Normalization and Rewriting for Answer Set Programming and Optimization

Jori Bomanson
Normalization and Rewriting for Answer Set Programming and Optimization

Jori Bomanson

A doctoral dissertation completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, on 30 October 2020 at 17:00.

Aalto University
School of Science
Computer Science
Computational Logic
Supervising professor
Professor Ilkka Niemelä, Aalto University, Finland

Thesis advisors
Professor Martin Gebser, University of Klagenfurt, Austria
Professor Tomi Janhunen, Aalto University, Finland

Preliminary examiners
Professor Francesco Ricca, University of Calabria, Italy
Professor Pedro Cabalar, University of Corunna, Spain

Opponent
Associate Professor Yuliya Lierler, University of Nebraska, Omaha

Aalto University publication series
DOCTORAL DISSERTATIONS 157/2020

© 2020 Jori Bomanson

ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)

Unigrafia Oy
Helsinki 2020

Finland
Abstract

Answer-set programming (ASP) is an automated problem-solving paradigm specifically suitable for combinatorial search and optimization problems. Such problems can be solved both in theory and also increasingly in practice by ASP technology. The purpose of this dissertation is to contribute to ASP technology and particularly to the efficiency of ASP solving tools. In more detail, the contributions of this thesis focus on ASP optimization algorithms as well as ASP search algorithms based on so-called lazy-grounding. To these ends, several methods are proposed and shown theoretically and experimentally to have measurable performance increasing potential. The methods belong to two novel classes of ASP transformations: optimization rewritings, aimed at enhancing optimization performance, and first-order lazy-normalizations aimed at extending the applicability of lazy-grounding in general. On the topic of transformations, this dissertation moreover presents novel systematic ways to test and verify the correctness of certain types of ASP transformations. These verification methods are distinct in the way they compare programs, which has been designed to reflect the behavior of standard ASP solvers to be as practically relevant as possible. Regarding practical optimization performance, the value of this work is substantial as evidenced by the excellent optimization performance of the proposed techniques in the highly competitive ASP Competition series. In particular, early versions of the techniques attained first place in the optimization track of the sixth edition of the series [56]. Moreover, the included work on lazy-grounding removed an obstacle previously preventing lazy-grounding from being applied to broad classes of programs that are central to the ASP paradigm.
Tiivistelmä

This dissertation presents research conducted in the Computational Logic group of the department of Computer Science at Aalto University during the years 2014–2020. My time at the department began already before these years as a research assistant supervised by Prof. Tomi Janhunen and Prof. Martin Gebser from the group, both of whom also continued as my doctoral advisors. I am grateful for their support, which was a crucial factor already in my decision to begin this doctoral project, and especially for the continuous support by Tomi over all these years. In addition, I am thankful to my supervisor, Prof. Ilkka Niemelä, particularly for encouragement. I would also like to thank my co-authors Benjamin Kaufmann and Prof. Torsten Schaub, as well as Antonius Weinzierl who taught me a great deal about science and especially about believing in one’s own work. Now that I am finally preparing for my defense, I am thankful to Prof. Yuliya Lierler for agreeing to be my opponent. Likewise, I am happy that my dissertation was examined by Prof. Pedro Cabalar and Prof. Francesco Ricca. Thank you for your appreciation and valuable feedback on my dissertation. During my study years, I also had the pleasure of interning twice at Google in California in a project hosted by Dr. Alex Brik. I would like to thank him for being an excellent host and a friend, and many others at Google for times of growth and fun in California. Moreover, I would like to thank Bart, Jussi, Lukas, Masoumeh, Mika, Saurabh, Shahab, Tommi, Yingying, and others for company at the CS department. Furthermore, I would like to thank my parents and my brother for unconditional support in various forms including but not limited to food and hot chocolate. I would also like to thank Judy for being my great friend! Finally, I would like to thank Luiza for bringing so much joy, wisdom and meaning to my life!

Espoo, September 28, 2020,

Jori Bomanson
Contents

Preface 1
Contents 3
List of Publications 5
Author's Contribution 7
List of Publications 9
Abbreviations 11

1. Introduction 13
  1.1 Contributions ..................................... 20
  1.2 Related Work ..................................... 23
    1.2.1 Relation to Other Publications by the Author .... 24
    1.2.2 Solving Paradigms ................................ 25
    1.2.3 Aggregates .................................... 26
    1.2.4 Translation Techniques in Non-Standard Settings ... 27
    1.2.5 Lazy-Grounding Techniques ......................... 29
  1.3 Dissertation Structure ............................. 30

2. Preliminaries 33
  2.1 ASP Syntax .................................. 33
  2.2 ASP Semantics .................................. 35
  2.3 ASP with First-Order Variables ..................... 36
  2.4 ASP Equivalence ................................ 38
  2.5 ASP Programs for Merging and Sorting ............... 40
  2.6 Comparator and Sorting Networks .................... 41
  2.7 ASP Programs that Encode Comparator and Sorting Networks 43

3. Normalization and Visible Strong Equivalence 45
  3.1 Normalization .................................. 46
  3.2 Visible Strong Equivalence ......................... 48
List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals, and which are listed here with authors in alphabetic order.


Author’s Contribution

Publication I: “LP2NORMAL — A Normalization Tool for Extended Logic Programs”

The author was responsible for all aspects of the publication.

Publication II: “Applying Visible Strong Equivalence in Answer-Set Program Transformations”

The author developed and wrote Section 4 on refining the main results of the paper for specific sub-classes of programs, wrote Section 7 on experiments, participated in writing Section 1 introducing the work and Section 5 on the correctness of example program transformations, developed some of the related software, and performed the computational experiments.

Publication III: “Enhancing Lazy Grounding with Lazy Normalization in Answer-Set Programming”

The author developed the encodings used in first-order normalization, participated in writing especially Section 4 on the developed encodings, and assisted in running the computational experiments.

Publication IV: “Rewriting Optimization Statements in Answer-Set Programs”

The author participated in developing the ideas, wrote the majority of the paper, prepared preliminary versions of the formal results and proofs, developed the related software, and performed the computational experiments.
Author’s Contribution

Publication V: “Boosting Answer Set Optimization with Weighted Comparator Networks”

The author participated in developing the ideas, wrote the majority of the paper, prepared all formal results and proofs except for Lemma 1 on translating comparator networks to answer set programs, developed the related software, and performed the computational experiments.
Other Publications

The following publications are related but not included in this dissertation.


<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASP</td>
<td>Answer-Set Programming</td>
</tr>
<tr>
<td>ASPCOMP</td>
<td>Answer-Set Programming Competition</td>
</tr>
<tr>
<td>CDLD</td>
<td>Conflict-Directed Lazy Decomposition</td>
</tr>
<tr>
<td>CNF</td>
<td>Conjunctive Normal Form</td>
</tr>
<tr>
<td>EVA</td>
<td>Enough Visible Atoms</td>
</tr>
<tr>
<td>IHS</td>
<td>Implicit Hitting Set</td>
</tr>
<tr>
<td>KR&amp;R</td>
<td>Knowledge Representation and Reasoning</td>
</tr>
<tr>
<td>MIP</td>
<td>Mixed Integer Programming</td>
</tr>
<tr>
<td>NP</td>
<td>Non-deterministic Polynomial time</td>
</tr>
<tr>
<td>OMT</td>
<td>Optimization Modulo Theories</td>
</tr>
<tr>
<td>PB</td>
<td>Pseudo-Boolean</td>
</tr>
<tr>
<td>PBO</td>
<td>Pseudo-Boolean Optimization</td>
</tr>
<tr>
<td>PCH</td>
<td>Propagator Call History</td>
</tr>
<tr>
<td>SAT</td>
<td>Boolean Satisfiability</td>
</tr>
<tr>
<td>SE</td>
<td>Strong Equivalence</td>
</tr>
<tr>
<td>SEQ</td>
<td>Sequential Counter</td>
</tr>
<tr>
<td>SMT</td>
<td>Satisfiability Modulo Theories</td>
</tr>
<tr>
<td>SWC</td>
<td>Sequential Weight Counter</td>
</tr>
<tr>
<td>VSE</td>
<td>Visible-Strong Equivalence</td>
</tr>
</tbody>
</table>
1. Introduction

This dissertation addresses the development and enhancement of automated solving techniques for combinatorial search problems. These problems are solvable, at least in theory, by enumerating vast numbers of candidate solutions and checking each candidate against given criteria. A more practical approach is to solve these problems instead by using, for example, Answer Set Programming (ASP)\cite{58,59,86,95}—a problem-solving approach in the style of Declarative Programming. A user solves problems via ASP by declaring the defining properties of the problems in a rule based language and using automated solving tools to find the solutions. On a methodological level, ASP is a form of Logic Programming: it relies on mathematical logic and rules for the specification of computational problems in place of standard programming languages. At the same time, ASP belongs to the field of Knowledge Representation and Reasoning (KR&R)\cite{17} in that it concerns the formal representation of knowledge and the task of computationally reasoning about the world. Since its inception, ASP has matured into a useful problem-solving paradigm, supported by efficient solving tools\cite{5,53} and successfully deployed in applications including product configuration\cite{106}, call-enter customer routing\cite{77}, and phylogenetic inference\cite{27,75}. Further applications are surveyed by Leone and Ricca (2016)\cite{77} and Erdem et al. (2016)\cite{44}. Moreover, the following contains a brief overview of central aspects of ASP as well as the specific focus points of this dissertation, while an interested reader may additionally consult the literature for a broader introduction\cite{26,70}.

In terms of computational complexity, in its most basic propositional form ASP tackles the prominent class of computational problems called nondeterministic polynomial time (NP) consisting of all problems that can be solved by a nondeterministic Turing machine in polynomial time. The class NP includes, for example, the decision version of the famous Traveling Salesperson (TSP) problem and the Boolean Satisfiability (SAT) problem. Particularly the hardest problems in this class, the so-called NP-complete problems, suit search based computational approaches, such as ASP, since they currently elude computationally lighter solutions running in deterministic polynomial time. Additionally, ASP provides solutions to problems of even higher complexity in the polynomial-time hierar-
Introduction

...problems in $\Delta^P_2$ via optimization [104], problems in $\Sigma^P_2$ via disjunctive logic programming [42], problems in $\Delta^P_3$ via optimization of disjunctive logic programs [78], and problems on arbitrary levels of the hierarchy via so-called stable-unstable semantics [24].

To make the context of this dissertation more concrete, the following program exemplifies how conveniently an ASP program can capture a combinatorial search problem. The problem concerns a graph and the search for a subset of no more than 10 graph nodes such that at least 100 nodes are reachable from the subset. As input, the program takes a graph encoded with the one-place predicate `node` and the two-place predicate `edge`.

% Non-deterministically pick up to 10 nodes from a graph.

{picked_node(U):node(U)} 10.

% Compute the set of nodes reachable from the picked ones.

reachable(U):-picked_node(U).

reachable(U):-reachable(V), edge(V,U).

% Reject answers with fewer than 100 reachable nodes.

:¬#count(U:reachable(U)) < 100.

For now, the reader may rely on the comments in the program for the intuition of its meaning. More detailed explanations of the syntax and semantics of programs are given in Chapter 2. Despite the simplicity of this example problem, the involved types of components find broad use. In fact, the program follows a common generate-define-test [37, 73, 79] programming pattern: it begins with a non-deterministic choice, continues with a definition of an auxiliary concept, and ends in a requirement. Using this program, an answer-set solver [5, 53, 104] can search for answers to the original problem from any given input graph. Indeed, the problem specification is the only instruction the programmer needs to provide to the solver because the ASP paradigm provides the language [29] for the specification and the methodology for the solver. The solver returns each found solution as an answer set. An answer set is a set of atoms representing a single solution. In this case, the relevant atoms would be of the form `picked_node(x)`, where `x` is a constant term identifying a node. From any single answer set $M$, one could then extract a solution, i.e., a node subset $U = \{x \mid \text{picked_node}(x) \in M\}$ that meets the original criteria. Hence, using an ASP solver, even a simple program like this can be useful in applications; indeed, it allows finding, for example, influential groups of individuals from a social network. This simplicity
exemplifies the convenience of ASP: the ease of expressing a problem in ASP can be close to the ease of expressing it in natural language.

The above-witnessed ease of modeling with ASP stems from the expressivity and practicality of the language primitives available. The examples of primitives thus far are a choice rule, two normal rules, and an integrity constraint involving a count aggregate. Aggregate refers to a numeric condition on a multiset that may involve counting, summation, or minimum finding, for example. In addition to these primitives, ASP features other aggregates as well as optimization statements [104]. This variety enables the concise expression of conditions that would otherwise require a non-trivial modeling effort. To understand this, consider the task of writing an aggregate-free alternative for the integrity constraint

\[-\#\text{count}\{U: \text{reachable}(U)\} < 100.\]

Such an encoding task is possible, a point demonstrated in this dissertation, but cumbersome from a modeling perspective. This difference in convenience between using rules with and without aggregates underlies the integral role aggregates have in practical problem encodings. This integrality is observable in many encodings submitted to the Answer Set Programming Competitions [30, 55, 56], as well as in research on introducing aggregates to databases [62], implementing aggregates for ASP [8, 51, 104], and studying aggregate semantics [45, 46, 61, 84].

The ASP paradigm supports optimization and does so in an elaboration tolerant [90] way. Elaboration tolerance is a desirable property of knowledge representation formalisms asking that small changes in any represented objects necessitate only small changes in the respective formal representations. In the context of ASP, such a change in a represented object could, for instance, be the change of a decision problem into an optimization problem. In optimization, the goal is to find solutions that minimize or maximize the value of some objective function instead of seeking solutions that only satisfy some given constraints. Optimization can be illustrated by extending the previous example program to an optimization program, where the goal is to find a set of ten nodes with a reachable closure of maximum size. Indeed, this can be done by replacing the constraint in the last line with the optimization statement:

\[\#\text{maximize}\{1, U: \text{reachable}(U)\}.\]

While this syntactic change is minor, the computational complexity of the obtained problem is significantly higher than before, which complicates the technical machinery required to solve it. Nevertheless, existing answer set solvers have such machinery built-in: they can perform Answer Set Optimization [7, 50]. Hence, this combination of a flexible modeling language and practical solver technology is the cornerstone of the elaboration tolerant ASP.

The main goal of this dissertation is to enhance the performance of ASP based tools for solving combinatorial search and optimization problems of the kinds
discussed above. While the performance of such tools has already received considerable attention in the field of ASP [5,53,104], many opportunities for further study exist. In particular, potentially performance-enhancing transformations of ASP programs have not been exhaustively studied. Therefore, such study is undertaken here; indeed, this dissertation develops transformation techniques to normalize and rewrite ASP encodings of problem instances, aiming to make the instances more efficiently solvable when possible. These transformations are developed and studied from both performance and correctness points of view, using both experimental and formal means. An additional goal of the dissertation is to contribute to the theory of verifying the correctness of ASP transformations via equivalence notions. In the following, the performance goals are motivated in general. Then, the main topics of this dissertation are overviewed: normalization and equivalence, lazy-normalization, and optimization rewriting. Afterward, Section 1.1 highlights the novel scientific contributions in this work, Section 1.2 presents related work, and, finally, Section 1.3 gives the structure of the remaining thesis.

Motivation

Improving the performance of ASP systems is valuable in itself, but also because it can strengthen the declarative nature of ASP. This is because, ideally, the main concern in declarative programming is the correctness of problem specifications. In the practice of ASP, however, different specifications of the same problem may yield vastly varying solving performance [52]. Therefore, performance issues may force an ASP programmer to understand and address the question of how a problem is solved when crafting a specification for it. This process may require significant expertise and dedication, which undermines the true declarativity of ASP. Indeed, the sensitivity of ASP to specification details is identified by Gebser and Schaub (2011) as one factor stalling ASP from gaining broader attention among users who are not experts in ASP. Hence, indeed, performance and robustness improvements to ASP solving methods are not only beneficial in their own right, but also as necessary steps towards higher levels of realized declarativity. Furthermore, the development of alternative solving approaches benefits the use of portfolio solvers [85], which combine several solvers possessing complementary solving characteristics into a single solver that is more robust than any of its constituents. Therefore the need to improve ASP solving performance is strong, especially concerning robustness.

Studying Normalization Techniques and Equivalence

Notions of ASP program equivalence are central in the ASP development and solving tasks. Indeed, both tasks often produce programs in various versions intended to be equivalent. This case applies to iterative program development as well as automated transformations typically performed in preprocessing. These
tasks generally pursue simpler programs easier to solve. A concrete example of an automated transformation is normalization, the process of compiling various rule types into normal rules [69]. Regarding the equivalence, its intended type depends on the context. To address these correctness considerations on a formal level, ASP literature includes studies on various equivalence notions for programs, including strong equivalence [80], relativized strong equivalence [112], and visible equivalence [66]. On a more practical level, ASP literature contains studies on the automated testing of these equivalence notions, i.e., of strong equivalence [96], relativized notions of equivalence [41], and visible equivalence [71]. Collectively, these development methods and studies indicate the central importance of equivalence checking in ASP.

While the mentioned process of normalization can benefit solver development, it also poses an obstacle for correctness analysis. To obtain an idea of the benefits, consider the following rule similar to the already seen constraint containing an aggregate:

\[
\text{s}\text{-}\text{count}(U: \text{reachable}(U)) < 3.
\]

Different normalization schemes can turn the rule into a set of normal rules. One such scheme produces the following simple and illustrative, although inefficient program:

\[
\text{r}\text{: reachable}(U_1, \text{reachable}(U_2), \text{reachable}(U_3), U_1 < U_2, U_2 < U_3).
\]

\[
\text{s}\text{-}\text{not}\text{r}.
\]

This program enumerates all ordered triples of relevant nodes and derives the indicator atom s in the absence of such triples. Already this simple program illustrates a key benefit of normalization: any tools used to process the output need to support only normal rules. Another benefit of normalization is that it may in fact improve subsequent solving performance as will be discussed in Chapter 3. Regarding the obstacles, one is that a proof of the correctness of a normalization should demonstrate identical behavior between the original and normalized rules in the context of any larger context program. In that sense, such a proof should demonstrate a strong relationship between the original and normalized rules. On the other hand, the proven relationship should be relaxed enough to allow helper atoms such as r. Discussion of normalization and equivalence continues in Chapter 3, which presents, moreover, a solution to the obstacles in the way.
Extending Lazy-Grounding with Aggregate Support via
Lazy-Normalization

The preceding example of normalization is in fact an example of first-order normalization with applications in an ASP solving approach called lazy-grounding. Lazy-grounding is an alternative to the solving approach ground-and-solve, which is currently predominant in ASP. Nevertheless, the ground-and-solve approach suffers from a problem called the grounding bottleneck that may occur in a preprocessing step built-in to ground-and-solve called grounding. Grounding implements a form of variable instantiation explained below together with an overview of ground-and-solve and lazy-grounding. The theme continues in Chapter 4 with a presentation of the first application of first-order normalization in the context of lazy-grounding.

Ground-and-solve works by first grounding a given program into a ground program that is simpler in form although larger in size and then solving that ground program. This grounding process substitutes first-order variables such as \( U \) and \( V \) that represent placeholders in the shown examples with constants that represent concrete values. The result is a ground program, or a propositional program, meaning that it is free of first-order variables. For example, suppose the normal rule \( \text{reachable}(U) :- \text{picked\_node}(U) \) was grounded in the context of a graph with nodes represented by the constants \( a \), \( b \), and \( c \). Then the atoms \( \text{reachable}(U) \) and \( \text{picked\_node}(U) \) would become variable-free in the grounding:

\[
\text{reachable}(a) :- \text{picked\_node}(a).
\]
\[
\text{reachable}(b) :- \text{picked\_node}(b).
\]
\[
\text{reachable}(c) :- \text{picked\_node}(c).
\]

Consequently, the program is Booleanized so that the only remaining unknowns in the program are the Boolean truth values of the variable-free atoms. The variable-free quality of ground programs simplifies their analysis and efficient solving. It is likely that this relative simplicity of solving has led to the predominance of ground-and-solve, reflected in the lack of alternative approaches submitted to the recent ASP Competitions (ASPCOMP) [30,55,56]. Likewise, most of the work in this dissertation follows the ground-and-solve approach.

Even so, ground-and-solve is inherently susceptible to the grounding bottleneck: for many classes of programs, grounding produces prohibitively large programs whose explicit construction is out of question. By extension, the grounding bottleneck prohibits the use of state-of-the-art ground-and-solve methods on many classes of programs.

In order for ASP solving to scale beyond the grounding bottleneck, fundamentally different solving approaches are required, and one such approach showing promise is that of lazy grounding. Lazy grounding is a solving approach
that interleaves grounding and solving steps as opposed to performing up-front
grounding. In each grounding step, only those ground rules are generated that
have been deemed relevant during the search. This way the generation of
ground rules is postponed as late as conveniently possible, and many ground
rules are avoided altogether. Such careful generation helps keep the grounding
size manageable in many cases where an up-front grounded program would be
prohibitively large. While this reduction improves ASP scalability in the face
of the grounding bottleneck, it also increases the challenge and complexity of
developing lazy-grounding.

This dissertation contributes to the challenging development of lazy-grounding
by bringing aggregate support to lazy-grounding for the first time. Indeed, before
this contribution, aggregate support was missing from existing lazy grounding
systems for ASP, which include GASP [97], ASPεRIX [76], OMIGA [35], and
ALPHA [110]. The aggregate support was added to ALPHA via a novel fram-
ework of first-order lazy-normalization that is presented in depth in Chapter 4
based on Publication III. In the framework, normalization is performed before
grounding, on the first-order level, which is in contrast to traditional normal-
ization performed after grounding [69], on the propositional level. In doing so,
the framework applies the transformation based approach of first-order nor-
malization. Such normalization is in fact exemplified by the previously shown
example on normalization, where the input and output programs involve first-
order variables formatted as capital letters. However, this particular example
of normalization is inefficient since it corresponds to a number of ground rules
exponential in the bound of the rule. Moreover, fully overcoming such ineffi-
ciency is challenging on the first-order level when the intention is to apply lazy
grounding on the normalized program. A complicating challenge in this context
is that the concrete number and identities of the inputs inside the aggregate,
represented in the example by the set \{U : \text{reachable}(U)\}, are not attainable
without sacrificing laziness. This challenge led to an innovation that enabled
efficient first-order normalization that is uniquely tailored for the challenging
context of lazy-grounding.

**Increasing Optimization Performance via Optimization Rewriting**

Performance is particularly important in ASP optimization. This importance
stems in part from the elevated complexity of searching specifically for optimal
answer sets as opposed to any answer sets. Accordingly, optimization requires
significant algorithmic effort, which is generally spent solving a sequence of
decision problems [50]. This algorithmic approach governs state-of-the-art ASP
optimization [7, 50], which is largely focused around two general strategies:
model-guided and core-guided solving [94]. These strategies complement the
performance of one another in that they excel in different contexts, as observed
for example in the work of Gebser et al. (2015) as well as in Section 5.5. The
magnitude of this difference invites further study of ASP optimization algorithms
with the goal of combining some of the positive qualities of both approaches. Indeed, this opportunity to increase optimization performance has been fruitfully exploited in Chapter 5 as well as in other ways in recent related work [4, 50].

Optimization performance is approached in this work by designing an innovative class of transformations: \textit{optimization rewritings}. These are transformations carried out in a preprocessing stage analogously to normalization discussed earlier, but with the sole goal of making the instances easier to solve. Like normalization, optimization rewritings are orthogonal in their applicability. That is, they may be combined flexibly with different subsequent solving strategies in pursuit of the best possible performance. Similarities with normalization continue in the design of specific schemes for optimization rewriting, which draw inspiration from the former. However, optimization rewriting allows a greater degree of freedom in this design process. Enabled by this freedom, this work explores several parameterized schemes for optimization rewriting that have no parallel in the context of normalization. This added freedom is revealed to be a key factor in bringing out positive performance results in experiments reported in Chapter 5. Some of these positive results are found in scenarios complementary to those on which existing methods excel. Being highly complementary, the improvements translate to greater robustness of state-of-the-art ASP solving systems taken as a whole. Such robustness is particularly significant in that it carries over to portfolio solving systems that combine several other solving systems into one [85]. In addition to these experiments, the proposed rewritings are analyzed formally to illuminate how optimization rewriting can bring the advertised performance improvements. These experimental and formal analyses of optimization rewriting also contribute to the general understanding of optimization performance built by the ASP community, hopefully facilitating further developments in the future.

1.1 Contributions

The key contributions of this dissertation are summarized in the following and in more depth in Chapters 3–5. Moreover, Figure 1.1 illustrates the connections between these contributions and Publications I–V from which they originate.

- In Publication I, the ASP normalization tool \texttt{LP2NORMAL} is described. The main function of the tool, developed by the current author, is to \textit{normalize} extended rules and particularly those involving aggregates away into normal rules. To this end, the tool implements numerous novel and existing translations from the literature [1, 11–13, 19, 40, 63, 69, 104, 105] and Publications VI and VII as well as combinations of thereof. The original goal of the tool was to implement extended rule support for normal-rule solvers. In this role, the tool is successfully used as a preprocessing

\footnote{Available at http://research.ics.aalto.fi/software/asp.}
component of solving pipelines that compile ASP to SAT prior to solving, for example in the Sixth Answer Set Programming Competition \cite{56}. Another motivation has been to aim for better solving performance benchmark-by-benchmark, by using the tool as a selectively-normalizing and rewriting preprocessor of other ASP solving systems. In this role, the tool carries out preprocessing on optimization criteria as well as extended rules in a pipeline combining LP2NORMAL with the state-of-the-art solver CLASP as a back-end solver in the mentioned competition. The preprocessing proved successful in pushing the performance of CLASP forward, as the pipeline won the optimization track of the competition and attained very good overall behavior in other sub-tracks as well \cite{56}.

- In Publication II, the challenges of comparing programs and verifying the correctness of program transformations such as those from Publication I are tackled. In this context, several practically and formally meaningful contributions are made. First, a novel notion of visible strong equivalence (VSE) is proposed as a generalization of the existing relations of visible equivalence \cite{66} and strong equivalence \cite{80}. The generalization relates together programs that yield visibly identical multisets of answer sets as printed out by standard answer set solvers when combined with a common context program. This manner of simultaneous consideration of visibility, multisets, and context programs importantly distinguishes the relation
Introduction

from prior work \cite{43,101,112,113}. As further detailed in the publication, visible strong equivalence is therefore uniquely well suited to the task of formalizing the correctness of program transformations intended to preserve answer sets as seen by the user. Second, several results are developed in order to facilitate correctness proofs that establish visible strong equivalence. Namely, model based characterization results are given so that the semantics of programs, from the point of view of VSE, can be captured in terms of so-called VSE-models. Importantly, sets of VSE-models can be compared to establish visible strong equivalence. A characterization of this nature is given for propositional programs in general, and a simplified version is tailored for positive programs. Moreover, a proof technique is developed to allow careful use of the simplified version for a class of nearly positive programs. Third, the characterization results are exploited in creating an automated verification tool that checks visible strong equivalence for a large, practically relevant class of programs that is free of hidden non-determinism. Finally, the verification tool is used to verify the correctness of a large number of normalizations produced by the normalization tool LP2NORMAL described in Publication I as well as another normalization tool, PBTRANSLATE \footnote{Available at \url{https://github.com/jbomanson/pbtranslate}.}. The performance of the verification process is studied at the same time, and certain practically actionable insights into the efficient verification of program transformations are gained.

• In Publication III, the state-of-the-art of the lazy-grounding approach to ASP solving is extended in its applicability. As a remedy to the fact that prior incarnations of the approach have been limited to normal rules only, the publication introduces a framework of lazy normalization as a measure to implement support for extended rules and aggregates. Straightforward normalization is obstructed by challenges inherent to the lazy-grounding approach, due to which first-order normalization is called for and full enumeration of the ground instances of any predicate must not be relied upon. Hence, the direct application of normalization techniques from the ground level, such as those from Publication I are not feasible. To overcome these challenges, the reported work extends the lazy-grounding ASP solver ALPHA with a novel enum predicate for the lazy enumeration of terms and developing carefully optimized lazy normalizations that exploit the predicate. The unprecedented combination of lazy-grounding and aggregate support is demonstrated to scale to new application areas in benchmarks that extend beyond the reach of traditional non-lazy approaches.

• In Publication IV, the theme of normalization is evolved into a novel type of program transformations called optimization rewritings. These rewritings tackle the problem of finding optimal answer sets efficiently by transforming the representation of optimization criteria. For this task, the
transformations re-purpose parts of known normalization techniques. Like such normalizations, the obtained rewritings introduce added structure in the form of auxiliary atoms that may accelerate standard ASP solvers in solving the instances. In experimental evaluations presented in the paper, optimization rewriting is shown to bring significant improvements to the performance of an optimizing answer set solver on a number of benchmark problems. Moreover, the publication presents a further refined technique of selective optimization rewriting, which takes the form of a framework that flexibly allows fine tuning the focus and amount of rewriting. Among the experimented rewriting techniques, these selective ones yield the strongest results.

- In Publication V, novel optimization rewriting techniques are developed along with the concept of weighted comparator networks. Weighted comparator networks lead to arguably simpler alternatives to the rewriting techniques presented in Publication IV. Despite this simplicity, the techniques yield competitive performance benefits in experiments, while being easier to implement and to control for output size. As a further contribution, the reasons behind the performance improvements are analyzed formally in the context of an abstract solver model and an illustrative class of programs called binomial optimization programs. Specifically, the analysis focuses on the number of no-goods [53] produced in a branch-and-bound ASP optimization process that are necessary in the final unsatisfiability check stage. As a result of the analysis, optimization rewriting is proven to have the potential to reduce the work of a solver due to such nogoods by an exponential factor. Moreover, experiments with an actual solver confirm tremendous work reductions in practice as well. While the technique is demonstrated on answer set optimization in the article, and also implemented in the ASP preprocessor called PBTRANSLATE, the technique is readily applicable to neighboring optimization paradigms also such as Maximum Satisfiability (MaxSAT) and pseudo-Boolean optimization (PBO).

1.2 Related Work

This section discusses other work related to this dissertation. To this end, related work by the current author is summarized in Section 1.2.1. Then, related solving paradigms are discussed in Section 1.2.2, the history of aggregates in Section 1.2.3, and the use of translation techniques outside of standard preprocessing settings in Section 1.2.4. Finally, lazy-grounding approaches are discussed in general in Section 1.2.5, since there are no prior lazy-grounding solutions supporting aggregates.
1.2.1 Relation to Other Publications by the Author

The relation between this dissertation and Publications VI–VIII is given in the following. Although these publications are related to this dissertation, they are not included in it.

Prior to this work, the current author took part in investigating the performance impact of novel cardinality rule normalizations in Publication VI and weight rule normalizations in Publication VII. Both publications adapted and improved sorting network based circuit designs [19] for use in ASP normalizations. In doing so, the publications paved the way to this work, which relates to the publications in several ways. First, Chapter 3 includes a discussion of these normalization techniques. Moreover, the chapter shows proof techniques for program transformations such as normalizations. Second, the novel class of transformations called optimization rewritings introduced in Publications IV and V and discussed in Chapter 5 is heavily inspired by normalization and specifically its performance potential. Indeed, the chapter completes a “trilogy” of research efforts on transformations concerning three essential language constructs in ASP: cardinality rules and weight rules in the prior publications and, finally, optimization statements in Chapter 5. Interestingly, the experimentally assessed performance of the techniques in this trilogy turned out to differ; specifically, the techniques for optimization statements had a vastly more positive performance impact than the prior techniques for extended rules. Finally, the theme of first-order normalization explored in Chapter 4 generalizes the concept of normalization of extended rules from the propositional level.

Publication VIII relates to this work in that it presents a novel transformation based technique for ASP solving. To enable the technique, the publication first develops the regular ASP language into ASP modulo Acyclicity by introducing a new modeling construct. This construct allows a user to easily model acyclicity constraints on graphs in such a way that a solver can use efficient techniques for checking the constraints. Acyclicity is a graph property held by graphs without cycles. A user of the extension can conveniently impose acyclicity constraints on any graphs defined by usual ASP rules when using the state-of-the-art solver CLASP [49], for example, which now incorporates the extension as a built-in feature. By using the new acyclicity constraint, the publication presents novel transformations that map programs from ASP to ASP modulo Acyclicity. While such transformations can be obtained trivially by performing no modifications, the presented transformations instead modify programs nontrivially and augment them with specific acyclicity constraints. With the help of these constraints, the modifications implement the checking of a stability condition inherent in the definition of answer sets, which is given formally in Section 2.2. The stability condition requires that the atoms in an answer set can be derived one-by-one using program rules and any already derived atoms in the manner of a deduction system. To enforce this stability condition, the acyclicity approach translates the relevant structure of a program into a graph encoding and then determines the
stability of an answer set by searching for an acyclic derivation for each of its atoms from that graph. With this acyclicity based implementation of stability in place, the resulting program can be solved via ASP modulo Acyclicity solvers that do not necessarily use other techniques to enforce stability. These other, more traditional techniques include so-called unfounded-set checking \cite{109} and loop nogood generation used in ground-and-solve, and computation sequences \cite{83} used in lazy-grounding.

1.2.2 Solving Paradigms

This section gives an overview on solving paradigms related to ASP.

The well-known **Boolean Satisfiability (SAT)** problem concerns the search for truth assignments that satisfy a propositional Boolean formula \cite{22}. Such an input formula is typically either given as or converted into **conjunctive normal form (CNF)**, which formally consists of a set of clauses \( \{l_1, \ldots, l_n\} \) that are sets of literals \( l_i \). A CNF is satisfied if all its clauses are satisfied, and a clause is satisfied if one or more of its literals is satisfied. The SAT problem has received significant research focus over decades, and this has resulted in increasingly efficient solver implementations compared in regularly held SAT Competitions \cite{15}. There is a strong relation between ASP and SAT on the ground level, and ASP essentially generalizes SAT. Namely, SAT instances can be translated with the help of auxiliary atoms into ground normal rules using a modular translation of linear size that can be carried out clause-by-clause \cite{95}. In the other direction, from ground normal rules to SAT, there is a translation of size \( O(n \log n) \) that introduces auxiliary atoms \cite{66,69}, but no modular translation is possible \cite{95}.

In the **pseudo-Boolean (PB)** satisfaction problem, a set of linear pseudo-Boolean constraints must be satisfied. These constraints are inequalities of the form \( w_1 l_1 + \ldots + w_n l_n \geq b \) between integer bounds \( b \) and pseudo-Boolean expressions \( w_1 l_1 + \ldots + w_n l_n \) with integer coefficients \( w_1, \ldots, w_n \) and Boolean variables \( l_1, \ldots, l_n \). The PB problem straightforwardly generalizes SAT: a clause \( \{l_1, \ldots, l_n\} \) is expressible as the PB constraint \( l_1 + \ldots + l_n \geq 1 \). Again, on the ground-level, ASP and PB are related in a similar fashion as ASP and SAT: PB straightforwardly translates linearly and modularly into ASP but ASP translates only non-modularly into PB. Indeed, since SAT can be translated modularly into PB, the negative modularity results between ASP and SAT from \cite{69} imply that no modular translation from ASP to PB may exist either. The **Mixed Integer Programming (MIP)** problem concerning linear inequalities over real and integer variables is expressive enough, however, to allow linear translations from ASP \cite{82}.

The **SAT modulo Theories (SMT)** concept refers to extensions of SAT that allow additional quantifier-free formulas subject to one or more specified theories, such as theories of integers, reals, sets, tuples, arrays, and bit vectors \cite{18}. The study of SMT is motivated by the fact that it is prohibitive to encode many application problems in plain SAT or similar formalisms due to size considerations. In fact,
this is the same motivation as is behind lazy-grounding in ASP, addressed in Chapter 4. Analogous extensions are under research in the ASP field as well, and have given rise to ASP modulo Theories (ASPmT) [49, 68].

Each of these search problem formalisms generalizes to a respective optimization variant. Namely, SAT generalizes to (partial weighted) MaxSAT, PB and MIP are already general enough to support optimization, and SMT generalizes to MaxSMT. Moreover, PB optimization coincides with 0-1 integer linear optimization (0-1 ILP), which is generalized by MIP. In each of these formalisms except MIP, the optimization problem is to find a solution that minimizes a given pseudo-Boolean expression, just as in answer set optimization. In MIP, the optimized objective function can be any linear function. Finally, MaxSMT is further generalized by Optimization Modulo Theories (OMT), which also allows for optimization criteria beyond pseudo-Boolean objective functions. Hence, these results together give strong support in favor of the idea of optimization rewriting.

1.2.3 Aggregates

This section outlines the history of aggregates in ASP and the basic implementation approaches considered in the literature. In this context, an aggregate refers to numeric constraints or conditions involving sets [84].

Aggregates have been used earlier in databases and DATALOG [62], where operations such as summation are captured by special predicates allowing for the expression of linear constraints with rules. However, the used approach is specific to DATALOG and nontrivial to transfer to ASP. The history of aggregates within ASP begins with choice rules, cardinality rules, and weight rules [104]. The latter two involve count and sum aggregates in their bodies, and these primitives, along with optimization statements, remain to be central primitives for modeling in ASP [57].

The study of aggregates in ASP has led to the identification of monotone, anti-monotone, and convex aggregates [84]. Examples of monotone aggregates include count and sum aggregates with only lower bounds, positive weights, and positive literals. The use of only an upper bound would give an example of an anti-monotone aggregate, and the use of both lower and upper bounds, an example of a convex aggregate. Since in general an aggregate is satisfied by many different interpretations of its atoms, its use in the head of a rule raises non-trivial semantic questions that also deal with potentially increased computational complexity. This challenge has been met with several proposals; cf. [45, 46, 61]. More recently, non-monotone aggregates have been rewritten into monotone ones accompanied by disjunctive rules [8], indicating that the computational complexity hidden in non-monotone aggregates can be isolated.

Concerning the implementation of language extensions such as aggregates, there are two basic implementation strategies: the native strategy [51], where the underlying solver is extended to support them internally, and the translation
based strategy \cite{69, 81}, where the extensions are compiled away into other language constructs supported by the solver.

The native approach is pursued by solvers such as CLASP \cite{53}, WASP \cite{6}, and historically SMODELS \cite{104}. The modern approach to this end is based on propagators, which monitor for impending conflicts between an aggregate and the currently explored part of the search space. Upon any conflicts, the propagators inform the solver of the reasons behind the conflicts in terms of nogoods \cite{51}. The translation based strategy on the other hand is realized in the form of normalization \cite{69}, in which extended rules are compiled specifically into normal rules. This topic and related work is discussed in more depth in Chapters 3 and 4, which deal with normalization on the ground and first-order levels, respectively. In brief, the associated tools on the ground-level are the preprocessing tools LP2NORMAL and PBTRANSLATE, and on the first-order level the lazy-grounding answer-set solver ALPHA. In other work, normalization appears as an optional feature in CLASP \cite{53}, which implements simpler normalization schemes than the above tools, and on the first-order level in the SeaLion Eclipse plugin \cite{100} for the purpose of debugging answer set programs. Normalization was first performed on the first-order level by Polleres et al. \cite{100}, but the normalization schemes therein consume more space than the ones described in this thesis. Mixing the two types of approaches is also possible, and in fact both LP2NORMAL and CLASP support a hybrid approach where heuristically chosen aggregates are translated away and the rest are left for native handling.

1.2.4 Translation Techniques in Non-Standard Settings

This section surveys adaptations of translation techniques outside of what could be considered standard preprocessing settings. Namely, translations in computational logic typically reduce problem instances in a preprocessing step from one formalism to a simpler one for further processing. This is the case, for example, when compiling various types of constraints into SAT \cite{40, 64}. As discussed below, such translation techniques have found various applications also within the solving process and also for transformation use cases with an emphasis on matters beyond the simplification of the input. Due to re-purposing translation techniques, these are related to the concept of optimization rewriting, to be discussed in Chapter 5.

The conflict-directed lazy decomposition (CDLD) approach incorporates PB-to-SAT translation techniques into a PB solver that also has native propagator based PB solving capabilities \cite{3}. The solver refines specific translations to be incremental and to work together with the native techniques so that it is able to begin with applying the native techniques and to gradually transition to using the translations during search. The benefit of the approach is robustness: the translation and native solving strategies have complementary performance characteristics, and by mixing the approaches the solver is able to dynamically make use of the right approach for the right problem \cite{3}. The approach is related to
the optimization rewriting techniques presented in Chapter 5 despite significant differences: (i) both aim to limit translation size: CDLD dynamically and this work statically, (ii) both make use of sorting networks: CDLD incrementally in the unweighted case and this work statically in the unweighted and weighted cases, (iii) both rely on solvers with native support: CDLD on support for PB constraints and this work on support for optimization. Setting aside the difference in context between unweighted and weighted cases, as well as decision and optimization problems, the optimization rewriting techniques from this work are positively distinguished by their virtue of being static preprocessing techniques that can be flexibly combined with different solving strategies. Moreover, it is in principle possible to incorporate weighted comparator networks from this work into CDLD, but investigating the resulting technique is left for possible future work.

In a closely related approach [2], PB-to-SAT translations are generated in full once deemed important during search. This relatively simpler approach avoiding incremental translation has yielded performance competitive with CDLD in experiments [2]. The incorporation of weighted comparator networks into such PB-to-SAT translations would be possible as well. Similarly, in the context of optimization, the optimizing SMT solver VZ adapts the above techniques for use in its pseudo-Boolean theory solving component, which incrementally generates small sorting networks [23]. It is not mentioned therein, however, if and how the generation has been applied to optimized expressions in addition to constraints.

Also connected with this theme of hybrid translations is the work of [21], where a cardinality constraint is translated into a hybrid SAT encoding consisting of two redundant translations. The idea is to make one of the translations as concise as possible and the other as supportive of unit-propagation as possible. Due to the redundancy, the second translation can be incomplete and hence relatively concise as well. Moreover, the redundancy of these translations is reminiscent of the redundancy in combining translations and native solving. Indeed, it is motivated by a similar goal: keeping translation size small while retaining the benefits of it.

The work in [103] independently investigates techniques coinciding with some of the optimization rewriting techniques from Publication IV, but in the context of MaxSMT and OMT. In more detail, the work studies techniques that encode sorters for groups of equal-weight literals being optimized. Therefore, the techniques are closely analogous to the optimization rewritings for uniform weights discussed in Section 5.1. Just as the techniques yield improvements in the computational experiments in Publication IV concerning ASP, they yield improvements in the empirical evaluations in [103] concerning MaxSMT and OMT.

Sorting networks, which are standard components in ASP normalizations and SAT encodings of pseudo-Boolean constraints, are also used in *(unsatisfiable)* core-guided optimization strategy for MaxSAT [93]. The strategy approaches solutions from unfeasible parts of the search space by targeting successively
relaxed problem variants until finding a solution. Until that, the reasons for unsatisfiability are captured as unsatisfiable cores, which can be expressed as groups of similar cardinality constraints and hence also using sorting networks [93]. This is different from the model-guided optimization strategy, where branch-and-bound algorithms are used to find models of successively more constrained problem variants until no solution can be found anymore [7]. The above approach for using sorting networks in MaxSAT differs significantly on the technical level from the comparator network based approach for ASP optimization presented in Chapter 5. These approaches apply generally different networks to different literals at different times, and the techniques from this work apply in particular to the model-guided approach, which is vastly different from the core-guided one. Moreover, the experimental results therein indicate that the methods are strongly complementary: they yield strong results on different problems.

1.2.5 Lazy-Grounding Techniques

This section gives a brief overview on lazy-grounding answer-set programming and the related work surrounding it, in order to give context to Chapter 4 on first-order lazy-normalization, and to draw a contrast with the rest of this thesis, which fits the ground-and-solve paradigm.

As briefly discussed in the introduction, the grounding process inherent in the ground-and-solve strategy generally increases the problem description polynomially, and in the worst case even exponentially. The increase is heavily problem-specific, and for many problems it is not an issue, but for some problems it is entirely prohibitive and it is said to form a grounding bottleneck. This has motivated an alternative line of research on lazy-grounding, which aims to circumvent the grounding bottleneck, and which is the topic of Chapter 4. The primary idea behind lazy-grounding is to interleave ground and solve steps, so as to generate only those ground rules that are deemed relevant during the search. The key benefit is that the ground rules that do not contribute to the formation of any answer set of interest need not be generated at all. Moreover, whenever only a single answer set is sought, which is often the case, this benefit is magnified. On the other hand, lazy grounding poses various research challenges of its own, surveyed by Weinzierl et al. (2019) [111]. This is expected, given that it attempts to combine the already demanding grounding and solving steps closely together. An example of these challenges is that while the avoidance of grounding rules protects the solver component from being over burdened by massive numbers of ground rules, it also restricts the view of the solver to the instance. This impedes typical decision heuristics used on the ground level from working efficiently directly in lazy-grounding [107].

Research on lazy-grounding answer-set programming has progressed in tandem with several solver implementations, which include GASP [97], ASPERiX [76], OMIGA [35], and ALPHA [110]. These existing systems all rely on com-
The idea behind this notion is that the stability of an answer set can be guaranteed if its atoms are derived in a non-deterministically chosen order that complies with the dependencies between the positive atoms of the rules involved in the derivation. This notion is suitable as a back-bone for lazy-grounding, since a search in such an order depends only on a well-defined set of rules, which the grounder can determine and ground on demand. The detection of rules that may fire at a given point in the search space, however, is the NP-complete problem of lazy grounding a rule, which the different solvers handle in different ways: GASP by encoding it as a constraint problem, ASPERiX with semi-naive grounding, and OMIGA with the use of RETE-networks [48], of which semi-naive grounding is reportedly fast enough [76] despite the sophistication of the competing techniques. Moreover, the derivation order and firing of rules are subject to non-deterministic decisions, which greatly affect search performance. The study of efficient heuristics that are applicable in this context is ongoing [107], and challenged by restrictions imposed by lazy-grounding as briefly mentioned.

The recent work on lazy-grounding also includes Publication III, the topic of Chapter 4, which brings support for a range of aggregates to the ALPHA solver. Indeed, existing lazy-grounding ASP solvers supported only normal rules prior to this advancement. In this case as well, the pursuit of efficiency in the context of lazy grounding necessitates innovations beyond those on the ground level, as will be discussed further in Chapter 4.

The grounding bottleneck is addressed to various degrees by other lines of research as well. For instance, Cuteri et. al [34] investigate the performance impact of separating constraints from the rest of the program and instantiating them lazily. Furthermore, CLINGO [49] has evolved into an ASP modulo Theories (ASPmT) solver, allowing for parts of an ASP program to be modeled in a theory defined by a user given procedural propagator implemented in, e.g., python or C. Naturally, program parts that are prone to blowing up in size are obvious candidates to being implemented with a propagator, which may or may not be feasible. Moreover, the Lazy-MX [32] system for the FO(ID) language successfully implements goal-driven lazy grounding for its language, which closely corresponds to the generate-define-test fragment of ASP. Finally, grounding has been entirely avoided by some systems that aim at query answering. This has been realized by evaluating programs of normal rules in a top-down Prolog-like manner [87], both on the ground level in GALLIWASP [88] and on the first-order level in S(ASP) [10].

1.3 Dissertation Structure

The rest of this dissertation consists of an overview organized into Chapters 2–6 as described in the following and then of Publications I–V. Chapter 2 presents the common preliminaries for the rest of the thesis. Chapter 3 concerns the
equivalence of programs in nontrivial settings that demand attention to the visibility of atoms. In order to motivate the considerations, the chapter surveys normalization techniques from Publication I, which make generous use of visibility via hiding atoms. Normalization is used here to implement support for extended rules and aggregates. Afterward, the chapter presents the notion of visible strong equivalence and summarizes the key formal and experimental results from Publication II. Characterization results included among these show how to capture the visibility-aware semantics of programs in ways that facilitate correctness proofs of program transformations. Chapter 4 instead focuses on program transformations on the first-order level and in the context of lazy grounding. The purpose is to enhance the state-of-the-art of lazy-grounding in ASP by extending it with aggregate support. The transformations achieve this by implementing first-order normalization. The chapter reports the results of Publication III. Chapter 5 evolves the theme of normalization of extended rules and aggregates into optimization rewriting of optimization statements. These transformations are applied to the problem of finding optimal answer sets efficiently by modifying the expression of optimization criteria in a manner inspired by normalization. The techniques are summarized together with formal correctness results and experimental performance results from Publication IV and Publication V. Chapter 6 closes this dissertation with a discussion of the significance of the presented results followed by an account of potential future work suggested by the results.
2. Preliminaries

This chapter presents central preliminaries for the rest of the thesis. First, ground answer set programs are defined, with their syntax given in Section 2.1 and semantics in Section 2.2. Then, first order programs are discussed in Section 2.3. Continuing with the semantic side, basic notions of program equivalence are reviewed in Section 2.4. Then, the classes of merging and sorting programs, which are important in normalization, are given in Section 2.5. Finally, the related concepts of comparator and sorting networks are described in Section 2.6, and their connection to ASP as established in Section 2.7.

2.1 ASP Syntax

This section describes the syntax and intuitive meaning of the class of (ground) answer set programs considered in this thesis. The description also covers relevant solving related syntax and nogoods [53].

Below are the definitions of the ASP rules and statements expressible in the forms (2.1–2.6) of ASP-Core-2 syntax [29]. The used symbols $a, a_1, \ldots, a_n$, $b_1, \ldots, b_k, c_1, \ldots, c_m$ stand for propositional atoms, which are the ASP counterpart of Boolean variables. In the forms (2.1–2.6), atoms $a$ and $a_1, \ldots, a_n$ signify head atoms and the rest body atoms. Atoms are also called positive literals and (default) negated atoms such as $\text{not } c_1, \ldots, \text{not } c_m$ are also called negative literals. The symbols $l_1, \ldots, l_n$ denote literals of either type. The term default refers to the characteristic property of ASP by which the failure to prove an atom justifies the derivation of its negation. Indeed, a positive literal can be intuitively read as the condition that a specific atom can be inferred using the rules of the program, whereas a negative literal reads as the assumption that the atom can not be inferred.

The most fundamental rule type in ASP is the normal rule (2.1), which roughly states that its head atom $a$ is inferable once its body fires, which occurs when all of its body literals have been inferred via other rules in the program.

\[
a : -b_1, \ldots, b_k, \text{not } c_1, \ldots, \text{not } c_m.
\] (2.1)
A choice rule (2.2) indicates that any subset of its head atoms $a_1, \ldots, a_n$ can optionally be inferred once its body fires.

$\{a_1, \ldots, a_n\} :- l_1, \ldots, l_n.$  \hspace{1cm} (2.2)

A cardinality rule (2.3) states that its head atom $a$ is inferable whenever a given number $k$ or more of its body literals $l_1, \ldots, l_n$ have been inferred. Intuitively, the body condition requires the pseudo-Boolean expression $l_1 + \cdots + l_n$ represented by the count aggregate $\#\text{count}(1: l_1; \ldots; n: l_n)$ to evaluate to at least $k$. Cardinality rules generalize normal rules (2.1) as can be seen when setting $k = n$.

$a :- \#\text{count}(1: l_1; \ldots; n: l_n) \geq k.$  \hspace{1cm} (2.3)

A weight rule (2.4) specifies that its head atom $a$ can be inferred if a given value $k$ matches or exceeds the sum of the inferred body literals among $l_1, \ldots, l_n$ weighted by their respective body weights $w_1, \ldots, w_n$. That is, the rule intuitively triggers once the pseudo-Boolean expression $w_1 l_1 + \cdots + w_n l_n$ represented by the sum aggregate $\#\text{sum}(w_1, 1: l_1; \ldots; w_n, n: l_n)$ evaluates to at least $k$. It naturally generalizes a cardinality rule, which can be obtained by setting $w_1 = \cdots = w_n = 1$.

$a :- \#\text{sum}(w_1, 1: l_1; \ldots; w_n, n: l_n) \geq k.$  \hspace{1cm} (2.4)

Both a set of weak constraints (2.5) and an optimization statement (2.6) call for minimization of the sum of the inferred literals among $l_1, \ldots, l_n$ weighted by the weights $w_1, \ldots, w_n$.

$\vdash l_1. [w_1, 1] \ldots \vdash l_n. [w_n, n]$  \hspace{1cm} (2.5)

$\#\text{minimize}(w_1, 1: l_1; \ldots; w_n, n: l_n).$  \hspace{1cm} (2.6)

A disjunctive rule (2.7) differs from normal rules in having a disjunctive head of atoms $a_1, \ldots, a_h$, of which at least one is minimally derived when the body of the rule is satisfied.

$a_1| \ldots | a_h :- l_1, \ldots, l_n.$  \hspace{1cm} (2.7)

The formal analysis of ASP search procedures relies in this thesis on the concepts of (partial) assignments $A$ and nogoods $\delta$ [53]. Syntactically, both are sets of signed literals $\sigma = T x$ or $\sigma = F x$ that assign atoms $x$ to true or false, respectively. They can be viewed as three-valued interpretations that generally assign only a subset of atoms, leaving some atoms undefined. Their roles differ
Preliminaries

in that assignments reflect potential search paths for a solver, whereas nogoods \( \delta \) demand for rejection of any assignments \( A \supseteq \delta \) that include them as subsets. These rejected assignments \( A \supseteq \delta \) are also said to conflict with \( \delta \). In this work, nogoods help model constraints \( \Gamma \), which are here defined as sets of nogoods. In order for a constraint \( \Gamma \) to be satisfied, all nogoods \( \delta \in \Gamma \) must be satisfied by the assignment \( A \) in question. Accordingly, an assignment is said to conflict with a constraint \( \Gamma \) if \( A \) conflicts with any nogood \( \delta \in \Gamma \).

A disjunctive (ground answer-set) program \( P \) is a set of normal, choice, cardinality, weight, and disjunctive rules of the forms (2.1–2.4) and (2.7); and, additionally, nogoods in the context of Chapter 5 on optimization. In Chapter 5, all program atoms are assumed to appear either as the heads of some rules or in nogoods. A program without disjunctive rules (2.7) is a (non-disjunctive) program. An optimization program \( O \) is a pair \( \langle P, e \rangle \) consisting of a program \( P \) and a pseudo-Boolean expression \( e \) as an objective function that can be written in either of the forms (2.5–2.6). A program of any of these types is called positive if it is free of choice rules, free of nogoods, and its rule bodies are free of default negation not. A rule \( r \) is positive if the program \( \{r\} \) is positive. In the sequel, each atom, literal, signed literal, rule, statement, and program is associated with a context-specific set of atoms called its signature \( \text{At}(\cdot) \). Signatures follow two rules: (1) the signature of an atom must contain the atom itself, and (2) the signature of every component must be a superset of the signatures of its components. Moreover, the set of all head atoms in a program \( P \) is denoted by head(\( P \)).

2.2 ASP Semantics

This section defines the answer set semantics of programs, as well as related formal prerequisites.

An interpretation \( I \subseteq \text{At}(P) \) of a program \( P \) corresponds to an assignment that is total, meaning that it defines a truth value for every atom \( a \in \text{At}(P) \). An interpretation \( I \) is represented by a set of atoms that includes atoms \( a \in I \) regarded as true and excludes atoms \( a \not\in I \) regarded as false. An interpretation \( I \subseteq \text{At}(P) \) is said to satisfy an atom \( a \) and a signed literal \( Ta \) if \( a \in I \), a negated atom \( \text{not} \ a \) and a signed literal \( Fa \) if \( a \not\in I \), a nogood \( \delta \) if it does not satisfy all \( \sigma \in \delta \), a normal rule \( r \) if it satisfies the nogood \( \{Fa, Tb_1, \ldots, Tb_k, Fc_1, \ldots, Fc_m\} \), a choice rule unconditionally, and a disjunctive rule if it satisfies the nogood \( \{Fa_1, \ldots, Fa_h, Tb_1, \ldots, Tb_k, Fc_1, \ldots, Fc_m\} \). Moreover, an interpretation \( I \subseteq \text{At}(P) \) satisfies a cardinality rule or a weight rule if it satisfies the head atom \( a \) or a subset of the literals \( l_1, \ldots, l_n \) that add up to \( k \) literals or \( k \) units of weight, respectively. Finally, \( I \subseteq \text{At}(P) \) satisfies a (disjunctive) program \( P \) if it satisfies all rules and nogoods in \( P \). Such satisfying interpretations constitute the set \( \text{CM}(P) \) of (classical) models of \( P \). The operator \( \models \) is used to indicate satisfaction as usual.
The reduct $P^I$ of a program $P$ is obtained from $P$ by (i) removing satisfied negative literals from rule bodies and unsatisfied atoms from choice rule heads, (ii) decrementing cardinality rule bounds for each literal removed in the first step, (iii) decrementing weight rule bounds by any weights removed in the first step, (iv) turning choice rules with remaining head atoms $a_1, \ldots, a_n$ into sets of $n$ normal rules with the respective head atoms, and (v) removing any remaining non-positive rules.

Given a family $F$ of sets, $\text{Min}(F)$ denotes the subset minimal elements $S \in F$, i.e., the elements $S \in F$ such that $(2^S \setminus \{S\}) \cap F = \emptyset$. Given also a set $X$ of elements to minimize with respect to, $\text{Min}_X(F)$ denotes the $X$-subset minimal elements $S \in F$ defined by the condition $S \in \text{Min}((S' \in F \mid S' \setminus X = S \setminus X))$. For convenience, the $\subseteq$-minimal models of a program $P$ are denoted by $\text{MM}(P) = \text{Min}(\text{CM}(P))$, and the $X$-minimal models of $P$ by $\text{MM}_X(P) = \text{Min}_X(\text{CM}(P))$. Given these concepts, answer sets of a program are defined in this dissertation as follows.

**Definition 1.** The set $\text{AS}(P)$ of answer sets of a program $P$ is the set of interpretations $I \subseteq \text{At}(P)$ such that $I \in \text{MM}_X(P^I)$, where minimization is over $X = \text{At}(P)$ and $X = \text{head}(P)$ in Chapters 3 and 5, respectively.

Here, the setting $X = \text{At}(P)$ reflects usual definitions [59, 60, 104], whereas $X = \text{head}(P)$ gives an altered definition that restricts minimization to head atoms. Minimization of atoms in $\text{head}(P)$ as opposed to $\text{At}(P)$ facilitates the coexistence of rules and nogoods in programs in a way that helps in Chapter 5 to analyze ASP solving. When $X = \text{head}(P)$, atoms $a \in \text{At}(P) \setminus \text{head}(P)$ without defining rules are treated classically. Consequently, for a program of only nogoods, this semantics coincides with the semantics of classical logic, and for a program of only rules, with the standard answer set semantics obtained with $X = \text{At}(P)$.

Note that positive programs $P$ without disjunctive rules (2.7) are guaranteed to have precisely one minimal model $M \in \text{MM}(P)$, which is also a unique answer set $M \in \text{AS}(P)$ of $P$.

At an interpretation $I \subseteq \text{At}(P)$, a pseudo-Boolean expression $e = w_1l_1 + \cdots + w_nl_n$ has the value $e(I) = \sum_{1 \leq i \leq n, l_i = 1} w_i$ obtained by summing the weights of satisfied literals. The answer sets of an optimization program $\langle P, e \rangle$ are given by the answer sets of $P$. The answer sets $M$ of $\langle P, e \rangle$ with the least value $e(M)$ are $e$-optimal.

### 2.3 ASP with First-Order Variables

Users of ASP typically write programs containing (first-order) variables described in this section. Such variables were already used in the examples of Chapter 1, where they were denoted by uppercase letters like $U$ and $V$. Rules with these variables can be read as shorthands; a single first-order rule $r$ with variables stands for a number of similarly structured ground rules without variables.
All these ground rule instances are obtained in a process called *grounding*, which substitutes the variables in $r$ with *terms* relevant to the program. For instance, given a universe $\{a, b, c\}$ of relevant terms, the rule $p(X) :- q(X)$ would stand for the subprogram

$$
\begin{align*}
p(a) & :- q(a), \\
p(b) & :- q(b), \\
p(c) & :- q(c).
\end{align*}
$$

Variables are generally used as arguments to *predicates*, like above, where variable $X$ occurs as arguments of predicates $p$ and $q$. Once grounded, predicates yield propositional atoms such as $p(a)$, $p(b)$, and $p(c)$, which simply have some structure in their names.

The semantics of first-order programs derives from ground programs. Indeed, the answer sets of a first-order program are given by the answer sets of its ground version—a program obtained by grounding each of its rules. Many ASP solving tools make use of this connection by including a * grounder component that grounds entire first-order programs into ground programs, and a solver component that then searches answer sets for the ground programs [72]. In practice, modern grounders apply semi-naive database evaluation techniques to avoid redundant work and irrelevant ground instances [72].

**Example 1.** The following three programs encode the same task as a first-order program $P$ and as two alternative grounded versions. As only the terms 1 and 2 appear in $P$, they form a natural universe $U = \{1, 2\}$ of relevant terms, called the Herbrand universe of $P$.

<table>
<thead>
<tr>
<th>First-order</th>
<th>Naive Grounding</th>
<th>Smarter Grounding</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e(1,2)$.</td>
<td>$e(1,2)$.</td>
<td>$e(1,2)$.</td>
</tr>
<tr>
<td>$e(2,2)$.</td>
<td>$e(2,2)$.</td>
<td>$e(2,2)$.</td>
</tr>
<tr>
<td>$p(X) :- e(X,Y)$.</td>
<td>$p(1) :- e(1,1)$.</td>
<td>$p(1) :- e(1,2)$.</td>
</tr>
<tr>
<td></td>
<td>$p(1) :- e(1,2)$.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$p(2) :- e(2,1)$.</td>
<td>$p(2) :- e(2,1)$.</td>
</tr>
<tr>
<td></td>
<td>$p(2) :- e(2,2)$.</td>
<td></td>
</tr>
</tbody>
</table>

On the left is the first-order version; it is what a user might write. In the middle is a naive ground version, say $P'$, useful for formally defining the semantics of $P$ as $\text{AS}(P) = \text{AS}(P')$. On the right is a smarter ground version closer to what a modern grounder [72] would produce. This smarter version is more useful for actually computing the answer sets of $P$, because it is free of some irrelevant ground instances of rules. All these versions have the same unique answer set $\{e(1,2), e(2,2), p(1), p(2)\}$. In fact, some grounders like the one in CLINGO [49] would simplify the entire program into the facts $\{e(1,2). e(2,2). p(1). p(2)\}$.}

In general, first-order ASP programs can be more elaborate in form than this. The ASP-Core 2 standard describes further features such as *function terms* and *term arithmetic* [29]. While function terms are not used in this dissertation, term arithmetic proves useful in Chapter 4. Arithmetic in a first-order program like $P = \{a(1). b(X + 1) :- a(X)\}$ resolves during grounding, turning $P$ into $\{a(1). b(2) :- a(2)\}$, for example.
2.4 ASP Equivalence

This section recalls the basic notions of weak and strong equivalence while a deeper investigation of equivalence is left for Chapter 3.

The answer set semantics leads immediately to a notion of equivalence based on a direct comparison of answer sets. Namely, two programs \( P \) and \( Q \) are defined to be (weakly) equivalent, denoted by \( P \equiv Q \), iff they have the same sets \( AS(P) = AS(Q) \) of answer sets. Note that this definition of equivalence is ambivalent to the specific syntactic classes of the compared programs. This desirable quality facilitates the comparison of even programs from different syntactic classes. Such cases arise, for example, when comparing programs against their normalized counterparts.

The semantics of answer set programs has a global and non-monotonic nature, due to which the answer sets of a program may both increase as well as decrease in number when new rules are added to the program. Moreover, two programs with the same sets of answer sets may end up with different sets of answer sets after such an addition. Hence, the equivalence of programs may be lost in such a way, as demonstrated below.

\[ Example \, 2. \] Consider the weakly equivalent programs \( P = \{a \mid b.\} \) and \( Q = \{a : \neg b. \, b : \neg a.\} \) satisfying \( P \equiv Q \) and having the sets of answer sets \( AS(P) = \{(a), (b)\} = AS(Q) \). The addition of the rules \( R = \{a : b. \, b : a.\} \) yields \( AS(P \cup R) = \{(a,b)\} \) and \( AS(Q \cup R) = \emptyset \), so that \( P \cup R \not\equiv Q \cup R \).

This is in contrast to the classical semantics of Boolean formulas, according to which the (classical) models of a formula completely capture its semantics. Hence, any two formulas with the same sets of models will always end up with the same sets of models after an addition of identical constraints. Therefore, the equivalence of formulas is preserved by such additions, and it is on that account a congruence relation for union. The lack of this congruence property hinders the use of weak equivalence particularly when justifying the correctness of non-trivial modular transformations. Such transformations process programs in parts and transform each part in isolation from the rest of the program, and they include the normalizations discussed in Chapter 3. The correctness of transforming each part is naturally described as the requirement that the answer sets of the entire program are preserved, regardless of the other program parts, and this is not captured by weak equivalence.

A strengthened notion of program equivalence does however have the congruence property for union. Namely, programs \( P \) and \( Q \) from a class of interest are called strongly equivalent, denoted by \( P \equiv_s Q \), iff \( P \cup R \equiv Q \cup R \) for any program \( R \) from the same class \([80]\). Strong equivalence implies weak equivalence, as can be seen by setting \( R = \emptyset \).

\[ Example \, 3. \] The programs below are strongly equivalent: instances of either one in the context of any larger program can be replaced with the other one, and there will be no change to the answer sets of the entire program.
Example 4. The weakly equivalent programs \( P \) and \( Q \) from Example 2 are not strongly equivalent, and this is witnessed by the given context program \( R \).

Moreover, both weak equivalence and strong equivalence admit alternative characterizations that may help establish equivalence results. For instance, study of weak equivalence is aided by the fact that answer sets of programs correspond exactly to equilibrium models of analogous propositional formulas [98,99]. Furthermore, and more importantly to this work, equilibrium models are a sub-class of here-and-there (HT) models, which characterize strong equivalence [99]. Strong equivalence is also characterized by strong equivalence (SE) models [80], and, in fact, SE-models and HT-models essentially coincide in the absence of classical negation [108]. Both of these model based characterizations of strong equivalence are independent of the class of context programs. Given these alternative characterizations, the sequel of this work builds on SE-models. This is convenient because SE-models can be defined in terms of concepts familiar from the definition of answer sets. Indeed, the set \( \text{SE}(P) \) of SE-models of a program \( P \) consists of the pairs \( \langle X, Y \rangle \), where \( X \subseteq Y \subseteq \text{At}(P) \) are interpretations such that \( Y \models P \) and \( X \models P^Y \) [?]. Given the sets of SE-models of two programs, \( P \) and \( Q \), their strong equivalence can be checked using the fact that \( P \equiv_s Q \) iff \( \text{SE}(P) = \text{SE}(Q) \) [?].

Example 5. The strong equivalence of the programs \( P \) and \( Q \) from Example 3 is reflected in their sets of SE-models \( \langle X, Y \rangle \in \text{SE}(P) = \text{SE}(Q) \):

\[
\begin{array}{c|c}
X & Y \\
\hline
a, b, c & a, b, c \\
\hline
a, b & a, b, d \\
& a, b, d \\
& a, b, c, d \\
& a, b, c, d \\
& a, b, c, d \\
\end{array}
\]

Answer set programs regularly contain atoms necessary for the inspection of solutions and atoms defined only to help check properties of solutions. In practice it is desirable to show the first kind of atoms and hide the rest from the user when
Preliminaries

presenting answer sets. This idea of atom visibility is formalized here as in [66] by partitioning the signature \( \text{At}(P) \) of \( P \) into the visible signature \( \text{At}_v(P) \) and hidden signature \( \text{At}_h(P) \), which are defined case-by-case for different programs. Moreover, the projection of an interpretation \( I \) onto a set \( A \) of atoms is defined, so that \( I \mid_A \) denotes \( I \cap A \). This notation is extended by shorthands \( I_v = I \mid_{\text{At}_v(P)} \) and \( I_h = I \mid_{\text{At}_h(P)} \), and generalized to sets \( S \subseteq 2^{\text{At}(P)} \) of interpretations, by writing \( S \mid_A = (I \mid_A \mid I \in S) \) and similarly for \((\cdot)_v \) and \((\cdot)_h \). The idea behind atom visibility is to inspect and compare answer sets \( M \in \text{AS}(P) \) only based on their visible parts \( M_v \). This idea is captured by the equivalence relation visible equivalence [66] defined using the concept of visible equality of sets of interpretations.

Definition 2. Given logic programs \( P \) and \( Q \) such that \( \text{At}_v(P) = \text{At}_v(Q) \), the sets of interpretations \( S_1 \subseteq 2^{\text{At}(P)} \) and \( S_2 \subseteq 2^{\text{At}(Q)} \) are visibly equal, denoted by \( S_1 = v S_2 \), iff there is a bijection \( f : S_1 \rightarrow S_2 \) such that for every \( I \in S_1 \), \( I_v = f(I)_v \).

Definition 3 (Visible Equivalence [66]). Logic programs \( P \) and \( Q \) are visibly equivalent, denoted by \( P \equiv_v Q \), iff \( \text{At}_v(P) = \text{At}_v(Q) \) and \( \text{AS}(P) = v \text{AS}(Q) \).

Visible equivalence generalizes weak equivalence since a completely visible signature \( \text{At}(P) = \text{At}_v(P) \) leads them to coincide. Indeed, visible equivalence becomes interesting once some atoms are hidden. The fact that \( \equiv_v \) is rooted in a bijective correspondence of answer sets respects the perspective of the user. This is generally called for, and particularly when counting answer sets, since it is entirely possible for multiple answer sets to coincide in their visible atoms, in which case their visible projection is printed multiple times. This bijectivity requirement is missing from equivalence notions that project away these multiples, such as the notions in [43].

Example 6. Programs \( P \) and \( Q \) inducing \( \text{AS}(P) = \{\{a\}, \{b\}\} \) and \( \text{AS}(Q) = \{\emptyset, \{b\}\} \) are made visibly equivalent by setting \( \text{At}_v(P) = \{b\} = \text{At}_v(Q) \) so that \( a \) is hidden and \( P \equiv_v Q \). On the other hand, if \( a \) is kept visible by having \( \text{At}_v(P) = \{a, b\} = \text{At}_v(Q) \), then the visible equivalence is broken, so that \( P \not\equiv_v Q \).

2.5 ASP Programs for Merging and Sorting

This section presents the concepts of sorting and merging programs. Such programs are used as building blocks of normalizations of cardinality rules in Publication VI and weight rules in Publication VII, as well as optimization rewritings discussed in Chapter 5.

The below definition formalizes the concept of literals sorted by truth values. In this thesis, the convention is to place false values at low indices and true values at high indices, so that the sequence \( \bot, \bot, \top, \top, \top \) is considered sorted whereas \( \top, \top, \top, \bot, \bot \) is not.

Definition 4. Literals \( s_1, \ldots, s_n \) are sorted at an interpretation \( I \) iff \( I \models s_i \)
implies \( I \models s_{i+1} \) for all \( 1 < i \leq n \), and sorted under a set \( S \) of interpretations iff 
\( s_1, \ldots, s_n \) are sorted at every \( I \in S \).

Programs that produce these sorted sequences most prominently include sorting and merging programs, which take as input either any sequences or specifically pairs of sorted sequences, respectively.

**Definition 5.** Let \( t_1, \ldots, t_n \) be literals, \( s_1, \ldots, s_n \) atoms, and \( P \) a program yielding exactly one answer set \( M_I \in \text{AS}(P \cup \{ t. \mid t \in I \}) \) for each \( I \subseteq \text{At}(\{t_1, \ldots, t_n\}) \). Then, \( P \) is

1. a sorting program with input \( [t_1, \ldots, t_n] \) and output \( [s_1, \ldots, s_n] \) iff for each \( I \subseteq \{t_1, \ldots, t_n\} \), we have \( M_I \not\models s_1, \ldots, M_I \not\models s_{n-k} \) and \( M_I \models s_{n-k+1}, \ldots, M_I \models s_n \), where \( k = |\{t_i \mid I \models t_i\}| \).

2. a merging program with inputs \( [t_1, \ldots, t_n] \), \( [h_1, \ldots, h_n] \) and output \( [s_1, \ldots, s_n] \) iff for each \( I \subseteq \{t_1, \ldots, t_n\} \) at which \( [t_1, \ldots, t_n] \) and \( [h_1, \ldots, h_n] \) are sorted, we have \( M_I \not\models s_1, \ldots, M_I \not\models s_{n-k} \) and \( M_I \models s_{n-k+1}, \ldots, M_I \models s_n \), where \( k = |\{t_i \mid I \models t_i\}| + |\{h_i \mid I \models h_i\}| \).

Moreover, for any program \( P' \supseteq P \), it is assumed that \( (\text{At}(P) \cup \text{At}(\{s_1, \ldots, s_n\})) \cap \text{head}(P' \setminus P) \subseteq \text{At}(\{t_1, \ldots, t_n\}) \).

Both types of programs can be constructed for any given number \( n \) of inputs in various ways, such as by encoding sorting networks, which are reviewed in Sections 2.6 and 2.7. For illustration, a small sorting program for \( n = 2 \) inputs is discussed below.

**Example 7.** A sorting program with input \([t_1, t_2]\) and output \([s_1, s_2]\) is simple to construct:

\[
\begin{align*}
s_1 & : - t_1, t_2. \\
s_2 & : - t_1. \\
s_2 & : - t_2. 
\end{align*}
\]

The rules for the output \( s_1 \) compute the logical AND of the inputs \( t_1 \) and \( t_2 \) whereas the rules for \( s_2 \) compute the logical OR. Therefore, any answer set of the program satisfies the same number of the outputs as of the inputs, so that the program implements a permutation. Moreover, \( s_1 \) clearly implies \( s_2 \), and hence the program meets the definition of a sorting program.

### 2.6 Comparator and Sorting Networks

This section reviews two types of circuits called comparator networks and sorting networks. They are a useful component in designs of normalizations and related rewritings used and developed in this thesis.

A comparator network is a specific type of permutation algorithm that can be represented by a fixed circuit-like network of computational elements called
comparators. A comparator network \( N \) of width \( n \) and depth \( d \) can be formalized as a set of comparators \((i, j, l)\) on wires \(1 \leq i < j \leq n\) and layers \(1 \leq l \leq d\) such that all wire-layer combinations \((i, l)\) and \((j, l)\) are unique in the network. The layers of a network also refer to the sub-networks \(L_1, \ldots, L_d\) that partition its comparators into those on the different respective layers \(l \in 1, \ldots, d\). The execution of a comparator network conceptually proceeds layer by layer, in increasing order of indices. The first layer processes the input of the network, and the subsequent layers process the outputs of their preceding layers. The execution of a layer, in turn, proceeds comparator by comparator in parallel. Formally, the layer function \( f \) of a layer \( L \) is to permute an input vector \( \vec{x} \) to an equal-length output vector where the values \( x_i \) and \( x_j \) of wires of its comparators \((i, j, l) \in L\) are swapped unless they are already correctly ordered so that \( x_i \leq x_j \). Furthermore, the function \( f \) of a network \( N \) with layer functions \( f_1, \ldots, f_d \) is to permute an input vector \( \vec{x} \) to an equal-length output vector \( f_d(\cdots f_1(\vec{x})\cdots) \).

A comparator network \( N \) that produces only sorted output vectors \( \vec{y} \) such that \( y_1 \leq \cdots \leq y_n \) is called a sorting network. An example of a sorting network is shown below.

**Example 8.** One possible sorting network for four inputs is shown below in two Knuth diagrams, in which horizontal lines depict wires \( i \) and vertical lines depict wires connected by comparators on layers \( l \). On the left, the network sorts the numbers \([2, 3, 4, 1]\) to \([1, 2, 3, 4]\), and on the right, the Boolean values \([0, 1, 1, 0]\) to \([0, 0, 1, 1]\). The idea is that each comparator takes the values on its wires from the left and outputs them in sorted order on the right. For example, the top-left comparator maps \([2, 3]\) to itself, and the bottom-left comparator sorts \([4, 1]\) to \([1, 4]\). Both of the shown input sequences turn out entirely sorted in the end, as is guaranteed by the definition of a sorting network.

\[
\begin{array}{ccc}
  l = 1 & l = 2 & l = 3 \\
  i = 1 & \begin{array}{c}
  2 \\
  3 \\
  4 \\
  1
\end{array} & \begin{array}{c}
  1 \\
  3 \\
  2 \\
  4
\end{array} \\
  i = 2 & \begin{array}{c}
  2 \\
  3 \\
  1 \\
  4
\end{array} & \begin{array}{c}
  1 \\
  2 \\
  3 \\
  4
\end{array} \\
  i = 3 & \begin{array}{c}
  1 \\
  4 \\
  4 \\
  4
\end{array} & \begin{array}{c}
  0 \\
  0 \\
  0 \\
  0
\end{array} \\
  i = 4 & \begin{array}{c}
  0 \\
  1 \\
  1 \\
  1
\end{array} & \begin{array}{c}
  0 \\
  1 \\
  0 \\
  1
\end{array}
\end{array}
\]

The precise class of input elements is not crucial in the definition of a sorting network. This is due to the interesting fact that comparator networks adhere to the 0-1 principle, by which a comparator network that sorts all possible Boolean input vectors is also guaranteed to sort input vectors with elements from any totally ordered domain \([74]\).
2.7 ASP Programs that Encode Comparator and Sorting Networks

This section describes how comparator networks from Section 2.6 translate naturally into ASP. The importance of the translation stems greatly from the fact that when applied to appropriate types of networks, such as the odd-even merging networks and odd-even sorting networks from [19], the translation yields programs that qualify as merging and sorting programs from Section 2.5. Such programs are immediately useful in normalizations. Further benefits of the translation in the context of optimization are demonstrated in Chapter 5.

Indeed, as discussed above, comparator networks $N$ can be translated into positive ASP programs $\text{ASP}(N)$ that permute Boolean vectors $\vec{x}$. The translation uses an atom $x_{il}$ to capture the wire value of wire $1 \leq i \leq n$ in the output $f_l(\cdots f_1(\vec{x}) \cdots)$ of each layer $1 \leq l \leq d$ and in the input when $l = 0$. The intention is that the atom is true iff the value is 1. Each comparator $\langle i, j, l \rangle \in N$ can be translated, as shown in Publication IV, into an instantiation of the small sorting program shown in Example 7 with input $[x_{i(l-1)}, x_{j(l-1)}]$ and output $[x_{il}, x_{jl}]$:

$$
\begin{align*}
x_{il} & : - x_{i(l-1)}, x_{j(l-1)} . \\
x_{jl} & : - x_{i(l-1)} . \\
x_{jl} & : - x_{j(l-1)} .
\end{align*}
$$

Wires $i$ untouched by comparators at layer $l$ are defined by a rule of inertia:

$$
x_{il} : - x_{i(l-1)} .
$$

An example application of the translation is shown below.

**Example 9.** The wires of network $N$ from Example 8 relate to atoms $x_{il}$ as illustrated below.

The network $N$ translates into the following program $\text{ASP}(N)$, which defines all but the input atoms $x_{i0}$; rules of inertia are shown in gray:
Formally, the translation establishes the following connection between the values of wires in the comparator network and the satisfaction of the atoms in the translation:

**Lemma 1** (Lemma 1 rephrased from Publication V). Let $N$ be a comparator network with layer functions $f_1, \ldots, f_d$ of width $n$, and $\text{ASP}(N)$ its translation into a positive answer set program. Moreover, let $\vec{x}$ be any Boolean input vector for $N$ and $\text{InF}(\vec{x}) = \{x_{i0} \mid x_{i0} = 1\}$ an encoding of the input vector $\vec{x}$ as facts. Then $\text{ASP}(N) \cup \text{InF}(\vec{x})$ has a unique answer set $X \subseteq \text{At}(\text{ASP}(N))$ such that for all wires $1 \leq i \leq n$ and indices $0 \leq l \leq d$, the atom $x_{il} \in X$ iff the value $(f_l(\cdots f_1(\vec{x})\cdots))_i = 1$.

The proof is by induction on the layer indices $0 \leq l \leq d$.

As a consequence of the result, sorting networks translate to sorting programs, and indeed, the translated program $\text{ASP}(N)$ detailed in Example 9 is such.

**Lemma 2.** Let $N$ be a sorting network of width $n$ and depth $d$ and $\text{ASP}(N)$ its translation into a positive answer set program. Then $\text{ASP}(N)$ is a sorting program with input $[x_{10}, \ldots, x_{n0}]$ and output $[x_{1d}, \ldots, x_{nd}]$.

**Proof.** Let $N$, $n$, $d$, and $\text{ASP}(N)$ be as above. Let $I \subseteq \{x_{10}, \ldots, x_{n0}\}$ be an interpretation of the inputs. Moreover, let $\vec{x}$ be a Boolean input vector for $N$ that represents $I$ such that $x_i = 1$ iff $x_{i0} \in I$. Then, Lemma 1 gives that $\text{ASP}(N) \cup \{t \mid t \in I\}$ has a unique answer set $M$ such that for all wires $1 \leq i \leq n$ and in particular the output at index $l = d$, the atom $x_{il} \in M$ iff $(f_1(\cdots f_1(\vec{x})\cdots))_i = 1$. Since the output values $f_d(\cdots f_1(\vec{x})\cdots)_i$ of the network form a sorted vector by assumption, and a permutation by virtue of the network being a comparator network, the translation $\text{ASP}(N)$ is a sorting program. $\square$
3. Normalization and Visible Strong Equivalence

This chapter discusses the concepts of normalization and visible strong equivalence in ASP. Normalization is the process of transforming programs consisting of various rule types (2.1–2.4) into ones consisting of only normal rules (2.1) [69]. While programs simplify in form when normalized, they remain the same in meaning. Hence, solving the resulting program of normal rules yields answer sets of the original, with possibly some added auxiliary atoms. This kind of simplification of form facilitates further processing, such as debugging [100], translation into other declarative problem representations [95], use of solving techniques to which it would be challenging to add extended rule support otherwise [47], and pursuit of increased solving performance on specific benchmark problems in Publication VI and Publication VII. In Section 3.1, a number of techniques for normalization are overviewed.

The discussion of normalization is followed by an introduction of an equivalence relation called visible strong equivalence (VSE) in Section 3.2. The relation is specifically intended to aid in establishing the correctness of practical program transformations such as normalizations. Normalized programs are intended to be interchangeable with their original versions regardless of the surrounding program. Accordingly, VSE is equipped with a unique combination of properties. Among these properties is the consideration of the visibility of atoms, so that hidden implementation details of programs can be abstracted away from obstructing comparisons. In order to help in proofs of visible strong equivalence, an associated model based characterization is developed in Section 3.3. Furthermore, the characterization conveniently enables an automated verification approach presented in Section 3.4. To verify VSE, the approach uses a pair of transformations and standard ASP solvers under certain assumptions on the compared programs. Moreover, in order to ease correctness proofs, the characterization is specialized for positive programs in Section 3.5 and then used in the context of a certain class of nearly positive programs in Section 3.6. These results are then applied both formally and experimentally to establish the general correctness of specific cardinality rule normalization schemes in Sections 3.7 and 3.8, respectively. Finally, VSE is contrasted with related work in Section 3.9.
3.1 Normalization

Below we begin with a brief survey of existing normalization techniques for cardinality rules (2.3) and weight rules (2.4), i.e., for rules with count and sum aggregates. The concept of a cone-of-influence in the context of normalization is then presented for the purpose of obtaining more succinct normalizations. All of the mentioned techniques are implemented in the normalization tool LP2NORMAL (v. 2.27)\(^1\).

Cardinality rules can be normalized in various ways, including with the counting grid normalization \([69, 104]\), which is used in the sequel as a concrete running example of a normalization scheme. This scheme encodes cardinality rules with \(n\) body literals and a bound of \(k\) in \(O(nk)\) normal rules and auxiliary atoms. The idea is to divide the evaluation of the involved count aggregate \(#\text{count}\{1: l_1; \ldots; n: l_n\} \geq k\) into subproblems concerning smaller aggregates \(#\text{count}\{j: l_j; \ldots; n: l_n\} \geq i\) for all \(1 \leq i \leq k\) and \(k - i + 1 \leq j \leq n - i + 1\). Using fresh atoms \(d_{i,j}\), the scheme captures partial results, which are defined inductively in terms of \(d_{i,j+1}\) and \(d_{i-1,j+1}\). The final answer is given by \(d_{k,1}\).

**Definition 6.** The counting grid normalization transforms a cardinality rule \(a :- \#\text{count}\{1: l_1; \ldots; n: l_n\} \geq k\) (2.3) to the set of normal rules:

\[
\begin{align*}
a & :- d_{k,1}. \\
d_{i,j} & :- d_{i,j+1}. & (1 \leq i \leq k \text{ and } k - i + 1 \leq j < n - i + 1) \\
d_{i,j} & :- d_{i-1,j+1}, l_j. & (1 < i \leq k \text{ and } k - i + 1 \leq j < n) \\
d_{1,j} & :- l_j. & (k \leq j \leq n)
\end{align*}
\]

**Example 10.** The counting grid normalization of the rule \(a :- \#\text{count}\{b; b; c: not c\} \geq k\) with the bounds \(k = 1\) and \(k = 2\) are:

<table>
<thead>
<tr>
<th>(k = 1)</th>
<th>(k = 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a :- d_{1,1})</td>
<td>(a :- d_{2,1})</td>
</tr>
<tr>
<td>(d_{1,1} :- d_{1,2})</td>
<td>(d_{2,1} :- d_{1,2}, b)</td>
</tr>
<tr>
<td>(d_{1,2} :- not c)</td>
<td>(d_{1,2} :- not c)</td>
</tr>
<tr>
<td>(d_{1,1} :- b)</td>
<td>(d_{1,1} :- b)</td>
</tr>
</tbody>
</table>

The correctness of normalization is established in Section 3.7. Essentially identical normalization designs include the Sinz counter \([105]\) and the sequential counter (SEQ) \([63]\). From another point of view, this counting process amounts to sorting the truth values of the input literals. Moreover, other types of sorting approaches are also applicable, such as merge sorting, where partial counts are recursively merged together. The use of odd-even merging networks in the

\(^1\)Available at [http://research.ics.aalto.fi/software/asp.](http://research.ics.aalto.fi/software/asp)
design leads to $O(n(\log n)^2)$ rules, and the use of direct mergers without auxiliary variables leads to an encoding of a totalizers with $O(n^2)$ rules [12].

The preceding translations contain no negation beyond that already present in the input literals. Hence the normalizations amount to encodings of monotone circuits with logical AND & OR gates. This is beneficial from the point of view of unit propagation performed by solvers to deduce unknown truth values during search. More compact normalizations are possible via the use of negation, such as encodings of binary adders having $O(n \log k)$ rules and atoms. However, such encodings reduce the deductions possible via unit propagation, and have been regarded as undesirable until the recent success of the PICAT-SAT compiler, which employs such encodings among various other techniques [114].

Practical weight rule normalizations in ASP generally fall into the following categories:

(i) Binary counter based designs, which are compact, but pose a challenge for propagation as in the case with cardinality rules.

(ii) Reduced-Ordered Binary Decision Diagrams (ROBDDs) [1] and Sequential Weight Counters (SWC) [63], which lead to alternative formulations of nearly equivalent normalizations if the SWC is simplified via partial evaluation of truth values and cone-of-influence optimization. Both are of size $O(nk)$. This size is worst-case exponential in the length of rules when bounds $k$ are considered to count logarithmically towards length. However, the normalizations are useful in practice for small rules and rules with structure in their weights, because structure alleviates the worst-case behavior. Indeed, reaching the asymptotic worst case requires sum aggregates with broadly varying magnitudes of weights such as $n$ successive powers of two.

(iii) Designs combining sorters and mergers into larger constructions [40], where the sorters sum up digits of separate levels of significance in unary, and the mergers combine the outputs into a binary or mixed-radix representation of the total sum via deferred carry propagation. These designs fall in the middle ground in asymptotic size between binary counters and SWC/ROBDD approaches by having a size of $O(n(\log n)^2 \log w_{\text{max}})$, where $w_{\text{max}}$ is the largest input weight. Moreover, their size can be compressed via structure sharing as shown in Publication VII. Namely, the included sorters generally have significant overlap in their input literals and this leads to redundancy in their structure. When merge sorters are used, this redundancy can be intentionally maximized in their construction so that it manifests in exact duplicates of sub-structures, which can then be eliminated easily.

When normalizing a ground cardinality or weight rule, there is only a single output atom of interest. Nevertheless, many rewriting schemes naturally yield programs that define a number of output atoms. For example, sorting based
normalizations of cardinality rules with \( n \) input atoms produce vectors of \( n \) output atoms, of which exactly one is necessary. Moreover, even selector based translations \([11]\) that are specifically designed to benefit from low bounds \( k \) produce at least \( k \) output atoms. This superfluous number of outputs can be reduced by performing cone of influence simplification proposed in Publication 1. The idea is to trace a “cone” of necessary atoms backwards from the wanted output, so as to determine all the atoms that impact the definition of that output directly or transitively. Rules that define those atoms can then be identified in a forward phase, yielding the simplified normalization. During the development of LP2NORMAL, this simplification has proved to be effective in reducing the size of especially sorter and SWC based normalizations. For example, cone-of-influence simplification reduces the size of odd-even merge sorting programs down from \( O(n(\log n)^2) \) to \( O(n(\log m)^2) \), where \( m \) is the distance from the bound to the closest of 1 and \( n + 1 \). The resulting asymptotic size matches the size \( O(n(\log k)^2) \) of selection networks used in SAT \([11]\) and even surpasses it for large \( k \).

### 3.2 Visible Strong Equivalence

Normalization brings about the question of how to establish the correctness of normalizations, as well how the correctness of a normalization is defined to begin with. These questions are addressed in the design of a novel equivalence relation, visible strong equivalence (VSE). Regarding preliminaries, let us recall from Section 2.4 the prior equivalence notions \( \equiv \), \( \equiv_v \), and \( \equiv_s \). Of these notions, visible equivalence \( \equiv_v \) \([66]\) incorporates bijective comparisons, whereas strong equivalence \( \equiv_s \) \([80]\) considers context programs. Both equivalence relations can be generalized by a single relation and, to that end, this section presents visible strong equivalence \( \equiv_v \). The consequent relationships between \( \equiv, \equiv_v, \equiv_s \) and \( \equiv_v \) are illustrated in Figure 3.1. In more detail, the generalization process involves the question of how visibility should affect the context programs. The solution in \( \equiv_v \) is to use context programs \( R \) that have visible and hidden signatures, precisely the same as with programs \( P \) and \( Q \), and to insist that the hidden atoms of \( R \) are not altered by \( P \) or \( Q \) and vice versa. To help formalize the conditions, programs \( P \) and \( R \) are said to mutually respect the hidden atoms of each other iff \( \text{At}(P) \cap \text{At}_h(R) = \emptyset \) and \( \text{At}_h(P) \cap \text{At}(R) = \emptyset \). Using this symmetric notion, and the concept of visible equality from Definition 2, \( \equiv_v \) is defined as
Definition 7 (Visible Strong Equivalence). Programs $P$ and $Q$ are visibly strongly equivalent, denoted by $P \equiv_{vs} Q$, iff $At_v(P) = At_v(Q)$ and $AS(P \cup R) =_{vs} AS(Q \cup R)$ for any context $R$ that mutually respects the hidden atoms of $P$ and $Q$.

Observe that this definition allows $R$ to use its own hidden atoms that $P$ and $Q$ cannot refer to. Moreover, $P$ and $Q$ may use hidden atoms with mutually coinciding names without interfering with one another. For example, $Q$ is allowed to be an exact copy of $P$ with the names of hidden atoms shuffled.

3.3 Model Based Characterization of Visible Strong Equivalence

In order to help prove or disprove VSE for given programs, a class of models called visibility based SE-models (VSE-models) is presented here. These sets of models, denoted $VSE(P)$ for programs $P$, are designed to capture the meaning of $P$ in a manner that is sensitive to visibility and oblivious to context. Specifically, given $VSE(P)$ one may comprehensively reason about the visible parts of the answer sets of $P$ in the context of any program $R$ that mutually respects the hidden atoms of $P$. In particular, the sets $VSE(P)$ and $VSE(Q)$ of VSE-models of two programs $P$ and $Q$ contain enough information to determine whether the visible strong equivalence $P \equiv_{vs} Q$ holds. The primary idea is to distinguish visible and hidden atoms by enforcing the hidden atoms to be false by default. This enforcement is achieved via minimization, which relies on the concept of the $At_h(P)$-minimal models of a program $P$, called hidden minimal models and denoted by $MM_h(P)$. Visible atoms are treated classically as in SE-models and, in fact, VSE-models and SE-models coincide in the absence of hidden atoms. For convenience, it is assumed that the reduct $P^Y$ of a program $P$ with respect to any interpretation $Y$ has the same visible and hidden signatures as $P$.

Definition 8. A VSE-model of a logic program $P$ is a pair $\langle X, Y \rangle$ of interpretations, where $X \subseteq Y \subseteq At(P)$, $X \in MM_h(P^Y)$, and $Y \in MM_h(P^Y)$.

Intuitively, the first components $X$ capture the behavior of $P$ in a context represented by $Y$. As mentioned, visibility is considered in interpreting hidden atoms minimally and visible atoms classically, and thus a fully classical interpretation is obtained when $At_h(P) = \emptyset$, in which case $VSE(P) = SE(P)$.

The comparison of VSE-models shown below, for the purpose of investigating $\equiv_{vs}$, relies on the operations of projecting sets $S$ of VSE-interpretations to obtain their first arguments $\pi_1(S) = \{ X \mid \langle X, Y \rangle \in S \}$ and second arguments $\pi_2(S) = \{ Y \mid \langle X, Y \rangle \in S \}$; and on selecting sets $\sigma_Z(S) = \{ \langle X, Y \rangle \in S \mid Y = Z \}$ of VSE-interpretations associated with a given interpretation $Z \subseteq At(P)$. This notation is also used for $Z_v \subseteq At_v(P)$, in which case the selection condition is $Y_v = Z_v$. 

49
Definition 9. Given programs $P$ and $Q$ such that $At_v(P) = At_v(Q)$, the respective sets $VSE(P)$ and $VSE(Q)$ visibly match, denoted $VSE(P) \equiv_{vs} VSE(Q)$, iff

1. $\pi_2(VSE(P)) \equiv_{v} \pi_2(VSE(Q))$ via a bijection $f$ and
2. for each matching pair of models $Y \in \pi_2(VSE(P))$ and $f(Y) \in \pi_2(VSE(Q))$,

$$\pi_1(\sigma_Y(VSE(P)))_v = \pi_1(\sigma_{f(Y)}(VSE(Q)))_v.$$  \hfill (3.1)

This concept of visibly matching VSE-models provides the link between them and $\equiv_{vs}$.

Theorem 1 (Theorem 3.21 from Publication II). For programs $P$ and $Q$ with $At_v(P) = At_v(Q)$, $VSE(P) \equiv_{vs} VSE(Q)$ iff $P \equiv_{vs} Q$.

The corresponding result and proof in Publication II is given in two parts: a soundness result (Publication II, Theorem 3.21) to establish that the visible match of VSE-models implies $\equiv_{vs}$ and a completeness result (Publication II, Theorem 3.27) to establish that the lack of a visible match precludes $\equiv_{vs}$.

Example 11 (Example 5.5 from Publication II). Recall the cardinality rule $a:\;\#count\; \{b; b_1; c; not\; c\} \geq 1$ and its translation $\{a:\; d_{1,1}; d_{1,1}; d_{1,2}; d_{1,1}; b; d_{1,2}; d_{not\; c}\}$, from Example 10 with the bound $k = 1$, and denote them by $P$ and $Q$, respectively. Moreover, divide their signatures so that $At_v(P) = \emptyset$, and $At_h(Q) = \{d_{1,1}, d_{1,2}\}$. Their VSE-models $\langle X, Y \rangle \in VSE(P)$ and $\langle X', Y' \rangle \in VSE(Q)$ are shown below, grouped by the hidden minimal models $Y \in MM_h(P)$ and $Y' \in MM_h(Q)$. The reduct $P^Y$ of each $Y$ is of the form $a:\;\#count\; \{b; b\} \geq k$, with the indicated bound $k$.

<table>
<thead>
<tr>
<th>$Y$</th>
<th>$k$</th>
<th>$X$</th>
<th>$Y'$</th>
<th>$X'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${a}$</td>
<td>0</td>
<td>${a}$</td>
<td>${a, d_{1,1}, d_{1,2}}$</td>
<td>${a, d_{1,1}, d_{1,2}}$</td>
</tr>
<tr>
<td>${c}$</td>
<td>1</td>
<td>$\emptyset \ldots {c}$</td>
<td>${c}$</td>
<td>$\emptyset \ldots {c}$</td>
</tr>
<tr>
<td>${a, b}$</td>
<td>0</td>
<td>${a}, {a, b}$</td>
<td>${a, b, d_{1,1}, d_{1,2}}$</td>
<td>${a, d_{1,1}, d_{1,2}}$, ${a, b, d_{1,1}, d_{1,2}}$</td>
</tr>
<tr>
<td>${a, c}$</td>
<td>1</td>
<td>$\emptyset \ldots {a, c}$</td>
<td>${a, c}$</td>
<td>$\emptyset \ldots {a, c}$</td>
</tr>
<tr>
<td>${a, b, c}$</td>
<td>1</td>
<td>$\emptyset \ldots {a, c}$</td>
<td>${a, b, c, d_{1,1}}$</td>
<td>$\emptyset \ldots {a, c}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>${a, b, c}$</td>
<td>${a, b, d_{1,1}, {a, b, c, d_{1,1}}}$</td>
<td></td>
</tr>
</tbody>
</table>

The bijective correspondence between $Y$ and $Y'$ as insisted by $\equiv_{vs}$ is evident in the table. Moreover, the remaining condition (3.1) can be confirmed row by row. It follows that $VSE(P) \equiv_{vs} VSE(Q)$ and by Theorem 1 that $P \equiv_{vs} Q$. $\blacksquare$

3.4 Automated Verification of Visible Strong Equivalence

The verification of visible strong equivalence can be automated via a practically feasible approach based on VSE-models. The approach is based on capturing
Normalization and Visible Strong Equivalence

Table 3.1. The classes of models captured as the answer sets of programs and their translations.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Captured models</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P, Q)</td>
<td>(\text{AS}(P), \text{AS}(Q))</td>
<td></td>
</tr>
<tr>
<td>(\text{Tr}<em>{vs}(P), \text{Tr}</em>{vs}(Q))</td>
<td>(\text{VSE}(P), \text{VSE}(Q))</td>
<td></td>
</tr>
<tr>
<td>(\text{Tr}<em>{eq}(P, Q), \text{Tr}</em>{eq}(Q, P))</td>
<td>counter models to (P \equiv_v Q)</td>
<td>(P, Q) have EVA</td>
</tr>
<tr>
<td>(\text{Tr}<em>{eq}(\text{Tr}</em>{vs}(P), \text{Tr}<em>{vs}(Q)), \text{Tr}</em>{eq}(\text{Tr}<em>{vs}(Q), \text{Tr}</em>{vs}(P)))</td>
<td>counter models to (P \equiv_{vs} Q)</td>
<td>(P, Q) have EVA</td>
</tr>
</tbody>
</table>

The VSE-models of pairs of programs via translations and using standard ASP solvers to search for discrepancies between them. An experimental evaluation of the performance of the approach is presented in Section 3.8. In order to keep the computational complexity of the reasoning task within reach of ASP solvers, the compared programs are assumed to have enough visible atoms (EVA) [71]. EVA means that the visible atoms of every answer set of the program in question contain enough information to deduce unique truth values for the hidden atoms. In particular, this is the case when the hidden part of the program is positive. Regarding the translations, the goal is to search for counter models to the claim that \(P \equiv_{vs} Q\) using ASP solvers, and to prove that none can be found. This is achieved via combining two translations \(\text{Tr}_{vs}(P)\) introduced in this work and \(\text{Tr}_{eq}(P, Q)\) from [71]. The purposes of these translations are summarized in Table 3.1. In words, the \(\text{Tr}_{vs}(P)\) translation maps a program \(P\) to one whose answer sets \(\text{AS}(\text{Tr}_{vs}(P))\) correspond one-to-one to the VSE-models \(\text{VSE}(P)\) of \(P\). The second translation, \(\text{Tr}_{eq}(P, Q)\), has answer sets that provide counter models to the visible equivalence of programs \(P\) and \(Q\) with the EVA property. Specifically, from each answer set of \(\text{Tr}_{eq}(P, Q)\) one can read a pair of visibly equal interpretations, one that is an answer set of \(P\) and one that is not an answer set of \(Q\). The absence of such counter models in either direction, i.e., the unsatisfiability of both \(\text{Tr}_{eq}(P, Q)\) and \(\text{Tr}_{eq}(Q, P)\) proves \(P \equiv_{vs} Q\). Finally, the combination of these translations into \(\text{Tr}_{eq}(\text{Tr}_{vs}(P), \text{Tr}_{vs}(Q))\) and the symmetric counterpart \(\text{Tr}_{eq}(\text{Tr}_{vs}(Q), \text{Tr}_{vs}(P))\) yield counter models to \(P \equiv_{vs} Q\) as their answer sets. Again, the unsatisfiability of both programs proves the equivalence of interest: \(P \equiv_{vs} Q\).

The translation \(\text{Tr}_{vs}(P)\), given in detail in Publication II, Section 6, defines fresh atoms \(a^*\) that capture the \(X\) part of VSE-models \(\langle X, Y \rangle \in \text{VSE}(P)\). The theorem below states this formally.

**Theorem 2** (Theorem 6.2 from Publication II). Let \(P\) be a program with the EVA property and \(\text{Tr}_{vs}(P)\) its translation to capture the VSE-models of \(P\). Then for any \(X \subseteq Y \subseteq \text{At}(P)\), \(\langle X, Y \rangle \in \text{VSE}(P)\) iff \(M = Y \cup \{a^* \mid a \in X\} \in \text{AS}(\text{Tr}_{vs}(P))\).

The proof is based on a straightforward comparison of pairs \(\langle X, Y \rangle\) of interpre-
Normalizations $X \subseteq Y \subseteq \text{At}(P)$ with interpretations $Y \cup X^* \subseteq \text{At}(\text{Tr}_{\text{vs}}(P))$.

This result leads to a correctness result for the entire two step translation approach. To this end, observe that the assumed EVA property guarantees $Y' = Y$ for every pair $Y \in \pi_2(\text{VSE}(P))$ and $Y' \in \pi_2(\text{VSE}(P))$, $Y'_v = Y_v$. Hence, the translation based method for weak/visible equivalence [71] implemented in $\text{Tr}_{\text{eq}}(\cdot, \cdot)$ works as intended here when applied to the outputs of $\text{Tr}_{\text{vs}}(\cdot)$.

**Corollary 1** (Corollary 6.3 from Publication II). *Let $P$ and $Q$ be programs with the EVA property. Then $P \equiv_{\text{vs}} Q$ iff $\text{Tr}_{\text{eq}}(\text{Tr}_{\text{vs}}(P), \text{Tr}_{\text{vs}}(Q))$ and $\text{Tr}_{\text{eq}}(\text{Tr}_{\text{vs}}(Q), \text{Tr}_{\text{vs}}(P))$ have no stable models.*

### 3.5 Model Based Characterization of Visible Strong Equivalence Tailored for Positive Programs

This section demonstrates that positive programs can be checked for visible strong equivalence also by inspecting their hidden minimal models instead of the more complex VSE-models. Inspection of hidden minimal models greatly simplifies equivalence proofs. The result is attained by studying the relationship between VSE-models and hidden minimal models.

**Lemma 3** (Lemma 4.1 from Publication II). *For positive programs $P$ and $Q$ with $\text{At}_v(P) = \text{At}_v(Q)$, it holds that $\text{VSE}(P) \equiv_{\text{vs}} \text{VSE}(Q)$ iff $\text{MM}_h(P) =_{\text{vs}} \text{MM}_h(Q)$.*

The proof of Lemma 3 given in the article simplifies the $=_{\text{vs}}$ condition until it coincides with the $=_{\text{v}}$ condition. The simplifications use the fact that positive programs $P$ remain unchanged under the reduction operation involved in the definition of answer sets, so that $P^M = P$.

The direct connection between hidden minimal models and visible strong equivalence for positive programs can be obtained from Lemma 3 and Theorem 1.

**Corollary 2** (Corollary 4.2 from Publication II). *For positive programs $P$ and $Q$ with $\text{At}_v(P) = \text{At}_v(Q)$, it holds that $P \equiv_{\text{vs}} Q$ iff $\text{MM}_h(P) =_{\text{v}} \text{MM}_h(Q)$.*

**Example 12.** Recall the programs $P$ and $Q$ from Example 11, but replace occurrences of the negative literal $\text{not } c$ by a new atom $c^-$ in both programs to make them positive. *Due to the programs being positive, Corollary 2 can be used in this case to verify $P \equiv_{\text{vs}} Q$ on the basis of the hidden minimal models $Y \in \text{MM}_h(P)$ and $Y' \in \text{MM}_h(Q)$ alone:*

<table>
<thead>
<tr>
<th>$Y$</th>
<th>$Y'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>${a}$</td>
<td>${a}$</td>
</tr>
<tr>
<td>${a, b}$</td>
<td>${a, b, d_{1,1}}$</td>
</tr>
<tr>
<td>${a, c^-}$</td>
<td>${a, c^-, d_{1,1}, d_{1,2}}$</td>
</tr>
<tr>
<td>${a, b, c^-=}</td>
<td>${a, b, c^-, d_{1,1}, d_{1,2}}$</td>
</tr>
</tbody>
</table>
A comparison with Example 11 highlights the relative ease of analyzing the correctness of transformations in the case of positive programs.

### 3.6 Proof Technique for Nearly Positive Programs

This section builds on the simple characterization for positive programs given in Corollary 2. The goal is to find a result with relevance to a larger class of programs, while still relying on the simple class of hidden minimal models. This effort brings about a generalization of the “if” direction of the mentioned corollary that is applicable to programs with only visible negation. The preconditions of the generalization are stated in terms of a simple program translation. The translation introduces fresh atoms of the form $a^-$ as placeholders for negative literals $\text{not } a$. For convenience, given a set $A$ of atoms, $\bar{A}$ will denote the set $\{a^- | a \in A\}$ of corresponding fresh atoms.

**Definition 10.** Given a program $P$ and a set $A \subseteq \text{At}_v(P)$ of visible atoms, the negation substitution translation $Tr_\equiv(P,A)$ consists of the rules of $P$ with every $\text{not } a$ replaced with $a^-$, where $a \in A$. The signature of the translation consists of the visible part $\text{At}_v(Tr_\equiv(P,A)) = \text{At}_v(P) \cup A^-$ and the hidden part $\text{At}_h(Tr_\equiv(P,A)) = \text{At}_h(P)$.

**Example 13.** The programs $P$ and $Q$ be from Example 11 have the negation substitution translations $Tr_\equiv(P,A) = \{a :- \#\text{count}\{b:b; c^-:c\} \geq 1\}$ and $Tr_\equiv(Q,A) = \{a :- d_{1,1}. d_{1,1} :- d_{1,2}. d_{1,1} : b. d_{1,2} :- c^-\}$ when $A = \{c\}$.

The interesting aspect about the negation substitution translation $Tr_\equiv(\cdot, \cdot)$ is that if it is applied to any visible atoms common to a pair of programs and the translated programs are shown to be visibly strongly equivalent, then the result carries over to the original untranslated programs as well.

**Theorem 3** (Theorem 4.10 from Publication II). For programs $P$ and $Q$ with $\text{At}_v(P) = \text{At}_v(Q)$, and a set $A \subseteq \text{At}_v(P)$ of visible atoms such that $A^- \cap (\text{At}(P) \cup \text{At}(Q)) = \emptyset$, it holds that $Tr_\equiv(P,A) \equiv_{vs} Tr_\equiv(Q,A)$ implies $P \equiv_{vs} Q$.

The proof extends the precondition $Tr_\equiv(P,A) \equiv_{vs} Tr_\equiv(Q,A)$ by congruence of VSE to $Tr_\equiv(P,A) \cup R \equiv_{vs} Tr_\equiv(Q,A) \cup R$, where $R$ consists of rules of the form $a^- : \text{not } a$. that define the relevant complement atoms $a^-$. Then, the complement atoms $a^-$ are hidden from both programs. The programs are argued to remain visibly strongly equivalent between one another and each of them to be visibly strongly equivalent with the respective program among $P$ and $Q$. The final result follows by transitivity of $\equiv_{vs}$.

The proof strategy sought in this section is finally formalized in the below proposition, which follows from Corollary 2 and Theorem 3. The result is based on the realization that for pairs of programs that are positive after being translated, the corollary on positive programs can be applied to fulfilling the
Normalization and Visible Strong Equivalence

preconditions of the preceding theorem. This is essentially possible when the untranslated programs have no choice rules and no negative conditions on hidden atoms.

**Proposition 1** (Proposition 4.12 from Publication II). For programs $P$ and $Q$ with $\text{At}_v(P) = \text{At}_v(Q)$ that are positive except for possibly any negative body atoms included in a set $A \subseteq \text{At}_v(P)$ of visible atoms such that $A^- \cap (\text{At}(P) \cup \text{At}(Q)) = \emptyset$, it holds that $\text{MM}_h(\text{Tr}_{\equiv}(P,A)) \equiv_v \text{MM}_h(\text{Tr}_{\equiv}(Q,A))$ implies $P \equiv_{vs} Q$.

**Example 14.** Recall the non-positive programs $P$ and $Q$ from Example 11 shown again below. Moreover, observe that their negation substitution translations $\text{Tr}_{\equiv}(P,A)$ and $\text{Tr}_{\equiv}(Q,A)$, where $A = \{c\}$, from Example 13 coincide with the positive counterparts of $P$ and $Q$ proven visibly strongly equivalent in Example 12.

<table>
<thead>
<tr>
<th>$P$</th>
<th>$Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a :- #\text{count}{b:b; \ c: \not c} \geq 1$</td>
<td>$a :- d_{1,1}$.</td>
</tr>
<tr>
<td>$d_{1,1} :- d_{1,2}$.</td>
<td>$d_{1,2} :- \not c$.</td>
</tr>
<tr>
<td>$d_{1,1} :- b$.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{Tr}_{\equiv}(P,A)$</th>
<th>$\text{Tr}_{\equiv}(Q,A)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a :- #\text{count}{b:b; \ c: c^-} \geq 1$</td>
<td>$a :- d_{1,1}$.</td>
</tr>
<tr>
<td>$d_{1,1} :- d_{1,2}$.</td>
<td>$d_{1,2} :- c^-$.</td>
</tr>
<tr>
<td>$d_{1,1} :- b$.</td>
<td></td>
</tr>
</tbody>
</table>

Hence, the analysis of hidden minimal models from Example 12 applies to $\text{Tr}_{\equiv}(P,A)$ and $\text{Tr}_{\equiv}(Q,A)$, yielding the result $\text{MM}_h(\text{Tr}_{\equiv}(P,A)) \equiv_v \text{MM}_h(\text{Tr}_{\equiv}(Q,A))$. Given this result, Proposition 1 implies visible strong equivalence $P \equiv_{vs} Q$. Notably, the analysis and the proposition involve no VSE-models. In this way, the convenience of reasoning by hidden minimal models can be extended to non-positive programs.

The converse of Proposition 1 does not hold in general, and therefore the result is not relevant to proving $\equiv_{vs}$ in every possible case where it holds. Nevertheless, the result does apply to choice-rule-free programs that use negation only on visible atoms that do not occur positively in the program. This class of programs is restricted, but does prominently include most single-rule programs and translations of them that do not introduce negation nor choice rules. Hence the result applies to establishing the correctness of important nontrivial normalizations, as demonstrated in the sequel.
3.7 Proof of Correctness of a Cardinality Rule Normalization

Visible strong equivalence is particularly suitable for establishing the correctness of normalizations also beyond the illustrative examples shown so far. For this purpose, a normalization is considered correct if it always maps input rules to visibly strongly equivalent output programs. To this end, the counting grid normalization of cardinality rules from Definition 6 is used as an example of a real-life normalization scheme of practical relevance. The normalization was already proven correct in Examples 11 and 12 for specific instances of cardinality rules. In contrast, this section concerns cardinality rules with any bound and any number of body literals. The results are obtained via the formal tools from Section 3.6. Publication II includes analogous results for choice rules as well.

The correctness of the normalization in general, can be conveniently approached by first studying the special case of positive cardinality rules.

**Lemma 4** (Lemma 5.4 from Publication II). Let $P$ be a program consisting of a positive cardinality rule

$$\texttt{#count}\{1:b_1; \ldots; n:b_n\} \geq k$$

and $Q$ its counting grid normalization. Then $\text{MM}_h(P) =_v \text{MM}_h(Q)$.

The proof maps models of $P$ bijectively to interpretations and shows the interpretations to be models of $Q$ by complete induction. Moreover, the proof shows that the mapping takes hidden minimal models of $P$ to hidden minimal models of $Q$, again by complete induction. It then repeats this in the other direction from $Q$ to $P$.

Lemma 4 and Corollary 2 imply the correctness of the counting grid normalization for positive cardinality rules. Indeed, from the result $\text{MM}_h(P) =_v \text{MM}_h(Q)$ of the lemma the corollary yields $P \equiv_{vs} Q$. Next, the result is extended to the general case covering also non-positive cardinality rules.

**Proposition 2** (Proposition 5.6 from Publication II). Let $P$ be a program consisting of a cardinality rule

$$\texttt{#count}\{1:l_1; \ldots; n:l_n\} \geq k,$$

in which no literal occurs with both polarities, and $Q$ its counting grid normalization. Then $P \equiv_{vs} Q$.

The proof is based on the observation that the negation substitution translation of the normalized program $Q$ can also be obtained by normalizing the negation substitution translation of $P$. The negation substitution translation ensures that the input and output programs of that normalization process are positive, and hence by Lemma 4 they have visibly equal hidden minimal models. The final result then follows by the results of Section 3.6.

Proposition 2 ensures that the counting grid translation can be used to normalize cardinality rules in all contexts that mutually respect the hidden atoms involved.
Normalization and Visible Strong Equivalence

Figure 3.2. Verification time taken to prove \( P \equiv_{VSE} Q \) for various alternative representations \( P \) and \( Q \) of a cardinality rule, obtained by the translations \( \text{id, grid, LP2NORMAL, and PBTRANSLATE} \). The verification process is directional here: the results in a single cell concern the search for VSE models of \( P \) with no counterpart in the VSE models of \( Q \), where \( P \) and \( Q \) are normalizations obtained via the translations indicated for the relevant row and column, respectively. Each cell shows contour lines for the CPU times 0.25, 1, 4, 16, 64 in seconds in a two-dimensional space parameterized by the fractional bound \( k/n \) and the number of literals \( n \). In short, the larger the “mountain”, the higher the CPU time.

3.8 Experimental Evaluation of Automated Verification

The preceding analysis of cardinality rule normalization is complemented here with a presentation of experimental results obtained by applying the automated verification approach of Section 3.4. By considering several instances of cardinality rule normalizations, the experiments provide a view of the scalability of the associated tools. Moreover, the experiments confirm the formally established correctness of the counting grid normalization scheme and its implementation for a range of input parameters.

The results, reported in Figure 3.2, concern the counting grid translation from Definition 6 as well as more advanced normalizations from the tools \( \text{LP2NORMAL} \) and \( \text{PBTRANSLATE} \). To this end, these tools were used with their default settings. In accordance with these defaults, \( \text{LP2NORMAL} \) heuristically mixed parts of
Normalization and Visible Strong Equivalence

several translations. Moreover, PBTRANSLATE used dynamic programming to pick structural parameters for Batcher’s odd-even merge sorters [19] and a library of small precomputed sorters. To gain additional insight into the verification process, all of these translations were cross-tested against one another in addition to being verified against the cardinality rule itself, denoted in the results by id. These tools employ combinations of various translations enhanced by intricate micro optimizations to reduce translation size. A complete formal account of their correctness is therefore beyond reasonable approach, as is usual with nontrivial software projects. Such complications do not hinder the translation based verification approach from Section 3.4, however.

The results indicate that the proposed automated verification approach scales without difficulty to cardinality rules with up to 20 literals, for all bounds and all of the considered translation combinations. Between the actual translations, performance interestingly scales significantly beyond this, and cardinality rules with 30 literals and more are within reach of verification. This discrepancy in performance may be explained by several factors. First, the translations may share some equivalent auxiliary atoms, and this may help the search. This may be the reason why particularly the grid and PBTRANSLATE normalizations are easy to verify against one another: they involve some number of similarly defined atoms. Another additional factor is that normalization techniques that add auxiliary atoms can reduce the number of nogoods that the solver needs to learn in order to handle the constraint by native propagation based means. This is particularly the case for small but hard instances where the constraint in question has a prominent role, such as is the case here as well as with “bottleneck” constraints [2] from the areas of SAT and SMT. A more detailed discussion of this topic is given in Chapter 5, where an analogous phenomenon is studied in the context of optimization.

3.9 Related work

The defining characteristics of VSE are its insistence on a bijective correspondence of answer sets, its focus on only visible atoms in the correspondence, and its consideration of context programs that mutually respect the hidden atoms of the compared programs. This is a unique combination of ways to address both visibility and context in a single relation. Namely, regarding existing notions, the visible equivalence recalled in Section 2.4 addresses visibility but not context [66]. Moreover, strong equivalence addresses context but not visibility nor the restriction of context programs [80]. In contrast, relativized strong equivalence [112] promisingly addresses both context and visibility, but it does this by restricting only context programs and not the atoms under comparison. Indeed, relativized strong equivalence allows context programs $R$ to use only a fixed set $A$ of atoms. The remaining atoms are thus left hidden from context programs, but not from answer set comparisons. To our knowledge the closest alternative
Normalization and Visible Strong Equivalence

to VSE is represented by the general framework of correspondence frames [43]. The framework is a flexibly parameterized representation of answer set program comparisons and equivalence tests. With the appropriate parameterization for a given pair of programs, the framework can be used to instantiate even \( \equiv_{\text{va}} \). Nevertheless, existing formal results for the framework concern only comparisons where sets of answer sets are projected onto a fixed vocabulary \( B \) and then checked for subset inclusion or equality as in \( \text{AS}(P \cup R)|_B = \text{AS}(Q \cup R)|_B \) [43]. In particular, the bijective approach of \( \equiv_{\text{va}} \) is not covered by the existing results and instead these projective equality tests are indifferent to multiplicities of answer sets. Multiplicities can be used, for example, in certain probabilistic settings to capture event frequencies [20].
4. First-Order Lazy-Normalization for Lazy-Grounding

This chapter discusses the normalization of aggregates to serve lazy-grounding and summarizes research findings from Publication III. Lazy-grounding is a gradually developing alternative solving approach for ASP with the potential to tackle problems of a scale unprecedented for ASP. The idea in lazy-grounding is to interleave grounding and solving steps so that information learned during solving can be used to guide grounding and particularly to avoid unnecessary grounding. A lazy-grounder can delay the generation of a ground instance of a first-order rule until arriving in a part of the search space where that particular rule instance fires. Such avoidance of grounding may help solve problems encoded as ASP programs for which full grounding is a bottleneck. Removal of such bottlenecks improves the extent to which the expressivity of ASP realizes in practice. Certainly, as a language, ASP is already tremendously expressive by allowing to capture broad classes of problems. As a practical problem-solving methodology, however, ASP allows to solve only some of those problems and mainly ones not inducing grounding bottlenecks. From this purely practical perspective, the expressivity of ASP effectively depends on performance; thus, significant enough increases in performance, as sought by lazy-grounding for example, can be considered to improve such, say, effective expressivity. On the topic of the expressivity of ASP, a notable part of it stems from aggregates such as those in cardinality and weight rules. These conditions have proven to be integral in ASP research [8, 45, 104]. As an example of research outcomes on aggregates, aggregates can be compiled into normal rules on the ground level via normalization, as discussed in the context of equivalence checking in Chapter 3. In other work on the ground level, aggregates have been implemented also natively in ASP solvers [51, 104]. Furthermore, first-order normalizations of aggregates have been developed; specifically, for debugging answer-set programs [100].

While both lazy-grounding and aggregates are well-established topics of ASP research, no prior work studies their combination. Indeed, aggregates are not supported by existing lazy-grounding systems, which so far include GASP [97], ASPeRiX [76], Omiga [35], and Alpha [110]. Moreover, normalization techniques from the ground level are not directly applicable in a lazy-grounding
First-Order Lazy-Normalization for Lazy-Grounding

Table 4.1. Different ways to think of or implement the novel concept of lazy normalization.

<table>
<thead>
<tr>
<th>Lazy Normalization</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i) Conceptually:</td>
<td>First-Order Normalization + Lazy Grounding</td>
</tr>
<tr>
<td>ii) In ALPHA:</td>
<td>First-Order Normalization + Lazy Grounding extended with enum</td>
</tr>
<tr>
<td>iii) Alternatively:</td>
<td>First-Order Normalizations from [100] + Lazy Grounding</td>
</tr>
<tr>
<td>iv) Alternatively:</td>
<td>Lazy Grounding extended with native aggregate handling</td>
</tr>
</tbody>
</table>

setting on the first order-level and prior first-order normalizations [100] are not asymptotically as compact as would be desirable.

Therefore, this work extends and combines the separate lines of work on lazy-grounding and aggregates by proposing a novel framework for first-order lazy-normalization. The framework has been designed specifically to enable concise normalizations that do not compromise the laziness of lazy-grounding. Moreover, the use of normalization helps in the implementation of the framework by hiding the complexity of aggregates from the sophisticated internals of lazy-grounder.

Specific lazy normalizations are developed in Publication III for monotone count and sum aggregates, demonstrating the success and potential of the framework. In the rest of this chapter, the novel concept of lazy normalization is presented in Section 4.1 along with a high-level discussion of the involved problems, proposed solutions, and alternative solutions. Then, Section 4.2 summarizes the results of an experimental evaluation of the efficiency of the proposed solutions. In the results, the proposed framework for lazy normalization shows scaling superior to ground-and-solve approaches.

### 4.1 Lazy Normalization

This section describes the concept of lazy normalization introduced in Publication III on a high level of abstraction. For more details, the reader is referred to the publication itself, which presents a formal framework for lazy-grounding first-order normalization accompanied by detailed encodings compatible with the framework.

As summarized in Table 4.1, lazy normalization conceptually refers to (i) the application of first-order normalization to an input program prior to solving the program via lazy-grounding. Because normalization generates only normal rules, the lazy-grounder needs to support nothing else and, nevertheless, the combined system will support aggregates. While conceptually this is simple, specific practical challenges motivated (ii) a more refined approach developed in Publication
First-Order Lazy-Normalization for Lazy-Grounding

III and implemented in the lazy-grounder ALPHA. To tackle these challenges, discussed more in the following, the approach introduces a novel enumeration built-in provided by the lazy-grounder for use in normalization. This approach was chosen over two alternative options. First, the lazy normalization concept could also be realized by (iii) combining existing first-order normalizations by Polleres et. al (2013) with existing lazy-grounders for normal rules. These prior normalizations are asymptotically large, however, and it is unclear whether more practical and compact normalizations are possible without an innovation such as the enumeration built-in. Second, aggregate support could be implemented without lazy-normalization by instead developing (iv) native handling for aggregates within a lazy-grounder. Developing such native support might require extensive effort to achieve laziness and high performance, both of which are readily obtained via lazy normalization. Therefore, native alternatives were left for potential future work.

A series of examples are given in the sequel that illustrate the challenges with first-order programs and normalization.

Example 15. Let us consider the simple task of selecting a set $S$ of up to four names from dictionary of animal names $D = \{\text{aardvark}, \ldots, \text{zorse}\}$. The task can be encoded and grounded in ASP as follows.

<table>
<thead>
<tr>
<th>First-order Encoding $P$</th>
<th>Grounding of $P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d(\text{aardvark})$.</td>
<td>$d(\text{aardvark})$.</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$d(\text{zorse})$.</td>
<td>$d(\text{zorse})$.</td>
</tr>
<tr>
<td>${s(X)} : \cdot d(X)$.</td>
<td>${s(\text{aardvark})} : \cdot d(\text{aardvark})$.</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>${s(\text{zorse})} : \cdot d(\text{zorse})$.</td>
<td></td>
</tr>
<tr>
<td>$:- #\text{count}(X:s(X)) &gt;= 5$.</td>
<td>$:- #\text{count}(\text{aardvark}:s(\text{aardvark})$; $\vdots$ $\text{zorse}:s(\text{zorse})) &gt;= 5$.</td>
</tr>
</tbody>
</table>

While the number of distinct animal names is moderate, larger dictionaries for entire languages would pose challenges for the efficient normalization of the ground version.

The practical challenges faced by lazy normalization relate to the fact that typical normalizations from the ground level are troublesome for lazy-grounding. Specifically, typical normalizations depend on a total order of their $n$ input literals. For example, sorting network based encodings may begin by mutually sorting element number $i$ with element number $i + 1$ for every odd $i \in \{1, \ldots, n\}$. As another example, counting grid style encodings operate by inductively counting the numbers of true elements among the first $i$ elements, then among the
first \(i+1\) elements, and so on. On the ground level, the elements can be easily ordered up front to associate indices to elements. Upfront ordering is possible also on the first-order level; for example, the natural ordering of elements provided as \(<\) in ASP-Core-2 [29] can be used to assign indices to elements.

**Example 16.** To see how extensive input ordering is possible in encodings, and to understand why it is problematic in lazy-grounding, let us recall the animal dictionary domain \(D\) captured with the predicate \(d(X)\) in Example 15. For illustration, the following encoding \(E\) defines a predicate, \(\text{map}(X,Y)\), which relates animals \(x \in D\) to indices \(i \in \{1, \ldots, |D|\}\) in ASP using \(<\):

\[
\begin{align*}
c(X,1) & \ :- \ d(X). \\
c(X,I+1) & \ :- \ c(X,I),r(X,I). \\
r(X,I) & \ :- \ Y < X, c(X,I),c(Y,I). \\
\text{map}(X,I) & \ :- \ c(X,I),\text{not} \ r(X,I).
\end{align*}
\]

In this encoding, predicate \(c(X,I)\) identifies candidate assignments and predicate \(r(X,I)\) rejects assignments misaligned with \(<\). By expressing the concept of the \(i\)-th element in ASP, this encoding facilitates the adaptation of normalization designs from the ground level. Despite its benefits, this encoding comes at the costs of yielding \(O(|D|^2)\) ground atoms and \(O(|D|^3)\) ground rules, and being laborious even for a lazy-grounder. A lazy-grounder will have trouble avoiding grounding \(E\), because generally large parts of \(E\) contribute to the definitions of even individual ground instances \(\text{map}(x,i)\). For example, in the corner case \(x = \max D\), the entire encoding needs to be grounded before \(\text{map}(x,|D|)\) can be derived. Finally, in a broader context, the domain of aggregate inputs may be defined with a nontrivial subprogram that needs to be fully grounded for the domain to be fully accessible.

Enumeration of large domains, such as \(D\) in the examples, may force a lazy-grounder to generate substantial groundings to reason about even small subsets of inputs, such as candidates for \(S\) in the examples. Example 16 already illustrated such degeneration towards a full grounding, but another risk of degeneration remains even assuming that term ordering is given a priori. To understand this risk, let us consider a normalization based on a full enumeration of a domain such as \(D\). Then, a lazy-grounder may have to ground a small fraction of \(N\) to reason about a small subset of inputs. The problem here is that even a small fraction of such a normalization \(N\) for a full enumeration of \(D\) can be meaningfully larger than a normalization for a small fraction of \(D\). For instance, a standard normalization for \(D\) based on Batcher’s odd-even networks has a depth of \(O(\log^2 |D|)\) and yields a grounding of size \(O(|D|\log^2 |D|)\) of which a portion of size \(O(\log^2 |D|)\) fires when any constant number of input elements fire. Such a firing portion will unavoidably be relevant to a lazy-grounder. In comparison, a normalization for any constant-size domain using the same networks is trivially of size \(O(1)\), resulting in a large proportional difference of \(O(\log^2 |D|)\).
Instead of relying on a full enumeration of an input domain, a lazy-normalization ideally minimizes the number of ground rule instances that fire as it explores different parts of the search space. To that end, \(<\) can be used to order only satisfied elements. This is used in the following example to demonstrate how such an ordering straightforwardly extends to an entire first-order normalization:

**Example 17.** Ordering of only satisfied elements is problematic but possible by, for example, replacing \(d(X)\) with \(s(X)\) in the encoding \(E\) from Example 16. Let us denote by \(E'\) the resulting encoding. Then, the first-order rule with a count aggregate from Example 15, say \(r\), can be straightforwardly transformed into another rule \(r'\) for normalization as follows.

<table>
<thead>
<tr>
<th>Original Rule (r)</th>
<th>Transformed Rule (r')</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{:-} #\text{count}(X:s(X)) \geq 5).</td>
<td>(\text{:-} \text{map}(X,5)).</td>
</tr>
</tbody>
</table>

Technically, the above completes a first-order normalization of \(r\) given by \(E' \cup \{r'\}\). The only problem with this normalization is the size of \(E'\), which is \(O(n^2)\) ground atoms and \(O(n^3)\) ground rules where \(n\) is the number of input terms \(X\) stemming from \(s(X)\) that have been grounded so far in the lazy-grounding process.

The above example shows one approach for accessing \(<\) without constructing a full enumeration and thus solving the ordering problem in lazy-grounding. It is not the first such solution, however, as Polleres et al. (2013) [100] give a similar encoding (\(tr_{opt}\) in [100]) for cardinality rules whose closest prior counterpart is the counting grid [69,104] recalled in Section 3.1. Both encodings follow an inductive scheme in building sub-results associated with input terms. Where these encodings differ is that whereas the counting grid resembles weak induction in that it builds those sub-results using the sub-results of immediate predecessor terms, the construction of Polleres et al. is larger and resembles complete induction in that it builds sub-results using the sub-results of all lesser terms. Use of the immediate predecessor relation as in the counting grid is problematic in lazy-grounding because resolving it over the entire input or the terms of only satisfied inputs may force a lazy-grounder to ground the entire set of inputs or generally vast parts of it, respectively. On the other hand, the larger encoding is favorable in lazy-grounding, because it allows the lazy-grounder to easily instantiate only a relevant part of the encoding when some potentially small number of inputs have been satisfied so far. The downside of the encoding is that whereas the counting grid is of size \(O(nk)\), the encoding of Polleres et al. (2013) is of size \(\Omega(n^2k)\).

In the work reported here, the ordering problem is solved instead by introducing the enumeration built-in \texttt{enum} for use in normalization schemes and to be provided by lazy-grounders. This solution exploits the insight that typical normalizations that require a total order of inputs are oblivious to the particular choice of an order. Hence, Publication III seizes this opportunity and follows whichever order the used lazy-grounder ends up grounding input terms in; this
is precisely the order that best facilitates truly lazy normalization. More specifically, enum provides a mapping from terms $t$ to indices $i$ that are unique within a context identified by a given term $ag$. While enum can be used in many contexts, here ag refers to individual first-order aggregates.

**Definition 11** (Publication III). An enumeration built-in is an atom 
$$\text{enum}(ag, t, i),$$
where the term $ag$ identifies an aggregate with a grounding order, $t$ is a ground term, and $i$ is the index of $t$ in the order.

By using enum, the publication provides a framework for lazy normalization where enum serves a key component in connecting the input terms of aggregates to normalization schemes. In the design of the framework, the normalizations take their input in the form of indexed input atoms. These indices enable the adoption of normalization schemes from the ground level to the first-order level. In that sense, the enumeration built-in serves as a dynamic analogue to the predicate $\text{map}(X, I)$ from Example 16 that allows to sidestep the grounding issues with the predicate. In particular, the framework solves the asymptotic size problem with the example and with prior first-order encodings by enabling the use of an $O(nk)$-size counting grid encoding as well as an $O(n(\log n)^2)$-size sorting network encoding for count aggregates or cardinality rules. Moreover, for sum aggregates or weight rules an $O(nk)$-size construction is enabled.

To fully realize the potential of the framework for lazy normalization, it is necessary to specify concise first-order encodings of count and sum aggregates. By virtue of the framework, the input can be accessed via indices, but nevertheless, the encodings themselves should also be crafted with the context of lazy normalization in mind. Using counting grid style encodings, this is natural. Moreover, to obtain even more concise encodings, Publication III introduces a sophisticated encoding of sorting networks, which is carefully tailored for lazy use. The encoding consists of two parts: one part defining a single sorting network able to grow in size to accommodate any count aggregate and another part that evaluates every count aggregate in isolation using a relevant subset of the sorting network. To enable this, a recursive sorting network scheme is used, so that well-defined parts of the network constitute smaller sorting networks sufficient for smaller aggregates. With the help of its recursive structure, as the sorting network grows to accommodate more input, its pre-growth version becomes part of its post-growth version. For sum aggregates, it is likewise challenging to create a sorting-network based encoding with similar properties and at the time such an encoding was left for potential future work.

### 4.2 Experimental Evaluation

This section covers experiments evaluating the performance level and feasibility of lazy normalization as a newly-implemented component of the lazy-grounding
First-Order Lazy-Normalization for Lazy-Grounding

ASP solver ALPHA. Given the novelty of aggregate support in lazy-grounding, no prior lazy-grounder is suitable for the comparison. Thus, the evaluation instead includes the well-known state-of-the-art ground-and-solve systems CLINGO and DLV2. Unfortunately, the fundamental differences between lazy-grounding and ground-and-solve systems make it harder to analyze the results specifically for the impact of lazy normalization. Nevertheless, the experiments do provide value in three ways. First, they shed light on the efficacy of lazy normalization and the still heavily developing current state of lazy-grounding against advanced and relatively mature ground-and-solve systems. Second, the results test the achievement of the primary goal of lazy-normalization, which is to enable support of aggregates in lazy-grounding in a truly lazy manner. Finally, the results give a baseline to compare any future implementations against. The novelty of aggregate support in lazy-grounding also shows in the lack of a well-established benchmark set for assessing the techniques; in particular, the traditional ASP Competitions explicitly exclude problems involving grounding issues [56].

Results of the evaluation are shown in Table 4.2, which for the above reasons consists of four novel benchmark classes. First, Simulation is a rule based simulation of a robot moving along the edges of a graph and with certain restrictions on its movement. As a particular challenge, the simulation poses a large number of possible paths to travel, which are limited in length with a cardinality constraint. In the results, this challenge proves unmanageable for the ground-and-solve systems as the graph grows, while the lazy-grounder ALPHA scales efficiently to larger graphs. The next benchmark, Summation, is a grounding stress-test with a large sum aggregate over a domain with non-deterministically chosen elements so that a trivial grounding is impossible. Handling the sum aggregate via lazy-normalization is straightforward as it avoids a grounding blow-up that impedes CLINGO. Numbers of DLV2 are skipped, because it did not compute correct answer sets for this benchmark. Furthermore, Dynamic Indegree Counting is a graph-related benchmark that tests the computation of indegrees for all nodes of a dynamically constructed graph using count aggregates. Such a computation could be involved in more complex problems, but is assessed here in isolation for the ease of analysis. Interestingly, both ALPHA and DLV2 perform well here, while CLINGO alone runs into timeouts on the larger instances. Finally, Exponential Space Saving is a benchmark that brings out an exponential difference in the space requirements between lazy-grounding lazy-normalization and ground-and-solve. For this benchmark, the number of variables in an aggregate has little impact on the performance of ALPHA on several instances large enough to cause the ground-and-solve systems to run out of memory.
Table 4.2. Runtime results in seconds on four benchmarks averaged over 10 runs per instance. Parentheses indicate the numbers of timeouts. Size is the number of time steps on the Simulation benchmark, the domain size for selected numbers on Summation, the approximate number of edges on Dynamic Indegree Counting, and the number of variables in the aggregate on Exponential Space Saving. Details refer to the number of vertices and the number of edges as a share of the maximum. Timeouts of 300 s and 900 s and memory limits of 8 GB and 40 GB are imposed on the first three benchmarks and the last one, respectively.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size</th>
<th>ALPHA</th>
<th>CLINGO</th>
<th>DLV2</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulation</td>
<td>100</td>
<td>2.6</td>
<td>0.8</td>
<td>4.7</td>
<td></td>
</tr>
<tr>
<td>Simulation</td>
<td>200</td>
<td>3.2</td>
<td>6.2</td>
<td>100.1(3)</td>
<td></td>
</tr>
<tr>
<td>Simulation</td>
<td>500</td>
<td>9.1</td>
<td>110.5</td>
<td>memout</td>
<td></td>
</tr>
<tr>
<td>Simulation</td>
<td>600</td>
<td>7.7</td>
<td>memout</td>
<td>memout</td>
<td></td>
</tr>
<tr>
<td>Simulation</td>
<td>800</td>
<td>27.8</td>
<td>memout</td>
<td>memout</td>
<td></td>
</tr>
<tr>
<td>Simulation</td>
<td>1,000</td>
<td>47.4(1)</td>
<td>memout</td>
<td>memout</td>
<td></td>
</tr>
<tr>
<td>Simulation</td>
<td>2,000</td>
<td>86.8(1)</td>
<td>memout</td>
<td>memout</td>
<td></td>
</tr>
<tr>
<td>Summation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Summation</td>
<td>100</td>
<td>1.9</td>
<td>1.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Summation</td>
<td>150</td>
<td>2.2</td>
<td>5.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Summation</td>
<td>200</td>
<td>2.5</td>
<td>15.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Summation</td>
<td>250</td>
<td>2.9</td>
<td>37.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Summation</td>
<td>300</td>
<td>3.0</td>
<td>96.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Summation</td>
<td>350</td>
<td>3.2</td>
<td>247.0(3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Summation</td>
<td>1,000</td>
<td>6.5</td>
<td>300.0(10)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic</td>
<td>1,000</td>
<td>4.4</td>
<td>1.5</td>
<td>2.5</td>
<td>100/10</td>
</tr>
<tr>
<td>Dynamic</td>
<td>3,000</td>
<td>13.0</td>
<td>13.7</td>
<td>8.1</td>
<td>100/30</td>
</tr>
<tr>
<td>Dynamic</td>
<td>4,000</td>
<td>11.3</td>
<td>25.1</td>
<td>4.0</td>
<td>200/10</td>
</tr>
<tr>
<td>Dynamic</td>
<td>5,000</td>
<td>18.7</td>
<td>38.9</td>
<td>3.4</td>
<td>100/50</td>
</tr>
<tr>
<td>Dynamic</td>
<td>6,000</td>
<td>13.9</td>
<td>62.7</td>
<td>6.1</td>
<td>250/10</td>
</tr>
<tr>
<td>Dynamic</td>
<td>12,000</td>
<td>36.5</td>
<td>271.8(1)</td>
<td>9.6</td>
<td>200/30</td>
</tr>
<tr>
<td>Dynamic</td>
<td>25,000</td>
<td>47.9</td>
<td>300.0(10)</td>
<td>39.3</td>
<td>500/10</td>
</tr>
<tr>
<td>Exponential</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exponential</td>
<td>10</td>
<td>0.8</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>Exponential</td>
<td>18</td>
<td>0.8</td>
<td>0.8</td>
<td>249.3</td>
<td></td>
</tr>
<tr>
<td>Exponential</td>
<td>20</td>
<td>0.9</td>
<td>3.3</td>
<td>900.0</td>
<td></td>
</tr>
<tr>
<td>Exponential</td>
<td>22</td>
<td>0.8</td>
<td>14.7</td>
<td>900.0</td>
<td></td>
</tr>
<tr>
<td>Exponential</td>
<td>24</td>
<td>0.9</td>
<td>63.7</td>
<td>900.0</td>
<td></td>
</tr>
<tr>
<td>Exponential</td>
<td>26</td>
<td>0.9</td>
<td>370.0</td>
<td>memout</td>
<td></td>
</tr>
<tr>
<td>Exponential</td>
<td>28</td>
<td>0.9</td>
<td>memout</td>
<td>memout</td>
<td></td>
</tr>
<tr>
<td>Exponential</td>
<td>500</td>
<td>12.3</td>
<td>memout</td>
<td>memout</td>
<td></td>
</tr>
<tr>
<td>Exponential</td>
<td>1,000</td>
<td>80.0</td>
<td>memout</td>
<td>memout</td>
<td></td>
</tr>
</tbody>
</table>
5. Optimization Rewriting

This chapter reviews a novel class of preprocessing techniques for ASP optimization presented in Publication IV and studied further in Publication V. A central issue in ASP optimization is the practical performance level of optimizing ASP solvers. The pursuit of high performance is challenged by the complexity of reasoning about optimal answer sets; determining whether a given atom is true in some optimal answer set is known to be in $\Delta^P_2$ [104] and $\Delta^P_3$ [78] in the absence and presence of disjunctive rules, respectively. Nevertheless, there is clear evidence that acceptable performance can be often achieved on many optimization problems, particularly with the use of a highly developed optimization algorithm appropriately chosen for the problem at hand [4, 50, 102]. Specifically, Gebser et al. (2015) [50] reported that certain problems are amenable to model-guided optimization, and certain other problems to core-guided optimization, which are optimization strategies overviewed in Section 1.2.4. Furthermore, Alviano and Dodaro (2016) [4] demonstrated that a practically important advantage of model-guided optimization can be incorporated into core-guided optimization as well: anytime behavior. An anytime optimization algorithm reports non-optimal solutions of increasing quality until an optimum solution is found, as opposed to staying silent until then. This increases the practicality of ASP optimization for problems whose optimum solutions are out of reach. Moreover, Saikko et al. (2018) [102] proposed an additional optimization strategy, adapted from MaxSAT [36] and based on core-guided solving and the implicit hitting set (IHS) problem [92]. The development of these different strategies has promisingly increased the optimization performance of available ASP technology.

The development of ASP optimization techniques has thus been well motivated in past research: by the practical importance of solving performance, by the challenge posed by complexity results, and by opportunities to make improvements. Despite this attention given to optimization algorithms, there has been no research to our knowledge in the ASP field on performance-enhancing preprocessing techniques for ASP optimization statements or weak constraints. In particular, no techniques similar to normalization of aggregates have been proposed for ASP optimization criteria. Given that normalization holds potential to improve performance for certain programs with aggregates, as shown in
Publication VI and Publication VII, it is warranted to study the possibility for similar improvements in the case of optimization.

To that end, this chapter presents optimization rewriting, a novel transformation approach aimed at increasing optimization performance. In the approach, performance improvements are sought via normalization-inspired transformation techniques. The direct application of normalization to optimization statements is, however, problematic. Indeed, on the one hand, the function of traditional normalization is to entirely replace some rule types of interest without changing the program semantically. This is impossible in the case of optimization statements, which associate values to answer sets, whereas normal rules do not. On the other hand, it is possible to write a program of normal rules whose answer sets coincide with the optimal answer sets of an optimization program. This is made generally impractical due to the fact that optimization statements elevate the computational complexity of ASP solving from \( \Delta^P_2 \) in the absence of disjunctive rules [104], and from \( \Sigma^P_2 \) to \( \Delta^P_3 \) in the presence of disjunctive rules [78]. Since normal rules are not equally expressive, a reduction from optimization statements to normal rules would be impractically large. For these reasons, optimization rewriting adds normal rules, as in normalization, but instead of removing optimization statements, modifies them. To make these additions and modifications, several designs are proposed in this work. A particular challenge in this design task is posed by the often significantly larger size of optimization statements compared to aggregates, which demands a greater focus on the scalability of optimization rewriting techniques. Moreover, the question of how optimization rewriting can generate these improvements is a complicated matter as it concerns the interplay between ASP programs and solvers. Indeed, performance depends on the implementation of ASP solvers and particularly on the sensitivity of ASP solvers to different syntactic representations of the same semantic concepts.

As main outcomes of the research, up to exponential performance gains are proven possible in a formal performance analysis and significant practical gains are found in extensive computational experiments. Results obtained from the formal analysis concern ASP-solvers performing branch-and-bound optimization with the help of propagators on a specific family of programs. More specifically, the analysis focuses on an abstract measure of the minimum work required to use propagators in such optimization. The exponential gains in this measure demonstrate that while optimization rewriting increases initial problem size, it may significantly reduce the size and number of nogoods learned by an optimizing ASP solver from a propagator. This result is enabled specifically by the atoms provided in optimization rewriting, which can help a solver concisely express these nogoods. As evident in the analysis, these reductions are enabled by the atoms introduced by the analyzed optimization rewriting technique. On the experimental side, the obtained improvements are particularly strong and easily obtained on simple optimization statements with limited numbers of distinct weights per statement, even when applying straightforward optimization rewriting.
Optimization Rewriting

ing techniques. In the research process, more complex optimization statements proved to demand more refined techniques. Nevertheless, several such refined techniques were found and then demonstrated to be efficient in the experiments as well.

The rest of this chapter is organized as follows. First, Sections 5.1, 5.2, and 5.3 respectively present, formally analyze, and experimentally evaluate optimization rewriting techniques for statements with uniform weights. Finally, Section 5.5 presents the results of an experimental evaluation of techniques for heterogeneous weights from both Publication IV and Publication V. To enable this combined evaluation, the techniques from Publication IV were evaluated anew for this dissertation in a benchmark setup identical to the one used in Publication V.

5.1 The Case of Uniform Weights

This section addresses the problem of designing correct optimization rewriting techniques for optimization statements (2.6) with specifically uniform weights $w_1 = \cdots = w_n$. Uniformly weighted optimization statements form a relatively simple type of optimization statements; indeed, they are essentially unweighted, and their weights can be assigned to one.

To obtain a concrete idea of optimization rewriting, consider the following example:

Example 18. Let us take a simple optimization statement:

\[
\text{#minimize}(1,a_1:a_1; 1,a_2:a_2).
\]

One possible optimization rewriting of this statement would be:

\[
b_1 :- a_1, a_2.
\]
\[
b_2 :- a_1.
\]
\[
b_2 :- a_2.
\]
\[
\text{#minimize}(1,b_1:b_1; 1,b_2:b_2).
\]

Functionally, the obtained program is identical to the original one. Namely, careful inspection of the rewriting reveals that it preserves optimization values—assuming the new atoms $b_1$ and $b_2$ do not appear elsewhere in the program. This stems from the definitions of the new atoms $b_1$ and $b_2$: the definitions guarantee that equally many of $b_1$ and $b_2$ are satisfied as of $a_1$ and $a_2$.

While Example 18 is a small example shown only for illustration purposes, the general approach in optimization rewriting is the same: some or all of the originally optimized atoms are replaced by newly defined atoms that keep optimization values unchanged.

69
In the problem of designing a general scheme for optimization rewriting, the goal is to provide a program transformation that preserves answer sets up to visible atoms, preserves optimization values, and increases optimization performance in the process. A possible outline of this design task is as follows. First, the preservation of visible atoms can be approached by augmenting the program to be rewritten with new rules and hidden atoms that implement a function of the literals being optimized as a subprogram. Definitions of those hidden atoms can encode the function output, which can then be included in a rewritten version of the optimization statement. The idea is to use new rules solely for this purpose of computing a function, so that the rules do not interfere with the definitions of any existing atoms, the rules do not introduce any non-determinism, and the rules do not alter the number of answer sets. Such use of rules maintains a natural bijective mapping between the answer sets of the original and rewritten programs. When considering specifically uniformly weighted optimization statements, the remaining task is then to preserve counts of satisfied optimized literals. Indeed, an optimization rewriting must ensure that for each bijectively mapped pair of answer sets, the number of literals in the original optimization statement satisfied by the original answer set equals the number of literals in the rewritten optimization satisfied by the answer set of the rewritten program. In the case of more complex statements with heterogeneous weights, the task is naturally to preserve sums of weights in an analogous manner. While adherence to these design steps guarantees the correctness of the resulting optimization rewriting, it remains to consider optimization performance. Some ideas of how to enable performance gains can be obtained from the analysis in Section 5.2.

Summarizing a part of the above discussion, given unit-weight optimization statements, a correct optimization rewriting can be obtained by generating a program that dynamically permutes the literals in the statement and places the result in a new optimization statement. As will be shown in the formal and experimental performance analyses, performance-increasing optimization rewritings that perform such permutation operations can be constructed from sorting programs as described in Definition 5. Different types of general schemes for sorting programs are discussed in Section 3.1 in the context of cardinality rule normalizations. In the context of optimization rewriting, such schemes are readily applicable as long as the programs are not pruned via cone-of-influence simplification described in Section 3.1 or other similar techniques. Regarding examples, the use of sorting programs was in fact already showcased in Example 18 with a program $P$ that essentially encodes a single comparator.

Entire sorting programs are not the only types of permuting programs applicable to optimization, and not always the most desirable. Indeed, they may grow infeasibly large when given large optimization statements, and therefore smaller programs are worth investigating. To this end, Publication IV proposes to combine many small sorting programs instantiated on groups of literals, resulting in permuting program of more scalable size. Given a fixed limit for
group size, optimization rewriting based on such programs reduces in size to $O(n)$ down from $O(n(\log n)^2)$ when using Batcher’s odd-even merge-sorters or similar [19] as building blocks.

5.2 Formal Analysis of Performance Potential

This section presents a formal analysis of the computational effort required to find an optimal answer set without or with optimization rewriting. Publication V contains the original analysis together with complete proofs and more elaborate explanations. To reasonably set the scope of the analysis, an abstract measure of computational effort is used and a fixed family of optimization programs is studied. The abstract measure is relevant to solvers that interact with optimization statements through propagators [39]. The analysis reveals a potential for exponential improvements in terms of this measure.

The family of programs used for the analysis is the following, in its decision and optimization versions:

Definition 12. The binomial program $P^n_k$ consists of the rules $\{x_1; \ldots; x_n\}$ and $:-\#\text{count}\{1:x_1; \ldots; n:x_n\} < k$.

Definition 13. The binomial optimization program $O^n_k$ is the optimization program $\langle P^n_k, 1x_1 + \cdots + 1x_n \rangle$.

These programs are based on sets of cardinality constraints that have been known to cause trouble to propagator based SMT solvers [2]. The answer sets of the decision version $P^n_k$ correspond to the $l$-subsets of $n$ elements for all $l \geq k$. Moreover, the optimal answer sets of the optimization version $\langle P^n_k, 1x_1 + \cdots + 1x_n \rangle$ correspond to exactly the $k$-subsets of $n$ elements, of which there are $\binom{n}{k}$. To solve the optimization program $\langle P^n_k, 1x_1 + \cdots + 1x_n \rangle$ it is necessary to find an answer set of the optimum value $k$ and to prove that no answer set of value less than $k$ exists. Hence, the solving task requires a proof of the unsatisfiability of $P^n_k$ augmented with the integrity constraint $:-\#\text{count}\{1:x_1; \ldots; n:x_n\} >= k$. This proof task is considerable for a branch-and-bound solver, even in a best-case scenario where the optimal answer set is guessed immediately. Thus, when considering best-case computations, the unsatisfiability of the augmented program can be studied instead of the optimization of $\langle P^n_k, 1x_1 + \cdots + 1x_n \rangle$. Indeed, the unsatisfiability proof is studied in the following, both because of this connection between best-case computations and because unsatisfiability proofs are practically important in ASP optimization.

Regarding propagators, the analysis assumes that the solver implements the integrity constraint $:-\#\text{count}\{1:x_1; \ldots; n:x_n\} >= k$ using a propagator. A propagator is basically a function through which the solver can query information on a constraint on demand in the form of nogoods:

Definition 14. A propagator $\pi$ for a constraint $\Gamma$ is a function from partial
assignments $A$ in conflict with $\Gamma$ to nogoods $\delta' \in \Gamma$ in conflict with $A$. The constraint $\Gamma$ of a propagator $\pi$ is denoted by $\Gamma_\pi$.

This definition is slightly different from, e.g., the definition of Drescher and Walsh (2012) in that here the partial assignments $A$ are assumed to be in conflict with the constraint $\Gamma$ of the propagator. Moreover, in the analysis, a propagator is assumed to be the only source of information about the respective constraint available to the solver. Furthermore, the solver is assumed to deduce about the constraint based on only the nogoods it has received so far from the propagator. In the following, this concept of nogoods received so far is formalized indirectly via a propagator call history (PCH), which captures the corresponding sequence of arguments the solver has passed to the propagator. The definition of a PCH also captures the assumption that the solver guarantees that these argument assignments are viable supported model [9] candidates and that they satisfy all nogoods the solver has previously received from the propagator. The supported model candidacy is checked here via a constraint $\Gamma_{\text{supp}}(P)$ defined in Publication V to hold for any extensions of the supported models of a program $P$.

**Definition 15.** A propagator call history (PCH) for an answer set program $P$ and a propagator $\pi$ is a sequence $A_1, \ldots, A_m$ of partial assignments such that for all $1 \leq i \leq m$, the partial assignment $A_i$ satisfies $\Gamma_{\text{supp}}(P)$, $\pi(A_1), \ldots, \pi(A_{i-1})$ and conflicts with $\Gamma_\pi$.

A length $m$ of a PCH provides an abstract measure of the amount of propagator-related work done in a solving process. This measure is particularly interesting at the end of the process, when solving is complete. The solving of an unsatisfiable program $P$ may complete only once a PCH is comprehensive enough to allow the solver to infer the unsatisfiability of $P$. Such a PCH is called complete.

**Definition 16.** Let $P$ be an answer set program and $\pi$ a propagator such that $P \cup \Gamma_\pi$ has no answer sets. A PCH $A_1, \ldots, A_m$ for $P$ and $\pi$ is complete if $P \cup \{\pi(A_1), \ldots, \pi(A_m)\}$ has no answer sets.

The preceding definitions help state the formal results of the analysis, which are provided in Propositions 3 and 4. The propositions concern the effort required, in view of PCH lengths, to solve the binomial program augmented with a conflicting constraint. The difference is that in Proposition 3 the constraint is expressed using the atoms from the binomial program and in Proposition 4 using the atoms of an additional sorting network encoding. Therefore, the propositions reflect solving strategies without and with optimization rewriting. They reveal an exponential difference in effort, as solving without optimization rewriting requires exponentially many calls to a propagator whereas solving with optimization rewriting requires at most a linear number of calls.

**Proposition 3** (Proposition 2 from Publication V). Let $n$ and $k$ be non-negative integers such that $k \leq n$, $\pi$ a propagator for the constraint

$$\{(T_x \mid x \in X) \mid X \subseteq \{x_1, \ldots, x_n\}, |X| = k\},$$

72
and let $A_1, \ldots, A_m$ be a complete PCH for the answer set program $P^n_k$ and $\pi$. Then $m = \binom{n}{k}$.

The proof of Proposition 3, given in Publication V, is based on the observation that $P^n_k$ is consistent with every proper subset of the nogoods that make up the considered constraint. Hence, the PCH may be complete only once the constraint is fully represented in the form of nogoods, which requires $m = \binom{n}{k}$ of them.

**Proposition 4** (Proposition 3 from Publication V). Let $n$, $k$, and $d$ be non-negative integers such that $k \leq n$, $N$ a sorting network of width $n$ and depth $d$, $\text{ASP}(N)$ the translation of $N$ into an answer set program, $P^n_k$ a binomial program on atoms $x_{10}, \ldots, x_{n0}$, $\pi$ a propagator for the constraint

$$\{(\mathbf{T}x \mid x \in X) \mid X \subseteq \{x_{1d}, \ldots, x_{nd}\}, |X| = k\},$$

and let $A_1, \ldots, A_m$ be a complete PCH for $\text{ASP}(N) \cup P^n_k$ and $\pi$. Then $m \leq n - k + 1$.

The proof of the proposition, provided in Publication V, demonstrates that any subset of nogoods produced by $\pi$ has at most a linear number of nogoods that are logically distinct in the context of $\text{ASP}(N) \cup P^n_k$. This logical distinctiveness is guaranteed for the nogoods returned by $\pi$ when given a complete PCH for $\text{ASP}(N) \cup P^n_k$ and $\pi$. Hence, such a complete PCH is at most of linear length.

In summary, the contrast between Propositions 3 and 4 demonstrates the performance potential of optimization rewriting. This demonstration relies on binomial programs and the fact that the programs themselves encode nearly no problem structure in atoms. Since atoms form the building blocks of nogoods, which are the primary unit of learning for typical answer set solvers, binomial programs provide little means for typical solvers to efficiently learn about the structure of the encoded problem. This lack of efficiency is highlighted by Proposition 3, which shows how binomial programs may force a solver to learn an exponential number of nogoods from a propagator to assert a simple bound. In contrast, simple bounds can be asserted with single nogoods on the output atoms of sorting networks. Indeed, the $k$-th output atom of a sorting network holds true if any $k$-subset of the inputs is all-true. Stated otherwise, those output atoms encode the minimized sum as a *unary number*. Unary numbers are straightforward for solvers to reason about. This relative increase in the strength of nogoods is behind Proposition 4, which shows how there is only a linear number of distinct nogoods that a solver might even possibly learn from a propagator after applying optimization rewriting in a context otherwise identical to the previous proposition. In this way, the analysis reveals exponential performance improvement potential. The analysis then raises the question of how strongly these results apply outside of the given abstract solver model and in the context of more complex optimization problems. This question is explored in the experimental evaluations presented in the sequel.
Table 5.1. Numbers of instances on which different pipelines found and proved optimal solutions. The instances are from benchmark problems with unit-weight optimization statements and Timetabling, on which there are small numbers of distinct weights. The pipelines use optimization rewriting techniques from Publication IV designed for such statements.

<table>
<thead>
<tr>
<th></th>
<th>Connected Still-Life</th>
<th>Crossing Minimization</th>
<th>Maximal Clique</th>
<th>Timetabling</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>120</td>
<td>84</td>
<td>186</td>
<td>57</td>
</tr>
<tr>
<td>clasp</td>
<td>22</td>
<td>50</td>
<td>51</td>
<td>28</td>
</tr>
<tr>
<td>so</td>
<td>50</td>
<td>72</td>
<td>143</td>
<td>36</td>
</tr>
<tr>
<td>64</td>
<td>105</td>
<td>76</td>
<td>153</td>
<td>39</td>
</tr>
</tbody>
</table>

# total number of instances
clasp no optimization rewriting, only CLASP
so rewriting with sorters for literals grouped by weight
64 rewriting with so applied to literals in chunks of 64

5.3 Experimental Evaluation with Homogeneous Weights

This section summarizes the results of an experimental evaluation of selected optimization rewriting techniques from Publication IV on benchmarks with uniformly weighted optimization statements. Given the uniformity of the weights, the rewriting techniques nearly coincide with techniques based on weighted comparator networks from Publication V on these benchmarks, which are thus not separately included here.

Table 5.1 shows the results on three benchmarks from the ASP Competitions [30, 54]: Connected Still-Life, Crossing Minimization, and Maximal Clique; as well as Curriculum based Course Timetabling [16, 25]. The included statements have unit weights \( w_i = 1 \) except on Timetabling, which has a few distinct weights. As for implementation, rewriting is performed with LP2NORMAL (v. 2.27)\(^1\) and answer-set solving with CLASP (3.3.3) [50]. In the results, optimization rewriting increases the numbers of optimally solved instances over plain CLASP using model-guided optimization. This is the case when rewriting with sorting programs for entire groups of equal-weight literals (so), and even more so when rewriting selectively with sorting programs for groups of at most 64 equal-weight literals each (64). As analyzed in more detail in Publication IV, the selective approach 64 reduces the numbers of search conflicts as effectively as the full approach so. Because the selective approach imposes less overhead due to generating smaller rewritings, it emerges as the generally more effective approach.

\(^1\)Available at http://research.ics.aalto.fi/software/asp.
5.4 The Case of Heterogeneous Weights: Weighted Comparator Networks and Alternatives

This section overviews advanced optimization rewriting techniques for optimization statements with potentially large and heterogeneous weights. The overview begins with a discussion of the challenges in this task and continues with a presentation of techniques to address it from Publication IV and Publication V.

A key challenge in rewriting optimization statements with large weights is that rewritten programs can simply grow infeasibly large. Large rewritten programs definitely arise when applying optimization rewriting techniques adopted from weight-rule normalization techniques: existing techniques for weight-rule normalization scale in size proportionally to the magnitude of weights or the number of bits in weights (Publication I). This is natural, since optimization rewriting and weight-rule normalization techniques both aim to represent the information and structure of their input in the form of atoms so that ASP solvers can reason about them efficiently. Notably, the information can be measured in the number of bits in their weights.

Table 5.2 summarizes specific techniques that address the challenge of managing rewriting size via two general approaches from Publication IV and Publication V. First, the approach of Publication IV is to take a complicated, far developed weight-rule normalization technique as a starting point. Specifically, the publication builds on a technique labeled as \textit{mr} in the table, which is based on a combination of sorting and merging networks [19] as components of a larger network [40] optimized via mixed-radix decompositions and structure sharing similar to Publication VII. Beginning from this starting point, the approach aims to manage rewriting size by introducing and applying the general technique of selective rewriting. A selective rewriting technique applies a “full” rewriting such as \textit{mr} to one or many smaller parts of an optimization statement. Whereas the idea is simple and the result is clearly a rewriting of reduced size, the challenge is to pick parts important enough for solving, so that benefits such as those analyzed in Section 5.2 are still obtained. To this end, Publication IV develops a general matrix based method for specifying parts to rewrite. Using the method, the publication expresses a particularly effective selective rewriting technique labeled as \textit{l1, l2, ...} in the table, which partitions the weight of each literal \textit{locally} into two parts. One part contains a configurable number of most significant digits and the other parts contain the rest. This local technique then proceeds to rewrite a statement consisting of literals weighted by the significant parts and then combines the rewritten statement with literals weighted by the less significant parts. For contrast, the publication also presents a selective rewriting technique labeled as \textit{g1, g2, ...}, which rewrites all but a configurable number of the least significant digits of each weight. Stated otherwise, the technique rewrites the \textit{globally} most significant digits of each weight.

In contrast, the other general approach, pursued in Publication V, is to design an optimization rewriting that has a naturally configurable size. To obtain
Table 5.2. Summary of optimization rewriting techniques for optimization statements with heterogeneous weights from Publication IV and Publication V.

<table>
<thead>
<tr>
<th>Construction</th>
<th>Aim</th>
<th>Publication</th>
<th>Label*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorting and merging networks arranged according to a mixed-radix decomposition of weights.</td>
<td>Adapt weight-rule normalization techniques to optimization rewriting.</td>
<td>IV</td>
<td>mr</td>
</tr>
<tr>
<td>Digit-wise selective version of $mr$.</td>
<td>Reduce the size of $mr$ or any other optimization rewriting.</td>
<td>IV</td>
<td>l1, l2, …</td>
</tr>
<tr>
<td>Global digit-wise selective version of $mr$.</td>
<td>Reduce the size of $mr$ or any other optimization rewriting.</td>
<td>IV</td>
<td>g1, g2, …</td>
</tr>
<tr>
<td>Weighted comparator network.</td>
<td>Provide a different and simpler alternative to the techniques from Publication IV.</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>Weighted sorting network.</td>
<td>Apply weighted comparator network rewriting to its fullest extent.</td>
<td>V</td>
<td>F</td>
</tr>
<tr>
<td>Depth-limited weighted comparator network.</td>
<td>Reduce the size of $F$.</td>
<td>V</td>
<td>L1, L2, …</td>
</tr>
<tr>
<td>Sparsely weighted sorting network.</td>
<td>Experiment with the placement of weights in $F$.</td>
<td>V</td>
<td>W1, W2, …</td>
</tr>
</tbody>
</table>

* These labels are used in Table 5.3.
such a rewriting, the publication introduces the novel concept of a *weighted comparator network*. These are comparator networks with weights associated to wires. Adding such weights leads to a natural interpretation of a weighted comparator network as a function that maps an input sequence not only to an output sequence, but also to a single additional value. This additional value is obtained as a weighted sum of all wire values and their associated weights. By drawing an analogy between these weighted sums of wire values and the weighted sums of literals in optimization statements, the publication turns weighted comparator networks into blueprints of optimization rewritings. The result is a general optimization rewriting approach where an optimization statement is transformed into a weighted comparator network that is then modified and transformed back into an optimization program. This general approach becomes concrete once given a scheme for generating comparator networks and a procedure to modify the weighted comparator network. Both of these aspects are highly configurable. For example, the comparator network may be chosen to be a sorting network or a comparator network obtained by truncating a sorting network to a limited depth as in table rows $F$, $L_1$, $L_2$, …, respectively. Moreover, the modification procedure may assign the weights of the weighted comparator network via a *propagation* process that produces either a relatively dense distribution of weights as in table rows $F$, $L_1$, $L_2$, … or a sparse distribution as in table row $W_1$, $W_2$, ….. A common property of these optimization rewriting techniques based on weighted comparator networks is that their size is independent of the magnitude of the weights in rewritten optimization statements. Hence, their size is straightforward to control for. On another note, on unit-weight optimization statements, the use of a full sorting network and dense propagation as in $F$ yields a technique coinciding with the sorting network based technique presented in Section 5.1. Hence, also the formal analysis from Section 5.2 applies to the technique in the case of unit weights. The performance impact of these configurations is assessed experimentally in Publication V and summarized in Section 5.5.

### 5.5 Experimental Evaluation with Heterogeneous Weights

This section presents the results of a unified and extended performance evaluation of the optimization rewriting techniques from Publications IV and V designed for heterogeneous weights. While the original publications already contain experimental evaluations, the respective benchmark environments differ. Hence, to enable the combined evaluation shown in this dissertation, the techniques from Publication IV were re-run on the benchmarks and with the settings used and described in Publication V. The results from the latter publication were included as is.

Benchmarks considered in this evaluation include the Bayesian Network Learning problem [33, 65] with three sets of instances, the Markov Network Learning problem [33, 65] with three sets of instances, the Markov Network
Learning problem [67], the MaxSAT problem with industrial instances from the 2014 MaxSAT Evaluation [89] encoded in ASP as in the Sixth ASP Competition [54], and the Curriculum Based Course Timetabling problem [16,25]. Moreover, the Fastfood and Traveling Salesperson problems are included from the Second ASP Competition [38] with new instances generated for the evaluation in Publication V.

The results, shown in Table 5.3, are presented in terms of two alternative scoring schemes $S_1$ and $S_2$ from the Seventh ASP Competition [55], of which $S_1$ originates from the Mancoosi International Solver Competition. This score $S_1$ is computed for a single pipeline among $M$ pipelines over a domain $D$ of $N$ instances as $S_1 = \frac{100}{MN} \sum_{I \in D} M(I)$, where $M(I)$ is the number of pipelines that produced at least some solutions to $I$ but no solution of higher quality than the pipeline in question. It is noteworthy that these $S_1$ scores are relative to the results of the other pipelines in the comparison, and are therefore not directly comparable to score values from the mentioned competitions. Furthermore, the score $S_2$ is computed as $S_2 = \frac{100}{N} C$, where $C$ is the number of confirmed optimal solutions produced by the pipeline to instances from $D$.

High-level descriptions of the used rewriting techniques can be found in Table 5.2. The respective techniques are implemented in two preprocessing tools LP2NORMAL (v. 2.27) and PBTRANSLATE (v. 0.2.0). Regarding details, to enable the feasible comparison of a good number of optimization rewriting techniques, the scope of the evaluation is limited to the single back-end solver CLASP (v. 3.3.3) [50]. Moreover, for simplicity, a single sorting network scheme is fixed as a base for the techniques from Publication IV and another for the techniques from Publication V. Both are sophisticated but different optimized versions of Batcher’s odd-even sorting networks [19].

For the selective rewritings $l_1, l_2, l_3$ and $L_k$ from Publications IV and V, respectively, the integer parameters are relative indicators of how much rewriting the techniques apply. Furthermore, $L_k$ is never larger than $l_1, l_2, l_3$ for any size parameter values as long as both techniques use similar sorting network schemes underneath. Hence, the results show how much rewriting is appropriate for each benchmark. Moreover, the table rows are organized to emphasize that the pipelines clasp/l0/r0/L0, $l_1, l_2, l_3$, mr form an increasing gradient in terms of the magnitude of rewriting, as do the pipelines clasp/l0/r0/L0, $L_4, L_8, L_{16}, L_{32}, F$. Studying the results reveals trends within some benchmarks: on Bayes Alarm the techniques from Publication IV reduce $S_1$ scores monotonically with more rewriting, whereas on MaxSAT the techniques increase $S_1$ scores consistently. Moreover, the techniques $L_k$ show an increasing trend on Traveling Salesperson. In contrast, the majority of the remaining benchmarks reveal sweet spots of parameter values. Most of these sweet spots differ from the trivial parameter values of $l0$ and $L_0$, proving that selectivity is an important contribution to optimization rewriting. Moreover, as the global selective rewriting

---

2Available at http://www.mancoosi.org/misc/.
3Available at https://github.com/jbomanson/pbtranslate.
variant $g7$ leads to consistently reduced performance, it is clear that differences between selective rewriting techniques can be significant.

Indeed, in light of the results, the different types of selective rewriting techniques appear to be meaningfully different beyond how they are parameterized for size. This difference is visible on the MaxSAT class, where $S_1$ scores increase consistently with more rewriting, but only when using techniques from Publication IV and not V. Indeed, $S_1$ scores for $L_k$ from Publication V are all comparatively lower, and peak at $k \in \{4, 8, 16\}$ instead of increasing with the larger parameter values $k \in \{32, \infty\}$. Hence, the larger techniques from Publication IV have an edge in this case, and the techniques from Publication V can not be scaled to comparable size nor do the results give reason to believe that such scaling would yield comparable performance. On some of the other benchmarks, the edge is with techniques from the other publication, and hence neither can be argued here to be properly more effective than the other.

Regarding the impact of distributing weights, the results for sparsely weighted comparator network based rewritings $W_k$ with sparsity factors $k > 1$ show generally weaker signs when compared against the densely weighted rewriting $F/L_{\infty}/W_1$. For all of these versions $W_k$, the introduced network is the same—only the optimization statement contents differ. The larger the factor $k$, the more sparsely weights are distributed over comparator networks, and the smaller the resulting optimization statements are. With $k = \infty$, weights are placed on only the first and last network layers and with $k = _-$ on only the first. That is, $k = _-$ adds an encoded sorting network, but leaves the original optimization statement intact. Interestingly, the edge cases $k \in \{1, \infty, _-\}$ yield higher performance than the intermediate values $k \in \{4, 8, 16, 32\}$ in general. Hence, these benchmarks appear to favor maximally fine and maximally sparse distributions of weights over the intermediate options. Moreover, even the version $k = _-$ performing no weight propagation shows improvement over $k = 1$ on some benchmarks, namely on Bayes Hailfinder and Timetabling. This phenomenon was not further studied in these already extensive experiments, although a precise understanding of their causes could prove fruitful.

As a further note, the rank of different techniques depends on the score type of interest. Indeed, the $S_1$ scores that are sensitive to optimization values differ from the $S_2$ scores, which reward only pipelines that reach optimality. In particular, clasp performs relatively better in view of the $S_1$ scores, indicating that it generally finds relatively high quality solutions even when it fails to reach and prove optimality. On the other hand, because optimization rewriting performs relatively better in view of the $S_2$ scores, it may hint that rewriting helps particularly in the unsatisfiability checks involved in optimality proofs. This hypothesis is also supported by the formal analysis in Section 5.2, which demonstrates the potential of such techniques in a purely abstract setting.

Overall, the results indicate that optimization rewriting can yield robust performance improvements for the state-of-the-art ASP solver CLASP. In these experiments, most although not all of the improvements stem from rewriting
techniques of reduced size: $l1$, $l2$, $l3$ based on the “full” rewriting $mr$ from Publication IV and $L_k$ based on the “full” rewriting $F$ from Publication V. This is the case particularly on benchmarks where the full rewritings slow down CLASP. Moreover, the techniques from these two publications complement each other as neither dominates the other. Therefore, the results give support for using a portfolio approach that picks the most promising technique for each given instance from a number of options, in the manner of the portfolio solver ME-ASP [85], for example. In the context of these results, such a portfolio approach would help extract the most benefits from optimization rewriting.
Table 5.3. Scores $S_1$ and $S_2$ on optimization benchmarks with heterogeneous weights for solving pipelines using either no rewriting or rewriting techniques from Publication IV or Publication V. Higher scores are better and the best pipelines are highlighted for each benchmark column. Some pipelines can be labeled in multiple ways and these labels are delimited by slashes.

<table>
<thead>
<tr>
<th></th>
<th>$S_1$</th>
<th></th>
<th>$S_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BA</td>
<td>BH</td>
<td>BW</td>
</tr>
<tr>
<td>claspuc1</td>
<td>27.2</td>
<td>26.1</td>
<td>20.3</td>
</tr>
<tr>
<td>clasp/l0/r0/L0</td>
<td><strong>88.3</strong></td>
<td>94.7</td>
<td>63.9</td>
</tr>
<tr>
<td>IV</td>
<td>$l1$</td>
<td>74.9</td>
<td>88.3</td>
</tr>
<tr>
<td></td>
<td>$l2$</td>
<td>73.1</td>
<td>79.5</td>
</tr>
<tr>
<td></td>
<td>$l3$</td>
<td>56.6</td>
<td>72.8</td>
</tr>
<tr>
<td></td>
<td>$g7$</td>
<td>38.6</td>
<td>39.5</td>
</tr>
<tr>
<td>IV</td>
<td>$mr$</td>
<td>36.9</td>
<td>53.1</td>
</tr>
<tr>
<td></td>
<td>$L4$</td>
<td>80.2</td>
<td><strong>95.7</strong></td>
</tr>
<tr>
<td></td>
<td>$L8$</td>
<td>83.9</td>
<td>93.8</td>
</tr>
<tr>
<td></td>
<td>$L16$</td>
<td>84.5</td>
<td>92.4</td>
</tr>
<tr>
<td></td>
<td>$L32$</td>
<td>87.5</td>
<td>94.1</td>
</tr>
<tr>
<td>IV</td>
<td>F/L∞/W1</td>
<td>71.2</td>
<td>72.0</td>
</tr>
<tr>
<td></td>
<td>W1</td>
<td>61.4</td>
<td>69.6</td>
</tr>
<tr>
<td></td>
<td>W8</td>
<td>69.1</td>
<td>64.2</td>
</tr>
<tr>
<td></td>
<td>W16</td>
<td>65.3</td>
<td>61.5</td>
</tr>
<tr>
<td></td>
<td>W32</td>
<td>50.6</td>
<td>51.1</td>
</tr>
<tr>
<td></td>
<td>W∞</td>
<td>52.0</td>
<td>65.2</td>
</tr>
<tr>
<td></td>
<td>W-</td>
<td>65.5</td>
<td>84.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BA</th>
<th>Bayes Alarm</th>
<th>BW</th>
<th>Bayes Water</th>
<th>MS</th>
<th>MaxSAT</th>
<th>TSP</th>
<th>Traveling Salesperson</th>
</tr>
</thead>
<tbody>
<tr>
<td>BH</td>
<td>Bayes Hailfinder</td>
<td>MN</td>
<td>Markov Network</td>
<td>T</td>
<td>Timetabling</td>
<td>F</td>
<td>Fastfood</td>
</tr>
</tbody>
</table>
Answer set programming (ASP) is a general purpose problem-solving paradigm for combinatorial search problems that has grown increasingly expressive and efficient over time. It allows to encode search problems in a convenient logic based language. Problems encoded in this way can be solved with various combinations of tools developed for ASP. The main goal of this work has been to advance the development and performance of ASP tools by focusing on transformation techniques. Publication I summarized a range of normalization techniques compatible with ASP solving tools, and Publication II contributed to the general development of program transformations by providing novel methods to verify their correctness. These methods were demonstrated on actual, complex normalization techniques, attesting to their usability. Concerning performance, Publication III addressed the so-called grounding bottleneck, which limits the scalability of traditional ASP solving tools. Specifically, the publication advanced the state of lazy-grounding, a promising ASP solving approach, devised to overcome the grounding bottleneck. Further concerning performance, Publications IV and V took previous techniques for normalization from Publications VI and VII, and refined the techniques into a new class of transformations for ASP optimization statements: optimization rewritings. In accompanying experimental evaluations, the developed optimization rewritings proved to boost the optimization performance of the state-of-the-art ASP solver CLASP on a number of benchmarks. In summary, the research reported in this dissertation advances the development and performance of ASP solving tools in a number of ways. In the following, this dissertation concludes with a discussion of the significance of these advancements in Section 6.1 and then with suggestions for future research in Section 6.2.

6.1 Significance of the Results

Publication II provides theoretical and practical tools to prove or disprove the equivalence of ASP programs. This is a central task in establishing the correctness of program transformations, which are common in the development of both
programs and solving systems. While there has been work on such correctness considerations in the past, none of the work has studied the same equivalence relation as in the publication, which models the standard behavior implemented by usual ASP solvers exactly. In contrast, Publication II addresses this exact view via a novel equivalence relation, visible strong equivalence, and is therefore of exceptionally practical significance. Among the author’s particular contributions are formal results that help establish this relation for programs from two classes: positive programs and a broader class of “nearly” positive programs. As the publication demonstrates, the results are applicable to the correctness analysis of normalization transformations, illustrating broad relevance within ASP. Moreover, the results can be interpreted to reveal a close connection between ASP and classical logic on the relevant classes of programs. Indeed, the results operate in terms of a generalization of classical models called hidden minimal models. Therefore, the publication may prove to have additional significance due to relating ASP and classical logic in this practically meaningful way.

The work in Publication III is significant for lazy-grounding ASP solving, due to extending the language fragment it supports, and thus narrowing an expressivity gap between lazy-grounding and ground-and-solve systems. Previously, this gap was large: the state-of-the-art of lazy-grounding systems focused on programs of only normal rules. In contrast, ground-and-solve systems have long supported the many language constructs standardized in ASP-Core-2 [29] such as aggregates. To narrow the gap, the publication introduced a general framework for lazily normalizing certain aggregates on the first-order level. This increase in expressivity is a step towards the long-term goal of lazy-grounding: to provide practical, efficient, expressive ASP systems that scale beyond the reach of traditional ground-and-solve approaches. As lazy-grounding continues to approach this ambitious goal, it stands the chance to unlock new application areas for ASP, to draw interest from other scientific communities dissuaded by the limitations of ground-and-solve systems, and to extract new value from existing ASP research in the process.

By way of improving optimization performance, Publications IV and V have the potential to impact the larger field of Computational Logic around ASP. On the technical level, the publications propose the novel concept of optimization rewriting as a general preprocessing method for optimization problems with pseudo-Boolean optimization criteria. For ASP, optimization rewriting proves to have performance increasing capability both in theory and in computational experiments. Despite the ASP-centric presentation of the publications, the proposed techniques are more general as they are readily applicable to several optimization problem formalisms besides ASP. Specifically, the techniques are applicable to any formalism that involves pseudo-Boolean optimization criteria and in which one can encode the operation of comparator networks. In particular, such formalisms include Mixed Integer Programming (MIP), Maximum Satisfiability (MaxSAT), Maximum Satisfiability Modulo Theories (MaxSMT), Optimization Modulo Theories (OMT), and pseudo-Boolean optimization (PBO)
Discussion

also known as 0-1 integer linear optimization (0-1 ILP). It is possible that the presented optimization rewriting techniques would yield improvements on many datasets and techniques employed in these formalisms as well. Some such improvements are already certain, as shown by an independent study [103] of MaxSMT and OMT techniques analogous to the uniform-weight techniques from Publication IV.

6.2 Future Research

It would be fruitful to extend the ASP equivalence checking work from Publication II by investigating further ways to facilitate equivalence proofs. For example, a particular aspect of these results that could be extended concerns Proposition 4.12 therein. This proposition shows that a specific, relatively simple correspondence between two “nearly” positive programs implies that the programs are visibly strongly equivalent. As demonstrated via concrete examples, the proposition helps prove the correctness of many meaningful program transformations. Nevertheless, the converse of the proposition does not hold. Therefore, it would be interesting to identify program classes for which the converse does hold and the proposition is thus complete. To this end, the discussion in the publication gives some specific ideas. In another possible line of future work, the equivalence checking implementation from Publication II could possibly be accelerated. There is potential for such acceleration, since the current implementation relies on the most general characterization results of the publication, Theorems 3.21 and 3.27 for all programs. Instead, the implementation would ideally identify programs or even parts of programs for which the more specialized characterization results of the publication hold, and apply the results. For the identification, the discussed, complete counterparts of Proposition 4.12 would be useful.

The work in Publication III on lazy-normalization for lazy-grounding opens up promising avenues for future work. Namely, the general lazy-normalization framework given in the publication could be applied to normalize aggregates besides the already considered monotone count and sum aggregates. Support for non-monotone aggregates, in particular, would close a major part of the remaining expressiveness gap between lazy-grounding and ASP-Core 2 [29]. Moreover, it could be fruitful to develop alternative first-order normalization schemes to complement the existing monotone count and sum aggregates for performance reasons. Whether such schemes can be made meaningfully more efficient or compact is an interesting question, when measuring compactness by numbers of ground rules produced during lazy-grounding. To this end, it would be particularly useful to develop a lazy-normalization for sum aggregates that is efficiently applicable in the lazy-grounding context and also comparable in size to ground-level sorting network based encodings. In addition to lazy-normalization, other approaches for aggregates could yield performance benefits in some cases.
Discussion

Regarding ASP optimization performance studied in Publication IV and Publication V, the developed optimization rewriting techniques are compatible with a number of other declarative problem-solving formalisms, as discussed in Section 6.1. Therefore, it would be interesting to evaluate the techniques on datasets for such formalisms, and to compare the design and impact of the techniques to any existing techniques for the other formalisms. In another line of research, it would be possible to combine the concepts of optimization rewriting and conflict-directed lazy decomposition (CDLD) [3] for potential further performance increases. On a technical level, each optimization rewriting technique would have to be individually modified from a preprocessing transformation into a CDLD-style incremental transformation. Depending on the technique, this can be nontrivial. Nevertheless, the relative simplicity of optimization rewriting techniques based on weighted comparator networks from Publication V could facilitate the task.
References


Normalization and Rewriting for Answer Set Programming and Optimization

Jori Bomanson