Algebraic Techniques for Satisfiability Problems

Dissertation

Fakultät für Elektrotechnik und Informatik
Gottfried Wilhelm Leibniz Universität Hannover

Henning Schnoor

Algebraic Techniques for Satisfiability Problems

Von der

Fakultät für Elektrotechnik und Informatik
der Gottfried Wilhelm Leibniz Universität Hannover

zur Erlangung des Grades eines

Doktors der Naturwissenschaften
Dr. rer. nat.

genehmigte Dissertation

von

Dipl.-Math. Henning Schnoor

geboren am 17. November 1978 in Husum

2007

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.
1. Aufl. - Göttingen : Cuvillier, 2007
Zugl.: Hannover, Univ., Diss., 2007

978-3-86727-176-9

Referent:	Heribert Vollmer, Leibniz Universität Hannover
Korreferentin:	Edith Hemaspaandra, Rochester Institute of Technology
Tag der Promotion:	5. März 2007

Nonnenstieg 8, 37075 Göttingen
Telefon: 0551-54724-0
Telefax: 0551-54724-21
www.cuvillier.de

1. Auflage, 2007
Gedruckt auf säurefreiem Papier

978-3-86727-176-9

I love deadlines. I like the whooshing sound they make as they fly by.

Douglas Adams, 1952 - 2001

Danksagung

Es gibt viele Menschen, ohne die ich diese Arbeit nicht hätte schreiben können. Zuerst möchte ich meinem Doktorvater Heribert Vollmer danken für die Möglichkeit, in der Komplexitätstheorie zu arbeiten, die Einführung in interessante Forschungsthemen, viele gute Ratschläge in den letzten Jahren, und vieles mehr. Außerdem möchte ich mich bei meinen Co-Autoren für die gemeinsame Forschung bedanken, insbesondere bei Nadia Creignou für die Zeit in Marseille, in der viele der Resultate aus Kapitel 5 entstanden.

Weiter bedanke ich mich herzlich bei Edith Hemaspaandra, für die Einladung nach Rochester, und die Unterstützung bei der Planung meines Postdoc-Jahres.

Außerdem bedanke ich mich bei den Menschen, die mich Mathematik gelehrt haben, vor allem bei Joachim Reineke für die vielen interessanten Vorlesungen über Algebra, bei Karsten Steffens und dem verstorbenen Helmut Pfeiffer für die Einführung in mathematische Logik.

Ich möchte mich ebenfalls bei meinen Eltern bedanken, die mir ermöglichten, Mathematik zu studieren — ohne Euch wäre all dies unmöglich gewesen! Vor allem möchte ich mich bei meiner Frau Ilka bedanken, für ihre Liebe und Unterstützung.

Acknowledgment

There are many people without whom this work could not have been completed. First, I want to thank my advisor Heribert Vollmer for the opportunity to work in complexity theory, introduction to fascinating research topics, the good advice over the years, and much more. Then I want to thank my co-authors for the research we did together, and I want to express special thanks to Nadia Creignou for the nice times I spent in Marseilles, and in which many of the results from Chapter 5 developed.

I also thank Edith Hemaspaandra for the invitation to Rochester, and the great support in planning my postdoc year.

Additionally, I thank the people who taught me mathematics, most of all Joachim Reineke for his many fascinating lectures on algebra, and Karsten Steffens and the late Helmut Pfeiffer for the introduction to mathematical logic.

I also want to thank my parents for letting me study mathematics — without you, none of this would have been possible! Most of all, I want to thank my wife Ilka, for her love and support.

Zusammenfassung

Meine Dissertation beschäftigt sich mit verschiedenen Verallgemeinerungen des klassischen Erfüllbarkeitsproblems, das in der Komplexitätstheorie eine zentrale Rolle spielt. Durch Erfüllbarkeitsprobleme lassen sich viele wichtige Komplexitätsklassen charakterisieren, und viele offene Fragen der Komplexitätstheorie können in diesem Kontext formuliert werden. Das klassische Erfüllbarkeitsproblem ist die Frage, ob eine gegebene aussagenlogische Formel, in der Variablen sowie die Konnektoren $\wedge$ (und), $\vee$ (oder) und $\neg$ (nicht) vorkommem, eine erfüllende Belegung hat. Dieses Problem lässt sich auf verschiedene Weisen verallgemeinern.

Auf der einen Seite kann man die oben genannten Konnektoren durch beliebige Kombinationen von Boole'schen Funktionen ersetzen. Für viele dieser eingeschränkten Klassen kann man zeigen, dass das Erfüllbarkeitsproblem und andere interessante Probleme im Formelkontext effiziente Lösungen besitzen. In dieser Arbeit werden zwei wichtige Probleme für diese Formeln betrachtet: das *Formula Value Problem* ist die Frage, ob eine gegebene variablenfreie Formel zu 0 oder zu 1 evaluiert. Dieses Problem spielt eine zentrale Rolle in vielen Algorithmen, die zur Lösung von Formelproblemen eingesetzt werden. Weiterhin wird die Komplexität des Problems, zu einer gegebenen Formel die Menge ihrer erfüllenden Belegungen zu berechnen, klassifiziert.

Auf der anderen Seite betrachtet man Formeln in *konjunktiver Normalform*. Hierbei beschreiben die Formeln eine Menge von Klauseln in einfacher Form, die simultan erfüllt sein müssen. Klassische Vertreter von Problemen in diesem Kontext sind 3SAT oder 2SAT, wo die Klauseln aus Disjunktionen von maximal 2 bzw. 3 Literalen bestehen. Auch hier kann man verallgemeinerte Klauseln betrachten, die beliebige Relationen zulassen. Solche Probleme werden als *Constraint Satisfaction Problems* bezeichnet. Für den Boole'schen Fall wurde die Komplexität dieser Probleme von Thomas Schaefer im Jahr 1978 betrachtet und er zeigte, dass solche Probleme entweder effizient lösbar oder bereits NP-vollständig sind. In dieser Arbeit wird dieses "Dichotomie"-Resultat auf verschiedene Weisen verallgemeinert. Für das Boole'sche Problem wird eine vollständige Klassifizierung der effizienten Fälle erzielt, und es zeigt sich, dass die Dichotomie-Eigenschaft, dass also nur eine endliche Menge von "Komplexitätsgraden" angenommen wird, auch hier gilt. Weiterhin wird die Verallgemeinerung des quantifizierten Problems betrachtet, wo in den Formeln zusätzlich die Quantoren $\exists$ und $\forall$ auftreten. Für diese Formeln werden Erfüllbarkeitsprobleme, Zählprobleme und die Komplexität des Äquivalenzproblems untersucht. Auch hier ergeben sich dichotomieartige Resultate: die Probleme sind effizient lösbar oder vollständig für Stufen in der Polynomialzeit-Hierarchie.

Die wichtigste Technik, um komplexitätstheoterische Ergebnisse für diese Probleme zu erzielen, ist die Anwendung von verschiedenen algebraischen Abschlussoperatoren auf der Menge der Funktionen und der Relationen. Insbesondere existiert eine Galois-Verbindung zwischen diesen beiden "Welten", die es erlaubt, Resultate zu übertragen. Eine wichtige Frage in der Arbeit ist daher auch die nach den Grenzen der Anwendbarkeit dieses algebraischen Ansatzes.

Abstract

This thesis deals with several generalizations of the classical satisfiability problem, which plays a central role in complexity theory. Many important complexity classes can be characterized using satisfiability problems, and many open questions in complexity theory can be phrased in this context. The classical satisfiability problem is the question, if a given propositional formula built from variables and the connectors $\wedge$ (and), $\vee$ (or), and $\neg$ (not), has a satisfying truth assignment. This problem can be generalized in several ways.

First, we can allow arbitrary connectors instead of the three mentioned above. For many of the classes of formulas which can be defined this way, it can be shown that the satisfiability problem and other interesting problems can be solved by efficient algorithms. In this thesis, we study the complexity of the question if a given variable-free Boolean formula evaluates to 0 or to 1. We also consider the problem of computing the set of satisfying assignments for a given formula.

Second, formulas in *conjunctive normal form* are studied. Here, formulas consist of clauses of a very simple form, which must be satisfied simultaneously. Classical examples of these problems are 3SAT or 2SAT, where the clauses consist of disjunctions of up to 2 or 3 literals. Again, generalizations can be studied, where arbitrary relations are allowed as clauses. These problems are called *constraint satisfaction problems*. For the Boolean case, the complexity of these problems was determined in 1978 by Thomas Schaefer. He showed that such problems can either be solved efficiently, or are already NP-complete. In this thesis, we generalize this "dichotomy"-result in several ways. For the Boolean problem, we give a complete classification of the cases where efficient algorithms exist and show that the dichotomy property still holds: there is only a finite number of "complexity degrees" which arise in this classification. Further, we study the corresponding quantified problem, where formulas may additionally contain the quantifiers $\exists$ and $\forall$. For these formulas, we study the complexity of satisfiability problems, counting problems, and the equivalence problem. Again, we show dichotomy-like results: the problems are either solvable by efficient algorithms, or complete for levels of the polynomial hierarchy.

The main techniques that we use to obtain results on the complexity classifications of these problems are applications of different algebraic closure operators on the set of functions and relations. There is an interesting Galois correspondence which allows to transfer results from one type of restriction to the other. An important question in this thesis is the question for limitations of the applicability of these techniques.

Schlagworte

- Komplexitätstheorie
- Erfüllbarkeitsprobleme
- Algebraische Methoden in der Komplexitätstheorie

Keywords

- Complexity Theory
- Satisfiability Problems
- Algebraic Methods in Complexity Theory

Contents

List of Figures

Introduction

In computer science, the main task is to study the structure of computational problems, and possible algorithms to solve them. Recursion theory has provided many answers to the question which of the problems appearing in a computer scientist's every day life can be solved with an algorithm, and, more importantly, which cannot. In fact, by a simple argument comparing the number of possible algorithms and the number of possible problems, it is evident that "most" problems cannot be solved by any algorithm at all. Recursion theory also provided a suitable model for computation, which is independent of whatever kind of computer hardware might be in fashion at a given date: the Turing machine is both universal enough to serve as the general definition of a "computer," and simple enough for the researcher to prove results without too much technical reliance on the model itself.

For problems appearing in practice, the answer "there is an algorithm to solve it" is not entirely satisfying. Usually, we are interested in an algorithm solving the problem at hand using as few resources as possible. Among the most important measurements of resources are the time needed for the computation, and the memory used by an algorithm. Therefore, questions like "is there an algorithm to solve this problem in a time which is linear in the input length" become important. This is where complexity theory has provided many answers, and, maybe as important, interesting questions.

One of the most basic questions that complexity theory considered was the question how to define an efficient algorithm. The answer which is generally agreed on by people working in the field today is the following: an algorithm is efficient if the time it needs to perform its task on a Turing machine is polynomial in the length of its input. It can be shown that this class is "robust," meaning that if we study the class of problems solvable on a "real" computer in polynomial time instead, we get the same class of problems. Thus, the class P containing all problems with a polynomial-time algorithm is considered to contain all problems which can be solved "efficiently." In addition to this class, many other complexity classes have been defined, which are meant to group problems where the computational power required to solve them is similar. The question to determine for some problem, in which complexity class it belongs, is therefore the same as asking what resources we need to solve it.

It is obvious that positive results in the way of "there is an efficient algorithm to solve problem A" can be shown by simply stating an algorithm for the problem at hand, proving its correctness and analyzing its running time. But what about negative results, proving that there is *no* efficient algorithm for a given problem? Results of this kind have proven to be much more difficult to achieve. In fact, for many problems which are very relevant in practice, it is unknown if an efficient algorithm can exist.

To be able to compare the complexity of given problems, the notion of *reductions* was introduced. In this way, even if we do not know if we can solve the problems A or B efficiently, it is possible to prove statements like "either both A and B have an efficient algorithm, or none of them has." Using this concept, it was shown that many problems appearing in practice are "equivalent" in a certain way: either all of them can be solved by an efficient algorithm, or none can. These problems form the fundamental class of NP-complete problems. The question if an efficient algorithm for this class of problems exists is one way to phrase complexity theory's most important open question: the "P-NP"-problem. The first known mention of this problem is in a letter by Kurt Gödel to John von Neumann, where he asks if there is a better method to prove first-order formulas than simply testing all combinations. Despite considerable efforts by computer scientists and mathematicians for more than 35 years, this question remains open. Most researchers believe that the answer to it is "no," and therefore to say that a problem is NP-complete is now generally understood as meaning that there probably is no efficient algorithm for it.

But what about cases "in between" efficiently-solvable and NP-complete? It is very conceivable that there are problems which are "easier" than the NP-complete problems, but still do not have an efficient algorithm. Under the assumption that the NP-complete problems themselves cannot be solved efficiently, this, and in fact a much stronger result, has been proven by Richard Ladner in [Lad75b]: there are infinitely many "degrees of complexity" between P and NP. It is of interest that very few "natural" problems appear to lie in these intermediate degrees. One of the well-known candidates for such a problem is graph isomorphism, which is the problem to determine if two given graphs are mathematically the same structure.

From the very beginning of the study of these problems, in fact starting withthe above-mentioned letter by Gödel, propositional formulas lay at the heart of the discussion. One of the most important problems in complexity theory is the *satisfiability problem*, which is the following: given a propositional formula, determine if there is an assignment which makes the formula true. For example, consider the formula

$$\varphi_1 = x \wedge (y \vee \overline{x}).$$

This formula can easily seen to be *satisfiable*, by setting both variables x and y to "true." On the other hand, consider the formula

$$\varphi_2 = (\overline{x} \vee y) \wedge (\overline{y} \vee z) \wedge x \wedge \overline{z}.$$

This formula is not satisfiable: for any "true/false"-assignment to the variables x, y, and z, the formula is false. The satisfiability problem seems like a simple enough question to be solved by an algorithm: simply test all of the possible truth assignments to the variables, and check if one of them makes the formula true. While this procedure certainly is correct, it cannot be considered efficient: for a formula in which n variables appear, there are 2^n many possible truth assignments to the variables which need to be tested. Assuming that a computer can test 1.000.000.000 assignments per second, this would mean that for a formula with 1000 variables, the algorithm would roughly take $3 \cdot 10^{284}$ years to check all possible assignments. Since formulas of this length do appear

in practical settings, this obviously is not satisfactory. However, there are no known algorithms which solve the problem significantly faster. The P-NP-problem can be stated as the question if there is an algorithm which can do significantly better, i.e., perform only a polynomial number of computation steps instead of the exponentially many tests needed for the complete search algorithm described above.

Although we do not know how to solve this problem efficiently, for a satisfiable formula, it is easy to "prove" that it is indeed satisfiable, by simply giving a satisfying assignment, as we did above for the formula φ_1. There are many problems which share this characteristic. For another example, consider the *Traveling Salesman Problem*. Here we are given a set of cities, a table of plane ticket costs for each city-to-city connection, and a number c. Our task is to determine if there is a round-trip which costs at most c Euros. Again, given such a round-trip, it is easy to check if it satisfies the cost bound. But it seems to be difficult to answer the question if such a trip exists. This property gives a characterization of the problems in the complexity class NP: we do not know how to solve them efficiently, but there are short and easily verifiable "proofs" to show that the answer to such a question is "yes."

The satisfiability problem was the first problem proven to be NP-complete, by Stephen Cook in [Coo71], and, independently, by Levin (a partial English translation of his result can be found in [Tra84]). From that starting point on, literally thousands of problems were proven to fall into this class, and entire books are devoted to proving these kinds of results [GJ79]. The search for NP-complete problems is motivated by two main reasons. First, if there is one problem of these which can be solved in polynomial time, then this immediately gives efficient algorithms for all the NP-complete problems. Therefore, it was hoped that if enough NP-complete problems were known, then there would be discovered some problem which is both NP-complete and efficiently solvable, thus proving that P=NP, and giving efficient algorithms for a vast number of practically relevant problems. However, this has not happened, and in fact, most researchers now believe that it never will, since there probably simply are no efficient algorithms for NP-complete problems. But the search for NP-complete problems still remains interesting: when analyzing the complexity of a problem occurring in practice, in order to prove that it is NP-complete, it is useful to have a problem as "similar as possible" to it for which completeness is known. If for a practical problem we know that it is NP-complete, then we know that with known algorithms and techniques, we cannot obtain an efficient solution, and we need to consider approximation algorithms. Therefore, knowing many completeness results helps to influence decisions in practical software design.

In the above mentioned Traveling Salesman Problem, the goal was to find a strategy of visiting cities. In the satisfiability problem, we search for a strategy to assign truth values to the variables. In both examples, there was no opponent we needed to take into account. When we add possible opponents, and study problems in a game-theoretic setting, then often problems which cannot be solved in NP anymore occur. Consider the following "game:" We are given a propositional formula φ, where the occurring variables are $x_1, \ldots, x_n$. Player A starts to assign a value to the first variable, x_1. Then player B may choose a value for the variable x_2, then it is A's turn again and he determines the value for x_3, and so on, until every variable has been assigned a value. Player A (the *universal player*) wins if the formula φ is false under this assignment, and player B (the

existential player) wins if the formula is true. The question if player A or B have a winning strategy in this game does not seem to be solvable in NP, because unlike with short and easily-checkable assignments for a formula, there does not seem to be short way to encode the strategies in this more general setting. The obvious approach would be to write down every possible "reply" to the other player's moves, but it is easy to see that this results in a table with exponentially many entries in the number of rounds of the game. Problems like these therefore lie in higher complexity classes - the classes arising here are those forming the *polynomial hierarchy*, and the class PSPACE. The example just discussed can be phrased as the validity problem for a *quantified Boolean formula*, where the variables controlled by A are quantified with $\forall$, and the variables controlled by B are quantified with $\exists$.

As mentioned, the class P is considered as the class of problems which can be solved efficiently. It is obvious that there are different "degrees" of efficiency, and hence it is natural to study complexity classes below P. To this end, alternative models of computation were introduced, allowing to obtain results on questions of efficient parallel algorithms, and algorithms with low space usage. Both extensions of the Turing machine and different, circuit-based models were introduced, which allow natural definitions of various complexity classes inside P. Similarly to the NP-complete problems, the notion of completeness for these classes was introduced to describe problems which are "among the hardest" in them. Again, it turned out that satisfiability problems related to restricted classes of Boolean formulas are typical examples for complete problems of these complexity classes. Therefore, a systematic study of these restricted satisfiability problems is of interest, to gain insight into those complexity classes with deep connections to Boolean formulas.

There are two different systematic ways of phrasing the restrictions of propositional formulas that we consider in this thesis. A propositional formula is usually defined to be built of propositional variables, constants, and the operators $\wedge, \vee$, and $\neg$, representing conjunction, disjunction, and negation. What happens if we remove one of them? It is obvious that removing either $\wedge$ or $\vee$ does not reduce the expressive power of the formulas, since we can simulate one of them using the other and negation: $x \wedge y$ is equivalent to $\overline{(\overline{x} \vee \overline{y})}$, and analogously $x \vee y$ can be expressed as $\overline{(\overline{x} \wedge \overline{y})}$. But what if we forbid negation? It is easy to see that the satisfiability problem for negation-free formulas is much simpler than the one for arbitrary formulas: We can simply set every variable occurring in a given formula to 1, and if this assignment does not satisfy the formula, then no assignment will. This problem is not only solvable by an efficient, i.e., a polynomial-time algorithm, but there are efficient parallel algorithms for this problem, as we will see as an easy corollary from the results in Chapter 2. But what about other possible operators, like implication? Or the binary exclusive-or? In fact, we can introduce any Boolean function as an operator allowed in propositional formulas. For each possible set of Boolean functions, this gives a restriction of formulas: the class of formulas built using variables and these connectors. Therefore, we can define an infinite number of possible restrictions in this way, and for each of these restrictions, we obtain a new version of the satisfiability problem, each with a potentially different complexity.

To consider an infinite set of problems, we need some structure on this set. One of the most important results in the classification of the expressive power of these restricted

formulas was Emil Post's work regarding certain "closed classes of Boolean functions." He proved his results already in the 1920s, but his work was not published until 1941, in [Pos41]. His results identify all the "classes of expressiveness" which can be generated by Boolean formulas restricted in this way, and therefore allow for a systematic study of restrictions of propositional formulas by limiting, or extending, the possible operators in the way suggested above. His classification is now known as *Post's lattice*.

One of the first known results applying Post's work to complexity theory and the study of satisfiability problems was achieved by Harry Lewis in [Lew79], where he examined the question which operators make the satisfiability problem NP-complete, and which combinations give efficient algorithms. In particular, he showed that this problem is "dichotomic:" the complexity degrees between NP-completeness and solvable in polynomial time mentioned above do not appear here. Dichotomy results are very interesting in complexity theory: for one, they show that the infinite class of problems in question breaks down to finitely many, if we are only interested in their "complexity behaviors." In this way, it is shown that an infinite class of problems can be considered "the same" from a computational point of view. Also, in many cases a dichotomy theorem demonstrates the exact point where the problem gets difficult, and can therefore give a precise description of the features which make the problems in question hard. Lewis' work gives a precise answer what kind of operators used in formulas make the problem "easy," and which make them NP-complete.

Another restriction is to remove one of the most important features from Boolean formulas, which is nesting. A usual Boolean formula can be nested to any degree. By only considering formulas in *conjunctive normal form*, the nesting degree is reduced to a constant. These are formulas of the form $C_1 \wedge \cdots \wedge C_n$, where the "clauses" C_i must be of a very simple and regular form. It turns out that if we allow arbitrary clauses with up to three variables, the satisfiability problem for these formulas is still NP-complete. If we restrict the number of variables appearing in each clause to 2, then the problem is solvable in nondeterministic logarithmic space, which is a subclass of P. But there are other possible restrictions on these clauses than just limiting the number of variables allowed to occur. A systematic study of these restrictions is known as the *constraint satisfaction problem*. In its non-uniform version, this problem studies so-called Γ-formulas, where the appearing clauses must take the form of some "templates" defined in a set Γ. For the Boolean case, Thomas Schaefer showed in [Sch78], that again, the problem is dichotomic: such a problem either can be solved in polynomial time, or is NP-complete. Surprisingly, this result can be proven by again applying Post's lattice mentioned above. Post's classification is used indirectly here, with an interesting "*Galois connection*" between Boolean functions and closure properties of the clauses allowed in the language Γ. It can be shown that both of these restrictions can be phrased in an algebraic context, and the lattices of closed sets that appear in both cases are dually isomorphic. This means that Post's analysis of the closed classes of Boolean functions also gives us a complete list of cases to study in the constraint satisfaction setting. However, this isomorphism does not seem to allow the direct transfer of complexity results from one of the restrictions to the other.

Constraint satisfaction problems have very interesting theoretical properties, as their dichotomic complexity behavior and the connections to universal algebra. But there also

is a vast number of practical applications. Constraint satisfaction problems generalize not only many well-studied cases of the satisfiability problem, but can be used to express almost any combinatorial problem which can be phrased as a set of local conditions. For example, constraint satisfaction problems play a role in database theory, electronic design automation, scheduling problems, and many other computational settings. On the theoretical side, they generalize problems like graph colorings, graph search, various flavors of satisfiability problems, and many more. Therefore, constraint satisfaction problems can be seen as the "combinatorial core of complexity theory" [CKS01], and hence learning about constraint formulas gives us better insight into many of complexity theory's questions.

In this thesis, we study various forms and generalizations of the satisfiability problem, which using the systematic restrictions explained above. In addition to the satisfiability problem itself, we also consider the closely related problems of model checking, enumeration, counting, and equivalence. The structure of the work is as follows: After recalling prerequisites from the literature and proving some initial results about formulas and relations of our own in Chapter 1, we start with considering formula restrictions in the Post sense. One of the simplest possible questions which can be asked in this context is the problem to determine if a given formula in which no variable appears is true. This problem, called the *formula value problem*, can be seen as the most basic satisfiability problem, where no assignment to the variables has to be considered, but a formula simply has to be evaluated. This task is one of the most important ones arising in algorithms dealing with propositional formulas. It turns out that this problem has efficient parallel algorithms for all types of formulas that we consider, and again we show that if we restrict the propositional operators appearing in the formula, the complexity of the problem decreases even further. Using Post's lattice, we show that there is a finite number of complexity classes such that for any choice of propositional operators, the formula value problem is complete for one of these classes. While this is not a "dichotomy" in the strictest sense, since there are more than two complexity cases arising here, it still shares the properties of dichotomy results which make them so interesting: the complexity of an infinite set of problems can be shown to only give complexities from a finite list.

In practice, knowing the actual solutions to a problem is often more interesting than simply knowing whether at least one solution exists. In Chapter 3, we therefore turn our attention to the problem of computing the set of satisfying assignments for a given propositional formula. This is not a decision problem like the ones mentioned up to now, where the answer to the question is simply "yes" or "no," but a problem where the task is to generate a set of assignments for a given formula. Hence, these problem cannot be grouped into the usual complexity classes of decision problems, like P or NP. Instead, we consider several notions of "efficient enumeration" suggested by David Johnson, Christos Papadimitriou, and Mihalis Yannakakis in [JPY88]. Assuming $P \neq NP$, for each possible restriction of propositional formulas, and each of the efficiency notions considered, we answer the question if such an algorithm exists.

In the remainder of the thesis, we study problems for formulas restricted in the constraint satisfaction context. In Chapter 4, we refine Schaefer's dichotomy theorem for formulas in conjunctive normal form, and consider the subclasses of polynomial time. It turns out that the Galois connection mentioned before has its limitations here: there are

cases which have the same algebraic behavior, but lead to different degrees of complexity. Hence we need to go beyond the classification provided by the algebraic properties, and perform a finer analysis of the cases. It turns out that the problem still is dichotomic in nature, revealing that each of these problems is equivalent to the standard "complete" problems of standard complexity classes inside P. Finally, in Chapter 5, we consider quantified constraint formulas. These are generalizations of the usual constraint formulas, where additionally the quantifiers $\exists$ and $\forall$ are allowed to occur. As hinted above, such formulas can be used to describe settings where two opponents are working against each other. It is well-known that adding these quantifiers to the formulas raises the complexity of the involved decision problems significantly: the problems we consider in this chapter are prototypical for the classes of the polynomial hierarchy, and for the class PSPACE, containing all computational problems which can be solved in polynomial space. We study various problems for these formulas: first, we consider the formula evaluation problem in this context, and the closely related model checking problem. Another decision problem which is very interesting is the equivalence problem, where we ask if two formulas have the same set of satisfying assignments. This question is very important in practice, since it can be used to decide whether two given database queries are equivalent, if a program behaves as its specification demands, or if two games have the same winning strategies.

Finally, we consider the *counting problem* for these formulas, which is the task to determine the number of satisfying assignments for a given formula. This problem arises in practice when we want to determine the number of elements in a database which match a given query. To study the complexity of these problems, counting complexity classes have been introduced, which have a close relationship to the classes of decision problems.

In all of the problems considered in this thesis, we show dichotomy-like results, showing that for an infinite set of problems, only a finite set of complexity classes arises, and the problems turn out to be complete for these classes. Hence, among adding to the list of complete problems for all kinds of classes, we show that all of these infinite classes of problems break down into finitely many complexity cases. Therefore, from a computational point of view, there are only finitely many different problems in this context. For the problems considered in Chapter 4, this can be made even stricter, as in fact we can show that the problems only give rise to finitely many equivalence classes under the much stronger notion of isomorphism.

Publications

The material about bases for co-clones, i.e., the results presented in Table 1.2, the surrounding discussion, and Lemma 1.5.6 previously appeared in [BRSV05]. Lemma 1.4.5 appeared in [Sch05], on which Chapter 2 is based. Chapter 4 previously appeared as [ABI+05], and the results on counting and the $\mathsf{QCSP}_k(\Gamma)$-problem from Chapter 5 appeared in the technical report [BBC+05]. Further results in that chapter contain unpublished work with Michael Bauland, Nadia Creignou, and Heribert Vollmer. The results from Chapter 3 are new.

Chapter 1

Preliminaries

In this chapter, we introduce the basic terminology for the topics that this thesis deals with, cite results from the literature about complexity theory and propositional formulas, and prove some initial results of our own which will be useful in later chapters.

1.1 Basic Notation and Mathematical Prerequisites

We assume that the reader knows the basic definitions and results from theoretical computer science, i.e., the concept of a Turing machine, a finite state machine, O-notation, etc. We also assume that the reader is familiar with basic mathematical structures, like graphs, lattices, partial orders, permutations, fields, and standard mathematical notation, like propositional and first-order logic. We also rely on basic concepts from "naive set theory," i.e., finite, countable, and uncountable sets.

In this work, 0 is a natural number. The symbol $\circ$ denotes the concatenation of strings and of functions. For sets A, B, the set $A + B$ is their disjoint union. For a Boolean value α, $\overline{\alpha}$ denotes its negation. For a function $f\colon A \to \{0,1\}$, the function $\overline{f}$ denotes its component-wise negation, i.e., $\overline{f}(\alpha) = \overline{f(\alpha)}$. With $\vee^k$, we denote k-ary disjunction, we use the symbol $\wedge^k$ for k-conjunction, and $\neg$ for negation. The binary exclusive-or is denoted with $\oplus$. As usual, we will consider decision problems as languages, i.e., we identify the problem to compute a "yes" or "no" answer with the problem to recognize the set of strings where the answer is "yes." In particular, our problems are sets $A \subseteq \Sigma^*$, where Σ^* denotes the set of all finite strings over some finite alphabet Σ.

A n-ary *Boolean function* is a function $f\colon \{0,1\}^n \to \{0,1\}$ for some $n \in \mathbb{N}$. For a set A, we denote its cardinality with $|A|$. We also use the notation $\#A$, when we talk about counting problems. For a string w, and strings $w_1, \ldots, w_n, z_1, \ldots, z_n$, the string $w[w_1/z_1, \ldots, w_n/z_n]$ is obtained from w by simultaneously replacing every occurrence of w_i with z_i for all relevant i.

1.2 Formulas and Circuits

The main subject of this thesis is the complexity of problems related to propositional formulas. Therefore, we will first define formulas, and their generalizations as circuits.

These concepts are interesting for two reasons: First, the problems that we study in this thesis are problems involving these structures, and second, the concept of circuit is essential for the definition of several relevant complexity classes. Usually, propositional formulas are defined as formulas where variables take values from the domain $\{0,1\}$ ("false" and "true"), and the formulas are built from variables, conjunction, disjunction, and negation. In our context, we allow the domain to be an arbitrary finite set D, and we also consider more general sets of allowed connectors than the set $\{\wedge, \vee, \neg\}$. For a set B of functions with finite arity on a domain D, we define *B-formulas* inductively: A variable x is a B-formula. If $\varphi_1, \ldots, \varphi_n$ are B-formulas, and f is an n-ary function from B, then $f(\varphi_1, \ldots, \varphi_n)$ is a B-formula. We often identify the function f and the symbol representing it in a formula. The meaning should always be clear from the context. We denote the set of occurring variables in a formula φ with $\mathrm{VAR}(\varphi)$. We often write $\varphi(x_1, \ldots, x_n)$ to indicate that $\mathrm{VAR}(\varphi) = \{x_1, \ldots, x_n\}$. For an assignment of values $I\colon \mathrm{VAR}(\varphi) \to D$, the *value of φ for the assignment I*, $\varphi(I)$, is defined as follows: If φ is the variable x, then $\varphi(I) = I(x)$. If φ is of the form $f(\varphi_1, \ldots, \varphi_n)$, then $\varphi(I)$ is defined as $f(\varphi_1(I), \ldots, \varphi_n(I))$. If $\mathrm{VAR}(\varphi) = \{x_1, \ldots, x_n\}$, or if there is some other canonical order on the variables of φ, then we say that φ represents the function f defined as $f(\alpha_1, \ldots, \alpha_n) = \varphi(I)$, where $I(x_i)$ is defined as α_i. We say that two formulas φ_1 and φ_2 are *equivalent*, if $\varphi_1(I) = \varphi_2(I)$ for all assignments I. In the case of Boolean formulas, i.e., if $D = \{0,1\}$, the assignments I are also called *truth assignments*. In this case, we also write $I \models \varphi$ if $\varphi(I) = 1$, and we say that I *satisfies* φ, or that φ is a *solution* of φ. The set of solutions of φ is denoted with $\mathrm{SOL}(\varphi)$. When no other domain is specified, we always assume D to be the Boolean domain $\{0,1\}$.

A formula can be represented as a graph, and the occurring structure is a tree. In generalizing formulas to circuits, we allow arbitrary acyclic graphs in this representation. The *Boolean circuit* is relevant for us for two reasons: For one, Boolean circuits present a way to encode Boolean functions succinctly. Second, as a computation model, the Boolean circuit can solve computational problems, and its power is used to define standard complexity classes. The implementation of an algorithm as a Boolean circuit is also a model suited very well for the study of hardware implementations. We will now define this formally. The following definition is based on Definition 1.6 in [Vol99].

Definition Let B be a set of Boolean functions. A *Boolean circuit* over B, or a *B-circuit with n input gates and m output gates* is a tuple

$$C = (V, E, \alpha, \beta, o_1, \ldots, o_m),$$

where (V, E) is a finite, acyclic, directed graph, $\alpha\colon E \to \mathbb{N}$ is an injective function, $\beta\colon V \to B \cup \{x_1, \ldots, x_n\}$, and $o_1, \ldots, o_m \in V$, such that the following conditions hold:

- If $v \in V$ has in-degree 0, then $\beta(v) \in \{x_1, \ldots, x_n\}$, or $\beta(v)$ is a 0-ary function from B,
- if $v \in V$ has in-degree $k > 0$, then $\beta(v)$ is a k-ary function in B.

Nodes in V are also called *gates*. A gate v with $\beta(v) \in \{x_1, \ldots, x_n\}$ is called an *input-gate*. The gates $o_1, \ldots, o_m$ are also called *output-gates*.

This definition of a Boolean circuit corresponds to the intuitive idea that a circuit consists of a set of gates which are either input gates, or compute some Boolean function (in our case, functions from B) with arguments taken from the predecessor gates. The set B is also called a *base*. The distinguished gates $o_1, \ldots, o_m$ are the output-gates, i.e., the value computed by the circuit is obtained by concatenating the results computed in these gates. The *size* of a circuit is the number of non-input gates, and the *depth* is the length of a longest path from an input- to an output-gate. The function computed by a circuit is defined in the canonical way: Once we know the values for the input-gates, we can inductively (since the graph is acyclic) compute the value for each gate $g \in V$. For non-commutative functions in B, the ordering α on the edges in the graph gives a well-defined function value. We formalize this concept in the following definition:

Definition Let $C = (V, E, \alpha, \beta, o_1, \ldots, o_m)$ be a Boolean circuit with n input gates and m output gates, and let $\alpha_1, \ldots, \alpha_n \in \{0,1\}$. Let v be a gate in C. We define the function f_v computed by the gate v on input $(\alpha_1, \ldots, \alpha_n)$ as follows:

- If v is an input-gate, i.e., $\beta(v) = x_i$ for $i \in \{1, \ldots, n\}$, we define $f_v(\alpha_1, \ldots, \alpha_n) =_{\text{def}} \alpha_i$.
- If v has in-degree 0, but is not an input-gate, then $f_v(\alpha_1, \ldots, \alpha_n) =_{\text{def}} \beta(v)$ (which in this case must be a constant).
- If v has in-degree k, and $v_1, \ldots, v_k$ are the predecessor gates of v in C such that $\alpha((v_1, v)) < \cdots < \alpha((v_k, v))$, then

$$f_v(\alpha_1, \ldots, \alpha_n) =_{\text{def}} \beta(v)(f_{v_1}(\alpha_1, \ldots, \alpha_n), \ldots, f_{v_k}(\alpha_1, \ldots, \alpha_n)).$$

We define the function $f_C \colon \{0,1\}^n \to \{0,1\}^m$, the *function computed by* C, as the bit-string $f_{o_1}(\alpha_1, \ldots, \alpha_n) \ldots f_{o_m}(\alpha_1, \ldots, \alpha_n)$.

Unlike a Turing machine or even a finite state machine, a given circuit, due to its limited number of input gates, only can compute values for a finite set of input combinations. In order for circuits to decide infinite languages, or compute functions with infinite domains, we consider circuit families. A *B-circuit family* $\mathcal{C}$ is a sequence of B-circuits $(C_n)_{n \in \mathbb{N}}$, such that for each $n \in \mathbb{N}$, C_n is a B-circuit with n input gates. We say that such a family computes the function $f \colon \{0,1\}^* \to \{0,1\}^*$, if for each $w \in \{0,1\}^*$, the value of $C_{|w|}$ on input w is $f(w)$. Such a family decides the language L, if it computes the characteristic function f_L of L, i.e., the function for which $f_L(w) = 1$ if $w \in L$, and $f_L(w) = 0$ otherwise.

Note that, as mentioned above, there is a close relationship between circuits and formulas: a formula can be seen as a tree-like circuit. Such circuits can be written as an equivalent formula, and the representation does not grow significantly in size. For a general circuit, the length of its "formula representation" can be exponential in the size of the original circuit.

1.3 Complexity Theory

We now explain the basic definitions and important results of complexity theory which play a role in this thesis.

1.3.1 Complexity classes

Following notation from [BDG95] and [BDG90], we introduce complexity classes in a standard way. Our model of computation is the Turing machine, although most of our results do not use this model explicitly.

Let $f\colon \mathbb{N} \to \mathbb{N}$ be a function. Then the class $\mathrm{DTIME}(f)$ ($\mathrm{DSPACE}(f)$) contains all problems which can be solved by a deterministic Turing machine in time (space) $O(f)$. Similarly, the class $\mathrm{NTIME}(f)$ ($\mathrm{NSPACE}(f)$) consists of problems or languages solvable by a nondeterministic Turing machine in time (space) $O(f)$. In order for our complexity classes to define sets in the usual mathematical sense, we need to restrict the languages occurring here. For convenience, the following restriction is usually made: it is obvious that any finite alphabet $\{q_0, \ldots, q_k\}$ can be represented by the binary notations of the numbers $0, \ldots, k$. It is easy to find, for any given natural problem, a representation over this set, which preserves the complexity of the original problem. Hence, we assume that all our languages are subsets of $\{0,1\}^*$.

Some of the most important complexity classes, which also play a role in the classifications arising in this work are:

$$\begin{aligned}
\mathrm{LOGSPACE} &=_{\mathrm{def}} \mathrm{DSPACE}(\log n), \\
\mathrm{NL} &=_{\mathrm{def}} \mathrm{NSPACE}(\log n), \\
\mathrm{P} &=_{\mathrm{def}} \bigcup_{k\in\mathbb{N}} \mathrm{DTIME}\left(n^k\right), \\
\mathrm{NP} &=_{\mathrm{def}} \bigcup_{k\in\mathbb{N}} \mathrm{NTIME}\left(n^k\right), \\
\mathrm{PSPACE} &=_{\mathrm{def}} \bigcup_{k\in\mathbb{N}} \mathrm{DSPACE}\left(n^k\right) = \bigcup_{k\in\mathbb{N}} \mathrm{NSPACE}\left(n^k\right) \quad [\mathrm{Sav70}].
\end{aligned}$$

The class $\oplus$LOGSPACE is defined to be the class of languages L for which there is a non-deterministic Turing machine M operating in logarithmic space, and for any word w, it holds that w is a member of L if and only if the number of accepting paths of M on input w is odd. For these classes, the following containments hold (see also Figure 1.1):

$$\mathrm{LOGSPACE} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE}.$$

It is also known that LOGSPACE $\neq$ PSPACE, but for the other inclusions, the question whether they are proper remains open (though most researchers believe that *all* of them are). In particular, the question if P = NP is probably the most important open problem in complexity theory. Intuitively, the class NP contains languages L which have the following property: if some word w is a member of L, then there is a short "proof"

for this fact, which can be verified in polynomial time. The proof is just an encoding of the accepting computation path of a given NP-machine deciding the language L. In a similar way, there are languages where there are short proofs for the fact that some word is *not* in the language. This leads to a more general concept, which can be described as follows:

Definition Let $\mathcal{C}$ be a complexity class of languages over the alphabet $\{0,1\}$. Then the class co$\mathcal{C}$ is defined as follows:

$$\text{co}\mathcal{C} =_{\text{def}} \{\{0,1\}^* \setminus L \mid L \in \mathcal{C}\}.$$

For complexity classes $\mathcal{C}$ which are defined by deterministic computation models, like deterministic Turing machines or circuits using a standard set of gates, it is easy to see that $\mathcal{C} = \text{co}\mathcal{C}$ holds: a given machine or circuit deciding the language L can be turned into one deciding the language $\{0,1\}^* \setminus L$ by simply exchanging accepting and rejecting states in the Turing machine, or by computing a single negation at the output gate of a circuit. However, for non-deterministic classes, this is not so easy. Natural examples for this are the classes NP and coNP. A prominent problem for the class NP is the satisfiability problem for propositional formulas. As explained in the introduction, this language contains all propositional formulas using variables, conjunction and negation, for which there is an assignment to the variables which makes it true. The problem is in NP, since a satisfying assignment can be guessed. This assignment also serves as the "proof" in the sense explained above. Now the "complement problem" (aside from syntactical correctness) is the problem to recognize the unsatisfiable formulas. This problem is in coNP, since a "proof" for a formula to be *not* unsatisfiable is, again, a satisfying solution. There does not seem to be a natural short proof for the unsatisfiability of a formula, and in fact, this problem is not believed to be in NP. The question whether NP and coNP are equal is a major open question in complexity theory. However, for space-bounded classes, an analogous result can be shown. The following theorem was proven independently by Neil Immerman [Imm88] and Robert Szelepcsényi [Sze88].

Theorem 1.3.1 ([Imm88, Sze88]) *Let* $f\colon \mathbb{N} \to \mathbb{N}$ *be a function such that* $f(n) \geq \log(n)$ *for all* $n \in \mathbb{N}$. *Then* $\text{NSPACE}(f) = \text{coNSPACE}(f)$.

In particular, the theorem implies that NLOGSPACE=coNLOGSPACE. We also say that these classes are *closed under complementation*, because for each language L in the complexity class, the "complement language" $\{0,1\}^* \setminus L$ is also a member of the class.

An often-used extension of the Turing machine is the notion of an *oracle Turing machine*. Such a machine is an ordinary Turing machine which has access to an "oracle," which means that it gets answers to certain questions "for free." An oracle Turing machine has a special query tape, on which it has write-only access. It also has three special states, donated with $q_?, q_+$, and q_-.

Let A be an arbitrary language, the "oracle language." We define the operation of an oracle machine M^A with access to the oracle A. On states $q \notin \{q_?, q_+, q_-\}$, M^A behaves

like an ordinary Turing machine. When M^A enters the state $q_?$, its next state is q_+, if the word on the query tape is in the language A, and it is q_- otherwise. The query tape is erased after such an operation. This transition counts as one step in the running time of the machine, therefore this concept captures the idea that M gets answers to questions of the form "does $w \in A$ hold" at no cost. Oracles can be used to consider questions like "what happens if we ignore the complexity for the problem A?" In giving the machines access to A via an oracle, we essentially disregard that it takes computational power to decide A.

This notion of oracles can be used to define complexity classes. For some complexity class $\mathcal{C}$ which is defined by a restriction of Turing machines and a language A, the class $\mathcal{C}^A$ denotes the set of languages which can be decided by a $\mathcal{C}$-machine with access to the oracle A. For some set S of problems, we define

$$\mathcal{C}^S =_{\text{def}} \bigcup_{A \in S} \mathcal{C}^S.$$

We can now define the classes of the polynomial time hierarchy, as introduced by Larry Stockmeyer in [Sto77]:

$$\begin{aligned} \Sigma_0^p =_{\text{def}} \Pi_0^p =_{\text{def}} \Delta_0^p &=_{\text{def}} \mathrm{P}, \\ \Delta_{k+1}^p &=_{\text{def}} \mathrm{P}^{\Sigma_k^p}, \\ \Sigma_{k+1}^p &=_{\text{def}} \mathrm{NP}^{\Sigma_k^p}, \\ \Pi_{k+1}^p &=_{\text{def}} \mathrm{co}\Sigma_{k+1}^p. \end{aligned}$$

These classes are called the classes of the *polynomial hierarchy*, which is denoted with PH and consists of the union of all the classes defined above. It can easily be shown that PH is a subset of PSPACE. The class Σ_1^p is the class NP, and Π_1^p is equal to coNP. For any $k \in \mathbb{N}$, the following inclusions hold:

$$\mathrm{P} \subseteq \Delta_k^p \subseteq \Sigma_k^p \subseteq \Delta_{k+1}^p \subseteq \Pi_{k+1}^p \subseteq \Delta_{k+2}^p.$$

Obviously, the classes Δ_k^p are closed under complementation for any $k \in \mathbb{N}$. The classes Σ_k^p are not believed to be closed under complementation. It is also believed that the inclusions above are proper, but this has not been proven. In particular, if $\mathrm{P} = \mathrm{NP}$, then all of these classes are identical to P. The classes of the polynomial hierarchy and other complexity classes arising in this thesis are presented in Figure 1.1. The classes $\Delta_0^{\mathcal{R}}$, $\Sigma_1^{\mathcal{R}}$, and $\Pi_1^{\mathcal{R}}$ which also appear in the figure are introduced in Chapter 2.

As explained above, problems in the class NP can be characterized as search problems, where the question is if some "solution" with a certain property, which easily can be verified, exists. Problems in coNP can be phrased similarly: here we basically ask if a "solution" does not exist. Equivalently, this can be stated as "all strings are *no* solution." In this way, we ask if all possible members from a given set have some easily verifiable property. By this informal argument, it is natural that the class NP is related to questions which can be phrased with an existential quantifier, and the class coNP similarly deals with questions which can be stated using a universal quantifier.

A canonical generalization of this observation to the Σ_k^p and Π_k^p-classes of the polynomial hierarchy can be found in any textbook about complexity theory. Following the

presentation from [Pap94], we say that a $(k+1)$-ary relation over $\{0,1\}^*$ is *polynomially balanced*, if there is some $l \in \mathbb{N}$ such that $(x, y_1, \ldots, y_k) \in R$ implies $|y_1|, \ldots, |y_k| \leq |x|^l$. Now the classes Σ_k^p and Π_k^p from the polynomial hierarchy can be characterized as follows:

Theorem 1.3.2 ([Wra77]) *Let L be a language, and let $k \geq 1$.*

1. *$L \in \Sigma_k^p$ if and only if there is a polynomial-time decidable, polynomially balanced $(k+1)$-ary relation R such that the following holds:*

$$L = \{x \mid \exists y_1 \forall y_2 \exists y_3 \ldots Q y_k (x, y_1, \ldots, y_k) \in R\},$$

   where Q is $\forall$ if k is even, and Q is $\exists$ if k is odd.

2. *$L \in \Pi_k^p$ if and only if there is a polynomial-time decidable, polynomially balanced $(k+1)$-ary relation R such that the following holds:*

$$L = \{x \mid \forall y_1 \exists y_2 \forall y_3 \ldots Q y_k (x, y_1, \ldots, y_k) \in R\},$$

   where Q is $\exists$ if k is even, and Q is $\forall$ if k is odd.

Note that the quantifiers in the above equations are over a set of strings of polynomial length, and not just a single Boolean value.

Another extension of the Turing machine is the *alternating Turing machine*. This concept was introduced by Ashok Chandra, Dexter Kozen, and Larry Stockmeyer in [CKS81]. A "traditional" nondeterministic Turing machine accepts if and only if there is one nondeterministic computation path which accepts the input. An alternating machine has two possibilities for nondeterministic choices. The first one, called "existential branches," is the same as in the usual nondeterministic model, and accepts if and only if at least one of the nondeterministic choices does. The other possibility is to accept if and only if every of the nondeterministic choices does. For a function $s\colon \mathbb{N} \to \mathbb{N}$, the class $\mathrm{ATIME}(s)$ contains the problems which can be solved by an alternating Turing machine in time s, and similarly, $\mathrm{ASPACE}(s)$ contains the problems solvable on such a machine with space s. In particular, the class AP contains the problems which can be solved on an alternating machine in polynomial time. It is obvious that the existential and universal branches can be used to simulate existential and universal quantifiers in propositional formulas. As we will see later, problems for these formulas are typical examples for problems in PSPACE, and hence it is not surprising that AP = PSPACE. Many interesting relationships between classes defined by alternating machines and those defined by other models of computation can be found in [Coo85].

There are also complexity classes defined using the computational power of circuits. For some set B of Boolean functions, and functions $s, d\colon \mathbb{N} \to \mathbb{N}$, the class $\mathrm{SIZE}-\mathrm{DEPTH}_B(s,d)$ is defined as the set of languages L which can be decided by a family of Boolean B-circuits $\mathcal{C} = (C_n)_{n\in\mathbb{N}}$ where the circuit C_n has size $O(s)$ and depth $O(d)$. Let $\mathcal{B}_0 =_{\mathrm{def}} \{\wedge, \neg\}$, and let $\mathcal{B}_1 =_{\mathrm{def}} \{\neg\} \cup \{\wedge^i \mid i \in \mathbb{N}\}$. It should be noted that infinite bases like $\mathcal{B}_1$ can enhance the computational power of circuits significantly, so we need to be careful with these. Since this does not play a big role in this thesis, we

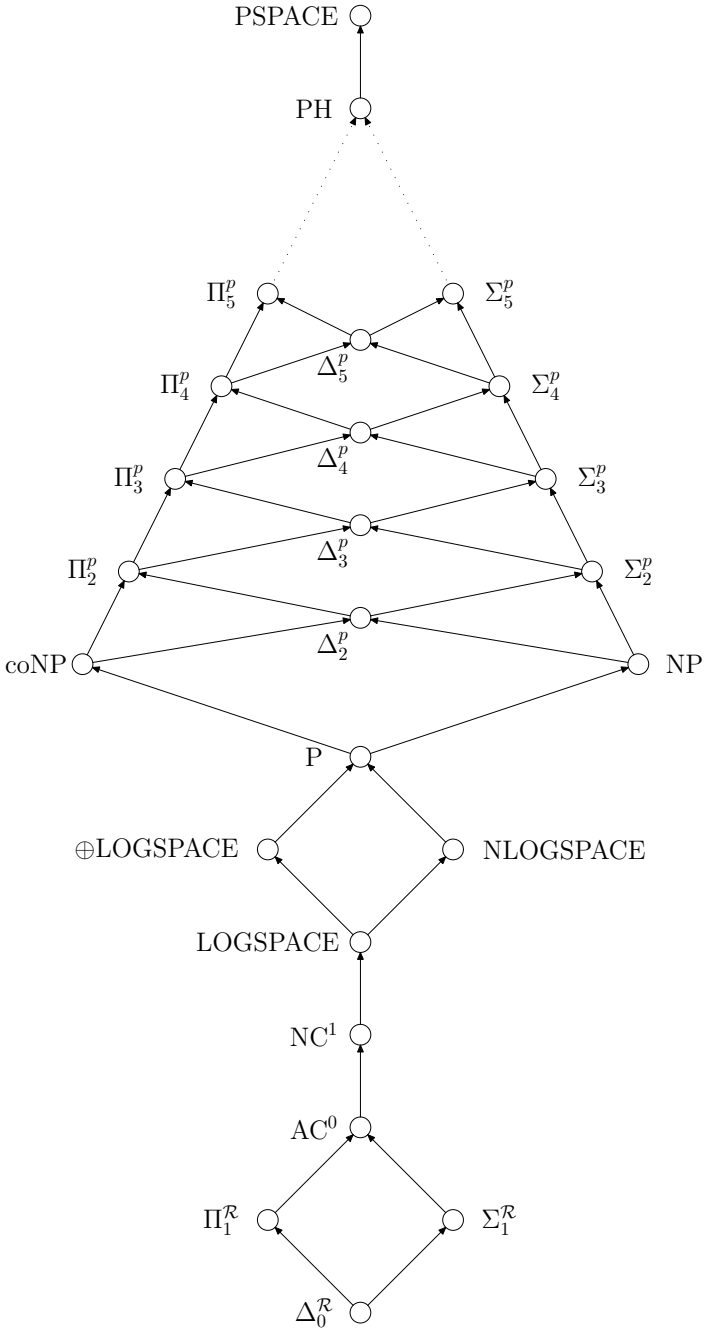

Figure 1.1: The polynomial hierarchy and other relevant complexity classes

only mention that infinite bases need to be "uniform" in some way, as $\mathcal{B}_1$ clearly is. For details and more results on Boolean circuits and their complexity, see [Vol99].

We now define some of the most important circuit complexity classes. Let k be a natural number. The class AC^k was defined by Stephen Cook, Ashok Chandra, Larry Stockmeyer, and Uzi Vishkin [Coo85, CSV84], and NC^k is named after Nicolas Pippenger, who first studied it [Pip79]:

- $\mathrm{AC}^i =_{\mathrm{def}} \bigcup_{k\in\mathbb{N}} \mathrm{SIZE-DEPTH}_{\mathcal{B}_1}\left(n^k, (\log n)^i\right)$,
- $\mathrm{NC}^i =_{\mathrm{def}} \bigcup_{k\in\mathbb{N}} \mathrm{SIZE-DEPTH}_{\mathcal{B}_0}\left(n^k, (\log n)^i\right)$.

It can easily be seen that for any $i \in \mathbb{N}$, it holds that $\mathrm{AC}^i \subseteq \mathrm{NC}^{i+1} \subseteq \mathrm{AC}^{i+1}$. Additionally, it is known that AC^0 is a proper subclass of NC^1 [Smo87]. The class NC, defined as the union of all NC^k, is often considered to capture the notion of efficient parallel algorithms. It is important to remember that circuit complexity classes display a certain amount of non-uniformity. It is easy, for example, to construct a family $\mathcal{C}$ of small circuits which, on input $w \in \{0,1\}^*$, output 1 if $w = 1^n$ for some n which is a member of the halting problem, and 0 otherwise. Since the halting problem is well-known to be undecidable (first shown by Alan Turing in [Tur36]), the inclusions $\mathrm{AC}^i \subseteq \mathrm{P}$ etc., do not hold. But we still want these complexity classes to capture the notion of "low complexity." To avoid the above-mentioned problem, we demand that our circuit families $\mathcal{C}$ are uniform, meaning that there must be a reasonably efficient algorithm which, on input 1^n, outputs a suitable encoding of C_n. This construction algorithm must be efficient enough to ensure that the construction of the circuit is not more complex than the operation performed by the circuit itself. However, for the purpose of this thesis, the above description of uniformity suffices.

Up to now, we only considered the complexity of decision problems. These can be seen as the computation of functions whose output value is either 0 or 1. However, many computational problems arising in practice are of a different nature, as for example the task to sort a given sequence of numbers. The complexity of these problems can be defined similarly. For alphabets Σ_1 and Σ_2, let $f\colon \Sigma_1^* \to \Sigma_2^*$ be a function, and let $\mathcal{C}$ be a complexity class for decision problems as defined above, where the computation model is a deterministic Turing machine or a circuit. We say that f is in the class $\mathrm{F}\mathcal{C}$, if it can be computed by a Turing machine or circuit with the same resource bounds as required by the class $\mathcal{C}$. Here, a Turing machine computes a function f, if for any input x, the Turing machine stops, and on a designated output-tape of the machine, the word $f(x)$ is written.

For example, the class FP is the class of functions which can be computed by a deterministic Turing machine in polynomial time, and the class FAC^0 is the class of functions which can be computed by circuits over the base $\mathcal{B}_1$ with polynomial size and constant depth. It is also possible to define the function computed by a non-deterministic Turing machine, but we only consider this for the special case of counting problems in Chapter 5.

1.3.2 Reductions

One of the most important tools developed in complexity theory is the concept of the reduction. The idea behind a reduction is that it allows us to compare the complexity of problems. For a reduction $\leq$, the relationship $A \leq B$ is usually meant to capture "A is not harder than B." More precisely, a reduction usually demands that if we can solve the problem B, then we can also, with very little additional resources, solve the problem A. Of course a key question here is what "very little resources" means. The answer to this question depends on the context: If we are talking about the class PSPACE, then a polynomial-time computable function might classify. When considering classes e.g., below LOGSPACE, the resource bound of polynomial time is rather meaningless. Usually, the lower the considered complexity classes are, the finer the required reductions get. We will now introduce some important reductions. For now, we restrict ourselves to decision problems. In Chapter 5, we will see that reductions can also be applied to other types of problems.

For languages $A \subseteq \Sigma_1^*$ and $B \subseteq \Sigma_2^*$, we say a function $f\colon \Sigma_1^* \to \Sigma_2^*$ is a *many-one reduction from A to B*, if for all $x \in \Sigma_1^*$, it holds that $x \in A$ if and only if $f(x) \in B$. We say that $A \leq_m^p B$, if there is a function $f \in \mathrm{FP}$ which is a many-one reduction from A to B. Analogously, we say that $A \leq_m^{\log} B$ ($A \leq_m^{\mathrm{AC}^0} B$) if there is a many-one reduction from A to B which can be computed in FLOGSPACE (FAC^0). The reduction $\leq_m^p$ is also called *polynomial time many-one reduction*, and $\leq_m^{\log}$ is called *logspace many-one reduction*.

For two problems A and B such that $A \leq_m^p B$ and $B \leq_m^p A$ holds, we also write $A \equiv_m^p B$, and say that the problems are *equivalent under polynomial time many-one reductions*. The problems equivalent in this way form a *$\leq_m^p$-degree of complexity*. We use the obvious generalizations for other reducibilities. All the notions of reducibility that we use in this thesis are both reflexive and transitive, and hence form a quasi-ordering on the set of languages over a given alphabet. Languages which are maximal with respect to this quasi-ordering play a special role, as they can be regarded to be among the "most difficult" languages in the corresponding complexity class. For a complexity class $\mathcal{C}$ and a language A, we say that A is *complete for $\mathcal{C}$ under $\leq_m^p$-reductions* if the following holds:

1. $A \in \mathcal{C}$,
2. For all $L \in \mathcal{C}$, it holds that $L \leq_m^p A$.

The second condition is also referred to as A is *hard for $\mathcal{C}$ under $\leq_m^p$-reductions*. For other reduction types like $\leq_m^{\log}$, $\leq_m^{\mathrm{AC}^0}$, etc., analogous notations are defined in the obvious way. As mentioned before, it is important to consider reductions which are suitable for the given complexity class considered. This is evident when we are talking about complete problems. We say that a complexity class $\mathcal{C}$ is *closed* under a reduction $\leq$, if the following holds: for all problems A, B such that $B \in \mathcal{C}$ and $A \leq B$, it follows that $A \in \mathcal{C}$. Now if we consider a complexity class $\mathcal{D} \subseteq \mathcal{C}$ which is closed under $\leq$-reductions, and show that some problem which is complete for $\mathcal{C}$ under $\leq$-reductions already is contained in the class $\mathcal{D}$, then the equality of these two classes follows. It is easy to see that the classes P, NP, and all classes from the polynomial hierarchy are closed under $\leq_m^p$-reductions. Hence, if there is a single problem which is complete for NP under $\leq_m^p$-reductions that can be solved in P, then $\mathrm{P} = \mathrm{NP}$ follows.

1.3.3 Complexity results for specific problems

Complexity theory has achieved very many results on the exact complexity of various problems. We introduce some specific problems and their complexity, as far as they are relevant to our work. The first problem we define now is the satisfiability problem, which already has been mentioned.

Problem:	SAT
Input:	A propositional formula φ using variables, $\wedge, \vee$, and $\neg$
Question:	Is there a truth assignment to the variables such that φ evaluates to true?

As mentioned before, the satisfiability problem was the first ever problem proven to be NP-complete, in a seminal paper by Stephen Cook:

Theorem 1.3.3 ([Coo71]) SAT *is complete for* NP *under* $\leq_m^{\log}$*-reductions.*

This theorem is one of the most famous results in theoretical computer science. Many restricted versions of this problem have been shown to be still NP-complete. One of the most important ones is the problem 3SAT, which we will define now. We say that a propositional formula φ is in 3CNF, if it is a conjunction of clauses, where each clause is the disjunction of exactly 3 literals. The satisfiability problem for these formulas is still NP-complete. This follows from an easy modification of Cook's proof for Theorem 1.3.3.

Problem:	3SAT
Input:	A propositional formula φ in 3CNF
Question:	Is φ satisfiable?

Theorem 1.3.4 ([Coo71]) 3SAT *is complete for* NP *under* $\leq_m^{\log}$*-reductions.*

The next result gives standard complete problems for the important complexity classes LOGSPACE and NLOGSPACE:

Problem:	(directed) Graph Accessibility Problem
Input:	A (directed) graph G, vertices s and t from G
Question:	Is there a path from s to t in G?

The complexity of these problems is summarized in the following Theorem:

Theorem 1.3.5 ([Pap94, Rei05]) *The accessibility problem for undirected graphs is complete for* LOGSPACE *under* $\leq_m^{\mathrm{AC}^0}$*-reductions. The graph accessibility problem for directed graphs is complete for* NLOGSPACE *under* $\leq_m^{\mathrm{AC}^0}$*-reductions.*

This theorem has a long history. It is easy to see that the directed graph accessibility problem can be solved in NLOGSPACE, and the hardness has been known for a long time, it can be shown with an elementary proof. For example, the result can be found as Theorem 16.2 in [Pap94]. Until 2004, it was not known that the undirected version of the problem can be solved in LOGSPACE. Instead, this problem was known to be complete for the class SL, which is the class of problems which can be solved by "symmetric

logspace machines," machines using a restricted form of non-determinism. Omer Reingold proved in [Rei05] that a deterministic logspace algorithm exists to solve this problem, hence showing that the complexity classes SL and LOGSPACE coincide. Since SL is, by definition, a superclass of LOGSPACE, this also immediately gives the hardness result.

1.4 Functions and Relations

For any domain D, and natural numbers k and n, let the n-ary function $\mathrm{id}_D^{k,n}$ be defined as $\mathrm{id}_D^{k,n}(x_1, \ldots, x_n) =_{\mathrm{def}} x_k$. The functions of the form $\mathrm{id}_D^{k,n}$ are called *projections* or *identity functions* on D. We say that some set C of functions over D is a *closed set of functions over D*, or a *clone over D*, if C satisfies the following conditions:

1. C contains all projections on D,
2. If $f_1, \ldots, f_n \in C$ are functions of arity $k_1, \ldots, k_n$, and $f \in C$ is an n-ary function then the function g defined as

$$g(x_1^1, \ldots, x_{k_1}^1, \ldots, x_1^n, \ldots, x_{k_n}^n) =_{\mathrm{def}} f(f_1(x_1^1, \ldots, x_{k_1}^1), \ldots, f_n(x_1^n, \ldots, x_{k_n}^n))$$

   is in C *(composition of functions)*,
3. If f is a k-ary function in C, then the function g defined as $g(x_1, \ldots, x_{k+1}) =_{\mathrm{def}} f(x_1, \ldots, x_k)$ is in C *(introduction of a fictive variable)*,
4. If f is a k-ary function in C, then the function g defined as $g(x_1, \ldots, x_{k-1}) =_{\mathrm{def}} f(x_1, \ldots, x_{k-1}, x_{k-1})$ is in C *(identification of variables)*.

The set of all finitary functions over D satisfies the above condition. Therefore the intersection in the following equation is not empty, and therefore the definition is valid: For any set B of functions on D, we define the *clone generated by B* as

$$[B] =_{\mathrm{def}} \bigcap \{C \mid C \text{ is a clone and } B \subseteq C\}.$$

We say that B is a *base* for $[B]$. It can easily be verified that the set $[B]$ is indeed closed under the above-mentioned transformations, and that the operator [.] is a closure operator, i.e., for all B_1 and B_2, it holds that $B_1 \subseteq [B_1]$, $B_1 \subseteq B_2$ implies $[B_1] \subseteq [B_2]$, and $[[B_1]] = [B_1]$.

For any closure operator C, the set of C-closed sets (i.e., sets B such that $C(B) = B$) forms a lattice: It is easy to see that with the definitions $C(B_1) \sqcap C(B_2) =_{\mathrm{def}} C(B_1) \cap C(B_2)$, and $C(B_1) \sqcup C(B_2) =_{\mathrm{def}} C(B_1 \cup B_2)$, the operators $\sqcap$ and $\sqcup$ fulfill the requirements of the infimum and supremum operations in a lattice where the partial order is set inclusion. Therefore, the set of clones over any domain D forms a lattice. For the Boolean case, i.e., the case where $|D| = 2$, this lattice has been classified by Emil Post in [Pos41]: All Boolean clones and their inclusion structure can be seen in Figure 1.2. A list of the classes and a base for each is given in Table 1.1. We will use the names introduced for the clones in Table 1.1 frequently in this thesis. This lattice is now known as *Post's lattice*. It has several nice properties: for one, it is countable, and the infinite parts of the lattice have a very uniform structure. Further, Post proved that for every

Boolean clone C, there is a finite *base*, i.e., a finite set B such that $[B] = C$. As we will see later, this lattice is a powerful tool for analyzing the complexity of problems related to Boolean formulas. For cases where the cardinality of the domain is larger than 2, it can be shown that the corresponding clone lattice is uncountable, and none of these larger versions have been classified completely. Since they are uncountable, it is obvious that here, almost all of the clones do not have a finite base. The higher complexity of the clone lattice for larger domains is one of the key reasons why classifying the complexity for problems in this context over non-Boolean domains is significantly more difficult.

We define a few properties of functions which play a role in Post's classification. For a function $f\colon D^k \to D$, we say that it does not depend on its i-th argument, if for all $\alpha_1, \ldots, \alpha_k, \beta \in D$, it holds that $f(\alpha_1, \ldots, \alpha_k) = f(\alpha_1, \ldots, \alpha_{i-1}, \beta, \alpha_{i+1}, \ldots, \alpha_k)$. A function which does not depend on all of its arguments is called *degenerated*. We say that f is *essentially unary*, if f depends on exactly one of its arguments, and f is *constant*, if it does not depend on any of its arguments. If the domain D is of the form $\{1, \ldots, n\}$ for some $n \in \mathbb{N}$, then we say that f is *monotone*, if $\alpha_1 \leq \beta_1, \ldots, \alpha_k \leq \beta_k$ implies $f(\alpha_1, \ldots, \alpha_k) \leq f(\beta_1, \ldots, \beta_k)$. For $\alpha \in D$, we say that f is *α-reproducing*, if $f(\alpha, \ldots, \alpha) = \alpha$. For $\alpha \in D$, a set $A \subseteq D^n$ is *α-separating*, if there exists some $i \in \{1, \ldots, n\}$ such that for all $(\alpha_1, \ldots, \alpha_n) \in A$, it holds that $\alpha_i = \alpha$. The function f is *α-separating* if $f^{-1}(\{\alpha\})$ is. We say that f is *α-separating of degree m*, if every subset $A \subseteq f^{-1}(\{\alpha\})$ with $|A| = m$ is α-separating. Finally, a Boolean function f is *linear*, if f can be written as $x_{i_1} \oplus \cdots \oplus x_{i_n} \oplus c$ for some variables $x_{i_1}, \ldots, x_{i_n}$ and a constant $c \in \{0, 1\}$. In addition, the function h_n occurring in the bases of Post's lattice as given in Table 1.1 is defined as

$$h_n(\alpha_1, \ldots, \alpha_{n+1}) =_{\text{def}} \bigvee_{i=1}^{n+1} (\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_{i-1} \wedge \alpha_{i+1} \wedge \cdots \wedge \alpha_{n+1}) .$$

Post's lattice displays a symmetry, which is given by the dualization of Boolean functions: For an n-ary Boolean function f, the function $\text{dual}(f)$ is defined as follows: $\text{dual}(f)(\alpha_1, \ldots, \alpha_n) =_{\text{def}} \overline{f(\overline{\alpha_1}, \ldots, \overline{\alpha_n})}$. A function f is *self-dual*, if $\text{dual}(f) = f$. For a set B of Boolean functions, we define $\text{dual}(B) =_{\text{def}} \{\text{dual}(f) \mid f \in B\}$. It is obvious that $\text{dual}(\text{dual}(f)) = f$, and $\text{dual}(\text{dual}(B)) = B$. In Post's lattice, the $\text{dual}(.)$-operator gives the symmetry in Figure 1.2: for the clone B, the set $\text{dual}(B)$ also is a clone, and is the mirror class of B in Post's lattice. Hence, the clones on the vertical symmetry axis are exactly those clones which are closed under dualization. The clones B and $\text{dual}(B)$ have very similar properties, and it often turns out that for a complexity classification, only one side of Post's lattice needs to be considered.

We now explain how the closure operator [.] is related to the complexity of problems defined by certain classes of formulas. For many problems in the context of propositional formulas, the complexity gets significantly lower if we look at a restricted class of formulas. For example, the satisfiability problem for propositional Boolean formulas is NP-complete due to Theorem 1.3.3. However, if we look at the set $B =_{\text{def}} \{\wedge, \vee, 0, 1\}$ and the corresponding B-formulas, then we get an easier problem: such a formula describes a monotone function. It can easily be seen that such a formula is satisfiable if and only if it is satisfied by the constant 1-assignment. This test surely can be performed in polynomial time. This example shows that restricting the class of formulas in this way,

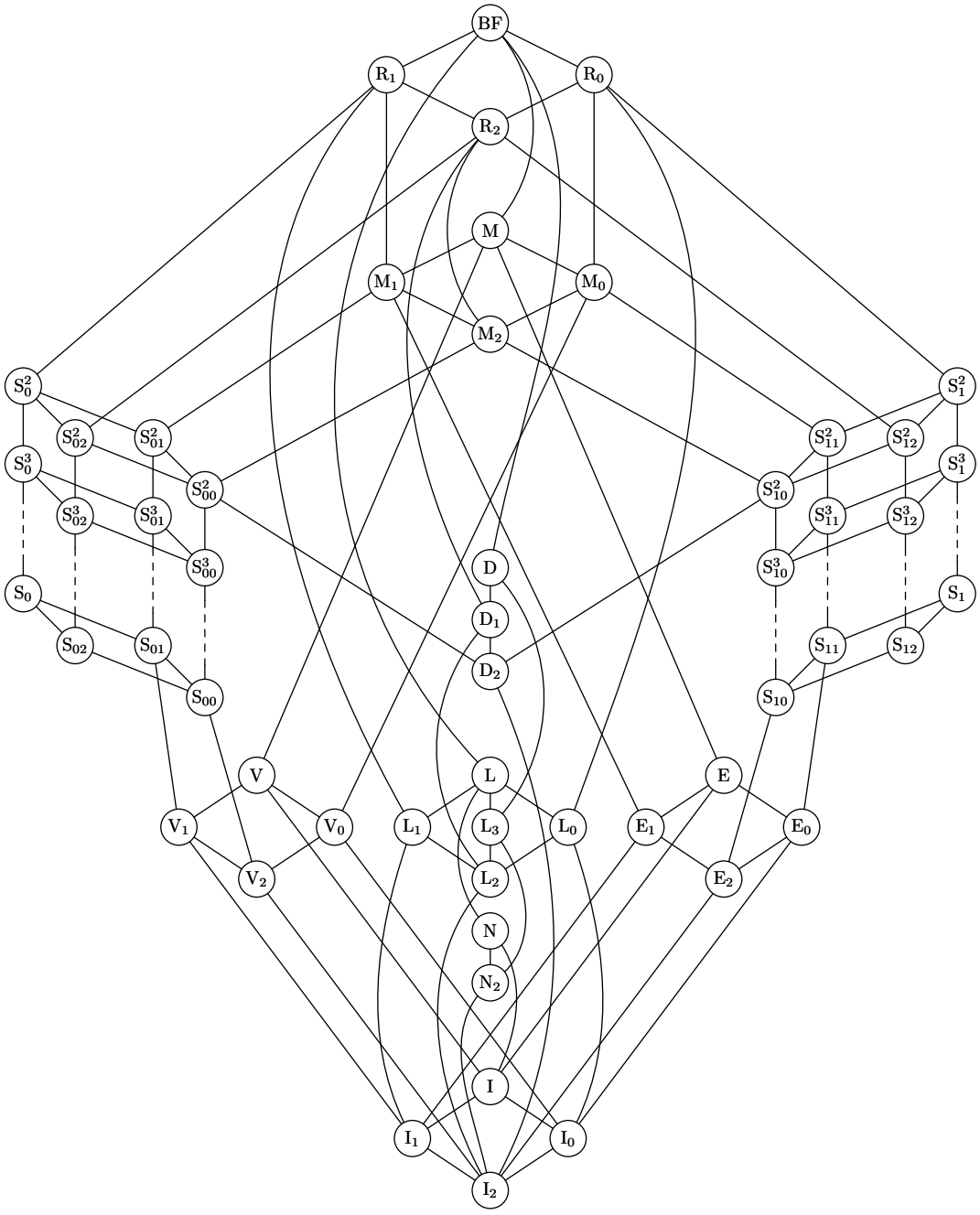

Figure 1.2: Graph of all closed classes of Boolean functions

Clone	**Definition**	**Base**
BF	All Boolean functions	$\{\vee, \wedge, \neg\}$
R_0	$\{f \in BF \mid f$ is 0-reproducing $\}$	$\{\wedge, \oplus\}$
R_1	$\{f \in BF \mid f$ is 1-reproducing $\}$	$\{\vee, \leftrightarrow\}$
R_2	$R_1 \cap R_0$	$\{\vee, x \wedge (y \leftrightarrow z)\}$
M	$\{f \in BF \mid f$ is monotone $\}$	$\{\vee, \wedge, 0, 1\}$
M_0	$M \cap R_0$	$\{\vee, \wedge, 0\}$
M_1	$M \cap R_1$	$\{\vee, \wedge, 1\}$
M_2	$M \cap R_2$	$\{\vee, \wedge\}$
S_0^n	$\{f \in BF \mid f$ is 0-separating of degree $n\}$	$\{\rightarrow, \text{dual}(h_n)\}$
S_0	$\{f \in BF \mid f$ is 0-separating $\}$	$\{\rightarrow\}$
S_1^n	$\{f \in BF \mid f$ is 1-separating of degree $n\}$	$\{x \wedge \overline{y}, h_n\}$
S_1	$\{f \in BF \mid f$ is 1-separating $\}$	$\{x \wedge \overline{y}\}$
S_{02}^n	$S_0^n \cap R_2$	$\{x \vee (y \wedge \overline{z}), \text{dual}(h_n)\}$
S_{02}	$S_0 \cap R_2$	$\{x \vee (y \wedge \overline{z})\}$
S_{01}^n	$S_0^n \cap M$	$\{\text{dual}(h_n), 1\}$
S_{01}	$S_0 \cap M$	$\{x \vee (y \wedge z), 1\}$
S_{00}^n	$S_0^n \cap R_2 \cap M$	$\{x \vee (y \wedge z), \text{dual}(h_n)\}$
S_{00}	$S_0 \cap R_2 \cap M$	$\{x \vee (y \wedge z)\}$
S_{12}^n	$S_1^n \cap R_2$	$\{x \wedge (y \vee \overline{z}), h_n\}$
S_{12}	$S_1 \cap R_2$	$\{x \wedge (y \vee \overline{z})\}$
S_{11}^n	$S_1^n \cap M$	$\{h_n, 0\}$
S_{11}	$S_1 \cap M$	$\{x \wedge (y \vee z), 0\}$
S_{10}^n	$S_1^n \cap R_2 \cap M$	$\{x \wedge (y \vee z), h_n\}$
S_{10}	$S_1 \cap R_2 \cap M$	$\{x \wedge (y \vee z)\}$
D	$\{f \in BF \mid f$ is self-dual$\}$	$\{x\overline{y} \vee x\overline{z} \vee (\overline{y} \wedge \overline{z})\}$
D_1	$D \cap R_2$	$\{xy \vee x\overline{z} \vee y\overline{z}\}$
D_2	$D \cap M$	$\{xy \vee yz \vee xz\}$
L	$\{f \in BF \mid f$ is linear$\}$	$\{\oplus, 1\}$
L_0	$L \cap R_0$	$\{\oplus\}$
L_1	$L \cap R_1$	$\{\leftrightarrow\}$
L_2	$L \cap R$	$\{x \oplus y \oplus z\}$
L_3	$L \cap D$	$\{x \oplus y \oplus z \oplus 1\}$
V	$\{f \in BF \mid f$ is constant or an $n-$ary OR function$\}$	$\{\vee, 0, 1\}$
V_0	$[\{\vee\}] \cup [\{0\}]$	$\{\vee, 0\}$
V_1	$[\{\vee\}] \cup [\{1\}]$	$\{\vee, 1\}$
V_2	$[\{\vee\}]$	$\{\vee\}$
E	$\{f \in BF \mid f$ is constant or an $n-$ary AND function$\}$	$\{\wedge, 0, 1\}$
E_0	$[\{\wedge\}] \cup [\{0\}]$	$\{\wedge, 0\}$
E_1	$[\{\wedge\}] \cup [\{1\}]$	$\{\wedge, 1\}$
E_2	$[\{\wedge\}]$	$\{\wedge\}$
N	$[\{\neg\}] \cup [\{0\}] \cup [\{1\}]$	$\{\neg, 1\}$
N_2	$[\{\neg\}]$	$\{\neg\}$
I	$[\{\text{id}\}] \cup [\{0\}] \cup [\{1\}]$	$\{\text{id}, 0, 1\}$
I_0	$[\{\text{id}\}] \cup [\{0\}]$	$\{\text{id}, 0\}$
I_1	$[\{\text{id}\}] \cup [\{1\}]$	$\{\text{id}, 1\}$
I_2	$[\{\text{id}\}]$	$\{\text{id}\}$

Table 1.1: Bases for all Boolean clones

i.e., by choosing another set of connectors than $\{\wedge, \vee, \neg\}$, can change the complexity of formula-related problems. These restricted classes of circuits and formulas give rise to a number of computational problems. For many problems which have been considered for arbitrary formulas, it can be shown that the complexity of the problem decreases if the set of allowed connectives is restricted. A classical example of this is the satisfiability problem, as discussed above:

Problem: $\mathsf{SAT}(B)$
Input: A B-formula φ
Question: Is φ satisfiable?

Theorem 1.3.3 states that $\mathsf{SAT}(\{\wedge, \vee, \neg\})$ is NP-complete. For other sets B, this problem is easier: as explained above, the problem $\mathsf{SAT}(\{\wedge, \vee, 0, 1\})$ can be solved in polynomial time. To get a complete classification of the complexity of this problem, and others in the context of B-formulas, the question when short formulas for some functions exist is of importance: suppose we have some decision problem $\mathsf{PROBLEM}(B)$, depending on some set B of Boolean functions, and suppose B_1 and B_2 are two sets of Boolean functions such that $B_1 \subseteq [B_2]$. We would like to conclude that the complexity of $\mathsf{PROBLEM}(B_1)$ is lower than the complexity of $\mathsf{PROBLEM}(B_2)$, since every function from B_1 can be expressed by a B_2-formula. Therefore we would expect the set B_2 to give a problem of at least the same complexity, we would expect that $\mathsf{PROBLEM}(B_1) \leq_m^p \mathsf{PROBLEM}(B_2)$. The canonical reduction from $\mathsf{PROBLEM}(B_1)$ to $\mathsf{PROBLEM}(B_2)$ is to replace every occurrence of some B_1-formula in a given instance with its equivalent B_2-formula. While this certainly does preserve all properties of formulas that we normally would be interested in, this straightforward reduction cannot necessarily be computed in polynomial time. For example, let $B_1 =_{\text{def}} \{\oplus\}$, and let $B_2 =_{\text{def}} \{\wedge, \vee, \neg\}$. Since the clone generated by B_2 is known to contain all Boolean functions, it is obvious that $B_1 \subseteq [B_2]$. Now consider the family of formulas $(\varphi_n)_{n\in\mathbb{N}}$, where $\varphi_n =_{\text{def}} x_1 \oplus x_2 \oplus \cdots \oplus x_n$. To express $x_1 \oplus x_2$ with B_2, we would use the formula $(x_1 \wedge \overline{x_2}) \vee (\overline{x_1} \wedge x_2)$. But what happens if we use this recursively? It can easily be seen that the $\{\wedge, \vee, \neg\}$-formula representing $x_1 \oplus x_2 \oplus \cdots \oplus x_n$ obtained by recursively applying this identity is of exponential size in n. This example illustrates that in general, the canonical reduction is not necessarily computable in polynomial time. The key problem here is that in the formula $(x_1 \wedge \overline{x_2}) \vee (\overline{x_1} \wedge x_2)$, each of the relevant variables appears twice. Therefore, recursive replacement of $\varphi \oplus \psi$ with $(\varphi \wedge \overline{\psi}) \vee (\overline{\varphi} \wedge \psi)$ leads to a "combinatorial explosion." We can avoid this problem, and give an efficient reduction from $\mathsf{PROBLEM}(B_1)$ to $\mathsf{PROBLEM}(B_2)$, if we can represent every function from B_1 with a "short" B_2-formula. We will now define what we mean by this:

Definition Let f be an n-ary Boolean function, and let B_1, B_2 be sets of Boolean functions. We say that B_1 *efficiently implements* f, if there is a B_1-formula φ such that $\mathrm{VAR}(\varphi) = \{x_1, \ldots, x_n, y_1, \ldots, y_k\}$, and for all $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_k \in \{0, 1\}$, it holds that $f(\alpha_1, \ldots, \alpha_n) = 1$ if and only if I is a solution for φ, where $I(x_i) = \alpha_i$, and $I(y_i) = \beta_i$ for all relevant i, and each variable x_i appears in φ at most once. Such a formula φ is called an *efficient B_1 implementation of f*. We say B_2 *efficiently implements* B_1, if B_2 efficiently implements every function from B_1.

If the sets B_1 and B_2 have the property defined above, then we can, in most cases, show that $\mathsf{PROBLEM}(B_1) \leq_m^p \mathsf{PROBLEM}(B_2)$ holds. This holds for any $\mathsf{PROBLEM}$ which depends only on the function represented by a given instance. We say a variable is *irrelevant* for the formula φ if the function represented by φ does not depend on the corresponding argument (note that the variables y_i in the above definition do not contribute to the function represented by the formula). In this case, the straightforward transformation by replacement can be performed in polynomial time.

Proposition 1.4.1 *Let B_1, B_2 be sets of Boolean functions such that B_1 is finite, and B_2 efficiently implements B_1. Then there is a polynomial-time computable function f which computes, for a given B_1-formula, an equivalent B_2-formula, with additional irrelevant variables.*

This immediately gives a polynomial time reduction for all problems which are invariant under transformations of the formulas which give an equivalent formula with additional irrelevant variables. Since this is obviously the case for the satisfiability problem, we immediately get the following Corollary:

Corollary 1.4.2 *Let B_1, B_2 be sets of Boolean functions such that B_1 is finite, and B_2 efficiently implements B_1. Then $\mathsf{SAT}(B_1) \leq_m^p \mathsf{SAT}(B_2)$.*

To our knowledge, the first known application of this technique to complexity theory was a result by Harry Lewis. The following theorem, similarly to Schaefer's Theorem which we will state later, proves a dichotomy for the classes P and NP. The fact that such dichotomy theorems exist is very interesting, because due to a classic result result by Richard Ladner [Lad75b], there are an infinite number of $\leq_m^p$-degrees between the classes P and NP, unless these classes coincide. We will state the theorem and give a variation of Lewis' proof, since it is an important example of how short formulas can be used.

Theorem 1.4.3 ([Lew79]) *Let B be a finite set of Boolean functions. Then $\mathsf{SAT}(B)$ is* NP*-complete under $\leq_m^p$-reductions if $f \in [B]$, where $f(x, y) = x \wedge \overline{y}$, and $\mathsf{SAT}(B)$ is solvable in polynomial time otherwise.*

The proof for this theorem makes extensive use of Post's lattice. The polynomial time cases follow easily: For any set $B \subseteq \mathrm{R}_1$, every B-formula φ is satisfiable by the all-1-assignment. For a set $B \subseteq \mathrm{M}$, a B-formula φ is satisfied if and only if the all-1-assignment satisfies it. In the case $B \subseteq \mathrm{L}$, the formula can easily be rewritten in a formula of the form $x_1 \oplus x_2 \oplus \cdots \oplus x_n \oplus c$, where $c \in \{0, 1\}$. Now this formula is satisfiable if and only if $c = 0$ and there is a variable occurring an odd number of times, or $c = 1$. In the case $B \subseteq \mathrm{D}$, if the all-0-assignment does not satisfy a B-formula φ, then the all-1-assignment does. Now a quick glance at Post's lattice presented in Figure 1.2 shows that in all remaining cases, $B \subseteq \mathrm{S}_1$ holds. Since the function f forms a base for this clone, it remains to show that $\mathsf{SAT}(B)$ is NP-hard for all sets B such that $f \in B$. To prove this, Lewis showed a lemma proving the existence of efficient implementations, which is the main technical difficulty in his proof of the dichotomy theorem:

Lemma 1.4.4 ([Lew79]) *Let B be a set of Boolean functions such that $[B] = \mathrm{BF}$. Then B efficiently implements $\{\wedge, \vee, \neg\}$.*

With the help of this lemma, the proof for the hardness part of Theorem 1.4.3 can be done easily: Let B be a set of Boolean functions such that $f \in [B]$. Since $[\{f\}] = \mathrm{S}_1$, it follows that $[B] \supseteq \mathrm{S}_1$. Post's lattice (Figure 1.2) reveals that $[B \cup \{1\}] = \mathrm{BF}$. Therefore, $B \cup \{1\}$ efficiently implements $\{\wedge, \vee, \neg\}$ due to Lemma 1.4.4. From Corollary 1.4.2, we conclude that $\mathsf{SAT}(B \cup \{1\})$ is NP-hard. It therefore remains to prove that $\mathsf{SAT}(B \cup \{1\}) \leq_m^p \mathsf{SAT}(B)$. For this, let φ be a $B \cup \{1\}$-formula, and let t be a variable not appearing in φ. In φ, replace every occurrence of the constant 1 with the variable t. Let φ' denote the result of this transformation. Now it is obvious that φ is satisfiable if and only if $\varphi' \wedge t$ is. The latter formula can be constructed, since the AND operator can be expressed with B, because it is in the clone S_1. The set B does not necessarily implement the AND function efficiently, but since this does not appear nested, and the B-formula for AND is fixed, this can be done in polynomial time. Hence the reduction is complete.

This classical example shows how efficient implementations can be used to get reductions. However, Lemma 1.4.4 only can be applied to *complete* sets B, i.e., sets B of Boolean functions where $[B] = \mathrm{BF}$. But in many cases, we need efficient implementations for the important functions $\vee$, $\wedge$, and $\neg$ for other sets B. The following lemma is an extension of Lewis' lemma above, and gives efficient implementations for a variety of bases.

Lemma 1.4.5 *Let B be a finite set of Boolean functions such that $0, 1 \in B$.*

1. *If $[B] \in \{\mathrm{V}, \mathrm{M}\}$, then B efficiently implements $\vee$.*
2. *If $[B] \in \{\mathrm{E}, \mathrm{M}\}$, then B efficiently implements $\wedge$.*
3. *If $\mathrm{N} \subseteq [B]$, then B efficiently implements $\neg$ via some formula φ. If $[B] \subseteq \mathrm{L}$, and no function in B is degenerated, then φ can be chosen in such a way that the variable x occurs in φ as the last symbol which is not a parenthesis.*

Proof. 1. Since $\vee \in \mathrm{V}_2 \subseteq [B]$, and $[B]$ contains the set of functions which can be represented by B-formulas, it follows that there is a B-formula $\varphi(x, y)$ such that φ represents $x \vee y$ and has a minimal number of occurrences of x and y. Let n be the number of occurrences of x, and m the number of occurrences of y in φ. Without loss of generality, let $m \leq n$, and assume $n \geq 2$. Let $\varphi_\#(x_1, \ldots, x_n, y_1, \ldots, y_m)$ be the formula obtained from φ by numbering the variable occurrences, i.e., renaming the i-th occurrence of x to x_i and accordingly for y.

Since $1 \in B$, we can construct a B-formula equivalent to

$$\varphi'(x, y) =_{\mathrm{def}} \varphi_\#(x_1/1, x_2/x, \ldots, x_n/x, y_1/y, \ldots, y_m/y).$$

The minimality of φ implies that $\varphi'(x, y)$ does not represent $x \vee y$. Since $B \subseteq \mathrm{M}$, the function represented by $\varphi_\#$ is monotone, and thus for all $\alpha, \beta \in \{0, 1\}$, it follows

that $\varphi'(x/\alpha, y/\beta) \geq \alpha \vee \beta$ holds. In particular, this implies the following:

$$\begin{aligned} \varphi'(x/0, y/1) &= 1, \\ \varphi'(x/1, y/0) &= 1, \\ \varphi'(x/1, y/1) &= 1. \end{aligned}$$

Now assume that $\varphi'(x/0, y/0) = 0$. Then it follows that φ' is a representation of the OR-function. This is a contradiction to the minimality of φ. Hence, we know that $\varphi'(x/0, y/0) = 1$, and this means that $\varphi'(x/\alpha, y/\beta) = 1$ for all Boolean values α and β, in particular:

$$\varphi'(x/0, y/0) = \varphi_{\#}(x_1/1, x_2/0, \ldots, x_n/0, y_1/0, \ldots, y_m/0) = 1,$$

and since $\varphi_{\#}$ is monotone, this implies

$$\varphi_{\#}(x_1/1, x_2/\alpha_2, \ldots, x_n/\alpha_n, y_1/\beta_1, \ldots, y_m/\beta_m) = 1. \tag{1.1}$$

for all $\alpha_2, \ldots, \beta_m \in \{0, 1\}$. Since $0 \in B$, we can construct the B-formula

$$\varphi''(x, y) =_{\text{def}} \varphi(x_1/x, x_2/0, x_3/x, \ldots, x_n/x, y_1/y, \ldots, y_m/y).$$

Observe that the following holds:

$$\begin{array}{lcl} \varphi''(x/1, y/0) & = & \varphi_{\#}(x_1/1, x_2/0, x_3/1, \ldots, x_n/1, y_1/0, \ldots, y_m/0) = 1 \\ \varphi''(x/1, y/1) & \geq & \varphi''(x/1, y/0) = 1 \\ \varphi''(x/0, y/1) & = & \varphi(x/0, y/1) = 1 \\ \varphi''(x/0, x/0) & = & \varphi(x/0, y/0) = 0 \end{array}$$

The first of these equations follows from equation 1.1, the second one is true because the function represented by φ'' is monotone, and the final two follow from the choice of φ. Thus, φ'' represents the OR function, and φ'' has one variable less than φ, which is a contradiction to the minimality of φ. Therefore, φ only contains two variable occurrences.

2. This follows with an analogous proof. It also follows from the duality implicit in Post's lattice.

3. Existence of the formula φ follows with Lemma 1 from [Lew79]. Lewis only states the result for complete sets B, but his proof only relies on the fact that negation and both constants can be expressed with B-formulas. Now observe that if $[B] \subseteq$ L holds, then every B-formula represents a function g which is symmetric in all of its relevant arguments (observe that all arguments to all functions in B are relevant). Therefore, the only variable x in φ can be moved to the end of the formula by swapping arguments, which does not change the function represented by the formula.

□

1.5 Constraints

Using B-formulas as defined in the previous section, we can generalize well-known restrictions of propositional formulas, like monotone formulas. But there are other restrictions which cannot be expressed in this way, like the restriction to 3CNF.

Constraint satisfaction problems are natural generalizations of these and other satisfiability problems, and can be used to express many combinatorial problems. Further key examples are search and colorability problems in graphs. They also can be seen as a homomorphism problem, where the task is to find out if there exists a homomorphism between two finite given relational structures [FV98]. We now define constraint satisfaction problems formally and then explain why they can be used to express some of the problems mentioned above.

Definition Let D be an arbitrary set. A *constraint language* Γ *over the domain* D is a finite set of nonempty, finitary relations over D. A Γ-*formula* φ is a formula of the form

$$\varphi(x_1, \ldots, x_m) = \bigwedge_{i=1}^{n} R_i(x_1^i, \ldots, x_{k_i}^i),$$

where $x_j^i \in \{x_1, \ldots, x_m\}$ for $1 \leq i \leq n, 1 \leq j \leq k_i$, and for all i, R_i is a k_i-ary relation from Γ. A clause of the form $R_i(x_1^i, \ldots, x_{k_i}^i)$ is called a *constraint* or a *constraint application*. An *assignment* I *for* φ is a function $I\colon \mathrm{VAR}(\varphi) \to D$. An assignment I *satisfies* φ, written as $I \models \varphi$, if $(I(x_1^i), \ldots, I(x_{k_i}^i)) \in R_i$ for all $1 \leq i \leq n$. A formula φ is *satisfiable* if there exist a satisfying assignment for φ.

The requirement that the constraint languages do not contain the empty relation is not just for convenience, but of some importance. We will discuss this after the statement of Theorem 1.5.2. For a single-element constraint language $\{R\}$, we often simply write R instead of $\{R\}$. We call a constraint language over a domain of cardinality two a *Boolean constraint language*. We can now define the constraint satisfaction problem $\mathsf{CSP}(\Gamma)$:

Problem:	$\mathsf{CSP}(\Gamma)$
Input:	A Γ-formula φ
Question:	Is φ satisfiable?

On first glance, the constraint satisfaction problem and the "usual" satisfiability problem for Boolean formulas seem to be very much alike. And in fact, the step from the normal propositional problem to the constraint problem is as subtle as the step from the problem SAT to the problem $\mathsf{3SAT}$. The significance of this step is that one very important aspect of Boolean formulas is lost: formulas can be nested arbitrarily, and in many problems, it is this nesting which gives the complexity. In a constraint satisfaction problem, nesting does not occur. The formula here consists of a collection of "local" conditions restricting the possible values of the appearing variables.

There is a canonical correspondence between formulas and relations: for each Boolean formula, the set of its satisfying assignments defines a relation. For example, the formula $x \vee y$ defines the relation $\{(0,1),(1,0),(1,1)\}$. This relation also is often referred to simply as OR. Therefore it is natural to write relations as Boolean formulas.

We now explain how the problem 3SAT defined above can be seen as a constraint satisfaction problem.

Definition We define the following Boolean relations:

$$
\begin{array}{llll}
R_{x\vee y\vee z} & =_{\text{def}} & \{0,1\}^3 \setminus \{(0,0,0)\}\,, & R_{\overline{x}\vee\overline{y}\vee z} =_{\text{def}} \{0,1\}^3 \setminus \{(1,1,0)\}\,,\\
R_{\overline{x}\vee y\vee z} & =_{\text{def}} & \{0,1\}^3 \setminus \{(1,0,0)\}\,, & R_{\overline{x}\vee y\vee\overline{z}} =_{\text{def}} \{0,1\}^3 \setminus \{(1,0,1)\}\,,\\
R_{x\vee\overline{y}\vee z} & =_{\text{def}} & \{0,1\}^3 \setminus \{(0,1,0)\}\,, & R_{x\vee\overline{y}\vee\overline{z}} =_{\text{def}} \{0,1\}^3 \setminus \{(0,1,1)\}\,,\\
R_{x\vee y\vee\overline{z}} & =_{\text{def}} & \{0,1\}^3 \setminus \{(0,0,1)\}\,, & R_{\overline{x}\vee\overline{y}\vee\overline{z}} =_{\text{def}} \{0,1\}^3 \setminus \{(1,1,1)\}\,.
\end{array}
$$

The constraint language Γ_{3SAT} contains all the relations defined above.

It is obvious that, for instance, the clause $R_{\overline{x}\vee y\vee\overline{z}}(x,y,z)$ is equivalent to $\overline{x}\vee y\vee\overline{z}$. Since the language Γ_{3SAT} contains all possible combinations of a 3SAT-clause, the problem $\mathsf{CSP}(\Gamma_{\mathsf{3SAT}})$ is essentially the same problem as 3SAT. Therefore, Theorem 1.3.4 immediately gives the following Corollary:

Corollary 1.5.1 *The problem* $\mathsf{CSP}(\Gamma_{\mathsf{3SAT}})$ *is* NP*-complete under* $\leq_m^{\log}$*-reductions.*

The relation between 3SAT and $\mathsf{CSP}(\Gamma_{\mathsf{3SAT}})$ is much stronger than just a logspace reduction: the original 3CNF-formula and the constraint formula have the exact same set of satisfying assignments, i.e., they are equivalent. Therefore, the reduction cannot only be applied to the satisfiability problem, but also to problems where we are interested in other properties of the involved formulas, for example, the problem to determine the number of solutions, or to enumerate all solutions. Similar relationships exist for other examples, as the aforementioned graph problems. For this reason, constraint satisfaction problems can be used to express various kinds of combinatorial problems in such a way that the CSP-instance contains all the information about the original problem. This immediately gives a vast number of applications of the constraint satisfaction problem, and explains one reason why these problems have received a lot of attention in computational complexity theory.

In addition to applications, these problems are also very interesting from a theoretical point of view. In the seminal paper [Sch78], Thomas Schaefer proved that for any Boolean constraint language Γ, the problem $\mathsf{CSP}(\Gamma)$ can either be solved in polynomial time, or it is NP-complete. In [FV98], Thomás Feder and Moshe Vardi proved that in a certain framework, the class of constraint satisfaction problems is the largest class of problems where dichotomy results are expected. It is conjectured that the dichotomy behavior mentioned above for the Boolean case also holds for arbitrary finite domains. Andrei Bulatov proved this for the case of three-element domains in [Bul06].

When looking at some problem which can be parametrized by its constraint language, like $\mathsf{CSP}(\Gamma)$, we again face the problem of how to handle all of these problems at once. Similarly to the case for B-formulas as defined previously, we can again utilize a closure operator, this time on the set of relations. The following operator has proven to be helpful in the context of constraints:

Definition Let Γ be a constraint language on the domain D. Then $\langle\Gamma\rangle$ contains all k-ary relations on D which can be defined by a formula φ of the following form:

$$\varphi(x_1,\ldots,x_k) = \exists y_1 \ldots \exists y_l \bigwedge_{i=1}^{n} R_i(z_1^i,\ldots,z_{k_i}^i),$$

where for $i \in \{1,\ldots,n\}$, R_i is a k_i-ary relation from $\Gamma \cup \{=\}$, and for each i and j, z_j^i is a variable from $\{x_1,\ldots,x_k,y_1,\ldots,y_l\}$. The set $\langle\Gamma\rangle$ is called the *co-clone generated by* Γ.

It can easily be verified that $\langle . \rangle$ is a closure operator, and hence by the same argument as for the clones, the sets closed under this operator, the co-clones, again form a lattice. The relation $=$ denoting the equality relation on D can be problematic when applying this closure operator in a complexity context, as we will see in Chapters 4 and 5.

There is a close relationship between functions and relations on a given domain. Therefore it is not surprising to discover that there is a strong correspondence between clones and co-clones. To explain the correspondence, we need one more definition:

Definition Let R be a k-ary relation on a domain D, and let $f\colon D^n \to D$ be an n-ary function on D. We say that R is *closed* under f, or that f is a *polymorphism* of R, if for all $(\alpha_1^1,\ldots,\alpha_k^1),\ldots,(\alpha_1^n,\ldots,\alpha_k^n) \in R$, it holds that

$$(f(\alpha_1^1,\ldots,\alpha_1^n),\ldots,f(\alpha_k^1,\ldots,\alpha_k^n)) \in R,$$

i.e., R is closed under coordinate-wise application of f. We denote the set of polymorphisms of R with $\mathrm{Pol}(R)$, and for a constraint language Γ, we define $\mathrm{Pol}(\Gamma) =_{\mathrm{def}} \bigcap_{R\in\Gamma} \mathrm{Pol}(R)$. For a set B of functions on D, we define $\mathrm{Inv}(B)$ to be the set of relations R on D such that $B \subseteq \mathrm{Pol}(R)$.

It follows immediately from the definition that the operators $\mathrm{Pol}(.)$ and $\mathrm{Inv}(.)$ form a *Galois connection* between the sets of relations and the sets of functions over the domain D. As noted by Marcel Erné in [Ern04], a Galois connection is only of practical use when the involved closure operators are known. In the case of the Galois connection induced by $\mathrm{Pol}(.)$ and $\mathrm{Inv}(.)$, the closure operators $\mathrm{Pol}(\mathrm{Inv}(.))$ on the set of functions and $\mathrm{Inv}(\mathrm{Pol}(.))$ on the set of relations are exactly the same operators as we already encountered. The following theorem was known in mathematics since 1968. To our knowledge, it was first applied in a complexity context by Peter Jeavons, David Cohen, and Marc Gyssens in the influential paper [JCG97].

Theorem 1.5.2 ([Gei68, JCG97]) *Let Γ_1,Γ_2 be constraint languages over a finite domain D, and let B be a set of functions on D.*

1. *The inclusion $\mathrm{Pol}(\Gamma_1) \subseteq \mathrm{Pol}(\Gamma_2)$ holds if and only if $\langle\Gamma_2\rangle \subseteq \langle\Gamma_1\rangle$ holds.*
2. *$[B] = \mathrm{Pol}(\langle\mathrm{Inv}(B)\rangle)$, and $\langle\Gamma_1\rangle = \mathrm{Inv}(\mathrm{Pol}(\Gamma_1))$.*

For constraint languages Γ_1,Γ_2, such that $\Gamma_1 \subseteq \langle\Gamma_2\rangle$, with help of additional existentially quantified variables and the equality relation, Γ_2 can express every relation in Γ_1. Therefore, the expressive power of Γ_2 is stronger than that of Γ_1. Seen in this way, the

above Theorem 1.5.2 states that the more polymorphisms a constraint language has, the weaker its expressive power is. Also note that for any set B of functions on some finite domain, the set $\mathrm{Inv}(B)$ is a co-clone, and for any set of relations Γ, the set $\mathrm{Pol}(\Gamma)$ is a clone.

It is easy to see that Theorem 1.5.2 fails if we allow the empty relation to be a member of a constraint language: obviously, the empty relation is closed under negation and conjunction, and hence is invariant under every function from the clone BF. Therefore, Theorem 1.5.2 would imply that this relation is a member of every possible co-clone. This can easily be seen not to hold: As a counter-example, consider the co-clone generated by the Boolean constraint language Γ containing the implication as only relation. It is easy to see that every Γ-formula is satisfiable by the constant assignments. Therefore, the empty relation cannot be obtained as a Γ-formula with additional existential variables. We believe that disallowing the empty relation in constraint languages is the most natural way to deal with this problem, since formulas in which the empty relation appear are obviously unsatisfiable, and hence not of much interest.

It is not surprising that constraint languages with high expressive power give rise to constraint satisfaction problems with high complexity. The following theorem states this formally. The result for polynomial time reductions follows easily from Theorem 1.5.2, and is due to [JCG97]. In Chapter 4, we will show that this reduction can also be performed in logarithmic space (see Corollary 4.2.3).

Theorem 1.5.3 ([JCG97]) *Let Γ_1, Γ_2 be constraint languages over a finite domain D such that $\mathrm{Pol}(\Gamma_2) \subseteq \mathrm{Pol}(\Gamma_1)$. Then $\mathsf{CSP}(\Gamma_1) \leq_m^p \mathsf{CSP}(\Gamma_2)$.*

One of the most important consequences of this theorem is that for a constraint language Γ, the complexity of $\mathsf{CSP}(\Gamma)$ depends only on the co-clone generated by Γ. A corollary of Theorem 1.5.2 is that the lattice of clones and the lattice of co-clones are very similar: the operators $\mathrm{Pol}(.)$ and $\mathrm{Inv}(.)$ give a dual isomorphism between these lattices (i.e., an isomorphism which reverses the order). Intuitively speaking, the co-clone lattice is the clone-lattice "upside down." Therefore, Post's classification of the Boolean clones immediately gives a classification of the Boolean co-clones as well. This classification can be used to show dichotomy results for problems over the Boolean domain in the constraint context. For example, Theorem 1.5.3 and Post's classification can be used to give a very short proof of Schaefer's classic dichotomy theorem, which we will state now.

Theorem 1.5.4 ([Sch78]) *Let Γ be a Boolean constraint language. If $\mathrm{Pol}(\Gamma)$ is either I_2 or N_2, then $\mathsf{CSP}(\Gamma)$ is* NP*-complete. Otherwise, $\mathsf{CSP}(\Gamma)$ can be solved in polynomial time.*

This theorem was one of the first complexity results in the context of constraint satisfaction problems, and is the starting point of a long line of research. A proof using the Galois connection can be found in [BCRV04]. Some of the properties that Schaefer used in his proof have more consequences than just ensuring the tractability of the satisfiability problem. In the following, the *ternary majority function* on the Boolean domain is the function $f\colon \{0,1\} \to \{0,1\}$ such that for $\alpha, \beta \in \{0,1\}$, it holds that $f(\alpha,\beta,\beta) = f(\beta,\alpha,\beta) = f(\beta,\beta,\alpha) = \beta$.

Definition Let R be a Boolean relation. We say that

- R is *Horn*, if $\wedge \in \mathrm{Pol}(R)$,
- R is *anti-Horn*, if $\vee \in \mathrm{Pol}(R)$,
- R is *bijunctive*, if the ternary majority function is a polymorphism of R,
- R is *affine*, if $x \oplus y \oplus z \in \mathrm{Pol}(R)$,
- R is *complementive*, if $\neg \in \mathrm{Pol}(R)$,
- R is 0-*valid*, if the constant 0-function is a polymorphism of R,
- R is 1-*valid*, if the constant 1-function is a polymorphism of R.

A Boolean constraint language Γ has one of the above properties if every relation in Γ has, and we say that Γ is *Schaefer* if Γ is Horn, anti-Horn, bijunctive or affine.

The following fact is often useful:

Proposition 1.5.5 *Let Γ be a Boolean constraint language which is Schaefer. Then $\Gamma \cup \{x, \overline{x}\}$ is Schaefer as well.*

Proof. This can easily be seen by verifying that each of the polymorphisms leading to the defining properties of Schaefer is a polymorphism of the relations $\{(0)\}$ and $\{(1)\}$ as well. □

In the literature, often problems called "satisfiability with constants," denoted with SAT_c, have been studied. These are problems where in addition to variables, the constants 0 and 1 may appear in the formulas. Proposition 1.5.5 implies that if we consider constraint languages which are Schaefer, then adding constants in the sense of SAT_c does not give higher complexity.

It is often convenient to know a base for each co-clone. Due to Theorem 1.5.2, we know that the lattice of co-clones is isomorphic to the lattice of clones with an order-reversing isomorphism. But in contrast to the lattice of clones, there are some Boolean co-clones which do not have a finite base, as proven by the following lemma:

Lemma 1.5.6 *There is no Boolean constraint language Γ such that $\langle\Gamma\rangle \in \{\mathrm{Inv}(\mathrm{S}_0), \mathrm{Inv}(\mathrm{S}_{02}), \mathrm{Inv}(\mathrm{S}_{01}), \mathrm{Inv}(\mathrm{S}_{00}), \mathrm{Inv}(\mathrm{S}_{10}), \mathrm{Inv}(\mathrm{S}_{11}), \mathrm{Inv}(\mathrm{S}_{12}), \mathrm{Inv}(\mathrm{S}_1)\}$.*

Proof. Assume that Γ is such a finite constraint language. Since Γ is finite, it follows that $\Gamma = \{R_1, \ldots, R_n\}$ for some Boolean relations $R_1, \ldots, R_n$. By definition, every clone S_{ab} is the intersection of all S_{ab}^k. Due to the correspondence between clones and co-clones exhibited in Theorem 1.5.2, it follows that every co-clone $\mathrm{Inv}(\mathrm{S}_{ab})$ is the union of all $\mathrm{Inv}\left(\mathrm{S}_{ab}^k\right)$. Therefore, for every $i \in \{1, \ldots, n\}$ there exists a k_i such that $R_i \in \mathrm{Inv}\left(\mathrm{S}_{ab}^{k_i}\right)$. Since the $\mathrm{Inv}\left(\mathrm{S}_{ab}^k\right)$-co-clones form a chain in the lattice, there exists some k such that $\Gamma \subseteq \mathrm{Inv}\left(\mathrm{S}_{ab}^k\right) \subsetneqq \mathrm{Inv}(\mathrm{S}_{ab})$, which is a contradiction. □

For all other Boolean co-clones, a finite base can be constructed; a list is presented in Table 1.2. The correctness of these bases can easily be verified by hand or by an algorithm: For a constraint language Γ and a clone C, it holds that $\langle\Gamma\rangle = \mathrm{Inv}(C)$ if and only if $\mathrm{Pol}(\Gamma) = C$. Since for each Boolean clone, a finite base and a finite set of neighbors in Post's lattice is known, this test can easily be performed. Also, it is easy to derive, from a base given for a co-clone, a base for the dual co-clone. Similarly, the case where a clone is the intersection of two others can be used to obtain a base. Thus, we omit the proofs for the correctness of the results in the table. We define some of the relations which appear in the given bases. NAE is an abbreviation for "not all equal," and DUP stands for "duplicate."

Definition Let m be a natural number. We define the following relations:

$$
\begin{aligned}
\mathrm{OR}^m &=_{\mathrm{def}} \{(\alpha_1,\ldots,\alpha_m) \in \{0,1\}^m \mid 1 \in \{\alpha_1,\ldots,\alpha_n\}\}, \\
\mathrm{NAND}^m &=_{\mathrm{def}} \{(\alpha_1,\ldots,\alpha_m) \in \{0,1\}^m \mid 0 \in \{\alpha_1,\ldots,\alpha_n\}\}, \\
\mathrm{EVEN^m} &=_{\mathrm{def}} \left\{(\alpha_1,\ldots,\alpha_m) \in \{0,1\}^m \mid \sum_{i=1}^{m} \alpha_i \text{ is even}\right\}, \\
\mathrm{ODD^m} &=_{\mathrm{def}} \left\{(\alpha_1,\ldots,\alpha_m) \in \{0,1\}^m \mid \sum_{i=1}^{m} \alpha_i \text{ is odd}\right\}, \\
\mathrm{NAE} &=_{\mathrm{def}} \{0,1\}^3 \setminus \{(0,0,0),(1,1,1)\}, \\
\mathrm{DUP} &=_{\mathrm{def}} \{0,1\}^3 \setminus \{(0,1,0),(1,0,1)\}, \\
\mathrm{R}_{n/m} &=_{\mathrm{def}} \{(\alpha_1,\ldots,\alpha_m) \in \{0,1\}^m \mid \alpha_1 + \cdots + \alpha_m = n\}.
\end{aligned}
$$

It should be noted that the version of Table 1.2 which appeared in [BRSV05] contained a mistake, and in that version, an incorrect base for the co-clone $\mathrm{Inv}(\mathrm{I}_1)$ was given.

Clone	**Remark**	**Base(s) of corresponding co-clone**
BF		$\{=\}$, $\{\emptyset\}$
R_0	dual of R_1	$\{\overline{x}\}$
R_1		$\{x\}$
R_2	$R_0 \cap R_1$	$\{x, \overline{x}\}$,$\{x\overline{y}\}$
M		$\{x \to y\}$
M_0	$M \cap R_0$	$\{x \to y, \overline{x}\}$, $\{\overline{x} \wedge (y \to z)\}$
M_1	$M \cap R_1$	$\{x \to y, x\}$, $\{x \wedge (y \to z)\}$
M_2	$M \cap R_2$	$\{x \to y, x, \overline{x}\}$, $\{x \to y, \overline{x \to y}\}$, $\{x\overline{y} \wedge (u \to v)\}$
S_0^n		$\{\mathrm{OR}^n\}$
S_0	$\cap_{m\geq 2} S_0^m$	$\{\mathrm{OR}^m \mid m \geq 2\}$
S_1^n	dual of S_0^n	$\{\mathrm{NAND}^n\}$
S_1	dual of S_0	$\{\mathrm{NAND}^m \mid m \geq 2\}$
S_{02}^n	$S_0^n \cap R_2$	$\{\mathrm{OR}^m, x, \overline{x}\}$
S_{02}	$S_0 \cap R_2$	$\{\mathrm{OR}^m \mid m \geq 2\} \cup \{x, \overline{x}\}$
S_{01}^n	$S_0^n \cap M$	$\{\mathrm{OR}^m, x \to y\}$
S_{01}	$S_0 \cap M$	$\{\mathrm{OR}^m \mid m \geq 2\} \cup \{x \to y\}$
S_{00}^n	$S_0^n \cap R_2 \cap M$	$\{\mathrm{OR}^m, x, \overline{x}, x \to y\}$
S_{00}	$S_0 \cap R_2 \cap M$	$\{\mathrm{OR}^m \mid m \geq 2\} \cup \{x, \overline{x}, x \to y\}$
S_{12}^n	dual of S_{02}^n	$\{\mathrm{NAND}^m, x, \overline{x}\}$
S_{12}	dual of S_{02}	$\{\mathrm{NAND}^m \mid m \geq 2\} \cup \{x, \overline{x}\}$
S_{11}^n	dual of S_{01}^n	$\{\mathrm{NAND}^m, x \to y\}$
S_{11}	dual of S_{01}	$\{\mathrm{NAND}^m \mid m \geq 2\} \cup \{x \to y\}$
S_{10}^n	dual of S_{00}^n	$\{\mathrm{NAND}^m, x, \overline{x}, x \to y\}$
S_{10}	dual of S_{00}	$\{\mathrm{NAND}^m \mid m \geq 2\} \cup \{x, \overline{x}, x \to y\}$
D		$\{x \oplus y\}$
D_1	$D \cap R_1$	$\{x \oplus y, x\}$
D_2	$D \cap M$	$\{x \oplus y, x \to y\}$,$\{x\overline{y} \vee \overline{x}yz\}$
L		$\{\mathrm{EVEN}^4\}$
L_0	$L \cap R_0$	$\{\mathrm{EVEN}^4, \overline{x}\}$,$\{\mathrm{EVEN}^3\}$
L_1	$L \cap R_1$	$\{\mathrm{EVEN}^4, x\}$,$\{\mathrm{ODD}^3\}$
L_2	$L \cap R_2$	$\{\mathrm{EVEN}^4, x, \overline{x}\}$, every $\{\mathrm{EVEN}^n, x\}$ where $n \geq 3$ is odd
L_3	$L \cap D$	$\{\mathrm{EVEN}^4, x \oplus y\}$,$\{\mathrm{ODD}^4\}$
V		$\{x \vee y \vee \overline{z}\}$
V_0	$V \cap R_0$	$\{x \vee y \vee \overline{z}, \overline{x}\}$
V_1	$V \cap R_1$	$\{x \vee y \vee \overline{z}, x\}$
V_2	$V \cap R_2$	$\{x \vee y \vee \overline{z}, x, \overline{x}\}$
E	dual of V	$\{\overline{x} \vee \overline{y} \vee z\}$
E_0	dual of V_1	$\{\overline{x} \vee \overline{y} \vee z, \overline{x}\}$
E_1	dual of V_0	$\{\overline{x} \vee \overline{y} \vee z, x\}$
E_2	dual of V_2	$\{\overline{x} \vee \overline{y} \vee z, x, \overline{x}\}$
N		$\{\mathrm{DUP}\}$
N_2	$N \cap L_3$	$\{\mathrm{DUP}, \mathrm{EVEN}^4, x \oplus y\}, \{\mathrm{NAE}\}$
I	$L \cap M$	$\{\mathrm{EVEN}^4, x \to y\}$
I_0	$L \cap M \cap R_0$	$\{\mathrm{EVEN}^4, x \to y, \overline{x}\}$,$\{\mathrm{DUP}, x \to y\}$
I_1	$L \cap M \cap R_1$	$\{\mathrm{EVEN}^4, x \to y, x\}$,$\{x \vee (y \oplus z)\}$
I_2	$L \cap M \cap R_2$	$\{\mathrm{EVEN}^4, x \to y, x, \overline{x}\}, \{R_{1/3}\}$, $\{x \to (y \oplus z)\}$

Table 1.2: Bases for all Boolean co-clones

Chapter 2

Very Basic Satisfiability: The Formula Value Problem

2.1 Introduction

When studying problems related to Boolean formulas in any way, one of the most basic computational tasks arising in any algorithm used in this context is the *Formula Value Problem*. This is the problem to evaluate a given formula without variables, and can be seen as the most basic version of satisfiability: for a formula without any variables, there is only one possible truth assignment (the empty assignment), and this satisfies the formula if and only if the formula evaluates to true.

It is obvious that the formula value problem can be solved in polynomial time. Its circuit version is one of the the most well-known problems which are complete for polynomial time, as shown by Richard Ladner [Lad75a]. In [Bus87], Samuel Buss showed that the formula version of the problem can be solved in NC^1, which means that the complexity for formulas is significantly lower than the problem for circuits. When considering formulas as tree-like circuits, this seems like a natural result: in a tree structure, the predecessors of each node are completely independent. Therefore, it is not surprising that the problem can be solved by an efficient parallel algorithm. For circuits, on the other hand, this is not possible: since for any gate in the circuit, its predecessors might "meet" deeper down in the circuit, the values of these gates cannot be considered to be independent. Therefore, this problem is one of the examples where the complexity for formulas and circuits actually differs. To our knowledge, such examples are only known for complexity classes below P - as long as only the $\leq_m^p$-degree of complexity is considered, the complexity seems to be the same if we consider formulas or circuits. This is certainly the case for the satisfiability problem, as the proof for Theorem 1.4.3 also can be applied to the circuit case. Other problems where B-circuits and B-formulas lead to the same $\leq_m^p$-degree of complexity are certain membership problems in Post's lattice, as shown by Elmar Böhler and Henning Schnoor in [BS05], and various satisfiability problems in the context of modal logics as shown by Michael Bauland, Edith Hemaspaandra, Henning Schnoor, and Ilka Schnoor in [BHSS06]. We will see in Chapter 3 that this holds when considering the enumeration problem for Boolean formulas and circuits.

We formally define the formula value problem as follows: Let D be a finite set, and

let B be a finite set of functions on D.

Problem: $\mathsf{VAL}^{\mathrm{F}}(B)$
Input: A variable-free B-formula φ and an element $\alpha \in D$
Question: Does φ evaluate to α?

For Boolean domains, this question can be stated more naturally as "does φ evaluate to true," i.e., the input for the problem in the Boolean case consists only of a formula, without the additional value α.

When considering this problem as defined above, a decision algorithm for a problem $\mathsf{VAL}^{\mathrm{F}}(B)$ not only has to evaluate the given formula, but has to ensure its syntactical correctness as well. For formulas written as strings, this involves counting the number of parentheses, and verifying that each function gets the appropriate number of arguments. This problem is harder to solve than the evaluation problem for some formulas itself. Hence, we consider the formula value problem as a *promise problem*: in our algorithms, we assume that the input is syntactically correct. For syntactically incorrect inputs, the output of the algorithm is not required to be correct. For similar reasons, for the problem $\mathsf{VAL}^{\mathrm{F}}(B)$, we assume that all the functions in B only have relevant variables. Otherwise, the "relevancy check" for parts of the formula would dominate the complexity of the decision problem. For technical reasons, we assume that the identity is always a member of the set B, and that our Turing machines have a separate symbol for each function in B, and for each $\alpha \in D$.

Hence, for the consideration of this problem, we define variable-free formulas as follows: if $\alpha \in D$, then α is a variable-free B-formula. If $\varphi_1, \ldots, \varphi_n$ are variable-free B-formulas and $f \in B$ is an n-ary function, then $f\varphi_1 \ldots \varphi_n$ is a variable-free B-formula. The value of such a formula is defined in the canonical sense: the value of $\alpha \in D$ is α itself, and if $\alpha_1, \ldots, \alpha_n$ are the values of $\varphi_1, \ldots \varphi_n$, then the value of $f\varphi_1 \ldots \varphi_n$ is defined as $f(\alpha_1, \ldots, \alpha_n)$. Note that in formulas defined in this way, no parentheses occur. Since we are dealing with formulas in prefix notation as opposed to infix notation, this does not give any ambiguity.

2.2 Logarithmic Time

Turing machines operating in logarithmic time obviously cannot read the entire input. Therefore, in order to define classes of problems solvable in logarithmic time, we need to consider special access modes for the input. There are several definitions in the literature, the following are taken from [RV97]. A deterministic logtime Turing Machine has access to the input x via an index tape, on which it writes a number j, enters a query state, and receives the j-th character of the input string, at cost 1. We assume that the index tape is not deleted after the query. The class LOGTIME contains all decision problems which can be solved by such a machine in logarithmic time. A stronger restriction is that we demand that the Turing machine may only read one character of its input. Problems which can be solved by deterministic logtime machines with this additional restriction form the class $\Delta_0^{\mathcal{R}}$. It is obvious that these classes are in fact different: the problem to determine, for a string from $\{0,1\}^*$, if it starts with a 1 and ends with a 1, can be solved in LOGTIME by simply checking these two positions in the string, but it obviously cannot

be solved by only looking at one character of the input. If we allow the machine to be nondeterministic, but still demand that it only makes one access to the input tape, then the class of languages decidable by these machines is the class $\Sigma_1^{\mathcal{R}}$, and the complements of languages in this class make up the class $\Pi_1^{\mathcal{R}}$.

It is obvious that the classes $\Delta_0^{\mathcal{R}}$ and $\Sigma_1^{\mathcal{R}}$ differ: it is easy to see that the language containing all strings over the alphabet $\{0,1\}$ which contain at least one occurrence of 1 can be recognized in $\Sigma_1^{\mathcal{R}}$, but it cannot be recognized in $\Delta_0^{\mathcal{R}}$, since for the strings 0^n1 and $0^{n-1}10$, the deterministic machine would need to read the same bit from the input, and therefore would miss the occurrence of the 1 for at least one of the two words. With a similar argument, it can be seen that this problem cannot be solved in LOGTIME.

Another way to define the access of a logtime machine to the input tape is *block read/write*: In this model, M can write two addresses $i \leq j$ on the index tape, and it then receives the string consisting of bit i to bit j of the input, at cost $\log|x| + (j - i)$.

For these very low complexity classes, we also need an appropriate notion of reduction. We use two reduction types here: first, we consider logtime-uniform projections, as introduced in [RV97], which were defined to formulate the "sharpest practical notion of reducibility." For our purposes, the interesting properties of this reduction are that it is transitive, closes our relevant complexity classes and contains the idea of a "finite replacement reduction." The formal definition, taken from [RV97], is as follows:

Definition Let A and B be languages, and f be a many-one reduction from A to B such that $|f(x)|$ is polynomial in $|x|$.

1. A reduces to B via a *deterministic logtime reduction* if there is a deterministic logtime Turing machine M that computes f in the following way: On input x and auxiliary input $|x|$ and j, M outputs $|f(x)|$ and the jth bit of $f(x)$. If additionally M only makes one query to the input, then the reduction is a *Ruzzo reduction*.
2. A reduces to B via a *DLT reduction* if there is a deterministic logtime Turing Machine M that works in block read/write mode and on input x and auxiliary input $|x|, i, j$ with $j - i = O(\log|x|)$, outputs bits i to j of $f(x)$ together with $|f(x)|$.

These reducibility concepts have different properties which are usually required from a reduction. The Ruzzo reduction is transitive, but not all relevant classes are closed under this reduction. The DLT reduction is not transitive, but closes our classes. Since we want both of these properties, we define that A reduces to B via a *deterministic logtime projection*, $A \leq_{\text{proj}}^{\text{dlt}} B$, if A reduces to B via a reduction function f which is both a Ruzzo and a DLT reduction. This reduction is both transitive and closes our complexity classes.

For a detailed explanation of these concepts, see [RV97]. Note that all of the reductions appearing in this chapter, with the exception of the reduction in Lemma 2.4.7, can also be computed by a Mealy automaton. Hence, the reductions in this chapter can basically be computed by almost any reduction type which has practical applications at all. The reduction in the lemma just mentioned does not work with the strict resource bounds we require for the $\leq_{\text{proj}}^{\text{dlt}}$-reductions. Here, we use deterministic logtime reductions as defined above. Finally, another degree of complexity which arises in the classification

of the formula value problem is the following:

Problem:	MOD_2
Input:	Boolean values $\alpha_1, \ldots, \alpha_n$
Question:	Is $\sum_{i=1}^{n} \alpha_i$ an even number?

This problem is essentially the computation of the n-ary parity function. This function plays a crucial role in circuit complexity theory, as for this function, it can be shown that is is in the class NC^1, but cannot be computed by AC^0-circuits (see [Smo87]). It naturally arises in the complexity classification of the formula value problem.

2.3 Tools

The first result in this context is an easy observation: Since in our formulas, constants from the domain D are allowed, it does not matter if we have these constants in our base B as well. Hence, we get the following trivial proposition:

Proposition 2.3.1 *Let B be a finite set of functions over a finite domain D. Then $\mathsf{VAL}^F(B \cup D) \equiv^{dlt}_{proj} \mathsf{VAL}^F(B)$.*

Proof. The reduction from $\mathsf{VAL}^{\mathrm{F}}(B)$ to $\mathsf{VAL}^{\mathrm{F}}(B \cup D)$ is trivial, since every correct instance for $\mathsf{VAL}^{\mathrm{F}}(B)$ also is a correct instance for $\mathsf{VAL}^{\mathrm{F}}(B \cup D)$. For the other direction, we replace every function symbol for a constant function with the corresponding constant from D. Note that if we use the same symbol for both the constant function "computing" the value α and the value $\alpha \in D$ itself, then this reduction is in fact the identity. □

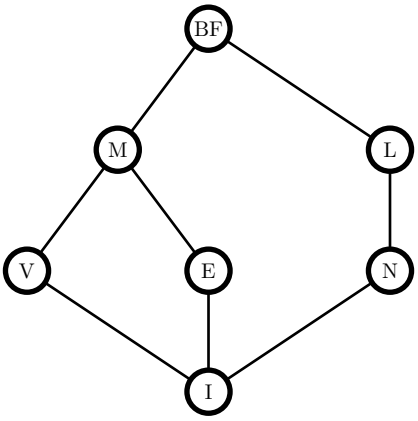

Figure 2.1: The Boolean clones containing the constants

For the Boolean case, Proposition 2.3.1 simplifies the task of determining the complexity of the problem $\mathsf{VAL}^{\mathrm{F}}(B)$ for all possible sets B significantly: there is an infinite set of problems of the form $\mathsf{VAL}^{\mathrm{F}}(B)$ to consider, since there are infinitely many sets of Boolean functions. We will later see that the complexity of $\mathsf{VAL}^{\mathrm{F}}(B)$ depends only on the clone generated by B, and Proposition 2.3.1 tells us that we only need to consider those clones which contain both Boolean constants 0 and 1. By definition of I, the clonescontaining both constants are exactly those which are a superset of the clone I, and Figure 1.2 shows that there are only finitely many of these, namely the ones shown in Figure 2.1.

We now exhibit one problem which is complete for the crucial class $\Sigma_1^{\mathcal{R}}$:

Proposition 2.3.2 *Let L be the language of strings over the alphabet $\{0,1\}$ which contain at least one occurrence of 1. Then L is complete for $\Sigma_1^{\mathcal{R}}$ under $\leq^{dlt}_{proj}$-reductions.*

Proof. It is obvious that L is in $\Sigma_1^{\mathcal{R}}$. Hence, let L_1 be some language over an alphabet Σ from this class, and let M be a machine which accepts it in logarithmic time with the restriction that at most one bit of the input string is read.

Then there exists a function $g\colon \mathbb{N} \times \Sigma \to \{0,1\}$ which can be computed in linear deterministic time, such that $g(i,c) = 1$ if and only if the machine M accepts, if it nondeterministically chooses to read the i-th bit of the input, and this bit is the character c (since the length of i is logarithmic in $|x|$).

The reduction function f is defined as follows: For $x \in \{0,1\}^*$, the length of $f(x)$ is the same as the length of x, and the i-th position of $f(x)$ is $g(i, x[i])$, where $x[i]$ denotes the i-th bit of x.

We claim that f is a many-one reduction from L_1 to L. For this, note that x is a word from L_1 if and only if there is some number i, such that if M reads the i-th bit of x, it accepts. By definition, this is equivalent to $g(i, x[i]) = 1$ for some i, and by definition of f, this is equivalent to $f(x) \in L$.

It is easy to see that f can be computed by a $\leq^{\text{dlt}}_{\text{proj}}$-reduction, since it captures the idea of a local replacement. From the definition, it is obvious that f can be computed by a Ruzzo reduction.

It also can be computed by a DLT reduction, since in order to compute bits i to j of $f(x)$, exactly the bits i to j of x are needed, and g can be computed in logarithmic time.

Again, since the length of the binary representations i and j are logarithmic in the length of x, this implies that g can be computed in time logarithmic in the length of x. □

2.4 Classification

We now give concrete complexity results for the formula value problem. To prove the general upper bound, we will briefly leave the Boolean domain for technical reasons. After that, we will consider the relevant Boolean clones and give complexity results for these cases.

2.4.1 General Upper Bound

The first complexity result here is the general upper bound: We show that for any finite set B of Boolean functions, the problem $\mathsf{VAL}^{\mathrm{F}}(B)$ can be solved in $\mathrm{ALOGTIME} = \mathrm{NC}^1$. The identity of these classes has been shown by Walter Ruzzo in [Ruz81]. The formula evaluation result has been proven by Samuel Buss in [Bus87] for Boolean formulas using only the operators $\wedge$, $\vee$, and $\neg$, which are presented in infix notation. This formulation does not give an easy way to generalize it to non-binary operators. But a more general result has been proven by Martin Beaudry and Pierre McKenzie in [BM95]: they consider formulas in infix notation, and hence their results are restricted to binary operations. But since their theorem handles arbitrary domains, we can "exchange domain size for arity:" starting with our Boolean formulas, in which operations of arbitrary arity appear, we construct formulas over a larger domain, but using only binary operations. This can

be formalized as follows. In the following lemma, $\mathrm{ar}(B)$ denotes the maximal arity of a function in B.

Lemma 2.4.1 *Let D be a finite domain, and let B be a finite set of finitary functions over D, such that $ar(B) \geq 3$. Then $\mathsf{VAL}^F(B)$ reduces to $\mathsf{VAL}^F(B')$ for a finite set B' of functions over a domain D', where $|D'| = |D|^2 + |D|$, $ar(B') = ar(B) - 1$, and $|B'| = |B| + 1$. The reduction can be computed in* NC^1.

Proof. We define the new domain D' as $D \cup (D \times D)$ (without loss of generality, we assume that D and $D \times D$ are disjoint sets). Now we define a new binary operation $*$ on D'. For $\alpha_1, \alpha_2 \in D$, we define $\alpha_1 * \alpha_2 =_{\mathrm{def}} (\alpha_1, \alpha_2)$. For arguments not of this form, $*$ is defined in an arbitrary way.

Now for any function $f \in B$ such that $n =_{\mathrm{def}} \mathrm{ar}(f) \geq 3$, define a new function f' of arity $n-1$ as follows: For arguments $\alpha_1, \ldots, \alpha_{n-1}$, if $\alpha_1 = (\beta_1, \beta_2)$ for some $\beta_1, \beta_2 \in D$, and $\alpha_2, \ldots, \alpha_{n-1}$, define $f'(\alpha_1, \ldots, \alpha_{n-1}) =_{\mathrm{def}} f(\beta_1, \beta_2, \alpha_3, \ldots, \alpha_4)$.

Now given a variable-free B-formula, we replace any occurrence of $ff_1f_2 \ldots f_n$ with $f' * f_1f_2 \ldots f_n$. The resulting formula is obviously equivalent to the original, and the construction can be performed in NC^1. □

Using this transformation allows us to apply the result from [BM95] to show our desired general upper bound. The main task which remains is to convert our formulas from prefix notation to infix notation. In order for the formulas to still give a well-defined value, we need to introduce parentheses to remove ambiguity. Note that the result in [BM95] is much more general than we need it for this context: they prove that even with formulas in infix notation with "missing" parentheses, it can still be decided in NC^1 if there is a possible way to evaluate the formula so that the result is a given constant.

Theorem 2.4.2 *Let B be a finite set of Boolean functions. Then $\mathsf{VAL}^F(B) \in \mathrm{NC}^1$.*

Proof. Let $k =_{\mathrm{def}} \mathrm{ar}(B)$. If $k > 2$, then by applying Lemma 2.4.1 $k-2$ times, we can reduce the problem to a problem having only at most binary operations. Since B does not depend on the input, this conversion can be performed by $k-2$ subsequent NC^1-computable steps, and hence the entire construction can be performed in NC^1. Similar to the proof of Proposition 3.1 in [BM95], we can reduce the problem to the case where B consists of a single binary operator.

Theorem 3.4 from [BM95] states that the evaluation of a formula in infix notation can be performed in NC^1. Hence, to obtain our desired result, it suffices to convert a given B-formula φ which is given in prefix notation into an infix representation. This representation exists, since due to the above we can assume that B only contains one binary operator, which we will call f.

For the conversion, for two subformulas φ_1 and φ_2, we convert the term $\psi = f\varphi_1\varphi_2$ to $(\varphi_1' f \varphi_2')$, where φ_1' and φ_2' are the infix representations of φ_1 and φ_2. For any subformula ψ, let $\mathrm{ext}(\psi)$ denote the length of its infix representation. Then it is easy to see that $\mathrm{ext}(\psi) = |\psi| + 2 \cdot \psi_f$, where ψ_f is the number of function symbols f appearing in ψ_f. The equality holds due to the added parentheses in the conversion.

The main operation required to compute the infix representation φ' of a given variable-free formula φ is to determine the place of the function symbols f, since the order of the constant symbols is preserved in the conversion.

To determine the position of f in the infix representation of ψ, we mainly need to compute the position of the last character of φ_1, which basically means counting to determine the arguments for each binary operation: we set a counter to 2 and increment it by 1 whenever we find a binary operation symbol, and decrease if by 1 whenever we find a constant. This can be done in parallel for each function symbol appearing in the formula.

Since counting can be done in NC^1, this can be performed in NC^1. Now the position of the symbol f in the infix representation φ' of φ is its position in φ plus $\mathrm{ext}(\varphi_1)$ plus the number of parentheses introduced in the part of φ' before the occurrence of φ'_1. This can be determined in a similar way by counting operation symbols and checking if their range extends beyond the position of f. If it does, then add 1 for the opening parenthesis, if it does not, add 2 to take into account both the opening and the closing parenthesis.

To determine this, the range of an operation symbol appearing in φ has to be checked, this can be done in the same way as outlined above.

Hence, we can compute this conversion in NC^1, and thus we get the desired result. □

2.4.2 Results for Individual Clones

As mentioned in Section 1.4, the symmetry in Post's lattice often gives a corresponding symmetry for the resulting complexities of the problem at hand. The Formula Value Problem is an example where this is true:

Proposition 2.4.3 *Let B be a finite set of Boolean functions. Then $\mathsf{VAL}^F(B)$ reduces to $\overline{\mathsf{VAL}^F(\mathrm{dual}(B))}$ with a $\leq^{dlt}_{proj}$-reduction.*

Proof. Let φ be a variable-free B-formula. We compute a $\mathrm{dual}(B)$-formula by replacing every function symbol f in φ with the symbol for the function $\mathrm{dual}(f)$. This includes exchanging the symbol for the constant 0 with the symbol for the constant 1, and vice versa. We call the resulting formula φ'. This transformation can obviously be performed by a $\leq^{\mathrm{dlt}}_{\mathrm{proj}}$-reduction, since this is just a character replacement. To prove the proposition it suffices to show that φ is true if and only if φ' is false. We show this claim by induction over the formula construction.

If φ is a constant, then this is true by definition. Now let $\varphi = f\varphi_1 \dots \varphi_n$ for an n-ary function f and formulas $\varphi_1, \dots, \varphi_n$. It is obvious by definition that $\varphi' = \mathrm{dual}(f)\,\varphi'_1 \dots \varphi'_n$, where the φ'_i denote the formulas resulting from the transformation applied to the φ_i. By induction, we know that for all relevant i, φ'_i evaluates to the negation of the truth value of φ_i. Let $\alpha_1, \dots, \alpha_n$ denote the truth values of $\varphi_1, \dots, \varphi_n$. Then the truth value of φ is simply $f(\alpha_1, \dots, \alpha_n)$. Due to the above, it follows that the truth value of φ' is $\mathrm{dual}(f)(\overline{\alpha_1}, \dots, \overline{\alpha_1})$, which due to the definition of the $\mathrm{dual}(.)$ operator is the negation of the truth value of φ. □

We now look at the individual clones that are interesting for our classification, i.e., the clones which contain both Boolean constants 0 and 1. We start with the simplest problem in this context, which naturally corresponds to the smallest of our relevant clones.

Lemma 2.4.4 *Let B be a finite set of Boolean functions such that $[B] = \mathrm{I}$. Then $\mathsf{VAL}^F(B) \in \Delta_0^{\mathcal{R}}$.*

Proof. Due to the prerequisites, B only contains symbols for the identity, and constants. Since we assume our sets B only contain functions which have no irrelevant variable, this means that all function symbols from B are either unary or nullary. Hence, a syntactically correct B-formula is of the form $f_1 \dots f_n c$, where the f_i are symbols for the identity function, and c is a symbol representing either 0 or 1.

To determine if such a formula is true, it is sufficient to read its very last character. This can be performed in $\Delta_0^{\mathcal{R}}$. □

Note that in the previous lemma, without the assumption that B only contains functions which depend on all of their variables, the complexity of the problem rises. Instead of looking at the last character only, the evaluation procedure has to follow a "relevant path" to evaluate the formula. Hence, our assumption is not a mere convenience, but does have an impact on our results. However, we believe that this version of the problem is more natural: in a concrete context, if we want to express some facts using propositional formulas, we are usually interested in a succinct representation. To use functions with irrelevant variables only adds redundancy, and this will not be used in practice.

The key reason why the above problem is so very simple is that the involved functions are trivial, and that since they are unary, we know where to look for the argument for their concatenation. It is natural that the complexity rises when we allow binary functions.

Lemma 2.4.5 *Let B be a finite set of Boolean functions. If $[B] = \mathrm{V}$, then $\mathsf{VAL}^F(B)$ is complete for $\Sigma_1^{\mathcal{R}}$ under $\leq_{proj}^{dlt}$-reductions. If $[B] = \mathrm{E}$, then $\mathsf{VAL}^F(B)$ is complete for $\Pi_1^{\mathcal{R}}$ under $\leq_{proj}^{dlt}$-reductions.*

Proof. We show the claim for the case $[B] = \mathrm{V}$, the case $[B] = \mathrm{E}$ follows from Proposition 2.4.3. Hence, let $[B] = \mathrm{V}$. It is obvious that $\mathsf{VAL}^{\mathrm{F}}(B) \in \Sigma_1^{\mathcal{R}}$, since a B-formula evaluates to true if and only if at least one of the constants in the formula is 1: This condition is obviously necessary, and since we do not allow irrelevant variables in our functions in B, the functions in B are all disjunctions of various arities. Hence, the condition is sufficient as well.

It remains to show that the problem is hard for $\Sigma_1^{\mathcal{R}}$ under $\leq_{\mathrm{proj}}^{\mathrm{dlt}}$-reductions. Let L denote the language of strings over the alphabet $\{0,1\}$ containing at least one occurrence of 1. This problem is hard for $\Sigma_1^{\mathcal{R}}$ due to Proposition 2.3.2. Due to Lemma 1.4.5, there exists a B-formula ψ in which the variables x and y, and possibly other variables $z_1, \dots, z_k$ appear, and both x and y appear exactly once, and $\psi(x, y, z_1, \dots, z_k)$ is equivalent to $x \vee y$. Let $\psi = \psi^B x \psi^M y \psi^E$ (as a string). Since all functions from B are commutative, we can swap arguments and achieve that ψ^E is empty. Let $c_1 \dots c_n$ be some string from $\{0,1\}^*$. Now define φ as the formula $\psi^B c_1 \psi^M \psi^B c_2 \psi^M \psi^B c_3 \dots \psi^B c_{n-1} \psi^M c_n$. Since the strings ψ^B and ψ^M are independent of the input, this can be performed by a uniform logtime projection. Since ψ represents the OR-operator, it is obvious that the formula φ constructed in this way evaluates to 1 if and only if at least one of the characters $c_1, \dots, c_n$ is 1. □

The problems involving just AND and OR operators covered in the above lemma are easy, because here one value "dominates" the entire truth evaluation of the formula in the following sense: in the case of OR, if there is at least one 1 appearing in the formula, then the formula evaluates to 1. In the case of AND, the presence of at least one zero ensures that the formula is false. It is natural that the formula value problem gets more difficult if for the formula evaluation, every symbol appearing in the function must be taken into account. This happens in the following case: in the case where we only have negations and identities, the number of negation symbols in the formula determines the truth value together with the single constant appearing. In the case where the binary exclusive-or is present, more constant symbols can appear, and each of them needs to be considered. The formula value problem for these cases is still relatively easy, since the functions in the considered clone are symmetric in all of their arguments. Hence, the complexity does rise, but due to the symmetry, the problem is still easier than the general formula evaluation problem. It should be noted that since it has been proven that the problem MOD_2 cannot be performed in AC^0 (this follows from Roman Smolensky's famous theorem [Smo87]), and $\Sigma_1^{\mathcal{R}}$ very trivially is subclass of AC^0, the following lemma gives a provably harder complexity bound than the one given in Lemma 2.4.5.

Lemma 2.4.6 *Let B be a finite set of Boolean functions such that $[B] = \mathrm{N}$ or $[B] = \mathrm{L}$. Then $\mathsf{VAL}^F(B)$ is equivalent to MOD_2 under $\leq^{dlt}_{proj}$-reductions.*

Proof. In both cases, Lemma 1.4.5 implies that there exists a B-formula ψ_{neg} in which the variable x appears exactly once as the last character in the formula, and $\psi_{\mathrm{neg}}(x)$ is equivalent to $\overline{x}$. We can, without loss of generality, assume that x is the only variable appearing in ψ_{neg} : other appearing variables can be replaced by constants from $\{0,1\}$, and since these variables are irrelevant due to the choice of ψ_{neg}, this transformation does not change the function represented by the formula. Now, let $c_1 \dots c_n$ be an instance of MOD_2, i.e., $c_1 \dots c_n$ is a string over the alphabet $\{0,1\}$. For the reduction, we construct a B-formula as follows: Define

$$\varphi =_{\mathrm{def}} \psi_1 \dots \psi_n 0, \text{ where } \psi_i =_{\mathrm{def}} \begin{cases} \psi_{\mathrm{id}}, & \text{if } c_i = 0, \\ \psi_{\mathrm{neg}}, & \text{if } c_i = 1. \end{cases}$$

Here ψ_{id} denotes a sequence of as many identity symbols as there are symbols in the formula ψ_{neg}. This ensures that the starting positions of the occurrences of ψ_{id} and ψ_{neg} do not depend on whether the previously occurring symbols denote negation or the identity. Therefore, the reduction again consists of a finite replacement table, and hence can be computed by a $\leq^{\mathrm{dlt}}_{\mathrm{proj}}$-reduction. It is obvious that the formula φ evaluates to 1 if and only if the number of occurrences of 1 in the string $c_1 \dots c_n$ is odd.

Now for the other direction, we show that $\mathsf{VAL}^{\mathrm{F}}(B)$ reduces to MOD_2. For this, consider that each n-ary function $f \in B$ is equivalent to $c \oplus x_{i_1} \oplus \dots \oplus x_{i_n}$, and all functions in B are associative. Let φ be a variable-free B-formula. A reduction can be performed by simply inserting a 0 for every constant 0 or variable value 0 appearing in the formula, and similarly a 1 for every 1 appearing either directly in the formula or indirectly in the function from B.

To be precise: for each occurrence of a function symbol f, write a 0 or a 1 corresponding to the constant c in the formula. Simply copy any symbols for constants appearing in φ. This obviously is a correct reduction which can be computed by a deterministic logtime projection. □

We now consider the case where we have both AND and OR functions present in our base B. Here, we cannot neglect most of the arguments, as was possible in Lemma 2.4.5, and there are no symmetries that we can use like in Lemma 2.4.6. In this case, the lower complexity bound for the problem matches the upper bound proven in Theorem 2.4.2: we get a NC^1-completeness result. Hence the formula value problem is one example where the monotone property of the involved functions does not give an easier problem, in contrast to the results for the satisfiability problem (see Theorem 1.4.3) and the enumeration problem for propositional formulas, which we consider in Chapter 3. The proof for the following Lemma uses the idea already exhibited in [Bus87] to express alternating Turing machines as monotone Boolean formulas. In our situation, the proof becomes more difficult technically, since we cannot use infix notation due to our more general setting.

Lemma 2.4.7 *Let B be a finite set of Boolean functions with $[B] \in \{\mathrm{M}, \mathrm{BF}\}$. Then $\mathsf{VAL}^F(B)$ is complete for NC^1 under deterministic log time reductions.*

Proof. The upper complexity bound follows from Theorem 2.4.2. With Lemma 1.4.5 or Lemma 1.4.4, we obtain formulas $f_\vee(x, y)$ and $f_\wedge(x, y)$ for $x \vee y$ and $x \wedge y$ with each of the variables occurring only once. By padding the formulas with symbols for the identity, we can achieve that these formulas are of the following form:

$f_\vee = f_\vee^B\, x\, f_\vee^M\, f_\vee^N\, y\, f_\vee^E$, and $f_\wedge = f_\wedge^B\, x\, f_\wedge^M\, f_\wedge^N\, y\, f_\wedge^E$ and it holds that $|f_\vee^E| = |f_\vee^M| = |f_\vee^N| = |f_\vee^E| = |f_\wedge^B| = |f_\wedge^M| = |f_\wedge^M| = |f_\wedge^E|$.

In the same way, we can get formulas f_0 and f_1 representing the constants that are split up into these parts, but without occurrences of x and y. We also assume that the formula parts have a length of some power of 2, and consider them as of length 1 for the rest of the proof—since division by and multiplication with a power of 2 is easy, the position calculation in the formula we want to construct can be performed by a logtime machine. Note that these formulas are not necessarily syntactically correct, since the symbol for the unary identity can appear as the last character in the formula. However, to avoid this problem we just have to ensure that in the complete formula we construct for the reduction, the last symbol is different from the identity.

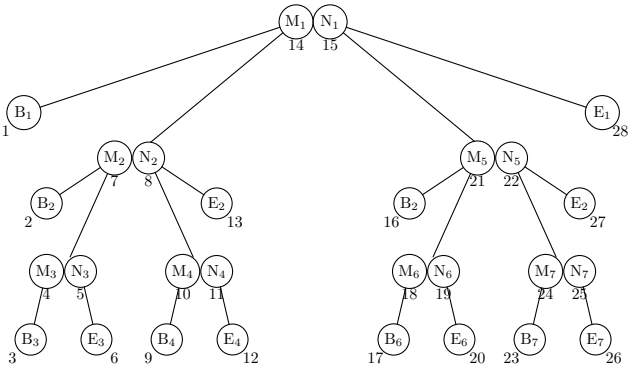

Figure 2.2: Example tree with height 3

The class NC^1 is the same as the class ALOGTIME, which is the class of languages recognized by an alternating Turing machine in logarithmic time. Let L be a language in ALOGTIME, and let M be an alternating Turing machine accepting L in logarithmic

time. Without loss of generality, assume that M branches binarily in every non-halting configuration. The execution tree of the machine M on a given input x directly corresponds to a Boolean formula using only AND (universal configurations), OR (existential configurations) and constants (halting configurations). The formula we want to construct is in prefix notation, which corresponds to a tree-walk on the configurations of M using depth-first search. More precisely: The root of this tree consists of nodes M_1 and N_1 representing the two middle parts of the outmost formula. It has four children: The leftmost is a node B_1 representing the beginning of the formula, the second one, φ_1 represents the first argument, the third child φ_2 is the second argument and finally the right-most child, E_1, represents the ending part of the formula. The order of the tree-walk is the same as the order in which these parts of the formula appear when written out: $\mathrm{B}_1\varphi_1\mathrm{M}_1\mathrm{N}_1\varphi_2\mathrm{E}_1$. The formulas φ_1 and φ_2 represent subtrees constructed in the same manner. Figure 2.2 is an example for such a tree with height 3. Here B_1 refers to the beginning part of the first (outmost) formula, this is $f_\vee^B$ if the first configuration of the alternating machine is existential, and $f_\wedge^B$ if it is universal, M_1 and N_1 refer to the two middle parts and E_1 to the end part.

The numbers 1 to 28 indicate the order in which the parts appear in the formula. It is obvious that the formula constructed in this way is true if and only if x is an element of L. For a canonical logtime reduction, it would be necessary to compute, on input x and n, the n-th bit of the formula. The main problem to solve here is to determine how the node n is labeled in the tree outlined above. The algorithm presented in Figure 2.3 calculates, on input h (height of the tree) and n, a *search string* to the node n as follows: Observe that the tree is "essentially binary", that is, every vertex has at most two subtrees which are not just single vertices. Now a search string to a node $\mathrm{B_i}, \mathrm{M_i}, \mathrm{N_i}$ or $\mathrm{E_i}$ consists of a word from $\{l, r\}^*$, denoting a left/right path to the corresponding $\mathrm{M_i}$ node, plus an indicator $\mathrm{B}, \mathrm{M}, \mathrm{N}$ or E, to determine which of these nodes is required. Given such a search string, a deterministic Turing Machine can simulate the alternating machine recognizing the language L and make the non-deterministic choices according to the search string, and thus determine whether this is an existential, universal, accepting or rejecting configuration. Based on the indicator, it is easy to look up the appropriate bit of the corresponding formula part.

A tree of the form outlined above with height h (the height is $\log|x|$) has $2^{h+2} - 4$ vertices. The node M_1 has the number $2^{h+1} - 2$, E_1 has number $2^{h+2} - 4$. The first B-node of the left child has number 2, for the right subtree, this number is 2^{h+1}. Given these numbers, it is easy to see that the algorithm in Figure 2.3 correctly computes the search string for a given node: In the conditions adding a "r" or a "l" to the string, the new number n is the number of the required node when considering the right or left subtree as an independent tree (the numbers added or subtracted to n correspond to the difference between the indexes of the B nodes of the three subtrees involved).

The algorithm does not run in logarithmic time, but we can work around that: Since we have the identity in our language, we can insert these without altering the value of the resulting formula. This corresponds to "leaving out" some numbers in the enumeration, and thus we can reduce the search string calculation to a search string verification as follows: On input (n, s, x), we put out the corresponding bit of the node described by s on input x, if the string s describes the way to node n in the tree, and the identity

symbol otherwise. Since there are $O(|x|)$ nodes in the tree, the binary length of n is $\log|x|$, the same holds for s. The resulting formula contains all formulas representing the configurations in the correct order, plus identity symbols. As mentioned above, we must assure that the final character of the generated formula is not the identity symbol. Therefore we must "switch" the very last character with the last "relevant" character, i.e., instead of writing the last symbol of E_1, we write in identity symbol, and as the very last character, we write the last symbol of E_1. This can easily be done, since we do not have to perform the verification in this case: Whenever the search string leads to the last symbol of E_1, write an identity, unless the input is $1\ldots1$, the very last symbol. The verification needs to be performed in time $O(\log|x|)$.

We proceed as follows: On tape one, we keep a copy of the search string. On tape two, we keep track of what we added to the number n, on tapes three and four we keep track of what we subtracted from n. On tape five, we keep the number h, which is initially set to $\log|x|$. For each symbol of the search string, we perform the additions and subtractions on n and h that the algorithm would have done when writing the symbol. This essentially requires adding the constant 1 to tapes two and three, and setting the leftmost bit of tape 4 to 1 (The statement $N =_{\text{def}} n-2^{h+1}+1$ is split up into an addition and a subtraction). After processing each symbol, tape 5 is decremented by 1. By standard amortized running time analysis, this can be done in time $O(\log n)$. When the search string is processed, we add to n the content of tape 2 and subtract tapes 3 and 4, which can be done in time $O(\log n)$ again. Finally, the conditions whether to put out M,N,E, or B can be verified in time $O(|n|)$, since these are easy patterns in binary representation. The length of the resulting formula is $O(2^{|n|+|s|}) = O(2^{\log n+\log n}) = O(n^2)$, where n is the length of $|x|$, and thus polynomial in $|x|$. Since these formulas have length of more than one bit, we still need to divide by the length, but that can be done easily—just assure the formula length is a power of 2.

Input: Height h and search number n
searchstring $=_{\text{def}} \epsilon$
loop
  if search=1 **then**
    output searchstring+"B"
  else if $n = 2^{h+1} - 2$ **then**
    output searchstring+"M"
  else if $n = 2^{h+1} - 1$ **then**
    output searchstring+"N"
  else if $n = 2^{h+2} - 4$ **then**
    output searchstring+"E"
    end.
  else if first bit of n is zero **then**
    // equivalent to $n < 2^{h+1} - 2$
    // after cases above have been considered
    $n =_{\text{def}} n - 1$
    searchstring $=_{\text{def}}$ searchstring+"l"
  else if first bit of n is one **then**
    // equivalent to $n > 2^{h+1} - 1$
    // after cases above have been considered
    $n =_{\text{def}} n - 2^{h+1} + 1$
    searchstring $=_{\text{def}}$ searchstring+"r"
  end if
  $h =_{\text{def}} h - 1$
end loop

Figure 2.3: Search Algorithm

□

2.5 Conclusion

With the results in this chapter, we are now in a position to prove a complete classification of the formula value problem for the Boolean case. Due to Proposition 2.3.1, we only need to consider the clones in which both constants appear, and hence, the following theorem gives the complete classification:

Theorem 2.5.1 *Let B be a finite set of Boolean functions. Then the following complexity classification holds:*

- *If* $[B \cup \{0,1\}] = \mathrm{I}$, *then* $\mathsf{VAL}^F(B) \in \Delta_0^{\mathcal{R}}$.
- *If* $[B \cup \{0,1\}] = \mathrm{V}$, *then* $\mathsf{VAL}^F(B)$ *is complete for* $\Sigma_1^{\mathcal{R}}$ *under* $\leq_{proj}^{dlt}$*-reductions.*
- *If* $[B \cup \{0,1\}] = \mathrm{E}$, *then* $\mathsf{VAL}^F(B)$ *is complete for* $\Pi_1^{\mathcal{R}}$ *under* $\leq_{proj}^{dlt}$*-reductions.*
- *If* $[B \cup \{0,1\}] \in \{\mathrm{N}, \mathrm{L}\}$, *then* $\mathsf{VAL}^F(B)$ *is equivalent to the problem* MOD_2 *under* $\leq_{proj}^{dlt}$*-reductions.*
- *Otherwise,* $\mathsf{VAL}^F(B)$ *is complete for* NC^1 *under deterministic log time reductions.*

Proof. In the cases not covered in the first four cases, Figure 2.1 shows that the set $B \cup \{0,1\}$ generates a clone which is either M or BF. The complexity classification for the individual cases then follows from the results previously shown in this chapter.

Since due to Proposition 2.3.1, the problems $\mathsf{VAL}^{\mathrm{F}}(B)$ and $\mathsf{VAL}^{\mathrm{F}}(B \cup \{0,1\})$ have the same complexity, this finishes the proof of the theorem. □

It is worth noting that although Theorem 2.5.1 gives a classification where the complexity of the problem $\mathsf{VAL}^{\mathrm{F}}(B)$ depends only on the clone generated by B, there is no "uniform" proof showing that if $[B_1] = [B_2]$, then $\mathsf{VAL}^{\mathrm{F}}(B_1) \equiv_{\mathrm{proj}}^{\mathrm{dlt}} \mathsf{VAL}^{\mathrm{F}}(B_2)$. Rather, this follows from the proofs for the individual cases in this chapter, and almost all of the individual result make use of the fact that for all relevant cases, Lemmas 1.4.4 and 1.4.5 guarantee the existence of short formulas to express the necessary Boolean connectives. This is a significant difference between B-formulas and B-circuits, since for circuits, such a uniform proof can easily be obtained by simple gate replacement. In the constraint context, uniform proofs often easily follow from an application of the Galois connection, Theorem 1.5.3. We will see examples of such results in Chapters 4 and 5.

We have seen that the formula value problem can be parametrized by restricting the occurring propositional operators, and that the complexity of the problem depends only on the clone generated by the allowed operators. However, to achieve this result we needed to make several assumptions. Hence, this problem shows that when considering very low complexity classes, like the classes below logarithmic time, it gets difficult to deal with "natural" problems. We also needed to use a less strict reduction, namely the deterministic logtime reduction to show our NC^1 hardness result for the case where the set $B \cup \{0,1\}$ generates one of the clones M and BF. However, the assumptions that we needed to make are not too unreasonable: syntactical correctness of formulas can usually be guaranteed in a practical setting, where the occurring formulas are typically generated by an algorithm. The restriction that the functions appearing in the set B do not have any non-relevant variables also can be seen as a natural one, as explained above. The

final assumption, that B contains a symbol for the identity, is similar to the condition of "neutral characters" which is sometimes used in formal language theory. The application of the less strict reductions for our NC^1 completeness result also can be justified, since NC^1 is a lot stronger in computational power than the logtime classes considered here, and these reductions are still strict compared to the resources of NC^1 (typically, AC^0-reductions are used when comparing the complexity of problems inside NC^1). Therefore, our results are still meaningful, but also demonstrate that when dealing with natural problems, the usefulness of complexity classes as low as logarithmic time is limited.

Still, the general result that formula evaluation can be performed in NC^1 has interesting practical applications, since this means that there is an efficient parallel algorithm for this problem. As mentioned in the introduction, our classification immediately implies complexity results for the satisfiability problem for formulas as well: since for any set B of Boolean formulas, the formula value problem can be computed in NC^1, this implies that the satisfiability problem for monotone formulas can also be decided in this class, since in order to decide this, it suffices to test if the given formula is satisfied by the all-1-assignment. With the hardness result from our work, it also follows that the satisfiability test for monotone formulas is NC^1-complete, if we allow the constant 0 in the base B (otherwise, such formulas are always satisfiable).

For other sets of Boolean functions, the formula value problem can actually be harder than the satisfiability problem. For example, let B be a finite set of Boolean functions such that $[B] = \mathrm{R}_1$. Then the formula value problem for B-formulas is complete for NC^1, since adding both constants to B gives a set which generates the clone BF. However, the satisfiability problem for B-formulas is trivial, since every B-formula is satisfied by the all-1-assignment. In this case, it is also trivial to compute one single solution, since an algorithm just needs to print out the all-1-assignment. For another example, consider a set B such that $[B] = \mathrm{D}$. Again, the satisfiability problem is trivial, since every B-formula is satisfiable. But the formula value problem again is NC^1-complete. In this case, we cannot easily generate a satisfying solution, since we just know that for each assignment I to the variables in a B-formula φ, at least one of I and its negation satisfies φ. But to decide which of them does, and print a solution, a formula value test has to be performed.

In the next chapter, we consider questions like these, where we not only look for one satisfying assignment of a formula, but study the problem to generate *all* solutions of a given B-formula.

Chapter 3

Enumerating all Solutions For Propositional Formulas

3.1 Introduction

In Chapter 2, we studied how restricting the allowed propositional operators affects the complexity of the problem to evaluate a variable-free formula. There are several other questions to consider in this context. One of the most famous and important problems in the context of propositional formulas is the satisfiability problem. Theorem 1.4.3 gives a complete classification of this problem with respect to polynomial time many-one reductions. A refinement of this statement to the complexity classes inside P was achieved for the case of Boolean circuits by Steffen Reith and Klaus Wagner in [RW00]. The logical next problem to consider, which is often relevant in practical applications, is the following: given a propositional formula, generate the set of its satisfying solutions. This problem has been considered in the constraint context. For Boolean constraint formulas, a complete classification was achieved by by Nadia Creignou and Jeans-Jaques Hébrard in [CH97]. For non-Boolean domains, the problem was studied by Henning Schnoor and Ilka Schnoor in [SS06a], but a full classification remains open. In this chapter, we study the problem in the B-formula context. It is obvious that in general, this problem cannot be solved efficiently, since we do not even know how to decide if some formula has a solution at all in polynomial time. But in a similar way to Chapter 2, we can show that by restricting the propositional operators allowed in the formulas, we obtain cases in which the problem is significantly easier than for the case where all of the usual connectors $\wedge, \vee$, and $\neg$ are allowed. For restricted classes of formulas, we obtain efficient algorithms for the enumeration problem, and we show that in all other cases, such algorithms cannot exist, unless P = NP.

3.2 Preliminaries

For a formula φ, its set of satisfying assignments can be exponential in the length of φ. Therefore, we cannot enumerate all of its solutions in polynomial time, and thus need to consider other notions of efficiency for this problem. In [JPY88], several possibilities

were considered for enumeration algorithms. We adopt the definitions for the context of enumerating solutions for propositional formulas. In this context, an enumeration algorithm has to perform the following task: given a formula φ, print each of its solution exactly once. Such an algorithm has *polynomial total time*, if the running time of the algorithm is polynomial in the length of the input formula and in the number of its satisfying solutions. It has *polynomial delay* if the time needed by the algorithm between its start and the printing of the first solution, the time between the printing of each two consecutive solutions, and the time between printing the last solution and the termination of the algorithm is bounded by a polynomial in the length of the input formula. It is *lexicographic order*, if it prints the solutions of φ in lexicographic order. As remarked in [JPY88], this condition is only of interest in the polynomial delay case, since with polynomial total time, we can just sort the printed solutions afterwards. Since sorting can be done in polynomial time, every algorithm satisfying the total polynomial time condition can be modified to also give the solutions in lexicographical order. Therefore we require a lexicographic order algorithm to additionally print the solutions with polynomial delay. An *incremental polynomial time* algorithm is not required to generate all solutions in some efficient way, but has to generate one additional solution: given a formula and a set of solutions for it, find another solution or determine that none exists in a time bounded by a polynomial in the entire input. In [JPY88], algorithms requiring only polynomial space were considered as well. Trivially, such an algorithm exists for enumerating solutions of formulas: since formula evaluation can be performed in NC^1 due to Theorem 2.4.2, the obvious "test all assignments and print the satisfying ones" approach can be used to obtain a polynomial space algorithm. Therefore, we disregard polynomial space algorithms in our study.

With the above, lexicographical order implies polynomial delay, which implies polynomial total time.

3.3 Algorithms

We present enumeration algorithms for the various cases here. We first present positive results for the strictest of our enumeration notions.

Theorem 3.3.1 *Let B be a finite set of Boolean functions such that $\mathsf{SAT}(B \cup \{0,1\})$ can be solved in polynomial time. Then B has an efficient lexicographic order enumeration algorithm, and an incremental polynomial time enumeration algorithm.*

Proof. We first show the existence of the lexicographical order algorithm. Let φ be a B-formula, such that $\mathrm{VAR}(\varphi) = \{x_1, \ldots, x_n\}$. We first check if $\varphi[x_1/0]$ is satisfiable, if yes, we recursively print the satisfying solutions of this formula with the additional assignment $x_1 = 0$. We do the same for $\varphi[x_1/1]$. For a formula without variables, we print the empty assignment. Since by the prerequisites, this test can be performed in polynomial time, this approach obviously gives a polynomial delay algorithm printing the solutions in lexicographic order.

For the incremental polynomial time algorithm, we are given a formula φ and a set of solutions for φ. We first sort these solutions, which can be done in polynomial time.

It obviously suffices to show that the following can be done in polynomial time: Given the formula φ and two assignments I_1, I_2, print a solution $J \models \varphi$ such that $I_1 \leq J \leq I_2$, or determine that such a solution does not exist. The lexicographic order enumeration algorithm constructed above can obviously be modified to only print the solutions which are lexicographically larger than a given assignment. Therefore this algorithm can be used to solve the problem at hand in polynomial time. □

A look at Post's lattice and Lewis' Theorem 1.4.3 immediately gives the following corollary:

Corollary 3.3.2 *Let B be a finite set of Boolean functions, such that $B \subseteq \mathrm{M}$ or $B \subseteq \mathrm{L}$. Then B has an efficient lexicographic order enumeration algorithm, and an incremental polynomial time enumeration algorithm.*

Proof. It follows from Theorem 1.4.3 and Figure 1.2 that $\mathsf{SAT}(B \cup \{0,1\}) \in \mathrm{P}$. Hence the result follows from Theorem 3.3.1. □

We now present cases where we do not have efficient lexicographic enumeration algorithms, but the weaker notion of polynomial delay algorithms.

Theorem 3.3.3 *Let B be a finite set of Boolean functions, such that $B \subseteq \mathrm{S}_0^2$, or $B \subseteq \mathrm{D}$. Then B has a polynomial delay enumeration algorithm and an incremental polynomial time enumeration algorithm.*

Proof. We show that for an assignment $I\colon \mathrm{VAR}(\varphi) \to \{0,1\}$, if $I \not\models \varphi$, then $\overline{I} \models \varphi$. This gives a polynomial-delay enumeration algorithm for $\mathrm{SOL}(\varphi)$, by testing the set of all assignments in an appropriate order: if $\mathrm{VAR}(\varphi) = \{x_1, \ldots, x_n\}$, then use an arbitrary order, for example the lexicographical order, on the assignments I with $I(x_1) = 0$, and for each of the assignments considered, test if I or $\overline{I}$ satisfies the formula. In the cases where the answer is "yes," print the corresponding assignment. Due to the above mentioned property, this gives at least one solution for each I considered, since if I is not a solution, then $\overline{I}$ is. Therefore, since it can be verified in polynomial time if a given assignment is a solution for the formula due to Theorem 2.4.2, this clearly gives a polynomial delay algorithm.

We now prove that this property holds for the sets B in question. For the case $B \subseteq \mathrm{D}$, this follows from the definition of the clone D. Therefore, assume that $B \subseteq \mathrm{S}_0^2$. Now assume that $I \not\models \varphi$ and $\overline{I} \not\models \varphi$. Since φ is a B-formula, and $B \subseteq \mathrm{S}_0^2$, we know that the function described by φ is 0-separating of degree 2. Thus every set S with $|S| = 2$ and $S \subseteq \varphi^{-1}(\{0\})$ is 0-separating. The set S defined as $S =_{\mathrm{def}} \left\{(I(x_1), \ldots, I(x_n)), (\overline{I}(x_1), \ldots, \overline{I}(x_n))\right\}$ meets these conditions, and hence is 0-separating. From the definition, it follows that there is some $i \in \{1, \ldots, n\}$ such that $I(x_i) = \overline{I}(x_i)$, which is a contradiction.

For the incremental polynomial time algorithm, assume that we are given the set of solutions in lexicographical order (otherwise, sort it in polynomial time). For each given solution I, test if its negation $\overline{I}$ also satisfies the formula. If yes, print it out and we are finished. Otherwise, find an assignment I such that neither I nor $\overline{I}$ are included in the

given list of solutions. One of the two is a solution for φ due to the above. If no new solution is obtained in this way, then obviously the given set of solutions is the complete set of solutions for φ. It is obvious that this can be performed in polynomial time. □

3.4 Hardness Results

We now show that the list of algorithms presented in Section 3.3 is complete, i.e., that in the cases not covered, algorithms of the corresponding types do not exist. Again, we start with the sharpest notion of efficiency in this context.

Lemma 3.4.1 *Let B be a finite set of Boolean functions such that $\mathrm{S}_{02} \subseteq [B]$ or $\mathrm{D}_1 \subseteq [B]$. Then there is no efficient lexicographic enumeration algorithm for B-formulas, unless* $\mathrm{P} = \mathrm{NP}$.

Proof. We prove that for any set B, if one of these algorithm exists, then $\mathsf{SAT}(B \cup \{0\}) \in \mathrm{P}$. The result then follows with Theorem 1.4.3, since due to Figure 1.2, $[B \cup \{0\}] = \mathrm{BF}$, and therefore this problem is NP-complete.

We show how a polynomial-time decision algorithm for this problem can be obtained from a lexicographic enumeration algorithm for B-formulas. To this end, let φ be a $B \cup \{0\}$-formula, such that $\mathrm{VAR}(\varphi) = \{x_1, \ldots, x_n\}$. Introduce a new variable x_0, and construct the formula $\varphi' =_{\mathrm{def}} \varphi[0/x_0]$. Then φ' is a B-formula. Now enumerate the solutions for φ' in lexicographic order.

We show that φ has a solution if and only if φ' is satisfiable, and the lexicographically first solution of φ' maps x_0 to 0, which clearly finishes the proof, since the lexicographic order enumeration algorithm has to produce the first solution in polynomial time, or determine that none exists.

First assume that φ has a solution I. Then, due to the construction of φ', it is obvious that the solution I' obtained by extending I with the assignment $I(x_0) =_{\mathrm{def}} 0$ is a solution for φ'. In particular, since a solution for φ' exists which sets x_0 to 0, so does the lexicographically minimal solution.

For the other direction, assume that the lexicographically minimal solution I of φ' satisfies $I(x_0) = 0$. Then obviously, the solution I restricted to the variables appearing in φ is a solution for this formula. □

To show non-existence of other types of efficient enumeration algorithms, we introduce a special version of the satisfiability problem:

Problem:	$\mathsf{SAT}^*(B)$
Input:	A B-formula φ
Question:	Does φ have a non-constant solution?

In Section 4.2 of Elmar Böhler's PhD thesis [Böh05], the following hardness result was proven for circuits. We show that it also holds for the more restricted formula case.

Lemma 3.4.2 *Let B be a finite set of Boolean functions such that $\mathrm{S}_{12} \subseteq [B]$. Then $\mathsf{SAT}^*(B)$ is* NP*-complete under $\leq_m^p$-reductions.*

Proof. Obviously, $\mathsf{SAT}^*(B)$ is in NP. For the hardness proof, observe that Post's lattice shows that $[B \cup \{0,1\}] = \mathrm{BF}$. Therefore, $\mathsf{SAT}(B \cup \{0,1\})$ is NP-complete due to Theorem 1.4.3. We show $\mathsf{SAT}(B \cup \{0,1\}) \leq_m^p \mathsf{SAT}^*(B)$.

Let φ be a $B \cup \{0,1\}$-formula. We introduce new variables t and f, to simulate the constants 1 and 0, respectively. Let $\varphi' =_{\mathrm{def}} \varphi[1/t, 0/f]$. Now observe that we can, since $\wedge \in \mathrm{S}_{12}$, using appropriate bracketing and a logarithmic tree construction, construct a B-formula $\psi \equiv x_1 \wedge \cdots \wedge x_n$.

The function $f(x,y,z) =_{\mathrm{def}} x \wedge (y \vee \overline{z})$ is in $S_{12} \subseteq [B]$. Therefore, since the AND-function is in the clone generated by B, and nesting does not appear, we can construct in polynomial time a B-formula equivalent to $\chi \equiv \varphi' \wedge (\psi \vee \overline{f}) \wedge t$.

We claim that φ is satisfiable if and only if χ has a non-constant solution. First, let φ be satisfiable, and let I be a satisfying assignment for φ. By construction of the involved formulas, the assignment I' obtained by enlarging I with the assignments $I'(f) = 0$ and $I'(t) = 1$, is a non-constant satisfying assignment for χ. Therefore, χ has a non-constant solution. Now, let $I \models \chi$, such that I is not constant. We make a case distinction (observe that $I(t) = 1$ must hold). If $I(f) = 0$, then the assignment I' obtained from I by restriction to $\mathrm{VAR}(\varphi)$, is a satisfying assignment of φ, therefore φ is satisfiable. If $I(f) = 1$, then by construction of χ, $I \models \psi$, i.e., $I(x) = 1$ for all $x \in \mathrm{VAR}(\varphi)$. Since $I(t) = I(f) = 1$, this implies that I is the constant 1-assignment, which is a contradiction to the choice of I. □

As our final hardness result in this chapter, we consider the weakest notion of enumeration algorithms: total polynomial time and incremental polynomial time algorithms.

Theorem 3.4.3 *Let B be a finite set of Boolean functions, such that $\mathrm{S}_{12} \subseteq [B]$. Then there is no total polynomial time enumeration algorithm and no incremental polynomial time algorithm for B-formulas, unless $\mathrm{P} = \mathrm{NP}$.*

Proof. We show that the existence of one of these algorithms for B-formulas implies that $\mathsf{SAT}^*(B)$ can be decided in polynomial time. The Theorem then follows from Lemma 3.4.2.

Let φ be a B-formula. First check if the constant assignments are solutions of φ, this can be done in polynomial time due to Theorem 2.4.2. If there is an incremental polynomial time algorithm, then this algorithm can directly be used to decide, in polynomial time, if there is an additional solution to φ.

Now assume that there is a polynomial total time enumeration algorithm for B-formulas, and let i be the number of constant solutions determined above. It is obvious that φ has a non-constant solution if and only if it has at least $i+1$ many solutions. This condition can easily be tested with the polynomial total time enumeration algorithm: the time the algorithm is allowed to spend if the number of solutions is at most i is, by definition, bounded by a polynomial in φ (since i can be at most 2). Therefore, we can simply start the algorithm, and wait if it finishes in this time. If it does, then its output is the full list of solutions for φ, and we obviously can decide if there is a non-constant solution present in this list. If it does not finish in this time, then there are more then i solutions, and thus there is a non-constant one. □

3.5 Conclusion

If P = NP, then for each set B of Boolean functions, an algorithm of each efficiency type we defined exists (this follows from Theorem 3.3.1). It is easy to see that an algorithm which satisfies the polynomial delay condition also is a polynomial total time algorithm. Since in our definition, a lexicographical enumeration algorithm also has polynomial delay, the following theorem gives a full classification. Figure 3.1 gives a graphical overview of the result.

Theorem 3.5.1 *Assume that* P $\neq$ NP, *and let B be a finite set of Boolean functions. Then the following holds:*

- *If $B \subseteq$ M or $B \subseteq$ L, then there is an efficient lexicographic enumeration algorithm, and an incremental polynomial time algorithm for B-formulas.*
- *Otherwise, if $B \subseteq$ D or $B \subseteq S_0^2$, then there is a polynomial-delay algorithm, and an incremental polynomial time algorithm for B-formulas. There is no efficient lexicographic enumeration algorithm for B-formulas.*
- *Otherwise, there is no total polynomial time enumeration and no incremental polynomial time algorithm for B-formulas.*

We have completely answered the question in which cases there is, for a given set B, an enumeration algorithm of each of the types considered in [JPY88]. In addition to the cases where the satisfiability problem for B-formulas is NP-complete, and therefore efficient enumeration algorithms obviously cannot be hoped for unless P = NP, we also showed that in the cases where tractability of the satisfiability problem follows from a simple "trick," like the knowledge that the all-1-assignment is a solution to the formulas, efficient enumeration algorithms do not exist. An interesting special case here is the case of self-dual formulas. The satisfiability problem again is easy, simply because any such formula is always satisfiable. But the property of self-duality does not only give one solution, it guarantees that half of the possible assignments are solutions. Therefore it is not surprising that these solutions also can be enumerated in an efficient way. Following the discussion at the end of Chapter 2, if B_1 is a subset of D and B_2 is a subset of R_1, then it is easier to compute a *single* solution of a B_2-formula than to compute one of a B_1-formula, but to determine *all* solutions of a B_1-formula is easier that to determine all solutions of a B_2-formula. However, since the property of self-duality does not say anything about the set of solutions where a given variable is set to 0, this does not help us to construct a lexicographical order enumeration algorithm.

A natural generalization of this problem is to consider Boolean circuits instead of formulas. However, it is easy to see that our proofs also give a classification of this problem for the circuit representation of formulas, and it is in fact the same classification. The reason for this is that the only property of formulas which we use is that we can test, in polynomial time, if a given assignment satisfies the formula. This can also be done in polynomial time for the circuit case. The hardness results obviously carry over to the circuit case, since every formula can be transformed in an equivalent circuit in polynomial time. Therefore, the circuit and the formula case give exactly the same classification for the enumeration algorithm types considered here.

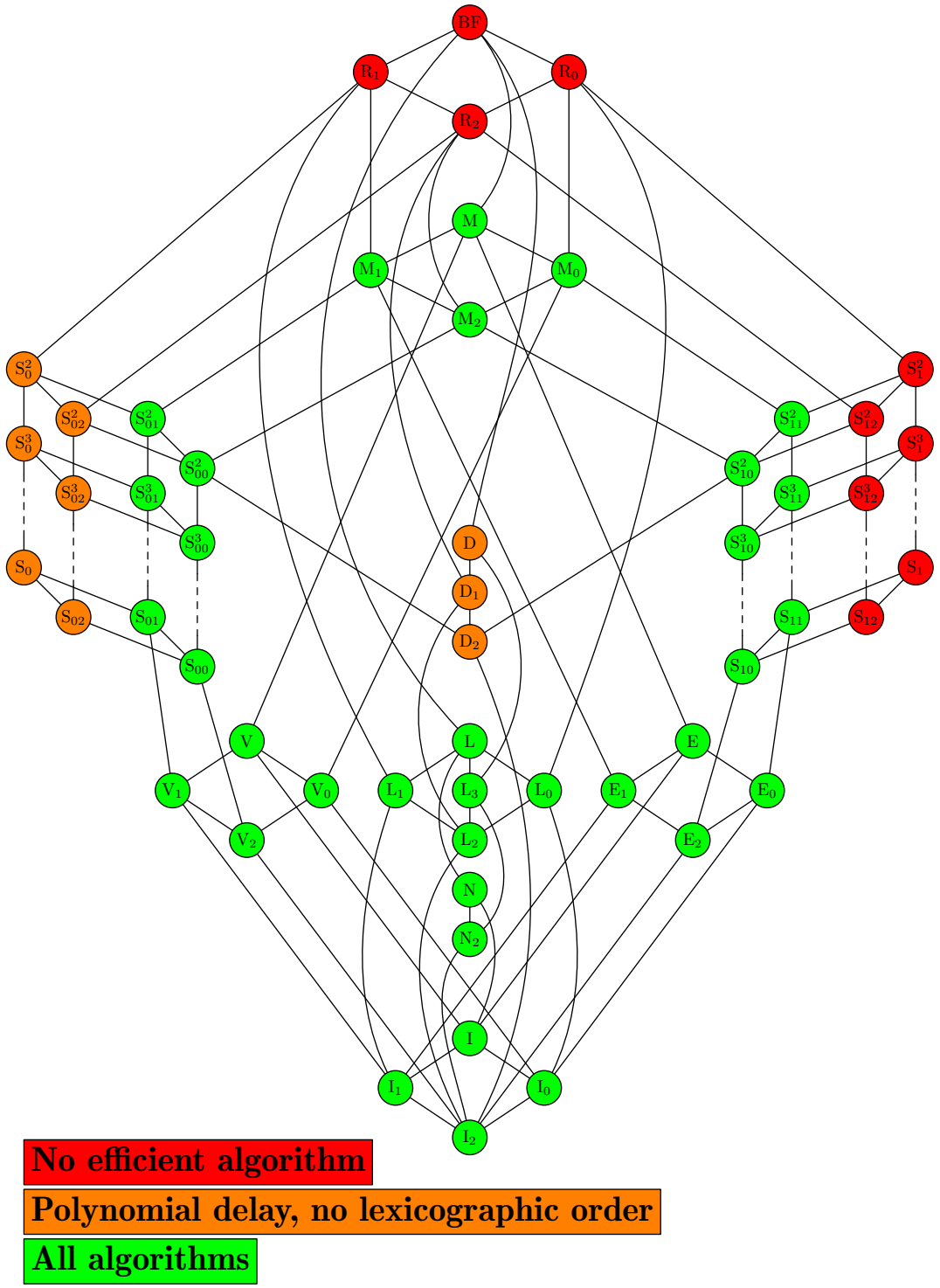

Figure 3.1: The complexity of the enumeration problem

It is of interest that similarly to the results from Chapter 2, we again could show that the existence of efficient algorithms for B-formulas depends only on the clone generated by B. But again, there is no "uniform" proof showing that if $[B_1] = [B_2]$, then B_1 and B_2 either both give an algorithm of one of the considered types, or both do not. In the remainder of this thesis, we will consider formulas in the constraint context, where "base-independence" results like this can usually be achieved easily by means of the Galois connection exhibited in Theorem 1.5.2. However, as we will see in the next chapter, these results cannot always be obtained for free, and there are cases in which we cannot achieve them at all.

Chapter 4

Constraint Satisfaction Problems in Polynomial Time

4.1 Introduction

Up to now, we only studied problems in the B-formula setting, where the formulas are defined as arbitrary nestings, but using only operators from a restricted set B. We will now consider formulas in the constraint setting, as defined in Section 1.5. The main difference to the B-formulas is that the main feature of nesting is absent from constraint formulas. Instead, constraint formulas can be seen as describing a set of local conditions, or constraints, on a set of variables. This also is the reason why a "formula value problem" for constraints is very easy to solve: Let φ be a conjunction of clauses $C_1 \wedge \cdots \wedge C_n$, where the C_i are defined over a finite constraint language Γ. If we consider the formula value problem, then each of these clauses only contains constants. The formula φ is true if and only if each clause C_i is true. Hence, for an unsatisfiability test, a logarithmic time Turing machine only needs to guess the position of the false clause, and then read constant-many bits. With a suitable encoding, when we allow one character for each of the possible combinations of relation symbols and following constants, this means that the "unsatisfiability problem" is $\leq_{\text{proj}}^{\text{dlt}}$-equivalent to the language L of words over the alphabet $\{0,1\}$ containing all words which contain at least one zero. Due to Proposition 2.3.2, this problem is in the class $\Sigma_1^{\mathcal{R}}$, and hence the formula value problem for constraint formulas can be solved in $\Pi_1^{\mathcal{R}}$. As long as the constraint language Γ contains at least one non-full relation, the problem is also complete for this class. Otherwise, the "formula value problem for Γ" is trivial, since all Γ-formulas with constants are true (recall that by definition, we do not allow our constraint languages to contain the empty relation). We therefore conclude that this problem is not very interesting to look at. This observation gives an interesting insight into the differences between B-formulas and constraint formulas: While the nesting of operators, which is a key feature of Boolean formulas, is not necessary to give a complex satisfiability problem (the problem $\mathsf{CSP}(\Gamma_{\mathsf{3SAT}})$ is NP-complete), it is required to give a complex formula evaluation problem.

In the constraint context, the satisfiability problem has achieved much attention. Due to Theorem 1.3.4, we know that 3SAT is NP-complete, and since this problem can be seen as a constraint satisfaction problem, we know that in general, CSP is NP-complete.

However, similarly to the B-formula case, there are cases of the constraint satisfaction problem which can be solved in polynomial time. As 3SAT, it is obvious that an analogously defined 2SAT also can be seen as a constraint satisfaction problem, and this is known to be solvable in polynomial time (in fact, nondeterministic logarithmic space). Also, the well-known Horn satisfiability problem can be defined in this context. Hence, the problem for which constraint language Γ there exist efficient algorithms is of interest. As stated in Theorem 1.5.4, Thomas Schaefer identified all tractable cases of the Boolean constraint satisfaction problem, and showed that the others are NP-complete. His result has been generalized to non-Boolean domains: Andrei Bulatov showed in [Bul06] that this dichotomy also holds for the case where the domain is of cardinality 3. Much work has been done for arbitrary domains, see for example the already mentioned [JCG97], and results by Victor Dalmau and Andrei Krokhin about polymorphisms leading to tractability in [Dal05] and [DK06]. A full classification of the problem for any domain of cardinality bigger than three is still open, but many results were achieved for identifying both tractable and intractable cases.

We now consider the tractable cases over the Boolean domain more closely. As seen in Chapter 2, there are very different complexities inside the class P for problems related to Boolean formulas. Therefore, we now analyze the exact complexity of the polynomial time cases in Schaefer's Theorem 1.5.4 more closely. The motivation for this study is driven by two reasons: first, for practical applications, it is interesting to know if there exist efficient parallel algorithms for the polynomial cases, or if they can be solved with low space bounds, e.g., in logarithmic space. Second, the most interesting feature of Schaefer's result from a theoretical point of view is that it displays a dichotomy, avoiding the infinitely many degrees of complexity which exist between P and NP if these classes differ. Kenneth Regan and Heribert Vollmer showed that there also exist infinitely many degrees of complexity inside the class P, even between the classes AC^0 and NC^1 ([RV97]). Therefore, the question if a dichotomy-like result also holds for the polynomial time cases is of interest. We show that this is indeed the case: under the strict notion of $\leq_m^{\mathrm{AC}^0}$-reducibility, the Boolean constraint satisfaction problems can be shown to just give 6 different complexity cases, namely the complexity classes AC^0, LOGSPACE, $\oplus$LOGSPACE, NLOGSPACE, P, and NP.

In our analysis, we show that the usual tool in the study of the complexity of constraint satisfaction problems, the Galois connection presented in Theorems 1.5.2 and 1.5.3 is only of limited help here: it fails to produce a reduction which closes all complexity classes arising in this classification. Hence, we need to go beyond the algebraic structure of constraint satisfaction problems provided by the lattices of clones and co-clones to achieve a full complexity classification of the problem.

4.2 Preliminaries and Algebraic Tools

The usual reduction given by the Galois connection in Theorem 1.5.3 is stated as a polynomial time reduction. A close inspection of the actual transformation and an application of Omer Reingold's result that search in undirected graphs can be performed in logarithmic space (Theorem 1.3.5) shows that this reduction actually can be computed in

LOGSPACE. However, this is not enough for our purposes, since we will see that there are constraint satisfaction problems in classes as low as AC^0, and this class is not closed under logspace reductions. Hence, in order to study the complexity of the constraint satisfaction problems inside P, we consider $\leq_m^{\mathrm{AC}^0}$-reductions, as introduced in Chapter 1. Unfortunately, we will soon see that a further refinement of the aforementioned reduction given by the Galois connection to AC^0 is not possible. Before we demonstrate this negative result, we prove the logspace reduction and show in which cases it can be used to give an AC^0-reduction.

Lemma 4.2.1 *Let Γ_1 and Γ_2 be constraint languages over a finite domain such that $\mathrm{Pol}(\Gamma_2) \subseteq \mathrm{Pol}(\Gamma_1)$, and let Γ_1 be finite. Then $\mathsf{CSP}(\Gamma_1) \leq_m^{\mathrm{AC}^0} \mathsf{CSP}(\Gamma_2 \cup \{=\})$.*

Proof. Let φ be a Γ_1-formula. We replace each constraint application $R(x_1, \ldots, x_n)$ occurring in φ with a conjunction of clauses as follows: Since φ is a Γ_1-formula, R is a relation from Γ_1. Since $\mathrm{Pol}(\Gamma_2) \subseteq \mathrm{Pol}(\Gamma_1)$, we know that due to Theorem 1.5.2, the clause $R(x_1, \ldots, x_n)$ is equivalent to $\exists y_1, \ldots, y_t S_1(z_1^1, \ldots, z_{i_1}^1) \wedge \cdots \wedge S_k(z_1^k, \ldots, z_{i_k}^k)$ for some relations $S_1, \ldots, S_k \in \Gamma_2 \cup \{=\}$ and variables $z_j^i \in \{y_1, \ldots, y_t, x_1, \ldots, x_n\}$. Hence the reduction can be achieved by replacing the clause $R(x_1, \ldots, x_n)$ with its equivalent conjunction, where the existential quantifiers are left out, and using new y-variables for each clause to be transformed. Since Γ_1 is finite, the replacement rules can be hard-coded into the AC^0-circuit computing the reduction. It is obvious that the resulting formula is satisfiable if and only if the original formula is, and that the reduction can be performed in AC^0. □

The lemma shows that if we have the equality relation present in our constraint languages, then the usual algebraic techniques are sufficient to classify the constraint satisfaction problem up to $\leq_m^{\mathrm{AC}^0}$-reductions. If we want to consider constraint languages which do not contain the equality relation, we need to manually remove equality clauses. The next lemma shows that we can do this, but not without cost: the transformation needs the power of logspace machines.

Lemma 4.2.2 *Let Γ be a constraint language. Then $\mathsf{CSP}(\Gamma \cup \{=\}) \leq_m^{\log} \mathsf{CSP}(\Gamma)$.*

Proof. Let φ be a $\Gamma \cup \{=\}$-formula. Consider the variables appearing in φ as a graph, where two vertices x and y are connected with an edge if and only if there is a clause $x = y$ in the formula φ. For each of the connected components of this graph, introduce a single new variable, and replace all of the variables in the component with this new one. Remove all the equality clauses.

It is obvious that the formula constructed in this way is satisfiable if and only if the original formula is satisfiable. The transformation can be computed in LOGSPACE, because in LOGSPACE, it can be tested if two vertices in an undirected graph are connected due to Theorem 1.3.5. □

The proof of Lemma 4.2.2 uses the search problem in undirected graphs, which is complete for LOGSPACE due to Theorem 1.3.5. It therefore gives good evidence that the reduction needs the entire computational power of LOGSPACE, and cannot be refined to a $\leq_m^{\mathrm{AC}^0}$-reduction. We will see in the following that this is indeed the case.

But first we state a more positive result, which directly follows from the above and the observation that an AC^0-reduction can be computed by a LOGSPACE-machine. The following corollary is a refinement of Theorem 1.5.3:

Corollary 4.2.3 *Let* Γ_1 *and* Γ_2 *be constraint languages over a finite domain such that* $\mathrm{Pol}(\Gamma_2) \subseteq \mathrm{Pol}(\Gamma_1)$. *Then* $\mathsf{CSP}(\Gamma_1) \leq_m^{\log} \mathsf{CSP}(\Gamma_2)$.

Hence, as long as we are interested in a complexity only up to logspace reductions, the Galois connection suffices. However, if we look closer than this, we see that inside LOGSPACE, the Galois connection has its limitations: we now exhibit two constraint languages which have the same set of polymorphisms, and hence generate the same co-clone, but give provably different complexities.

Lemma 4.2.4 *Let* Γ_1 *be the Boolean constraint language defined by* $\Gamma_1 =_{def} \{x, \overline{x}\}$, *and let* $\Gamma_2 =_{def} \Gamma_1 \cup \{=\}$. *Then the following holds:*

1. $\mathrm{Pol}(\Gamma_1) = \mathrm{Pol}(\Gamma_2)$,
2. $\mathsf{CSP}(\Gamma_1) \in \mathrm{AC}^0$,
3. $\mathsf{CSP}(\Gamma_2)$ *is hard for* LOGSPACE *under* $\leq_m^{\mathrm{AC}^0}$*-reductions.*

Proof.

1. $\mathrm{Pol}(\Gamma_2) \subseteq \mathrm{Pol}(\Gamma_1)$ holds since $\Gamma_1 \subseteq \Gamma_2$. Since any function is a polymorphism of the equality relation, the other direction holds as well.
2. Let φ be a Γ_1-formula. Then φ is a conjunction of literals. Hence, φ is unsatisfiable if and only if there is some variable which appears both as a positive and as a negative literal. Thus, unsatisfiability can easily be verified by an AC^0-circuit. Since AC^0 is closed under complementation, the result follows.
3. We show that the undirected graph accessibility problem can be reduced to the complement of $\mathsf{CSP}(\Gamma_2)$. Since LOGSPACE is closed under complementation, the claim then follows from Theorem 1.3.5. Let G be a graph, and let s, t be vertices in G. We construct a formula φ, where there is a variable for every vertex in the graph, and for each edge (x, y) in G, we introduce a clause $x = y$. Additionally, we introduce the clauses s and $\overline{t}$. This transformation can obviously be computed by an AC^0-reduction.

   If there is a path in G from s to t, then obviously, the formula φ is not satisfiable: since an equality clause is introduced for every edge, the variables must take the same value in a satisfying assignment for φ. But φ also demands that s must be true, and t must be false. This is clearly a contradiction. For the other direction, suppose that there is no path from s to t in G. Since G is an undirected graph, this means that s and t are not connected in G. Hence, they are not connected with equality clauses in the formula φ either. Therefore, we can simply set s and everything connected to s to true, and every other variable to false, satisfying the formula.

□

The main message of the lemma is that when considering $\leq_m^{\mathrm{AC}^0}$-reductions, we need to look at a finer classification than the one given by the co-clone structure. In the constraint languages exhibited in Lemma 4.2.4, the presence of the equality relation was the only difference between the two languages. It is easy to see that for the problem to get LOGSPACE-hard, it suffices to have a relation which is "nearly the equality relation," or "implements" it. For example, we could do the same construction as in the proof for the lemma if we had the relation $\oplus$, by simulating a clause $x = y$ with $(x \oplus z) \wedge (z \oplus y)$. We capture this generalization with the following definition:

Definition Let Γ be a Boolean constraint language, and let R be an m-ary relation. We say that Γ *can express* R, if there is a Γ-formula φ with variables $x_1, \ldots, x_m, t_1, \ldots, t_n$, such $\exists t_1, \ldots, t_n \varphi(x_1, \ldots, x_m, \ldots, t_n)$ is equivalent to $R(x_1, \ldots, x_m)$. We say that φ *expresses* R.

If R is the equality relation, then we just say "Γ expresses equality." It is obvious that if some constraint language can express equality, then adding the equality relation itself to the language does not make a significant difference for the complexity of its constraint satisfaction problem:

Proposition 4.2.5 *Let Γ be a constraint language which can express equality. Then* $\mathsf{CSP}(\Gamma \cup \{=\}) \leq_m^{\mathrm{AC}^0} \mathsf{CSP}(\Gamma)$.

Proof. The reduction is achieved by simply replacing any occurring equality clauses with the Γ-formula that expresses equality. □

The proposition implies that if we only consider constraint languages which can express equality, then the Galois connection suffices to completely classify the complexity of the constraint satisfaction problem. The following is an easy corollary from the previous results:

Corollary 4.2.6 *Let Γ_1 and Γ_2 be Boolean constraint languages such that* $\mathrm{Pol}(\Gamma_1) = \mathrm{Pol}(\Gamma_2)$, *and both Γ_1 and Γ_2 can express equality. Then* $\mathsf{CSP}(\Gamma_1) \equiv_m^{\mathrm{AC}^0} \mathsf{CSP}(\Gamma_2)$.

Proof. We show that $\mathsf{CSP}(\Gamma_1) \leq_m^{\mathrm{AC}^0} \mathsf{CSP}(\Gamma_2)$. The other direction follows due to symmetry. Due to Lemma 4.2.1, we know that $\mathsf{CSP}(\Gamma_1) \leq_m^{\mathrm{AC}^0} \mathsf{CSP}(\Gamma_2 \cup \{=\})$, and from Proposition 4.2.5, it follows that $\mathsf{CSP}(\Gamma_2 \cup \{=\}) \leq_m^{\mathrm{AC}^0} \mathsf{CSP}(\Gamma_2)$. Due to the transitivity of the $\leq_m^{\mathrm{AC}^0}$-reduction, the result follows. □

The question how to deal with languages which cannot express equality remains open for now. However, we will show that for the cases which actually arise in the Boolean constraint satisfaction problem, we can answer this question on a case-by-case basis.

As mentioned before, and encountered in Chapter 2, the symmetry in Post's lattice often gives a symmetry in the involved complexity classes as well. In the constraint context, this result can be stated as follows:

Definition Let R be a Boolean relation. We define

$$\overline{R} =_{\text{def}} \{(\overline{\alpha_1}, \ldots, \overline{\alpha_n}) \mid (\alpha_1, \ldots, \alpha_n) \in R\}.$$

For a constraint language Γ, we define $\overline{\Gamma} =_{\text{def}} \left\{\overline{R} \mid R \in \Gamma\right\}$.

It is obvious that the relations R and $\overline{R}$ are isomorphic. Hence, the following proposition is natural:

Proposition 4.2.7 *Let Γ be a finite constraint language. Then $\mathsf{CSP}(\Gamma) \equiv_m^{\mathrm{AC}^0} \mathsf{CSP}\left(\overline{\Gamma}\right)$.*

Proof. The reduction is achieved by simply replacing each occurrence of a relation R with the relation $\overline{R}$. From a satisfying assignment for one formula we can obtain a satisfying assignment for the other formula by negating every variable assignment. The reduction in the converse direction follows due to symmetry. □

There is a natural connection between the complement of a relation and the polymorphisms of the original relation:

Lemma 4.2.8 *Let R be a Boolean relation. Then $\mathrm{Pol}(R) = \mathrm{dual}\left(\mathrm{Pol}\left(\overline{R}\right)\right)$.*

Proof. Due to symmetry (obviously, for a set B of Boolean functions, it holds that $\mathrm{dual}(\mathrm{dual}(B)) = B$, and $\overline{\overline{R}} = R$), it suffices to show one inclusion. Therefore, let $f\colon \{0,1\}^m \to \{0,1\}$ be an m-ary polymorphism of R. We need to show that $\mathrm{dual}(f)$ is a polymorphism of $\overline{R}$. Hence, let $(\alpha_1^1, \ldots, \alpha_n^1), \ldots, (\alpha_1^m, \ldots, \alpha_n^m)$ be tuples from $\overline{R}$. By definition, it follows that $(\overline{\alpha_1^1}, \ldots, \overline{\alpha_n^1}), \ldots, (\overline{\alpha_1^m}, \ldots, \overline{\alpha_n^m})$ are tuples from R. Since f is a polymorphism of R, it follows that

$$(f(\overline{\alpha_1^1}, \ldots, \overline{\alpha_1^m}), \ldots, f(\overline{\alpha_n^1}, \ldots, \overline{\alpha_n^m}))$$

is a tuple from R. Thus, the coordinate-wise negation of this tuple is an element of $\overline{R}$. Due to the definition of the dual(.)-operator, this is the same element as $\mathrm{dual}(f)$ applied to the coordinates of the original tuples. Hence, $\mathrm{dual}(f)$ is a polymorphism of $\overline{R}$, as claimed. □

The obvious corollary of Corollary 4.2.3, Proposition 4.2.7 and Lemma 4.2.8 is the following:

Corollary 4.2.9 *Let Γ_1 and Γ_2 be Boolean constraint languages such that $\mathrm{Pol}(\Gamma_1) = \mathrm{dual}(\mathrm{Pol}(\Gamma_2))$. Then $\mathsf{CSP}(\Gamma_1) \equiv_m^{\log} \mathsf{CSP}(\Gamma_2)$.*

4.3 Algorithms

We will now give upper complexity bounds for the Boolean constraint satisfaction problems. It is obvious that the problem is in NP for any finite constraint language Γ. The following results can be found in e.g., [Sch78] and [BCRV04].

Proposition 4.3.1 *Let* Γ *be a Boolean constraint language. Then the following holds:*

1. *If* $\mathrm{Pol}(\Gamma) \notin \{\mathrm{N}_2, \mathrm{I}_2\}$, *then* $\mathsf{CSP}(\Gamma) \in \mathrm{P}$,
2. *if* $\mathrm{L}_2 \subseteq \mathrm{Pol}(\Gamma)$, *then* $\mathsf{CSP}(\Gamma) \in \oplus\mathrm{LOGSPACE}$,
3. *if* $\mathrm{D}_2 \subseteq \mathrm{Pol}(\Gamma)$, *then* $\mathsf{CSP}(\Gamma) \in \mathrm{NLOGSPACE}$,
4. *if* $\mathrm{I}_0 \subseteq \mathrm{Pol}(\Gamma)$ *or* $\mathrm{I}_1 \subseteq \mathrm{Pol}(\Gamma)$, *then every* Γ*-formula is satisfiable and hence* $\mathsf{CSP}(\Gamma)$ *is trivial.*

Note that for the last point, it is important to remember that our constraint languages are not allowed to contain the empty relation: The empty relation is in fact invariant under the 1-ary constant functions (but not under the 0-ary constants). We now give our first upper bound. The proof for the following lemma is based on the proof for Theorem 6.5 in [CKS01].

Lemma 4.3.2 *Let* Γ *be a Boolean constraint language such that* $\mathrm{S}_{02} \subseteq \mathrm{Pol}(\Gamma)$ *or* $\mathrm{S}_{12} \subseteq \mathrm{Pol}(\Gamma)$. *Then* $\mathsf{CSP}(\Gamma) \in \mathrm{LOGSPACE}$.

Proof. We show the claim for the case $\mathrm{S}_{02} \subseteq \mathrm{Pol}(\Gamma)$, the claim for $\mathrm{S}_{12} \subseteq \mathrm{Pol}(\Gamma)$ follows from Corollary 4.2.9. Note that due to Lemma 1.5.6, there is no finite constraint language whose polymorphisms are exactly the clone S_{02}. Since Γ is finite, this implies that $\mathrm{S}_{02}^k \subseteq \mathrm{Pol}(\Gamma)$ for some $k \geq 2$.

Due to Table 1.2, we know that $\mathrm{Pol}\left(\left\{\mathrm{OR}^k, x, \overline{x}\right\}\right) = \mathrm{S}_{02}^k$. Due to Corollary 4.2.3, and since LOGSPACE is closed under $\leq_m^{\log}$-reductions, we can therefore assume that $\Gamma = \left\{\mathrm{OR}^k, x, \overline{x}\right\}$.

Now for this constraint language, a Γ-formula is satisfiable if and only if for every $\mathrm{OR}(x_{i_1}, \ldots, x_{i_k})$-clause, at least one of the variables $x_{i_1}, \ldots, x_{i_k}$ does not appear as a negative literal. Hence, to verify that such a formula is not satisfiable, it suffices to compare one of the OR-clauses and k many literals. This can be performed by an AC^0-circuit. Therefore, the constraint satisfaction problem for Γ can be solved in $\mathrm{AC}^0 \subseteq \mathrm{LOGSPACE}$, putting our original problem inside LOGSPACE as well. □

The proof for Lemma 4.3.2 reveals that this is not the best answer we can give for the constraint languages involved: the algorithm uses a reduction to a constraint satisfaction problem which can be solved in a much lower class than LOGSPACE. However, due to Lemma 4.2.4, we know that if we add the equality relation to the constraint language $\left\{\mathrm{OR}^k, x, \overline{x}\right\}$, we get a problem which is hard for LOGSPACE under AC^0-reductions. Hence, we will need to revisit these constraint languages in order to get a complete classification. But first, we give another upper bound complexity result:

Lemma 4.3.3 *Let* Γ *be a Boolean constraint language such that* $\mathrm{S}_{00} \subseteq \mathrm{Pol}(\Gamma)$ *or* $\mathrm{S}_{10} \subseteq \mathrm{Pol}(\Gamma)$. *Then* $\mathsf{CSP}(\Gamma) \in \mathrm{NL}$.

Proof. Again we show the claim for the case $\mathrm{S}_{00} \subseteq \mathrm{Pol}(\Gamma)$, the dual case follows with Corollary 4.2.9. Due to Lemma 1.5.6, we conclude that $\mathrm{Pol}(\Gamma) \supseteq \mathrm{S}_{00}^k$ holds for some $k \geq 2$. Table 1.2 now shows that $\mathrm{Pol}\left(\left\{\mathrm{OR}^k, x, \overline{x}, \rightarrow\right\}\right) = \mathrm{S}_{00}^k$. Hence, we can assume, due to Corollary 4.2.3, that Γ is equal to this set. Now it is easy to see that a Γ-formula is

unsatisfiable if and only if for some clause $\mathrm{OR}(x_{i_1}, \ldots, x_{i_k})$, for every appearing variable, there is a $\rightarrow$-path in the formula to a variable which appears as a negative literal. Since search in directed graphs can be performed in NL due to Theorem 1.3.5, and NL is closed under complementation due to Theorem 1.3.1, this completes the proof. □

In contrast to the previous result, here we needed the entire power of NL to solve the constraint satisfaction problem. We will later see that this is not a coincidence: for the constraint languages covered by Lemma 4.3.3, there are no easier cases.

4.4 The Equality Relation

As mentioned before, the question whether a constraint language contains or can express the equality relation makes a significant difference complexity-wise. Hence, we now study which constraint languages can express this relation, and which cannot. Fortunately, it turns out that in most cases, our constraint languages can express equality, and hence due to Corollary 4.2.6, we know that in this case, all constraint languages giving the same co-clone give rise to constraint satisfaction problems of equal complexity. As we will see later in Chapter 5, this question is not only of interest when considering low complexity classes, but also if we study problems different from satisfiability, where identification of variables does not leave the properties of the formulas that we are interested in unchanged.

Lemma 4.4.1 *Let* Γ *be a finite Boolean constraint language such that* $\mathrm{Pol}(\Gamma) \subseteq \mathrm{M}$, $\mathrm{Pol}(\Gamma) \subseteq \mathrm{L}$, *or* $\mathrm{Pol}(\Gamma) \subseteq \mathrm{D}$. *Then* Γ *can express equality.*

Proof. For M, observe that due to Table 1.2, the relation $x \rightarrow y$ is invariant under M. Hence, for any constraint language Γ satisfying $\mathrm{Pol}(\Gamma) \subseteq \mathrm{M}$, there exists a $\Gamma \cup \{=\}$-formula in which additional new existentially quantified variables occur, and which is equivalent to $x \rightarrow y$. Equality clauses in this formula involving existentially quantified variables can be removed using variable identification. Equality clauses between x and y cannot appear, since the assignment $I(x) = 0, I(y) = 1$ satisfies the formula, but not the equality constraint $x = y$. Hence, we can construct a Γ-formula which additional existentially quantified variables which is equivalent to $x \rightarrow y$. Now the conjunction $(x \rightarrow y) \wedge (y \rightarrow x)$ expresses equality.

For the other two cases, the argument is very similar. In the case $\mathrm{Pol}(\Gamma) \subseteq \mathrm{L}$, observe that due to Table 1.2, the relation EVEN^4 is invariant under L. Again, in a formula defining $\mathrm{EVEN}^4(x_1, x_2, x_3, x_4)$, no equality clauses between the x_i can appear. Hence, we can build a Γ-formula with existential quantifiers expressing this relation, and finally express $x = y$ as $\exists z \mathrm{EVEN}^4(z, z, x, y)$. For the final case $\mathrm{Pol}(\Gamma) \subseteq \mathrm{D}$, we can express $x \oplus y$ in a similar way, and then express $x = y$ as $\exists z(x \oplus z) \wedge (y \oplus z)$. □

Lemma 4.4.1 and Corollary 4.2.6 imply that for the constraint languages with the corresponding set of polymorphisms, we do not need to "look inside the co-clones." But Lemma 4.2.4 and the discussion after the proof for Lemma 4.3.2 indicate that in the cases $\mathrm{S}_{02} \subseteq \mathrm{Pol}(\Gamma)$ and $\mathrm{S}_{12} \subseteq \mathrm{Pol}(\Gamma)$, there is need for a more detailed analysis. The following Lemma gives the complete picture of the situation here:

Lemma 4.4.2 *Let R be a Boolean relation such that $\mathrm{S}_{02} \subseteq \mathrm{Pol}(R)$. Then either R can be expressed using only literals and OR-clauses, or R can express equality. There is an algorithm deciding which case occurs.*

Proof. Assume that R is an n-ary relation, and $\mathrm{S}_{02} \subseteq \mathrm{Pol}(R)$. Again, due to Lemma 1.5.6, we know that these sets cannot be equal, and hence it holds that $\mathrm{S}_{02}^k \subseteq \mathrm{Pol}(R)$ for some $k \geq 2$. Table 1.2 shows that $\mathrm{Pol}(\{x, \overline{x}, \mathrm{OR}^m\}) = \mathrm{S}_{02}^k$, and hence Theorem 1.5.2 tells us that we can construct a formula φ which is equivalent to $R(x_1, \ldots, x_n)$, and only contains equalities, literals, and m-ary OR-clauses, where some of the variables may be existentially quantified. Without loss of generality, assume that R is not the empty relation, hence the formula φ is satisfiable.

We now give an algorithm which simplifies this formula. We can assume that equality clauses only appear between the variables $x_1, \ldots, x_n$, i.e., the ones which are not existentially quantified. Occurrences of equality between existentially quantified variables can be removed by variable identification. Repeat the following steps as long as changes occur in the formula:

1. For any clause $t_1 = t_2$ where t_1 or t_2 also appear as a literal, remove the equality clause and insert the corresponding literals for t_1 and t_2.
2. For each OR-clause, remove every variable which also appears as a negative literal, because this variable cannot be set to 1.
3. Remove each OR-clause in which a variable appears which also appears as a positive literal, because these clauses are trivially true.
4. In an OR-clause containing variables which are connected with $=$, remove all of them except one.

It is obvious that the transformations above do not change the relation represented by the formula φ. The steps need to be performed until no change happens anymore, since in step 2, or step 3, an OR-clause can be reduced to a literal, and thus literals can be added during the execution of this algorithm.

The algorithm terminates, since in each step, one of the following occurs: an equality clause is removed, an OR-clause is removed, or an OR-clause is reduced in arity. Since clauses are neither introduced nor extended in arity, this can only be done a finite number of times.

Now, if after this transformation, no equality clause remains, then obviously, R can be expressed by a formula containing only the m-ary OR and literals. Otherwise, let $x_{i_1} = x_{i_2}$ be a remaining clause (remember that equality only occurs between the variables $x_1, \ldots, x_n$). Without loss of generality, assume that this clause is $x_1 = x_2$. Now consider the formula $\psi = \exists x_3, \ldots, x_n R(x_1, \ldots, x_n)$. We claim that ψ expresses equality, concluding the proof.

We show that $I(x_1) = \alpha_1, I(x_2) = \alpha_2$ is a satisfying assignment to ψ if and only if $\alpha_1 = \alpha_2$. Since $x_1 = x_2$ is a clause in ψ, it is obvious that if $I \models \psi$, then $\alpha_1 = \alpha_2$. Now let $\alpha_1 = \alpha_2$. We prove that I is a satisfying assignment for ψ. We extend the assignment I to the existentially quantified variables as follows:

- For every variable t which is connected to x_1 and x_2 with equality clauses, define $I(t) =_{\text{def}} I(x_1)$.
- For every variable t which appears as a negative literal, define $I(t) =_{\text{def}} 0$.
- For all other variables t, define $I(t) =_{\text{def}} 1$.

We claim that I satisfies the constraints appearing in ψ. Obviously, all literals are satisfied by the assignment: The variables x_1 and x_2 cannot appear as literals due to step 1 of the simplification procedure, and since R is not empty, there cannot be any contradicting literals. By construction, all of the equality clauses are satisfied: The equality $x_1 = x_2$ is satisfied due to the choice of I, and for every variable connected to these, the equality constraints in which they appear are satisfied due to the construction of I. It remains to show that all of the appearing OR-clauses are satisfied. Hence, let $\mathrm{OR}(t_1, \ldots, t_j)$ be a clause. None of the variables $(t_1, \ldots, t_j)$ appear as negative literals due to step 2 of the simplification procedure. The clause cannot only contain variables connected to x_1 with $=$-clauses, since then it would have been reduced to a literal in step 4 of the procedure. Hence, the clause contains a variable which neither appears as a negative literal, nor is connected to x_1 with equality constraints. By choice of I, all of these variables are set to 1, and hence the OR-clause is satisfied. □

Note that the algorithm to test if a relation can implement equality can be made much simpler than the one explained above: it is sufficient to test for each pair of variables, if the relation, where all other variables are quantified existentially, gives the equality relation. However, the above proof also shows that in the case where this is not the case, we have a representation of the relation as a conjunction of literals, which is required for our complexity results involving these relations.

Lemma 4.4.2 helps us to give a complete characterization of the complexity of constraint languages with this set of polymorphisms:

Corollary 4.4.3 *Let* Γ *be a Boolean constraint language such that* $\mathrm{S}_{02} \subseteq \mathrm{Pol}(\Gamma)$ *or* $\mathrm{S}_{12} \subseteq \mathrm{Pol}(\Gamma)$. *Then* $\mathsf{CSP}(\Gamma)$ *is either solvable in* AC^0, *or complete for* LOGSPACE *under* $\leq_m^{\mathrm{AC}^0}$*-reductions.*

Proof. Due to Proposition 4.2.7 and Lemma 4.2.8, it suffices to show the claim for the case $\mathrm{S}_{02} \subseteq \mathrm{Pol}(\Gamma)$.

If there is a relation in Γ which can express equality, then $\mathsf{CSP}(\Gamma)$ is hard for LOGSPACE due to Lemma 4.2.4 and Corollary 4.2.6. The upper bound follows from Lemma 4.3.2.

Hence, assume that none of the relations in Γ can express equality. Then, Lemma 4.4.2 states that every relation in Γ can be expressed using only OR-clauses of bounded arity, and literals. Hence, each Γ-formula can be transformed by an $\leq_m^{\mathrm{AC}^0}$-reduction, into a formula in which only these constraints appear. Now for these formulas, satisfiability can be tested in AC^0, this is shown in the proof for Lemma 4.3.2. □

4.5 Hardness Results

As mentioned before, for the sets of polymorphisms that we deal with in this section, we know that our constraint languages can express equality. Hence, we can apply Corollary 4.2.6, and know that our complexity results are independent of the actual constraint language that we consider.

Lemma 4.5.1 *Let* Γ *be a Boolean constraint language such that* $\mathrm{Pol}(\Gamma) \in \{\mathrm{D}_1, \mathrm{D}\}$. *Then* $\mathsf{CSP}(\Gamma)$ *is complete for* LOGSPACE *under* $\leq_m^{\mathrm{AC}^0}$*-reductions.*

Proof. Note that due to Table 1.2, it holds that $\mathrm{Pol}(\oplus) = \mathrm{D}$, and $\mathrm{Pol}(x_1 \wedge (x_2 \oplus x_3)) = \mathrm{D}_1$. Hence, by Lemma 4.4.1 and Corollary 4.2.6, we can assume that Γ only contains these relations.

From Theorem 1.3.5, we know that the class known as SL, ("symmetric logspace") and LOGSPACE coincide. Hence, Problem 4.1 in Section 7 of [AG00] shows that the satisfiability problem for formulas in which positive literals and clauses of the form $x \oplus y$ is complete for LOGSPACE, giving the proof for the case $\mathrm{Pol}(\Gamma) = \mathrm{D}_1$.

Due to Corollary 4.2.3, and since $\mathrm{D}_1 \subseteq \mathrm{D}$, this also gives the upper bound for the case $\mathrm{Pol}(\Gamma) = \mathrm{D}$. It remains to show hardness for this case. In order to do this, we reduce $\mathsf{CSP}(\{x_1 \wedge (x_2 \oplus x_3)\})$ to $\mathsf{CSP}(\{\oplus\})$.

The reduction works as follows: for each clause x, introduce a clause $x \oplus f$ for a single new variable f. In this way, we generate a formula in which only $\oplus$-clauses appear. Now if the original formula is satisfiable, then extending a satisfying assignment for it with the additional assignment $f = 0$ gives a solution for the new formula. On the other hand, if I satisfies the new formula, then, since $\oplus$ is closed under negation, we know that the assignment $\overline{I}$, which is obtained by $\overline{I}(x) =_{\mathrm{def}} \overline{I(x)}$ for every variable x, is a satisfying assignment for the formula as well. Hence, if the formula is satisfiable, then there is a satisfying assignment I such that $I(f) = 0$. Obviously, this assignment restricted to the variables appearing in the original formula satisfies it. This completes the reduction. □

The next case that we will consider deals with Horn and anti-Horn formulas. The complexity result that we give was known to hold for logspace reductions (see for example [Pla84] for Horn formulas). We now prove that this is also true for the stricter $\leq_m^{\mathrm{AC}^0}$-reduction.

Lemma 4.5.2 *Let* Γ *be a Boolean constraint language such that* $\mathrm{Pol}(\Gamma) \in \{\mathrm{E}_2, \mathrm{V}_2\}$. *Then* $\mathsf{CSP}(\Gamma)$ *is complete for* P *under* $\leq_m^{\mathrm{AC}^0}$*-reductions.*

Proof. The upper bound follows from Proposition 4.3.1. We show the lower bound for the case $\mathrm{Pol}(\Gamma) = \mathrm{V}_2$, the proof for the dual case $\mathrm{Pol}(\Gamma) = \mathrm{E}_2$ then follows from Lemma 4.2.8, Proposition 4.2.7, Lemma 4.4.1, and Corollary 4.2.6. For the lower bound, we reduce from the problem $\mathsf{SAT}^C(\{(x \wedge (y \vee z)), c_0\})$. This is the problem to determine if a given Boolean circuit over the base containing the function $(x \wedge (y \vee z))$ and the constant zero function is satisfiable. This problem is hard for P due to Theorem 16 in [RW00].

The idea of the proof is the following: to simulate a single gate in the original circuit, we use a single constraint. It turns out that this is possible using only relations which

are invariant under the Boolean OR-operator. Let C be a $\{(x \wedge (y \vee z)), c_0\}$-circuit. For each gate g in C, introduce a variable x_g. The variable x_g in the constraint formula will be used to represent the value computed by the gate g in the circuit C. For the gate g, additionally introduce a constraint as follows:

- If g is a c_0-gate, then add a constraint $\overline{x_g}$.
- If g is a gate computing the function $x \vee (y \wedge z)$, then let g_x, g_y, g_z be the predecessor gates of g in C. Now introduce a constraint $x_g \rightarrow (x_{g_x} \wedge (x_{g_y} \vee x_{g_z}))$.
- If g is the output-gate, then add a clause x_g.

All the relations appearing in the construction are invariant under the Boolean OR, and hence for the constraint language Γ' containing these, it holds that $\mathrm{Pol}(\Gamma) \supseteq \mathrm{V}_2$. Hence, we can use the given contraint language Γ to express these relations. We now claim that the original circuit C is satiable if and only if the conjunction of the constraints mentioned above, which we will call φ, is.

First, assume that C is satisfiable. Let I be a satisfying assignment for C, and now assign to every variable x_g in φ the value that the gate g computes when I is given as input to the circuit. By construction, it is obvious that all constraints in φ are satisfied (since I is a satisfying assignment, it holds that x_g is true for g denoting the output gate of the circuit).

For the other direction, assume that φ is satisfiable, and let I be a satisfying assignment. Assign to all input gates from C the values that I assigns to the corresponding variables in φ. By an easy induction, it can be shown that for all gates g in C, it holds that the value computed by the gate g on this input is 1 whenever the variable x_g is 1. Hence, this holds for the output gate g in particular, and since x_g is a clause in φ, this implies that the output gate computes the value 1, i.e., we have constructed a satisfying assignment for the circuit, as required. □

The next case can again be covered by a reduction from a restricted satisfiability problem for Boolean circuits:

Lemma 4.5.3 *Let Γ be a Boolean constraint language such that $\mathrm{Pol}(\Gamma) \in \{\mathrm{L}_2, \mathrm{L}_3\}$. Then $\mathsf{CSP}(\Gamma)$ is $\leq_m^{\mathrm{AC}^0}$-complete for $\oplus$LOGSPACE.*

Proof. The upper bound follows from Proposition 4.3.1. For the lower bound, due to Lemma 4.4.1 and Corollary 4.2.6, it suffices to exhibit one constraint language Γ with the required polymorphism that gives rise to a problem which is hard for $\oplus$LOGSPACE. This means that we can, without loss of generality, assume that Γ consists of the relations represented by the formulas x and $x = y \oplus z$, since these relations are invariant under L_2, and hence the original constraint language can express these relations.

We now show that $\mathsf{SAT}^C(\{\oplus\})$, which is the satisfiability problem for circuit containing only $\oplus$-gates, reduces to $\mathsf{CSP}(\Gamma)$. Since Theorem 12 of [RW00] shows that this problem is complete for $\oplus$LOGSPACE, the claim then follows. Let C be a circuit in which only $\oplus$-gates occur. The reduction is straightforward: as in the proof for Lemma 4.5.2, we introduce a variable x_g for each gate g in the circuit. Now, to simulate a gate g computing the function $y \oplus z$, where the predecessor gates are g_y and g_z, simply introduce a

clause $x_g = (x_{g_y} \oplus x_{g_z})$. For the output gate g, insert a clause x_g. It is obvious that this formula simulates the circuit, and is satisfiable if and only if the circuit is.

Now assume that $\mathrm{Pol}(\Gamma) = \mathrm{L}_3$. In this case, we cannot express a term like x_g to force the variable corresponding to the output-gate to 1, since L_3 contains negation, and therefore for every satisfying assignment I to a Γ-formula, the negated assignment $\overline{I}$ satisfies it as well. Therefore, we cannot simply simulate a circuit with this constraint language. Instead, we give a reduction from the case just proven before: we show that for every constraint language Γ satisfying $\mathrm{Pol}(\Gamma) = \mathrm{L}_2$, there exists a constraint language Γ', such that $\mathrm{Pol}(\Gamma') \supseteq \mathrm{L}_3$, and $\mathsf{CSP}(\Gamma) \leq_m^{\mathrm{AC}^0} \mathsf{CSP}(\Gamma')$. Since due to the comments at the beginning of the proof it is sufficient to show the hardness result for a single constraint language with this set of polymorphisms, the general claim follows.

For an n-ary relation $R \in \Gamma$, we define the relation R' as the $(n+1)$-ary relation

$$R' =_{\mathrm{def}} (\{0\} \times R) \cup \left(\{1\} \times \overline{R}\right).$$

By construction, this relation is invariant under negation, but it is obvious that it contains all the information that R contains. It is straightforward to verify that R' is still closed under L_2, since R is. Since $\mathrm{L}_3 = [\mathrm{N}_2 \cup \mathrm{L}_2]$, and the set of polymorphisms always is a clone, this implies that R' is closed under L_3. We now define Γ' as the constraint language containing the relation R' for each relation R in Γ. We show that $\mathsf{CSP}(\Gamma)$ can be reduced to $\mathsf{CSP}(\Gamma')$ as folllows:

Let φ be a Γ-formula. Introduce a new variable t. For each clause $R(x_1, \ldots, x_n)$ in φ, introduce the clause $R'(t, x_1, \ldots, x_n)$. Let φ' denote the conjunction of these terms. By construction, φ' is a Γ'-formula. The reduction obviously can be performed in AC^0. It remains to prove that φ is satisfiable if and only if φ' is.

First, let I be a satisfying assignment for φ. It is obvious that by extending this assignment with $I(t) = 0$, we obtain a solution for the formula φ'. On the other hand, if I satisfies φ', then we can assume, without loss of generality, that $I(t) = 0$ (otherwise, consider the assignment $\overline{I}$, which satisfies the formula as well, since every relation in Γ' is closed under negation). Trivially, by restricting this assignment to the variables appearing in the original formula φ, we get a solution for φ. This concludes the proof. □

The final result is the matching lower bound for Lemma 4.3.3:

Lemma 4.5.4 *Let Γ be a Boolean constraint language such that $\mathrm{Pol}(\Gamma) \subseteq \mathrm{M}_2$. Then $\mathsf{CSP}(\Gamma)$ is hard for NL under $\leq_m^{\mathrm{AC}^0}$-reductions.*

Proof. Again, since Γ can express equality and the results from Section 4.4, it suffices to present a single constraint language with the required properties. Let $\Gamma = \{x, \overline{x}, x \to y\}$. We show that the search problem in directed graphs can be reduced to $\mathsf{CSP}(\Gamma)$, the result then follows from Theorem 1.3.5.

Let G be a directed graph, and let s, t be vertices in G. Introduce a clause s and a clause $\overline{t}$, and for each edge (x, y) in the graph, a constraint $x \to y$. As in the proof for the LOGSPACE-hardness part of Lemma 4.2.4, it is obvious that there is a path from s to t in G if and only if φ is not satisfiable: if there is a path from s to t, then the formula demands that s must be set to 1, and t to 0. But at the same time, the formula forces s

to imply t, which is a contradiction. If there is no from s to t, then an assignment can simply set s and everything implied by s to 1, and every other variable to 0, satisfying the formula. Since NL is closed under complementation due to Theorem 1.3.1, this concludes the proof. □

4.6 Conclusion

The results from this chapter give a complete complexity classification of the Boolean constraint satisfaction problem. In Figure 4.1, the classes and complexities are shown in a graphical representation. The classes where the complexity reads "LOGSPACE-complete / AC^0" are the cases where the complexity does not only depend on the polymorphisms. In this case, the algorithm from Lemma 4.4.2 gives the precise answer. Note that for those cases where the constraint satisfaction problem is NP-complete, we know that the polymorphisms of the constraint language are a subset of N_2. In this case, Lemma 4.4.1 shows that the constraint language can express equality, and hence these problems are also equivalent under $\leq_m^{\mathrm{AC}^0}$-reductions. The complete classification of the complexity of the Boolean constraint satisfaction problem is summarized by the following theorem, which is a refinement of Theorem 5.1 from [Sch78] and Theorem 6.5 from [CKS01]:

Theorem 4.6.1 *Let Γ be a Boolean constraint language.*

- *If $\mathrm{I}_0 \subseteq \mathrm{Pol}(\Gamma)$ or $\mathrm{I}_1 \subseteq \mathrm{Pol}(\Gamma)$, then every constraint formula over Γ is satisfiable, and therefore $\mathsf{CSP}(\Gamma)$ is trivial.*
- *If $\mathrm{Pol}(\Gamma) \in \{\mathrm{I}_2, \mathrm{N}_2\}$, then $\mathsf{CSP}(\Gamma)$ is complete for* NP *under $\leq_m^{\mathrm{AC}^0}$-reductions.*
- *If $\mathrm{Pol}(\Gamma) \in \{\mathrm{V}_2, \mathrm{E}_2\}$, then $\mathsf{CSP}(\Gamma)$ is complete for* P *under $\leq_m^{\mathrm{AC}^0}$-reductions.*
- *If $\mathrm{Pol}(\Gamma) \in \{\mathrm{L}_2, \mathrm{L}_3\}$, then $\mathsf{CSP}(\Gamma)$ is complete for* $\oplus$LOGSPACE *under $\leq_m^{\mathrm{AC}^0}$-reductions.*
- *If $\mathrm{S}_{00} \subseteq \mathrm{Pol}(\Gamma) \subseteq \mathrm{S}_{00}^2$ or $\mathrm{S}_{10} \subseteq \mathrm{Pol}(\Gamma) \subseteq \mathrm{S}_{10}^2$ or $\mathrm{Pol}(\Gamma) \in \{\mathrm{D}_2, \mathrm{M}_2\}$, then $\mathsf{CSP}(\Gamma)$ is complete for* NLOGSPACE *under $\leq_m^{\mathrm{AC}^0}$-reductions..*
- *If $\mathrm{Pol}(\Gamma) \in \{\mathrm{D}_1, \mathrm{D}\}$, then $\mathsf{CSP}(\Gamma)$ is complete for* LOGSPACE *under $\leq_m^{\mathrm{AC}^0}$-reductions.*
- *Otherwise, $\mathrm{S}_{02} \subseteq \mathrm{Pol}(\Gamma) \subseteq \mathrm{R}_2$ or $\mathrm{S}_{12} \subseteq \mathrm{Pol}(\Gamma) \subseteq \mathrm{R}_2$, and either $\mathsf{CSP}(\Gamma)$ is in AC^0, or $\mathsf{CSP}(\Gamma)$ is complete for* LOGSPACE *under $\leq_m^{\mathrm{AC}^0}$-reductions.*

The theorem reveals an important corollary. We say that two languages are AC^0-*isomorphic*, if there is a bijection f between them such that f and f^{-1} can be computed in FAC^0. As with all mathematical structures, an isomorphism gives a very close relationship between two languages: they are "identical" as long as we are interested only in their features which are relevant for AC^0-computation.

Corollary 4.6.2 *For any Boolean constraint language Γ, $\mathsf{CSP}(\Gamma)$ is AC^0-isomorphic either to $0\Sigma^*$ or to the standard complete set for one of the following complexity classes:* NP, P, $\oplus$LOGSPACE, NL, LOGSPACE.

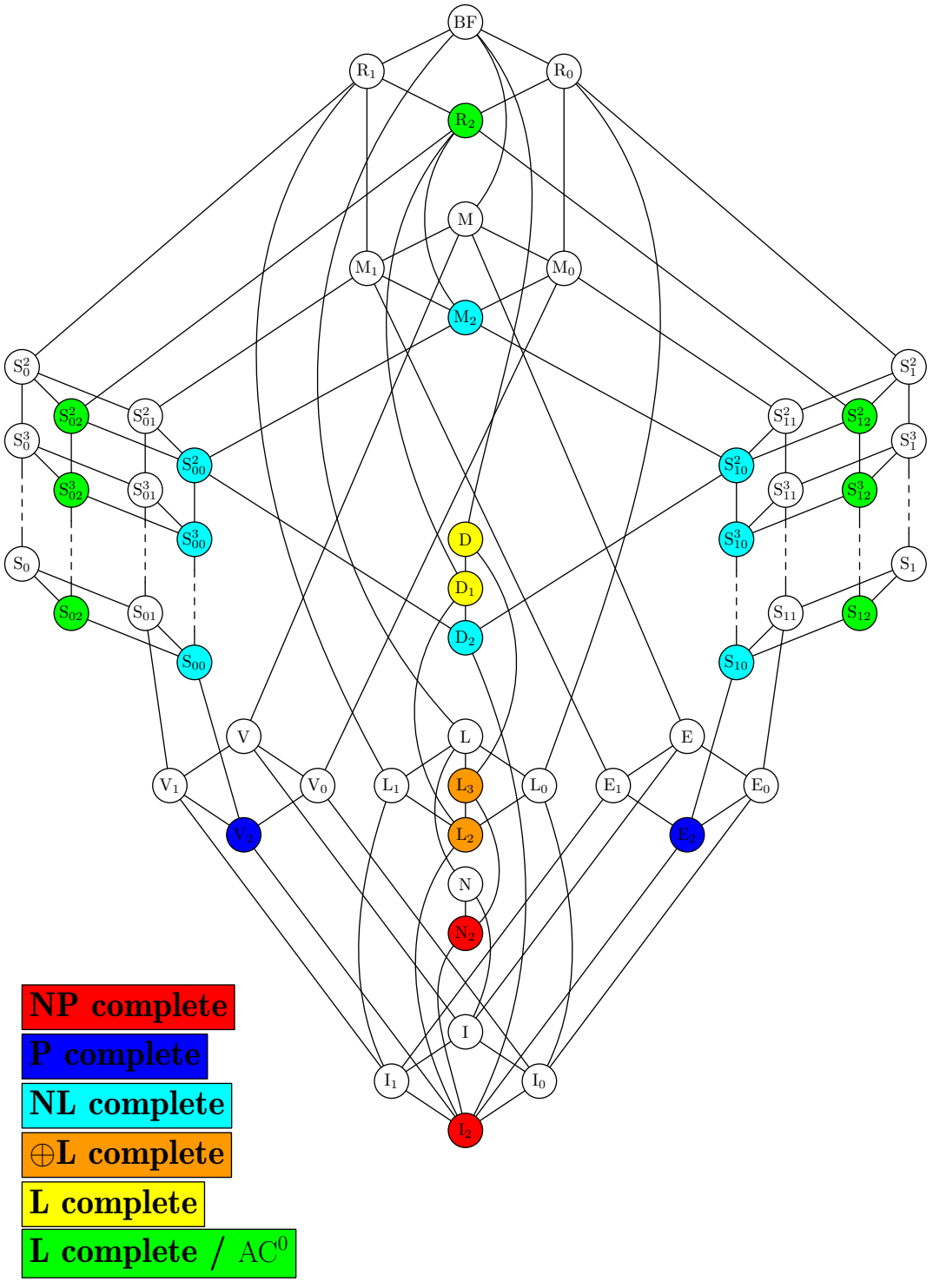

Figure 4.1: The complexity of CSP (Γ)

Proof. Theorem 4.6.1 implies that if $\mathsf{CSP}(\Gamma)$ cannot be solved in AC^0, then it is complete for one of the classes NP, P, NLOGSPACE, LOGSPACE, or $\oplus$LOGSPACE under $\leq_m^{\mathrm{AC}^0}$-reductions. By [Agr01], all complete problems for these classes are AC^0-isomorphic. Therefore, the constraint satisfaction problems considered in this chapter are, in particular, isomorphic to the standard complete problem in the corresponding class.

Now for the cases where $\mathsf{CSP}(\Gamma)$ can be solved in AC^0, any problem $A \in \mathrm{AC}^0$ can be reduced to $\mathsf{CSP}(\Gamma)$ via a length-squaring, invertible AC^0-reduction as follows: we first check if a given word x is in the language A, and if it is not, take a fixed syntactically incorrect instance, and add as many clauses as needed, which encode the string x, for example in the indices of the used variables. If x is in the language A, then by repeating a single satisfiable clause often enough, with only alternating the variables to code x, but never sharing a variable, a satisfiable instance of arbitrary length can be obtained from which the original word x can be obtained by an FAC^0-function.

It is obvious that any problem which can be solved in AC^0 can be AC^0-reduced to the problem $0\Sigma^*$ with a length-squaring and invertible AC^0-reduction, by simply first solving the problem, and depending on the answer computing either a 0 or a 1, and then simply copying the instance often enough to produce the required length. Therefore, AC^0 isomorphism to $0\Sigma^*$ now follows from [ABI97], since here it was shown that any two problems which can be reduced to one another with length-squaring and invertible AC^0-reductions are AC^0-isomorphic. □

We have completely answered the question for the exact complexity of the Boolean constraint satisfaction problem. In the process, we showed that the Galois connection can be refined to give a logarithmic space reduction, but a further refinement to $\leq_m^{\mathrm{AC}^0}$-reductions is not possible. The feature of the Galois connection stopping us from refining it to $\leq_m^{\mathrm{AC}^0}$-reductions is that the co-clone closure operator allows the introduction of equality clauses, which, as we have seen, can be used to express search problems in undirected graphs, and therefore gives hardness results for LOGSPACE. The presence of the equality relation can lead to difficulties for other reasons as well: While we can use variable identification to remove occurrences of the equality relation and end up with a formula that is satisfiable-equivalent to the original formula, this transformation does not give a fully equivalent formula. Therefore, when considering problems different from satisfiability, we expect to encounter difficulties. In Chapter 5, we will see that this is indeed a problem when we consider the equivalence of formulas, a context in which, naturally, we are interested in transformations preserving equivalence.

There is a second feature which is present in the co-clone closure $\langle\Gamma\rangle$ extending relations definable by simple Γ-formulas, which is the introduction of existentially quantified variables. While this feature is not problematical for the satisfiability problems studied in this chapter, it is for other questions. It can be shown that for the enumeration problem for constraint formulas, the existential quantifiers make an application of the Galois connection impossible: In [SS06a], it was shown that there are constraint languages Γ_1 and Γ_2, such that $\Gamma_1 \subseteq \langle\Gamma_2\rangle$, but the solutions of Γ_2-formulas can be enumerated by a polynomial-delay algorithms, and those for Γ_1-formulas cannot. Hence there are two possible reasons why the Galois connection can fail: If we consider complexity classes below logarithmic space, the connection fails due to the introduction of equality constraints.

For problems different than satisfiability, it fails due to the fact that the reduction provided by the Galois connection only preserves satisfiability, and not equivalence. While there does not seem to be an easy way out for the first problem, the second one can be solved by algebraic means: there is a modification of the standard Galois connection, which refines it to the case where we do not allow existential quantification in the coclone closure operator. However, this tool does not come without cost: instead of the well-known Post's lattice, we need to study the partial functions on the Boolean domain. In [SS06b], Henning Schnoor and Ilka Schnoor show how this refined Galois connection can be used to obtain classification results for Boolean constraint problems.

Chapter 5

Quantified Constraints: Decision and Counting

5.1 Introduction

We will generalize the results from Chapter 4 in several ways: first, instead of the simple Γ-formulas considered there, we will look at quantified formulas. The standard constraint satisfaction problem can be seen as a special case of the quantified problem, where all appearing variables are existentially quantified. The problem to decide whether a quantified Boolean formula evaluates to true is the standard complete problem for the class PSPACE. When we restrict the problem in limiting the number of quantifier alternations which may appear in the formula, we obtain problems which are complete for the corresponding levels of the polynomial hierarchy. For the Boolean case, a dichotomy theorem concerning these problems has been proven by Edith Hemaspaandra in [Hem04]. We will use the algebraic approach to constraint satisfaction problems to obtain an extension of her results to the non-Boolean case.

Second, we look at problems other than satisfiability: similarly to the unquantified case considered in [BHRV02], the question of equivalence for quantified formulas is of interest. We also study the model checking problem, which is closely related to satisfiability. Beyond decision problems, we consider the counting problem corresponding to quantified formulas, which is the following: given a quantified formula with free variables, determine the number of possible assignments to the free variables such that the formula evaluates to true.

Counting problems for propositional formulas have been known to give complete problems for natural counting complexity classes since [Val79a, Val79b], in which many of the basic notions for counting complexity were introduced. In the context of counting problems, the question for suitable reductions is often a difficult one. While for the decision problems, the concept of the many-one reduction is very often sufficient to prove hardness results, this is not so easy for counting problems. The natural generalization of the many-one reduction to this context is the *parsimonious reduction*. However, as we will soon show, this reduction fails to give completeness results for many problems which we would naturally consider to be complete for the complexity class in question. For certain counting problems in the context of Boolean constraint satisfaction problems, *comple-*

mentive reductions were introduced in [BCC+04]. These reductions close the relevant complexity classes, and therefore can be used to prove completeness results for natural problems arising in this context. However, a generalization to non-Boolean domains has not yet been successful.

For our results in this area, we mostly restrict ourselves to parsimonious reductions . Due to the above-mentioned problems, compromises need to be made here, since there are limitations to the application of parsimonious reductions in the constraint context. Our way out is a redefinition of the counting problem, and we believe that this is a good compromise between considering natural problems and achieving sharp complexity bounds: as we will see, the redefined problem is "equivalent" to the original one in a very natural way.

5.2 Counting Problems and Reductions

We give some background on the study of the complexity of counting problems. Except for the enumeration algorithms studied in Chapter 3, the problems considered up to now have been decision problems, i.e., problems for which the answer is either "yes" or "no." Computational problems where the goal is to compute the number of solutions of some formula cannot be expressed in this context in a satisfying way. However, complexity theory has developed powerful tools to deal with questions like this. In contrast to the case of enumeration problems, there is a rich theory of complexity classes and suitable reductions dealing with counting problems.

A *counting problem* is a computational problem where the task is to compute a function $f\colon \Sigma^* \to \mathbb{N}$. In the counting context, the class FP is considered to capture the notion of efficient computation. One of the most important higher counting classes is the class $\# \cdot \mathrm{P}$, which easily can be seen to be a superset of FP. This class, introduced by Leslie Valiant in [Val79a, Val79b] is defined as follows: A function $f\colon \Sigma^* \to \mathbb{N}$ is in $\# \cdot \mathrm{P}$, if there is a non-deterministic polynomial-time Turing machine M which, on input w, has exactly $f(w)$ accepting computation paths. Hence, one of the prototypical problems in the class $\# \cdot \mathrm{P}$ is the problem #SAT, which we define now.

Problem:	#SAT
Input:	A propositional formula φ
Output:	$\#\mathrm{SOL}(\varphi)$

It is obvious that this problem can be solved in $\# \cdot \mathrm{P}$, since a Turing machine can simply branch into as any paths as there are possible truth assignments for the formula, and on each computation path, accept if and only if the corresponding assignment satisfies the formula. Other typical problems which can be solved in this class are counting the number of colorings for a graph, counting perfect matchings in bipartite graphs, and computing the permanent of a Boolean matrix [Val79a].

To deal with counting problems which cannot be solved in $\# \cdot \mathrm{P}$, a general notion of counting classes has been introduced by Lane Hemaspaandra and Heribert Vollmer in [HV95]. We consider counting problems of a special form only: For alphabets Σ and Γ, let $R \subseteq \Sigma^* \times \Gamma^*$ be a binary relation between their strings. Then the counting problem $\#R$ is the following: Given a string $x \in \Sigma^+$, determine the cardinality of the

set $\{y \in \Gamma^* \mid (x, y) \in R\}$. The members of this set are also called *witnesses for* x. The relation R is called the *witness relation* of the counting problem. This concept can easily be seen to be a generalization of the class $\# \cdot \mathrm{P}$ as defined above, where R contains the pairs (x, y), for instances x and encodings y of an accepting path. For a satisfiable propositional formula, the set of its witnesses is the set of its satisfying assignments.

Definition Let $\mathcal{C}$ denote a complexity class of decision problems. The class $\# \cdot \mathcal{C}$ contains all counting problems whose witness relation satisfies the following conditions:

- There is a polynomial p such that for all $x \in \Sigma^*$, every witness y has a length restricted by $p(|x|)$,
- The problem "given x and y, is y a witness for x?" can be solved in $\mathcal{C}$.

It is obvious that this definition gives the same class $\# \cdot \mathrm{P}$ as the one above: for a fixed nondeterministic polynomial-time Turing machine M, the question "given x and y, is y an accepting computation path of M on input x?" can be solved in polynomial time, and such paths can be encoded with polynomial length. For the classes $\mathcal{C}$ of the polynomial hierarchy, the corresponding counting classes $\# \cdot \mathcal{C}$ form the *counting hierarchy*.

In order to compare the complexity of counting problems, we introduce a suitable reduction. Let $\#R$ and $\#S$ be counting problems with witness relations $R \subseteq \Sigma_1^* \times \Gamma_1^*$ and $S \subseteq \Sigma_2^* \times \Gamma_2^*$. A polynomial-time computable function $f\colon \Sigma_1^* \to \Sigma_2^*$ is called a *parsimonious reduction* from $\#R$ to $\#S$, if for all $x \in \Sigma_1^*$, the cardinality of the witness set for x (with respect to R) is the same as the cardinality of the witness set of $f(x)$ (with respect to S). It is obvious that the classes from the counting hierarchy are closed under this reduction.

Parsimonious reductions are the natural generalization of polynomial time many-one reductions to counting problems: in both cases we have a polynomial-time transformation between the possible input instances of the corresponding problems which preserve the answer to the question that we are interested in.

Later, we will see that this definition is too strict to capture some problems where we would naturally assume the complexity to be "as hard as it gets." In the constraint context, there are satisfiability problems, where the problem if a given formula has a solution at all is NP-complete, but the corresponding counting problem is not complete for $\# \cdot \mathrm{P}$ under parsimonious reductions (this can be proven without any complexity theoretical assumptions). A slightly less strict reduction type is the *weak parsimonious reduction* or *counting reduction*, as defined by Viktória Zankó [Zan91]. A counting reduction consists of two polynomial-time computable functions f and g. This pair forms a counting reduction from $\#R$ to $\#S$, if for all x, it holds that the cardinality of the witness set of x (with respect to R) is the same as $g(s)$, where s is the cardinality of the witness set of $f(x)$ (with respect to S). Hence, parsimonious reductions are the special case of counting reductions where the function g is simply the identity. Although the step from parsimonious to counting reductions seems subtle, there is a significant difference: unlike parsimonious reductions, counting reductions do not close the classes of the counting hierarchy, unless the counting hierarchy collapses (this was shown by Seinosuke Toda and Osamu Watanabe in [TW92]).

As commented above, for counting problems, it is often unclear what kind of reduction is suitable. We will give an easy example demonstrating that this is a problem in the constraint context. Assume we have some Boolean constraint language Γ, such that $\mathrm{Pol}(\Gamma)$ is the set N_2, i.e., contains only the identity and the negation. Due to Theorem 1.5.4, the constraint satisfaction problem $\mathsf{CSP}(\Gamma)$ is NP-complete. But it is easy to see that the counting problem $\#\mathsf{SAT}(\Gamma)$, which is the problem to determine the number of satisfying assignments of a given Γ-formula, cannot be complete for $\# \cdot \mathrm{P}$ under parsimonious reductions: since Γ is closed under negation, for every Γ-formula φ and every solution $I \models \varphi$, the assignment $\overline{I}$ obtained by negating the assignment of I for every single variable, is a solution for φ as well. Hence, the number of satisfying solutions for φ is always an even number, and therefore we cannot reduce any counting problem to $\#\mathsf{SAT}(\Gamma)$ where odd results may appear (it is obvious that such problems appear even in FP, consider the "counting problem" where the answer is always 3). However, we still want to regard this problem as one of the "hardest" problems in this context, in other words, we want it to be $\# \cdot \mathrm{P}$-complete, for reasons which we will now explain.

There are two possible ways to relate the counting problem for these formulas to a problem which can be shown to be $\# \cdot \mathrm{P}$-hard under parsimonious reductions. The first one is based on the following idea: Since the number $\#\mathrm{SOL}(\varphi)$ is always even, this means that the last bit of its representation as a binary number is fixed. Hence, the actual computational task is to determine the remaining bits of this number, or equivalently to compute $\frac{1}{2}\#\mathrm{SOL}(\varphi)$. It turns out that this problem can be proven to be $\# \cdot \mathrm{P}$-complete under parsimonious reductions. This is the main reason why we still want to regard the original problem as hard for $\# \cdot \mathrm{P}$: it is not natural to have a problem where to compute the string $f(x)$ is "more difficult" than to compute the string $f(x)$ augmented with a zero. It is obvious that similar issues arise when we consider non-Boolean domains, if for a constraint language Γ, the set $\mathrm{Pol}(\Gamma)$ contains every permutation on its domain D. We could again ask for the number of solutions divided by some constant here, but as it will turn out, this is not as easy as in the Boolean case. The other option is to not consider the number of solutions for a given formula, but the number of "classes of solutions," which are defined as equivalence classes under any permutation. If we know that each solution gives us a large class of solutions by applying every permutation of the domain, then the question how many of these classes exist becomes relevant.

For the Boolean case, there is a third possibility, which is to introduce a new type of reduction for counting problems. In [BCC$^+$04], the notion of a complementive reduction was introduced to deal with the above mentioned issue. This reduction can also be applied to the quantified problems we consider here, but since it does not seem to generalize to the non-Boolean case, we do not consider this reduction. Completeness results of the Boolean counting problems we consider here using complementive reductions were achieved in the technical report [BBC$^+$05], which contains some of the results of this chapter.

5.3 Quantified Constraint Formulas

In the constraint satisfaction problem, we are essentially asking if a sentence where all appearing variables are existentially quantified is true. In this chapter, we additionally

allow variables to be universally quantified. We only consider quantified formulas in prenex form, i.e., formulas of the form $\psi = Q_1X_1Q_2X_2 \dots Q_nX_n\varphi(x_1, \dots, x_k, y_1, \dots, y_k)$, where $Q_i \in \{\exists, \forall\}$, $X_1, \dots, X_n$ are sets of variables which contain $x_1, \dots, x_k$, φ is a propositional formula, called the *kernel* of ψ, and where $y_1, \dots, y_k$ are additional variables not appearing in the *quantifier block* $Q_1X_1Q_2X_2 \dots Q_nX_n$. These variables are called *free variables*. We denote the set of free variables of φ with $\mathrm{FVAR}(\varphi)$. There is a canonical order on the variables appearing in a quantified formula which mirrors the dependence of the possible assignments. This order, which we will denote with $<_\varphi$, is defined as follows: restricted to the free variables, the order is arbitrary. For a free variable y and a quantified variable x, $y <_\varphi x$ always holds. For two quantified variables x_1 and x_2, $x_1 <_\varphi x_2$ holds if the quantification of x_1 happens before the one of x_2 in the quantifier block.

Let X be the set of variables in ψ which are universally quantified, i.e., let $X = \bigcup \{X_i \mid Q_i = \forall\}$, and similarly let Y be the set of existentially quantified variables in ψ. If the kernel φ of ψ is a propositional formula where variables take values from the domain D, then an assignment $I \colon \mathrm{FVAR}(\psi) \to D$ is a *satisfying assignment*, or a *solution* of ψ, if the following holds: For every assignment $U \colon X \to D$ for the universally quantified variables, there is a function $E \colon Y \to D$ such that the assignment $I \cup U \cup E$ satisfies φ, and for each $x_i \in Y$, the value $E(x_i)$ depends only on the assignment I and the values $U(x_j)$ for $x_j <_\varphi x_i$, i.e., the values of the variables universally quantified before the quantification of x_i occurs (note that in the literature, often a dependence on existential variables x_j quantified earlier is allowed as well, but this can be avoided, by coding the behavior of $E(x_j)$ into the function $E(x_i)$). Here the assignment $I \cup U \cup E$ is defined as usual for the union of functions with disjoint domains. It is straightforward to verify that this definition is equivalent to the standard definition of truth in a quantified formula. As usual, we denote the set of solutions of φ with $\mathrm{SOL}(\varphi)$. A quantified formula is *closed* if it does not have any free variable. Such a formula is either true or false.

It should be noted that even though quantifiers are a typical feature of predicate logic, they do not enrich the propositional language in our context: For a Boolean formula φ in which the variable x occurs, the formula $\forall x\varphi$ can be considered as a shorthand for $\varphi[x/0] \wedge \varphi[x/1]$, and similarly, $\exists x\varphi$ is equivalent to $\varphi[x/0] \vee \varphi[x/1]$. Therefore, a quantified formula is only an abbreviation for a purely propositional formula. Obviously, analogous constructions can be used in the non-Boolean case, as long as the domain remains finite. However, quantifiers allow us to present formulas in a succinct way: The quantifier-removing expansion described above will usually lead to a formula which is exponential in length compared to the original. Therefore it is not surprising that QBF, the truth evaluation for quantified formulas, is significantly more difficult than the same problem for the quantifier-free formulas considered in Chapter 2. In fact, this is one of the standard complete problems for PSPACE. In a similar way, it can also be restricted by limiting the propositional operators allowed to appear in the formula. This problem has been considered in [RW00] for Boolean circuits and in [Sch78, CKS01] for constraint formulas, and full complexity classifications were achieved.

We now study problems for quantified formulas where the propositional kernel is in constraint form. For a constraint language Γ, a *quantified Γ-formula with $k-1$ quantifier alternations*, is a formula of the form $Q_1X_1Q_2X_2 \dots\dots Q_kX_k\varphi$, where φ is a Γ-formula,

and $Q_1, \ldots, Q_k$ are either $\exists$ or $\forall$, where $Q_i \neq Q_{i+1}$ for $i \in \{1, \ldots, k-1\}$, and $X_1, \ldots, X_k$ are sets of variables appearing in φ. Depending on whether we start with an existential or a universal quantifier, we make the following distinction:

Definition Let Γ be a constraint language, and let k be a natural number.

- A $\Sigma_k(\Gamma)$-formula is a quantified Γ-formula with at most $k-1$ quantifier alternations where the first quantifier is $\exists$,
- A $\Pi_k(\Gamma)$-formula is a quantified Γ-formula with at most $k-1$ quantifier alternations where the first quantifier is $\forall$.

Since we are dealing with formulas in conjunctive normal form, the question if the last quantifier is existential or universal makes a significant difference: the problem to decide whether a given Γ-formula is satisfiable is NP-complete in general. However, the problem to decide if such a formula is a tautology is trivially solvable in P, since this is the case if and only if each of the clauses is a tautology. For a fixed constraint language Γ, this can be tested in polynomial time. Therefore, when dealing with formulas from the constraint context, adding a $\forall$-quantifier after the last $\exists$-quantifier does not raise the complexity of the problems anymore. Hence, we are interested in formulas where the last quantifier is existential. Therefore we define the following:

Definition Let Γ be a constraint language, and let k be a natural number. If k is odd, then a $\mathsf{QCSP}(\Gamma)_k$-formula is a $\Sigma_k(\Gamma)$-formula. If k is even, then a $\mathsf{QCSP}(\Gamma)_k$-formula is a $\Pi_k(\Gamma)$-formula.

In addition, a $\mathsf{QCSP}(\Gamma)$-formula is a $\mathsf{QCSP}(\Gamma)_k$-formula for an arbitrary k, i.e., the set of $\mathsf{QCSP}(\Gamma)$-formulas is the union of the sets of all $\mathsf{QCSP}(\Gamma)_k$-formulas. We now define the problems that we are interested in for these formulas. In the following, let Γ be a constraint language over the domain D, and let k be a natural number. The first problem we consider is a variation of the classical QBF problem:

Problem:	$\mathsf{QCSP}_k(\Gamma)$
Input:	A closed $\mathsf{QCSP}_k(\Gamma)$-formula φ
Question:	Is φ true?

A closely related problem is the following, which is called the *model checking problem for* $\mathsf{QCSP}_k(\Gamma)$*-formulas*. This can be seen as a slight generalization of the $\mathsf{QCSP}_k(\Gamma)$-problem, where we can additionally fix some values for the free variables.

Problem:	$\mathsf{QMCK}_k(\Gamma)$
Input:	A $\mathsf{QCSP}_k(\Gamma)$-formula φ and an assignment $I \colon \mathrm{FVAR}(\varphi) \to D$
Question:	Does I satisfy φ?

The final decision problem which we study is the problem to decide whether two formulas are equivalent. Equivalence in this context is defined as solution-equivalence, i.e., formulas φ_1 and φ_2 are equivalent if every assignment $I \colon \mathrm{FVAR}(\varphi_1) \cup \mathrm{FVAR}(\varphi_2) \to D$ is a solution for φ_1 if and only if it is a solution for φ_2. As usual, we write this as $\varphi_1 \equiv \varphi_2$. The following is called the *equivalence problem for* $\mathsf{QCSP}_k(\Gamma)$*-formulas*:

Problem: $\mathsf{QEQUIV}_k(\Gamma)$
Input: Two $\mathsf{QCSP}_k(\Gamma)$-formulas φ_1 and φ_2
Question: Does $\varphi_1 \equiv \varphi_2$ hold?

Formulas with free variables obviously also give rise to a canonical counting problem, namely the problem to determine the number of satisfying solutions to the formula, i.e., the number $\#\mathrm{SOL}(\varphi)$. Hence, we define the following:

Problem: $\#\mathsf{QCSP}_k(\Gamma)$
Input: A $\mathsf{QCSP}_k(\Gamma)$-formula φ
Output: $\#\mathrm{SOL}(\varphi)$

It is easy to see that the problem $\#\mathsf{QCSP}_k(\Gamma)$ can be regarded as a counting problem in the sense defined above, where the witness relation contains the tuples (φ, I), where φ is a formula, and I a satisfying assignment. It is obvious that such an assignment can be represented by a string which is polynomial in the length of the formula. Due to Theorem 1.3.2, it is not surprising that computational problems related to these formulas have a connection to the polynomial hierarchy. We will show that the decision problems just defined are indeed either complete for some level of the polynomial hierarchy, or can be decided in polynomial time.

Recall that in our discussion earlier, we explained that for constraint languages invariant under all permutations of the domain, we cannot hope to achieve parsimonious reductions from arbitrary problems in classes even as low as FP. We now formalize the approach mentioned above: to instead count the number of classes of solutions. The following definitions are meant to capture this idea. As usual, let S_D denote the group of permutations on the set D.

Definition

- A constraint language Γ over a domain D is called *permutative*, if $\mathrm{Pol}(\Gamma)$ contains every unary permutation of D, i.e., if $S_D \subseteq \mathrm{Pol}(\Gamma)$.
- Let X be a finite set of variables, and let A be a set of assignments $I \colon X \to D$ for the variables to some finite domain D. We say that A is *permutative*, if it is closed under permutations, i.e., if for every $I \in A$, and every bijection $\Pi \in S_D$ the function $\Pi \circ I$ is a member of A.
- For a permutative set A of assignments from X to D, we say that two assignments I_1 and I_2 are *equivalent*, if there is a permutation Π of the domain D such that $I_1 = \Pi \circ I_2$.
- The set of equivalence classes of a set A under the equivalence relation defined above is denoted with $^{\mathrm{SOL}(\varphi)}/_{S_D}$.

Note that a Boolean constraint language is permutative if and only if it is complementive. For a permutative constraint language Γ and a natural number k, we define the following problem:

Problem: $\#\mathsf{EQCSP}_k(\Gamma)$
Input: A $\mathsf{QCSP}_k(\Gamma)$-formula φ
Output: $|^{\mathrm{SOL}(\varphi)}/_{S_D}|$

For this problem to be well-defined, we need to prove the following Proposition:

Proposition 5.3.1 *Let Γ be a permutative constraint language, let $k \geq 1$, and let φ be a QCSP_k-formula. Then* $\mathrm{SOL}(\varphi)$ *is permutative.*

Proof. In [Jea98] (also see [BBJK03]), it was show that any relation which can be described by a $\mathsf{QCSP}(\Gamma)$-formula inherits all *surjective polymorphisms* of Γ. Since permutations on the domain D are by definition surjective on D, this in particular applies to permutations of the domain. □

As mentioned before, the other option instead of considering the number of equivalence classes is to consider the number of solutions divided by 2 in the case of a Boolean constraint language which is complementive. For the case of arbitrary finite domains D, the canonical generalization is to consider the number of solutions divided by $|S_D| = |D|!$ However, unlike in the Boolean case, these problems are not the same. Consider, for example, a domain D with cardinality 3, and a formula in which only one variable occurs. Then the number of solutions will never be a multiple of $|D|! = 6$, since there are only 3 possible assignments. In general, for an assignment I for the free variables of some formula, if two permutations Π_1, Π_2 of D only differ for values which do not appear in I, then Π_1 and Π_2 applied to I give the same solution. Therefore, the question for satisfying solutions divided by the number $|D|!$ gives a different problem. This cannot happen in the Boolean case, since there are only two permutations on the Boolean domain, and they differ for all values. This problem also only makes sense to consider for formulas in which the solution set is indeed a multiple of $|D|!$. This obviously holds for formulas which have $|D|$ free variables which are forced to take different values using disequality constraints. For some assignment I to these variables which satisfies the inequality clauses and two different permutations Π_1 and Π_2 on the domain D, it is obvious that $\Pi_1 \circ I$ and $\Pi_2 \circ I$ differ. Therefore, one could study the problem to compute, for such a formula, the number of its solutions divided by $|D|!$. This problem seems too artificial to consider it, but for completeness, the reduction in Theorem 5.5.2 shows that for these formulas, the mentioned problem is also complete for the corresponding level of the counting hierarchy under parsimonious reductions.

However, for Boolean constraint languages which are complementive, the problems to consider the number of solutions divided by two and the number of equivalence classes of solutions are indeed the same. This follows directly from the above discussion.

Proposition 5.3.2 *Let Γ be a constraint language over a Boolean domain D such that Γ is complementive, and let φ be a $\mathsf{QCSP}_k(\Gamma)$-formula for some $k \in \mathbb{N}$. Then* $|\mathrm{SOL}(\varphi)| = 2 \cdot |^{\mathrm{SOL}(\varphi)}/_{S_D}|$.

We now show that the algebraic framework applied to the unquantified constraint satisfaction problem in Chapter 4 also works for quantified problems. Similarly as in

Chapter 4, we will have to be careful with the equality constraints resulting from the application of the Galois connection. In the problems we consider here, the equality relation does not give problematic complexity in the reductions themselves (since we are talking about classes in the polynomial hierarchy, logspace reductions are strict enough for our study), but other than in the satisfiability context, removal of equality constraints by identifying variables does not preserve all properties of the formulas which we are interested in. However, if our constraint languages can implement the equality relation as defined in Chapter 4, then these problems do not arise, as the following Proposition shows:

Proposition 5.3.3 *Let Γ_1 and Γ_2 be constraint languages such that Γ_2 implements equality and $\mathrm{Pol}(\Gamma_2) \subseteq \mathrm{Pol}(\Gamma_1)$ holds. Then for any $k \geq 1$, the following holds:*

1. $\mathsf{QCSP}_k(\Gamma_1) \leq_m^{\log} \mathsf{QCSP}_k(\Gamma_2)$,
2. $\mathsf{QMCK}_k(\Gamma_1) \leq_m^{\log} \mathsf{QMCK}_k(\Gamma_2)$,
3. $\mathsf{QEQUIV}_k(\Gamma_1) \leq_m^{\log} \mathsf{QEQUIV}_k(\Gamma_2)$,
4. $\#\mathsf{QCSP}_k(\Gamma_1)$ *reduces to* $\#\mathsf{QCSP}_k(\Gamma_2)$ *with a parsimonious reduction (which can be computed in logspace).*
5. $\#\mathsf{EQCSP}_k(\Gamma_1)$ *reduces to* $\#\mathsf{EQCSP}_k(\Gamma_2)$ *with a parsimonious reduction (which can be computed in logspace).*

Proof. We show how we can transform a quantified Γ_1-formula φ_1 into a quantified Γ_2-formula φ_2. Since $\mathrm{Pol}(\Gamma_2) \subseteq \mathrm{Pol}(\Gamma_1)$, we can express every relation from Γ_1 appearing in φ_1 equivalently as a conjunction of relations from $\Gamma_2 \cup \{=\}$ with additional existentially quantified variables, due to Theorem 1.5.2. The additional existential variables can be put into the last quantifier block of the resulting formula φ_2, which contains existentially quantified variables due to the definition of $\mathsf{QCSP}_k(\Gamma)$-formulas. The formula φ_2 constructed in this way obviously is equivalent to φ_1. Any occurring equality constraints can be removed by using the Γ_2-implementation of the equality relation which exists due to the prerequisites. Since the resulting formula is equivalent to the original, it is obvious that this is a correct reduction for all of the involved problems. □

The above proposition in particular shows that if $\mathrm{Pol}(\Gamma_2) \subseteq \mathrm{Pol}(\Gamma_1)$, then we can reduce our problems over the constraint language Γ_1 to the same problem over the constraint language $\Gamma_2 \cup \{=\}$. We now need to remove the occurrences of the equality relation. This can easily be done for all of our problems, except for the equivalence problem.

Lemma 5.3.4 *Let Γ_1 be a constraint language over the domain D, and let $k \geq 1$. Then the following holds:*

1. $\mathsf{QCSP}_k(\Gamma \cup \{=\}) \leq_m^{\log} \mathsf{QCSP}_k(\Gamma)$,
2. $\mathsf{QMCK}_k(\Gamma \cup \{=\}) \leq_m^{\log} \mathsf{QMCK}_k(\Gamma)$,
3. $\#\mathsf{QCSP}_k(\Gamma \cup \{=\})$ *reduces to* $\#\mathsf{QCSP}_k(\Gamma)$ *with a parsimonious reduction (which can be computed in logspace).*

4. $\#\mathsf{EQCSP}_k(\Gamma \cup \{=\})$ *reduces to* $\#\mathsf{EQCSP}_k(\Gamma)$ *with a parsimonious reduction (which can be computed in logspace).*

Proof. We show how equality constraints from a given formula φ can be removed in a way which preserves the properties of φ which are relevant to the considered problems. We first check if in φ, there are variables x and y such that y is a universally quantified variable, and $x <_\varphi y$, and there is a clause $x = y$ in φ. If this is the case, φ is clearly unsatisfiable, and we print out the clause $\forall x_1 \dots x_n R(x_1, \dots, x_n)$ for a non-full relation $R \in \Gamma$ (which we can assume to exist without loss of generality, otherwise all of our problems are trivial). If the case above does not appear, then all cliques of variables connected with equality clauses in φ consist of variables where at most the first one, which we denote by x, is universally quantified. Hence, the following is an equivalent transformation of the formula: remove the existential variables appearing in the clique from the quantifier block, and in the formula, replace them with x.

The resulting formula φ' can be computed in logarithmic space, since the dominating procedure here is graph accessibility in an undirected graph, and this can be performed in logspace due to Theorem 1.3.5. It now remains to remove equality occurrences between free variables. We do this by identifying variables which are connected with equality constraints. For the counting problems, it is obvious that identifying variables which are forced to take the same value with equality constraints does not change the number of solutions of a formula, or the number of equivalence classes of solutions. In particular, the truth value of a closed formula is invariant under this transformation.

For the model checking problem, if we identify two variables x and y in renaming every occurrence of y to x, and the assignment I given in the instance does not give the same value to x and y, we produce a false instance. In this way we ensure that the original assignment is a solution for the original formula if and only if it is one for the new formula. □

The combination of the results above gives the following corollary:

Corollary 5.3.5 *Let* Γ_1 *and* Γ_2 *be constraint languages over the domain* D *such that* $\mathrm{Pol}(\Gamma_2) \subseteq \mathrm{Pol}(\Gamma_1)$. *Then the following holds:*

1. $\mathsf{QCSP}_k(\Gamma_1) \leq_m^{\log} \mathsf{QCSP}_k(\Gamma_2)$,
2. $\mathsf{QMCK}_k(\Gamma_1) \leq_m^{\log} \mathsf{QMCK}_k(\Gamma_2)$,
3. $\#\mathsf{QCSP}_k(\Gamma_1)$ *reduces to* $\#\mathsf{QCSP}_k(\Gamma_2)$ *with a parsimonious reduction (which can be computed in logspace).*
4. $\#\mathsf{EQCSP}_k(\Gamma_1)$ *reduces to* $\#\mathsf{EQCSP}_k(\Gamma_2)$ *with a parsimonious reduction (which can be computed in logspace).*

Proof. By Proposition 5.3.3, and since $\Gamma_2 \cup \{=\}$ obviously implements equality, we conclude that the reductions in all cases can be constructed for reducing the problem over the constraint language Γ_1 to the problem over the language $\Gamma_2 \cup \{=\}$. Now an application of Lemma 5.3.4 shows that this problem can be reduced to the problem over the constraint language Γ_2, as claimed. □

Note that the equivalence problem is absent from the above list. This is because for this problem, variable identification does not necessarily preserve the properties of the formulas which we are interested in: If we identify the variables x and y, in renaming every occurring y to x, then the new formula is completely independent of the variable y, while the old formula might not be. However, we will see that this is not a major problem, and we can still achieve a full complexity classification for the Boolean quantified equivalence problem.

For the other problems, the results above show that the logspace complexity of our problems depends only on the set of polymorphisms of the constraint languages, and again it suffices to show the complexity results for a single representative of a given co-clone. In most cases, the relations from Table 1.2 serve as members of these representative constraint language. For the non-Boolean case, we need a generalization of the Boolean relation NAE to arbitrary domains: for a finite domain D, the relation NAE^D consists of the three-tuples over the domain D where not all elements take the same value, i.e., $\mathrm{NAE}^D =_{\mathrm{def}} \{(x,y,z) \in D^3 \mid |\{x,y,z\}| \geq 2\}$.

Since for two domains with the same cardinality, all constraint satisfaction problems over these domains have the exact same structure, we often only write $=^m$, $\neq^m$ or NAE^m to denote the relations $=^D$, $\neq^D$, or NAE^D over an arbitrary but fixed domain D with $|D| = m$.

As a start, we prove general complexity bounds for the problems we consider. Canonical upper bounds easily follow from Theorem 1.3.2, and previous work done in this area shows that there are constraint languages for which lower bounds matching these upper bounds can be proven. Note that depending on whether our formulas start with an existential or with a universal quantifier, our problems $\mathsf{QCSP}_k(\Gamma)$ are in Σ^p_k or in Π^p_k, due to Theorem 1.3.2. For easier notation and to avoid case distinctions, we define the following: For k odd, let $\mathrm{QPH_k} =_{\mathrm{def}} \Sigma^p_k$, and for k even, let $\mathrm{QPH_k} =_{\mathrm{def}} \Pi^p_k$.

Proposition 5.3.6 *Let k be a natural number. Then the following holds:*

1. *Let Γ be a constraint language on the finite domain D.*
   - $\mathsf{QCSP}_k(\Gamma) \in \mathrm{QPH_k}$,
   - $\mathsf{QMCK}_k(\Gamma) \in \mathrm{QPH_k}$,
   - $\mathsf{QEQUIV}_k(\Gamma) \in \Pi^p_{k+1}$,
   - $\#\mathsf{QCSP}_k(\Gamma) \in \# \cdot \mathrm{QPH_k}$,
   - *If Γ is permutative, then* $\#\mathsf{EQCSP}_k(\Gamma) \in \# \cdot \mathrm{QPH_k}$,
2. $\#\mathsf{QCSP}_k(\Gamma_{\mathsf{3SAT}})$ *is* $\# \cdot \mathrm{QPH_k}$*-complete.*

Proof. 1. For both QCSP_k and QMCK_k the upper complexity bound is clear, since evaluating formulas with $k-1$ alternations is a typical problem for the corresponding level of the polynomial hierarchy due to Theorem 1.3.2. The upper bound for $\#\mathsf{QCSP}_k$ is obvious for the same reason: by the very definition of the counting classes, and with the observation that a solution to a formula can always be represented in polynomial length, it follows that for any constraint language Γ, if $\mathcal{C}$ is a complexity class such that $\mathsf{QMCK}_k(\Gamma)$ is in $\mathcal{C}$, then $\#\mathsf{QCSP}_k(\Gamma) \in \# \cdot \mathcal{C}$. For the

problem #EQCSP_k, the witnesses consist of the equivalence classes from ${}^{\mathrm{SOL}(\varphi)}/_{S_D}$. Since the size of the domain D is a constant, so is $|D|!$, and therefore these classes, containing at most $|D|!$ elements, are polynomial in the length of φ. Since the set of solutions is permutative due to Proposition 5.3.1, it suffices to check for one of the assignments contained in the class if it is a solution for the formula. This test can be performed in $\mathrm{QPH_k}$.

For the $\mathsf{QEQUIV}_k(\Gamma)$ upper bound, observe that two $\mathsf{QCSP}_k(\Gamma)$-formulas φ_1 and φ_2 are equivalent if and only if for all $(\alpha_1, \ldots, \alpha_n)$, it holds that $(\alpha_1, \ldots, \alpha_n)$ is a model for φ_1 if and only if the tuple is a model for φ_2. Since this condition can be checked in Σ_k^p or Π_k^p, it follows that equivalence can be tested in Π_{k+1}^p.

2. It is well-known that the problem #$\mathsf{3SAT}_k$ to compute the number of solutions of a 3CNF-formula with at most $k-1$ quantifier alternations, where the last quantifier is $\exists$, is complete for $\# \cdot \mathrm{QPH_k}$. The result was stated for arbitrary CNF-formulas in [DHK05], and can be derived from the proofs in [Wra77]. In the same way as Corollary 1.5.1, this is exactly the same problem as #$\mathsf{QCSP}_k(\Gamma_{\mathsf{3SAT}})$.

□

The following result was stated in [Sch78] without proof. A proof can be found in [CKS01], where the theorem appears as Theorem 6.12. Note that the theorem shows that the problem is still solvable in polynomial time even if the number of alternations is not bounded by a constant.

Theorem 5.3.7 ([Sch78],[CKS01]) *Let Γ be a Boolean constraint language such that Γ is Schaefer. Then there is a polynomial time algorithm, which for any $\mathsf{QCSP}(\Gamma)_k$-formula decides if it is true.*

5.4 Affine Constraint Languages

Affine constraint languages play a special role in our problems. As it will turn out later, these are the only Boolean cases where our counting problems can be solved in polynomial time (unless $\# \cdot \mathrm{P} = \mathrm{FP}$). Since the algorithm for the counting and for the equivalence problem use the same construction, we present our methods for dealing with affine languages in this section.

In the non-quantified case, affine languages lead to tractable cases because a formula over an affine language Γ can be seen as a system of linear equations over the field of natural numbers modulo 2. The well-known Gaussian elimination algorithm can be used to solve such equation systems, and hence can be applied to solve various constraint problems for these languages.

We now show how we can apply these methods to quantified formulas. The set of solutions of an (inhomogeneous) linear equation system with n variables is either empty or an affine subscape of $\{0,1\}^n$, i.e., of the following form:

$$S = \{\mathbf{v} + \alpha_1 \cdot \mathbf{b_1} + \cdots + \alpha_k \cdot \mathbf{b_k} \mid \alpha_1, \ldots, \alpha_k \in \{0,1\}\},$$

where $\mathbf{v}, \mathbf{b_1}, \ldots, \mathbf{b_k}$ are vectors from $\{0,1\}^n$, and the set $\{\mathbf{b_1}, \ldots, \mathbf{b_k}\}$ is *linear independent*, i.e., none of the $\mathbf{b_i}$ is the all-zero vector, and none of them can be written as the sum of a selection of other $\mathbf{b_i}$s. We call the set $\{\mathbf{b_1}, \ldots, \mathbf{b_k}\}$ the *base vectors of* S, and the vector $\mathbf{v}$ the *affine point of* S. The number k is the *dimension* of the space S. We call this representation the *affine representation of* $\mathrm{SOL}(\varphi)$, and we say that S is generated by the affine point $\mathbf{v}$ and the base vectors $\mathbf{b_1}, \ldots, \mathbf{b_k}$. The linear independence ensures that different choices of $\alpha_1, \ldots, \alpha_k$ lead to different vectors in the sum $\mathbf{v} + \alpha_1 \cdot \mathbf{b_1} + \cdots + \alpha_k \cdot \mathbf{b_k}$, and hence the cardinality of S is exactly 2^k. Therefore, counting problems can easily be solved if we can determine this representation of the set $\mathrm{SOL}(\varphi)$ for some Γ-formula.

Similarly, equality can efficiently be tested for two such sets S_1 and S_2 when given in the above representation as an affine vector space: $S_1 \subseteq S_2$ holds if and only if the vector $\mathbf{v}$ in the defining equation for the set S_1 is a member of S_2, and each base vector of S_1 can be written as sum of base vectors from S_2. Both of these conditions can be solved by simply solving a system of linear equations. Hence, equivalence for Γ-formulas can be tested in polynomial time as well, if we can generate the affine representation of the solution set for any Γ-formula in polynomial time. Obviously, the question if a formula without any free variables evaluates to true can be solved with the same procedure. Hence, the following theorem is the key to all of our tractability results for affine constraint languages:

Theorem 5.4.1 *Let Γ be an affine Boolean constraint language. Then there exists a polynomial-time procedure which computes, for a given $\mathsf{QCSP}_k(\Gamma)$-formula φ, the affine representation of* $\mathrm{SOL}(\varphi)$, *or decides that the set is empty.*

Proof. Since the proof of Proposition 5.3.3 gives an equivalent transformation of the involved formulas, we can assume that $\Gamma = \{\mathrm{EVEN}^4, x, \overline{x}, =\}$. We now rewrite the kernel of φ as a system of linear equations over $\mathrm{GF}(2)$ as follows: For variables z_1, z_2, z_3, z_4, the clause $\mathrm{EVEN}^4(z_1, z_2, z_3, z_4)$ is equivalent to $z_1 + z_2 + z_3 + z_4 = 0$, where addition is over $\mathrm{GF}(2)$. A clause $z_1 = z_2$ can be rewritten as $z_1 + z_2 = 0$. The clause z_1 obviously is equivalent to $z_1 = 1$, and the clause $\overline{z_1}$ is equivalent to $z_1 = 0$. Hence, we can assume that the kernel of φ is given as a set of linear equations.

We now describe an iterative transformation procedure converting the system given by φ into an equivalent system ψ which only contains free variables. For every equation in the kernel of φ, and let z be the $<_\varphi$-maximal variable appearing in the equation, where $<_\varphi$ is the order of variables in the quantified formula φ as defined previously. If z is a free variable, then this equation only contains free variables, and we do not perform any operation on this equation. If z occurs an even number of times, we can simply remove it and get an equivalent equation. Otherwise, if z is universally quantified, then this equation is contradictory, and hence φ is unsatisfiable. In this case, let ψ be the equation system containing the single equation $0 = 1$.

Now, assume that z is an existentially quantified variable, and let the equation be $t_1 + \cdots + t_k + z = t$, where $t_1, \ldots, t_k$ and t are variables or constants. Now, since z is maximal in the $<_\varphi$-order, an assignment to the existentially quantified variables of φ assigns the value to z as the "last" value, i.e., depending on all variables before. Therefore, in order to satisfy the equation, the only possible assignment for z is the value $t_1 + \cdots + t_k + t$. Therefore, the variable z is redundant, since its assignment is completely

determined by the assignment of the preceding variables. We can therefore remove z from the quantifier block, and replace z with the sum $t_1 + \cdots + t_k + t$ in every equation where z appears. Afterwards, in any equation, using that $z + z$ is equivalent to 0, we remove multiple occurrences of any variables or constants. Finally, we remove any variable which does not appear in any equation anymore from the quantifier block.

We perform these steps until no changes occur in the formula anymore. In each step, at least one quantified variable is removed. Hence the process terminates after linearly many steps. Each step can clearly be performed in polynomial time: since each variable and constant only appears once in each equation, the equations themselves are short. The number of equations is not increased at any time. It is obvious that each step performed by the algorithm does not change the solution set of the equation system.

After the algorithm finishes, we end up with an equation system in which no quantified variable appears anymore: As long as some quantified variable remains, there is an equation on which an operation is performed by the above steps. Therefore, we have constructed a system of linear equations in which only the free variables from φ occur. For this system, using the classical Gaussian elimination algorithm from linear algebra, it is easy to determine an affine representation of the solution set of φ. □

Note that the above proof, although removing all quantifiers from a given formula, does not directly provide a reduction to the unquantified case of the problems we are interested in, since the length of the equations appearing cannot be bounded by a constant, and hence cannot be expressed by a finite, fixed constraint language. Theorem 5.4.1, the discussion above, and Proposition 5.3.2 imply the following Corollary. Again, note that the polynomial determining the running time of the algorithm is independent of the number k.

Corollary 5.4.2 *Let Γ be a Boolean constraint language, such that Γ is affine. Then there is a polynomial-time algorithm which for all $k \in \mathbb{N}$, solves $\mathsf{QCSP}_k(\Gamma)$, $\#\mathsf{QCSP}_k(\Gamma)$, and $\mathsf{QEQUIV}_k(\Gamma)$. If Γ is complementive, then $\#\mathsf{EQCSP}_k(\Gamma)$ can be solved in polynomial time.*

5.5 Complexity Results for Counting

Proposition 5.3.6 shows that the counting problem for the constraint language Γ_{3SAT} is hard for the corresponding level of the counting hierarchy. Due to Corollary 5.3.5, and since I_2 is a subset of every clone appearing in Post's lattice, this implies that this problem is hard for any constraint language Γ which is invariant only under the functions from the clone I_2. We now show that if we add negation to the invariants, then the problem $\#\mathsf{EQCSP}$ is hard for the corresponding class. As mentioned in our earlier discussion, this is the best result that we can hope for in this context, since the problem $\#\mathsf{QCSP}(\Gamma)_k$ is not even hard for FP under parsimonious reductions.

Lemma 5.5.1 *Let Γ be a Boolean constraint language such that $\mathrm{Pol}(\Gamma) = \mathrm{N}_2$, and let k be a natural number. Then $\#\mathsf{EQCSP}_k(\Gamma)$ is complete for $\# \cdot \mathrm{QPH_k}$ under parsimonious reductions,*

Proof. The upper complexity bound follows from Proposition 5.3.6. For the lower bound, due to Corollary 5.3.5, it is sufficient to show that there is one Boolean relation R such that $\mathrm{Pol}(R) \supseteq \mathrm{N}_2$, and the hardness result holds. Due to Proposition 5.3.6, we know that $\#\mathsf{QCSP}_k(\Gamma_{\mathsf{3SAT}})$ is hard for $\# \cdot \Sigma_k^p$ under parsimonious reductions. Since $\mathrm{Pol}\left(\mathrm{R}_{1/3}\right) = \mathrm{I}_2$ (see Table 1.2), Corollary 5.3.5 implies that $\#\mathsf{QCSP}_k(\mathrm{R}_{1/3})$ is hard for $\# \cdot \Sigma_k^p$ under parsimonious reductions as well. We show that $\#\mathsf{QCSP}_k(\mathrm{R}_{1/3})$ reduces to $\#\mathsf{QCSP}_k(\mathrm{R}_{2/4})$ with a reduction which doubles the number of solutions. Since $\mathrm{R}_{2/4}$ is obviously invariant under negation, $\mathrm{Pol}\left(\mathrm{R}_{2/4}\right) \supseteq \mathrm{N}_2$, and therefore this reduction, with an application of Proposition 5.3.2, concludes the proof.

The reduction is as follows. Let φ be a $\mathsf{QCSP}_k(\mathrm{R}_{1/3})$-formula, and define $Y =_{\mathrm{def}} \mathrm{FVAR}(\varphi)$. We first construct a formula φ', which is defined by $\varphi \wedge \mathrm{R}_{1/3}(y_1, y_2, y_2)$ for new free variables $y_1, y_2 \in Y$, which do not appear in any other clause. It is obvious that any solution for φ can be extended to a solution of φ', by setting y_1 to true and y_2 to false. On the other hand, any solution to φ' must assign these values to y_1 and y_2, and the restriction of a solution for φ' to the variables in φ obviously gives a solution to the latter formula. Thus, $\#\mathrm{SOL}(\varphi') = \#\mathrm{SOL}(\varphi)$ holds. Assume that φ' is of the following form:

$$\varphi' = Q_1 X_1 \ldots \exists X_i \wedge \bigwedge_{j=1}^{m} \mathrm{R}_{1/3}(z_1^j, z_2^j, z_3^j),$$

where $z_l^j \in Y \cup \{y_1, y_2\} \cup X_1 \cup \cdots \cup X_i$ for all relevant j, l.

We now construct a $\mathrm{R}_{2/4}$-formula ψ: ψ has the free variables $Y \cup \{y_1, y_2\}$ appearing in φ', and additionally a free variable t. We define the formula ψ as follows:

$$\psi' = Q_1 X_1 \ldots \exists X_i \wedge \bigwedge_{j=1}^{m} \mathrm{R}_{2/4}(z_1^j, z_2^j, z_3^j, t) \wedge \mathrm{R}_{2/4}(y_1, y_2, y_2, t),$$

We claim that $\#\mathrm{SOL}(\psi) = 2\#\mathrm{SOL}(\varphi')$. To prove this, we show the following claim:

Claim *An assignment $I\colon \mathrm{FVAR}(\psi) \to \{0,1\}$ such that $I(t) = 1$ is a solution for ψ if and only if I, restricted to the variables in φ', is a solution for φ'.*

Since $\mathrm{R}_{2/4}$ is invariant under negation, so is the set of solutions for ψ (this follows from Proposition 5.3.1). Hence, exactly half of the solutions for ψ fulfill the condition that t is assigned the value 1, and thus the claim proves the first of the above equations. We now give the proof of the claim:

Proof. It is easy to see that if we assign the value 1 to the variable t, then the variables y_1 and y_2 must be assigned the values 1, resp 0 in any satisfying solution I for ψ, and the clause $R_{2/4}(z_1^j, z_2^j, z_3^j, t)$ is satisfied by such an I if and only if the clause $\mathrm{R}_{1/3}$ is. This immediately implies the claim. □

The above claim immediately implies that the set of solutions is doubled in the transformation from φ to ψ. Therefore, Proposition 5.3.2 shows that the transformation gives a parsimonious reduction to the problem $\#\mathsf{EQCSP}_k(\Gamma)$, proving the lemma. □

For a Boolean constraint language to be invariant under N_2 in the Boolean case means that the relations in the constraint language are invariant under every permutation of the domain $\{0,1\}$. It is natural to expect that the counting problems we just proved to be hard for the Boolean domain are not easier when we generalize this notion to arbitrary domains. This is proven in the following theorem:

Theorem 5.5.2 *Let Γ be a constraint language over a finite domain D such that $|D| \geq 2$, and every polymorphism of Γ is a permutation of D. Then for any $k \geq 2$, $\#\mathsf{EQCSP}_k(\Gamma)$ is complete for $\# \cdot \Sigma_k^p$ under parsimonious reductions, and $\#\mathsf{QCSP}_k(\Gamma)$ is complete for $\# \cdot \Sigma_k^p$ under counting reductions.*

It should be noted that hardness results for the counting hierarchy under counting reductions is not a very strong result, since the closure of $\# \cdot \mathrm{P}$ under counting reductions already gives the entire counting hierarchy ([TW92]). However, since our construction gives the result "for free," we mention it here.

Proof. We show the theorem by induction on $|D|$. For $|D| = 2$, the result follows from Lemma 5.5.1. Due to Table 1.2, this implies that hardness holds for the constraint language containing only the Boolean NAE-relation. Assume that the claim holds for $|D| = m$, and observe that the relations NAE^m and NAE^{m+1} are invariant under any permutation of their respective domains. Hence, due to Corollary 5.3.5, it suffices to show that $\#\mathsf{EQCSP}_k(\mathrm{NAE}^m)$ can be reduced to $\#\mathsf{EQCSP}_k(\mathrm{NAE}^{m+1})$ with a parsimonious reduction. We denote the m-element domain $\{0,\ldots,m-1\}$ with D_m, and the $m+1$-element domain $\{0,\ldots,m\}$ with D_{m+1}. Further note that we can express the clause $\neq^{m+1}(x,y)$ as $\mathrm{NAE}^{m+1}(x,x,y)$. Therefore, it suffices to construct a $\left\{\mathrm{NAE}^{m+1}, \neq^{m+1}\right\}$-formula in our reduction.

Let φ be a NAE^m-formula, with $\mathrm{FVAR}(\varphi) = X = \{x_1,\ldots,x_{n_x}\}$, existentially quantified variables $Y = \left\{y_1,\ldots,y_{n_y}\right\}$, and universally quantified variables $Z = \{z_1,\ldots,z_{n_z}\}$. We construct an intermediate formula φ' as follows:

$$\varphi' =_{\mathrm{def}} \varphi \wedge \bigwedge_{1 \leq i < j \leq m} (x_{n_x+i} \neq^m x_{n_x+j}),$$

where $x_{n_x+1},\ldots,x_{n_x+m}$ are new free variables, which do not appear anywhere else in the formula. It is obvious that an assignment $I\colon \mathrm{FVAR}(\varphi') \to D_m$ is a solution for φ' if and only if the following points hold:

- I, restricted to the variables appearing in φ, is a solution for φ.
- I assigns every variable $x_{n_x+1},\ldots,x_{n_x+m}$ a distinct value, which is equivalent to $\{I(x_{n_x+1}),\ldots,I(x_{n_x+m})\} = D_m$.

These points imply that $|{}^{\mathrm{SOL}(\varphi)}/_{S_D}| = |{}^{\mathrm{SOL}(\varphi')}/_{S_D}|$. We now construct the reduction formula ψ as follows:

- Copy the formula φ', and replace every relation symbol NAE^m with the symbol NAE^{m+1}
- For each variable v which is free or existentially quantified, add a new clause $(v \neq^{m+1} w)$, for a single new free variable w.

- For each universally quantified variable z_i, introduce a new variable z_i', and replace $\forall z_i$ with $\forall z_i'$. In the block of existential quantifiers following the quantification of $\forall z_i'$, add $\exists t_{i,1} \dots \exists t_{i,m-1} \exists z_i$ for new variables $t_{i,1}, \dots, t_{i,m-1}$. For these variables, add inequality clauses $(t_{i,j} \neq^{m+1} t_{i,k})$ for all relevant $j \neq k$, and clauses $(t_{i,j} \neq^{m+1} w)$ for all relevant j.

For the formula ψ constructed above, Proposition 5.3.1 implies that its set of solutions is invariant under all permutations of the domain. Additionally, any solution I for ψ needs to assign m different values to the variables $x_1, \dots, x_{n_x+m}$.

We now show that $|^{\mathrm{SOL}(\varphi')}/_{S_D}| = |^{\mathrm{SOL}(\psi)}/_{S_D}|$. Instead of talking about equivalence classes for solutions, we talk about canonical representatives of classes. For each equivalence class $\overline{I}$, let I_0 be a canonical representative chosen in a unique way which ensures that I_0 does not assign the value m to any of the variables $x_1, \dots, x_{n_x+m}$, and assigns the value m to the variable w. Since the involved relations are invariant under every permutation of the domain, such a representative always exists, for example, let I_0 be minimal in its equivalence class with respect to lexicographical ordering.

Claim *Let I_0 be a canonical representative as defined above. Then $I_0 \models \varphi'$ if and only if $I_0' \models \psi$, where I_0' is the assignment I_0 augmented with the assignment $I_0(w) = m$.*

Proof. The idea to obtain the assignment is the following: since I_0 is a solution for φ', we can, for any assignment U_m to the universal variables in φ', obtain an existential assignment E_m that satisfies the kernel of φ', which we will denote with φ_k'. Now we only need to choose an assignment U_m which "simulates" the assignment U_{m+1} closely enough. We define this assignment as follows: for a universally quantified variable z_j in φ', let

$$U_m(z_j) =_{\mathrm{def}} \begin{cases} U_{m+1}(z_j'), & \text{if } U_{m+1}(z_j') \in \{0, \dots, m-1\}, \\ m-1, & \text{otherwise.} \end{cases}$$

Then U_m only assigns values from the domain D_m, and hence is a valid universal assignment for φ'. Since I_0 is a solution for φ', there is an existential assignment E_m, which assigns every existentially quantified variable appearing in φ' a value from the domain U_m, depending only on the assignment for the free variables and the universal variables quantified earlier, such that $I_0 \cup U_m \cup E_m \models \varphi_k'$. Based on this assignment, we define an existential assignment E_{m+1} for the formula ψ as follows: for an existential variable z_j appearing in φ' as well, simply define $E_{m+1} =_{\mathrm{def}} E_m(z_j)$. For a variable z_i which is universally quantified in φ' and existentially quantified in ψ, define $E_{m+1}(z_i) = U_m(z_i)$. For each group $t_{i_1}, \dots, t_{i,m-1}$, assign these the $m-2$ distinct values from $D_{m+1} \setminus \{m, E_{m+1}(z_j)\}$.

We claim that the assignment $I_{m+1} =_{\mathrm{def}} I_0' \cup U_{m+1} \cup E_{m+1}$ satisfies ψ_k. We first consider the inequality clauses added in the step from φ' to ψ. By construction, every inequality constraint involving the variable w is satisfied: by choice of I_0', it holds that $I_0'(w) = m$, and I_0 does not assign this value to any other of the free variables. By construction of the E_{m+1}-assignment, none of the existentially quantified variables is assigned the value m. The universally quantified variables do not appear in a disequality clause with w. Hence, all of the clauses involving w are satisfied. Further, the inequalities between the

$t_{i,j}$-variables are satisfied by construction of E_{m+1} as well. Again by construction of E_{m+1}, the inequality clauses between z_i and the corresponding $t_{j,k}$-variables are satisfied.

It remains to consider the clauses which are already present in the formula φ'. Let $I_m =_{\text{def}} I_0 \cup U_m \cup E_m$, and let $I_{m+1} =_{\text{def}} I'_0 \cup U_{m+1} \cup E_{m+1}$. We claim that for any variable v appearing in φ', it holds that $I_m(v) = I_{m+1}(v)$. This implies that all NAE^{m+1}-constraints in the formula ψ' are satisfied by the assignment I_{m+1}, since $\text{NAE}^m \subseteq \text{NAE}^{m+1}$.

For free variables, this equality holds by definition. For variables v which are existentially or universally quantified in φ', this holds by construction of E_{m+1} (remember that a variable which was universally quantified in the φ' is existentially quantified in ψ). Therefore, the claim holds for all variables, and we conclude that I_{m+1} satisfies the kernel of ψ, and thus we know that I'_0 is a solution for ψ, as claimed.

For the other direction, assume that I'_0 is a canonical representative of a class of solutions for ψ. We show that I_0 is a solution for φ'. Similarly, let U_m be an assignment to the universal variables appearing in φ'. Since I'_0 is a solution for ψ, there is an assignment E_{m+1} to the existentially quantified variables of ψ, such that for each such variable z, the value $E_{m+1}(z)$ depends only on the values of the free variables and of the universal variables quantified before z in ψ, and such that I_{m+1} defined as $I'_0 \cup E_{m+1} \cup U_m$ satisfies the kernel of ψ. By definition, it holds that $I_{m+1}(w) = m$. Hence, due to the inequality constraint between all free and existentially quantified variables and w, all of these variables except w must be assigned a value different from m by I_0. In particular, it holds that $I_{m+1}(z_j) = I_{m+1}(z'_j)$ for every z_j which is universally quantified in φ'. Hence, we can define $E_m =_{\text{def}} E_{m+1}$, and the assignment $I_0 \cup U_m \cup E_m$ satisfies the kernel of φ'. □

Since for any canonical representative I_0 which is a solution for ψ, it must hold that $I(w) = m$, this proves the correctness of the reduction: due to the above, it holds that $|^{\text{SOL}(\psi)}/_{S_D}| = |^{\text{SOL}(\varphi')}/_{S_D}|$, and earlier we proved that $|^{\text{SOL}(\varphi')}/_{S_D}| = |^{\text{SOL}(\varphi)}/_{S_D}|$. Therefore, the transformation is a parsimonious reduction. Considering the problem of computing the number of solutions divided by the constant $|D|!$, observe that for the formulas ψ and φ', it holds that since there appear $|D|+1$, resp. $|D|$ many free variables which are forced to take different values, these formulas match the properties from discussion above Proposition 5.3.2. It is obvious by construction that $\#\text{SOL}(\varphi') = m! \cdot \#\text{SOL}(\varphi)$, and $\#\text{SOL}(\psi) = (m+1) \cdot \#\text{SOL}(\varphi') = (m+1)! \cdot \#\text{SOL}(\varphi)$. Now note that if φ already contains m free variables which are forced to take different values with inequality constraints, then the step from φ to φ' can be omitted, and we get $\#\text{SOL}(\psi) = (m+1) \cdot \#\text{SOL}(\varphi)$. Therefore, inductively, the reduction can be made parsimonious if we consider the problem to compute the number of solutions divided by $|D|!$. The construction also immediately gives a counting reduction to the problem $\#\mathsf{QCSP}_k(\Gamma)$, since the number of satisfying solutions of the original formula can be obtained from the number of solutions of the resulting formula by a simple division. □

The above Lemma 5.5.1 and Theorem 5.5.2 show that for any finite domain of size at least 2, if all polymorphisms are essentially unary permutations, then our counting problems are hard for the corresponding level of the polynomial hierarchy. For the Boolean case, it is not surprising that we can extend this hardness result to the case where our relations are also invariant under the constant functions, since this property only adds

very few additional solutions. This gives the corresponding hardness result for the case where all polymorphisms are unary. For the non-Boolean case, there are more additional functions than just the constants. However, we still can prove a similar result: if for a constraint language Γ, its set of polymorphisms contains only essentially unary functions or constants, then the counting problems we consider are hard for the corresponding class in the polynomial hierarchy. The following lemma proves this for both the Boolean and for arbitrary finite domains:

Lemma 5.5.3 *Let Γ be a constraint language over a finite domain D such that $|D| \geq 2$, and every polymorphism of Γ is essentially unary or a constant. Then for any $k \geq 2$, $\#\mathsf{EQCSP}_k(\Gamma)$ is complete for $\# \cdot \Sigma_k^p$ under parsimonious reductions.*

Proof. Lemma 5.5.1 (for the Boolean case) and Theorem 5.5.2 (for the non-Boolean case) show that for a relation R over the domain D such that the set of polymorphisms of R only contains unary permutations, the hardness result holds. In particular, the proofs show that the hardness results hold for the relation NAE^m, where m is the cardinality of the domain D. We now construct a relation R with the desired closure properties which still allows us to prove hardness results. We define R to be the $m+3$-ary relation

$$R =_{\text{def}} \left\{(\alpha_1, \ldots, \alpha_m, \beta_1, \beta_2, \beta_3) \mid |\{\alpha_1, \ldots, \alpha_m\}| \leq m-1 \text{ or } (\beta_1, \beta_2, \beta_3) \in \mathrm{NAE}^D\right\}.$$

We claim that R is closed under every unary function on D. Let $f\colon D \to D$, and let $(\alpha_1, \ldots, \alpha_m, \beta_1, \beta_2, \beta_3)$ be a tuple from R. If $|\{\alpha_1, \ldots, \alpha_m\}| \leq m-1$, then obviously, $|\{f(\alpha_1), \ldots, f(\alpha_m)\}| \leq m-1$ holds as well, and therefore it follows that $(f(\alpha_1), \ldots, f(\alpha_m), f(\beta_1), f(\beta_2), f(\beta_3)) \in R$. Therefore assume that $|\{\alpha_1, \ldots, \alpha_m\}| = m$, and hence $(\beta_1, \beta_2, \beta_3) \in \mathrm{NAE}^D$, without loss of generality, assume that $\beta_1 \neq \beta_2$. We now make a case distinction. If the function f is a permutation, then obviously $f(\beta_1) \neq f(\beta_2)$, and therefore $(f(\beta_1), f(\beta_2), f(\beta_3)) \in \mathrm{NAE}^D$. On the other hand, if f is not a permutation, and since the variables α_i take all possible values of the domain, there are indices $1 \leq i < j \leq m$, such that $f(\alpha_i) = f(\alpha_j)$. Therefore, it follows that $|\{f(\alpha_1), \ldots, f(\alpha_m)\})| \leq m-1$. In both cases, we conclude that $(\alpha_1, \ldots, \alpha_m, \beta_1, \beta_2, \beta_3) \in R$, and therefore, f is a polymorphism of R.

We now show that $\#\mathsf{EQCSP}_k(\mathrm{NAE}^D)$ reduces to $\#\mathsf{EQCSP}_k(R)$ with a parsimonious reduction. Let φ be a $\mathsf{QCSP}_k(\mathrm{NAE}^D)$-formula, and let φ be of the form

$$\varphi = Q_1 X_1 \ldots \forall X_{i-1} \exists X_i \bigwedge_{j=1}^{p} \mathrm{NAE}^D(x_1^j, x_2^j, x_3^j).$$

We construct a $\#\mathsf{EQCSP}_k(R)$-formula ψ as follows: let $t_1, \ldots, t_m$ be new variables, and define

$$\psi =_{\text{def}} Q_1 X_1 \ldots \forall X_{i-1} \forall t_1 \ldots \forall t_m \exists X_i \bigwedge_{j=1}^{p} R(t_1, \ldots, t_m, x_1^j, x_2^j, x_3^j).$$

We show that these formulas are equivalent, which establishes the reduction. First, assume that $I\colon \mathrm{FVAR}(\varphi) \to D$ is a solution for φ, and let U be an assignment to

the universal variables in ψ. If U does not assign the t_i distinct values, then, by definition of R, every clause is satisfied, and so is the formula ψ. Therefore assume that $|\{U(t_1), \ldots, U(t_m)\}| = m$, and let U' denote the restriction of U to the variables appearing in φ. Since I is a solution for φ, let E be an assignment for the existential variables appearing in φ, such that $I \cup E \cup U'$ satisfies the kernel of φ. It is obvious that $I \cup E \cup U$ also is a solution for ψ, since all NAE^D-clauses are satisfied.

For the other direction, let $I\colon \mathrm{FVAR}(\varphi) \to D$ be a solution for ψ, and let U be an assignment for the universal variables in φ. Define U' as the extension of U to the variables appearing in ψ in such a way that $U'(t_i) = i - 1$. In particular, this implies $|\{U'(t_1), \ldots, U'(t_m)\}| = m$. Now, let E be an assignment to the existential variables in φ such that $I \cup E \cup U'$ satisfies the kernel of ψ. Since the values assigned to the variables t_i by this assignment are fixed, any dependence of the values given by E to the existentially quantified variables in X_i can be avoided by hard-coding this dependence in the assignment function. Therefore, E also is an assignment for the existential variables appearing in φ satisfying the necessary dependence conditions. Since $|\{U'(t_1), \ldots, U'(t_m)\}| = m$, we know that the assignment $I \cup E \cup U'$ must satisfy $\mathrm{NAE}^D(x_1^j, x_2^j, x_3^j)$ for all relevant j, and therefore, this is also a satisfying assignment to the kernel of φ. Thus, I is a solution of φ. □

For Boolean constraint languages which are Schaefer, the complexity is significantly lower:

Theorem 5.5.4 *Let Γ be a Boolean constraint language which is Schaefer, and not affine. Then, for any $k \in \mathbb{N}$, the problem $\#\mathsf{QCSP}_k(\Gamma)$ is $\# \cdot \mathrm{P}$-complete under counting reductions.*

Proof. Due to Theorem 5.3.7 and Proposition 1.5.5, the problem to decide, for a given $\mathsf{QCSP}_k(\Gamma)$-formula with constants if it is true, can be solved in polynomial time. This obviously puts the corresponding counting problem in $\# \cdot \mathrm{P}$, since the witnesses consist of truth assignments, which can be encoded as strings of polynomial length. The hardness result follows directly from [CH96], since this paper gives the result that already the counting problem for Γ-formulas without quantifiers is hard for $\# \cdot \mathrm{P}$ under counting reductions, and this problem trivially reduces to $\#\mathsf{QCSP}_k(\Gamma)$. □

For the Boolean case, the results just presented and Corollary 5.4.2 give a complete picture. A graphical representation of the result can be found in Figure 5.1.

Theorem 5.5.5 *Let Γ be a Boolean constraint language, and let k be a natural number. Then the following holds:*

- *If Γ is affine, then $\#\mathsf{EQCSP}_k(\Gamma)$ can be solved in polynomial time. Further, if Γ is also complementive, then $\#\mathsf{QCSP}_k(\Gamma)$ can be solved in polynomial time.*
- *Otherwise, if Γ is Schaefer, then $\#\mathsf{QCSP}_k(\Gamma)$ is complete for $\# \cdot \mathrm{P}$ under counting reductions.*
- *Otherwise, if Γ is complementive, then $\#\mathsf{EQCSP}_k(\Gamma)$ is complete for $\#\cdot\mathrm{QPH_k}$ under parsimonious reductions, and $\#\mathsf{QCSP}_k(\Gamma)$ is complete for $\# \cdot \mathrm{QPH_k}$ under counting reductions.*

- *Otherwise,* $\#\mathsf{QCSP}_k(\Gamma)$ *is complete for* $\# \cdot \mathrm{QPH_k}$ *under parsimonious reductions.*

5.6 Decision Problems

We now consider the decision problems defined for our quantified formulas. We will see that many of the results easily follow from the results on the counting problems just considered.

5.6.1 Quantified Formula Evaluation

The following observation shows how our results for counting can be applied to our decision problems:

Proposition 5.6.1 *Let* Γ *be a constraint language over the finite domain* D*, such that* $\#\mathsf{QCSP}_k(\Gamma)$ *or* $\#\mathsf{EQCSP}_k(\Gamma)$ *are hard for* $\# \cdot \mathrm{QPH_k}$ *under parsimonious reductions which can be computed in logarithmic space. Then* $\mathsf{QCSP}_k(\Gamma)$ *is hard for* $\mathrm{QPH_k}$ *under* $\leq_m^{\log}$*-reductions.*

Proof. From prerequisites, it follows that $\#\mathsf{QCSP}_k(\Gamma_{\mathsf{3SAT}})$ reduces to $\#\mathsf{QCSP}_k(\Gamma)$, or to $\#\mathsf{EQCSP}_k(\Gamma)$ under a parsimonious reduction which can be computed in logarithmic space. It is obvious that a reduction which preserves the number of solutions or the number of classes of solutions also preserves truth of a given formula. Hence, this reduction is a $\leq_m^{\log}$-reduction. The claim then follows from Proposition 5.3.6. □

For the Boolean domain, this suffices to prove a complete classification of the problem. A representation of the complexities for the classes in Post's lattice can be found in Figure 5.2.

Theorem 5.6.2 *Let* Γ *be a Boolean constraint language, let* $k \geq 2$*. Then the following holds:*

- *If* Γ *is Schaefer, then* $\mathsf{QCSP}_k(\Gamma) \in \mathrm{P}$.
- *Otherwise,* $\mathsf{QCSP}_k(\Gamma)$ *is complete for* $\mathrm{QPH_k}$ *under* $\leq_m^{\log}$*-reductions.*

Proof. The polynomial time result follows directly from Theorem 5.3.7. The $\mathrm{QPH_k}$ upper bound follows from Proposition 5.3.6. Note that if Γ is not Schaefer, then $\mathrm{Pol}\,(\Gamma) \subseteq \mathrm{N}$. Therefore, Lemma 5.5.3 implies that $\#\mathsf{EQCSP}_k(\Gamma)$ or $\#\mathsf{QCSP}_k(\Gamma)$ is complete for $\# \cdot \mathrm{QPH_k}$ under parsimonious reductions, and thus the above Proposition 5.6.1 states that $\mathsf{QCSP}_k(\Gamma)$ is hard for $\mathrm{QPH_k}$ under $\leq_m^{\log}$-reductions. □

For non-Boolean domains, Proposition 5.6.1 and Lemma 5.5.3 yield the following hardness result:

Corollary 5.6.3 *Let* Γ *be a constraint language over a finite domain* D *such that* $|D| \geq 2$*, and every polymorphism of* Γ *is essentially unary or a constant. Then for any* $k \geq 2$*,* $\mathsf{QCSP}_k(\Gamma)$ *is complete for* $\mathrm{QPH_k}$ *under* $\leq_m^{\log}$*-reductions.*

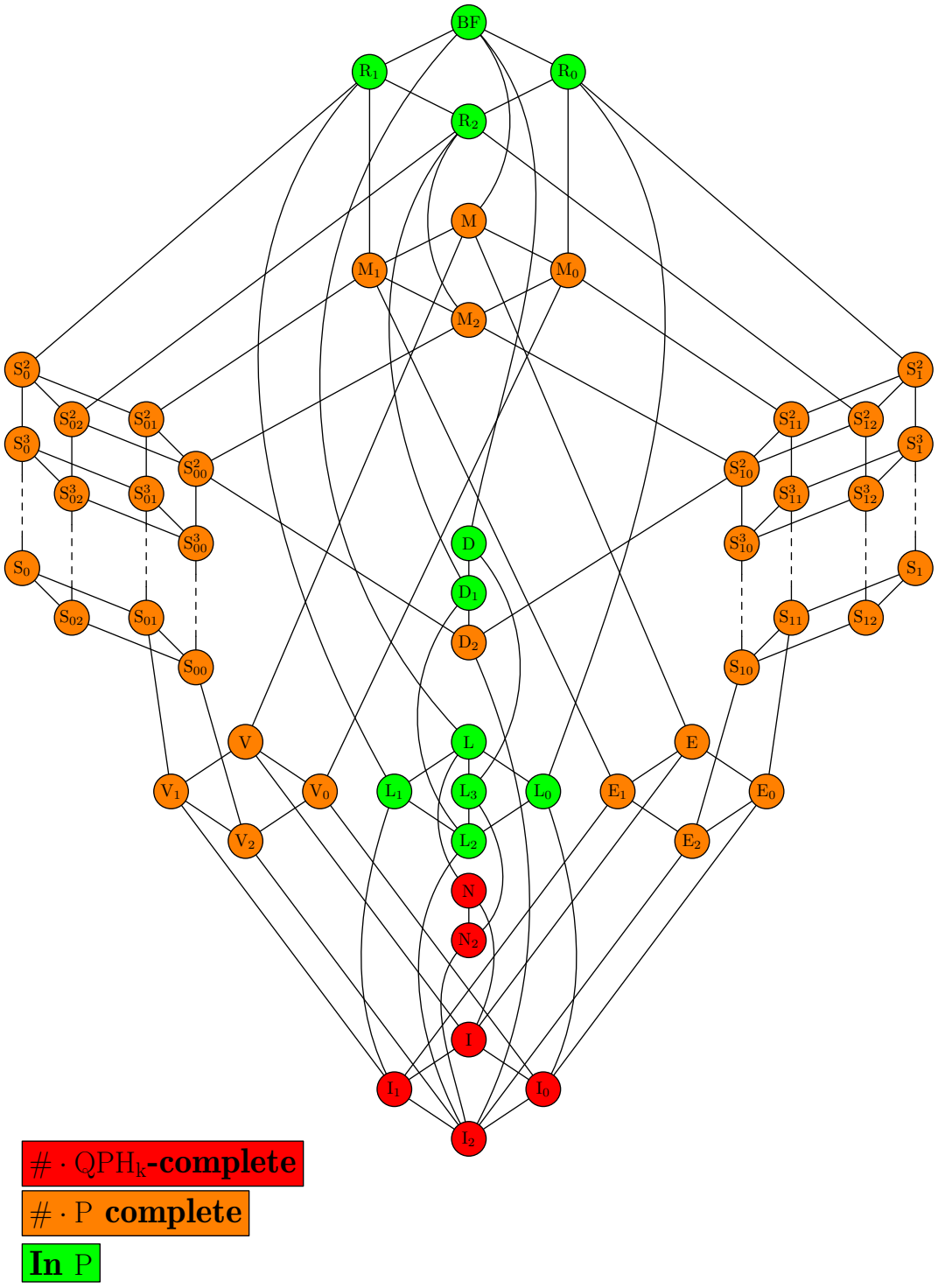

Figure 5.1: The complexity of $\#\mathsf{QCSP}_k(\Gamma)$ or $\#\mathsf{EQCSP}_k(\Gamma)$

5.6.2 Quantified Model Checking

The model checking problem is of course closely related to the formula evaluation problem. For Boolean constraint languages, our results for the latter immediately give a complete classification of the model checking problem as well, and in fact, the same complexity cases arise here. The following theorem states the result, a graphical representation of both results can be found in Figure 5.2

Theorem 5.6.4 *Let Γ be a Boolean constraint language, and let k be a natural number. Then the following holds:*

- *If Γ is Schaefer, then $\mathsf{QMCK}_k(\Gamma) \in \mathrm{P}$.*
- *Otherwise, $\mathsf{QMCK}_k(\Gamma)$ is complete for $\mathrm{QPH_k}$ under $\leq_m^{\log}$-reductions.*

Proof. We first consider the polynomial time cases. Due to Proposition 1.5.5, the constraint language $\Gamma \cup \{x, \overline{x}\}$ is Schaefer as well. Now, let φ be a QCSP_k-formula, and let $I \colon \mathrm{FVAR}(\varphi) \to \{0,1\}$ be an assignment to the free variables. We construct a formula ψ as follows: For each variable $z \in \mathrm{FVAR}(\varphi)$, add $\exists z$ in an arbitrary existential part of quantifier block into φ. Further, if $I(z) = 0$, add a clause $\overline{z}$, and if $I(z) = 1$, add a clause z. It is obvious that I is a solution for φ if and only if the formula ψ constructed in this way evaluates to true. Hence, this problem can be decided in polynomial time due to Theorem 5.3.7.

Now assume that Γ is not Schaefer. Since $\mathsf{QCSP}_k(\Gamma)$ trivially reduces to $\mathsf{QMCK}_k(\Gamma)$, the hardness result is immediate. The upper bound follows from Proposition 5.3.6. □

For non-Boolean domains, Corollary 5.6.3 and the obvious reduction from $\mathsf{QCSP}_k(\Gamma)$ to $\mathsf{QMCK}_k(\Gamma)$ give the following result:

Corollary 5.6.5 *Let Γ be a constraint language over a finite domain D, such that $|D| \geq 2$, and every polymorphism of Γ is essentially unary or a constant. Then for any $k \geq 2$, $\mathsf{QMCK}_k(\Gamma)$ is complete for $\mathrm{QPH_k}$ under $\leq_m^{\log}$-reductions.*

5.6.3 The Equivalence Problem

For the study of the complexity of the equivalence problem, we can again take advantage of the duality in Post's lattice. Remember that in Chapter 4, we defined for a Boolean relation R, the relation $\overline{R}$ to contain the set of negations of tuples from R, and for a Boolean constraint language Γ, the language $\overline{\Gamma}$ contains the relation $\overline{R}$ for every relation R in Γ.

Proposition 5.6.6 *Let Γ be a Boolean constraint language, and let k be a natural number. Then $\mathsf{QEQUIV}_k(\Gamma) \equiv_m^{\mathrm{AC}^0} \mathsf{QEQUIV}_k(\overline{\Gamma})$.*

Proof. The proof is very similar to the proof for Corollary 4.2.9. Let φ and ψ be $\mathsf{QCSP}_k(\Gamma)$-formulas. We compute the formulas φ' and ψ', obtained from the original formulas by replacing each application of a relation R with the application of the relation $\overline{R}$ to the same variables. We claim that φ is equivalent to ψ if and only if φ' is equivalent

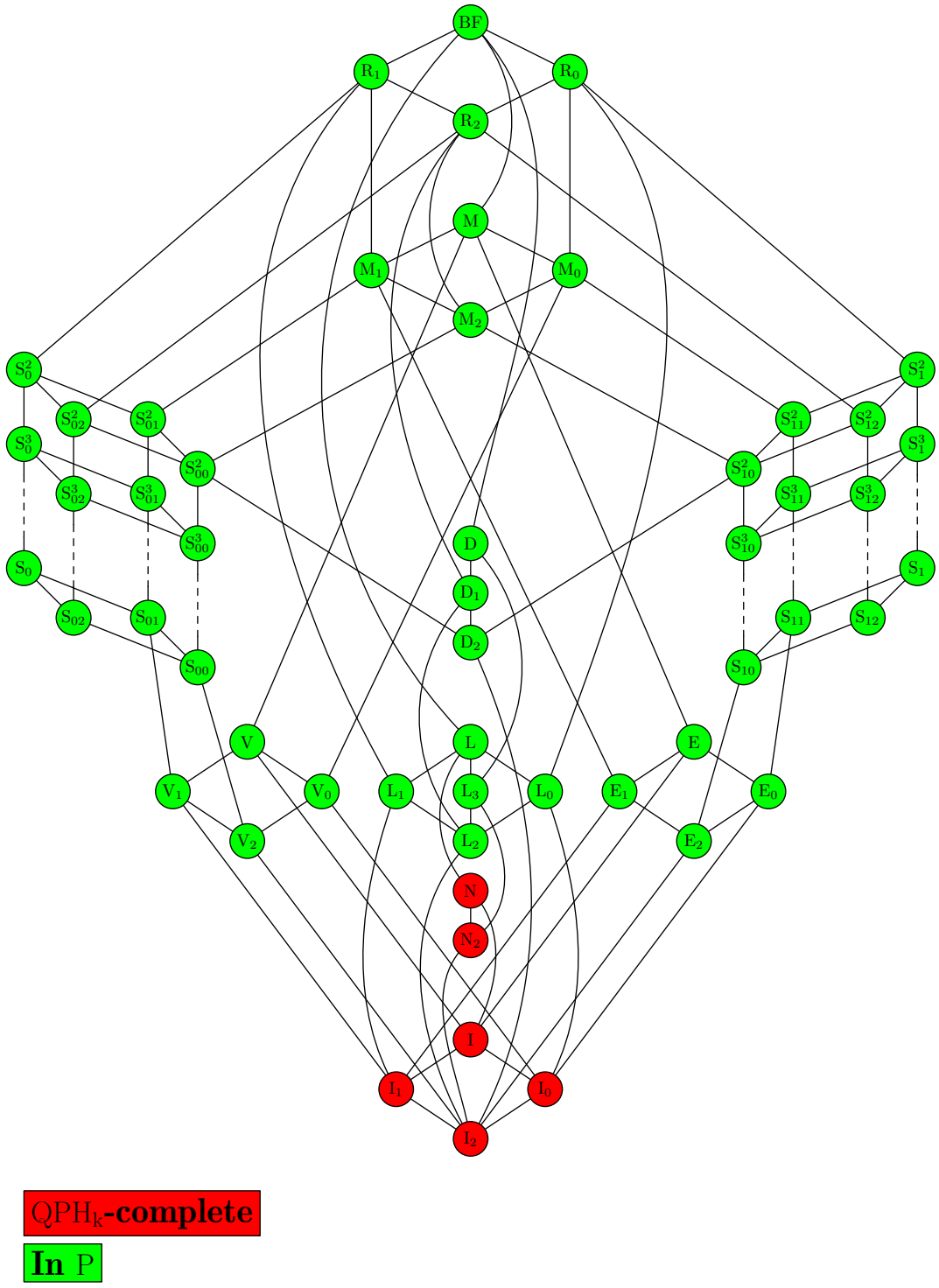

Figure 5.2: The complexity of $\mathsf{QCSP}_k(\Gamma)$ and of $\mathsf{QMCK}_k(\Gamma)$

to ψ'. Due to symmetry, since $\overline{\overline{R}} = R$, it suffices to show one of these implications. We claim that for any assignment $I\colon \mathrm{FVAR}(\varphi) \to \{0,1\}$, it holds that $I \models \varphi$ if and only if $\overline{I} \models \varphi'$, where $\overline{I}(x) =_{\mathrm{def}} \overline{I(x)}$ for all free variables x of φ. This clearly completes the proof.

We show the claim by induction over the quantifier structure of φ. Again, due to symmetry, it suffices to show one implication. If no quantifiers appear, then the claim holds by definition of $\overline{R}$. Now let φ be a formula for which the claim holds, and let z be a variable appearing in φ. We show that it also holds for $\exists z\varphi$ and $\forall z\varphi$. Let I be a solution for $\exists z\varphi$. Then there is some $\alpha \in \{0,1\}$, such that the assignment I, augmented with the assignment $I(z) = \alpha$, is a satisfying model for φ. Since the claim holds for φ, this implies that the assignment $\overline{I}$ with an additional assignment $\overline{I}(z) = \overline{\alpha}$ satisfies φ', and hence, $\overline{I}$ satisfies $\exists z\varphi'$. Finally, let I be a solution for $\forall z\varphi$. This is equivalent to the following: both I_0 and I_1, where I_α is the assignment I augmented with $I(z) = \alpha$, are satisfying solutions for φ. Due to induction, we know that both assignments $\overline{I_\alpha}$ are solutions for φ'. This means that $\overline{I}$ is a solution for $\forall z\varphi'$, as claimed. □

We now give an upper complexity bound for the Schaefer cases, which basically holds for the same reason as the general coNP-bound for equivalence in the unquantified formula case.

Lemma 5.6.7 *Let Γ be a Boolean constraint language which is Schaefer, and let $k \in \mathbb{N}$. Then $\mathsf{QEQUIV}_k(\Gamma) \in$ coNP.*

Proof. Due to Theorem 5.6.4, we know that the model checking problem for $\mathsf{QCSP}_k(\Gamma)$-formulas can be solved in polynomial time. Now given two Γ-formulas φ_1 and φ_2, we can guess an assignment which is a model for φ_1 but not for φ_2 (or vice versa) in NP. Hence, the "non-equivalence problem" is in NP, and therefore the equivalence problem is in coNP. □

For Horn- and anti-Horn formulas, we also get the corresponding hardness result:

Theorem 5.6.8 *Let Γ be a Boolean constraint language such that $\mathrm{Pol}(\Gamma) \in \{\mathrm{E}_2, \mathrm{V}_2\}$, and let $k \in \mathbb{N}$. Then $\mathsf{QEQUIV}_k(\Gamma)$ is coNP-complete under $\leq_m^{\log}$-reductions.*

Proof. We show the theorem for the case $\mathrm{Pol}(\Gamma) = \mathrm{E}_2$. The case $\mathrm{Pol}(\Gamma) = \mathrm{V}_2$ then follows from Proposition 5.6.6 and Lemma 4.2.8, since $\mathrm{dual}(\mathrm{E}_2) = \mathrm{V}_2$. The upper bound follows directly from Lemma 5.6.7. Now observe that relations invariant under E_2 are exactly those which can be expressed by Horn formulas. Hence, the result follows from Theorem 7.5.4 in [BL99], if we can show that any Horn-formula in the usual sense with existentially quantified variables can be re-written as one where only clauses of bounded arity occur. It then follows that there is a finite constraint language Γ, which can be used to express these clauses with bounded arity, which has the required set of polymorphisms, and for which the equivalence problem is coNP-hard. From Proposition 5.3.3, it then follows that hardness holds for any constraint language Γ such that $\mathrm{Pol}(\Gamma) = \mathrm{E}_2$, and Γ can express equality. Due to Lemma 4.4.1, and since $\mathrm{E}_2 \subseteq \mathrm{M}$, we know that every constraint language with this set of polymorphisms can express equality, and hence we

have shown that the hardness result holds for arbitrary constraint languages fulfilling the requirement.

Let φ be a Horn-formula without restriction on the arity of the clauses, i.e., a formula in conjunctive normal form where each clause is a disjunction of literals, where at most one literal per clause is positive. We show how to rewrite individual clauses with bounded arity Horn clauses, this can obviously be used to re-write the entire formula.

Assume that a clause without a positive literal occurs, i.e., a clause $C = \overline{x_1} \wedge \cdots \wedge \overline{x_n}$. For such a clause, we introduce a new existentially quantified variable t_C, and change the clause into $C' = \overline{x_1} \wedge \cdots \wedge \overline{x_n} \wedge \overline{t_C}$ and add a clause $\overline{t_C}$ to the entire formula. It is obvious that this is an equivalent transformation, and therefore we can assume that each of the clauses contains exactly one positive literal, or is a single negative literal. Now such a clause C can be written as

$$C = x_1 \wedge \cdots \wedge x_n \rightarrow y.$$

If $n \leq 2$, then we leave this clause unchanged. Otherwise, we can split up C into two clauses of smaller arity, with the help of a new existentially quantified variable t_C, and is equivalent to the following (where l is some number such that $1 < l < n$, for example choose $l = \lfloor \frac{n}{2} \rfloor$):

$$C' = \exists t_C (x_1 \wedge \cdots \wedge x_l \rightarrow t_C) \wedge (x_{l+1} \wedge \cdots \wedge x_n \wedge t_C \rightarrow y).$$

By repeatedly applying this splitting procedure, we can rewrite each clause with bounded arity. The procedure needs to be applied at most once for every variable appearing in the original formula, and obviously each step can be performed in polynomial time. Hence, this gives a polynomial-time transformation. We now prove that the clauses C and C' are in fact equivalent. Assume that there is an assignment $I\colon \{x_1, \ldots, x_n, y\} \rightarrow \{0, 1\}$ which satisfies C. We have several cases to consider:

If $I(y) = 1$, then the clause C' can obviously be satisfied by choosing the value 1 for the existentially quantified variable t_C. If $I(x_i) = 0$ for some $i \leq l$. Then the clause C' can be satisfied by choosing 0 for the variable t_C. Finally, if $I(x_i) = 0$ for some $i \geq l$, then C' can be satisfied by choosing 1 for t_C.

For the other direction, assume that an assignment I satisfies the clause C', and assume that it does not satisfy C. In this case, it follows that $I(x_i) = 1$ for each relevant i, and $I(y) = 0$. In order to satisfy the first clause of C', the variable t_C then needs to take the value 1. But this assignment does not satisfy the second clause, hence we have a contradiction.

The above shows that each Horn formula can be rewritten into one where only literals, and Horn clauses of bounded arity appear, by only adding existentially quantified variables to the formula, which are quantified at the very end of the quantifier block. The transformation above converts the formulas into equivalent ones where each clause is either a literal, or an implication of the form $x_1 \rightarrow y$ or $x_1 \wedge x_2 \rightarrow y$. Hence the hardness result holds for the constraint language Γ containing the relations representing literals, and the relations represented by the clauses just mentioned. It is easy to see that these are closed under conjunction, and hence this concludes the proof. □

Similarly as for the counting problems studied earlier, we can show here as well that the constant polymorphisms do not "help us to solve the equivalence problem:"

Theorem 5.6.9 *Let* Γ *be a constraint language such that* $\mathrm{Pol}(\Gamma) \subseteq \mathrm{E}$, *or* $\mathrm{Pol}(\Gamma) \subseteq \mathrm{V}$. *Then* $\mathsf{QEQUIV}_k(\Gamma)$ *is* coNP*-hard for any* $k \in \mathbb{N}$.

Proof. We prove the Theorem for the case $\mathrm{Pol}(\Gamma) \subseteq \mathrm{E}$. Again, the dual case $\mathrm{Pol}(\Gamma) \subseteq \mathrm{V}$ then follows from Proposition 5.6.6 and Lemma 4.2.8. Let Γ be a constraint language such that $\mathrm{Pol}(\Gamma) = \mathrm{E}_2$. We show that $\mathsf{QEQUIV}_1(\Gamma) \leq_m^p \mathsf{QEQUIV}_k(\Gamma')$ for some constraint language Γ' for which $\mathrm{Pol}(\Gamma') \supseteq \mathrm{E}$ holds. The result then follows from Theorem 5.6.8 and Proposition 5.3.3, since due to Lemma 4.4.1, the involved constraint languages here can express equality.

For an n-ary relation $R \in \Gamma$, we construct a $n+2$-ary relation R which can be used to express R for our purposes, and which has both constant polymorphisms. We define

$$R' =_{\mathrm{def}} \{(\alpha_1, \ldots, \alpha_{n+2}) \mid (\alpha_1, \ldots, \alpha_n) \in R \text{ or } \alpha_1 = \cdots = \alpha_{n+2}\}.$$

It is obvious that R' is closed under both constant polymorphisms. We claim that R' is also closed under conjunction, and hence under every function from E. Therefore, let $(\alpha_1, \ldots, \alpha_{n+2}), (\beta_1, \ldots, \beta_{n+2})$ be tuples from R'. There are several cases to consider. If one of these tuples is the constant 0-tuple, then the result of the conjunction again is this constant tuple, and therefore is an element of the relation R'. If one of these is the constant 1-tuple, then the result gives the second tuple, and this is an element of R' as well. Therefore, assume that both tuples are non-constant. Due to the definition of R', this implies that $(\alpha_1, \ldots, \alpha_n)$ and $(\beta_1, \ldots, \beta_n)$ are elements of R, and since R is closed under conjunction, it follows that $(\alpha_1 \wedge \beta_1, \ldots, \alpha_n \wedge \beta_n)$ is an element of R'. Hence, the conjunction of the $n+2$-tuples is an element of R'.

Now, let Γ' be defined as $\{R' \mid R \in \Gamma\} \cup \{\rightarrow\}$. As mentioned above, it suffices to prove hardness for this special choice of Γ. The relation $\rightarrow$ is invariant under conjunction, since conjunction is in the clone M, and due to Table 1.2, $\rightarrow$ is invariant under M. Obviously, $\rightarrow$ contains both constants. Now let φ_1 and φ_2 be $\mathsf{QCSP}_1(\Gamma)$-formulas with the same set of free variables, i.e., let

$$\varphi_1 = \exists x_1, \ldots, x_n \bigwedge_{i=1}^{l_1} R_i^1(u_1^i, \ldots, u_{k_i}^i),$$

$$\varphi_2 = \exists y_1, \ldots, y_m \bigwedge_{i=1}^{l_2} R_i^2(v_1^i, \ldots, v_{k_i}^i),$$

where the occurring R_i^1 and R_i^2 are k_i-ary relations from Γ, the occurring variables u_t^i are either from $\{x_1, \ldots, x_n\}$, or from the set $(z_1, \ldots, z_k)$ of free variables of φ_1, and similarly the variables v_t^i are from $\{y_1, \ldots, y_m, z_1, \ldots, z_k\}$.

We define formulas ψ_1 and ψ_2 as follows:

$$\psi_1 = \exists x_1, \ldots, x_n \bigwedge_{i=1}^{l_1} R_i'^1(u_1^i, \ldots, u_{k_i}^i, t_1, t_2),$$

$$\psi_2 = \exists y_1, \ldots, y_m \bigwedge_{i=1}^{l_2} R_i'^2(v_1^i, \ldots, v_{k_i}^i, t_1, t_2),$$

where t_1 and t_2 are additional free variables. Further, we add, for each free variable z, the clause $z \to t_1$ and the clause $t_2 \to z$ to both formulas. Then, by definition, ψ_1 and ψ_2 are $\mathsf{QCSP}_k(\Gamma')$-formulas. The relationship between these formulas is as follows:

Claim *For* $i \in \{1, 2\}$, an assignment I satisfies ψ_i if and only if I is constant, or I restricted to the variables appearing in φ_i satisfies φ_i, and $I(t_1) = 1$, $I(t_2) = 0$.

Proof. First assume that I satisfies one of the two conditions. If I is constant, then the existential variables in ψ_i can be assigned the same value as I assigns to the free variables, and therefore, ψ_i is satisfied. If $I(t_1) = 1$ and $I(t_2) = 0$, then obviously, the implication clauses added in the transformation from φ_i to ψ_i are satisfied. For the clauses involving the relations from Γ, note that the "or $\alpha_1 = \cdots = \alpha_{n+2}$" case never occurs, and therefore it holds that I is a satisfying assignment to ψ_i if and only if I restricted to the variables occurring in φ_i satisfies the latter formula.

Now for the other direction, assume that I is a satisfying assignment for ψ_i. If $I(t_1) \neq I(t_2)$, then as above, it follows that the restriction of I is a solution for φ_i. Therefore, assume that $I(a) = I(b)$. If I is constant, then the claim holds. Now assume that I is not constant. Since $I(t_1) = I(t_2)$, this implies that there is some free variable z such that $I(z) \neq I(t_1)$. We make a case distinction:

If $I(z) = 0$, then, since the clause $t_2 \to z$ appears in ψ_i, and I satisfies the formula, we know that $I(t_2) = 0$. This is a contradiction to the assumption $I(z) \neq I(t_2)$. Similarly, if $I(z) = 1$, then due to the clause $z \to t_1$, we know that $I(t_1) = 1$ must hold, again a contradiction. Therefore, we conclude that I is constant. □

We now show that φ_1 and φ_2 are equivalent if and only if ψ_1 and ψ_2 are. Since the formulas can obviously be computed in polynomial time, this concludes the proof.

First, assume that φ_1 and φ_2 are equivalent, and let I be some assignment to the free variables of ψ_1 such that $I \models \psi_1$. We show that I is also a solution for ψ_2, the equivalence of the formulas then follows due to symmetry. Due to the claim above, we have two cases to consider: if I is a constant assignment, then, by the claim above, we know that I is a solution for ψ_2. Otherwise, due to the claim above, we know that $I(t_1) = 1$, $I(t_2) = 0$, and I restricted to the variables appearing in φ_1 is a solution to the latter formula. Since φ_1 and φ_2 are equivalent, this implies that the restriction of I is a solution for φ_2 as well, and due to the claim above, it follows that I is a solution of ψ_2.

Now assume that ψ_1 and ψ_2 are equivalent, and let I be a solution for φ_1. Again, due to symmetry, it suffices to show that I is a solution for φ_2 as well. Due to the claim above, we know that the solution I' obtained from I by augmenting it with the assignments $I'(t_1) = 1$, and $I'(t_2) = 0$, is a solution for ψ_1. Since ψ_1 and ψ_2 are equivalent, this implies that I' is also a solution for ψ_2. Therefore, due to the claim above, it follows that I is a solution for φ_1, as claimed. □

For languages restricted even more than Schaefer, we even can solve the problem in polynomial time:

Theorem 5.6.10 *Let Γ be a Boolean constraint language such that $\mathrm{S}_{10} \subseteq \mathrm{Pol}(\Gamma)$, $\mathrm{S}_{00} \subseteq \mathrm{Pol}(\Gamma)$, or $\mathrm{D}_2 \subseteq \mathrm{Pol}(\Gamma)$, and let k be a natural number. Then $\mathsf{QEQUIV}_k(\Gamma)$ can be solved in polynomial time.*

Proof. We want to apply quantified resolution to convert the formulas into equivalent ones over the same constraint languages which are quantifier-free. The procedure relies on Theorem 7.4.6 in [BL99], which shows the correctness of the transformation we describe. In order to apply this theorem to our case, we need to prove two facts: first, we show that the transformation produces a formula of the same constraint language. Second, we show that the procedure can be computed in polynomial time. Since for quantifier-free formulas over these constraint languages equivalence can be checked in polynomial time due to [BHRV02], this finishes the proof.

We first describe how quantified resolution works. We follow the presentation in Section 7.3 of [BL99]. Starting with a quantified formula φ whose kernel is in CNF, new clauses are generated with the following rules:

1. In each clause of the kernel of φ, remove all literals over universally quantified variables which are $<_\varphi$-maximal in the variables occurring in the clause.
2. For two clauses α_1 and α_2, if y is a variable which is free or existentially quantified, and y appears positively in α_1, and negatively in α_2, obtain a new clause α, which is the disjunction of all literals occurring in α_1 except the positive occurrences of y, and all literals occurring in α_2 except the negative occurrence of y.

The formula ψ we construct contains of all clauses which are contained in the original formula φ, or which can be generated with the above procedure, and which only contain free variables. Therefore, by definition, ψ is an unquantified formula. Due to Theorem Theorem 7.4.6 in [BL99], ψ is equivalent to φ. Therefore it remains to prove the above points.

If $\mathrm{S}_{10} \subseteq \mathrm{Pol}(\Gamma)$ holds, then, by Lemma 1.5.6, we know that $\mathrm{S}_{10}^m \subseteq \mathrm{Pol}(\Gamma)$ holds for some natural number m. Due to Proposition 5.3.3, and the results from Table 1.2, we can therefore assume that Γ is the set $\{\mathrm{NAND}^m, x, \overline{x}, x \to y, x = y\}$. Note that $x = y$ can be written as $(x \to y) \wedge (y \to x)$, and $x \to y$ can be written as $\overline{x} \vee y$. We now show that applying the above resolution rule to Γ-clauses again gives a Γ-clause. When applying the rule, at least one of the clauses must contain a positive literal. This means that not both of them can be NAND^m-clauses, since $\mathrm{NAND}^m(x_1, \ldots, x_m)$ is the same as $\overline{x_1} \vee \cdots \vee \overline{x_m}$. It is obvious that removing $\forall$-quantified variables from Γ-clauses again gives a Γ-clause (in the case of NAND^m, simply repeat one of the remaining variables to match the arity, and we can obviously disregard empty clauses). Therefore, we have the following cases to consider:

- Let α_1 be a NAND^m-clause in which y appears, and let α_2 be the positive literal y. Assume that $\alpha_1 = (\overline{x_1} \vee \cdots \vee \overline{x_{m-1}} \vee \overline{y})$, and $\alpha_2 = y$. Then the clause generated is simply $(\overline{x_1} \vee \cdots \vee \overline{x_{m-1}})$, this can be written as $(\overline{x_1} \vee \cdots \vee \overline{x_{m-1}} \vee \overline{x_{m-1}})$ to give again a clause of arity m.
- Let α_1 be a NAND^m-clause in which y appears, and let α_2 be the clause $y \vee \overline{x}$ for some variable x. Then, if $\alpha_1 = (\overline{x_1} \vee \cdots \vee \overline{x_{m-1}} \vee \overline{y})$, the resulting clause from

quantified resolution is the clause $(\overline{x_1} \vee \cdots \vee \overline{x_{m-1}} \vee \overline{x})$, and therefore, again a Γ-clause.

- Let α_1 and α_2 both be implications. If y appears negatively in α_1 and positively in α_2, then α_1 is of the form $\overline{y} \vee x$, and α_2 is of the form $y \vee \overline{z}$. Hence, the generated clause is $x \vee \overline{z}$, and can again be written as an implication.
- Let α_1 be an implication, and let α_2 be a literal, such that y appears negatively in α_1, and positively in α_2, or vice versa. It is obvious that the resulting clause is a literal.
- If α_1 and α_2 are both literals, then the only case where the rule can be applied is if they are contradictory. In this case, we simply generate an unsatisfiable formula, i.e., let $\psi = y \wedge \overline{y}$, and finish the algorithm.

The case analysis above shows that the resolution procedure again produces a Γ-formula. It remains to show that we can, in polynomial time, produce all of the clauses involving only free variables which can be obtained in this way. However, this is obvious, since the generated clauses are of constant length, bounded by m. Therefore, there are only polynomially many possible clauses, and since each resolution step clearly can be performed in polynomial time, this means that the set of clauses which can be obtained by resolution can be polynomially computed.

The case $\mathrm{S}_{00} \subseteq \mathrm{Pol}\,(\Gamma)$ is also solvable in polynomial time, since $\mathrm{dual}\,(\mathrm{S}_{00}) = \mathrm{S}_{10}$. Hence the result follows from Proposition 5.6.6 and Lemma 4.2.8. It remains to consider the case $\mathrm{Pol}\,(\Gamma) \supseteq \mathrm{D}_2$.

It can easily be verified that every Boolean relation which is at most binary is invariant under D_2. Hence, we can assume, without loss of generality, that Γ consists of all these relations. Note that each of these can be written as a conjunction of clauses which are disjunctions of at most two literals. Now for these types of clauses, it is obvious that an application of the quantified resolution procedure again gives a clause from this language. Therefore, clauses generated by quantified resolution are again Γ-clauses. Since the arity of the clauses is restricted by 2, it again follows that there are only polynomially (in fact, quadratic) many clauses which can be generated, and by the same reasoning as above, this allows all clauses to be generated in polynomial time, thus finishing the proof. □

Note that the argument from the above proof fails for Horn clauses: if we consider two Horn clauses $(\overline{x_1} \vee \overline{x_2} \vee y)$ and $(\overline{y} \vee \overline{x_3} \vee x_4)$, then the resulting clause is $(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$. Inductively, we can generate clauses with as many literals as there are variables in the formula, and in particular, we cannot restrict the length of the occurring clauses with a constant. This is the reason why the problem is coNP-complete for Horn formulas, and solvable in polynomial time for the restricted classes covered by Theorem 5.6.10.

For the non-Schaefer cases, we can establish lower bounds which match the general upper bound.

Theorem 5.6.11 *Let Γ be a constraint language over a domain D, $|D| \geq 2$, such that all polymorphisms of Γ are constant or essentially unary. Then, for any $k \in \mathbb{N}$, the problem $\mathsf{QEQUIV}_k(\Gamma)$ is complete for Π^p_{k+1}.*

Proof. The upper bound follows from Proposition 5.3.6. If suffices to show that if k is even, then $\overline{\mathsf{QCSP}_{k+1}(\Gamma)} \leq_m^p \mathsf{QEQUIV}_k(\Gamma)$ and if k is odd, then $\mathsf{QCSP}_{k+1}(\Gamma) \leq_m^p \mathsf{QEQUIV}_k(\Gamma)$: If k is even, then by definition, $\mathrm{QPH_{k+1}}$ is the class Σ_{k+1}^p. Due to Corollary 5.6.3, we know that the problem $\mathsf{QCSP}_{k+1}(\Gamma)$ is complete for Σ_{k+1}^p. Therefore, the problem $\overline{\mathsf{QCSP}_{k+1}(\Gamma)}$ is complete for Π_{k+1}^p, and hence the result follows. Similarly, if k is odd, then $\mathrm{QPH_{k+1}}$ is the class Π_{k+1}^p, and thus QCSP_{k+1} is complete for Π_{k+1}^p, and the result follows in the same way. Hence, it remains to prove the above-mentioned reduction.

If k is even, a QCSP_{k+1}-formula is of the form $\varphi = \exists X_1 \forall X_2 \ldots \exists X_{k+1} \psi$ for a Γ-formula ψ. It is obvious that this formula is false, and hence in $\overline{\mathsf{QCSP}_{k+1}(\Gamma)}$, if and only if $\forall X_2 \ldots \exists X_{k+1} \psi$ is equivalent to a false Γ-formula, which exists due to the above. Now, assume that k is odd. Then a QCSP_{k+1}-formula starts with a $\forall$-quantifier. Hence, let $\varphi = \forall X_1 \ldots \exists X_2 \ldots \exists X_{k+1} \psi$, for some Γ-formula ψ. Then this formula is true if and only if $\exists X_2 \ldots \exists X_{k+1} \psi$ is equivalent to a Γ-tautology, which easily can be constructed. Therefore, in both cases the reduction is complete. □

For the Boolean case, our results give a complete classification of the problem. A graphical representation can be found in Figure 5.3.

Theorem 5.6.12 *Let Γ be a Boolean constraint language, and let k be a natural number. Then the following holds:*

- *If Γ is affine, $\mathrm{S_{01}} \subseteq \mathrm{Pol}(\Gamma)$, $\mathrm{S_{00}} \subseteq \mathrm{Pol}(\Gamma)$, or $\mathrm{D_2} \subseteq \mathrm{Pol}(\Gamma)$, then $\mathsf{QEQUIV}_k(\Gamma)$ can be solved in polynomial time.*
- *Otherwise, if Γ is Schaefer, then $\mathsf{QEQUIV}_k(\Gamma)$ is* coNP*-complete.*
- *Otherwise, $\mathsf{QEQUIV}_k(\Gamma)$ is complete for Π_{k+1}^p.*

5.7 Conclusion

For quantified constraint formulas, we have studied the natural problems of truth evaluation, model checking, equivalence, and counting of satisfying assignments. For the Boolean case, we showed that the characteristic property of constraint problems, which in many cases leads to dichotomy results, again holds for these problems. In the case of counting problems, the lower bounds could not be proven using the usual parsimonious reduction in the case where the constraint language is complementive. However, with showing that one bit of the resulting natural number is fixed, and the problem to compute the remaining bits is complete for the corresponding counting class, we feel that the complexity of this problem is, for all intents and purposes, equivalent to the usual complete problems for these classes under parsimonious reductions. For the Schaefer cases, hardness could only be proven using counting reductions. While these reductions are not sufficiently strict to compare the different levels of the counting hierarchy, they still allow to distinguish the classes FP and $\# \cdot \mathrm{P}$. Therefore, we consider the Boolean case to be solved in a satisfying way.

The picture is less clear for non-Boolean domains. Unlike in the Boolean case, the two possible ways to deal with permutative constraint languages that we discussed do not

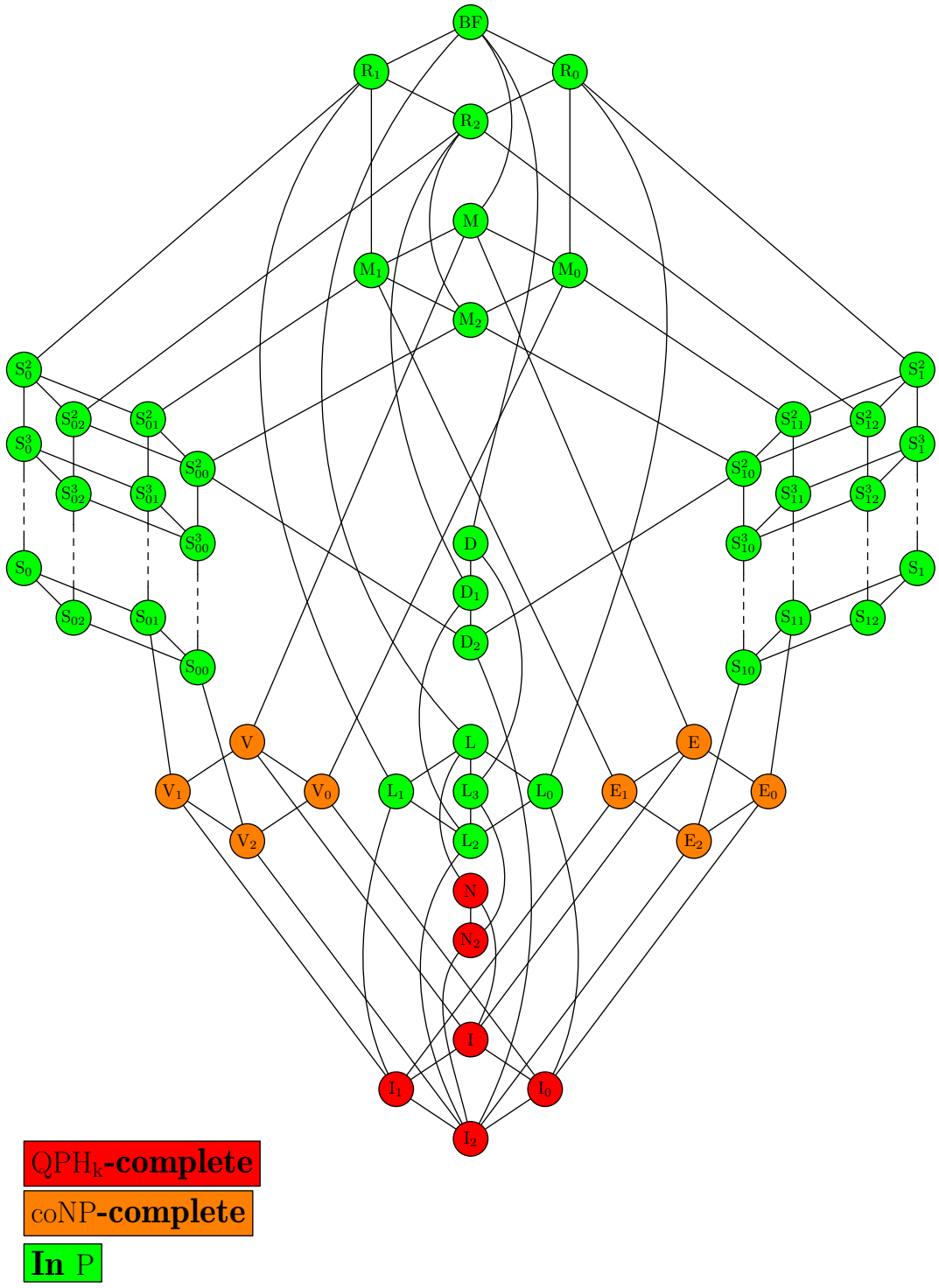

Figure 5.3: The complexity of $\mathsf{QEQUIV}_k(\Gamma)$

coincide here. We found the counting of equivalence classes to be a more natural problem than the other possible solution, where we need to demand very artificial properties of the formulas involved. For both of these problems, we achieved tight hardness results under parsimonious reductions. However, we consider our hardness results for non-Boolean domains extremely unlikely to cover all cases.

In addition, there are obvious possible generalizations of the polynomial time result for affine constraint languages to the non-Boolean case: it is easy to identify relations which basically express liner equations. An n-ary relation R over the domain $\{0, \ldots, m-1\}$ has this property if and only if R is of the form $\{(\alpha_1, \ldots, \alpha_n) \mid \alpha_1 + \cdots + \alpha_n = \alpha\}$ for some $\alpha \in \{0, \ldots, m-1\}$, where the addition is the addition of natural numbers modulo m. If m is a prime, then these relations again describe linear equations over a finite field, and the methods used for affine constraint languages can be used without modification. However, it is very much possible that in the non-Boolean case, more tractable cases for both the decision and the counting problems considered here can exist. Hence, in contrast to the Boolean case, it is unclear where the boundary between tractable and intractable cases lies, and our results do not lead to a natural conjecture. It is conceivable to obtain a full complexity classification of quantified problems for the case of 3-element domains, since such a result was achieved for unquantified formulas by Andrei Bulatov in [Bul06].

In addition to the formulas with bounded quantifier alternation that we considered here, it is also interesting to look at the case where we make no restrictions to the structure or length of the quantifier block of a formula at all. For the satisfiability problem, this already has been considered in Thomas Schaefer's original article [Sch78]. For the other problems that we defined, a classification for the unrestricted case easily follows from the proofs in this chapter. It is well-known that the satisfiability or truth evaluation problem for unrestricted quantified propositional formulas is PSPACE-complete (by again considering the problem for the constraint language Γ_{3SAT}). In light of Theorem 1.3.2, it is to be expected that the relationship of formulas with bounded quantifier alternation to the unrestricted case is similar to the relationship of the classes in the polynomial hierarchy to the class PSPACE, and this similarity is in fact mirrored by the complexity of the problems we studied in this chapter. It is obvious that PSPACE, or $\# \cdot \mathrm{PSPACE}$ is an upper bound for the decision and counting problems we considered in the case of formulas with unrestricted quantifiers. Evidently, the hardness proofs we gave in this chapter also yield hardness proofs for the class PSPACE if we study the problems for unrestricted formulas instead. However, results for this case can be obtained more easily by applying the algebraic connection between quantified constraints and surjective polymorphisms [Jea98], which we briefly mentioned in the proof for Proposition 5.3.1. A close inspection of our proofs for the polynomial time and coNP results shows that these results also hold if the quantifier structure is unrestricted. Therefore, our proofs give dichotomy theorems for the cases with unrestricted quantifiers, and hence we achieve a complete classification of the complexity in the Boolean case.

In Chapter 4, the complexity classification immediately implied the isomorphism of the involved problems to the standard complete problems, or in fact any complete problem of the corresponding complexity classes (see Corollary 4.6.2). Therefore, it is natural to ask if an analogous result easily follows from our complete complexity classifications for the Boolean case in this chapter. Questions like these have in fact been considered for

the class NP and polynomial time many-one reductions: The famous Berman-Hartmanis conjecture aims for a polynomial-time computable bijection between any two languages which are complete for NP under $\leq_m^p$-reductions. This was suggested to hold in general and indeed proven for many well-known NP-complete problems in [BH77]. But as the name suggests, this conjecture is unproven. It is immediately clear that the conjecture implies that the classes P and NP indeed differ, since otherwise, there would be a finite language which is isomorphic to an infinite one, and this obviously cannot happen. Although analogous results of the conjecture hold for stricter types of reductions (see the proof of Corollary 4.6.2) and also for the many-one reduction usually applied in recursion theory [Myh55], it is unclear if this can be indeed proven for $\leq_m^p$-reductions. In fact, it is widely believed today that the conjecture does not hold (for example, it fails relative to random oracles [KMR95], also see [You88]). Also, this conjecture would not even suffice to prove an analogous result to Corollary 4.6.2, since this would require an analogous result to the Berman-Hartmanis conjecture for all the involved classes of the polynomial hierarchy.

Concluding Remarks

We have studied restricted satisfiability problems in two contexts, that of formulas restricted by the set of propositional operators appearing, and that of constraint formulas. We studied different problems for these restriction types: the formula value problem and the enumeration problem for the first, and a refinement of the complexity of the satisfiability problem for the second type.

We showed that Schaefer's famous dichotomy, Theorem 1.5.4, can be generalized in many ways. Constraint satisfaction problems retain their dichotomy-like behavior if we refine the reductions to $\leq_m^{\mathrm{AC}^0}$-reductions, as shown in Chapter 4, lift the typical NP-complete or coNP-complete problems to the polynomial hierarchy using quantifiers, and also if we study other computational goals, even if we leave the context of decision problems, as shown in Chapter 5. In all of our dichotomy results for the Boolean domain, there is an algorithm which can be used to determine the complexity of the problem induced by a given constraint language Γ or set of functions B. The reason for this is that given Post's finite bases for the clones from Table 1.1, it easily can be tested which clone is generated by the set B, or what the clone of polymorphisms of Γ is. Additionally, the criterion from Lemma 4.4.2 can be tested by an algorithm.

For the problems considered in this thesis, we have shown that the algebraic closures given by the $[.]$-operator in the B-formula case, and the $\langle . \rangle$-operator in the constraint formula case determine the complexity of the problems in question. However, there are limitations to this result. As shown in Chapter 4, the closure operator fails to precisely determine the complexity of constraint satisfaction problems for complexity classes beyond LOGSPACE. In Chapter 5, we saw that similar issues arise when studying problems other than satisfiability. For the formula case, in order to achieve the result that the complexity of the formula value problem considered in Chapter 2, several technical restrictions on the set of Boolean functions considered were necessary.

An obvious question that our work leaves open is the question to study the "reverse combinations," of computational problems and restrictions. However, most of these problems already have been solved in the literature: The "formula value problem" for constraint formulas can be solved by a finite state machine for a fixed constraint language Γ (see the remarks at the beginning of Chapter 4). The question of enumeration algorithms in the constraint context was completely answered for the Boolean case by Nadia Creignou and Jeans-Jaques Hébrard in [CH97], and results for the non-Boolean case can be found in [SS06a]. As mentioned before, a finer analysis of the satisfiability problem for Boolean circuits, which are closely related to Boolean formulas, has been studied in [RW00]. The complexity of determining truth of quantified propositional formulas and counting problems for propositional formulas were classified by Steffen Reith in his PhD

thesis [Rei01], the classification of the counting problem for the quantified case remains open.

Another problem which might seem interesting is enumeration for quantified Boolean constraint formulas. However, it is easy to see from the results in [CH97] that in the case where Γ is Schaefer, efficient algorithms always exist. The reason for this is basically that since due to Theorem 5.6.4, we can solve the question if for a given Γ-formula φ, the formulas $\varphi[x/0]$ and $\varphi[x/1]$ are satisfiable in polynomial time. Hence, an algorithm exactly like the one presented in the proof for Theorem 3.3.1 can be shown to work here, and hence we get positive enumeration results even for the strictest notions of efficiency considered in Chapter 3. In the case where Γ is not Schaefer, an efficient enumeration algorithm cannot exist, unless the polynomial hierarchy collapses, since the question if a given $\mathsf{QCSP}_k(\Gamma)$-formula has a solution at all is complete for a class in the k-th level of the polynomial hierarchy. Therefore, enumeration results for Boolean quantified constraint formulas easily follow from known results.

There is an interesting difference between the behavior of the closure operators with respect to complexity in the different formula restriction contexts we considered: for constraint formulas, the Galois connection allows to give a general and usually quite easy proof for the fact that the complexity depends only on the closure $\langle\Gamma\rangle$ of a given constraint language - the closure operator works "*a priori*." for the formulas considered in Chapters 2 and 3, such a general result was not proven. Rather, it follows from the individual cases (the closure operator works "*a posteriori*"). These examples are typical for the complexity analysis of B-formulas, see for example the proof of the main theorem from [Lew79], results about formulas in [Rei01], and a complexity classification for B-formulas in the context of modal logic [BHSS06]. It would be very interesting to come up with a systematic way to show these "independence" results here. Lemma 1.4.4 by Lewis and our extension, Lemma 1.4.5 give a few ideas on how the way to such a general theorem might look like. It is however unclear if an analogous statement can be proven for other clones, and it seems highly unlikely that it can be achieved for *all* clones, since the proofs for the mentioned Lemmas heavily rely on the presence of both constant functions. A systematic study of these issues would be very interesting.

Lebenslauf

17. November 1978	Geburt in Husum Mutter: Wiebe Loni Schnoor geb. Böttcher Vater: Wilfried Heinrich Walter Schnoor
August 1985 - Juni 1989	Besuch der Jens-Iwersen-Schule Hattstedt
August 1989 - Juni 1998	Besuch der Theodor-Storm-Schule Husum
Juni 1998	Abitur
Juli 1998 - Juli 1999	Zivildienst in der Kirchengemeinde Hattstedt
Oktober 1999 - Juli 2004	Studium des Faches Mathematik mit Studienrichtung Informatik an der Universität Hannover
Oktober 2000 - Juli 2004	Verschiedene Täigkeiten als studentische Hilfskraft an der Universität Hannover. Darunter: Korrektur von Hausübungen, Betreuung von Programmierübungen, Abhalten von Stundenübungen, Betreuung von Praktika
Oktober 2001	Vordiplom
Juli 2004	Diplom Auszeichnung für die Diplomarbeit
Seit 15. August 2004	Wissenschaftlicher Mitarbeiter am Institut für Theoretische Informatik, Fakultät für Elektrotechnik und Informatik Leibniz Universität Hannover
Seit Oktober 2004	Promotionsstudium des Faches Informatik an der Leibniz Universität Hannover
2. Dezember 2005	Hochzeit mit Ilka Schnoor, geb. Johannsen

Bibliography

[ABI97] Eric Allender, José Balcazar, and Neil Immerman. A first-order isomorphism theorem. *SIAM Journal on Computing*, 26:557–567, 1997.

[ABI+05] Eric Allender, Michael Bauland, Neil Immerman, Henning Schnoor, and Heribert Vollmer. The complexity of satisfiability problems: Refining Schaefer's Theorem. In *Proceedings of the 30th International Symposium on Mathematical Foundations of Computer Science*, pages 71–82, 2005.

[AG00] Carme Alvarez and Raymond Greenlaw. A compendium of problems complete for symmetric logarithmic space. *Computational Complexity*, 9(2):123–145, 2000.

[Agr01] Manindra Agrawal. The first-order isomorphism theorem. In *Foundations of Software Technology and Theoretical Computer Science: 21st Conference, Bangalore, India, December 13-15, 2001. Proceedings*, Lecture Notes in Computer Science, pages 58–69, Berlin Heidelberg, 2001. Springer Verlag.

[Böh05] Elmar Böhler. *Algebraic Closures in Complexity Theory*. PhD thesis, Fakultät für Mathematik und Informatik, Universität Würzburg, 2005.

[BBC+05] Michael Bauland, Elmar Böhler, Nadia Creignou, Steffen Reith, Henning Schnoor, and Heribert Vollmer. Quantified constraints: The complexity of decision and counting for bounded alternation. Technical Report TR05-024, ECCC Reports, 2005.

[BBJK03] Ferdinand Börner, Andrei Bulatov, Peter Jeavons, and Andrei Krokhin. Quantified constraints: algorithms and complexity. In *Proceedings 17th International Workshop on Computer Science Logic*, volume 2803 of *Lecture Notes in Computer Science*, Berlin Heidelberg, 2003. Springer Verlag.

[BCC+04] Michael Bauland, Philippe Chapdelaine, Nadia Creignou, Miki Hermann, and Heribert Vollmer. An algebraic approach to the complexity of generalized conjunctive queries. In *Proceedings 7th International Conference on Theory and Applications of Satisfiability Testing*, pages 181–190, 2004.

[BCRV04] Elmar Böhler, Nadia Creignou, Steffen Reith, and Heribert Vollmer. Playing with Boolean blocks, part II: Constraint satisfaction problems. *SIGACT News*, 35(1):22–35, 2004.

[BDG90] José Balcázar, Josep Diáz, and Joaquim Gabarró. *Structural Complexity II.* Springer Verlag, Berlin Heidelberg New York, first edition, 1990.

[BDG95] José Balcázar, Josep Diáz, and Joaquim Gabarró. *Structural Complexity I.* Springer Verlag, Berlin Heidelberg New York, second edition, 1995.

[BH77] Leonard Berman and Juris Hartmanis. On isomorphism and density of NP and other complete sets. *SIAM Journal on Computing*, 6:305–322, 1977.

[BHRV02] Elmar Böhler, Edith Hemaspaandra, Steffen Reith, and Heribert Vollmer. Equivalence and isomorphism for Boolean constraint satisfaction. In *Computer Science Logic*, volume 2471 of *Lecture Notes in Computer Science*, pages 412–426, Berlin Heidelberg, 2002. Springer Verlag.

[BHSS06] Michael Bauland, Edith Hemaspaandra, Henning Schnoor, and Ilka Schnoor. Generalized modal satisfiability. In *23nd Symposium on Theoretical Aspects of Computer Science*, pages 500–511, 2006.

[BL99] Hans Kleine Büning and Theodor Lettmann. *Propositional Logic: Deduction and Algorithms*, volume 48 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1999.

[BM95] Martin Beaudry and Pierre McKenzie. Circuits, matrices, and nonassociative computation. *J. Comput. Syst. Sci.*, 50(3):441–455, 1995.

[BRSV05] Elmar Böhler, Steffen Reith, Henning Schnoor, and Heribert Vollmer. Bases for Boolean co-clones. *Information Processing Letters*, 96:59–66, 2005.

[BS05] Elmar Böhler and Henning Schnoor. The complexity of the descriptiveness of Boolean circuits over different sets of gates. Technical Report 357, Fachbereich Mathematik und Informatik, Universität Würzburg, 2005.

[Bul06] Andrei A. Bulatov. A dichotomy theorem for constraint satisfaction problems on a 3-element set. *Journal of the ACM*, 53(1):66–120, 2006.

[Bus87] Samuel R. Buss. The Boolean formula value problem is in ALOGTIME. In *Proceedings 19th Symposium on Theory of Computing*, pages 123–131. ACM Press, 1987.

[CH96] Nadia Creignou and Miki Hermann. Complexity of generalized satisfiability counting problems. *Information and Computation*, 125:1–12, 1996.

[CH97] Nadia Creignou and Jeans-J. Hébrard. On generating all solutions of generalized satisfiability problems. *Informatique Théorique et Applications/Theoretical Informatics and Applications*, 31(6):499–511, 1997.

[CKS81] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *Journal of the Association for Computing Machinery*, 28:114–133, 1981.

[CKS01] Nadia Creignou, Sanjeev Khanna, and Madhu Sudan. *Complexity Classifications of Boolean Constraint Satisfaction Problems*. Monographs on Discrete Applied Mathematics. SIAM, 2001.

[Coo71] Stephen A. Cook. The complexity of theorem proving procedures. In *Proceedings 3rd Symposium on Theory of Computing*, pages 151–158. ACM Press, 1971.

[Coo85] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[CSV84] Ashok K. Chandra, Larry J. Stockmeyer, and Uzi Vishkin. Constant depth reducibility. *SIAM Journal on Computing*, 13:423–439, 1984.

[Dal05] Victor Dalmau. Generalized majority-minority operations are tractable. In Prakash Panangaden, editor, *Proceedings of the Twentieth Annual IEEE Symp. on Logic in Computer Science, LICS 2005*, pages 438–447. IEEE Computer Society Press, June 2005.

[DHK05] Arnaud Durand, Miki Hermann, and Phokion G. Kolaitis. Subtractive reductions and complete problems for counting complexity classes. *Theoretical Computer Science*, 340(3):496–513, 2005.

[DK06] Victor Dalmau and Andrei A. Krokhin. Majority constraints have bounded pathwidth duality. Technical Report NI06017-LAA, Isaac Newton Institute for Mathematical Sciences, 2006.

[Ern04] Marcel Erné. Adjunctions and galois connections: origins, history and development. pages 1–138, 2004.

[FV98] Tomás Feder and Moshe Y. Vardi. The computational structure of monotone monadis SNP and constraint satisfaction: a study through Datalog and group theory. *SIAM Journal on Computing*, 28(1):57–104, 1998.

[Gei68] David Geiger. Closed systems of functions and predicates. *Pac. J. Math*, 27(2):228–250, 1968.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. Freeman, New York, 1979.

[Hem04] Edith Hemaspaandra. Dichotomy theorems for alternation-bounded quantified boolean formulas. *CoRR*, cs.CC/0406006, 2004.

[HV95] Lane Hemaspaandra and Heribert Vollmer. The satanic notations: counting classes beyond #P and other definitional adventures. *Complexity Theory Column 8, ACM-SIGACT News*, 26(1):2–13, 1995.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17:935–938, 1988.

[JCG97] Peter G. Jeavons, David Cohen, and Marc Gyssens. Closure properties of constraints. *Journal of the ACM*, 44(4):527–548, 1997.

[Jea98] Peter G. Jeavons. On the algebraic structure of combinatorial problems. *Theoretical Computer Science*, 200:185–204, 1998.

[JPY88] David Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. On generating all maximal independent sets. *Inf. Process. Lett.*, 27(3):119–123, 1988.

[KMR95] Stuart A. Kurtz, Stephen R. Mahaney, and James S. Royer. The isomorphism conjecture fails relative to a random oracle. *Journal of the Association for Computing Machinery*, 42:401–420, 1995.

[Lad75a] Richard E. Ladner. The circuit value problem is log space complete for P. *SIGACT News*, 7(1):12–20, 1975.

[Lad75b] Richard E. Ladner. On the structure of polynomial-time reducibility. *Journal of the ACM*, 22:155–171, 1975.

[Lew79] Harry R. Lewis. Satisfiability problems for propositional calculi. *Mathematical Systems Theory*, 13:45–53, 1979.

[Myh55] John Myhill. Creative sets. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 1:97–108, 1955.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, MA, 1994.

[Pip79] Nicholas Pippenger. On simultaneous resource bounds. In *Proceedings 20th Symposium on Foundations of Computer Science*, pages 307–311. IEEE Computer Society Press, 1979.

[Pla84] David A. Plaisted. Complete problems in the first-order predicate calculus. *Journal of Computer and System Sciences*, 29(1):8–35, 1984.

[Pos41] Emil L. Post. The two-valued iterative systems of mathematical logic. *Annals of Mathematical Studies*, 5:1–122, 1941.

[Rei01] Steffen Reith. *Generalized Satisfiability Problems*. PhD thesis, Fachbereich Mathematik und Informatik, Universität Würzburg, 2001.

[Rei05] Omer Reingold. Undirected st-connectivity in log-space. In *STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 376–385, New York, NY, USA, 2005. ACM Press.

[Ruz81] Walter L. Ruzzo. On uniform circuit complexity. *Journal of Computer and Systems Sciences*, 21:365–383, 1981.

[RV97] Kenneth W. Regan and Heribert Vollmer. Gap-languages and log-time complexity classes. *Theoretical Computer Science*, 188:101–116, 1997.

[RW00] Steffen Reith and Klaus Wagner. The complexity of problems defined by Boolean circuits. Technical Report 255, Institut für Informatik, Universität Würzburg, 2000. To appear in *Proceedings of the International Conference on Mathematical Foundation of Informatics*, Hanoi, Oct. 25–28, 1999.

[Sav70] John E. Savage. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and Systems Sciences*, 4:177–192, 1970.

[Sch78] Thomas J. Schaefer. The complexity of satisfiability problems. In *Proceedings 10th Symposium on Theory of Computing*, pages 216–226. ACM Press, 1978.

[Sch05] Henning Schnoor. The complexity of the Boolean formula value problem. Technical Report, Institut für Theoretische Informatik, Leibniz Universität Hannover, 2005.

[Smo87] Roman Smolensky. Algebraic methods in the theory of lower bounds for Boolean circuit complexity. In *Proceedings 19th Symposium on Theory of Computing*, pages 77–82. ACM Press, 1987.

[SS06a] Henning Schnoor and Ilka Schnoor. Enumerating all solutions for constraint satisfaction problems. In *Complexity of Constraints*, number 06401 in Dagstuhl Seminar Proceedings. Schloss Dagstuhl, Germany, 2006.

[SS06b] Henning Schnoor and Ilka Schnoor. New algebraic tools for constraint satisfaction. In *Complexity of Constraints*, number 06401 in Dagstuhl Seminar Proceedings. Schloss Dagstuhl, Germany, 2006.

[Sto77] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1977.

[Sze88] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26:279–284, 1988.

[Tra84] Boris A. Trakhtenbrot. A survey of Russian approaches to perebor (brute-force search) algorithms. *Annals of the History of Computing*, 6(4):384–400, 1984. Partial English translation of L. Levin, *Universal Search Problems* (in Russian), Problemy Peredachi Informatsii (= Problems of Information Transmission), 9(3), pp. 265–266, (1973).

[Tur36] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[TW92] Seinosuke Toda and Osamu Watanabe. Polynomial time 1-Turing reductions from #PH to #P. *Theoretical Computer Science*, 100:205–221, 1992.

[Val79a] Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[Val79b] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal of Computing*, 8(3):411–421, 1979.

[Vol99] Heribert Vollmer. *Introduction to Circuit Complexity – A Uniform Approach*. Texts in Theoretical Computer Science. Springer Verlag, Berlin Heidelberg, 1999.

[Wra77] Celia Wrathall. Complete sets and the polynomial-time hierarchy. *Theoretical Computer Science*, 3:23–33, 1977.

[You88] Paul Young. Juris Hartmanis: fundamental contributions to isomorphism problems. In *Structure in Complexity Theory Conference*, pages 138–154, 1988.

[Zan91] Viktória Zankó. #P-completeness via many-one reductions. *International Journal of Foundations of Computer Science*, 2:77–82, 1991.

Index