1) move(5, 3, 1) move(6, 3, 1) move(7, 3, 1) move(8, 3, 1)
moven(9, 3, 1) move(10, 3, 1) move(11, 3, 1) move(12, 3, 1) move(13, 3, 1)
moven(14, 3, 1) move(15, 3, 1) negmove(0, 1, 2) move(1, 1, 2) move(2, 1, 2)
moven(3, 1, 2) move(4, 1, 2) move(5, 1, 2) negmove(6, 1, 2) move(7, 1, 2)
moven(8, 1, 2) move(9, 1, 2) move(10, 1, 2) move(11, 1, 2) move(12, 1, 2)
moven(13, 1, 2) move(14, 1, 2) move(15, 1, 2) negmove(0, 3, 2) negmove(1, 3, 2)
moven(2, 3, 2) move(3, 3, 2) move(4, 3, 2) move(5, 3, 2) move(6, 3, 2) move(7, 3, 2)
moven(8, 3, 2) move(9, 3, 2) move(10, 3, 2) move(11, 3, 2) move(12, 3, 2)
moven(13, 3, 2) move(14, 3, 2) move(15, 3, 2) negmove(0, 1, 3) move(1, 1, 3)
moven(2, 1, 3) move(3, 1, 3) move(4, 1, 3) move(5, 1, 3) move(6, 1, 3) move(7, 1, 3)
moven(8, 1, 3) move(9, 1, 3) move(10, 1, 3) move(11, 1, 3) move(12, 1, 3)
moven(13, 1, 3) move(14, 1, 3) move(15, 1, 3) negmove(0, 2, 3) move(1, 2, 3)
moven(2, 2, 3) move(3, 2, 3) move(4, 2, 3) move(5, 2, 3) move(6, 2, 3) move(7, 2, 3)
negmove(8, 2, 3) move(9, 2, 3) negmove(10, 2, 3) move(11, 2, 3) negmove(12, 2, 3)
moven(13, 2, 3) move(14, 2, 3) negmove(15, 2, 3) disk(4) disk(3) disk(2)
disk(1) peg(3) peg(2) peg(1) time(15) time(14) time(13) time(12)
time(11) time(10) time(9) time(8) time(7) time(6) time(5) time(4)
time(3) time(2) time(1) time(0) moven(1,2,3,1,2,3) moven(1,3,4,1,2,3)
moven(2,1,4,3,2,1) moven(1,1,2,3,1,2) moven(1,4,5,1,2,3)
moven(2,2,5,3,2,1) moven(1,2,3,3,1,2) moven(1,6,7,1,2,3)
moven(2,4,7,3,2,1) moven(1,4,5,3,1,2) moven(1,7,8,1,2,3)
moven(2,5,8,3,2,1) moven(1,5,6,3,1,2) moven(1,8,9,1,2,3)
moven(2,6,9,3,2,1) moven(1,6,7,3,1,2) moven(1,9,10,1,2,3)
moven(2,7,10,3,2,1) moven(1,7,8,3,1,2) moven(1,10,11,1,2,3)
moven(2,8,11,3,2,1) moven(1,8,9,3,1,2) moven(1,11,12,1,2,3)
moven(2,9,12,3,2,1) moven(1,9,10,3,1,2) moven(1,12,13,1,2,3)
moven(2,10,13,3,2,1) moven(1,10,11,3,1,2) moven(1,13,14,1,2,3)
moven(2,11,14,3,2,1) moven(1,11,12,3,1,2) moven(1,14,15,1,2,3)
moven(2,12,15,3,2,1) moven(1,12,13,3,1,2) moven(2,2,5,1,2,3)
moven(2,3,6,1,2,3) moven(3,1,8,2,1,2) moven(2,1,4,3,1,2)
moven(2,5,8,1,2,3) moven(3,3,10,3,2,1) moven(2,6,9,1,2,3)
moven(3,4,11,3,2,1) moven(2,3,6,3,1,2) moven(2,7,10,1,2,3)
moven(3,5,12,3,2,1) moven(2,4,7,3,1,2) moven(2,8,11,1,2,3)
moven(3,6,13,3,2,1) moven(2,5,8,3,1,2) moven(2,9,12,1,2,3)
moven(3,7,14,3,2,1) moven(2,6,9,3,1,2) moven(2,10,13,1,2,3)
moven(3,8,15,3,2,1) moven(2,7,10,3,1,2) moven(2,11,14,1,2,3)
moven(2,8,11,3,1,2) moven(2,12,15,1,2,3) moven(2,9,12,3,1,2)
moven(2,10,13,3,1,2) moven(3,3,10,1,2,3) moven(3,4,11,1,2,3)
moven(3,6,13,1,2,3) moven(3,8,15,1,2,3) moven(3,1,8,3,1,2)
moven(3,2,9,3,1,2) moven(3,4,11,3,1,2) moven(3,6,13,3,1,2)
moven(2,0,3,2,3,1) moven(1,0,1,2,1,3) moven(1,2,3,1,3,2)
moven(1,3,4,1,3,2) moven(2,1,4,2,3,1) moven(1,1,2,2,1,3)
moven(1,4,5,1,3,2) moven(2,2,5,2,3,1) moven(1,2,3,2,1,3)
moven(1,5,6,1,3,2) moven(2,3,6,2,3,1) moven(1,3,4,2,1,3)
moven(1,6,7,1,3,2) moven(2,4,7,2,3,1) moven(1,4,5,2,1,3)
moven(1,7,8,1,3,2) moven(2,5,8,2,3,1) moven(1,5,6,2,1,3)
moven(1,8,9,1,3,2) moven(1,6,7,2,1,3) moven(1,9,10,1,3,2)
moven(2,7,10,2,3,1) moven(1,7,8,2,1,3) moven(1,10,11,1,3,2)
moven(1,8,9,2,1,3) moven(1,11,12,1,3,2) moven(2,9,12,2,3,1)
moven(1,9,10,2,1,3) moven(1,12,13,1,3,2) moven(1,10,11,2,1,3)
moven(1,13,14,1,3,2) moven(2,11,14,2,3,1) moven(1,11,12,2,1,3)
moven(1,14,15,1,3,2) moven(2,12,15,2,3,1) moven(1,12,13,2,1,3)
moven(2,2,5,1,3,2) moven(3,0,7,2,3,1) moven(2,3,6,1,3,2)
moven(2,0,3,2,1,3) moven(2,4,7,1,3,2) moven(3,2,9,2,3,1)
moven(2,2,5,2,1,3) moven(2,6,9,1,3,2) moven(2,3,6,2,1,3)
moven(2,4,7,2,1,3) moven(2,8,11,1,3,2) moven(2,5,8,2,1,3)
moven(2,6,9,2,1,3) moven(2,10,13,1,3,2) moven(2,11,14,1,3,2)
moven(2,8,11,2,1,3) moven(2,10,13,2,1,3) moven(3,3,10,1,3,2)
moven(3,5,12,1,3,2) moven(3,7,14,1,3,2) moven(3,0,7,2,1,3)
moven(3,8,15,1,3,2) moven(3,1,8,2,1,3) moven(3,2,9,2,1,3)
moven(3,3,10,2,1,3) moven(3,4,11,2,1,3) moven(3,5,12,2,1,3)
moven(1,1,2,3,2,1) moven(1,2,3,3,2,1) moven(1,3,4,3,2,1)
moven(1,4,5,3,2,1) moven(1,5,6,3,2,1) moven(1,6,7,3,2,1)
moven(1,7,8,3,2,1) moven(1,8,9,3,2,1) moven(1,9,10,3,2,1)
moven(1,10,11,3,2,1) moven(1,13,14,2,1,3) moven(2,11,14,3,1,2)
moven(1,11,12,3,2,1) moven(1,14,15,2,1,3) moven(2,12,15,3,1,2)
moven(1,12,13,3,2,1) moven(3,8,15,3,1,2) moven(2,12,15,2,1,3)
moven(3,7,14,2,1,3) moven(2,0,3,1,3,2) moven(1,0,1,2,3)
moven(1,2,3,2,3,1) moven(1,3,4,2,3,1) moven(2,1,4,1,3,2)
moven(1,1,2,1,2,3) moven(1,4,5,2,3,1) moven(1,5,6,2,3,1)
moven(1,6,7,2,3,1) moven(1,8,9,2,3,1) moven(1,10,11,2,3,1)
moven(1,12,13,2,3,1) moven(1,13,14,2,3,1) moven(3,0,7,1,3,2)
moven(3,1,8,1,3,2) moven(2,0,3,1,2,3) moven(2,1,4,1,2,3)
moven(3,0,7,1,2,3) moven(3,1,8,1,2,3) moven(1,0,1,2,3,1)
moven(1,1,2,2,3,1) moven(1,13,14,3,1,2) moven(1,14,15,3,1,2)
moven(4,0,15,2,1,3) moven(1,0,1,1,3,2) moven(1,1,2,1,3,2)
moven(1,13,14,3,2,1) moven(1,14,15,3,2,1)
C.1 README

Project page: [https://sourceforge.net/projects/galliwasp/](https://sourceforge.net/projects/galliwasp/)

C.1.1 Description

Galliwasp is a goal-directed answer set solving system. It accepts grounded answer set programs in either normal logic or smodels format and finds produces partial answer sets (if the program has an answer set) using a top-down execution algorithm.

The software consists of two parts: the compiler and the interpreter. An ASP grounder, such as lparse or gringo 4.0, is required but not included. The compiler is written in Prolog, while the interpreter is written in standard C. It has been successfully built under Linux and Mac OS X using gcc and Windows using MinGW.

C.1.2 License

Galliwasp is distributed under GNU Public License, see the file LICENSE included in the distribution for details.

C.1.3 Package Contents

This README should be part of a distribution containing the following files and directories:

- compiler/ – Source code directory for the compiler
- interpreter/ – Source code directory for the interpreter
• CHANGES – Changelog

• config.h – Configuration file used by interpreter

• COPYING – GNU Public License

• INSTALL – Dummy file, just directs the reader here

• Makefile.old – The main Makefile for the project (old version)

• README – This file

• Various files for running the ‘configure’ script and building the Makefile

C.1.4 Installation

Building Galliwasp requires the GNU Compiler Collection (GCC), SWI Prolog and GNU make. It is also assumed that the commands gcc, swipl and make may be used to invoke the three, respectively. Building has been tested using GCC version 4.6.2, SWI-Prolog 64-bit version 6.3.3 and GNU make version 3.81.

If the build environment supports shell scripting (ex. POSIX OSes or Windows w/ MinGW), run .configure to create the Makefile. If the configure script cannot be run, rename Makefile.old to Makefile before continuing.

Finally, to build the compiler and interpreter, simply type make from the project directory. This will create two executables: gw, the interpreter and gwc, the compiler, which can then be moved as needed.

C.1.5 Grounding

To execute a program, it must first be grounded. Simple programs may be grounded by hand, but a third-party grounder will be needed for more complex programs. The compiler
has been tested with text input from lparse (http://www.tcs.hut.fi/Software/smodels/) and smodels input from lparse and gringo 4.0-rc2 (http://potassco.sourceforge.net). Note that gringo’s text output differs substantially from that of lparse and is not currently supported.

The choice of both grounder and input format can affect the output and runtime performance of the interpreter. In particular, grounders may apply transformation to a program before Galliwasp has access to it. For example, the smodels input format separates the goals in a rule based on negation. When compared to equivalent text format programs, this can result in answer sets being produced in a different order, certain literals being omitted from answer sets and differing runtime performance. Additionally, if a grounder simplifies the program by removing goals that will always succeed, this may lead to incorrect results when using dynamic consistency checking (DCC).

Optionally, text format programs may contain a ‘compute’ statement, indicating the number of solutions to compute and the query to execute if the interpreter is run in automatic mode, e.g.

\[
\text{compute } N \{ q_1, \ldots, q_n \}.
\]

where \( N \) is the number of solutions to compute and the list of literals in braces forms the query.

C.1.6 Basic Usage

The compiler reads grounded problem instances from a given file and writes them to either stdout or a file specified with the \(-o\) switch:

\[
gwc \text{ input\_file}
\]

\[
gwc \text{ input\_file } -o \text{ output\_file}
\]

To see an overview of the compiler’s command-line options, type:

\[
gwc -?
\]
Once a program has been compiled, the compiled version can be read by the interpreter from stdin or a file, e.g.

\[ \text{gwc input\_file | gw} \]
\[ \text{gw <compiled\_input>} \]

The interpreter can run in either automatic or interactive modes. Interactive mode can only be used if the compiled program is read from a file rather than stdin. To run in automatic mode, the program instance must contain a compute statement, a non-empty NMR check or at least one optimization statement. When available, the interpreter will default to automatic mode. Otherwise, the interpreter will run in interactive mode. The `-i` switch can be used to force the interpreter to use interactive mode, ignoring the compute statement, if present, e.g.

\[ \text{gw -i <compiled\_input>} \]

For an overview of the available command-line options, type

\[ \text{gw -h} \]

or

\[ \text{gw -?} \]

When run in automatic mode, the interpreter will simply execute the query given by the compute statement to find partial answer sets and print them, stopping when the specified number of sets has been computed or no more sets satisfying the query exist.

In interactive mode, the user will be presented with a prompt to enter a query. For an overview of valid queries, enter `help.` at the prompt. To exit the interpreter when finished, enter `exit.` at the prompt. After each answer set has been printed, the user can enter `.` to accept the set and return to the query prompt. Alternatively, the user can enter `;` to reject the set and find the next one, if it exists. In this way, a user can find as many answer sets as desired. Finally, entering `h` at the prompt will give an overview of both options.
C.1.7 Optimization Statements

As of version 1.2.1, optimization statements (minimization and maximization) are supported. However, due to the implementation, the results may not always be what one would expect. The handling of optimization statements is as follows:

- Optimization statements are executed first, before the query or NMR check.
- Statements are processed in the order they are encountered, first to last.
- Statements yield the most optimal solution allowed by the current partial answer set.

If an answer set exists for a program, the first answer set returned is guaranteed to be optimal w.r.t. the first optimization statement in the program. However, optimality w.r.t. subsequent statements is not guaranteed. Furthermore, if a solution count greater than 1 is specified, subsequent answer sets may be less and less optimal.

C.1.8 Dynamic Consistency Checking (DCC)

As of version 1.3.0, Galliwap supports dynamic consistency checking (DCC). Under normal execution, every NMR sub-check will be enabled by default, and thus eventually executed unless failure can be determined before they are reached. With DCC, NMR sub-checks are DISABLED by default and relevant sub-checks are activated whenever a literal succeeds during execution.

DCC has two primary applications:

- When a query relates to only part of a larger program, DCC can result in improve performance and produce more relevant partial answer sets. For example, suppose a program is created by merging instances of the N-queens problem and the Schur numbers problem. Under normal execution, any answer set will contain solutions to both problems. However, with DCC enabled, either problem can be solved individually based on user-supplied queries.
• DCC can allow partial answer sets to be found for consistent subsets of an inconsistent program (no answer set exists). Note that this will not work in every case: if the call graph of a program is viewed as a set of connected components, DCC will only allow a query to succeed if the connected components accessed by the query are consistent. For example, if we take a program that has at least one answer set and add the rule:

\[ p :- \text{not } p. \]

where \( p \) is not present anywhere else in the program, no answer set will exist for the modified program. However, DCC will allow success for any query that does not contain \( p \) or \( \text{not } p \).

For consistent programs, any partial answer set returned using DCC is guaranteed to be a subset of some valid answer set, UNLESS EITHER OF THE FOLLOWING CONDITIONS HOLD:

• The program was compiled with the simplification option enabled. This can be disabled by running Galliwap’s compiler with the \(-ns\) switch.

• The grounder performed simplification operations on the program before Galliwap’s compiler had access to it.

Additionally, the results produced by a query may not always be as expected. If a compiled program that does not have a hard-coded query is run in automatic mode, you may simply get an empty set. This is because no NMR sub-checks can be relevant to an empty query. On the other hand, larger-than-expected sets may be returned depending on the connected components of the program’s call graph.

C.2 Compiler Usage and Options

Usage: gwc [options] [-o OutputFile] InputFile
‘gwc’ compiles a grounded ASP program using the Galliwasp compiler.

Command-line switches are case-sensitive!

General Options:

- **-h, -?, --help**  Print this help message and terminate.
- **-o**  May be used to specify an output file to use instead of stdout.
- **-v, --verbose**  Enable verbose progress messages.
- **-vv, --veryverbose**  Enable very verbose progress messages.
- **-t, --text**  Print transformed program in text mode.

Compilation Options:

- **-f, --enable-factorization**
- **-nf, --disable-factorization**

   Enable or disable factorization of common prefixes.

   DEFAULT: ENABLED.

- **-s, --enable-simplification**
- **-ns, --disable-simplification**

   Enable or disable simplification of rules based on facts. This option should always be DISABLED when running a program with dynamic consistency checking, otherwise you may get incorrect results!

   DEFAULT: ENABLED.
C.3 Interpreter Usage and Options

Usage: gw [options] [InputFile]

'gw' executes a compiled ASP program with the Galliwas p interpreter. If no input file is specified, input will be read from stdin. Command-line switches are case-sensitive!

General Options:

- h, -?, --help        Print this help message and terminate.
- s, -sX, --solutions=X
          Where X is an integer \( \geq 0 \), can be used to specify the number of solutions to find. -s0 indicates that all solutions should be found. -s without a number will use the default value 0. This can be useful for overriding solution counts hard-coded into a program.
- v, --verbose
          Enable timing information and verbose warning messages. Note that time values may eventually wrap around. On systems with 32-bit clock types, wrapping will occur after 2147 seconds (about 36 minutes).
- vv, -veryverbose
          Enable additional timing information. Implies -v.
- t, --trace
          Print trace information during execution.

Mode Options:
-i, --interactive  Run in interactive mode, allowing the user to enter queries and accept or reject solutions. Requires that the program be read from a file rather than stdin.

-ni, --automatic  Run in automatic mode, executing the program using the hard-coded query and solution count. Solution count still be overridden via the corresponding switch. Requires that the program be read from a file rather than stdin. DEFAULT MODE.

Output Options:

-a, --print-all  Print the NAF (negation as failure) literals in an answer set in addition to any positive literals. NAF literals are of the form ‘not p’. Note that classically negated literals (of the form ‘-p’) are treated as positive literals internally and will always be printed.

-na, --print-positive  Print only the positive literals in an answer set. This includes classically negated literals due to their internal representation as positive literals. DEFAULT MODE.

Execution Options:
-f, --enable-fact-priming

-nf, --disable-fact-priming

Enable or disable priming the CHS with facts before execution. Enabling this option will generally improve performance. DEFAULT: ENABLED.

-r, --enable-nmr-reordering

-nr, --disable-nmr-reordering

Enable or disable reordering of the NMR check goals during execution.

If enabled, NMR sub-checks that become deterministic will be executed sooner, allowing for faster failure detection. Enabling this option will generally improve performance. DEFAULT: ENABLED.

-q, --enable-query-reordering

-nq, --disable-query-reordering

Enable or disable the placing of reordered NMR check goals immediately after the current query goal.

If enabled, reordered NMR check goals will be inserted immediately after the current query goal, or at the beginning of the query proper if the current query goal is an optimization statement. If disabled, reordered goals will only be placed ahead of other NMR check goals. This option has no effect.
if NMR reordering is disabled.

The performance impact of this option will vary based on how the query restricts the search space. You may wish to disable this option if a program may have multiple answer sets but the query will cause most to be rejected. DEFAULT: ENABLED.

-S, --enable-simplify-nmr
-nS, --disable-simplify-nmr
Enable or disable the simplification of NMR sub-checks based on goals that succeed during execution. This option is implied by NMR check reordering and will only have an affect execution if reordering is disabled. DEFAULT: ENABLED.

-A, --enable-goal-assumption
-nA, --disable-goal-assumption
Enable or disable goal assumption, which preloads goals into the CHS before they are expanded, to allow for faster detection of failure. Pre-loaded goals are not used for coinductive success. Enabling this option will generally improve performance. DEFAULT: ENABLED.

-d, --enable-dynamic-nmr
-nd, --disable-dynamic-nmr
Enable or disable dynamic NMR check construction, which enforces only those NMR sub-checks that
relate to goals in the partial answer set (or their
negations). This can improve performance in
consistent programs and allow partial answer sets
to be found for consistent portions of inconsistent
programs. DEFAULT: DISABLED.

-p, --enable-positive-loops
-np, --disable-positive-loops

Enable or disable coinductive success on "positive
loops", that is, without any negations between a
recursive call and its ancestor. This is normally
disallowed by the ASP semantics, but has some
interesting applications. DEFAULT: DISABLED.

Stack Options:

The default values for stack-related options should be suitable for most
cases. However, if you receive an error related to stack space or wish to
fine-tune the interpreter to improve performance, you may wish to adjust
these values.

There are currently 5 stacks, each for reverting specific types of
changes when backtracking. Each stack has two options: the maximum length
and the initial length. Increasing the maximum length will allow a stack
to accommodate larger or more complex programs. Alternatively, setting
the maximum length to 0 will allow a stack unlimited growth, so long as
the memory can be allocated. Increasing the initial length can improve
performance by reducing the number of times a stack will need to be expanded, at the cost of additional memory usage when the extra space is not necessary. The individual stacks are explained along with their options below.

--and-stack=X
--initial-and-stack=X The AND stack keeps track of the current goal in each selected clause.
DEFAULT MAXIMUM LENGTH: 0
DEFAULT INITIAL LENGTH: 2048

--choicepoint-stack=X
--initial-choicepoint-stack=X
The choicepoint stack keeps track of the currently selected clause for goals with two or more clauses available.
DEFAULT MAXIMUM LENGTH: 0
DEFAULT INITIAL LENGTH: 2048

--chs-stack=X
--initial-chs-stack=X The CHS stack keeps track of changes made to the coinductive hypothesis set (CHS), which also forms the candidate answer set.
DEFAULT MAXIMUM LENGTH: 0
DEFAULT INITIAL LENGTH: 2048
--activation-stack=X
--initial-activation-stack=X

The activation stack keeps track of which NMR check goals and clauses have been deactivated based on the success or failure of literals during execution.
DEFAULT MAXIMUM LENGTH: 0
DEFAULT INITIAL LENGTH: 2048

--query-stack=X
--initial-query-stack=X

The query stack keeps track of changes made to the order of query goals by dynamic reordering.
DEFAULT MAXIMUM LENGTH: 0
DEFAULT INITIAL LENGTH: 2048

C.4 Compiled Program Format

While Galliwasp’s compiler accepts input in either the commonly used smodels format [Niemelä and Simons 1997] or the text format produced by lparse [Syrjänen 2000], the interpreter only accepts input in a special format produced by the compiler. In this section, we discuss this compiled program format (CPF) and provide a grammar for parsing it.

As explained in section 4.3.1 the motivation behind splitting Galliwasp into two components was to maximize the performance of the interpreter. The same holds true for the design of the CPF. In addition to the program rules, NMR check and symbol table, the CPF contains a number of other components to simplify processing by the interpreter. In general,
any steps the can be performed at compile time without altering runtime execution are off-
loaded onto the compiler and their output included in the compiled program. This includes
simple steps like sorting rules and symbols to simplify indexing and computing size metrics
to allow the interpreter to immediately allocate memory for internal data structures. How-
ever, more complex operations are also performed: two indexes, one for reordering the NMR
check and one for use in DCC, are computed at compile time and passed to the interpreter
as part of the compiled program.

C.4.1 CPF Grammar

Below we provide a Prolog definite clause grammar (DCG) to parse a CPF file into a simple
parse tree. Note that Prolog DCG’s accept a string of tokens produced from a file by a
tokenizer. For convenience, it is assumed that the tokenizer wraps integers with int() and
strings with str(). For example, 5 would be tokenized as int(5). Additionally, the CPF
expects newlines in various places, so it is assumed these are left in place by the tokenizer.
Otherwise, terminals are separated by single spaces in the file. Finally, while the CPF
requires sorting in several places, sort order is not tested by this grammar. Rules are sorted
by head, symbol table entries by string symbol, and call index entries by call.

% A compiled program consists of several components:
% - Program size metrics
% - Flags for compiler options used
% - Rules
% - Query components
% - A symbol table for literals
% - An index to assist in NMR check reordering
% - An index to assist in dynamic consistency checking (DCC)
parse_cpf(Program) -->
size_metrics(M),
compiler_options(O),
{M = metrics(_, RuleCount, _)},
rules(RuleCount, R),
query_components(Q),
symbol_table(S),
nmr_reordering_index(N),
dcc_index(D),
{Program = program(M, O, R, Q, S, N, D)}.

% Size metrics provided by the compiler allow the interpreter to allocate
% space for various data structures without needing to guess, reallocate
% later, or read the data more than once. Rule count must be greater than
% or equal to one, while the other values must be greater than or equal to
% zero. In practice, the symbol count will always be greater than zero.
size_metrics(metrics(SymbolCount, RuleCount, TotalGoals)) -->
[int(SymbolCount)], ['\n'],
{SymbolCount >= 0},
[int(RuleCount)], ['\n'],
{RuleCount >= 1},
[int(TotalGoals)], ['\n'],
{TotalGoals >= 0}.

% The interpreter uses compiler options solely to provide warnings when
% options may affect output or correctness. For example, simplification can
% interfere with dynamic consistency checking. Each option can be 0 or 1.
compiler_options(options(Simplify, Smodels, Factorize)) -->
  [int(Simplify)], % Fact-based simplification.
  [int(Smodels)], % Program used smodels input format.
  [int(Factorize)], % Factorization of common prefixes.
  ['\n'].

% Program rules. The number of rules must match the rule count given in the
% size metrics and a program must contain at least one rule. Finally, rules
% must be sorted by their head.
rules(RCnt, rules([Rule | Rules])) -->
  {RCnt > 0},
  rule(Rule),
  {RCnt2 is RCnt - 1},
  rules2(RCnt2, Rules).

rules2(RCnt, [Rule | Rules]) -->
  {RCnt > 0},
  rule(Rule),
  {RCnt2 is RCnt - 1},
  rules2(RCnt2, Rules).

rules2(0, []) -->
  [].

% A rule is an integer head followed by a body.
rule(rule(Head, Body)) -->
  [int(Head)],
body(Body).

% The query components include a solution count, the total "fixed" length
% (optimization statement + NMR check), optimization statement goals, NMR
% check goals, and finally the query itself. The solution count and query
% proper may be overridden. A solution count of zero indicates that all
% solutions should be found.
query_components(query(SolCnt, FixLength, Optimize, NMRChk, Query)) -->
  [int(SolCnt)], [\n],
  \{SolCnt >= 0\},
  [int(FixLength)], [\n], % Opt. statement goals + NMR check goals.
  \{FixLength >= 0\},
  body(Optimize), % Goal lists are stored the same way as rule bodies.
  body(NMRChk),
  body(Query).

body(Goals) -->
  [int(GoalCount)], % Zero or more goals, zero = fact.
  goals(GoalCount, Goals), % Exactly GoalCount goals.
  [\n].

goals(GoalCount, [Goal | Goals]) -->
  \{GoalCount > 0\},
  [int(Goal)],
  \{GoalCount2 is GoalCount - 1\},
  goals(GoalCount2, Goals).
goals(0, []) -->
  [].

% The symbol table associate integer symbols with their strings. Note that
% the total number of symbols may differ from the number of symbol table
% entries. This is because literals created during compilation do not have
% strings associated with them. The symbol table cannot be empty.
symbol_table(sym_table(SymTable)) -->
  [int(Length)], ['\n'],
  symbol_table_entries(Length, SymTable). % Exactly Length entries.

symbol_table_entries(Length, [Entry | Entries]) -->
  {Length > 0},
  sym_entry(Entry),
  {Length2 is Length - 1},
  symbol_table_entries2(Length2, Entries).

symbol_table_entries2(Length, [Entry | Entries]) -->
  {Length > 0},
  sym_entry(Entry),
  {Length2 is Length - 1},
  symbol_table_entries2(Length2, Entries).
symbol_table_entries2(0, []) -->
  [].

% Symbol table entries are simply a positive integer followed by the
% associated string. Strings for negative literals are created by appending
% "not " to the string for the positive literal.
sym_entry(entry(NumSymbol, StringSymbol)) -->
    [int(NumSymbol)],
    [str(StringSymbol)],
    ['\n'].

% The NMR check reordering index stores a tuple for each goal in an NMR
% sub-check, as well as goals in rules that were created by factorizing
% OLON rules. This index allows reordering to be performed in constant
% time. Note that the index will be empty if a program contains no OLON
% rules.
nmr_reordering_index(nmr_index(Index)) -->
    [int(Length)], ['\n'], % The number of index entries.
    nmr_reordering_index_entries(Length, Index).

nmr_reordering_index_entries(Length, [Entry | Entries]) -->
    {Length > 0},
    nmr_reordering_entry(Entry),
    {Length2 is Length - 1},
    nmr_reordering_index_entries(Length2, Entries).

nmr_reordering_index_entries(0, []) -->
    [].

% For each goal in the reordering index, the following information is
% stored:
% - Call: the goal itself.
% - Caller: the head of the rule containing the goal.
% - Clause: the position of the clause in which the Call appears. Count is
%   for clauses with head Caller only. Starts at 0 for the first
%   clause with Caller as its head.
% - Goal: the position of the goal in the rule body, starting at 0.
nmr_reordering_entry(entry(Call, Caller, Clause, Goal)) -->
  [int(Call)],
  [int(Caller)],
  [int(Clause)],
  [int(Goal)],
  ['\n'].

% The DCC index links NMR sub-checks with the literals whose success will
% activate them during execution with DCC. If a program contains no OLON
% rules, the index will be empty.
dcc_index(dcc_index(Index)) -->
  [int(Length)], % Number of entries (two lines each)
  [int(TotalGoals)], ['\n'], % Used to allocate space index.
dcc_index_entries(Length, Index).

dcc_index_entries(Length, [Entry | Entries]) -->
  {Length > 0},
  dcc_index_entry(Entry),
  {Length2 is Length - 1},
  dcc_index_entries(Length2, Entries).
dcc_index_entries(0, []) -->
    [].

% Each DCC index entry consists of a list of NMR sub-check goals to be
% activated, followed by a list of literals which will trigger the
% activation upon success. Both lists are stored the same way as rule
% bodies.
dcc_index_entry(entry(NMRGoals, Literals)) -->
    body(NMRGoals),
    body(Literals).
REFERENCES


VITA

Kyle Marple was born on August 1, 1986 in Lawton, Oklahoma. He first became interested in computer science while taking a computer programming course at Lawton High School. He received his Bachelor of Science in Computer Science from Cameron University in December 2007. He continued to study computer science at The University of Texas at Dallas, receiving his Master of Science in Computer Science in May 2009.

It was at The University of Texas at Dallas that he was first introduced to logic programming, and eventually answer set programming. He has since developed Galliwasp, the first practical, goal-directed answer set programming system, and has published several papers on the topic.