Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Jori Bomanson

Developing Efficient Encodings for Weighted Expressions in Answer Set Programs

Master's Thesis
Espoo, August 15, 2014

Supervisor: Docent Tomi Janhunen, Aalto University
Advisor: Dr. Martin Gebser, Aalto University

Aalto University
School of Science

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF MASTER'S THESIS

Author:	Jori Bomanson		
Title: Developing Efficient Encodings for Weighted Expressions in Answer Set Programs			
Date:	August 15, 2014	**Pages:**	85
Major:	Information and Computer Science	**Code:**	T-119
Supervisor:	Docent Tomi Janhunen		
Advisor:	Dr. Martin Gebser		

Answer set programming is a declarative problem solving paradigm suitable for searching solutions to combinatorial search problems. Propositional answer set programs, studied in this thesis, consist of rules that state logical connections between atomic propositions, or atoms. A program represents the problem of finding truth assignments, called answer sets, that satisfy the rules in the program, under the condition that by default atoms are false. Answer set programming can be used as a general purpose problem solving mechanism, by writing programs whose answer sets correspond to solutions of a chosen search problem, and then using automated tools to find them.

In this work, we focus on normalizing a particular type of rules, weight rules, into so called normal rules. We develop normalization strategies that extend existing translations applied in answer set programming and propositional satisfiability checking. In particular, we propose to incorporate a base selection heuristic and a structure sharing algorithm into a weight rule translation that decomposes the rule in a mixed-radix base. Both the previous and novel techniques have been implemented in a normalization tool, and we experimentally evaluate the effect of our methods on search performance. The proposed techniques improve on the compared normalization methods in terms of conciseness, the number of conflicts encountered during search, and the amount of time needed to find answer sets using a state-of-the-art solving back-end. On certain benchmark classes, the normalization techniques improve even on native weight-handling capabilities of the solver.

Keywords:	answer set programming, cardinality rule, normalization, translation, weight rule, weighted expression
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Jori Bomanson		
Työn nimi: Tehokkaiden esitystapojen kehittäminen painosäännöille vastausjoukko-ohjelmissa			
Päiväys:	15. elokuuta 2014	**Sivumäärä:**	85
Pääaine:	Tietojenkäsittelytiede	**Koodi:**	T-119
Valvoja:	Dosentti Tomi Janhunen		
Ohjaaja:	Dr. Martin Gebser		

Vastausjoukko-ohjelmointi on kombinatoristen hakuongelmien ratkontaan soveltuva ongelmanratkontamenetelmä, jossa ohjelmointi koostuu loogisten yhteyksien deklaratiivisesta määrittelemisestä. Työssä käsitellyt vastausjoukko-ohjelmat koostuvat säännöistä, jotka määräävät atomisten lauseiden, tai atomien, väliset yhteydet. Tämänmuotoiseen ohjelmaan kytkeytyy formaali ongelma, jonka ratkaisemiseksi on löydettävä yksi tai useampi totuusjakelu näille atomeille, eli vastausjoukko, joka täyttää ohjelmaan kirjatut säännöt. Jokainen atomi on lisäksi asetettava epätodeksi paremman tiedon puutteessa. Vastausjoukko-ohjelmointi muodostaa yleiskäyttöisen ongelmanratkontamekanismin, sillä on mahdollista kirjoittaa ohjelma jonka vastausjoukoista on luettavissa kiinnostuksen kohteena olevan hakuongelman ratkaisut ja näiden vastausjoukkojen etsintää varten löytyy automatisoituja työkaluja.

Tässä diplomityössä keskitytään erilaisten sääntötyyppien välisiin muunnoksiin ja erityisesti niin kutsutuiden painosääntöjen uudelleenkirjoittamiseen vain normaalisääntöjä käyttäen. Työn tavoitteena on kehittää tähän soveltuvia normalisointimenetelmiä olemassaolevien tekniikoiden pohjalta. Diplomityössä esitellään uusi heuristiikka sekakannan hakemiseksi sekä rakenteenjakoalgoritmi erään normalisointimenetelmän vaatiman atomi- ja sääntömäärän karsimiseksi. Näiden ja muiden tekniikoiden käyttöä varten on kirjoitettu tietokoneohjelma, jonka avulla tehtyjä kokeellisia tuloksia esitellään lopuksi. Saatujen tuloksien perusteella työssä kehitetyt tekniikat tuovat parannuksia muunnoksien tiiviyteen, haussa kohdattujen umpikujien määriin sekä erään johtavan ratkaisinohjelman aikavaatimuksiin. Tiettyjen testiongelmien kohdalla työssä esistetyt tekniikat auttavat ylittämään jopa ratkaisinohjelmaan sisäänrakennettujen painosääntömenetelmien tehokkuuden.

Asiasanat:	kardinaliteettisääntö, käännös, normalisointi, painosääntö, vastausjoukko-ohjelmointi
Kieli:	Englanti

Acknowledgements

I would like to thank my supervisor for all the guidance he has given me over the years, and especially for his instructions and patience in supervising my thesis. I would also like to thank my advisor, whose watchfulness and expertise led to countless improvements to the work. Moreover, I want to thank the answer set programmers at the University of Potsdam for a very pleasant visit.

Finally, I wish to thank my family for their long-lasting support regarding studies and life.

Espoo, August 15, 2014

Jori Bomanson

Contents

Chapter 1

Introduction

Answer set programming (ASP) [11] is a declarative problem solving paradigm. As such, the focus of programming *answer sets* lies on describing search problems to the extent that subsequently carried out automated reasoning techniques can implement the actual search. In this way ASP relates to integer linear programming [40], constraint satisfaction [44], propositional satisfiability [9], and numerous other formalisms in deviating from the traditional programming setting, in which the programmer is responsible for determining both what is to be computed and how. The syntax of ASP inherits its basic elements from *logic programming with negation* [37]. Thus, for one, an answer set program consists of a set of rules. Each rule intuitively contributes to a definition of how the truth value of a specific primitive, atomic proposition is determined. A typical rule states that on a specified condition, written in terms of such *atoms*, a given *head* atom follows logically. When several of such rules are combined to form a program, a nontrivial computational problem arises, which is to find truth values for all of the atoms in the program that are consistent and justifiable in light of the rules. By writing appropriate rules, the answers to a program can be made to parallel the solutions to a selected search problem. This approach, combined with the use of automated ASP solvers [28, 34, 36, 41] to find the answers, and thus the solutions, constitutes a general purpose problem solving technique. Furthermore, answer set programs admit useful language constructs that help in encoding application problems. Most fundamentally, the conditions within rules generally rely on an intrinsic property of answer set programs, which is the role of *negation*. Namely, it is possible to express conditions on both the existence of a logical derivation for an atom and the lack of such, and this expressivity shapes the characteristics of *encoding* and *solving* problems via ASP.

The theoretical foundations of ASP began with the *stable model semantics* [29] proposed as an approach to support the intuition of *negation as failure* [13] in logic programs. This means that in programs, consisting of normal rules at the time, a negative reference to an atomic proposition is deemed to be satisfied when the program does not support a derivation for the atom. On the other hand, an atom is considered satisfied only once it has a derivation. This leads to asymmetry between positive and negative references to atoms, or *literals*, in that atoms are regarded *false by default*, and true only once proven to be so. In contrast, in classical propositional logic consistency suffices in either case, that is, if a literal does not contradict any known facts, no further justification is needed to consider it a fact as well. Consequently the solutions to an answer set program, which make up its so called *stable models*, are more scarce than its plain, classical *models*.

Due to the above recounted properties, logic programs under the stable model semantics are a form of *nonmonotonic* reasoning. Indeed, restricting concerns to specific classes of models, as above to stable models, and the ensuing ability to follow commonsense reasoning and to find conclusions under lack of evidence, are aspects of nonmonotonic reasoning. In doing so, these nonstandard formal systems model the phenomenon in commonsense reasoning that a larger set of initial assumptions does not necessitate a larger set of conclusions [38].

1.1 Normal and Extended Rules

An answer set program is a set of rules. Generally the rules fall into a number of classes, such as normal, choice, cardinality, and weight rules. The first of these is the most fundamental. The other three constitute *extended rule* types, which advance expressivity and conciseness [41]. For the benefit of answer set programmers, the input languages of systems such as LPARSE [43] and GRINGO [25] include support for these extended rule types. This is one of the aspects behind the success of ASP, and plays a part in providing rich modeling capabilities that have resulted in a wide adoption of the paradigm. In this section, normal rules are introduced, followed by, via examples, rules of the extended types. Thereafter, in Chapter 2, we continue with a more formal consideration of specifically normal, cardinality, and weight rules, which are the most central to this thesis.

Every type of rule considered in this work consists of a *head* and a *body* condition. If the body of a rule holds, the head is implied. Both heads and bodies are written in terms of atoms. Consequently, rules provide a way to determine truth values for atoms, and encode dependencies between them. The different types of rules vary in the form of the two components. A *normal rule* is of the most basic type, and includes an atom for the head and a list of possibly negated atoms for the body. These components are written as a normal rule in the form

$$a \leftarrow b_1, \ldots, b_n, {\sim}c_1, \ldots, {\sim}c_m. \tag{1.1}$$

The head atom a is implied by the rule (1.1) if the atoms $b_1, \ldots, b_n$ are derivable but none of $c_1, \ldots, c_m$ is. Such rules form the foundation of ASP that is supplemented by extended rule types considered in the following.

A frequent concern of answer set programs is the need to express *choices* between options. Whereas the stable model semantics necessitates the derivation of any atom a to be justified by the program at hand, there still remain occasions on which this behaviour is uncalled for. In other words, we may want to forego the stipulation of a default value for chosen atoms, and give equal grounds for choosing either true or false by default. A way of expressing this in a normal logic program is to state that the lack of a derivation for either the true or false value suffices to justify the choice of the opposite value. That is, for an atom a we can introduce a new atom, say b, with the intuitive reading that a is false. Now, we can express our intent with the rules

$$\begin{aligned} a &\leftarrow {\sim}b. \\ b &\leftarrow {\sim}a. \end{aligned} \tag{1.2}$$

The consequence is that, given any consistent candidate set of atoms whose stability is to be verified, whichever is the truth value atom a takes, the above pair of rules gives a derivation for it.

This construction effectively changes the intuitive interpretation of the default negation operator $\sim$ applied to a chosen atom to the classical one. The choice to either include or exclude a is accepted as long as it is consistent with everything else. The effect of the rules in (1.2) can also be attained with an extended rule, called a *choice rule*. In our example, we would write $\{a\}$ to denote it. More generally, for a set of atoms $a_1, \ldots, a_n$, we would write $\{a_1, \ldots, a_n\}$ to express the same for each a_i. These two are examples of choice rule *heads*, which can be combined with body conditions to obtain the general form

$$\{a_1, \ldots, a_n\} \leftarrow b_1, \ldots, b_m, {\sim}c_1, \ldots, {\sim}c_k. \tag{1.3}$$

The rule (1.3) shares the consequences of $\{a_1\}, \ldots, \{a_n\}$ on the condition that all of $b_1, \ldots, b_m$ can be derived while none of $c_1, \ldots, c_k$ can. In case the body is omitted, its absence is understood as a satisfied condition.

Another frequently occurring pattern is one where a constraint is to be placed on a number of literals, stating that the number of them that are derivable by the rules of the program must stay within a given range. Alternatively, an atom is sought to be derived when such a condition on the *cardinality* of a set of atoms is met. As with choice rules, we first consider the prospects for imposing such conditions via normal rules. Let us take the set of literals $b, c, d, e, {\sim}f, {\sim}g$ for an example, and an atom a to be the target of our derivation, which is to take place whenever three or more of the input literals are satisfied. In this setting, we may write the following normal logic program to accomplish the task:

$$\begin{array}{lll} a \leftarrow b, c, d. & a \leftarrow c, d, e. & \ldots \quad a \leftarrow e, {\sim}f, {\sim}g. \\ a \leftarrow b, c, e. & a \leftarrow c, d, {\sim}f. & \\ \vdots & \vdots & \\ a \leftarrow b, {\sim}f, {\sim}g. & a \leftarrow c, {\sim}f, {\sim}g. & \end{array} \tag{1.4}$$

The above contains a normal rule for each triple of the input literals. Fully written out, it would thus contain $\binom{6}{3} = 20$ normal rules, each with 3 body literals. This is a clumsy way of declaring our intentions, and even much more so than was the case with choice rules. For larger literal sets, this becomes infeasible quickly. However, there exists support for concisely representing rules such as those in (1.4). This comes in the form of another extended rule type, namely that of *cardinality rules*. To express our chosen limit of three satisfied literals on the same set in the body of a single rule, one would write $a \leftarrow 3 \leq \langle b, c, d, e, {\sim}f, {\sim}g \rangle$. In general, a cardinality rule takes the form

$$a \leftarrow k \leq \langle b_1, \ldots, b_n, {\sim}c_1, \ldots, {\sim}c_m \rangle. \tag{1.5}$$

We intuitively interpret it to mean that if in total k or more of the literals $b_1, \ldots, b_n$ and ${\sim}c_1, \ldots, {\sim}c_m$ are satisfied, then a is derived from the rule.

As a further extension in answer set programs, there are *weight rules*. They generalize cardinality rules by allowing weights for body literals. Thus whereas a cardinality rule's condition is essentially dependent on the sum of binary variables, the condition of a weight rule depends on a linear combination of such variables and specified constants. Practical applications for weight rules are numerous. Let us consider an example where we have a set of items, each with an associated integer cost. Moreover, let us be asked to select a subset of those items, and infer whether the total cost overflows a threshold value. An instance of such a problem could contain the atoms $b, c, d,$ and e, each denoting that a corresponding item is selected. These atoms could cost $50, 90, 10,$ and 40 units, respectively. To derive a marker atom, say a, whenever the total cost equals or exceeds, say 100, we could now write

$$\begin{array}{l} a \leftarrow b, c. \\ a \leftarrow b, d, e. \\ a \leftarrow c, d. \\ a \leftarrow c, e. \end{array} \tag{1.6}$$

The rules in (1.6) have been chosen by enumerating all subsets of b, c, d, e whose total cost is at least 100, and whose proper subsets' costs are less. This encoding scheme is a straight-forward generalization of the one used in presenting cardinality rules. Consequently, the drawbacks are inherited and the number of required rules can be exponential in the number of input literals. The behaviour of the rules in (1.6) can again be obtained more concisely in ASP. Namely, we can instead write $a \leftarrow 100 \leq \langle b = 50, c = 90, d = 10, e = 40 \rangle$. The general form taken by weight rules is the following:

$$a \leftarrow k \leq \langle b_1 = w_1, \ldots, b_n = w_n, {\sim}c_1 = w_{n+1}, \ldots, {\sim}c_m = w_{n+m} \rangle. \tag{1.7}$$

The associated interpretation is that if the limit k is matched or exceeded by the sum of those weights w_i for which the corresponding literal b_i or ${\sim}c_{i-n}$ is derived, then the head atom a is derived as well. Inequalities similar to these have many names in the literature, including pseudo-Boolean constraints, knapsack constraints and 0-1 linear integer inequalities [2].

1.2 Implementing Extended Rule Support

As outlined in Section 1.1, there are useful extensions to normal logic programs that help a programmer to write down elaborate definitions. For such cases, we considered the programmer's point of view and the contrast between specifying normal and extended logic programs. In particular, we presented a number of extended rules as concise substitutes for classes of normal logic programs. In this section, we briefly consider the complementary task of handling extensions on the solving side. The available approaches can be roughly categorized in two. First, support for extended rules can be built in a solver, which then applies deduction rules or any other suitable methods for handling the extended rules *natively*. As the other approach, the extended rules can be fully *translated* into normal rules, or simply *normalized*, in a preprocessing stage [33]. Afterward, a chosen ASP solving technique can be applied to the resulting program, whether or not it supports the extensions natively. In the remainder of this section, we give a general overview of both techniques.

An ASP solver with native extended rule support admits input containing extended rules encoded as is, rather than normalized. Internally, the systems vary in the way they process normal and extended rules. In the following, we refer to a number of solving systems with extended rule support, and summarize their relevant solving techniques.

1. The SMODELS program looks for inevitable and possible consequences for each type of extended rule during its operation, in addition to performing stability checks on candidate answer sets [41]. The consequences are captured by a variation of an algorithm of Dowling and Gallier for computing

deductive closures of sets of positive normal rules [20]. For weight rules, the extended algorithm used in SMODELS keeps track of the sum of body weights for each weight rule as computation progresses.

2. The DLV system [34] for ASP supports aggregates [23]. These aggregates are set-oriented functions that intuitively map multisets to constants such as counts, sums, products, minimas, maximas and averages. These aggregates can be used in combination with lower and upper bounds to encode cardinality and weight rules as well. The DLV system comprises a grounder of its own in addition to a propositional solver. An interesting optimization employed within the grounder and the solver performs duplicate set recognition, in order to internally handle a single set appearing in several aggregates compactly without duplication. This feature is not found in other ASP systems to our knowledge.

3. The Conflict-Driven Nogood Learning ASP solver CLASP [28] carries out a bounding technique known as *unit propagation* on a pair of pseudo-Boolean constraints for each weight rule [26]. Each constraint pair defines the immediate circumstances required to satisfy the body of a weight rule. This dedicated treatment of weight rules is conceptually carried out on a logical specification that is formalized by an exponential number of *nogoods* capturing the pseudo-Boolean constraints. Instead of explicitly generating these sets of nogoods, the solver operates directly on the pseudo-Boolean constraints to deduce the outcomes of unit propagation on them. In addition, the solver implements dedicated *unfouded set* checking for ruling out unwanted self-justifying loops passing through weight rules.

4. The PBMODELS system incrementally translates an input program into a growing propositional formula extended with weight constraints [36]. The formulas are processed repeatedly by invoking a pseudo-Boolean solver having native support for the extension. After each such call, the returned model is checked for stability. In the negative case the formula is augmented with appropriate *loop formulas*, which rule out the particular model from further consideration, and the search continues.

As mentioned earlier, native solving methods tailored for extended rules are complemented by translation-based techniques. The idea is to replace each extended rule with a set of normal rules that serve the same purpose. The approach thus essentially reverses the steps taken in Section 1.1, by expanding concise declarations into significantly larger sets of simple rules. The kind of naïve encodings of cardinality and weight rules used therein, however, can quickly cause trouble as the operated literal sets grow in size. As a remedy, are other known encodings that are more concise, although more elaborate as well. For example, the ASP solver CMODELS [30] employs built-in cardinality and weight rule translations based on [24] prior to running a solving procedure. In the following, we discuss such improved encodings introduced for use in either ASP or *Boolean satisfiability* (SAT), but which have encodings in both formalisms. We call the following translations *monotone*, because they are modeled after *monotone circuits*. In the setting of ASP, this entails that the encodings do not introduce negation. Restricted use of negation is beneficial from a correctness point of view that is considered in Section 2.2.

For cardinality rules with in total n body literals and a lower bound of k, there is a *counting grid*-based translation with $\mathcal{O}(k \times n)$ auxiliary atoms and rules [33, 41, 42]. The idea in the translation is to count satisfied body literals one by one, or *sequentially*, until the cardinality test against k can be deduced directly. Another translation synthesizes a *merge-sorter* in $\mathcal{O}(n \times (\log n)^2)$ or $\mathcal{O}(n \times (\log k)^2)$ gates, clauses, or rules [5, 8, 10], which orders the truth values of body literals, such that the comparison with k can again be done directly. For weight rules, there are pseudo-polynomial translations requiring $\mathcal{O}(k \times n)$ new atoms and rules [2, 31]. Furthermore, there are *polynomial watchdog* translations based on *networks of totalizers* [7] that are related with *networks of sorters* [21]. All of the mentioned methods provide significantly improved translation sizes in terms of required rules, in comparison to straightforward encodings that, for example, do not make use of auxiliary atoms.

Despite the intersection between ASP and SAT translations, there are caveats concerning the use of translations in ASP that have been adapted from translations into SAT. In particular, in ASP, a substitute for a weight rule must preserve any *positive, cyclic dependencies* in the program that pass throught the rule. Such dependencies arise between head atoms and positive body atoms, and these dependencies propagate transitively. If the translation does not equally reflect the dependencies, the answer sets of the program can be affected even when its classical models are not. This potential shortfall, exemplified in the following, warrants careful studying of the correctness of translations in ASP, and in Section 2.2, we review and develop techniques for ensuring correctness in this light.

Example 1 *Consider the program $P = \{a \leftarrow 1 \leq \langle a, {\sim}a \rangle.\ \}$ and a hypothetical translation $Q = \{a.\ \}$. The program P has no answer sets whereas Q has one, which consists of a. In Section 2.1 we familiarize with how to find answer sets, but for now we can informally explain the discrepancy as follows. In P, the truth of atom a follows once it is already derived or underivable. The former case is ruled out because a is to be false by default and the latter because the underivability of a leads to a contradiction. For Q, the former obstacle does not exist, and a is true. Thus the substitution of P with Q is generally unsound in ASP.*

In summary, techniques for implementing extended rule support can be divided into two categories: native and translation-based methods. The former is built-in, whereas the latter can be carried out in a preprocessing stage. The purpose of this thesis, is to explore the feasibility and performance of the translation-based approach, particularly for weight rules. Motivation to this end lies in several factors. First, translations have been extensively researched in the pseudo-Boolean and SAT community, but less so in ASP. This presents an opportunity to transfer the result of that research into the area of ASP. Furthermore, the ability to implement translations in preprocessing stages gives versatility in allowing to easily combine translation techniques with selected solving methods, which may or may not support extended rules. Moreover, translations have potential for improving the performance of subsequently carried out search procedures, even in comparison to native techniques.

1.3 Thesis Objectives and Content

In this section, we give the primary goals, contributions, and structure of the thesis.

The general purpose in this work is to explore the feasibility of the translation-based approach to handling cardinality and weight rules. To this end, our strategy is to identify and refine promising existing techniques, and to develop new translation schemes in order to witness improvements in terms of conciseness and search performance. In particular, the focus is on weight rules of substantial size, which thus benefit from well scaling normalization techniques. A specific objective is to utilize compact translation techniques, originating from circuit design and SAT research, as building blocks of more elaborate constructions. The correctness of the resulting, novel constructions is to be studied both formally and experimentally, and the goal in this regard is to achieve results sufficient in generality even in the context of ASP. Taking another approach, we aim to also simplify weight rules prior to normalization or further processing. Our final objective within the scope of this work is to gain an understanding of the performance implications of the various considered simplification and normalization techniques.

The main contributions of this work consist of novel enhancements of the (global) polynomial watchdog translation [7] adapted in this thesis from SAT to ASP. The enhancements are obtained by utilizing mixed-radix bases in decomposing weight rules, and by analyzing, manipulating, and eliminating parts of the translation by the use of an algorithm for structural sharing. Moreover, techniques proposed for use with ASP transformations in [33] are applied in proving the correctness of the translation, and, in this process, the methods are tailored to a class of almost positive translations. Regarding practical contributions, we have implemented several cardinality and weight rule translations in a normalization tool, LP2NORMAL2, which is used in experimentally assessing the size and performance implications of the translation techniques.

The structure of this thesis is the following. In Chapter 2, the syntactic and semantic concepts of ASP relevant to this work, together with formal tools for correctness considerations, are described. Moreover, representational issues concerning finite domain integers are discussed, and a unary notation is defined for encoding them. Building on these preliminaries, a number of existing normalizations of cardinality and weight rules are presented in Chapter 3. In Chapter 4, novel schemes for weight rule normalization are proposed. Then, we consider prospects for simplifying weight rules in Chapter 5, where a number of conditionally applicable simplification steps are defined. In Chapter 6, the performance ramifications of the discussed normalization and simplification techniques are evaluated, in terms of conciseness and solving time. Furthermore, the computational effort required to perform verification checks for normalizations is studied therein. Finally, we discuss related work and present the conclusions in Chapter 7.

Chapter 2

Preliminaries

In this chapter, we specify the formal concepts of ASP relevant to this work. Section 2.1 covers basic notions of ASP, including language constructs used in the thesis. Equivalence notions for answer set programs are considered in Section 2.2, where we describe and develop formal tools for comparing programs and verifying the correctness of translations. Finally, a unary notation for expressing finite domain integers in terms of Boolean values, or literals, is discussed in Section 2.3.

2.1 Answer Set Programs

In the following, we describe *weight constraint programs* (WCPs) and *normal logic programs* (NLPs), which are the respective source and target languages of our normalizations. The scope of this thesis involves only the propositional case, and thus only ground programs, which are typically obtained using grounders operating on first-order languages.

Weight constraint programs are written in terms of *propositional atoms* a, their *default negations* $\sim a$, integer *bounds* k, and nonnegative integer *weights* w. A propositional atom a is also known as a *positive literal* and simply as an *atom*. The default negation $\sim a$ of a is also called a *negative literal*, and in this work, moreover as the *negation* of a. Both are types of *literals* l. An atom a is said to appear in a literal l if either $l = a$ or $l = \sim a$. Intuitively, an atom is the most primitive object bearing a truth value, and an occurrence of an atom is interpreted as the condition that the atom can be proven true, and an occurrence of its negation, that it is not.

A *weighted literal* $l = w$ combines a literal l with a weight w. Given sequences of literals $L = \langle l_1, \ldots, l_n \rangle$ and weights $W = \langle w_1, \ldots, w_n \rangle$, we write $L = W$ to abbreviate the weighted literals, or the *weighted expression*, $l_1 = w_1, \ldots, l_n = w_n$. In certain contexts, a weighted expression is also denoted as a list $\langle L = W \rangle$. We mix abbreviations and single weighted literals freely within weighted expressions.

Example 2 *Given sequences $L_1 = \langle a, b\rangle$, $L_2 = \langle c, {\sim}d\rangle$, and $W = \langle 2, 3\rangle$, we write $\langle L_1 = W, L_2 = W, e = 1\rangle$ to denote $\langle a = 2, b = 3, c = 2, {\sim}d = 3, e = 1\rangle$.*

A single literal l is allowed to appear in a weighted expression multiple times in part of several weighted literals, and further, a single weighted literal $l = w$ to have multiple occurrences as well. For example, we consider the following expressions all legal and distinct: $\langle a = 7\rangle$, $\langle a = 4, a = 3\rangle$, $\langle a = 2, a = 2, a = 3\rangle$.

A *weight rule* constitutes a *head* atom a, an integer *bound* k, and a weighted expression $L = W$, written in the form

$$a \leftarrow k \leq \langle L = W\rangle. \tag{2.1}$$

The right-hand side of (2.1), consisting of k and $L = W$, is called the body of the rule, and the contained literals L and weights W are respectively identified as *body literals* and *body weights*. The interpretation of a weight rule is that if the sum of body weights associated with derivable atoms and negations of underivable atoms matches or exceeds the bound k, then the head a is derived. A special case of weight rules is obtained when $W = \langle 1, \ldots, 1\rangle$, and rules of such form are called *cardinality rules*. If it further holds that $L = \langle l_1, \ldots, l_k\rangle$, the rule is a *normal rule* and is written in the simplified form

$$a \leftarrow l_1, \ldots, l_k. \tag{2.2}$$

Intuitively, a normal rule implies the head a once every body literal l_i is satisfied. The body of (2.2) is thus interpreted as a conjunction, which in the case of $k = 0$ is considered to be true and to invariably imply the head a. For this special case we use the shorthand '$a.$', and call the rule a *fact*. A WCP, or simply a *program*, is a finite set P of weight rules. An NLP is a finite set P of normal rules. The *signature*, denoted by $\mathrm{At}(\cdot)$, of an object containing atoms, is some fixed superset of those atoms [1]. Namely, for a positive literal $l = a$ and for a negative literal $l = {\sim}a$, we require that $\mathrm{At}(l) \supseteq \{a\}$, for a sequence of literals $L = \langle l_1, \ldots, l_n\rangle$, that $\mathrm{At}(L) \supseteq \bigcup_{i=1}^{n} \mathrm{At}(l_i)$, for a weight rule $r = (a \leftarrow k \leq \langle L = W\rangle)$, that $\mathrm{At}(r) \supseteq \{a\} \cup \mathrm{At}(L)$, and for a WCP $P = \{r_1, \ldots, r_n\}$, that $\mathrm{At}(P) \supseteq \bigcup_{i=1}^{n} \mathrm{At}(r_i)$. We further identify the part of a signature $\mathrm{At}(\cdot)$ that consists of atoms appearing in negative literals, and denote it with $\mathrm{At}_{\sim}(\cdot)$. Using this notation, we define a *positive rule* to be a rule r with $\mathrm{At}_{\sim}(r) = \emptyset$, and likewise, a *positive program* to be a program P with $\mathrm{At}_{\sim}(P) = \emptyset$.

[1]Typically $\mathrm{At}(\cdot)$ gives only the atoms that occur in the argument. However, it is convenient to allow $\mathrm{At}(\{a \leftarrow b.\}) = \mathrm{At}(\emptyset) = \{a, b\}$ when comparing programs $\{a \leftarrow b.\}$ and $\emptyset$, for example.

In order to formalize the stable model semantics of WCPs, and thus also that of NLPs, we first define their classical semantics. An interpretation $I \subseteq \mathrm{At}(P)$ of P is said to *satisfy* a positive literal $a \in \mathrm{At}(P)$ iff $a \in I$ and this is expressed by writing $I \models a$. Conversely, I satisfies a negative literal $\sim a \in \mathrm{At}(P)$ iff $a \notin I$, again denoted by $I \models \sim a$. With respect to an interpretation I, the *satisfied weight sum* of the weighted literals $\langle L = W \rangle$ is given by

$$v_I(L = W) = \sum_{\substack{1 \leq i \leq n \\ M \models l_i}} w_i \tag{2.3}$$

In the special case of cardinality rule bodies, we use the term *satisfied cardinality* for (2.3) and abbreviate it with $v_I(L)$. With the help of (2.3), we now define the body $k \leq \langle L = W \rangle$ of (2.1) to be satisfied in I iff $k \leq v_I(L = W)$. Furthermore, if in an interpretation I for a rule r of the form (2.1), the satisfaction of the body implies the satisfaction of the head then r is satisfied in I and we write $I \models r$. That is, we have $I \models r$ unless both $I \not\models a$ and $I \models k \leq \langle L = W \rangle$. Finally, an interpretation $M \subseteq \mathrm{At}(P)$ of a program P is also a *model* of the program iff it satisfies all the rules in the program. In notation, $M \models P$ iff $M \models r$ for every $r \in P$. A model $M \models P$ is further specified to be *subset minimal*, or simply minimal, iff no other model of P is a subset of it. In general a program can have several minimal models, but every positive program P has a unique minimal model which is called the *least model* and denoted $\mathrm{LM}(P)$.

The *stable models* of a program P form a restricted class of models described in terms of *reducts*. The reduct of a WCP P with respect to an interpretation $M \subseteq \mathrm{At}(P)$ is a positive program, denoted by P^M, constructed as follows. For each weight rule in P the reduct P^M contains the positive weight rule obtained by (i) removing the negative literals from the body and (ii) subtracting the bound by the weight of each removed literal satisfied in M [41]. For a weight rule of the form (1.7) this gives rise to a reduced rule $a \leftarrow k' \leq \langle b_1 = w_1, \ldots, b_n = w_n \rangle$ where $k' = k - v_M(\sim c_1 = w_{n+1}, \ldots, \sim c_m = w_{n+m})$. Finally, an interpretation M is a stable model of a WCP P iff M is the least model of the reduct, that is, iff $M = \mathrm{LM}(P^M)$. This forms a generalization of the original stable model semantics defined for NLPs in [29]. Generally a program P may have any number of stable models and the set of all of them is denoted by $\mathrm{SM}(P)$. A stable model is also called an *answer set*.

Example 3 *Consider a WCP P consisting of the rules*

$$a \leftarrow 4 \leq \langle b = 3, c = 3, d = 4 \rangle. \qquad b \leftarrow 1 \leq \langle \sim d = 1 \rangle.$$
$$c \leftarrow 2 \leq \langle a = 2 \rangle. \qquad d \leftarrow 1 \leq \langle \sim b = 1 \rangle.$$

The interpretations $M_1 = \{a, c, d\}$ *and* $M_2 = \{a, b, c\}$ *are both models of* P*. The reduct* P^{M_1} *consists of the rules*

$$a \leftarrow 4 \leq \langle b = 3, c = 3, d = 4 \rangle. \qquad b \leftarrow 1 \leq \langle \rangle.$$
$$c \leftarrow 2 \leq \langle a = 2 \rangle. \qquad d \leftarrow 0 \leq \langle \rangle.$$

and P^{M_2} *of the rules*

$$a \leftarrow 4 \leq \langle b = 3, c = 3, d = 4 \rangle. \qquad b \leftarrow 0 \leq \langle \rangle.$$
$$c \leftarrow 2 \leq \langle a = 2 \rangle. \qquad d \leftarrow 1 \leq \langle \rangle.$$

It follows that $\mathrm{LM}(P^{M_1}) = \{a, c, d\} = M_1$*, and* $\mathrm{LM}(P^{M_2}) = \{b\} \neq M_2$*. Consequently,* $M_1 \in \mathrm{SM}(P)$ *whereas* $M_2 \notin \mathrm{SM}(P)$*.*

2.2 Visible Strong Equivalence

Several aspects of correctness arise when translations or simplification techniques for altering pieces of programs are developed. To begin with, potential differences between the stable models of the swapped parts are of interest. In this respect we want the original program part P and the substitute Q to be semantically the same, at least in the sense of *weak equivalence* which requires that $\mathrm{SM}(P) = \mathrm{SM}(Q)$. It turns out that this is on the one hand too weak and on the other hand too strong for our purposes. Due to the nonmonotonicity of ASP, the addition of the same rules to weakly equivalent programs may lead to programs that are no longer so [22]. Thus, when parts of a program are substituted with others, the remaining program gives a *context* corresponding to such added rules. For a sufficiently strengthened relation, *strong equivalence* was introduced [35]. To be strongly equivalent, two programs P and Q need to have the same answer sets when each is combined with an arbitrary context program R, that is, $\mathrm{SM}(P \cup R) = \mathrm{SM}(Q \cup R)$. This directly rules out the aforementioned difficulty caused by nonmonotonicity. However, both weak and strong equivalence are too strict to allow for auxiliary atoms to be used. In the following we consider *visible strong equivalence* [33], a generalization of strong equivalence developed to allow for the use of auxiliary atoms, typically hidden from answer sets presented to the user.

In order to compare the visible behaviour of programs, we distinguish between different parts of the signature $\mathrm{At}(P)$ of a program P, by partitioning it into *visible* and *hidden* atoms, denoted by $\mathrm{At}_\mathrm{v}(P)$ and $\mathrm{At}_\mathrm{h}(P)$. In general they are freely chosen subsets of $\mathrm{At}(P)$ satisfying both $\mathrm{At}_\mathrm{v}(P) \cap \mathrm{At}_\mathrm{h}(P) = \emptyset$ and $\mathrm{At}_\mathrm{v}(P) \cup \mathrm{At}_\mathrm{h}(P) = \mathrm{At}(P)$. As a convention, the complete, visible, and hidden signatures of a program P are assumed to carry over to the reduct P^Y for any Y. The purpose of this partition is the same as for the typical public and private visibility specifiers for, e.g., variables in traditional programming languages. We use this concept in hiding auxiliary atoms introduced in translations. Only the visible parts of such translations are of interest when their correctness is evaluated. Namely, for programs P and Q with $A = \mathrm{At}_{\mathrm{v}}(P) = \mathrm{At}_{\mathrm{v}}(Q)$, we say they are *visibly equivalent* iff their sets of stable models $\mathrm{SM}(P)$ and $\mathrm{SM}(Q)$ are *visibly equal*, as defined in the following.

Definition 1 *Given programs P and Q with $A = \mathrm{At}_{\mathrm{v}}(P) = \mathrm{At}_{\mathrm{v}}(Q)$, and sets of interpretations $S_1 \subseteq \mathbf{2}^{\mathrm{At}(P)}$ and $S_2 \subseteq \mathbf{2}^{\mathrm{At}(Q)}$, the sets S_1 and S_2 are* visibly equal*, denoted by $S_1 =_{\mathrm{v}} S_2$, if and only if there is a bijection $f : S_1 \to S_2$ such that for every $Y \in S_1$,*

$$Y \cap A = f(Y) \cap A. \tag{2.4}$$

Visible equivalence is generalized to visible strong equivalence as follows.

Definition 2 (Visible Strong Equivalence [33]) *Programs P and Q are* visibly strongly equivalent*, denoted $P \equiv_{\mathrm{vs}} Q$, if and only if $\mathrm{At}_{\mathrm{v}}(P) = \mathrm{At}_{\mathrm{v}}(Q)$ and $\mathrm{SM}(P \cup R) =_{\mathrm{v}} \mathrm{SM}(Q \cup R)$ for any context program R such that $(\mathrm{At}(P) \cup \mathrm{At}(Q)) \cap \mathrm{At}_{\mathrm{h}}(R) = (\mathrm{At}_{\mathrm{h}}(P) \cup \mathrm{At}_{\mathrm{h}}(Q)) \cap \mathrm{At}(R) = \emptyset$.*

The visible strong equivalence relation $\equiv_{\mathrm{vs}}$ admits a model-theoretic characterization. The characterization relies on models that are minimal with respect to a set of atoms, defined as follows.

Definition 3 *A model $M \models P$ of a program P is H-minimal for $H \subseteq \mathrm{At}(P)$ iff there is no other model $N \models P$ such that $N \setminus H = M \setminus H$ and $N \cap H \subset M \cap H$.*

For a program P we write $\mathrm{MM}_H(P)$ for the set of all H-minimal models of P. Intuitively, when H-minimal models are concerned, an interpretation over $\mathrm{At}(P)\setminus H$ uniquely fixes truth values for H, if the combination can possibly result in a model. The above gives rise to *visible strong equivalence models*.

Definition 4 ([33]) *A VSE-model of a program P is a pair $\langle X, Y\rangle$ of interpretations where $X \subseteq Y \subseteq \mathrm{At}(P)$ and $X, Y \in \mathrm{MM}_{\mathrm{At}_{\mathrm{h}}(P)}(P^Y)$.*

We write $\mathrm{VSE}(P)$ to denote the set of all VSE-models of P, which together capture the semantics of P subject to substitutions. The intuition behind a VSE-model $\langle X, Y\rangle$ is that Y takes the role of the program R giving the context in which P is reduced, and X relates to the behaviour of the reduced program P^Y. We also define the *second projection* of a set of VSE-models S to be $[S]_2 = \{Y \mid \langle X, Y\rangle \in S\}$. Using these notations, the strong equivalence of programs can be studied by comparing their VSE-models.

Definition 5 *Given programs P and Q with $A = \mathrm{At_v}(P) = \mathrm{At_v}(Q)$, the respective sets* $\mathrm{VSE}(P)$ *and* $\mathrm{VSE}(Q)$ visibly match, *denoted* $\mathrm{VSE}(P) \overset{\mathrm{v}}{=} \mathrm{VSE}(Q)$, *if and only if* $[\mathrm{VSE}(P)]_2 =_{\mathrm{v}} [\mathrm{VSE}(P)]_2$ *via a bijection* $f : [\mathrm{VSE}(P)]_2 \to [\mathrm{VSE}(Q)]_2$ *such that for every* $Y \in [\mathrm{VSE}(P)]_2$,

$$\{X \cap A \mid \langle X, Y\rangle \in \mathrm{VSE}(P)\} = \{X \cap A \mid \langle X, f(Y)\rangle \in \mathrm{VSE}(Q)\}. \qquad (2.5)$$

The above definition is illustrated by an example.

Example 4 *For programs*

$$P = \{a \leftarrow 3 \leq \langle b = 1, c = 2, d = 2\rangle.\ \} \textit{ and}$$
$$Q = \{a \leftarrow c, d.\quad a \leftarrow b, x.\quad x \leftarrow c.\quad x \leftarrow d.\},$$

there is a bijection between $[\mathrm{VSE}(P)]_2$ *and* $[\mathrm{VSE}(Q)]_2$ *that proves* $\mathrm{VSE}(P) \overset{\mathrm{v}}{=} \mathrm{VSE}(Q)$, *when* $\mathrm{At_v}(P) = \mathrm{At_v}(Q) = \{a, b, c, d\}$ *and x is subject to minimization, as shown below.*

X	Y	Y'	X'
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\emptyset, \{a\}$	$\{a\}$	$\{a\}$	$\emptyset, \{a\}$
$\emptyset, \{a\}, \{b\}, \{a, b\}$	$\{a, b\}$	$\{a, b\}$	$\emptyset, \{a\}, \{b\}, \{a, b\}$
$\emptyset, \{a\}, \{c\}, \{a, c\}$	$\{a, c\}$	$\{a, c, x\}$	$\emptyset, \{a\}, \{c, x\}, \{a, c, x\}$
$\emptyset, \{a\}, \{d\}, \{a, d\}$	$\{a, d\}$	$\{a, d, x\}$	$\emptyset, \{a\}, \{d, x\}, \{a, d, x\}$
$\emptyset, \{b\}$	$\{b\}$	$\{b\}$	$\emptyset, \{b\}$
$\emptyset, \{c\}$	$\{c\}$	$\{c, x\}$	$\emptyset, \{c, x\}$
$\emptyset, \{d\}$	$\{d\}$	$\{d, x\}$	$\emptyset, \{d, x\}$
$\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{a, b, c\}$	$\{a, b, c\}$	$\{a, b, c, x\}$	$\emptyset, \{a\}, \{b\}, \{c, x\}, \{a, b\}, \{a, c, x\}, \{a, b, c, x\}$
$\emptyset, \{a\}, \{b\}, \{d\}, \{a, b\}, \{a, d\}, \{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d, x\}$	$\emptyset, \{a\}, \{b\}, \{d, x\}, \{a, b\}, \{a, d, x\}, \{a, b, d, x\}$
$\emptyset, \{a\}, \{c\}, \{d\}, \{a, c\}, \{a, d\}, \{a, c, d\}$	$\{a, c, d\}$	$\{a, c, d, x\}$	$\emptyset, \{a\}, \{c, x\}, \{d, x\}, \{a, c, x\}, \{a, d, x\}, \{a, c, d, x\}$
$\emptyset, \{a\}, \{b\}, \{c\}, \{d\}, \{a, b\}, \{a, c\}, \{a, d\}, \{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d, x\}$	$\emptyset, \{a\}, \{b\}, \{c, x\}, \{d, x\}, \{a, b\}, \{a, c, x\}, \{a, d, x\}, \{a, b, c, x\}, \{a, b, d, x\}, \{a, c, d, x\}, \{a, b, c, d, x\}$

The relation $\overset{\mathrm{v}}{=}$ in Definition 5 is particularly meaningful due to the following result.

Proposition 1 (Characterization of Visible Strong Equivalence [33])
For programs P and Q with $\mathrm{At_v}(P) = \mathrm{At_v}(Q)$, $\mathrm{VSE}(P) \overset{\mathrm{v}}{=} \mathrm{VSE}(Q)$ *implies* $P \equiv_{\mathrm{vs}} Q$.

In consequence, the behaviour of programs, under any context allowed by Definition 2, can be compared on the basis of their VSE-models. This provides a tool for arguing about the correctness of translations. For the purposes of this thesis, we develop further methods applicable to restricted classes of programs. First, a more lenient matching criterion is defined for positive programs, for which sets of *hidden minimal models* contain sufficient information for the purpose of demonstrating visible strong equivalence.

Proposition 2 *For positive programs P and Q with $\mathrm{At_v}(P) = \mathrm{At_v}(Q)$, it holds that $\mathrm{VSE}(P) \overset{\mathrm{v}}{=} \mathrm{VSE}(Q)$ if and only if $\mathrm{MM}_{\mathrm{At_h}(P)}(P) =_{\mathrm{v}} \mathrm{MM}_{\mathrm{At_h}(Q)}(Q)$.*

Proof The domains and ranges of the bijections for proving $\overset{\mathrm{v}}{=}$ and $=_{\mathrm{v}}$ coincide, such that $[\mathrm{VSE}(P)]_2 = \mathrm{MM}_{\mathrm{At_h}(P)}(P)$ and $[\mathrm{VSE}(Q)]_2 = \mathrm{MM}_{\mathrm{At_h}(Q)}(Q)$. Moreover, the "only if" part follows from Definition 5. Let us thus consider the other direction, take a bjiection f as in Definition 1, and prove the condition (2.5) to hold for f. For simplicity, let us denote $A = \mathrm{At_v}(P) = \mathrm{At_v}(Q)$. For every $Y \in \mathrm{MM}_{\mathrm{At_h}(P)}(P)$, $Y \cap A = f(Y) \cap A$ and thus

$$\{X \cap A \mid X \subseteq Y\} = \{X \cap A \mid X \subseteq f(Y)\}.$$

Moreover,

$$\{X \cap A \mid X \in \mathrm{MM}_{\mathrm{At_h}(P)}(P)\} = \{f(X) \cap A \mid X \in \mathrm{MM}_{\mathrm{At_h}(P)}(P)\}, \text{ and}$$
$$\{X \cap A \mid X \in \mathrm{MM}_{\mathrm{At_h}(P)}(P)\} = \{X \cap A \mid X \in \mathrm{MM}_{\mathrm{At_h}(Q)}(Q)\}.$$

Thus in combination,

$$\begin{aligned}\{X \cap A \mid X \subseteq Y, \quad X \in \mathrm{MM}_{\mathrm{At_h}(P)}(P)\} = \\ \{X \cap A \mid X \subseteq f(Y), X \in \mathrm{MM}_{\mathrm{At_h}(Q)}(Q)\}.\end{aligned}$$

Given that P and Q are positive, (2.5) follows.

In conclusion, there is a bijection proving $\mathrm{VSE}(P) \overset{\mathrm{v}}{=} \mathrm{VSE}(Q)$ if and only if there is a bijection to prove $\mathrm{MM}_{\mathrm{At_h}(P)}(P) =_{\mathrm{v}} \mathrm{MM}_{\mathrm{At_h}(Q)}(Q)$. □

In order to apply Proposition 2 more widely, and not only for purely positive programs, we make use of a lemma in the following. To this end, we first present notation for substituting literals in programs. For a program P and a *substitution pair* l/l' composed of literals l and l', we write $P[l/l']$ to denote the program obtained from P by substituting every non-negated occurrence of l in P with l'. For example,

$$\{a \leftarrow 1 \leq \langle a, {\sim}a\rangle.\}[{\sim}a/b] = \{a \leftarrow 1 \leq \langle a, b\rangle.\}.$$

We extend this notation to a set of substitution pairs $A = \{l_1/l'_1, \ldots, l_n/l'_n\}$ by writing $P[A]$ for the program $P[l_1/l'_1] \cdots [l_n/l'_n]$, under the conditions that $\mathrm{At}(l_i) \cap \mathrm{At}(l'_j) = \emptyset$ for $i, j \in \{1, \ldots, n\}$ and $\mathrm{At}(l_i) \cap \mathrm{At}(l_j) = \mathrm{At}(l'_i) \cap \mathrm{At}(l'_j) = \emptyset$ when $i \neq j$. The preconditions assure that the outcome is invariant to the order of the substitutions. In certain settings, the effect of a substitution can be counterbalanced by the addition of an appropriate rule.

Lemma 1 *For a program P including atom $a \in \mathrm{At}_{\sim}(P) \cap \mathrm{At}_{\mathrm{v}}(P)$, an atom $\overline{a} \notin \mathrm{At}(P)$, and the program $Q = P[\sim a/\overline{a}] \cup \{\overline{a} \leftarrow \sim a.\}$ with $\mathrm{At}_{\mathrm{v}}(P) = \mathrm{At}_{\mathrm{v}}(Q)$, it holds that $P \equiv_{\mathrm{vs}} Q$.*

Proof Let $f : [\mathrm{VSE}(P)]_2 \to [\mathrm{VSE}(Q)]_2$ be the bijection $f(Y) = Y \cup \{\overline{a} \mid a \notin Y\}$. For every $Y \in [\mathrm{VSE}(P)]_2$, it holds that $f(Y) \cap \mathrm{At}_{\mathrm{v}}(P) = (Y \cap \mathrm{At}_{\mathrm{v}}(P)) \cup (\{\overline{a} \mid a \notin Y\} \cap \mathrm{At}_{\mathrm{v}}(P)) = Y \cap \mathrm{At}_{\mathrm{v}}(P)$. Thus f satisfies (2.4).

Now if $a \in Y$, then $\overline{a} \notin f(Y) = Y$, $Q^{f(Y)} = P[\sim a/\overline{a}]^Y$, and specifically, since $\mathrm{At}_{\mathrm{h}}(Q) = \mathrm{At}_{\mathrm{h}}(P) \cup \{\overline{a}\}$, and $a \in \mathrm{At}_{\mathrm{v}}(P)$ is not subject to minimization,

$$\mathrm{MM}_{\mathrm{At}_{\mathrm{h}}(Q)}(Q^{f(Y)}) = \mathrm{MM}_{\mathrm{At}_{\mathrm{h}}(P)}(P^Y). \tag{2.6}$$

Also, if $a \notin Y$ then $\overline{a} \in f(Y) = Y \cup \{\overline{a}\}$, $Q^{f(Y)} = P[\sim a/\overline{a}]^Y \cup \{\overline{a}.\}$, and in particular

$$\begin{aligned} \{X \cap \mathrm{At}_{\mathrm{v}}(P) \mid \quad & X \in \mathrm{MM}_{\mathrm{At}_{\mathrm{h}}(Q)}(Q^{f(Y)})\} \quad = \\ \{X \cap \mathrm{At}_{\mathrm{v}}(P) \mid \quad & X \in \mathrm{MM}_{\mathrm{At}_{\mathrm{h}}(P)}(P^Y)\}. \end{aligned} \tag{2.7}$$

Thus for every Y, either (2.6) or (2.7) holds. Consequently,

$$\begin{array}{llll} \{X \cap \mathrm{At}_{\mathrm{v}}(A) \mid & & \langle X, f(Y)\rangle \in \mathrm{VSE}(Q)\} & = \\ \{X \cap \mathrm{At}_{\mathrm{v}}(A) \mid & X \subseteq Y, & X, Y \in \mathrm{MM}_{\mathrm{At}_{\mathrm{h}}(Q)}(Q^{f(Y)})\} & = \\ \{X \cap \mathrm{At}_{\mathrm{v}}(A) \mid & & \langle X, Y\rangle \in \mathrm{VSE}(P)\}. & \end{array}$$

Thus f satisfies (2.5). □

An example substitution and its correctness are demonstrated in the following.

Example 5 *Given programs $P = \{a \leftarrow \sim b.\}$ and $Q = P[\sim b/\overline{b}] \cup \{\overline{b} \leftarrow \sim b.\} = \{a \leftarrow \overline{b}.\ \overline{b} \leftarrow \sim b.\}$, if we select $\mathrm{At}_{\mathrm{v}}(P) = \mathrm{At}_{\mathrm{v}}(Q) = \{b\}$, then $\overline{b} \notin \mathrm{At}_{\mathrm{v}}(P)$ and by Lemma 1 we have that $P \equiv_{\mathrm{vs}} Q$. To verify this, observe that the VSE-models of P and Q visibly match via the mapping of $Y \in [\mathrm{VSE}(P)]_2$ to $Y' \in [\mathrm{VSE}(Q)]_2$ shown below.*

X	Y		Y'	X'
$\{a\}$	$\{a\}$		$\{a, \overline{b}\}$	$\{a, \overline{b}\}$
$\emptyset, \{b\}$	$\{b\}$		$\{b\}$	$\emptyset, \{b\}$

If we on the other hand select $\mathrm{At_v}(P) = \mathrm{At_v}(Q) = \{a\}$*, then* $\mathrm{MM}_{\mathrm{At_h}(P)}(P) = \{\{a\}, \{b\}\}$ *and* $\mathrm{MM}_{\mathrm{At_h}(Q)}(Q) = \{\{a, b\}, \{a, \overline{b}\}, \{b\}\}$*, implying that* $\mathrm{VSE}(P) \overset{\mathrm{v}}{\neq} \mathrm{VSE}(Q)$*. Therefore* $P \not\equiv_{\mathrm{vs}} Q$ *[33]. This kind of visible duplication of models is avoided in Lemma 1 by requiring the substituted atom to be visible.* ■

Different partitions of the signature of a program to visible and hidden parts are possible. With an abuse of notation, we write $P = P'$ for two programs consisting of the same set of rules, but having potentially different signatures. The visible strong equivalence of programs generally depends on those partitions but is however invariant to some changes in signatures. In particular, visible atoms common to both programs can be hidden safely as follows.

Lemma 2 *Given programs* $P = P'$ *and* $Q = Q'$ *with* $\mathrm{At}(P) = \mathrm{At}(P')$*,* $\mathrm{At}(Q) = \mathrm{At}(Q')$*, and* $\mathrm{At_v}(P') = \mathrm{At_v}(Q') \subseteq \mathrm{At_v}(P) = \mathrm{At_v}(Q)$*,* $P \equiv_{\mathrm{vs}} Q$ *implies* $P' \equiv_{\mathrm{vs}} Q'$*.*

Proof Let R be any applicable context for P' and Q' as in Definition 2, such that

$$(\mathrm{At}(P') \cup \mathrm{At}(Q')) \cap \mathrm{At_h}(R) = (\mathrm{At_h}(P') \cup \mathrm{At_h}(Q')) \cap \mathrm{At}(R) = \emptyset$$

and thus

$$(\mathrm{At}(P) \cup \mathrm{At}(Q)) \cap \mathrm{At_h}(R) = (\mathrm{At_h}(P) \cup \mathrm{At_h}(Q)) \cap \mathrm{At}(R) = \emptyset.$$

Therefore R is also applicable for P and Q. Now let $\mathrm{SM}(P \cup R) =_{\mathrm{v}} \mathrm{SM}(Q \cup R)$ via a bijection f. For every $Y \in \mathrm{SM}(P \cup R)$,

$$Y \cap \mathrm{At_v}(P) = f(Y) \cap \mathrm{At_v}(P).$$

and also

$$Y \cap \mathrm{At_v}(P') = f(Y) \cap \mathrm{At_v}(P').$$

Therefore $P \equiv_{\mathrm{vs}} Q$ implies $P' \equiv_{\mathrm{vs}} Q'$. □

Now if we have programs P and Q that agree on both $N = \mathrm{At}_\sim(P) = \mathrm{At}_\sim(Q)$ and $A = \mathrm{At_v}(P) = \mathrm{At_v}(Q)$, while satisfying $N \subseteq A$, then we may use the following strategy for checking whether $P \equiv_{\mathrm{vs}} Q$.

1. For each atom $a \in N$, we substitute $\sim a$ in both P and Q with a new atom, say $\overline{a}$. In notation, we form the substitution pairs $S = \{\sim a/\overline{a} \mid a \in N\}$, and then the programs $P[S]$ and $Q[S]$.
2. The set of rules $R = \{\overline{a} \leftarrow \sim a. \mid a \in N\}$ is formed to compensate for the substitutions.

3. By repeated application of Lemma 1, it can be shown that $P \equiv_{\mathrm{vs}} P[S] \cup R$ and $Q \equiv_{\mathrm{vs}} Q[S] \cup R$.

4. Now showing that $P[S] \equiv_{\mathrm{vs}} Q[S]$ also proves that $P \cup R \equiv_{\mathrm{vs}} Q \cup R$, when all new atoms $\overline{a}$ are visible in each program.

5. By Lemma 2, it follows as well that $P \cup R \equiv_{\mathrm{vs}} Q \cup R$ when every $\overline{a}$ is hidden.

6. The above finally entails that $P \equiv_{\mathrm{vs}} Q$, by Proposition 1.

The benefit of this approach lies on the fact that $P[S]$ and $Q[S]$ are positive, and thus Proposition 2 can be applied in proving $P[S] \equiv_{\mathrm{vs}} Q[S]$. The strategy is summarized as follows.

Proposition 3 *For programs P and Q with $N = \mathrm{At}_{\sim}(P) = \mathrm{At}_{\sim}(Q)$, and $A = \mathrm{At}_{\mathrm{v}}(P) = \mathrm{At}_{\mathrm{v}}(Q)$ such that $N \subseteq A$, and for the substitution pairs $S = \{\sim a/\overline{a} \mid a \in N\}$, it holds that $\mathrm{MM}_{\mathrm{At}_{\mathrm{h}}(P)}(P[S]) =_{\mathrm{v}} \mathrm{MM}_{\mathrm{At}_{\mathrm{h}}(Q)}(Q[S])$ implies $\mathrm{VSE}(P) \overset{\mathrm{v}}{=} \mathrm{VSE}(Q)$.*

2.3 Literal Number Representations

Answer set programs support literals in the two-valued Boolean domain. Additionally, integers are used for weights of such literals within *weighted expressions*. In assisting different normalizations of weight rules, we will further make use of two extended representations of literals in programs, which, in their separate ways, allow us to operate on literal numbers in larger finite domains as well. The first of these representations is covered in this section and is that of unary numbers, and more specifically unary numbers encoded in terms of literals. In the other representation, numbers are decomposed in a mixed-radix base and their digits are expressed in unary notation. Mixed-radix numbers will be discussed in Section 4.1. Either representation of finite domain variables is naturally founded on the two-valued literals natively available in ASP. The range of available literal domains in a program is then, in a way, expanded, to ease the encoding of different rules using auxiliary variables.

We deal with unary numbers as follows. Given an interpretation M, a sequence of literals $L = \langle l_1, \ldots, l_n \rangle$ is associated with the number $k \in \{0, \ldots, n\}$ if l_i is satisfied for each $i \leq k$ but for no $i > k$. If this holds for every interpretation M under consideration, such as every stable model of a program at hand, we call L a *unary literal number* with *value* k, and each l_i is identified as its i^{th} digit. In other terms, a unary literal number is a sequence of literals in descending order of truth values, with a value given by the satisfied cardinality $v_M(L)$. We denote the set of atoms underlying a unary literal number $L = \langle l_1, \dots, l_n \rangle$ with $\mathrm{At}(L) = \{a \mid \exists i \in \{1, \dots, n\} : l_i = a$ or $l_i = {\sim}a\}$. Literal numbers of this form can be produced by means of *sorting* and *merging* discussed in Chapter 3. In the rest of this section we define simple operations on unary numbers.

A constant multiple of a unary number can be obtained, and used, for example, to represent numbers in domains of the form $\{0, n\}$. Indeed, for the k-fold multiple of a literal l, we write

$$l^k = \underbrace{l, \dots, l}_{k \text{ times}}.$$

For example, the unary literal number $\langle a^3 \rangle = \langle a, a, a \rangle$ for some atom a represents a variable in $\{0, 3\}$. More generally, for an arbitrary unary literal number $\langle l_1, \dots, l_n \rangle$ and a constant integer k we define multiplication according to

$$\langle l_1, \dots, l_n \rangle \times k = \langle l_1^k, \dots, l_n^k \rangle.$$

A unary literal number can likewise be easily incremented by a constant. We denote an arbitrary fact by $\top$ and write that

$$L + k = \langle \top^k, l_1, \dots, l_n \rangle.$$

Furthermore, a division operation with a constant can be resolved without auxiliary rules. Given a unary literal number L as before and a constant k, their division involves picking every k^{th} literal of L, such that

$$L/k = \langle l_k, l_{2k}, \dots, l_{k \times \lfloor n/k \rfloor} \rangle.$$

Unlike the presented operations between a unary literal number and a constant, the calculation of the residue $A \bmod k$ of a unary literal number $A = \langle a_1, \dots, a_n \rangle$ consisting of atoms only and a divisor k, involves rules. We assume that $k \leq n$, let $m = k \times \lfloor n/k \rfloor$, and encode the residue in $R = \langle r_1, \dots, r_{d-1} \rangle = A \bmod k$, by using the program

$$\begin{aligned}
&\mathsf{Modulo}_k(A, R) = \\
&\qquad \{r_i \leftarrow a_{j \times k + i}, {\sim}a_{k \times \lceil j/k \rceil}. &&\mid 1 \leq i < k, 1 \leq j \leq m\} \cup \\
&\qquad \{r_{i - m \times k} \leftarrow a_i. &&\mid m \times k < i \leq n\}.
\end{aligned}$$

Finally, a unary number can be trivially compared with a constant. This follows directly from their definition and the consequent interpretation of the i^{th} digit l_i of a unary literal number. Namely, the digit l_i determines whether the number is greater than or equal to i.

Chapter 3

Existing Weight Rule Normalizations

In this chapter we review a variety of translations of cardinality and weight rules. We begin by inspecting the former case here, and then broaden our view by looking into different translations as well as weight rules in the following sections. In the translation of a cardinality rule, we seek to substitute it with a set of normal rules. The resulting normal logic program has to share the semantic properties of the original rule. To aid the understanding and engineering of such operations the resulting sets of primitives can be conceptually and recursively partitioned into larger building blocks. For example, a substituting program can represent a counting circuit consisting of several smaller subcircuits that build up the number of satisfied body literals in some number system.

Intuitively, the set of body literals in a cardinality rule encodes the number of satisfied literals among them, say s. To perform a *cardinality check*, that is, to see whether $k \leq s$, we can directly extract the required information from the body literals by enumerating all of their k-subsets, as discussed in Section 1.1. If at least one body consists of satisfied literals only, the check succeeds. All cardinality rule translations in general answer this question by extracting the same information. In place of direct enumeration, however, they construct more suitable representations of the number s, to the extent required to check whether $k \leq s$. This is common to other extended rule translations as well. In such, information concerning a property, here s, is preserved through operations that mold it into a more suitable form. The eventual representation then either reveals the sought answer immediately, or lessens the burden of doing so.

In view of the above, there are a number of relevant and noteworthy representations to be mentioned here. First we consider, in Section 3.1, the use of a unary encoding in representing the number of satisfied body literals s. Such a representation is indeed applicable to cardinality checking, for if we have a unary literal representation $\langle l_1, \ldots, l_n \rangle$ of s, we may derive the head of the cardinality rule from l_k. This follows directly from our interpretation of such literals given in

Section 2.3. Second, counting in unary, as above, coincides with sorting binary values. Indeed, if we have variables whose values fall in two classes, one less than the other, then the result of sorting them gives a vector representing a unary number. In our case with literals, sorting in the suitable order reveals, again, the cardinality of those literals. In Section 3.3 we give an overview of merge-sorting based techniques to this end. In addition to these two views on unary numbers, the binary number system has been utilized in cardinality constraint translations. Such translations are out of the scope of this thesis. In the case where Boolean satisfiability is the target language, they have been found to be outperformed by synthetizations of *monotone circuits*, despite the latters' greater sizes.

The semantics we pursue in the case of counting and sorting is derived from that of cardinality rules. We require that a sorter of n literals must be interchangeable with a set of n appropriately selected cardinality rules. Relying on the notion of visible strong equivalence [33], we capture this goal by asserting that, for input literals $\{l_1, \ldots, l_n\}$ and output atoms $\langle h_1, \ldots, h_n \rangle$,

$$\mathsf{Sorter}_n(\{l_1, \ldots, l_n\}, \langle h_1, \ldots, h_n \rangle) \equiv_{\mathrm{vs}} \left\{ \begin{array}{c} h_1 \leftarrow 1 \leq \langle l_1, \ldots, l_n \rangle. \\ h_2 \leftarrow 2 \leq \langle l_1, \ldots, l_n \rangle. \\ \vdots \\ h_n \leftarrow n \leq \langle l_1, \ldots, l_n \rangle. \end{array} \right\}. \tag{3.1}$$

The formulation of the above equivalence reveals a favourable property of this type of an approach. That is, given a sorter abiding to (3.1), we can not only substitute an encountered cardinality rule $h_i \leftarrow i \leq \langle l_1, \ldots, l_n \rangle$ on the right-hand side with the sorter on the left-hand side, but also a whole set of cardinality rules differing only in their bounds.

In the rest of this chapter, we first cover a technique of sequential counting for cardinality and weight rules in Sections 3.1 and 3.2. Then we review a more concise, merge-sorting based translations for cardinality rules in Section 3.3. We finish this chapter by discussing a straightforward but naïve generalization of sorting techniques from cardinality to weight rules.

3.1 Sequential Counting

In this section we follow [42] in encoding *sequential counters*, but where clauses were used for the original SAT encoding, we use the appropriate, corresponding rules for ASP. We first describe the design in increasing granularity, after which we formally define it bottom-up. The result closely resembles an ASP translation in a grid formation studied in [41] and [33].

The sequential counter program is constructed to take n input literals $l_1, \ldots, l_n$, and to count the number of true literals among them up to a specified threshold k. The resulting number is specified in unary notation using output literals $s_{n,1}, \ldots, s_{n,k}$. The program is built from a sequence of layers, one for each input literal l_i. The layer corresponding to l_i determines a partial count, encoded in $S_i = \langle s_{i,1}, \ldots, s_{i,k} \rangle$, in terms of l_i and the previous layer's output S_{i-1}. The purpose of the layer is to incorporate l_i to the total count, which is higher by one if l_i is satisfied than if not. By equating false with 0 and true with 1, we can express the i^{th} partial count as $\sum_{j=1}^{i} l_j$. The corresponding literal sequence S_i encodes this value in unary notation, such that, for a stable model M, the said count is $v_M(S_i)$. Thus the last, or n^{th}, layer ends up representing the sum $\sum_{j=1}^{n} l_j$, that is, the cardinality of the set of input signals. This sum can then be compared with any limit $k' \leq k$, or limits, in order to obtain the wanted output; and usually $k' = k$.

As the sequential counter can be split into the above described layers, the layers can in turn be divided further. We express them in terms of small if-then-else (ITE) programs. These ITEs can be viewed as circuits of AND and OR gates, for example, or straight away as clauses or rules. Hence, the sequential counter can be regarded as a hierarchical specification of a normal logic program. An ITE program has for input a *control* literal and two other literals. Its output is an atom that takes the value of one of the two other literals. When the control signal is true, the output is determined by the first of the two, and when false, by the second. In Figure 3.1, two designs for an ITE are displayed on the left and right sides of an abstract symbol denoting the ITE. The designs are given in terms of logical AND and OR gates, which are easily expressible in ASP. The larger of the two is a generally applicable version, whereas the more succint one is not. Namely, the latter has a prerequisite, by which, out of the two input literals under selection, the second one must imply the first. This condition is satisfied in our use case, and thus the smaller, simplified ITE is applicable.

The rules encoding a simplified ITE are provided in the following definition.

Definition 6 *The if-then-else program for an input literal l, atoms a_1 and a_2, of which a_2 implies a_1, and an output atom s, is*

$$\mathsf{Ite}(l, a_1, a_2, s) = \left\{ \begin{array}{l} s \leftarrow a_1,\ l. \\ s \leftarrow a_2. \end{array} \right\}. \tag{3.2}$$

We can now use (3.2) to build the layers of a sequential counter. In each, the associated input signal l_i is used as the control variable for a series of ITE programs. When it is false, the unary representation of the count from layer $i - 1$ is only copied to the output, and when it is true, the count is also incremented by shifting the representation to the "right" by one. Two example layers are shown on

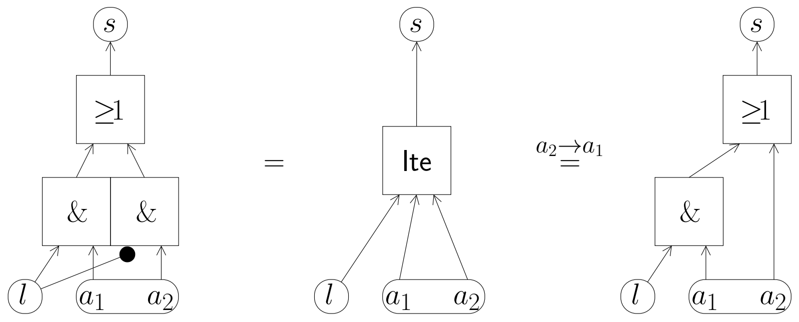


Figure 3.1: An if-then-else circuit in terms of logical gates on the left, displayed abstractly in the middle, and in a simplified form on the right. The circle arrow head stands for negation. The equality on the right holds only on the condition that l_2 implies l_1.

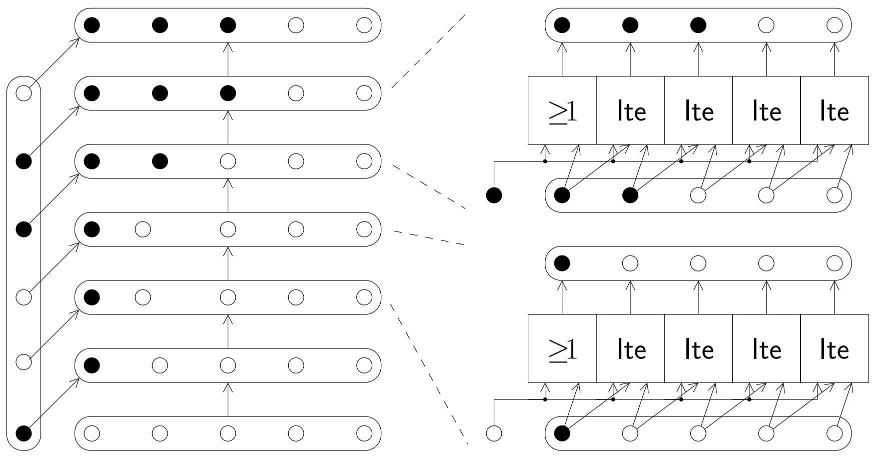


Figure 3.2: Literal values for a sequential counter on the left and the ITE-based designs for two of its layers on the right. Filled circles denote derived literals.

the right of Figure 3.2. The copying and incrementing behaviour are seen in the lower and upper examples, respectively. We define the rules of a layer as follows.

Definition 7 *The layer of a sequential counter program for an input literal l and atoms $a_1, \ldots, a_k$, of which each a_i with $i > 1$ implies a_{i-1}, and output atoms $s_1, \ldots, s_k$, is*

$$\begin{aligned}&\mathsf{SeqLayer}_k(l, \langle a_1, \ldots, a_k \rangle, \langle s_1, \ldots, s_k \rangle) = \\ &\quad \{s_1 \leftarrow l.\ s_1 \leftarrow a_1.\} \cup \bigcup_{i=2}^{k} \mathsf{Ite}(l, a_{i-1}, a_i, s_i).\end{aligned} \tag{3.3}$$

Except for the first, each output s_i above is defined by an ITE program, which forces the value of either the lesser input a_{i-1} to be shifted to the output or the default input a_i to be carried on to the output. The final construction of a sequential counter is a matter of joining appropriate layers together. Figure 3.2 gives an example of a combination of layers comprising a sequential counter of $n = 6$ inputs that counts up to $k = 5$. This process is formalized in the following definition.

Definition 8 *The sequential counter program for input literals $l_1, \ldots, l_n$ and output atoms $s_{n,1}, \ldots, s_{n,k}$ is*

$$\begin{aligned}&\mathsf{Seq}_{n,k}(\langle l_1, \ldots, l_n \rangle, \langle s_{n,i}, \ldots, s_{n,k} \rangle) = \\ &\quad \bigcup_{i=1}^{n} \mathsf{SeqLayer}_k(l_i, \langle s_{i-1,1}, \ldots, s_{i-1,k} \rangle, \langle s_{i,1}, \ldots, s_{i,k} \rangle).\end{aligned} \tag{3.4}$$

In (3.4), for $i \in \{1, \ldots, n-1\}$, the symbols $s_{i,1}, \ldots, s_{i,k}$ reference auxiliary atoms used to capture intermediary inputs to layers. We demonstrate the presented definitions by an example, which we also use to introduce partial symbolic evaluation in this context.

Example 6 *The sequential counter $\mathsf{Seq}_{4,3}(\langle b_1, b_2, b_3, {\sim}c_1 \rangle, \langle s_{4,1}, s_{4,2}, s_{4,3} \rangle)$ consists of the following rules:*

$s_{4,1} \leftarrow {\sim}c_1.$ $s_{4,1} \leftarrow s_{3,1}.$	$s_{4,2} \leftarrow s_{3,1}, {\sim}c_1.$ $s_{4,2} \leftarrow s_{3,2}.$	$s_{4,3} \leftarrow s_{3,2}, {\sim}c_1.$ $s_{4,3} \leftarrow s_{3,3}.$
$s_{3,1} \leftarrow b_3.$ $s_{3,1} \leftarrow s_{2,1}.$	$s_{3,2} \leftarrow s_{2,1}, b_3.$ $s_{3,2} \leftarrow s_{2,2}.$	$s_{3,3} \leftarrow s_{2,2}, b_3.$ $s_{3,3} \leftarrow s_{2,3}.$
$s_{2,1} \leftarrow b_2.$ $s_{2,1} \leftarrow s_{1,1}.$	$s_{2,2} \leftarrow s_{1,1}, b_2.$ $s_{2,2} \leftarrow s_{1,2}.$	$s_{2,3} \leftarrow s_{1,2}, b_2.$ $s_{2,3} \leftarrow s_{1,3}.$
$s_{1,1} \leftarrow b_1.$		

The rules on the top left, outside the dashed diagonal are unnecessary in the case that only the last output $s_{4,3}$ is of interest, and can be dropped in that case. The rules outside on the bottom right can be likewise omitted, since the auxiliary atoms defined by them can never be proven true. Simplification steps such as these constitute forms of partial symbolic evaluation that can often be used to prune redundant or unnecessary parts of a program. ■

Due to the used unary representation, a sequential counter essentially sorts its inputs up to some number k, and thus is applicable for translating cardinality rules. Thus if we have a rule (2.1) with only unit weights, that is, $a \leftarrow k \leq \langle L = W \rangle$ with $W = \langle 1, \ldots, 1 \rangle$, and an auxiliary sequence of atoms $S = \langle s_1, \ldots, s_k \rangle$ up to the bound k, we can substitute the original rule with the program $\mathsf{Seq}_{n,k}(L, S) \cup \{a \leftarrow s_k.\}$. This amounts to normalizing the rule into an NLP. Regarding translation size, the counter consists of n layers, each containing k head atoms. A constant number of rules are used to define each head atom. Thus the total number of both atoms and rules required by the translation are $\mathcal{O}(k \times n)$.

3.2 Weighted Sequential Counting

The sequential counter translation of [42], covered in Section 3.1, deals with expressing cardinality constraints. The translation lends itself to weight rules as well, in the form of a generalization used both in ASP [24] and SAT [31]. In this section, we first describe the generalized translation via dynamic programming, and then formally define the translation similarly to the weightless case.

The dynamic programming approach behind sequential weight counters stems from the following observation: in order for the satisfied weight sum to be no less than a given bound, it must either hold that the satisfied sum of all except any chosen input literal already satisfies the condition, or the excluded literal is satisfied and its weight added to the satisfied weight sum of the rest overcomes the bound. In both cases a smaller subproblem is left to be checked, and we may consider them and further subproblems recursively. In the worst case such binary branching leads to an exponential number of subproblems on a branching depth, of which there is a linear number. The combinatorial explosion is, however, bounded by the number of distinct sums no greater than the bound that can be formed from the weights. Consequently, many of the otherwise exponential number of cases tend to coincide, and the total number of subproblems is in $\mathcal{O}(k \times n)$.

Sequential weight counters can be viewed similarly to sequential counters when we relate a level in the discussed branching tree with a layer of the counter. Consequently, we encode them similarly, beginning with the layers.

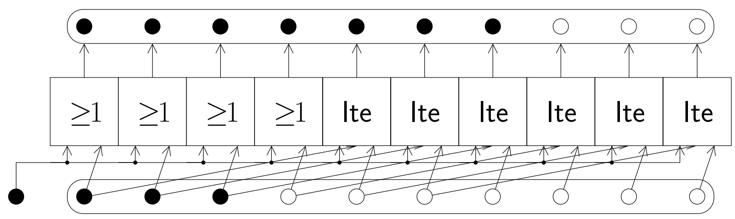


Figure 3.3: The layer $\mathsf{WSeqLayer}_{10,4}(l, \langle a_1, \dots, a_{10}\rangle, \langle s_1, \dots, s_{10}\rangle)$ of a sequential weight counter. In this case, the control literal on the left is derived, and causes the encoded output value to exceed the input by the specified weight 4.

Definition 9 *The layer of a sequential weight counter program for an input literal l and atoms $a_1, \dots, a_k$, of which each a_i with $i > 1$ implies a_{i-1}, and output atoms $s_1, \dots, s_k$, is*

$$\mathsf{WSeqLayer}_{k,w}(l, \langle a_1, \dots, a_k\rangle, \langle s_1, \dots, s_k\rangle) = \bigcup_{i=1}^{w} \{s_i \leftarrow l.\ s_i \leftarrow a_i.\} \ \cup \ \bigcup_{i=w+1}^{k} \mathsf{Ite}(l, a_{i-w}, a_i, s_i). \tag{3.5}$$

In comparison to Definition 3.3, the effect of the control literal l is effectively magnified by w due to the use of a_{i-w} in place of a_{i-1}. This change is also reflected in Figure 3.3, in contrast with Figure 3.2. Also note that, in case $w = 1$, Definitions 3.5 and 3.3 coincide. The layers are joined together to form the complete counter as follows.

Definition 10 *The sequential weight counter program for weighted input literals $l_1 = w_1, \dots, l_n = w_n$ and output atoms $s_{n,1}, \dots, s_{n,k}$ is*

$$\mathsf{WSeq}_{n,k}(\langle l_1, \dots, l_n\rangle, \langle s_{n,i}, \dots, s_{n,k}\rangle) = \bigcup_{i=1}^{n} \mathsf{WSeqLayer}_{k,w_i}(l_i, \langle s_{i-1,1}, \dots, s_{i-1,k}\rangle, \langle s_{i,1}, \dots, s_{i,k}\rangle). \tag{3.6}$$

In (3.6), for $i \in \{1, \dots, n-1\}$, the symbols $s_{i,1}, \dots, s_{i,k}$ reference auxiliary atoms as in (3.4). The analysis and use of the sequential weight coounter follow those of the weightless case, and again, the translation size is $\mathcal{O}(k \times n)$.

3.3 Merge Sorting

In this section we introduce merge-sorting in the context of logic programs and cardinality rule normalizations. The result of the normalization is a logic program designed after a circuit that conceptually sorts binary signals. This coincides with the input-output contract of a sequential counter, which counts the true inputs in unary notation. Suitable choices for the underlying circuit include sorters based on odd-even mergers [8], pairwise mergers [39], and totalizers [7].

Merging is a process involving three sorted sequences, two of which are combined into the third output sequence. The result gives the concatenation of the two inputs in sorted order. A sequence of numbers can be arranged to be sorted by recursively merging parts of it together. Given a list of numbers, one may split it in two, sort both halves and then merge the halves together. In this manner the original problem is divided into smaller sub-problems which can in turn be solved recursively. Thus, sorting can be implemented by the way of merging. Specifically, we may construct circuits to perform merging and, by extension, sorting. These circuits can then be encoded into normal logic programs. In what follows, a few different types of mergers are discussed.

Batcher [8] introduced odd-even mergers built of *comparators*. Already on its own, an odd-even merger is a recursive construction. It consists of a logarithmic number of levels in terms of its input length. On each level there are two input sequences to be merged together. First, their elements are split into four sequences by separating elements with odd and even indices from each other. The process keeps the order of elements within the new sequences consistent with their original order. Thus we obtain a left-odd sequence, a right-odd sequence, a left-even sequence and a right-even sequence. Afterward, the two odd sequences are merged together as are the even ones. Finally, the two resulting sequences are combined with what can be characterized as a *balanced merger* [15]: a vector of comparators, each producing two consecutive output values.

The balanced merger is another construction that produces sorted sequences. Whereas a merger can be thought of as a sorter with preconditions, a balanced merger can in turn be described as a merger with additional preconditions. Their inputs must be balanced: the left sequence can contain at most two 1's more than the right one, and the right sequence can contain at most as many 1's as the left one. With these preconditions satisfied, the balanced merger can be kept flat and of linear size in its combined input length. In summary then, a *merge-sorter* consists of a merger and two smaller sorters, and a merger consists of a balanced merger and two smaller mergers. The total number of comparators required to construct a merge-sorter for n inputs is in $\mathcal{O}(n \times (\log n)^2)$. When only the k^{th} output is to be derived, $\mathcal{O}(n \times (\log k)^2)$ comparators suffice [14]. Examples of a merger and a sorter are given in Figure 3.4.

We define odd-even sorters in terms of *comparators*.

Definition 11 *The* comparator *from the input literals l_1 and l_2 to the output atoms*

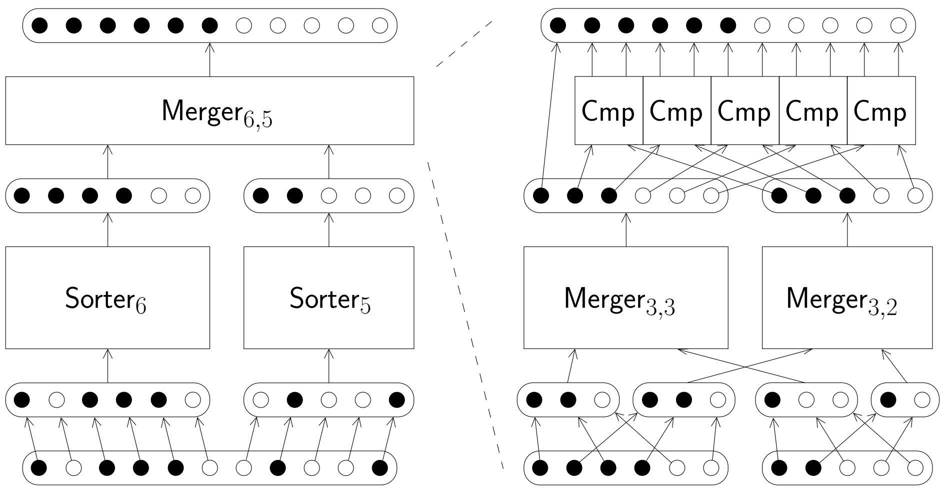


Figure 3.4: A merge-sorter on the left and an odd-even merger on the right.

s_1 and s_2 is the program

$$\mathsf{Cmp}(l_1, l_2, s_1, s_2) = \left\{ \begin{array}{l} s_1 \leftarrow l_1. \\ s_1 \leftarrow l_2. \\ s_2 \leftarrow l_1, l_2. \end{array} \right\}. \tag{3.7}$$

The purpose of a comparator is to permute the truth values of the inputs l_1 and l_2 such that the "greater" of the two is placed in s_1 and the "lesser" in s_2. Thus if neither input is satisfied, the outputs are false. If precisely one input is true, then s_1 is implied while s_2 is not. Only if both inputs are satisfied, both outputs are assigned to true. The rules in (3.7) are used to achieve this by essentially defining s_1 and s_2 to be the logical OR and AND of the inputs, respectively.

We say that a pair of numbers n and m are *balanced* if $m \leq n \leq m + 2$. Similarly, if the numbers of satisfied literals in a pair of sequences of literals are balanced, we say that the sequences are balanced. Two sorted and balanced sequences of literals can be merged using a linear number of comparators.

Definition 12 *The* balanced merger *from the sequences of literals $D = \langle d_1, \ldots, d_n \rangle$ and $E = \langle e_1, \ldots, e_m \rangle$ into the atoms $S = \langle s_1, \ldots, s_{n+m} \rangle$ is the program*

$$\begin{aligned} &\mathsf{BalancedMerger}(D, E, S) = \\ &\quad \{s_1 \leftarrow d_1.\} \\ &\quad \cup \{s_{n+m} \leftarrow d_n. \mid n = m + 2\} \\ &\quad \cup \{s_{n+m} \leftarrow e_m. \mid n = m\} \\ &\quad \cup \textstyle\bigcup_{i=1}^{\min\{n-1,m\}} \mathsf{Cmp}(d_{i+1}, e_i, s_{2i}, s_{2i+1}) \end{aligned} \tag{3.8}$$

The values of the output atoms S in (3.8) are guaranteed to be ordered from true to false under any model M of the program if the inputs D and E are sorted and balanced. In addition, the total number of true and false values are preserved, such that $v_M(S) = v_M(D) + v_M(E)$. The problem of merging two sequences, which are not necessarily balanced, is solvable by balanced merging. The approach relies on slicing two given input sequences into four, which are all sorted. Furthermore, two pairs formed out of them are guaranteed to be balanced. The slices can then be merged together as follows.

Definition 13 *The* odd-even merger *from the sequences of literals $L = \langle l_1, \ldots, l_n \rangle$ and $L' = \langle l'_1, \ldots, l'_m \rangle$ into the atoms $S = \langle s_1, \ldots, s_{n+m} \rangle$ is the program given, for $n = 0$, $m = 0$ or $n = m = 1$, by*

$$\mathsf{Merger}(L, L', S) = \begin{cases} \bigcup_{i=1}^{n} \{s_i \leftarrow l_i.\} & \textit{if } m = 0, \\ \bigcup_{i=1}^{m} \{s_i \leftarrow l'_i.\} & \textit{if } n = 0, \\ \mathsf{Cmp}(l_1, l'_1, s_1, s_2) & \textit{if } n = m = 1, \end{cases}$$

and for $n \geq 1, m \geq 1$, and $n + m \geq 3$, by

$$\begin{aligned} \mathsf{Merger}(L, L', S) = \\ & \mathsf{Merger}((L+1)/2, (L'+1)/2, D) \\ & \cup \mathsf{Merger}(L/2, L'/2, E) \\ & \cup \mathsf{BalancedMerger}(D, E, S). \end{aligned} \tag{3.9}$$

In (3.9) the symbols D and E refer to sequences of auxiliary atoms of length $\lceil |L|/2 \rceil + \lceil |L'|/2 \rceil$ and $\lfloor |L|/2 \rfloor + \lfloor |L'|/2 \rfloor$, respectively. The result S satisfies $v_M(S) = v_M(L) + v_M(L')$ under any model M of the program. The used arithmetic notation on sequences of literals is detailed in Section 2.3. Consequently, the expression $(L + 1)/2$ denotes the odd-indexed elements $\langle l_1, l_3, \ldots \rangle$, and $L/2$ the even-indexed elements $\langle l_2, l_4, \ldots \rangle$. The odd-even merger works due to the fact that, under any interpretation M of the program's atoms, the total of the values encoded in the inputs are preserved under the performed arithmetic operations, and then correctly added up by the different mergers. In the following we omit M from the $v_M(\cdot)$ notation for simplicity. For any unary literal number N, and in particular L and L', it holds that

$$\begin{aligned} v((N+1)/2) &= \lceil v(N)/2 \rceil, \\ v(N/2) &= \lfloor v(N)/2 \rfloor, \\ v(N) &= v((N+1)/2) + v(N/2). \end{aligned}$$

Therefore

$$\begin{aligned} v(D) &= v((L+1)/2) + v((L'+1)/2) \\ &= \lceil v(L)/2 \rceil + \lceil v(L')/2 \rceil , \qquad (3.10) \\ v(E) &= v(L/2) + v(L'/2) \\ &= \lfloor v(L)/2 \rfloor + \lfloor v(L')/2 \rfloor , \qquad (3.11) \end{aligned}$$

and

$$\begin{aligned} v(S) &= v(D) + v(E) \qquad (3.12) \\ &= \lceil v(L)/2 \rceil + \lceil v(L')/2 \rceil + \lfloor v(L)/2 \rfloor + \lfloor v(L')/2 \rfloor \\ &= v(L) + v(L'). \end{aligned}$$

In order for the balanced merger to function as intended, such that S is indeed sorted, we need to further establish that D and E are balanced. For any nonnegative rational number x, we have that $\lfloor x \rfloor \leq \lceil x \rceil \leq \lfloor x \rfloor + 1$. For any two nonnegative rational numbers x and y, it follows that $\lfloor x \rfloor + \lfloor y \rfloor \leq \lceil x \rceil + \lceil y \rceil \leq \lfloor x \rfloor + \lfloor y \rfloor + 2$. By comparing (3.10) and (3.11) to this, we conclude that D and E are indeed balanced. A merge-sorter is easily constructed using mergers.

Definition 14 *The* odd-even merge-sorter *from the sequence of literals L into the atoms S, with $|S| = |L|$ and with L partitioned into L_1 and L_2 such that $L_1 \neq \emptyset$ and $L_2 \neq \emptyset$ unless $|L| = 1$, is the program given, for $|L| \leq 2$, by*

$$\mathsf{Sorter}(L, S) = \mathsf{Merger}(L_1, L_2, S),$$

and for $|L| \geq 3$, by

$$\begin{aligned} &\mathsf{Sorter}(L, S) = \\ &\qquad \mathsf{Sorter}(L_1, H_1) \\ &\qquad \cup \mathsf{Sorter}(L_2, H_2) \\ &\qquad \cup \mathsf{Merger}(H_1, H_2, S). \end{aligned} \qquad (3.13)$$

In (3.13) the symbols H_1 and H_2 denote sequences of auxiliary atoms of length equal to those of L_1 and L_2, respectively. The partition of L into L_1 and L_2 can be freely selected under the stated conditions. Choosing L_1 and L_2 such that $|L_1| = \lceil |L|/2 \rceil$ and $|L_2| = \lfloor |L|/2 \rfloor$ leads to a sorter with a logarithmic number of recursion levels, and a total of $\mathcal{O}(n \times (\log n)^2)$ atoms and rules. Finally, if we denote a sorter program with P, and hide all of its auxiliary atoms, then under any hidden and output minimal model model $M \in \mathrm{MM}_{\mathrm{At_h}(P) \cup \mathrm{At}(S)}(P)$, it holds that $v_M(L) = v_M(S)$ and that the values of S are guaranteed to be in sorted order.

There are alternatives to the odd-even merger. In one of them, each output is specified without any auxiliary variables using a linear number of clauses or rules. A merge-sorter built of such primitives is called a *totalizer* [7]. The total number of used clauses is $\mathcal{O}(n^2 \times \log n)$, and the one of atoms is $\mathcal{O}(n \times \log n)$. Another alternative is that of pairwise (odd-even) mergers presented in [39], and taken to use in translating pseudo-Boolean constraints to SAT in [14], where it was also demonstrated to be highly related to the odd-even merger. Such mergers are of the same size as regular odd-even mergers on the condition that the whole output is to be produced. In practice, when parts of a merger can be left out, there are differences, however [14]. Different mergers have also been combined to produce new kinds of sorters, by using an appropriate type of merger for each input size. In [3] this approach was introduced by defining a sorter constructed to minimize a user-given function on the number of produced variables and clauses. The algorithm therein makes choices between, among others, odd-even mergers and the "direct" mergers used in totalizers.

Mergers have also been used to form selection networks [14], or cardinality networks [5]. Such a network, say a *selector*, differs from a sorter in that it only produces a "leading" subset of the output of a sorter. The said subset consists of only a specified number of the least significant outputs. Such a relaxation can be taken into account in the structure of the network. A way to accomplish this is to partially evaluate any sorter, in a similar way as in which sequential counters were pruned in Section 3.1.

3.4 Weighted Sorting

In this section, we generalize the concept of sorting in ASP to that of *weight sorting*. We assert that a weight sorter with a total body literal weight of k must be interchangeable with a set of k appropriate weight rules. More precisely, for weighted literals $l_1 = w_1, \ldots, l_n = l_k$, we let $k = \sum_{i=1}^{n} w_i$, and require that

$$\begin{aligned}\mathsf{WSorter}_{n,k}(\langle l_1 = w_1, \ldots, l_n = w_n\rangle, \langle h_1, \ldots, h_k\rangle) \equiv_{\mathrm{vs}} \\ \left\{\begin{array}{c} h_1 \leftarrow 1 \leq \langle l_1 = w_1, \ldots, l_n = w_n\rangle. \\ h_2 \leftarrow 2 \leq \langle l_1 = w_1, \ldots, l_n = w_n\rangle. \\ \vdots \\ h_k \leftarrow k \leq \langle l_1 = w_1, \ldots, l_n = w_n\rangle. \end{array}\right\}.\end{aligned} \tag{3.14}$$

Thus, as before, the result is a unary number. In this case it gives the satisfied weight of the input literals. For use in practice and in later sections, we define the following naïve weight sorter implementation

$$\begin{aligned}\mathsf{WSorter}_{n,k}(\langle l_1 = w_1, \ldots, l_n = w_n\rangle, \langle h_1, \ldots, h_k\rangle) = \\ \mathsf{Sorter}_k(\langle l_1^{w_1}, \ldots, l_n^{w_n}\rangle, \langle h_1, \ldots, h_k\rangle).\end{aligned} \tag{3.15}$$

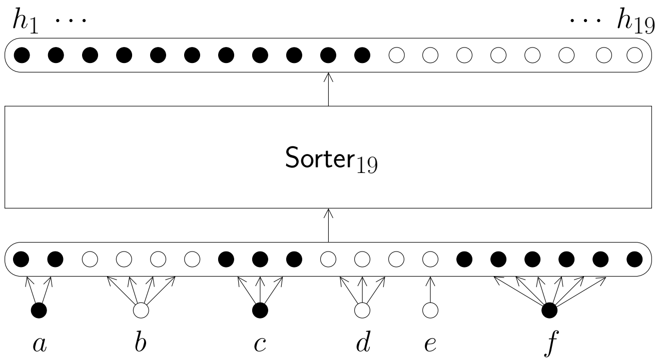


Figure 3.5: A WSorter$_{6,19}$ program implemented by wiring the input $\langle a = 2, b = 4, c = 3, d = 3, e = 1, f = 6\rangle$ to a unary sorter with output $\langle h_1, \ldots, h_{19}\rangle$. Filled markers denote atoms derived from the satisfied inputs a, c, and f.

Given any implementation of a sorter, the above gives one for a weight sorter. In general the size of such sorters is prohibitive, because the sum of weights k may be large, that is, $k \gg n$. On the other hand, the value k may be close to n, in which case a naïve implementation may prove to be preferrable to more involved encoding schemes. A circuit representation with marked truth values for an example expression is given in Figure 3.5. Our primary use for this method is, however, to use it as a *base case* for some more advanced techniques, which work by reducing the literal weights down to the point where this naïve weight sorter becomes viable. We will explore such a translation in Chapter 4.

Both of the sorting concepts dealt with here are respectively applicable to the encoding and normalization of cardinality and weight rules. A cardinality rule $h_k \leftarrow k \leq \langle l_1, \ldots, l_n\rangle$ can be substituted with the rules of Sorter$_n(\langle l_1, \ldots, l_n\rangle, \langle h_1, \ldots, h_n\rangle)$, as suggested by (3.1), when each h_i with $i \neq k$ is hidden together with any introduced auxiliary atoms. Likewise, by (3.14), a weight rule $h_i \leftarrow k \leq \langle l_1 = w_1, \ldots, l_n = w_n\rangle$ can be substituted with the program WSorter$_{n,k}(\langle l_1 = w_1, \ldots, l_n = w_n\rangle, \langle h_1, \ldots, h_k\rangle)$ when the appropriate atoms are hidden.

Chapter 4

Novel Weight Rule Normalizations

In 2006, Eén and Sörensson proposed translations from pseudo-Boolean constraints into SAT, of which one used *sorting networks* [21]. Inspired by Bailleux and Boufkhad's use of a unary encoding in translating cardinality constraints [6], Eén and Sörensson equipped a counting circuit with unary sorters for the translation of pseudo-Boolean constraints. The resulting *weight sorting network*, as we identify it here, derives a mixed-radix number encoded in propositional variables that captures the satisfied weight sum of the input expression. This number is then conveniently comparable with the bound of the constraint, and is so used to infer the satisfiability of the constraint.

In 2009, Bailleux et al. provided a refined translation, the *(global) polynomial watchdog* [7]. The modified translation is arguably simpler, and the circuit it synthesizes is *monotone*. In the context of circuits, this entails that every intermediary signal output by any of its subcircuits can only either remain or become true whenever an input signal is flipped from false to true. Equivalently, the circuit consists of only logical OR and AND gates. In practice, whereas monotone circuits tend to be larger than the most consice non-monotone circuits for the same task, they have also been found on many occasions to inhibit more favourable propagation properties when encoded for SAT solvers relying on unit propagation. In the translation, the monotonicity property is attained by offsetting both the left and the right-hand sides of the involved inequality by an additional *tare*. This is due to the fact that, with a careful choice of the tare, the final comparison part of the circuit becomes trivial and furthermore, depends only on the most significant digit of the mixed-radix number to be calculated. In the original translation of Eén and Sörensson, it is the production of the digits of lower significance that breaks the monotonicity.

In terms of asymptotical complexity the sorting networks and global polynomial watchdog are of equal size when the same sorters and mergers are used. Namely, if we let b stand for the total number of bits in the binary representations of input weights, the number of atoms and rules in a weight sorting network is $\mathcal{O}(b \times (\log b)^2)$ [21].

In the following sections we first explore how to encode mixed-radix numbers in logic programs. Then we provide the counting and comparing parts of the translation, respectively. Finally, we look into opportunities to optimize the translation, first by choosing a suitable mixed-radix base, and then by reusing identical merging constructions within and between sorters.

4.1 Mixed-Radix Numbers

In this section we briefly discuss the uses of *mixed-radices* and introduce notation for dealing with them. A *mixed-radix base* is a possibly infinite vector $B = \langle b_1, b_2, \ldots \rangle$ of positive integers. Special cases of such include the *binary* and *decimal* bases $\langle 2, 2, \ldots \rangle$ and $\langle 10, 10, \ldots \rangle$, respectively. In this work we solely deal with *finite* mixed-radix bases $B = \langle b_1, \ldots, b_k \rangle$, and refer to them as *bases* when clear from the context. The *radices* $b_1, \ldots, b_k$ of a base B are listed from the least significant b_1 to the most significant b_k, and the i^{th} radix is accessed with $B_i = b_i$. The length of a base is denoted with $|B|$.

For a nonnegative integer x in base B, we write x_i for the i^{th} least significant digit of x in B. In order to conveniently reconstruct the value of a number when given its representation in a base, we define the cumulative product of all radices preceding the i^{th} position to be the i^{th} *place value* of the base and denote it with $B_i^{\Pi} = \prod_{j=1}^{i-1} B_j$. Now the value v of a mixed-radix number x in base B can be calculated as

$$v = \sum_{i=1}^{|B|} \left(x_i \times B_i^{\Pi} \right). \tag{4.1}$$

Example 7 *For the base $B = \langle 4, 2, 5, 9 \rangle$ and number $x = 326 = 7352_B$,*

$$\begin{array}{llll}
B_1 = 4, & B_1^{\Pi} = 1, & x_1 = 2, & x_1 \times B_1^{\Pi} = 2, \\
B_2 = 2, & B_2^{\Pi} = 4, & x_2 = 5, & x_2 \times B_2^{\Pi} = 20, \\
B_3 = 5, & B_3^{\Pi} = 8, & x_3 = 3, & x_3 \times B_3^{\Pi} = 24, \\
B_4 = 9, & B_4^{\Pi} = 40, & x_4 = 7, & x_4 \times B_4^{\Pi} = 280.
\end{array}$$

Observe that $2 + 20 + 24 + 280 = 326$. ■

Traditionally it is imposed that every digit of a number x is less than the corresponding radix, that is, $x_i < B_i$. Under this constraint, there are two aspects to note. First, a number expressed in a finite base is restricted by the above to a finite domain, namely the set $\{0, \ldots, \prod_{i=1}^{|B|} B_i - 1\}$. Second, every number x expressible in B then has a unique representation in which the digits of x in

B are uniquely determined by the value of x. In this work we deal with mixed-radix numbers both under, and free from this constraint, and call them respectively unique and non-unique mixed-radix numbers.

As in Section 2.3, we aim to encode numbers using literals. Therefore we hierarchically refine the discussed mixed-radix notation by encoding the individual digits expressed using such notation as unary literals. In particular, for a base $B = \langle b_1, \ldots, b_k \rangle$, a *mixed-radix literal number* L is encoded with

$$\begin{aligned} L = \langle &\langle l_{1,1}, \ldots, l_{1,b_1-1} \rangle, \\ &\langle l_{2,1}, \ldots, l_{2,b_2-1} \rangle, \\ &\quad \vdots \\ &\langle l_{k,1}, \ldots, l_{k,b_k-1} \rangle \rangle, \end{aligned} \tag{4.2}$$

where for each $i \in \{1, \ldots, k\}$ the sequence $\langle l_{i,1}, \ldots, l_{i,b_i-1} \rangle$ stands for the i^{th} unary literal digit in the domain $\{0, \ldots, b_i - 1\}$. Thus, denoting each such digit with L_i, we can access the value of the i^{th} digit under an interpretation M with $v_M(L_i)$. Now, if we write $x = \langle v_M(L_1), \ldots, v_M(L_k) \rangle$, then the value of L under M is given by equation (4.1) and we denote it as well with $v_M(L)$. Furthermore, we refer to the set of atoms occurring in L with $\mathrm{At}(L) = \bigcup_{i=1}^{k} \mathrm{At}(L_i)$.

Example 8 *A variable in the domain $\{0, \ldots, 359\}$ can be represented in the base $B = \langle 4, 2, 5, 9 \rangle$ with the following mixed-radix literal*

$$\begin{aligned} \langle &\langle a_{1,1}, a_{1,2}, a_{1,3} \rangle, \\ &\langle a_{2,1} \rangle, \\ &\langle a_{3,1}, a_{3,2}, a_{3,3}, a_{3,4} \rangle, \\ &\langle a_{4,1}, a_{4,2}, a_{4,3}, a_{4,4}, a_{4,5}, a_{4,6}, a_{4,7}, a_{4,8} \rangle \rangle. \end{aligned}$$

A particular value for the variable, such as $153 = 3401_B$, is expressed by the interpretation $\{a_{1,1}, a_{3,1}, a_{3,2}, a_{3,3}, a_{3,4}, a_{4,1}, a_{4,2}, a_{4,3}\}$.

As a final note, we let the subscript-base notation distribute over weighted expressions, such that an expression of the form

$$(k \leq \langle l_1 = w_1, \ldots, l_n = w_n \rangle)_B \tag{4.3}$$

is recognized as a shorthand for

$$k_B \leq \langle l_1 = (w_1)_B, \ldots, l_n = (w_n)_B \rangle. \tag{4.4}$$

4.2 Digit-wise Sums

The weight sorting network translation begins by a selection of a suitable base B. For now we will not concentrate on the selection, but simply assume that one has been chosen. However, in practice, the binary base tends to work well. The next step after selecting a base is to collect the literals of the weight rule under consideration into *buckets*, of which there is at most one for each radix. Suppose the weight rule has the set of weighted literals $L = W$. Each bucket can be presented as a weighted expression. For the i^{th} digit of base B, we denote the corresponding *bucket expression* with $L = W_i$ standing for the weighted literals $l_1 = (w_1)_i, \ldots, l_n = (w_n)_i$. For illustration purposes let us first consider the binary base $B = \langle 2, \ldots \rangle$ and the weighted expression $L = W$. Now each bucket $L = W_i$ contains the weighted literal $l_j = 1$ for exactly those j for which the i^{th} bit of w_j is 1. The bucket expression formula $L = W_i$ also works for non-binary bases, in the way that a literal is "collected" into a bucket possibly many times by using non-unit weights. The precise number of such multiples is given by the appropriate digit $(w_j)_i$ for bucket i and weighted literal $l_j = w_j$. In the following we give an example of collecting weights to buckets.

Example 9 *In base $B = \langle 3, 4, 2, 6 \rangle$ the weights $W = \langle 137, 34, 27 \rangle$ are represented as 5112_B, 1031_B, and 1010_B. Their digits are gathered to buckets as follows:*

$$\begin{aligned} W_1 &= \langle 2, 1, 0 \rangle, \\ W_2 &= \langle 1, 3, 1 \rangle, \\ W_3 &= \langle 1, 0, 0 \rangle, \\ W_4 &= \langle 5, 1, 1 \rangle. \end{aligned}$$

■

The buckets are collected in order to add up the numbers on a per-position basis, or *digit-wise*. Whereas more traditionally, a set of numbers would be added up two at a time, here we first add their digits together within the buckets, in parallel. This approach, orthogonal to such tradition, allows us to leave carry digits for later consideration and lends itself to convenient, circuit-based implementations.

In our notation, a program that sorts a bucket $L = W_i$ into a sequence $H_i = \langle h_{i,1}, \ldots, h_{i,k} \rangle$ is given by $\mathsf{WSorter}_{n,k}(L = W_i, H)$, where n stands for the number of literals in L, and k for the sum of weights in W_i. The combined outcome of sorting every bucket is a mixed-radix literal that represents the sum of the numbers in base B, although not uniquely. That is, a particular satisfied weight sum may be expressed by several different interpretations of the output. This complicates efforts to compare the sum with a given bound and therefore, in the following section, issues arising from the lack of uniqueness are considered. A formula for the rules deriving the digit-wise sum is given in the following definition.

Definition 15 *The* digit-wise sorter program *from the set of weighted literals $L = W$ into the non-unique mixed-radix literal H in base B is the set of rules*

$$\mathsf{WDigitwiseSorter}_B(L = W, H) = \bigcup_{i=1}^{|B|} \mathsf{WSorter}(L = W_i, H_i). \qquad (4.5)$$

The value encoded in the digit-wise sum captures the satisfied sum of the weighted input expression. This relation is formalized in the following proposition, in the setting that all input literals are positive. In this particular case, the digit-wise sorter forms an NLP and the existence of a unique minimal model is guaranteed.

Proposition 4 *For a weighted expression $L = W$ with $\mathrm{At}_\sim(L) = \emptyset$, a non-unique mixed-radix literal H in base B, an interpretation $X \subseteq \mathrm{At}(L)$, and*

$$M = \mathrm{LM}(\mathsf{WDigitwiseSorter}_B(L = W, H) \cup \{a. \mid a \in X\}),$$

it holds that $v_X(L = W) = v_M(H)$.

Continuing from the previous example, and demonstrating the program (4.5), we present two examples.

Example 10 *The digit-wise sum of $W = \langle 137, 34, 27\rangle$ in base $B = \langle 3, 4, 2, 6\rangle$ is*

$$\begin{aligned}\langle 2 + 1 + 0&,\\ 1 + 3 + 1&,\\ 1 + 0 + 0&,\\ 5 + 1 + 1\rangle&,\end{aligned}$$

which evaluates to $\langle 3, 5, 1, 7\rangle$, and represents $7153_B = 198 = 137 + 34 + 27$. ■

Example 11 *Given the weighted expression $\langle L = W\rangle$ standing for the list of weighted literals $\langle a = 137, b = 34, c = 27\rangle$, the digit-wise sum of $L = W$ in base B is encoded in $H = \langle\langle h_{1,1}, \ldots, h_{1,3}\rangle, \langle h_{2,1}, \ldots, h_{2,5}\rangle, \langle h_{3,1}\rangle, \langle h_{4,1}, \ldots, h_{4,7}\rangle\rangle$ as defined by* $\mathsf{WDigitwiseSorter}_B(L = W, H) =$

$$\begin{aligned}&\mathsf{WSorter}_{3,3}(\langle a = 2, b = 1, c = 0\rangle, \langle h_{1,1}, \ldots, h_{1,3}\rangle) \cup\\ &\mathsf{WSorter}_{3,5}(\langle a = 1, b = 3, c = 1\rangle, \langle h_{2,1}, \ldots, h_{2,5}\rangle) \cup\\ &\mathsf{WSorter}_{3,1}(\langle a = 1, b = 0, c = 0\rangle, \langle h_{3,1}\rangle) \cup\\ &\mathsf{WSorter}_{3,7}(\langle a = 5, b = 1, c = 1\rangle, \langle h_{4,1}, \ldots, h_{4,7}\rangle)\end{aligned}$$

which can be implemented with

$$\begin{array}{ll}
\mathsf{Sorter}_3(\langle a, a, b\rangle, & \langle h_{1,1}, h_{1,2}, h_{1,3}\rangle) \cup \\
\mathsf{Sorter}_5(\langle a, b, b, b, c\rangle, & \langle h_{2,1}, h_{2,2}, h_{2,3}, h_{2,4}, h_{2,5}\rangle) \cup \\
\mathsf{Sorter}_1(\langle a\rangle, & \langle h_{3,1}\rangle) \cup \\
\mathsf{Sorter}_7(\langle a, a, a, a, a, b, c\rangle, & \langle h_{4,1}, h_{4,2}, h_{4,3}, h_{4,4}, h_{4,5}, h_{4,6}, h_{4,7}\rangle).
\end{array}$$

■

A graphical representation of a digit-wise sorting program is given in Figure 4.1. The displayed construction is used as the first step to translate, for a chosen bound k, the rule

$$\begin{aligned} a \leftarrow k \leq \langle b = 13, c = 7, d = 1, e = 11, f = 19, g = 19, h = 10, \\ \sim i = 13, \sim j = 6, \sim k = 13, \sim l = 3, \sim m = 4\rangle. \end{aligned} \tag{4.6}$$

4.3 Unique Mixed-Radix Sums

The outcome of the digit-wise sum described so far is a non-unique mixed-radix representation of the sum of all satisfied weight rule body literals. Furthermore, the digits in the representation are encoded in unary. The lack of uniqueness in the representation is unwelcome because it complicates the comparison with a bound. Therefore the next step is to transform the sum into a proper, unique representation of itself. This is conducted by merging the sorted output of each bucket, except for the first one, with *carry-digits* extracted from less significant output. The process begins by merging carries extracted from the least-significant bucket 1 with the output of bucket 2. The output of the above step is then used as the source of carries to be merged with the output of bucket 3, which is then likewise combined with the output of bucket 4 and so on. Each of these steps produces a unary literal number with significance corresponding to that of the used input bucket. There are two things to consider here: how the extraction of carries takes place, and what is left for output. The first of these points is straightforward. Computing the carries corresponds to dividing the unary input number with the significance ratio of the input and output digits. In the case of levels i and $i + 1$, the ratio is B_i. The division, as described in Section 4.1 can be implemented by picking, in this case, every ${B_i}^{\text{th}}$ digit of the unary input literal in question. The second point is implemented similarly, but with the modulo operation in place of division. This requires negation and unfortunately complicates correctness considerations. The output of the modulo operations, combined with the output of the most significant merger, make up a unique mixed-radix representation of the sum. The part of a

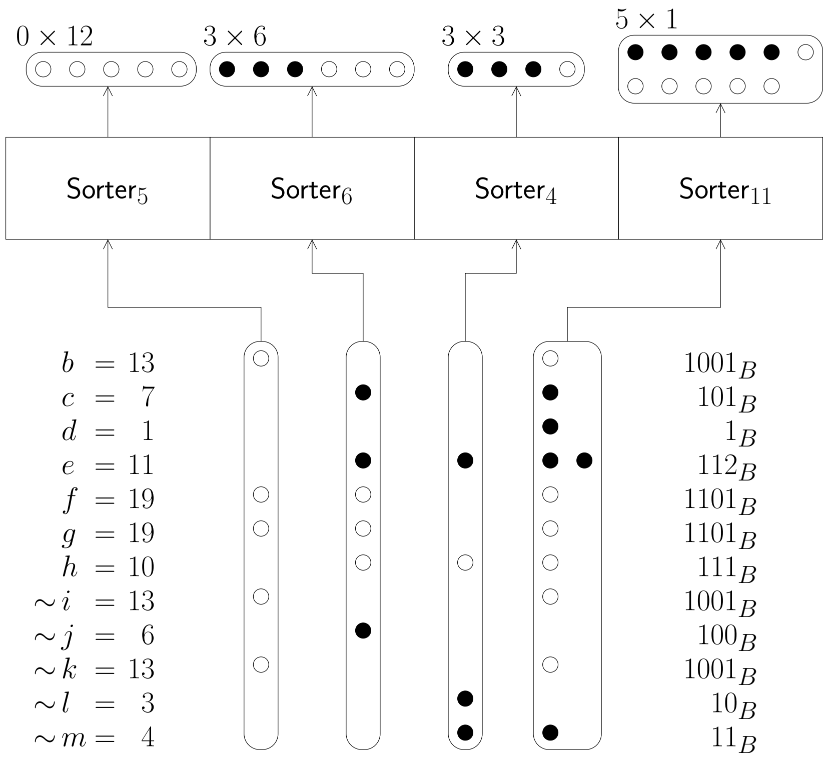


Figure 4.1: A WDigitwiseSorter$_B$ program composed of naïve unary sorters for accumulating the digit-wise sum of the weighted expression on the left in base $B = \langle 3, 2, 2, b_4 \rangle$ with $b_4 > |H_4|$. Derivations stemming from a model M of the program, which satisfies the literals $c, d, e, \sim j, \sim l$, and $\sim m$, are designated with filled markers. From right to left, the sorters count in multiples of $B_1^\Pi = 1$, $B_2^\Pi = 3$, $B_3^\Pi = 6$, and $B_4^\Pi = 12$. The mixed-radix literal H produced as the outcome encodes the sum $v_M(H) = 0\times 12 + 6\times 3 + 3\times 3 + 5\times 1 = 7+1+11+6+3+4 = 32$.

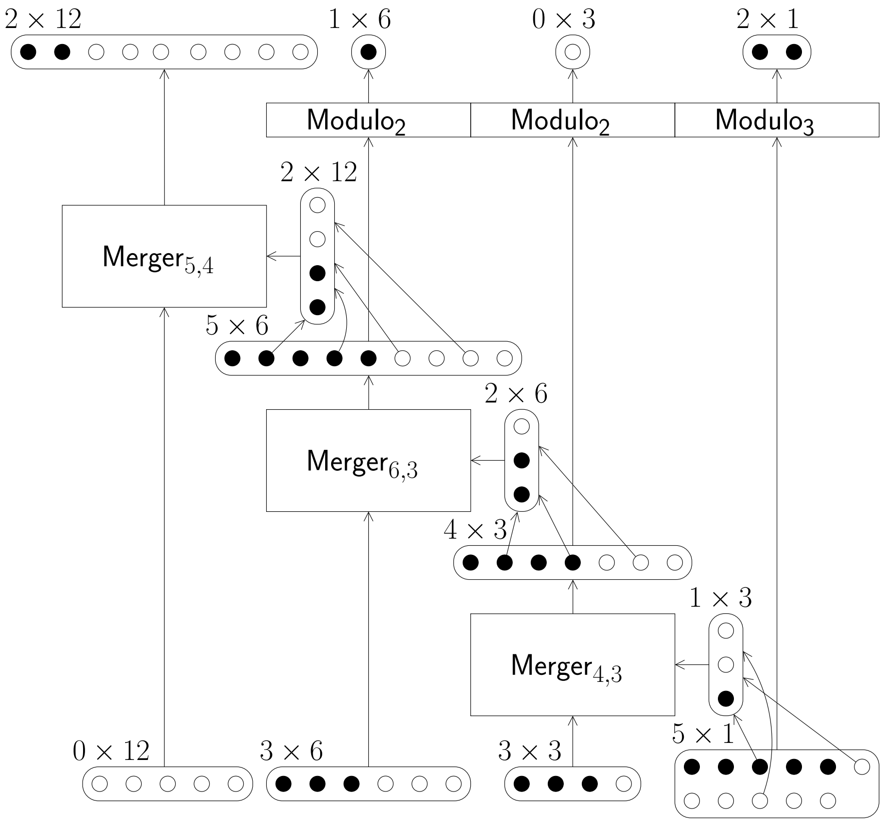


Figure 4.2: A WCarry$_B$ program for deriving a unique mixed-radix literal S from the digit-wise sum H in Figure 4.1, with the added requirement that $B_4 > |S_4|$. As part of a WCarryMerger$_B$, each merger for $1 < i \leq |B|$ produces R_i by combining H_i with carries, extracted from an intermediate literal digit R_{i-1} via the division R_{i-1}/B_{i-1}. For example, since $B_1 = 1$, every third literal of $R_1 = H_1$ is used in deriving R_2. For $1 \leq i < |B|$, the intermediary R_i is also processed into the unique output literal digit S_i by a modulo program, as part of WCarryModulo$_B$.

sorting network used to form the unique mixed-radix representation is shown in Figure 4.2, for four input buckets.

A program for extracting carries as described above is defined in the following.

Definition 16 *The* carry merger program *from the mixed-radix literal H to the mixed-radix literal R in base B, for which we set $R_1 = H_1$, is the set of rules*

$$\mathsf{WCarryMerger}_B(H, R) = \bigcup_{i=2}^{|B|} \mathsf{Merger}(R_{i-1}/B_{i-1}, H_i, R_i). \tag{4.7}$$

The carry merger program only concerns with propagating carries upwards. The most significant digit of the resulting mixed-radix literal R is unique to the satisfied sum encoded in the input H. The less significant digits of R are not deducted for the carries extracted from them. To remedy this, we define the program.

Definition 17 *The* carry modulo program *from the mixed-radix literal R with $\mathrm{At}_{\sim}(R) = \emptyset$ to the unique mixed-radix literal S in base B, for which we set $S_{|B|} = R_{|B|}$, is the set of rules*

$$\mathsf{WCarryModulo}_B(R, S) = \bigcup_{i=1}^{|B|-1} \mathsf{Modulo}_{B_i}(R_i, S_i). \tag{4.8}$$

The programs (4.7) and (4.8) can together be used to turn a non-unique mixed-radix literal into a unique one as follows.

Definition 18 *The* carry program *from the mixed-radix literal H to the unique mixed-radix literal S in base B is the set of rules*

$$\mathsf{WCarry}_B(H, S) = \mathsf{WCarryMerger}_B(H, R) \cup \mathsf{WCarryModulo}_B(R, S) \tag{4.9}$$

The mixed-radix literal S defined as the outcome of Definition 17, and thus also 18, is guaranteed to be unique in the sense described in Section 4.1. That is, every value expressible in S corresponds to a unique interpretation of $\mathrm{At}(S)$. Furthermore, the leap from the input H to the output S preserves the encoded value as determined by $v_M(\cdot)$ of (4.1).

We may now combine Definitions 15 and 18 to construct a program that forms the unique mixed-radix sum of a given weighted expression.

Definition 19 *The* counting weight sorting network program *from the weighted expression $L = W$ to the mixed-radix literal S in base B is the set of rules*

$$\begin{aligned}&\mathsf{WSortingNetwork}_B^{\mathsf{count}}(L = W, S) = \\ &\qquad \mathsf{WDigitwiseSorter}_B(L = W, H) \cup \mathsf{WCarry}_B(H, S).\end{aligned} \tag{4.10}$$

In Definition 19, the symbol H refers to an auxiliary (non-unique) mixed-radix literal that captures the digit-wise sum. Now we have defined the counting part of the translation. An example of the underlying circuit is given in Figure 4.3, for a base with 4 digits. The figure contains components responsible for all of the phases described so far from first to last in order from bottom to top.

4.4 Bound Checking via Comparing

A mixed-radix literal number as produced in the previous section can be compared with a constant. The following program checks whether a given mixed-radix literal number is greater than or equal to a specified constant.

Definition 20 *The* lexicographical comparison program *between a mixed-radix literal S in base B and a constant k, with an output atom h, is the set of rules*

$$\mathsf{WCompare}_{B,k}(S,h) = \left\{ \begin{array}{l} c_0. \\ h \leftarrow c_{|B|}. \end{array} \right\} \cup \bigcup_{i=1}^{|B|} \left\{ \begin{array}{l} S_{i,0}. \\ c_i \leftarrow S_{i,k_i+1}. \\ c_i \leftarrow S_{i,k_i}, c_{i-1}. \end{array} \right\}. \tag{4.11}$$

The symbols $c_1, \ldots, c_{|B|}$ in (4.11) denote hidden auxiliary atoms with the following behaviour. If out of the numbers encoded in the first i digits of S and k, the one extracted from S is no less than that from k, the atom c_i is derived. The last of these atoms, $c_{|B|}$, and likewise the visible output atom h, thus tells whether the value of S is greater than or equal to k. This is guaranteed by the logical consequences of the program. In a context where only positive, fixed literals are given for input, the program moreover has a unique minimal model that captures the intended output in the following way.

Proposition 5 *For a mixed-radix literal S with $\mathrm{At}_{\sim}(S) = \emptyset$ in base B, a literal h, an interpretation $X \subseteq \mathrm{At}(S)$, and*

$$M = \mathrm{LM}(\mathsf{WCompare}_{B,k}(S,h) \cup \{a. \mid a \in X\}),$$

it holds that $h \in M$ if and only if $k \leq v_X(S)$.

Joining together the parts for counting and comparing, from Definitions 19 and 20, respectively, gives the full translation program as follows.

Definition 21 *The* full weight sorting network program *of a weight rule $h \leftarrow k \leq \langle L = W \rangle$ is the set of rules*

$$\begin{aligned} &\mathsf{WSortingNetwork}^{\mathsf{full}}_{B,k}(L = W, h) = \\ &\qquad \mathsf{WSortingNetwork}^{\mathsf{count}}_{B}(L = W, S) \cup \mathsf{WCompare}_{B,k}(S,h). \end{aligned} \tag{4.12}$$

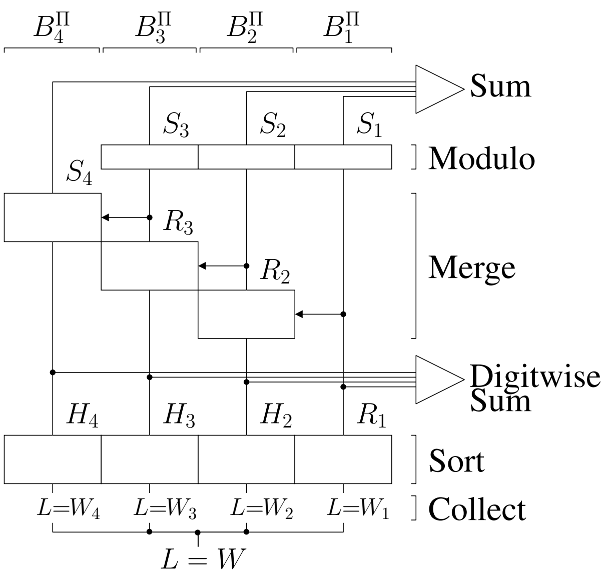


Figure 4.3: The combined counting part of a sorting network, composed of the programs in Figures 4.1 and 4.2.

Due to the use of negation in (4.12), it is not immediately clear whether visible strong equivalence between the input rule and the output program holds, even if only positive literals are given for input. This issue is, however, averted in the following section where a modified and simplified translation is considered.

4.5 Tares for Simplification

There are special cases of input parameters for which the comparison program of the previous section can be simplified away almost completely, leaving behind only a single rule. This is possibly when all but the most significant digit of the bound k is non-zero. The reasoning is that, in such a case, all but one of the auxiliary atoms in the lexicographical comparison program (4.11) are true under any model. Consequently, the program is indifferent to all but the most significant digit of the mixed-radix input literal. Most importantly, this nullifies the need for computing residues when a sorting network is constructed solely for comparing with k. In other words, no WCarryModulo and therefore no Modulo program needs to be included. Moreover, any weight rule is transformable to a form which satisfies these premises. A translation resulting from these simplifications has been given in [7] in the context of pseudo-Boolean constraints and SAT. In this section, we identify the required special case of weight rules, the simplifications applicable for them, and a transformation into them.

In the special case, in base B we prefer a bound k such that $k'_i = 0$ for all $i \in \{1, \dots, |B|-1\}$. In other words, we seek a bound represented in B in the form $k = \langle 0, \dots, 0, m\rangle = B^{\Pi}_{|B|} \times m$, where m is a nonnegative integer. That is, k must be divisible by $B^{\Pi}_{|B|}$. With such a bound, the auxiliary atoms $c_1, \dots, c_{|B|-1}$ in the lexicographical comparison program (4.11) can be turned into facts. The remaining auxiliary atom $c_{|B|}$, can be defined with $c_{|B|} \leftarrow S_{|B|,k_{|B|}}$. Additionally, the output h can be defined directly with $h \leftarrow S_{|B|,k_{|B|}}$. Consequently, only the most significant digit $S_{|B|}$ of the mixed-radix input literal S is required for the comparison. Thus, the weight sorting network program (4.10), when constructed for such a comparison, can be simplified by substituting the contained carry program (4.9) with only its first component, the carry merger program (4.7). That is, the other component, the carry modulo program (4.8), can be dropped.

Regarding the criteria of the special case, we can transform any given weight rule $a \leftarrow k \leq \langle L = W\rangle$ into a form satisfying them. Namely, into $a \leftarrow k + t \leq \langle L = W, \top = t\rangle$, where $\top$ is a fact and t is an appropriately chosen nonnegative integer. We call t a *tare* and choose it to be the least nonnegative integer causing $k + t$ to be divisible by $B^{\Pi}_{|B|}$. That is, $t = (B^{\Pi}_{|B|} \times \lceil k/B^{\Pi}_{|B|}\rceil) - k$. In the transformation, both sides of the involved inequality are adjusted by an equal amount, and thus the adjustment is sound. The changes described in this section are summarized in the following definition.

Definition 22 *The* simplified weight sorting network program *of the weighted expression $L = W$ in base B for testing the bound k into h using the tare t is the program*

$$\begin{aligned}
&\mathsf{WSortingNetwork}^{\mathsf{tare}}_{B,k,t}(L = W, h) = \\
&\quad \mathsf{WDigitwiseSorter}_B(\langle L = W, \top = t\rangle, H) \\
&\quad \cup\, \mathsf{WCarryMerger}_B(H, S) \\
&\quad \cup\, \{h \leftarrow S_{|B|,(k+t)_{|B|}}.\}.
\end{aligned} \tag{4.13}$$

If L consists of only atoms and no negative literals, then the rules of (4.13) form a positive NLP, in contrast to the “full” translation scheme (4.12), in which negation is in use. This makes it easier to prove the correctness of the translation, even in the case that L contains negative literals as well.

Proposition 6 *For a base B, a weight rule $h \leftarrow k \leq \langle L = W\rangle$ with $\mathrm{At}_{\sim}(L) = \emptyset$, an interpretation $X \subseteq \mathrm{At}(L)$, and*

$$M = \mathrm{LM}(\mathsf{WSortingNetwork}^{\mathsf{tare}}_{B,k}(L = W, h) \cup \{a. \mid a \in X\}),$$

it holds that $h \in M$ if and only if $k \leq v_X(L = W)$.

Theorem 1 *For a base B, a weight rule $h \leftarrow k \leq \langle L = W \rangle$, and a tare t, it holds that* $\mathsf{WSortingNetwork}^{\mathsf{tare}}_{B,k,t}(L = W, h) \equiv_{\mathrm{vs}} \{h \leftarrow k \leq \langle L = W \rangle .\}$.

Proof Let $P = \{h \leftarrow k \leq \langle L = W \rangle .\}$ and $Q = \mathsf{WSortingNetwork}^{\mathsf{tare}}_{B,k,t}(L = W, h)$ with $A = \mathrm{At_v}(P) = \mathrm{At_v}(Q) = \{h\} \cup \mathrm{At}(L)$. First assume that $\mathrm{At}_{\sim}(L) = \emptyset$. Now P and Q are positive and one needs to show that $\mathrm{MM}_{\mathrm{At_h}(P)}(P) =_{\mathrm{v}} \mathrm{MM}_{\mathrm{At_h}(Q)}(Q)$. The program P has no hidden atoms, that is, $\mathrm{At_h}(P) = \emptyset$, and thus the classical models of P give $\mathrm{MM}_{\mathrm{At_h}(P)}(P) =$

$$\begin{array}{ll} \{X \subseteq \mathrm{At}(P) & \mid X \models h \text{ if } X \models k \leq \langle L = W \rangle \ \} = \\ \{X \cup \{h \mid k \leq v_X(L = W)\} & \mid X \subseteq \mathrm{At}(L)\}. \end{array} \tag{4.14}$$

For every interpretation X over the input literals $\mathrm{At}(L)$, by Proposition 6 there is a minimal model M of Q agreeing with X that is unique to X. In notation, $M = \mathrm{LM}(Q \cup \{a. \mid a \in X\})$. Furthermore, $h \in M$ if and only if $k \leq v_X(L = W)$. Consequently, the hidden minimal models of Q can be written as $\mathrm{MM}_{\mathrm{At_h}(Q)}(Q) =$

$$\{M, M \cup \{h\} \quad \mid X \subseteq \mathrm{At}(L), M = \mathrm{LM}(Q \cup \{a. \mid a \in X\})\},$$

and their projections onto the visible signature A as

$$\begin{array}{ll} \{M, M \cup \{h\} & \mid X \subseteq \mathrm{At}(L), M = \mathrm{LM}(Q \cup \{a. \mid a \in X\}) \cap A\} = \\ \{M, M \cup \{h\} & \mid X \subseteq \mathrm{At}(L), M = X \cup \{h \mid k \leq v_X(L = W)\}\}. \end{array} \tag{4.15}$$

The sets (4.14) and (4.15) are equal. Now define a bijection f with $f(X) = \mathrm{LM}(Q \cup \{a. \mid a \in X\})$ and $f^{-1}(X) = X \cap A$. The hidden minimal models of P and Q are visibly equal via f, that is, $\mathrm{MM}_{\mathrm{At_h}(P)}(P) =_{\mathrm{v}} \mathrm{MM}_{\mathrm{At_h}(Q)}(Q)$. By Proposition 2, it follows that $P \equiv_{\mathrm{vs}} Q$.

Now let $N = \mathrm{At}_{\sim}(L)$ and assume that $N \neq \emptyset$. All atoms occurring negatively in either P or Q are in L and consequently $N = \mathrm{At}_{\sim}(P) = \mathrm{At}_{\sim}(Q)$. Now construct the set of substitution pairs $S = \{b_a \leftarrow {\sim}a. \mid a \in N\}$ and apply it to obtain $P[S]$ and $Q[S]$. Since $N \subseteq A$, by Proposition 3, it is sufficient to show that $\mathrm{MM}_{\mathrm{At_h}(P)}(P[S]) =_{\mathrm{v}} \mathrm{MM}_{\mathrm{At_h}(Q)}(Q[S])$. The programs $P[S]$ and $Q[S]$ are positive, and thus $=_{\mathrm{v}}$ follows as shown for any positive programs. In conclusion, $P \equiv_{\mathrm{vs}} Q$. □

This theorem entails that all weight rules in any program can be substituted with their translations defined in (4.13), and the answer sets of the combined translated program are guaranteed to visibly match those of the original.

4.6 Base Heuristic

The sorting network translation is parametrized by a mixed-radix base, and in this section we consider the selection of that base. The translation works with any base capable of expressing numbers up to the sum of body weights and a tare. The choice between different bases affects the magnitude and number of digits required to represent the intermediary and final outputs of the normalization. The larger the digits, the larger sorters and mergers are produced. Thus an optimization problem is presented, and finding a base that leads to small translation sizes is of interest. Previously, this problem has been approached by using a binary base [7], or by exhaustively enumerating primes less than 20 for use as radices [21]. Moreover, in [16] this *optimal base problem*, as it is formalized therein, is solved with exhaustive search algorithms capable of optimizing given measures of the base.

In contrast to previous work, we use a greedy heuristic in constructing the mixed-radix base. The radices are chosen one by one from the least to the most significant based on estimates of the consequential translation size. In the following it is assumed, for simplicity, that $\max\{w_1, \ldots, w_n\} \leq k \leq \sum_{j=1}^{n} w_j$. Moreover, the order of the size of sorters and mergers used as primitives is denoted by $\mathrm{z}(n) = n \times (\log n)^2$. When selecting the i^{th} radix b_i, we form an intermediary base $B = \langle b_1, \ldots, b_{i-1}, \infty \rangle$, and calculate $s = \sum_{j=1}^{n} ((w_j)_i \bmod b)$. Then, in terms of $W_i = (w_1)_i, \ldots, (w_n)_i$, we pick

$$b_i = \underset{\substack{b \text{ is prime, } b \leq \max \\ \{2, (w_1)_i, \ldots, (w_n)_i\}}}{\arg\max} \left(\begin{array}{l} \mathrm{z}(s) + \mathrm{z}\left(n/2 + \min\left\{\lceil s/b \rceil, \lfloor k_i/b \rfloor\right\} + 1\right) \\ \qquad + \mathrm{z}\left(3/4 \times n\right) \times \log_2(1/(2 \times n \times b) \times \sum_{j=1}^{n} (w_j)_i) \end{array} \right).$$

The idea of the three added terms is to generously estimate the size of the primitives entailed by the choice of a prime b. The first reflects the size of the sorter, and the corresponding merger, of the i^{th} bucket expression. Similarly the second addend corresponds to the immediately following components, and the third to the entire remaining structure. Radices are picked until $\prod_{j=1}^{i} b_j > \max\{w_1, \ldots, w_n\}$. At this point, any tare is assigned either to $t = 0$, if none is used, or to $t = (\lceil k / \prod_{j=1}^{i-1} b_j \rceil \times \prod_{j=1}^{i-1} b_j) - k$. To be able to express the sum of all body weights, combined with the tare, the most recently picked, and most significant radix is adjusted to $b_i = \lfloor (\sum_{j=1}^{n} w_j + t) / \prod_{j=1}^{i-1} b_j \rfloor + 1$. Then the selection finishes and returns the base $B = \langle b_1, \ldots, b_i \rangle$. In Chapter 6, we compare the effect of heuristically chosen mixed-radix bases with binary bases having $b_j = 2$ for $1 \leq j < i$.

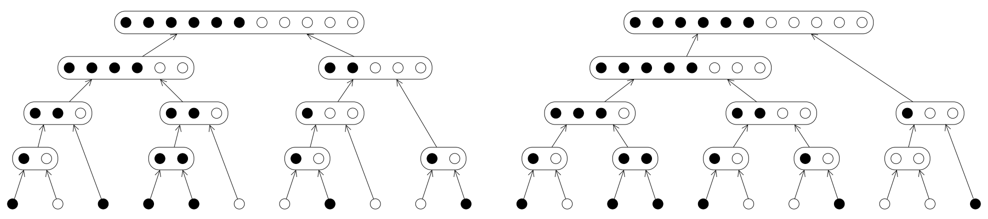


Figure 4.4: Two merge-sorter structures for the same sorting task.

4.7 Structure Sharing

The digit-wise sorter in (4.5) consists of a number of sorters. When implemented via merge-sorting, discussed in Section 3.3, the sorters are recursively composed of mergers. Each sorter is responsible for ordering the contents of a bucket expression into a sorted sequence, and each merger for ordering a subsequence. When sorters are taken for trees of mergers, the combined construction forms a *forest of mergers* containing the input literals as a common set of leaves. Such a graph representation of a weight sorting network for the rule (4.6) with the bound $k = 24$ is given in Figure 4.5. In this section, a novel *structure sharing* algorithm is introduced to exploit this view into the structure of the digit-wise sorter, and to compress the forest, and thereby to compress the corresponding normal logic program as well. Compactness is gained by reshaping the forest and eliminating duplicated subtrees. The role of reshaping is to expand opportunities for duplicate elimination. The latter is for instance achievable by structural hashing [21].

To illustrate the idea of the algorithm, we begin by considering an abstract setting. To this end, observe that merging unary numbers, or simply adding numbers up, is a *commutative* and *associative* operation. Let us denote any operator with these properties by $\oplus$. For operands a, b, and c, due to commutativity, the expression $a \oplus b$ equals $b \oplus a$, and due to associativity, the expression $(a \oplus b) \oplus c$ equals $a \oplus (b \oplus c)$. These identities for reordering expressions can be used analogously in reshaping merge-sorter structures as shown in Figure 4.4. Furthermore, reordering generally helps to reveal common subexpressions. With the help of auxiliary variables, or auxiliary atoms in case of logic programs, such common structure can be captured and reused. In the following, we demonstrate this idea behind the algorithm in the abstract setting with $\oplus$.

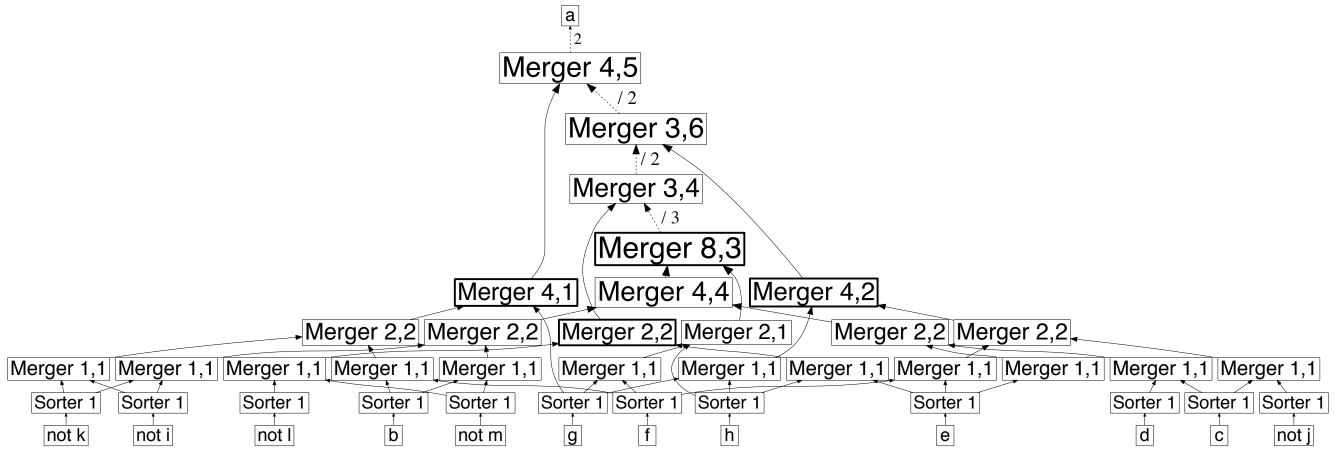


Figure 4.5: A regular weight sorting network built from mergers. The mergers denoted with thick outlines form a digit-wise sorter together with the mergers below them. Those above them belong to a carry program. The division markings give the radices of the base $B = \langle 3, 2, 2, b_4 \rangle$, which are used to derive carry digits by division. The number 2 at the top indicates that the second output digit of the top merger gives the final result.

Algorithm 1 Plan Structure Sharing

```
1: function PLAN($L = W, B$)
2:     let $C := \{[l_1^{(w_1)_i}, \dots, l_n^{(w_n)_i}] \mid 1 \le i \le |B|\}$
3:     while $\exists S \in C : \exists x, y \in S : x \ne y$
4:         let $(x, y) := \arg\max_{x,y \in \biguplus C} \sum_{S \in C} \begin{cases} \#_S(x) \times \#_S(y) & \text{if } x \ne y \\ (\#_S(x) \times (\#_S(x) - 1))/2 & \text{if } x = y \end{cases}$
5:         let $z := [x, y]$
6:         for each $S \in C$
7:             let $j := \min\{\#_S(x), \#_S(y)\}$
8:             set $S := \begin{cases} (S \setminus [x^j, y^j]) \uplus [z^j] & \text{if } x \ne y \\ (S \setminus [x^{2\lfloor j/2 \rfloor}]) \uplus [z^{\lfloor j/2 \rfloor}] & \text{if } x = y \end{cases}$
9:     return $C$
```

Example 12 *Consider some x, y, z, produced from operands a, b, c, d, e, f using*

$$x = ((a \oplus b) \oplus c) \oplus e, \qquad y = (b \oplus c) \oplus d, \qquad z = ((a \oplus c) \oplus e) \oplus f.$$

By commutativity and associativity, the right-hand sides can be reordered as

$$x = (a \oplus e) \oplus (b \oplus c), \qquad y = (b \oplus c) \oplus d, \qquad z = ((a \oplus e) \oplus c) \oplus f.$$

Due to the reordering, we may capture two common subexpressions in auxiliary variables $u = a \oplus e$ and $v = b \oplus c$, and reduce the number of required applications of $\oplus$ in total from eight to six by further writing

$$x = u \oplus v, \qquad y = v \oplus d, \qquad z = (u \oplus c) \oplus f.$$

■

To carry out structure sharing, we propose Algorithm 1. Thereby, we denote a *multiset* S on a set $X = \{x_1, \dots, x_n\}$ of ground elements with respective *multiplicities* $i_1, \dots, i_n$ by $[x_1^{i_1}, \dots, x_n^{i_n}]$. The superscript i_j can be omitted from $x_j^{i_j}$ if $i_j = 1$. The multiplicity i_j of $x_j \in X$ is referred to by $\#_S(x_j)$, and x_j is said to have i_j occurrences in S. For any $x \notin X$, $\#_S(x) = 0$. Furthermore, we write $x \in S$ iff $x \in X$ and $\#_S(x) > 0$. We denote the *multiset sum* of two multisets S_1 and S_2 by $S_1 \uplus S_2$, defined such that $x \in S_1 \uplus S_2$ iff $x \in S_1$ or $x \in S_2$, and that for $x \in S_1 \uplus S_2$, $\#_{S_1 \uplus S_2}(x) = \#_{S_1}(x) + \#_{S_2}(x)$. At the beginning of the algorithm, the bucket expressions $L = W_i$ are gathered into a collection C of multisets, where the literals $L = \langle l_1, \dots, l_n \rangle$ form the common ground elements and the digits $W_i = \langle (w_1)_i, \dots, (w_n)_i \rangle$ give the multiplicities for $1 \le i \le |B|$. Then,

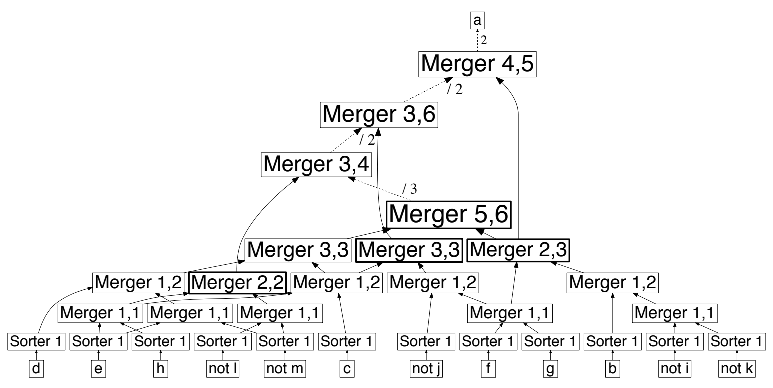


Figure 4.6: A structure-shared weight sorting network.

iteratively, pairs (x, y) of elements with heuristically maximal joint occurrences in C are selected to form new multisets z replacing common occurrences of (x, y) in each $S \in C$. The introduced multisets z are in the sequel handled like regular ground elements, and the algorithm proceeds until every $S \in C$ consists of a single multiset. The resulting collection C generally comprises nested multisets, which we interpret as a directed acyclic graph, intuitively consisting of a number of overlaid trees with the literals $l_1, \dots, l_n$ as leaves, multisets z as roots, and inner nodes giving rise to mergers. A structure-shared version of the construction shown in Figure 4.5 is displayed in Figure 4.6.

Example 13 *Considering the weighted literals $a = 9, b = 3, c = 7, d = 2, e = 5, f = 4$ and the base $B = 2, 2, 9$, Algorithm 1 yields the following merge-sorter structure:*

$$\begin{aligned}
C &:= \{[a, b, c, e], [b, c, d], [a^2, c, e, f]\}, \\
C &:= \{[[a, e], b, c], [b, c, d], [a, [a, e], c, f]\}, \\
C &:= \{[[a, e], [b, c]], [[b, c], d], [a, [a, e], c, f]\}, \\
&\vdots \\
C &:= \{[[[a, e], [b, c]]], [[[b, c], d]], [[[a, [a, e]], [c, f]]]\}.
\end{aligned}$$ ■

Intuitively, Algorithm 1 begins with a specification C of which multisets to sort and terminates with a modified specification C that gives the structure of shared merge-sorters for doing so. In order to ensure that the initial multisets are reflected correctly in the final result, we define *canonical multisets* $\mathrm{c}(\cdot)$ associated with occurrences of tree nodes. For an m-fold occurrence l^m of a literal l, we let $\mathrm{c}(l^m) = [l^m]$. For an m-fold occurrence S^m of a multiset $S = [x_1^{i_1}, \dots, x_n^{i_n}]$, we let $\mathrm{c}(S^m) = \biguplus_{j=1}^{n} \mathrm{c}(x_j^{m \times i_j})$. The algorithm can now be shown to maintain the following invariant: for every root multiset $S \in C$, the canonical subset $\mathrm{c}(S)$ is fixed.

Intuitively, the invariant states that the leaves of the trees stay unaltered. To see this, consider two literals or multisets x and y, and their combination $z = [x, y]$, as in the algorithm. Furthermore, take any $S \in C$, and let $j = \min\{\#_S(x), \#_S(y)\}$. Assuming $x \neq y$, it needs to be shown that

$$\mathrm{c}(S) = \mathrm{c}((S \setminus [x^j, y^j]) \uplus [z^j]).$$

Due to the conservative choice of j, the above follows if

$$\mathrm{c}([x^j, y^j]) = \mathrm{c}([z^j]).$$

This holds since

$$\mathrm{c}([z^j]) = \mathrm{c}(z^j) = \mathrm{c}([x, y]^j) = \mathrm{c}(x^j) \cup \mathrm{c}(y^j) = \mathrm{c}([x^j, y^j]).$$

The case for $x = y$ holds as well because then

$$\mathrm{c}(z^{\lfloor j/2 \rfloor}) = \mathrm{c}([x^2]^{\lfloor j/2 \rfloor}) = \mathrm{c}(x^{2\lfloor j/2 \rfloor}).$$

The canonical multisets of tree nodes consequently capture and preserve the bucket expressions, as multisets, throughout the execution of Algorithm 1. Furthermore, each iteration replaces at least one common occurrence of a pair of elements with a multiset, which then forms an internal node. Given finite input, the algorithm thus always terminates. If the internal nodes are viewed, not only as multisets, but as blueprints of mergers, the resulting collection of merge-sorters will not only preserve the canonical multisets, but also sort the corresponding bucket expressions. In summary, the outcome can be interpreted as a structurally shared digit-wise sorter.

Algorithm 1 can be modified to initialize the collection C differently, in order to decrease the runtime of the algorithm without significantly affecting the quality of the results. We call the modified structures *head-started* digit-wise sorters and weight sorting networks. Let $G = \{l_1, \dots, l_n\}$. For the modification, we first make use of an equivalence relation, say R, defined for literals $l_i, l_j \in G$ such that $l_i \; R \; l_j$ iff $w_i = w_j$. We denote the partition of G into equivalence classes by

$$\mathcal{F} = \{\{l' \in G \mid l \; R \; l'\} \mid l \in G\}.$$

Intuitively, $\mathcal{F}$ consists of subsets of the initial ground elements of C, which can be used to reconstruct C, and which are in a sense maximal. The modification is completed by replacing C on line 2 of Algorithm 1, denoted here by $C = \{S_1, \dots, S_{|B|}\}$, with any collection $C' = \{S'_1, \dots, S'_{|B|}\}$ of multisets having $\mathcal{F}$ for ground elements such that $\mathrm{c}(S'_i) = \mathrm{c}(S_i)$ for $1 \leq i \leq |B|$. This can be achieved by repeatedly performing, for any $S \in C$ and subset $A \subseteq S$ such that $A \in \mathcal{F}$, the

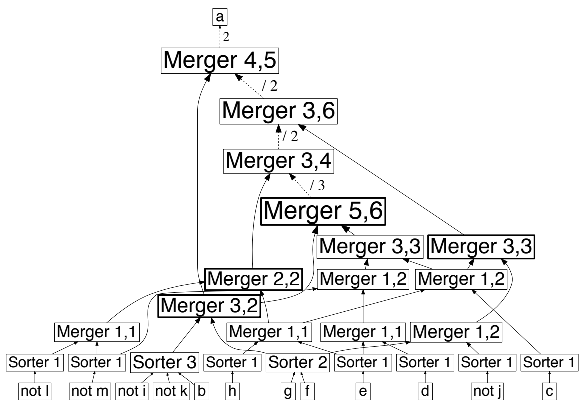


Figure 4.7: A head-started, structure-shared weight sorting network.

step $S := (S \setminus A) \uplus [A]$, until no such A exists. The intention is to add a sorter for each equivalence class $A \in \mathcal{F}$ to the translation. Moreover, each substitution step above indicates a reference to the output of the sorter of a class A. The effect of head start to the weight sorting network in Figure 4.6 is displayed in Figure 4.7. The initialization steps used to obtain the sorters in the latter figure are illustrated in the following.

Example 14 *Given the rule* (4.6)*, the corresponding partition to equivalence classes is*

$$\mathcal{F} = \{\{b, {\sim}i, {\sim}k\}, \{c\}, \{d\}, \{e\}, \{f, g\}, \{h\}, \{{\sim}j\}, \{{\sim}l\}, \{{\sim}m\}\}$$

and therefore, by performing head start, the initial collection

$$\left\{ \begin{array}{l} [b, f, g, {\sim}i, {\sim}k], [c, e, f, g, h, {\sim}j], \\ [e, h, {\sim}l, {\sim}m], [b, c, d, e^2, f, g, h, {\sim}i, {\sim}k, {\sim}m] \end{array} \right\}$$

is replaced with

$$\left\{ \begin{array}{l} [[b, {\sim}i, {\sim}k], [f, g]], [c, e, [f, g], h, {\sim}j], \\ [e, h, {\sim}l, {\sim}m], [[b, {\sim}i, {\sim}k], c, d, e^2, [f, g], h, {\sim}m] \end{array} \right\}.$$

■

Chapter 5

Weight Rule Simplification

In this chapter we aim to simplify weight rules before or even without translating them. Simplification opportunities provide for smaller weight rules to translate, which in turn lead to potentially more compact normalizations. In Section 5.1 we consider possibilities for dropping body literals from a rule as well as substituting several literals. Aside from the simplest cases, the used techniques require the addition of auxiliary atoms and rules and careful checking for the correctness of the performed steps. In Section 5.2 we analyze the quotients and residues of weights obtained when dividing them with a selected divisor. Such an operation reveals a number of opportunities for reducing the magnitudes of body weights. This operation can be useful especially in combination with translation techniques that depend on these magnitudes, such as those developed in Chapter 4.

5.1 Removal of Literals

In this section, we describe simplification techniques that aim to reduce the number of body literals in a given weight rule. The first of these techniques is designed to simply drop literals from a weight rule without any compensation steps. The approach proceeds by checking literals one by one in order from literals with small weights to literals with large weights. Let us assume we have a weighted expression $\langle L = W, l = w \rangle$ where $L = W$ stands for $l_1 = w_1, \ldots, l_n = w_n$, in which $w \leq w_i$ for $1 \leq i \leq n$, and which appears in a rule of the form

$$a \leftarrow k \leq \langle L = W, l = w \rangle. \tag{5.1}$$

The goal is now to identify whether the weighted literal $l = w$ can be dropped from the rule (5.1) without any effect. This is done as follows. If there is no subset $X \subseteq \mathrm{At}(L)$, such that $v_X(L = W) < k \leq v_X(L = W) + w$, then (5.1) can be safely substituted with (2.1). Under this condition, the weighted literal $l = w$

never holds a pivotal role in surpassing the bound. In particular, if

$$w < k - \max\Big\{\sum_{i\in I} w_i \;\Big|\; I \subseteq \{1,\ldots,n\}, \sum_{i\in I} w_i < k\Big\},$$

then the substitution can proceed.

Example 15 *The rule*

$$a \leftarrow 50 \leq \langle b_1 = 30, b_2 = 20, b_3 = 15, {\sim}c_1 = 10, {\sim}c_2 = 3\rangle.$$

can be substituted with

$$a \leftarrow 50 \leq \langle b_1 = 30, b_2 = 20, b_3 = 15, {\sim}c_1 = 10\rangle.$$

Indeed, the subset sum of $30, 20, 15,$ *and* 10 *that is greatest among those less than* 50 *is* 45*. It holds that* $45 + 3 < 50$*, and thus* ${\sim}c_2 = 3$ *can be excluded.*

A second technique, described in the following, similarly relies on a special case. Suppose we again have a rule of the form (5.1), with the exception that now $k \leq w$. Such a large weight w dominates the bound k in the sense that it is sufficient alone in satisfying the body. In this case, the weighted literal $l = w$ can be dropped, and its effect compensated by an added normal rule. Specifically, if $k \leq w$, then (5.1) can be substituted with

$$\begin{aligned} &a \leftarrow k \leq \langle L = W\rangle. \\ &a \leftarrow l. \end{aligned} \tag{5.2}$$

The above simplification step generalizes to pairs of literals as follows. Suppose we have a pair of weighted literals $l = w$ and $l' = w'$, such that $w \leq w'$, and a rule of the form

$$a \leftarrow k \leq \langle L = W, l = w, l' = w'\rangle. \tag{5.3}$$

Assume further that $k \leq w + w'$. In preparation for the actual simplification step, observe that, under the stated assumptions, the rule (5.3) can be substituted with

$$\{a \leftarrow k \leq \langle L = W, q = w, l' = w' - w\rangle.\} \cup \mathsf{Cmp}(l, l', q, a). \tag{5.4}$$

In (5.4), an auxiliary atom q is introduced to capture the disjunction of l and l'. The program Cmp is given as Section 3.3. The transformation from (5.3) to (5.4) leads to a weight rule with a lowered weight for a single body literal. This presents an opportunity to apply the previously described literal removal simplification, now targeted at $l' = w' - w$. The combination leads to the following new simplification method. Assume still that $w \leq w'$ and $k \leq w + w'$. Now if

$$w' - w < k - \max\Big\{\sum_{i\in I} w_i + v \;\Big|\; I \subseteq \{1,\ldots,n\}, v \in \{0, w\}, \sum_{i\in I} w_i + v < k\Big\},$$

then (5.3) can be substituted with

$$\{a \leftarrow k \leq \langle L = W, q = w\rangle.\} \cup \mathsf{Cmp}(l, l', q, a). \tag{5.5}$$

When applicable, this simplification technique thus reduces the number of body literals in a weight rule at the cost of introducing normal rules. If applied repeatedly, the number of added rules is linear in the number of removed body literals. In light of the superlinearity of the weight rule translations presented so far, in Chapters 3 and 4, the simplification technique can be regarded favourable.

Example 16 *Consider the rule*

$$a \leftarrow 50 \leq \langle b_1 = 30, b_2 = 20, b_3 = 15, {\sim}c_1 = 10, {\sim}c_2 = 33\rangle. \tag{5.6}$$

By letting $l = b_1$ *and* $l' = {\sim}c_2$*, it can be substituted with the program*

$$\begin{aligned}
&a \leftarrow 50 \leq \langle q = 30, b_2 = 20, b_3 = 15, {\sim}c_1 = 10\rangle.\\
&a \leftarrow b_1, {\sim}c_2.\\
&q \leftarrow b_1.\\
&q \leftarrow {\sim}c_2.
\end{aligned}$$

The justification follows from Example 15, given that $33 - 30 = 3$.

A final technique presented in this section is designed to extract a *normal part* from a weight rule. Suppose we have a weight rule of the form (2.1) with $\langle L = W\rangle$ defined as before with the addition that $w_1 \geq w_2 \geq \cdots \geq w_n$. Now, let us split $\langle L = W\rangle$ in two parts after the greatest index p, for which

$$\sum_{\substack{1\leq i\leq n\\ i\neq p}} w_i < k.$$

The induced prefix $l_1, \ldots, l_p$ forms what we call the normal part of the rule, constituting all literals invariably required for the body to be satisfied. If even one of them is unsatisfied, the remaining literals and weights are insufficient to meet the bound. In the case that $0 < p$, the normal part can be extracted from the weight rule (2.1) by creating a substitute program, using a new bound $k' = k - \sum_{i=1}^{p} w_i$, as follows.

$$\begin{aligned}
&q \leftarrow k' \leq \langle l_{i+1} = w_{i+1}, \ldots, l_n = w_n\rangle.\\
&a \leftarrow l_1, \ldots, l_i, q.
\end{aligned} \tag{5.7}$$

In (5.7), the symbol q denotes a new auxiliary atom. In the corner case that $p = n$, the outcome can be written plainly as

$$a \leftarrow l_1, \ldots, l_n. \tag{5.8}$$

Example 17 *Consider the simplified weight rule from Example 16*

$$a \leftarrow 50 \leq \langle q = 30, b_2 = 20, b_3 = 15, {\sim}c_1 = 10\rangle.$$

The atom q *forms a normal part since* $20 + 15 + 10 < 50$*, and the rule can be simplified into*

$$\begin{aligned}&q' \leftarrow 20 \leq \langle b_2 = 20, b_3 = 15, {\sim}c_1 = 10\rangle.\\&a \leftarrow q, q'.\end{aligned}$$

By combining Examples 16 and 17, and applying further simplifications recursively, the rule (5.6) fully translates into the program

$$\begin{array}{ll} a \leftarrow b_1, {\sim}c_2. & a \leftarrow q, q'. \\ q \leftarrow b_1. & q' \leftarrow b_2. \\ q \leftarrow {\sim}c_2. & q' \leftarrow b_3, {\sim}c_1. \end{array}$$

The rules on the left are from Example 16. Out of the rules on the right, the first is from Example 17, the second is derived as (5.2) and the third as (5.8).

5.2 Residue Analyzis

In this section the goal is to reduce the weights of literals in the body of a weight rule, instead of the number of such literals. The motivation for reducing weights stems from the fact that lower weights provide for more concise translations, for example, when using weight sorting networks of Chapter 4.

Consider a weight rule (2.1) containing a weighted expression $\langle L = W\rangle = \langle l_1 = w_1, \ldots, l_n = w_n\rangle$ and a bound $k > 2$. Let us suppose that we have chosen a positive integer $d < k$. The choice of d determines weight quotients Q and residues R as follows

$$\begin{aligned}Q &= \langle q_1, \ldots, q_n\rangle = \langle \lfloor w_1/d \rfloor, \ldots, \lfloor w_n/d \rfloor\rangle,\\R &= \langle r_1, \ldots, r_n\rangle = \langle w_1 \bmod d, \ldots, w_n \bmod d\rangle.\end{aligned}$$

Assuming that $\sum R < d$, the above decomposition reveals four conditions on which the rule can be simplified.

Case 1 If $\lfloor k/d \rfloor \notin \{v_I(L = Q) \mid I \subseteq \mathrm{At}(L)\}$, then substitute (2.1) with $a \leftarrow \lfloor k/d \rfloor \leq \langle L = Q\rangle$. The idea is that if the sum of the residues of the weights is small enough, then the residues matter only if the quotients can result in a “tie”.

Example 18 *Consider the rule*

$$a \leftarrow 72 \leq \langle b_1 = 22, b_2 = 40, b_3 = 20, b_4 = 63, b_5 = 22 \rangle.$$

Using the divisor $d = 10$*, it holds that* $\lfloor k/d \rfloor = 7, Q = \langle 2, 4, 2, 6, 2 \rangle$*, and* $\{v_I(L = Q) \mid I \subseteq \mathrm{At}(L)\} = \{2, 4, 6, 8, 10, 12, 14, 16\}$*, implying that the rule can be substituted with*

$$a \leftarrow 7 \leq \langle b_1 = 2, b_2 = 4, b_3 = 2, b_4 = 6, b_5 = 2 \rangle.$$

Case 2 If $\sum R < (k \bmod d)$ then substitute (2.1) with $a \leftarrow \lfloor k/d \rfloor + 1 \leq \langle L = Q \rangle$.

Example 19 *For the rule*

$$a \leftarrow 39 \leq \langle b_1 = 11, b_2 = 20, b_3 = 10, b_3 = 30, b_3 = 12 \rangle,$$

using the divisor $d = 5$*, it holds that* $k \bmod d = 4, R = \langle 1, 0, 0, 0, 2 \rangle$*, and* $\sum R = 3$*, implying that the rule can be substituted with*

$$a \leftarrow 8 \leq \langle b_1 = 2, b_2 = 4, b_3 = 2, b_4 = 6, b_5 = 2 \rangle.$$

Case 3 If for some $i \in \{1, \ldots, n\}$, it holds that $k \bmod d \leq r_i$ and $r_i = \sum R$, then set $q_i := q_i + 1$ and substitute (2.1) with $a \leftarrow \lfloor k/d \rfloor + 1 \leq \langle L = Q \rangle$. In this setting, there is one non-zero residue and it is meaningful on its own. Since there is only one, it can be accounted for by incrementing the respective quotient.

Example 20 *For the rule*

$$a \leftarrow 72 \leq \langle b_1 = 20, b_2 = 40, b_3 = 27, b_4 = 60, b_5 = 20 \rangle,$$

using the divisor $d = 10$*, it holds that* $k \bmod d = 2, R = \langle 0, 0, 7, 0, 0 \rangle$*, and for* $i = 3$ *that* $r_i = \sum R$*, implying that the rule can be substituted with*

$$a \leftarrow 8 \leq \langle b_1 = 2, b_2 = 4, b_3 = 3, b_4 = 6, b_5 = 2 \rangle.$$

Case 4 If $k \bmod d \leq \min R$ then substitute (2.1) with $a \leftarrow \lfloor k/d \rfloor \leq \langle L = Q \rangle$. Intuitively, the requirement here is for the residues of the weights to always overcome the residue of the bound unless all literals in L are false. Under this condition, the impact of the residues is fixed. In the case that all of the literals are false, given that $1 < d < k$, both the original and new bodies are unsatisfied and no discrepancy arises.

Example 21 *For the rule*

$$a \leftarrow 71 \leq \langle b_1 = 21, b_2 = 42, b_3 = 21, b_4 = 63, b_5 = 21 \rangle,$$

using the divisor $d = 10$*, it holds that* $k \bmod d = 1, R = \langle 1, 2, 1, 3, 1 \rangle$*, and* $\min R = 1$*, implying that the rule can be substituted with*

$$a \leftarrow 7 \leq \langle b_1 = 2, b_2 = 4, b_3 = 2, b_4 = 6, b_5 = 2 \rangle.$$

It is nontrivial to choose a suitable divisor d that leads to satisfied premises for any of the simplification steps. However, a plausible approach is to try a selected number of divisors $1 < d < k$ to see if any of them yields success.

Chapter 6

Experiments

In this chapter, we describe a tool implementing several of the normalizations presented and discussed in this thesis. The implemented normalizations are evaluated experimentally, in terms of the number of produced rules, as well as several performance metrics reported by the state-of-the-art ASP solver CLASP. To this end, a group of benchmark classes have been selected and solved, both before and after the normalization of extended rules. These experiments have been repeated with various normalization techniques in order to compare their effect on the search performance. The relevant tool is discussed briefly in Section 6.1, the size experiments are presented in Section 6.2, and the performance experiments in Section 6.3. Moreover, the implementations have been subject to automated equivalence checks, in order to ensure their correctness for sampled cardinality and weight rules. A further performance evaluation of the computational effort involved is presented in Section 6.4.

6.1 LP2NORMAL2

The techniques presented in this thesis are implemented in the tool LP2NORMAL2 (v. 1.10).[1] The program reads and writes logic programs in the output format of the grounders LPARSE and GRINGO and the input format of solvers like SMODELS and CLASP. Given such input, the tool normalizes any included extended rules, and in particular cardinality and weight rules, into normal rules. The outcome is a normal logic program that is visibly strongly equivalent to the original program. An example command line for using the tool is shown in Figure 6.1. In this section, we briefly describe how the relevant methods are implemented in LP2NORMAL2, and how they differ from their formal definitions.

[1]Available at `http://research.ics.aalto.fi/software/asp`.

```
gringo logic-program.lp | lp2normal2 | clasp
```

Figure 6.1: An example command line for grounding, normalizing, and finding an answer set for a logic program.

The behaviour of LP2NORMAL2 is controlled through command line options, which can be used to prohibit the normalization of given types of rules, or determine the applied normalization techniques. A list of all options and their effects is printed in response to `--help` and `--help=2` arguments, in a short and long format, respectively. Regarding cardinality rules, there are options for the techniques presented in Chapter 3, in addition to a number of other methods. Among them, there is an automatic cardinality rule translation that, at the moment of writing this thesis, constructs a merge-sorter built from primitives chosen automatically from odd-even mergers and totalizer components. The choices are made in order to reduce the sum of the numbers of used auxiliary atoms and rules, in comparison to either technique used alone. This is also used as a default setting for cardinality rule normalization.

Options for weight rules cover translations based on sequential weight counters and weight sorting networks. The latter are implemented using tares, and by default, heuristically chosen mixed-radices and structure sharing, as described in Sections 4.5, 4.6, and 4.7. Furthermore, the merging and sorting primitives used in building weight sorting networks are determined by options used to specify cardinality rule normalizations. For example, if odd-even merge-sorters are chosen, then they are used for normalizing cardinality rules and constructing weight sorting networks. In the experiments following in this chapter, the default, automatic mode is used for weight sorting networks.

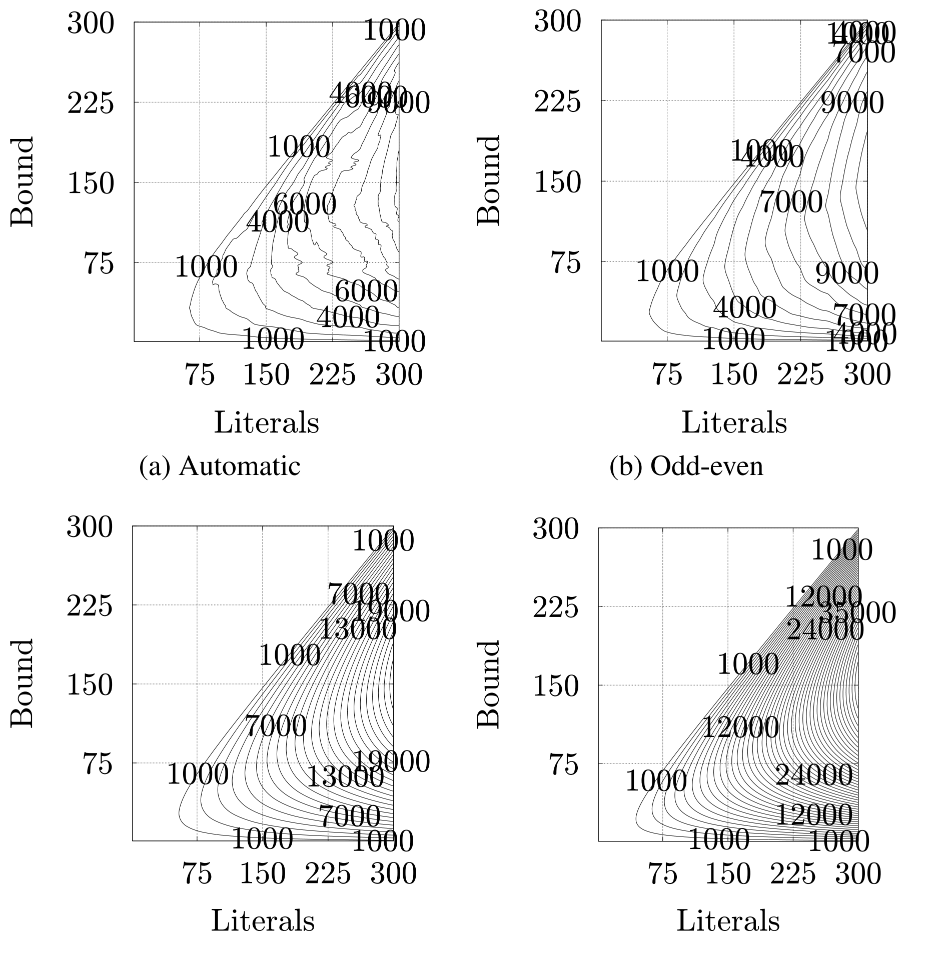
Moreover, all of the cardinality and weight rule normalizations described in this thesis have been enhanced with a form of *symbolic evaluation*. The constructions underlying the normalizations generally produce more information than necessary to perform a single bound check. Therefore, the constructions are symbolically evalated to include only required rules. In terms of implementation, this is a matter of tracing a *cone of influence* that originates from a wanted portion of the output and propagates deeper into the constructions. In the process, required auxiliary atoms are identified, and afterward, the rules needed to define the remaining necessary atoms are generated. The pruning can result in noticeably more concise normalizations. Finally, the weight rule simplification techniques presented in Chapter 5 are implemented mostly as described, with the addition that the included residue analysis technique is carried out by considering prime numbers less than the bound and the greatest body weight for divisors. Moreover, all potentially demanding subset-sum-type calculations involved in the simplifications are aborted if results are not obtained quickly. The effect of symbolic evaluation is discussed in Section 6.2, where the size of translations is under experimentation, and the significance of the weight rule simplifications in Section 6.3, where solving performance is being considered.

6.2 Compactness of Normalizations

In this section, we study the compactness of the normalization techniques covered in this thesis experimentally. Despite knowning asymptotic limits for the size of normalizations, including the ones discussed in this section, the witnessed size of translations in practice is of interest. The normalization techniques covered in this thesis can all be improved by pruning, or symbolic evaluation, bringing in a factor that is not conveyed in the known theoretical limits. For the purpose of these experiments, the size of a translation is measured as the number of produced normal rules. The number relates to the amount of memory required by tools used to process the results, and thus may further affect the performance of such tools. Actual performance experiments will, however, follow in Section 6.3.

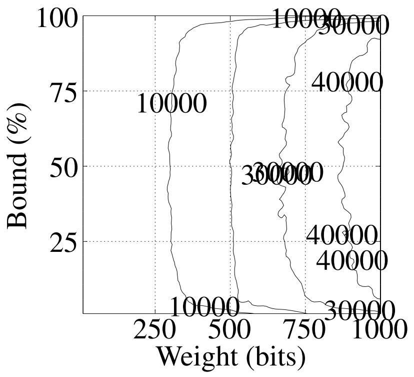
The cardinality rule normalization techniques, discussed in Chapter 3, were evaluated for conciseness, and the results are given in Figure 6.2. For each literal count $1 \leq n \leq 300$ and a bound $1 \leq k \leq n$, a weight rule (2.1) with $W = \langle 1, \ldots, 1 \rangle$ was normalized using four techniques. The numbers of produced normal rules are displayed using level curves, such that each gap between two curves represents a difference of $1{,}000$ normal rules. For each normalization, parameters with k close to $n/2$ gave rise to the largest translations. The concisest normalization scheme out of the four turned out to be the automatic translation choosing between odd-even mergers and totalizer components, resulting in slightly above $10{,}000$ normal rules at maximum within the chosen range of parameters. The automated translation scheme was a slight improvement over odd-even merge-sorters, described in Section 3.3, which were encoded in over $11{,}000$ rules at worst. In contrast to the asymptotic upper bound of $\mathcal{O}(k \times (\log n)^2)$ for the size of odd-even merge-sorters, the results indicate an almost symmetric benefit from bounds k close to n as well as to 1. In turn, totalizers, from Section 3.3, required well over $20{,}000$ rules. Finally, sequential counters, discussed in Section 3.1, exceeded $40{,}000$ in the number of rules in the worst case. In light of these results, the odd-even merge-sorting technique is significantly more concise compared to the totalizer and sequential counter based techniques. Despite this, the incorporation of totalizer components within odd-even merge-sorting structures leads to size benefits, as witnessed by the conciseness of the automatic technique.

In Figure 6.3, similar size measurements for weight rule normalizations are shown. We drew 5 weight rules at random for every combination of a bound percentage p between $1\,\%$ and $100\,\%$, and a number of bits b between 1 and $1{,}000$.

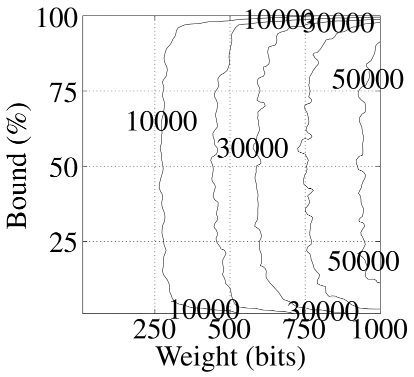


(a) Automatic
(b) Odd-even
(c) Totalizer
(d) Sequential

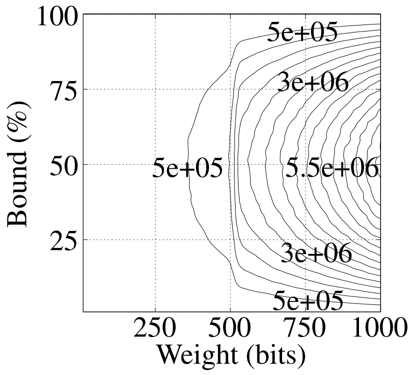
Figure 6.2: Level curves of numbers of rules in (a) sorters using automatically selected mergers, (b) odd-even merge-sorters, (c) totalizers and (d) sequential counters.



(a) Sharing and Mixed-radices



(b) No sharing nor mixed-radices



(c) Sequential

Figure 6.3: Level curves of numbers of rules in (a) weight sorting networks with sharing and mixed-radices, (b) weight sorting networks without sharing nor mixed-radices, and (c) sequential weight counters. The curves are set $10{,}000$ normal rules apart for (a) as well as (b), and $500{,}000$ normal rules apart for (c).

The random sampling was carried out, by first placing b bits at random into a binary $m \times n$-matrix of $m = 2 \times b/(1 + \lfloor \log_2(b) \rfloor)$ rows and $n = 1 + \lfloor \log_2(b) \rfloor$ columns. Every column $1 \leq i \leq n$ was then interpreted as the binary representation of a body weight w_i, and the bound of the rule was chosen to be $k = (p/100\,\%) \times \sum_{i=1}^{n} w_i$. Each drawn weight rule was normalized separately, and the median number of normal rules for every set of 5 weight rules was recorded for display in the figures. These experiments were repeated using three normalization techniques: weight sorting networks with and without mixed-radices and structure sharing, as described in Chapter 4, and sequential weight counters, as described in Section 3.2.

The effect of the proposed mixed-radix and structure sharing techniques in the results is positive, and the effect strengthens as the number of bits in weights increases. Interestingly, significant reductions in the number of rules due to the bound take place only for markedly low and high percentages. Regarding the sequential weight counter, the produced normal programs are far less concise for most of the chosen range of parameters. Moreover, the sampling technique brings about dramatic jumps in the results around powers of two for values of b. At these barriers the number of literals, n, increases, and evidently leads to more complex sequential weight counters, whereas the weight sorting network based techniques do not show such visible behaviour. Instead, the latter appear to depend on the number of bits more directly, as suggested by the order of their size, $b \times (\log_2 b)^2$. As with the results for cardinality normalizations, presented in Figure 6.2, the effect of the bound is positive for both low and high values, and for all of the tried translations. This holds despite the fact that, as far as we know, no asymptotic limit given for the techniques reflects this. We attribute this behaviour to the applied symbolic evaluation, discussed in Section 6.1.

6.3 Effect on Solver Performance

In this section we assess the effects of different normalization techniques on solver performance. We begin by looking into benchmark problems with cardinality rules and then more general weight rules. For performance evaluation, instances are first preprocessed by performing simplification and normalization after which they are passed on to the ASP solver CLASP (v. 3.0.4) [27]. All of the benchmarks are run sequentially on a Linux machine with Intel Xeon E5-3650 CPUs. Each run is allowed a maximum of 3GB of memory and 20 minutes of CPU time.

Table 6.1 displays sums of runtimes in seconds, numbers of constraints and conflicts reported by the back-end solver CLASP on listed benchmark classes using different normalization options. The "Native" column lists results obtained using CLASP without normalizations, and instead relying on the native weight handling built into CLASP [26]. For these and all other results, internal translations of CLASP were turned off as well. The following three columns correspond to merge-sorting based normalizations of Section 3.3, which were applied prior to running the back-end solver. The merge-sorting columns vary by the used mergers from an "Automatic" selection of mergers, to "Odd-even" mergers of Section 3.3, to "Totalizers" [7]. The automatic mode chooses between the latter two options, for each encoded merger, and aims to minimize the sum of output atoms and rules as described in Section 6.1. The "Sequential" column stands for the sequential counting based normalization from Section 3.1. The problem instances belong to classes of NP-complete problems from the second answer set programming competition [19]. Only benchmark problems containing significant cardinality rules are included and only cardinality rules are normalized. The first two benchmark classes, ConnectedDominatingSet and WeightBoundedDominatingSet, involve graph theoretical search problems in which the size of a solution is constrained by a cardinality rule. The latter of these contains many small weight rules in addition, which are left as is. The following benchmark problem, WireRouting, includes several cardinality rules with bounds of 1, 2, and 3. The remaining three problems contain small cardinality rules with bounds of 1 and 2. On the last three problems, all of the cardinality rules belong to special cases handled in a simplification stage of LP2NORMAL2. In particular, a cardinality rule with n body literals and a bound of 1 is replaced with n normal rules, and a cardinality rule with a bound of 2 and a limited number of body literals $n \leq 6$ is replaced with $(n \times (n-1))/2$ rules.

Regarding the results in Table 6.1, all of the employed normalization techniques show significant improvements in terms of solving time on the ConnectedDominatingSet benchmark class. The increase in constraints due to normalization is modest on these instances, while the number of conflicts is reduced by an order of magnitude. On the WeightBoundedDominatingSet instances, normalizations lead to significant numbers of added constraints, and even deteriorate search in terms of conflicts. The native configuration terminates faster in consequence. Concerning the WireRouting problem, the addition of rules generated via normalization is balanced by a decrease in the number of conflicts encountered during search. Measured in runtime, the effect of these changes is neutral when using the automated encoding selection, and negative when using the other normalization methods. For the GraphColoring instances, the effect of normalization is less significant. Both constraints and conflicts increase in numbers leading to increased runtimes as well, although the net effect is insignificant. Similarly on the Labyrinth instances, normalizations yield an adverse outcome on all three measures: time, constraints and conflicts are increased due to normalizations. On the Solitaire benchmark, normalizations appear to be mildly beneficial. As for the different normalization strategies, there is a clear division between the merge-sorting

# Instances ↓ Benchmark	Native	Automatic	Odd-even	Totalizer	Sequential
21 C-DominatingSet	937	22	33	15	62
# Constraints	116,512	142,834	150,217	160,169	171,297
# Conflicts	50,412,862	317,198	451,457	268,140	753,748
29 WB-DominatingSet	47	225	261	758	3,455
# Constraints	21,943	287,563	265,921	679,025	941,592
# Conflicts	1,622,922	2,325,754	2,147,653	5,581,009	18,012,508
23 WireRouting	585	548	747	769	876
# Constraints	702,514	1,365,094	1,378,994	1,360,296	1,402,470
# Conflicts	2,092,700	1,564,975	1,607,154	1,873,924	1,714,031
29 GraphColouring	21,117	21,480	21,625	21,407	21,435
# Constraints	143,400	179,130	179,130	179,130	179,130
# Conflicts	148,988,685	155,723,398	160,301,493	167,312,140	155,370,861
29 Labyrinth	2,910	5,557	5,105	4,844	4,827
# Constraints	26,041,106	28,987,293	28,987,293	28,987,293	28,987,293
# Conflicts	1,681,467	2,025,191	2,025,137	2,025,200	2,024,988
27 Solitaire	7,875	6,400	6,372	6,341	6,365
# Constraints	480,249	690,228	690,228	690,228	690,228
# Conflicts	24,809,684	21,482,729	20,254,937	17,101,669	20,859,466
150 Summary	33,471	34,232	34,143	34,134	37,020
# Constraints	27,505,724	31,652,142	31,651,783	32,056,141	32,372,010
# Conflicts	229,608,320	183,439,245	186,787,831	194,162,082	198,735,602

Table 6.1: Sums of runtimes in seconds, numbers of constraints, and conflicts encountered by CLASP on instances with cardinality rules.

based techniques and the sequential counter. On the last three benchmark classes, the normalization results are indifferent to the used normalization scheme due to the performed simplifications. Between the different mergers, the most pronounced difference is realized on the WeightBoundedDominatingSet instances, on which the use of totalizers leads to over twice the constraints in comparison to odd-even mergers. On the first three benchmark classes, automatic merger selection brings gains on average in comparison to both odd-even merge-sorters and totalizers.

Table 6.2 contains sums of runtimes, constraints and conflicts obtained on another set of benchmarks and normalization schemes. For this set of results, cardinality rules are left as is, and only weight rules are normalized. Prior to normalization, the simplification strategies of Chapter 5 are applied until a fixpoint is reached. The "Native" column is as before. The following four columns indicate normalizations utilizing four configurations of techniques presented in Chapter 4. The "Mixed" header indicates the use of a heuristically chosen mixed-radix base, described in Section 4.6, in contrast to a binary base indicated by the "Binary" header. Likewise, the "Shared" headers indicate that structural sharing of Section 4.7 is enabled, while "Independent" headers indicate that it is not. The remaining two normalizations indicated by "SWC" and "ROBDD" are included for reference, and are based on Sequential Weight Counting described in Chapter 3 and Reduced Ordered Binary Decision Diagrams (ROBDDs) [2]. The benchmark

# Instances		Mixed		Binary			
↓ Benchmark	Native	Shared	Independent	Shared	Independent	SWC	ROBDD
11 Bayes-Find	202	30	164	246	165	1,721	2,290
# Constraints	34,165	347,450	417,768	325,033	353,381	4,948,058	4,976,930
# Conflicts	12,277,288	181,957	822,390	1,056,764	868,056	616,930	713,076
11 Bayes-Prove	1,391	492	1,316	631	890	2,587	2,682
# Constraints	34,165	344,637	414,967	322,212	350,596	4,947,717	4,976,746
# Conflicts	52,773,713	1,393,935	3,293,955	1,933,103	3,165,312	1,459,105	1,113,222
11 Markov-Find	2,426	2,770	1,845	2,682	2,966	5,224	5,276
# Constraints	1,580,164	2,176,067	2,296,063	2,309,147	2,436,769	36,699,300	3,693,6376
# Conflicts	1,771,663	1,276,599	1,092,467	1,130,776	1,178,797	318,771	374,477
11 Markov-Prove	2,251	3,294	3,428	3,255	3,229	5,402	5,370
# Constraints	1,580,164	2,182,157	2,302,171	2,307,991	2,435,603	36,694,525	36,931,627
# Conflicts	1,806,525	1,788,800	1,720,270	1,521,272	1,452,042	317,555	388,755
38 Fastfood	10,277	12,843	14,156	13,756	13,479	17,867	18,268
# Constraints	928,390	2,880,725	3,640,856	2,826,606	3,667,538	11,860,656	12,893,886
# Conflicts	122,423,130	47,566,08	42,794,938	44,148,615	49,035,512	8,940,612	5,975,006
12 Inc-Scheduling	257	1,340	1,330	1,481	1,581		
# Constraints	2,304,166	7,161,226	8,166,527	7,274,513	8,570,210		
# Conflicts	82,790	127,628	134,987	218,224	173,849		
15 Nomystery	4,907	4,236	3,332	4,290	3,512	4,739	5,311
# Constraints	845,321	1,678,580	2,330,329	1,725,458	2,459,603	5,115,156	5,124,634
# Conflicts	10,765,572	3,216,072	2,161,566	3,207,353	2,092,378	2,047,501	1,834,677
109 Summary	21,715	25,009	25,576	26,345	25,827		
# Constraints	7,306,535	16,770,842	19,568,681	17,090,960	20,273,700		
# Conflicts	201,900,681	55,551,076	52,020,573	53,216,107	57,965,946		
109 Summary	21,715	24,758	26,611	26,524	26,063		
without	7,306,535	17,279,805	21,632,440	17,665,922	22,358,451		
simplification	201,900,681	52,264,536	46,809,044	56,247,153	51,814,629		

Table 6.2: Sums of runtimes in seconds, numbers of constraints, and conflicts encountered by CLASP on instances with weight rules.

classes are selected from five application areas, dealing with Bayesian network structure learning [18, 32], Markov network structure learning [17], the Fastfood logistics problem [10] as well as the Incremental scheduling and Nomystery planning tasks from the fourth answer set programming competition [4]. Each instance from the first two classes originally contained an optimization statement, which was converted into a weight rule. More specifically, we created two variants of each original problem, indicated by the suffixes "Find" and "Prove" in the table: the first using the tightest bound possible while keeping the problem satisfiable, and the second by adding the most generous bound possible while keeping the problem unsatisfiable. The other instances already were decision problems. The last six rows display summaries, and for the last three, simplification techniques of Chapter 5 are turned off.

From Table 6.2, we find that the proposed normalizations lead to mostly reduced number of conflicts and solving time on both Bayesian problem variations. At best, the number of constraints increases by roughly an order of magnitude, whereas conflicts reduce by two orders of magnitude. There is, however, high variance between the different option combinations, which in this case appear to favor both the use of mixed-radix bases and structure sharing. The reference normalizations lead to reduced conflicts as well, but the translation sizes in terms of constraints refrain the total solving time from decreasing. On the Markov instances, solving times fluctuate only slightly from one technique to the other. For the Fastfood benchmark, normalizations vastly decrease the number of encountered conflicts during the search, and the SWC and ROBDD based techniques more than the others. Despite this, the negative effect of the number of introduced constraints leads to higher total runtimes for all normalization schemes. Interestingly, on the Incremental scheduling instances, both constraint and conflict counts increase due to normalizations, and together amount to significantly higher runtimes. The problem size resulting from the SWC and ROBDD translation schemes is even prohibitive here, and in view of missing entries, caused by surpassed memory limits, they are also omitted in summaries. The normalizations yield runtime improvements on the Nomystery planning problem, although the proposed structure sharing appears to hurt performance in this case. Regarding simplifications, the summaries indicate that they amount to noticeable reductions in translation size, but do not lead to apparent, consistent runtime improvements.

Overall, we conclude that both cardinality and weight rule normalizations are mostly beneficial for the subsequent search in terms of numbers of conflicts. For certain benchmark classes the reductions lead to decreased solving time, whereas for other classes the increased translation sizes counterweigh the benefits and lead to increased solving time. Out of the inspected normalization techniques, the proposed ones proved to be more succint and lead to better runtimes than the reference techniques. The novel structure sharing algorithm decreased translation sizes noticeably with both mixed-radix and binary bases.

The LP2NORMAL2 (v. 1.7) preprocessing tool was submitted to the fifth answer set programming competition [12] as part of several translation-based solving systems. One of the submitted systems, LP2NORMAL2+CLASP, relates with the approach followed in this section by first normalizing instances and then using CLASP as a solving back-end. However, for the submission normalizations were configured to apply only conditionally such that very large cardinality and weight rules were left unnormalized. Regarding benchmarks, the competition benchmark suite consisted of four tracks classified according to language features. The LP2NORMAL2+CLASP system came in first place in the “Advanced” track of the competition, which specifically involved cardinality and weight rules.[2]The success gives an indication of the feasibility of the translation-based approach even in a truly competitive setting.

[2]The results are available at `https://www.mat.unical.it/aspcomp2014`.

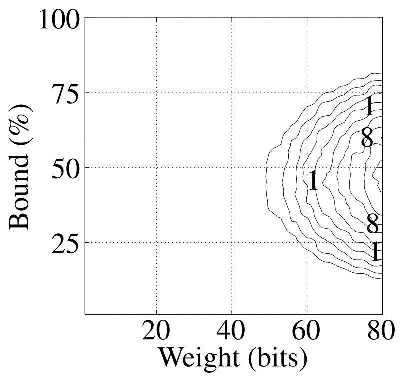


Figure 6.4: Level curves of logarithmic runtimes of CLASP for verifying that all VSE-models of a weight rule map to VSE-models of its normalization, when using weight sorting networks with mixed-radices and sharing. Contour lines are drawn at doubling time intervals and annotated with runtimes in seconds.

6.4 Correctness Testing

In the developement of the normalization tool LP2NORMAL2, we have performed automated equivalence checking as a quality control procedure. Using the tools CLASSIC, LPEQ, and CLASP, we have verified that normalizations of selected cardinality and weight rules remain visibly strongly equivalent to the original rules. These tools allow, given enough computational resources, to prove the existence or lack of a counterexample to the visible matching of the VSE-models of a weight rule and its normalization. Namely, by a search of a VSE-model of one program P, which does not visibly correspond to a VSE-model of the other program Q, the tools either prove or disprove the visible strong equivalence $P \equiv_{\mathrm{vs}} Q$, discussed in Section 2.2. In this section, we present experimental results on the computational effort required to perform such equivalence checks.

Similarly to the size experiments in Section 6.2, we sampled weight rules from those having a bound set to a ratio between $1\,\%$ and $100\,\%$ of their total body weight, and the number of bits in their weights ranging between 1 and 80. For each pair of these parameters, 5 weight rules were drawn again, and the median verification time taken to prove that VSE-models of the weight rule have correspondents within the VSE-models of the normalization was recorded. The timing results for weight sorting networks with mixed-radices and sharing are given in Figure 6.4. We carried out such tests also without mixed-radices and sharing, as well as using sequential weight counters. Furthermore, we checked the other direction as well, to ensure that VSE-models of the normalizations mapped to VSE-models of the original weight rules. Intriguingly, these other results were, by visual inspection, identical to the shown ones to such an extent that they are not displayed. In contrast to the earlier size measurements, the level curves here are set apart logarithmically, such that each gap corresponds to a doubling of verification time. Taking the changed range of the x-axis into account, Figure 6.4 indicates a primarily exponential relation between verification time and translation size displayed in Figure 6.3(a).

Chapter 7

Discussion

This chapter provides a discussion of related work and conclusions.

7.1 Related Work

Translations for cardinality rules include schemes in which satisfied literals are counted one by one. One of such translations is structured in a grid formation that consists of $\mathcal{O}(k \times n)$ atoms and rules, and which has been previously used in ASP in [33, 41]. In this work, we followed an almost identical sequential counter-based translation proposed for SAT in [42]. A number of other cardinality translations rely on merge-sorting, in which input is split and added up recursively in halves, and intermediate results are expressed in unary notation. The strategy used for adding up the numbers varies between translations. For one, odd-even merge-sorters of size $\mathcal{O}(n \times (\log n)^2)$ were introduced as circuits in [8] and have since been used to encode cardinality constraints into SAT [5, 21] and ASP [10]. Similarly, pairwise sorters introduced in [39], which are of the same asymptotic size, have been used in SAT encodings [14, 15]. Another type of a merger gives rise to totalizers [7] consisting of only $\mathcal{O}(n \times \log n)$ atoms at the expense of $\mathcal{O}(n^2 \times \log n)$ clauses or rules. In [3], the choice of different sorting and merging primitives is exploited by picking the most suitable primitive individually for each input size. We implemented a similar automated mode in LP2NORMAL2 for making choices between odd-even and totalizer-based components. Designed for especially small bounds k, an asymmetric, cascading structure resulting in cardinality networks is proposed in [5] as an alternative to merge-sorters, while still relying on merging primitives. The so far mentioned cardinality translations are monotone in the sense that they can be viewed as circuits constructed solely of AND and OR gates.

A more concise, linear sized SAT encoding that lacks this property is given by Warners in [45], where partial sums of satisfied literals are added up in binary representation. The encoding of Warners is applicable to weight rules in general, and in terms of size is of the order of $\mathcal{O}(n \times \log w_{\max})$, where $w_{\max}$ denotes the largest weight. Regarding monotone translations, the sequential counters in [42] generalize to weight rules as shown in [31] and as described in this thesis. In the generalized translation, satisfied literals are again counted one by one, and only the domain of the added numbers increases. The size of the translation remains $\mathcal{O}(k \times n)$. A very similar translation is presented in the context of ASP in [24], where weight rules are studied in terms of *nested expressions*. The translation therein relies on a dynamic programming approach. Moreover, Abío et al. [2] encode pseudo-Boolean constraints, which are similar to weight rules, as Reduced Ordered Binary Decision Diagrams, and further encode them in SAT. These three translations, given in [24], [31], and [2], relate closely to each other, and our experimental results on the latter two gave similar results for both.

A completely different approach to the above, introduced in [21] and followed in [7], relies on performing arithmetic in a mixed-radix or binary base using mergers and sorters for adding numbers. Furthermore, in [7] the arithmetic calculations are offset by the use of a tare. The resulting translation is consequently simpler and monotone. For this work, we adapted the latter, (global) polynomial watchdog translation of [7], originally used in SAT, for use in ASP by making use of sorting and merging primitives encoded in ASP. We developed the translation further by using heuristically chosen mixed-radix bases and a structure sharing algorithm to compress the translation. Regarding mixed-radix bases, a more exhaustive method for finding such is used in [21] and complete search algorithms are presented in [16]. The order of the size of these translations, in [21], [7], and this thesis, is the same as that of a sorter with as many inputs as there are bits in the binary representation of all input weights. Given the different sorters used in each work, this makes respective orders of $b \times (\log b)^2$, $b^2 \times \log b$, and $b \times (\log b)^2$, where b stands for the number of bits.

7.2 Conclusions

Cardinality and weight rules are important primitives in ASP. They allow a concise representation of statements about the cardinality of sets of literals, and more generally, of weighted sums of literals. Similar concepts exist in other research areas, and weight rules can be closely equated with, for example, 0-1 linear integer inequalities and pseudo-Boolean constraints. In this thesis, we explored a translation-based approach to implementing support for these extended rule types in ASP. Several strategies for substituting cardinality and weight rules with sets of normal rules were studied. In this way, weighted expressions found within the bodies of weight rules were encoded using normal logic programs.

A number of existing translations from the areas of SAT and ASP were described and defined. They were used both for reference, and for use as building blocks of novel translation schemes for weight rules. The existing and proposed techniques were compared with each other, as well as native weight rule solving techniques, by carrying out experimental studies on the conciseness of the translations and subsequent solving performance. In order to realize the experiments, a normalization tool was developed and implemented. By both measures, conciseness and performance, the novel methods led to improvements compared to earlier translation-based techniques. Using any of the translations generally reduced the number of encountered conflicts during search, which was, however, counterbalanced by the size of the encodings. On certain benchmark classes, the net effect on solving time was in clear favour of the novel translations. Furthermore, we similarly assessed the effect of weight rule simplification methods, and found them to slightly improve the compactness of subsequently executed translations, at no significant impact on the efficiency of search, however. The developed normalization tool LP2NORMAL2 was submitted to the fifth answer set programming competition, where it won the first place in the “Advanced” track of the competition as part of a selectively normalizing system with CLASP as back-end. The results emphasize the feasibility and promise of the translation-based approach, as well as the studied and developed translation schemes in particular.

The proposed translations were further studied in terms of correctness. To this end, equivalence notions for judging program transformations were used to prove that the proposed weight rule translations are correct particularly in the context of ASP, and within any sensible context program. In this process, a proof technique for showing the visible strong equivalence of programs from a wider class of programs with limited use of negation was devised. In addition to these formal considerations, we also studied correctness aspects experimentally, by using automated tools to prove the correctness of our implementations of translations for randomly drawn instances of weight rules. The time requirements to perform these checks were noted, and found especially feasible for small weight rules.

Bibliography

[1] *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 8148 of *Lecture Notes in Computer Science*, 2013. Springer.

[2] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Valentin Mayer-Eichberger. A new look at BDDs for pseudo-Boolean constraints. *Journal of Artificial Intelligence Research*, 45:443–480, 2012.

[3] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. A parametric approach for smaller and better encodings of cardinality constraints. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming*, volume 8124 of *Lecture Notes in Computer Science*, pages 80–96. Springer, 2013.

[4] Mario Alviano, Francesco Calimeri, Günther Charwat, Minh Dao-Tran, Carmine Dodaro, Giovambattista Ianni, Thomas Krennwallner, Martin Kronegger, Johannes Oetsch, Andreas Pfandler, Jörg Pührer, Christoph Redl, Francesco Ricca, Patrik Schneider, Martin Schwengerer, Lara Katharina Spendier, Johannes Peter Wallner, and Guohui Xiao. The fourth answer set programming competition: Preliminary report. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning* DBL [1], pages 42–53.

[5] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks: a theoretical and empirical study. *Constraints*, 16(2):195–221, 2011.

[6] Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2003.

[7] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New encodings of pseudo-Boolean constraints into CNF. In *International Conference on Theory and Applications of Satisfiability Testing*, volume 5584 of *Lecture Notes in Computer Science*, pages 181–194. Springer, 2009.

[8] Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314. Thomson Book Company, Washington D.C., 1968.

[9] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, 2009. IOS Press.

[10] Jori Bomanson and Tomi Janhunen. Normalizing cardinality rules using merging and sorting constructions. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning* DBL [1], pages 187–199.

[11] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[12] Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca. The design of the fifth answer set programming competition. *The Computing Research Repository*, abs/1405.3710, 2014.

[13] Keith L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, 1977.

[14] Michael Codish and Moshe Zazon-Ivry. Pairwise cardinality networks. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 154–172. Springer, 2010.

[15] Michael Codish and Moshe Zazon-Ivry. Pairwise networks are superior for selection. 2012. `http://www.cs.bgu.ac.il/~mcodish/Papers/Pages/pairwiseSelection.html`.

[16] Michael Codish, Yoav Fekete, Carsten Fuhs, and Peter Schneider-Kamp. Optimal base encodings for pseudo-Boolean constraints. In *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 189–204. Springer, 2011.

[17] Jukka Corander, Tomi Janhunen, Jussi Rintanen, Henrik Nyman, and Johan Pensar. Learning chordal Markov networks by constraint satisfaction. In *Proceedings of the 27th International Conference on Annual Conference on Neural Information Processing Systems*, volume 26 of *Advances in Neural Information Processing Systems*, pages 1349–1357, 2013.

[18] James Cussens. Bayesian network learning with cutting planes. In *Proceedings of the 27th International Conference on Uncertainty in artificial Intelligence*, pages 153–160. AUAI Press, 2011.

[19] Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Mirosław Truszczyński. The second answer set programming competition. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 5753 of *Lecture Notes in Computer Science*, pages 637–654. Springer, 2009.

[20] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.

[21] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2 (3-4):1–25, 2006.

[22] Thomas Eiter, Michael Fink, and Stefan Woltran. Semantical characterizations and complexity of equivalences in answer set programming. *ACM Transactions on Computational Logic*, 8(3), 2007.

[23] Wolfgang Faber, Gerald Pfeifer, Nicola Leone, Tina Dell'Armi, and Giuseppe Ielpa. Design and implementation of aggregate functions in the dlv system. *Theory and Practice of Logic Programming*, 8(5-6):545–580, 2008.

[24] Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5(1-2):45–74, 2005.

[25] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo: A new grounder for answer set programming. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271. Springer, 2007.

[26] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. On the implementation of weight constraint rules in conflict-driven ASP

solvers. In *Proceedings of the 25th International Conference on Logic programming*, volume 5649 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2009.

[27] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187: 52–89, 2012.

[28] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188: 52–89, 2012.

[29] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic programming*, pages 1070–1080. MIT Press, 1988.

[30] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, 2006.

[31] Steffen Hölldobler, Norbert Manthey, and Peter Steinke. A compact encoding of pseudo-Boolean constraints into SAT. In *Proceedings of the 35th Annual German Conference on KI 2012: Advances in Artificial Intelligence*, volume 7526 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 2012.

[32] Tommi Jaakkola, David Sontag, Amir Globerson, and Marina Meila. Learning Bayesian network structure using LP relaxations. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, volume 9 of *JMLR Proceedings*, pages 358–365. JMLR.org, 2010.

[33] Tomi Janhunen and Ilkka Niemelä. Applying visible strong equivalence in answer-set program transformations. In *Lifschitz Festschrift, Vol. 7265 of LNCS*, pages 363–379. Springer, 2012.

[34] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[35] Vladimir Lifschitz, David Pearce, and Agustín Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.

[36] Lengning Liu and Miroslaw Truszczynski. Pbmodels - software to compute stable models by pseudoboolean solvers. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 3662 of *Lecture Notes in Computer Science*, pages 410–415. Springer, 2005.

[37] John W. Lloyd. *Foundations of Logic Programming*. Springer, 1987.

[38] Victor W. Marek and Miroslaw Truszczynski. *Nonmonotonic logic - context-dependent reasoning*. Artificial Intelligence. Springer, 1993.

[39] Ian Parberry. The pairwise sorting network. *Parallel Processing Letters*, 2 (2):3, 1992.

[40] Alexander Schrijver. *Theory of linear and integer programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1999.

[41] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1):181–234, 2002.

[42] Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, 2005.

[43] Tommi Syrjänen and Ilkka Niemelä. The smodels system. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 2173 of *Lecture Notes in Computer Science*, pages 434–438. Springer, 2001.

[44] Edward P. K. Tsang. *Foundations of constraint satisfaction*. Computation in cognitive science. Academic Press, 1993.

[45] Joost P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, 1998.