
DISSERTATION

# A Solver for Quantified Boolean Formulas in Negation Normal Form

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

Ao.Univ.Prof. Dr.rer.nat. Uwe Egly
und
Dr.techn. Stefan Woltran
als verantwortlich mitwirkendem Universitätsassistenten

E184/3
Institut für Informationssysteme
Arbeitsbereich Wissensbasierte Systeme

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

DI Martina Seidl
Matrikelnr. 9825087
Wienerstr. 230, 3400 Klosterneuburg

Wien, am 20. Februar 2007

.....

# Deutsche Zusammenfassung

Quantifizierte Boolsche Formeln (QBFs) stellen eine Erweiterung der Formeln der Aussagenlogik dar, bei der zusätzliche Quantifikationen über aussagenlogische Variable erlaubt sind. Aus der Komplexitätstheorie ist bekannt, dass beliebige QBFs PSPACE-vollständige Probleme adäquat repräsentieren können, und dass geeignete (bzgl. der Anzahl von Quantoralternationen eingeschränkte) QBFs die Problemmengen der polynomiellen Hierarchie charakterisieren. Eine Vielzahl von Problemen aus dem Bereich der Wissensrepräsentation und AI liegen in unterschiedlichen Problemmengen der polynomiellen Hierarchie oder sind PSPACE-vollständig. Diese Probleme können gelöst werden, indem man sie in entsprechende "äquivalente" QBFs übersetzt und die QBFs auswertet. Ein QBF Solver löst also das Originalproblem, indem er die resultierende Übersetzungsformel evaluiert. Wegen seiner Einfachheit und der Verfügbarkeit praktisch effizienter Solver ist dieser Übersetzungsansatz sehr attraktiv. Fast alle state-of-the-art QBF Solver sind allerdings nicht in der Lage, beliebige QBFs zu verarbeiten, sondern sie beschränken sich auf QBFs, deren Struktur fix vorgegeben ist. Konkret handelt es sich dabei um die Konjunktive-Pränex-Normal Form (prenex conjunctive normal form, PCNF). Die oben genannten Übersetzungen von Problemen nach QBF liegen üblicher Weise nicht in PCNF vor, und es ist daher eine Transformation in PCNF notwendig, bevor ein Solver die Formeln evaluieren kann. Diese Transformation ist in mehrfacher Hinsicht problematisch: (i) die Formelgröße und die Anzahl der Variablen in der Formel erhöhen sich, (ii) Information über die Struktur der Formel geht verloren, und (iii) die Transformation ist nicht deterministisch, d.h. die PCNF einer QBF ist im Allgemeinen nicht eindeutig. Wir konnten in umfassenden Tests zeigen, dass die Laufzeiten der Solver extrem von der Auswahl der verwendeten Tranformationsstrategie abhängig sind. Aus diesen Gründen verfolgen wir einen anderen Ansatz, um diese Probleme zu vermeiden: wir verzichten auf den Schritt der Normalformtransformation und verarbeiten Formeln beliebiger Struktur. Wir schlagen zwei Beweisverfahren vor, um quantifizierte Boolsche Formeln mit beliebiger Struktur auswerten zu können:

- eine Variante der Bibelschen Konnektionsmethode für QBFs;
- eine Verallgemeinerung des Verfahrens von Davis, Putnam, Logeman und Loveland.

Die meisten QBF Solver implementieren DPLL, und auch in dieser Arbeit wird besonders auf dieses Verfahren fokussiert. Der zu Grunde liegende Algorithmus ist sehr einfach, allerdings erzielt DPLL sowohl im Bereich der Evaluierung von Formeln der Aussagenlogik als auch bei der Evaluierung von QBFs momentan die größten Erfolge. Damit DPLL konkurrenzfähig ist, bedarf es Optimierungstechniken wie Dependency-Directed Backtracking

oder Miniscoping, die im auf Formeln in PCNF eingeschränkten Fall weniger mächtig oder gar nicht möglich sind. Im Solver qpro haben wir diese verallgemeinerte Variante von DPLL implementiert. Der Vergleich zu mehreren state-of-the-art Solvern in umfangreichen Tests zeigte, dass der Verzicht auf die Normalformtransformation sich tatsächlich vorteilhaft auf den Evaluierungsprozess auswirken kann, und dass qpro viele Formeln effizienter lösen kann als andere, über Jahre entwickelte Systeme.

# Abstract

Quantified Boolean formulas (QBFs) extend the formulas of propositional logic by quantification over propositional variables. From the field of complexity theory, it is well known that arbitrary QBFs are suitable to represent PSPACE-complete problems in an adequate manner and that certain QBFs, restricted with respect to the number of quantifier alternations, characterise the problem sets of the polynomial hierarchy.

Various problems of knowledge representation and artificial intelligence are located within the polynomial hierarchy or are PSPACE-complete. Such problems can be solved by translating them into "equivalent"QBFs. Therefore, a QBF solver is the means to find solutions to the original problems by evaluating the translations. Due to its simplicity and the availability of solvers which have to be proven efficient in practice, this translation approach is very attractive. But almost all state-of-the-art QBF solvers are unable to process arbitrary QBFs; they restrict themselves to QBFs with a fixed structure format, namely the prenex conjunctive normal form (PCNF).

The previously mentioned translations of different problems to QBFs are usually not in PCNF, therefore an extra transformation step is necessary before a solver can be applied. Due to multiple reasons, this transformation is problematic: (i) the size of the formula and the number of variables increase, (ii) information about the structure of the formula is lost, and (iii) the transformation is not deterministic, i.e., the PCNF of a formula is, in general, not unique. In numerous tests, we were able to show that the runtimes of the solvers depend to a great extend on the selection of the transformation strategy.

We follow a different approach to avoid those problems: we abandon the step of normal form transformation and directly process formulas of arbitrary structure. We propose two decision procedures to evaluate QBFs of arbitrary structure:

- a variant of Bibel's connection method;
- a variant of the algorithm by Davis, Putnam, Logeman, and Loveland (`DPLL`).

While most current solvers implement a variant of `DPLL`, we also focus on `DPLL`. The basic algorithm is very simple; nevertheless, it is very successful in solving formulas from propositional logic as well as in solving QBFs if certain optimisation techniques are included. We implemented a variant of `DPLL` in the solver `qpro`. In multiple tests we were able to show that the abandonment of PCNF is indeed advantageous with respect to the evaluation process and that `qpro` performs very competitively when compared to other state-of-the-art solvers.

# Acknowledgements

I am indebted to a number of people without whom this thesis would have never been completed. First of all, I would like to thank my advisors Uwe Egly and Stefan Woltran for their valuable support through the last years, the many fruitful discussions, and their comments on this text. Also, I would like to thank Nadia Creignou for the hospitality in Marseille and for the examination of this thesis.

I am deeply grateful to Gerti Kappel and A Min Tjoa who gave me the opportunity to work at the Institute of Software Technology.

Furthermore, I want to acknowledge the people who provided non-PCNF benchmarks, in particular Guiqiang Pan and Johannes Oetsch. Very special thanks go to Michael Zolda.

Last, I would like to thank my family — my parents Werner and Christa, my sister Andrea, my brother Christoph, my grandmother Elfi, and Mick, the dog — my friends, and especially Bastian for their emotional support. Mere words cannot express my gratitude.

This work was partially supported by FWF under grant P18019, ÖAD under grant Amadée 2/2006, and FFG under grant FIT-IT-810806.

# Contents

# Chapter 1

# Introduction

The investigation of reasoning, of understanding the operations of the human mind has attracted the interest of researchers working in various fields for a long time. How do humans draw conclusions? And can we go one step further and simulate and automate this ability? At the moment this seems to be a hopeless task. So we restrict ourselves to certain areas within well-defined bounds. The preferred tool of choice in computer science comprises the usage of the formal languages of logic and the implementation of these languages in practical systems.

The automation of logic deduction and reasoning by machines can be traced back for centuries. Computers, as we know them today, were not the first means capable of automatically making logical decisions (a detailed introduction of computing before computers is given in [3]). The first approach in history was to construct special-purpose machines for solving logical problems.

At the end of the 13th century, Ramon Llull, a Spanish theologian, already proposed a logical device to calculate permutations of terms in his *Ars Magna* (the goal was to prove the truths of Christianity). Around 1777, Charles Stanhope developed the *Stanhope Demonstrator* which is considered as the first machine to accept challenges from the area of mathematical logic. But not only the technical and mechanical progress played a keyrole in the continuous improvement of logical devices. Much effort was spent on the axiomatisation of logic — the names of Leibnitz, Boole, and De Morgan, among others, should be mentioned in this context.

The first machine able to process logical problems faster than a human was the logic machine by William Stanley Jevons in 1869. It was called *Logical Piano* due to its piano-like keyboard. In 1885 Allan Marquand, an American art historian, proposed to construct an electrical version of Jevon's machine. Plans have been found, but it is unclear whether it has been realised or not. It was only in 1936 that Benjamin Burack built the first electronic logic machine. But that was just the beginning: In 1947, Theodore A. Kalin and William Burckhart built a machine to calculate the truth assignments of propositional logic formulas for up to twelve variables. In the 1950s, logic machines were at their zenith as widely used and common appliances, yet at the same time, they were at the end of their line as general-purpose computers successfully entered the scene.

Ever since the introduction of digital general-purpose computers, programs have been developed to answer questions from logic by using efficient inference engines and automatic reasoning tools realised in terms of software. The enormous interest in this area is due to the fact that languages of logic can be used for the representation of application problems (like planning, scheduling, formal verification, and more). Automatic reasoning tools can be applied to evaluate the corresponding logical sentences, which now encode the original problem.

The language selected for representing a specific problem has to be expressive enough to capture all desired features of the problem domain on the one hand, on the other hand the complexity of the formalism should not be so inherent that the practical application of the decision procedure is limited to only a few pathological encodings. But even if the worst case complexity of a decision procedure seems to make it practically useless (e.g., because of exponential cost in time in the worst case complexity or even worse undecidability) and even if the problem is theoretically intractable, this does not mean that these formalisms have to be abandoned in practice. In this case, it is necessary to develop and implement clever pruning techniques to prevent the occurrence of the worst case in many situations to make the solving process possible in most cases. If a reasoning tool for a formalism becomes efficient in the average case, then it becomes attractive for many applications.

A very important and extensively studied formalism used for a multitude of encodings are *propositional Boolean formulas*. The determination of the truth value of a propositional Boolean formula is called the *Boolean Satisfiability Problem* or SAT. SAT is prototypical for the complexity class of problems in NP; indeed it was the first problem which has been shown to be NP-complete [25]. Instances of this problem appear in many different areas (e.g., hardware verification, combinatorial problems like "The Travelling Salesperson Problem" or the "3-Colouring Problem" of a graph).

The most widely implemented SAT solving algorithm was developed by Davis, Putnam, Logeman, and Loveland. The original decision method proposed by Davis and Putnam in [28] was a part of one of the first procedures to prove the validity of first-order predicate formulas explicitly designed for efficient execution on a computer. In a first step, level saturation was performed to obtain a propositional formula and in a second step, this formula was solved using Davis' and Puntnam's method. In [26], Davis, Logeman, and Loveland presented a refinement, but it was exceeded by methods like Robinson's resolution calculus [85] in the case of first-order theorem proving. The success of such methods is based on unification which allows for a target-oriented instead of saturation-based instantiation of the formulas.

After several years of disuse in the scientific community, the interest in this method (called `DPLL` after it inventors) was revived — this time it was not in the context of first-order theorem proving but in the field of satisfiability checking for propositional logic. It became obvious that the simple `DPLL` technique performed extremely well in solving this satisfiability problem. After more than 40 years, `DPLL` still ranks among the most efficient basic algorithms for complete SAT solving.

Since then, `DPLL` has been steadily enhanced and extended. Numerous extensions and pruning techniques have been developed, integrated, and practically implemented. Programs for the evaluation of propositional formulas as well as of QBFs are called *solvers*.

Despite the NP-hardness of the decision problem, many instances of propositional formulas can be solved quite efficiently in practice. In the last years, SAT solvers proved to be very effective tools for processing industrial-scale problems. Researchers spent much effort in the tuning of their solvers and the success verifies their efforts. State-of-the-art solvers are often capable of handling formula instances containing thousands of variables.

But sometimes propositional Boolean formulas are insufficient. Many real world problems fall into a harder category than NP. Even though it is not known for sure in complexity theory, such problems can probably not be formulated efficiently using propositional Boolean formulas. An alternative would be the usage of the very expressive and well understood classical first-order predicate logic. Unfortunately, predicate logic is not decidable any more. And for many problems it is too expressive. What we need is a weaker formalism — something between first-order predicate and propositional logic.

One logic suitable for the representation of such problems is the language of quantified Boolean formulas (QBFs). QBF extend the formulas of propositional logic by quantification over propositional variables. From the field of complexity theory, it is well known that arbitrary QBFs are suitable to represent PSPACE-complete problems in an adequate manner and that certain QBFs, restricted with respect to the number of quantifier alternations, characterise the problem sets in the polynomial hierarchy.

Various problems of knowledge representation and artificial intelligence are located in different complexity classes within the polynomial hierarchy or are PSPACE-complete. Such problems can be solved by translating them into "equivalent" QBFs. Therefore, a QBF solver is the means to find solutions to the original problems by evaluating the translations.

Many efficient QBF solvers have been developed with the intention of continuing the success story of the SAT solvers. But compared to SAT solvers, QBF solvers are still in their infancy as the extensive research in this area has only started a few years ago. Again, the `DPLL` algorithm (to be more precise an extension of this decision procedure) is widely used.

Most available QBF solvers are unable to process arbitrary formulas, but expect the formulas to be of a certain structure (*prenex conjunctive normal form*). Every QBF can be transformed into an equivalent QBF in prenex conjunctive normal form. Unfortunately, this transformation has its price. It usually results in an increase of formula size and the numbers of variables, a loss of structure and, in most cases, the normal form is not unique. Deciding how to perform the transformation enormously influences the solving process and it is usually impossible to predict which strategy is going to be the best, as this depends on the concrete solver used and on the kind of problem which should be solved.

In this thesis, we introduce a novel approach for the evaluation of quantified Boolean formulas. We abandon the assumption that the QBF, we want to evaluate, is transformed to prenex conjunctive normal form (PCNF), where all quantifiers are moved to the left of the formula and the remaining propositional part of the formula is transformed to conjunctive normal form. We do not restrict ourselves to process only sets of clauses. As the normal form is usually not unique, we do not depend on the optimisations made by other tools which perform this preprocessing step, since real-world problems rarely occur in PCNF.

Therefore, we integrate well researched ideas and techniques by generalising them so that formulas of arbitrary structure can be solved. We will present a prototypical implementation of the algorithm — the solver qpro. The comparison to top state-of-the-art solvers shows that our approach is very successful. The experimental evaluation yields promising and interesting results. Many classes of problems exist where other solvers encounter enormous problems; but not qpro. By taking advantage of structural information, like the scope of a quantifier, qpro is able to evaluate formulas which were formerly out of scope for any available solver.

## 1.1 Quantified Boolean Formulas

*Quantified Boolean formulas* (QBF) extend propositional formulas by allowing quantifiers over atomic propositions. An *atomic proposition* is a statement that can be interpreted as true (T) or as false (F), and which cannot be further split up. Examples are statements like "It rains." or "The street is wet." Atomic propositions are also called (propositional) *variables* or *atoms*.

Arbitrary formulas of propositional logic can then be constructed by assembling those atoms to more complex formulas which themselves can be combined to even more complex formulas and so on. To build these formulas in the language of propositional logic, the following connectives are usually available: the negation $\neg$, the conjunction $\wedge$, the disjunction $\vee$, the implication $\rightarrow$, and the equivalence $\leftrightarrow$. With this we can express a statement like "If it rains, the street is wet." by $r \rightarrow w$ where $r$ abbreviates "It rains." and $w$ stands for "The street is wet." and where the conditional is expressed by the implication. If we know that it rains, we can automatically infer the fact that the street is wet. This simple example illustrates how automatic reasoning works by making implicitly given knowledge explicit. The semantics of the connectives is well-defined and is shown in the following table where $\phi$ and $\psi$ stand for arbitrary formulas:

| $\phi$ | $\psi$ | $\neg\phi$ | $\phi \wedge \psi$ | $\phi \vee \psi$ | $\phi \rightarrow \psi$ | $\phi \leftrightarrow \psi$ |
|---|---|---|---|---|---|---|
| F | F | T | F | F | T | T |
| F | T | T | F | T | T | F |
| T | F | F | F | T | F | F |
| T | T | F | T | T | T | T |

The truth value of an atomic formula is determined by an *interpretation*. An interpretation maps each atom either to T or to F. With an interpretation and the semantics of the connectives, an arbitrary formula can be evaluated. If a formula evaluates to true under a certain interpretation, we say that this interpretation satisfies the formula or that the formula is satisfiable (i.e., an interpretation exists that makes the formula true). A formula which is satisfied by all interpretations is called valid, a formula which has no satisfying interpretation is called unsatisfiable.

Obviously, the number of all possible variable assignments is exponential in the number of variables, but checking whether a given interpretation satisfies a formula or not is possible in a polynomial amount of time with respect to the size of the formula. From

a complexity point of view, the problem of testing if a formula of propositional logic is satisfiable can be placed in NP. As mentioned before, the SAT problem is also NP-complete; it was the first problem considered as characterising the class of NP problems [25]. Dually, the validity checking problem is located in coNP.

As NP and coNP are complexity classes which contain prominent problems from very different fields like graph theory, planning, scheduling, etc., a very natural approach is to encode all of the problems as SAT problems and develop just one program to solve them all. This approach was successful and available solvers perform so well that they are able to handle problems of industrial scale and that they can be used for e.g. hardware verification in practice [14, 33, 83].

For many problems, propositional logic is insufficient as a host language, because the encoding would blow-up exponentially in size. Here we come back to the previously mentioned quantified Boolean formulas. QBFs allow for quantifiers over the propositional variables.

If we restrict the formulas to a certain structure, a QBF looks as follows:

$$\mathsf{Q}_1 x_1 \mathsf{Q}_2 x_2 \ldots \mathsf{Q}_n x_n \phi,$$

where $\mathsf{Q}_i \in \{\forall, \exists\}$, $x_i$ are propositional variables, and $\phi$ is a purely propositional formula. $\mathsf{Q}_1 x_1 \mathsf{Q}_2 x_2 \ldots \mathsf{Q}_n x_n$ is called *prefix* and $\phi$ is called *matrix*. When we introduce QBFs later, we will also allow quantifiers to occur inside the formula $\phi$, but for the moment it is sufficient to consider formulas which have the given structure. To check whether a QBF evaluates to true is a mixture between satisfiability and validity testing. A QBF is said to be true if the formula is true for every assignment of an universally quantified variable and for some assignment of an existentially quantified variable. For example, the following QBF evaluates to true

$$\forall x \exists y (x \vee \neg y) \wedge (\neg x \vee y),$$

but if we simply swap the quantifiers, the resulting formula evaluates to false:

$$\exists x \forall y (x \vee \neg y) \wedge (\neg x \vee y).$$

Intuitively, solving a QBF can be seen as a two player game between the universal player who tries to make the QBF false by all means and an existential player who tries to make the QBF true. More formally, QBFs can be considered as a restricted variety of second order logic where the arity of the predicates is restricted to zero (and there are, in consequence, no function symbols and object variables). QBFs are closely related to a very restricted subclass of classical first-order logic. This subclass allows only predicates with arity one, no function symbols and object variables. These variables are interpreted over the two-valued domain {true, false}.

## 1.2 Historical Overview

*Quantified Boolean logic* (i.e., the extension of propositional logics by quantifiers over propositional variables) has been introduced long before the first implementation of a QBF

solver. Originating from complexity theory, the practical interest in these formulas has arisen only a few years ago because of the success of SAT solvers on large-scale instances and real-world problems.

In his "Theory of Implication" [86], B. Russel provided the first analysis of systems with quantifiers over propositional variables in 1906. Further investigations were presented in "Untersuchung über den Aussagenkalkül" by Lukasiewicz and Tarski in 1930.

At the beginning of the seventies of the last century, QBFs became very popular in the context of the newly evolving field of complexity theory. Meyer and Stockmeyer [70] showed that the evaluation problem of quantified Boolean formulas is PSPACE-complete. The complexity class PSPACE contains those problems which can be computed by deterministic Turing machines requiring only polynomial space with respect to the problem size. The same authors introduced the concept of the *polynomial hierarchy* [71] in analogy to the arithmetic hierarchy.

Based on the class $\Sigma_1^p = \text{NP}$, they constructed a whole hierarchy of complexity classes. A problem which is located in NP can be calculated by a nondeterministic Turing machine in polynomial time. They defined infinitely many classes $\Sigma_{k+1}^p$ "*as the family of sets of words accepted in nondeterministic polynomial time by Turing machines with oracles for sets* $\Sigma_k^p$" [71]. They also showed that the evaluation problem for a QBF is complete for a certain class in the polynomial hierarchy depending on the number of quantifier alternations in the prefix. To be more precise, the problem of checking the satisfiability of a QBF of the form $\exists X_1 \forall X_2 \ldots \mathsf{Q}_k X_k \phi$ is complete for $\Sigma_k^p$ with $\mathsf{Q}_k = \exists$ if $k$ is odd and $\mathsf{Q}_k = \forall$ if $k$ is even and $\phi$ is purely propositional.

In [89, 94] other classes like $\Pi_k^p$ and $\Delta_k^p$ where introduced. Together with $\Sigma_k^p$ they are considered to be the elements which build up the polynomial hierarchy.

Since then, QBFs have been used as a tool for complexity analysis by showing that a QBF can be encoded efficiently in a given formalism and this formalism is therefore located at some level of the polynomial hierarchy (which can be seen by the number of quantifier alternations in the prefix of the QBF). Examples are PSPACE-completeness proofs for intuitionistic logics [88], for several modal logics [63], and for nonmonotonic logics [57].

Still it took a long time before the first solver was implemented. Finally in 1995, a first implementation was presented by Kleine-Büning et al. [59] based on an extension of resolution (Q-resolution). In 1998, Cadoli et al. [22] presented a more promising approach based on a generalisation of the algorithm by Davis, Putnam, Logeman, and Loveland (`DPLL`). As mentioned before, `DPLL` is one of the most widely used and most successful approaches to solve the SAT problem of propositional logic.

Basically, there are two approaches to tackle this problem: *bottom-up*, where the variables are eliminated from the innermost to the outermost quantifier and *top-down*, where the variable elimination is achieved vice versa. `DPLL` is a top-down decision procedure; variables are succinctly assigned truth values until the whole formula evaluates to true or to false. Alternatives to `DPLL` are the usage of *Q-resolution*, a variant of propositional resolution, where the quantifications have to be taken into account, or the usage of *binary decision diagrams* (BDD), where the QBF is represented as a directed acyclic rooted graph, which is reduced until the formula evaluates to true or to false. In the worst case,

BDDs have an exponential space requirement.

With the advancements in solver development, the number of new applications increased as well and with this, the challenges a solver has to face. The QBF community has grown continuously throughout the last years. A common communication platform is the `QBFLIB` which can be found at `www.qbflib.org`.

Today, QBFs are recognised as a promising paradigm for the encoding of various computationally hard reasoning problems from different fields of computer science (especially those located in the surroundings of artificial intelligence and knowledge representation).

## 1.3 Normal Forms

QBFs are expressive enough for the representation of many important problems for knowledge representation and AI. Unfortunately, the encoding of real-world problems in QBFs does not usually result in formulas with the desired smooth structure that is necessary for most current QBF solvers. There are two ways to deal with this problem:

1. Use solvers which are capable of processing arbitrary QBFs.
2. Transform a non-normal form QBF to normal form.

Until now, the second approach was usually taken. Before initiating the actual solving process, a preprocessing step was necessary to obtain the formula in a suitable form. This means thatthat the formula was transformed to an equivalent QBF of the following structure:

$$\mathsf{Q}_1 X_1 \mathsf{Q}_2 X_2 \dots \mathsf{Q}_n X_n \bigwedge_{i=0}^{m} (\phi_m),$$

where $\mathsf{Q}_1 X_1 \mathsf{Q}_2 X_2 \dots \mathsf{Q}_n X_n$ is a sequence of quantifiers and the $\phi_i$ are disjunctions of possibly negated propositional variables.

Fortunately, the normal form transformation of every formula terminates and always results in a normal form. Unfortunately, this normal form is not unique. And even worse, different normal forms of one formula are not of the same quality with respect to the solving process. For example, quantifiers usually occur somewhere in the formula and not only on the left-hand side. So they have to be shifted to obtain a prefix. Certain dependencies have to be respected but, to a certain degree, there is a freedom of choice (i.e., if the quantifiers occur in different subformulas they are independent of each other). Consider the following example:

$$\forall x_1 \exists y_1 ((\forall x_2 \exists y_2 \forall z_2 \phi) \vee (\exists y_3 \psi)).$$

In this example, the following dependencies hold: $x_2, y_2, z_2$, and $y_3$ depend on $x_1$, and $y_1$ and $y_2$ must not occur before $x_1$ in the prefix. The variable $z_2$ depends on $y_2$ and $x_2$. Some equivalent QBFs in prenex normal form are

- $\forall x_1 \exists y_1 y_3 \forall x_2 \exists y_2 \forall z_2 (\phi \vee \psi)$,

- $\forall x_1 \exists y_1 \forall x_2 \exists y_2 y_3 \forall z_2 (\phi \vee \psi)$,

- $\forall x_1 \exists y_1 \forall x_2 \exists y_2 \forall z_2 \exists y_3 (\phi \vee \psi)$.

The first and the second "normalised" QBF are minimal with respect to the number of quantifier alternations imposed by the original formula. According to complexity theory, one more alternation indicates that the problem is located one level higher in the polynomial hierarchy. A natural assumption is: the less quantifier alternations, the better for the solvers. During the solving process it is necessary to choose which variable is treated next, but in contrast to propositional logic, where any variable may be processed at any time, in QBF the choice of freedom for the variable selection is restricted to the currently processed quantifier block. So called branching heuristics will work better if there are fewer and therefore larger quantifier blocks.

If we do not respect the quantifier dependencies imposed by the original formula, the resulting formula, e.g., $\exists y_3 \forall x_1 \exists y_1 \forall x_2 \exists y_2 \forall z_2 (\phi \vee \psi)$ might not have the same meaning as the original QBF from the example. Therefore, this is not a correct normal formal of the original formula.

In [35], we showed that different arrangements of the quantifiers in the prefix have a great impact on the performance of the solvers. Zolda [96] refined our results and proposed further strategies to construct quantifier prefixes. In short, there are good and less good normal forms with respect to the solving process. In general, there is not always a straightforward answer regarding which shifting strategy is preferable. Together with Zolda, we developed a quantifier shifting tool, `qst`, which supports 14 different strategies for constructing the quantifier prefix. It is, of course, very time- and resource-consuming to run a test set 14 times to learn which strategy behaves better than the others. Even small pretests can be costly and time-consuming.

The ambiguity is not the only problematic aspect of the normal transformation. The disruption of the structure of the formulas is almost as bad as the multiple ways to equivalently transform the input problem. The scope of the quantifiers increases drastically when they are shifted in front of the formula and forcing a structure which allows only for propositional clauses inside the formula. A lot of information which can be profitable for the solver with respect to the running time is lost. For example, originally obvious contradictions and tautologies might be hidden by the construction of clauses. A current trend is the recovery of the original structure in the normalised QBF during the solving process to prune the search space.

We propose the solver qpro [36], which processes formulas that have a more general structure than the previously introduced formulas of restricted structure. To the best of our knowledge qpro is the only available solver which works efficiently on QBFs in non-normal form (i.e., which implements an algorithm which is polynomial in space) and which has not yet been proven to be incorrect. In numerous tests [75, 73, 76, 77, 74], we could show that qpro is able to solve real world problems. qpro performs extremely competitively compared with current state-of-the-art solvers.

## 1.4 Thesis Overview

In this thesis we show how we developed the QBF solver qpro, which is capable of processing formulas in non-prenex non-clausal normal form. We consider all different stages of the development starting from a theoretical point of view and then shifting towards the practical realization.

In Chapter 2 we review the syntax and semantics of the language of quantified Boolean formulas and some associated terminology. We further discuss syntactical and semantical properties that are the basis for the rest of this work. The following chapter provides an overview of state-of-the-art QBF solving. We take a closer look at existing approaches and we study available implementations which are only capable of processing QBFs in prenex conjunctive normal form. In Chapter 4, we argue why we decided to follow a novel approach by abandoning the restriction of the formula structure and why we investigate decision methods for QBFs of arbitrary structure. Based on these insights, we present two proof procedures — one is a generalisation of the well-known DPLL procedure, i.e., a search-based technique, the other one is a proof theoretical method, namely the connection-based matrix characterisation for QBFs. The next three chapters focus on the efficient implementation of the DPLL method. In Chapter 5, we show how to prune DPLL by the inclusion of various techniques which are used in our implementation. Finally, in Chapter 6, we show how we realized our solver qpro. In Chapter 7, we compare our implementation to current state-of-the-art solvers. In the last chapter we conclude and discuss future work.

# Chapter 2

# Preliminaries

We formally introduce the language of *quantified Boolean formulas* (QBFs) in this chapter. We review their syntax and semantics at first. Furthermore, we provide some basic terminology and prove some fundamental properties of QBFs which will be used extensively in this thesis.

## 2.1 Syntax

In this section, we discuss how to construct the set of quantified Boolean formulas and describe important syntactical properties of them.

### 2.1.1 The Construction of a QBF

**Definition 2.1.1 (Alphabet of $\mathcal{L}_\mathcal{P}$)**
The *alphabet* (or signature) of the quantified Boolean language $\mathcal{L}_\mathcal{P}$ consists of the following symbols:

1. the truth constants $\top$ (*verum*) and $\bot$ (*falsum*);
2. a countable set of *propositional variables* $\mathcal{P} = \{p, q, p_0, p_1, ...\}$;
3. the unary connective $\neg$ (*negation*);
4. the binary connectives $\vee$ (*or*) and $\wedge$ (*and*);
5. the quantifier symbols $\forall$ (*universal*) and $\exists$ (*existential*);
6. *parentheses* ( and ).

Propositional variables are also called *atoms*. When we talk about atoms in a metalanguage, we use the symbols $x$, $y$, $z$ possibly sub- or superscripted, primed, with bars, etc. Uppercase letters like $X$, $Y$, $Z$ refer to sets of atoms (i.e., to subsets of $\mathcal{P}$).

**Definition 2.1.2 (Quantified Boolean Formula)**
The *quantified Boolean language* over a set of propositional variables $\mathcal{P}$ is the smallest set $\mathcal{L}_\mathcal{P}$ such that the following conditions hold.

1. $\mathcal{P} \cup \{\top, \bot\} \subset \mathcal{L}_\mathcal{P}$.
2. If $\phi \in \mathcal{L}_\mathcal{P}$ then $(\neg\phi) \in \mathcal{L}_\mathcal{P}$.
3. If $\phi, \psi \in \mathcal{L}_\mathcal{P}$ then $(\phi \circ \psi) \in \mathcal{L}_\mathcal{P}$ where $\circ \in \{\vee, \wedge\}$.
4. If $\phi \in \mathcal{L}_\mathcal{P}$, $x \in \mathcal{P}$ and $\mathsf{Q}'x$ does not occur in $\phi$ then $(\mathsf{Q}x\ \phi) \in \mathcal{L}_\mathcal{P}$ where $\mathsf{Q}, \mathsf{Q}' \in \{\forall, \exists\}$.

A member of $\mathcal{L}_\mathcal{P}$ is called a *quantified Boolean formula* (QBF).

To name QBFs, we use lowercase Greek letters like $\phi$, $\psi$, $\gamma$ (possibly with subscripts, primes, bars, etc.). A formula of the form $\phi \vee \psi$ is called *disjunction*, a formula of the form $\phi \wedge \psi$ is called *conjunction*. Additionally, we introduce the connectives $\rightarrow$ (*implication*) and $\leftrightarrow$ (*equivalence*) as syntactic shorthands. The string $\phi \rightarrow \psi$ stands for $\neg\phi \vee \psi$ and $\phi \leftrightarrow \psi$ abbreviates $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$.

We do not restrict QBFs to a certain structure as we allow arbitrary nestings of conjunctions and disjunctions. Quantifiers may appear anywhere in the formula. Obviously, $\mathcal{L}_\mathcal{P}$ extends the language of propositional logic by the introduction of the quantifier symbols $\exists$ and $\forall$. In other words, formulas of propositional logic can be expressed by a subset of $\mathcal{L}_\mathcal{P}$.

**Example 2.1.1** The following expressions are QBFs.

1. $\top$;
2. $x$;
3. $\forall x(\exists y(x \vee (y \rightarrow z)))$;
4. $\neg(x \vee (\forall y(x \wedge y)))$.

The first and the second formula are just a constant and a variable. Therefore they are among the smallest possible formulas. The third one contains the shorthand for the implication and introduces quantifiers over the variables $x$ and $y$ (but not $z$).

**Example 2.1.2** The expression

$$\forall x(\exists x(x \vee (y \rightarrow z)))$$

is not a QBF, because of the fourth condition in Definition 2.1.2.

By convention, the outermost parentheses of a formula can be omitted. Further parentheses may be dropped if the omission does not result in ambiguities.

**Definition 2.1.3 (Rules of Precedence)**
For QBFs, we use the following *rules of precedence*.

- $\neg$, $\exists$, and $\forall$ bind stronger than $\wedge$.
- $\wedge$ binds stronger than $\vee$.
- $\vee$ binds stronger than $\rightarrow$ and $\leftrightarrow$.
- For operators of the same priority the parentheses are considered to be left-associative.

To depict a formula graphically, we introduce the notion of structure tree.

**Definition 2.1.4 (Structure Tree)**
The *structure tree* of a QBF $\phi$ is defined on the structure of $\phi$ as follows.

- Truth constants and propositional variables are the leaf nodes of the structural tree.
- The root node is labelled by $\phi$.
- A root labelled by $\neg\psi$ has one child whose label contains the formula $\psi$.
- A node labelled by $\psi_1 \circ \psi_2$ ($\circ \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$ has two children labelled by the formulas $\psi_1$ and $\psi_2$.
- A node labelled by $\mathsf{Q}x\,\psi$ ($\mathsf{Q} \in \{\forall, \exists\}$) has one child with a label containing the formula $\psi$.

To make the structure tree more compact, we do not include whole QBFs in the labels of the nodes. We reduce the formulas to their top-level symbols which are called *main connectives*.

**Definition 2.1.5 (Main Connective)**
The *main connective* of a QBF $\phi$ denotes the primary symbol of $\phi$. If $\phi = \neg\psi$ then the main connective is "$\neg$", if $\phi = \psi_1 \circ \psi_2$ then the main connective is "$\circ$" ($\circ \in \{\vee, \wedge, \leftrightarrow, \rightarrow\}$), and if $\phi = \mathsf{Q}x\,\psi$ then the main connective is "$\mathsf{Q}x$" ($\mathsf{Q} \in \{\forall, \exists\}$).

We sometimes call formulas by the name of their main connectives. For example, a QBF of the form $\neg\phi$ is called *negation*, a QBF of the form $\phi_1 \vee \phi_2$ is called *disjunction* and so on. Note that the notion of main connective is only defined for complex QBFs, i.e., variables and truth constants do not possess main connectives.

**Example 2.1.3** The structure tree of the QBF

$$\forall x \exists y((x \vee y) \wedge (y \vee x))$$

is given in Figure 2.1.

Although the formula is split into its single components, we can reconstruct the original formula by traversing the tree in-order.

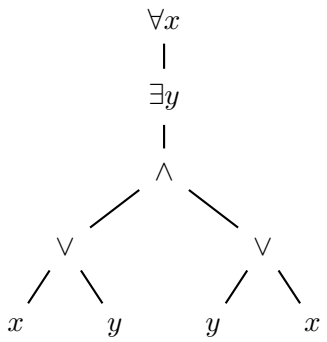



Figure 2.1: Structure tree of $\forall x \exists y((x \vee y) \wedge (y \vee x))$.

### 2.1.2 Basic Terminology and Syntactic Properties

Having defined the language of quantified Boolean formulas, we continue with the introduction of basic terminology to denote particular elements of the language.

Sometimes we are not interested in just one single QBF but in a set of formulas, e.g., for the representation of the background knowledge in a certain situation. Such a set is called a *theory*.

**Definition 2.1.6 (Theory)**
A *theory* $\mathcal{T}$ is a set of arbitrary QBFs.

**Definition 2.1.7 (n-ary Conjunction/Disjunction)**
Let $\mathcal{T} = \{\phi_1, \ldots, \phi_n\}$ be a set of QBFs where the main connective of $\phi_i$ is different from "$\wedge$" (resp. "$\vee$") for $1 \leq i \leq n$.

Then the *n-ary conjunction* is defined by

$$\bigwedge_{\phi \in \mathcal{T}} \phi = \bigwedge_{i=1}^{n} \phi_i = \phi_1 \wedge \cdots \wedge \phi_n,$$

and respectively, the *n-ary disjunction* is defined by

$$\bigvee_{\phi \in \mathcal{T}} \phi = \bigvee_{i=1}^{n} \phi_i = \phi_1 \vee \cdots \vee \phi_n.$$

If $\mathcal{T} = \emptyset$ then $\bigwedge_{\phi \in \mathcal{T}} = \top$ and $\bigvee_{\phi \in \mathcal{T}} = \bot$.

**Definition 2.1.8 (Quantifier Block)**
The sequences $\exists x_1 \exists x_2 \cdots \exists x_n$ and $\forall y_1 \forall y_2 \cdots \forall y_m$ can also be written as one *quantifier block*, namely $\exists X$ and $\forall Y$, where $X = \bigcup_{i=1}^{n} x_i$ and $Y = \bigcup_{i=1}^{m} y_i$.

Sometimes we are temporary not interested in the QBF as a whole, but we want to address certain subparts of the formula. In what follows, we assign names to often considered components.

**Definition 2.1.9 (Literal)**
A *literal* $l$ is an atom $x$ (called *positive* literal) or its negation $\neg x$ (called *negative* literal) for $x \in \mathcal{P}$.

In the following, the letters $l$ and $k$ denote literals.

**Definition 2.1.10 (Literal Variable)**
The *literal variable*, $\mathsf{var}(l)$, of a literal $l$ is $l$ if $l = x$ and $x$ if $l = \neg x$.

**Definition 2.1.11 (Complementary Literal)**
Let $l$ be a literal. The *complementary literal* $\overline{l}$ is defined as follows.

$$\overline{l} = \begin{cases} x & \text{if } l = \neg x; \\ \neg x & \text{if } l = \ \ x. \end{cases}$$

**Definition 2.1.12 (Complementary Quantifier Symbol)**
The *complementary quantifier* $\overline{\mathsf{Q}}$ of a quantifier $\mathsf{Q}$ is defined as follows.

$$\overline{\mathsf{Q}} = \begin{cases} \forall & \text{if } \mathsf{Q} = \exists; \\ \exists & \text{if } \mathsf{Q} = \forall. \end{cases}$$

**Definition 2.1.13 (Subformula)**
The set of *subformulas* of a QBF $\phi$ (denoted by $\mathsf{sub}(\phi)$) is the smallest set that contains $\phi$ and which is closed under the following rules:

1. If $\neg\psi$ is a subformula of $\phi$, so is $\psi$.
2. If $\psi_1 \circ \psi_2$ is a subformula of $\phi$, so are $\psi_1$ and $\psi_2$, where $\circ \in \{\vee, \wedge\}$.
3. If $\mathsf{Q}x\,\psi$ is a subformula of $\phi$, so is $\psi$, where $\mathsf{Q} \in \{\exists, \forall\}$.

**Definition 2.1.14 (Proper Subformula)**
If $\psi$ is a *subformula* of $\phi$ and if $\phi \neq \psi$, then $\psi$ is called a *proper* subformula of $\phi$.

**Definition 2.1.15 (Cardinality of a Formula)**
The *cardinality of a formula* $\phi$ (written as $|\phi|$) is $(|\mathsf{sub}(\phi)| - 1)$, i.e., the cardinality of a formula is given by the number of its proper subformulas.

**Example 2.1.4** Let $\phi = \forall x \exists y((x \to y) \wedge (y \to x))$. Then

$$\begin{aligned} \mathsf{sub}(\phi) = \{&\forall x \exists y((x \to y) \wedge (y \to x)), \\ &\exists y((x \to y) \wedge (y \to x)), \\ &(x \to y) \wedge (y \to x), \\ &(x \to y), (y \to x), x, y\} \end{aligned}$$

Note that all elements of $\mathsf{sub}(\phi)$ are proper subformulas except the first. Obviously, $|\phi| = 6$.

**Definition 2.1.16 (Complexity of a QBF)**
The *complexity* $\mathsf{k}(\phi)$ of a QBF $\phi$ is given as follows.

$$\mathsf{k}(\phi) = \begin{cases} 0 & \text{if } \phi \in (\mathcal{P} \cup \{\top, \bot\}); \\ 1 + \mathsf{k}(\psi) & \text{if } \phi = \neg\psi; \\ 1 + \mathsf{k}(\psi_1) + \mathsf{k}(\psi_2) & \text{if } \phi = \psi_1 \circ \psi_2, \circ \in \{\vee, \wedge\}; \\ 1 + \mathsf{k}(\psi) & \text{if } \phi = \mathsf{Q}x\,\psi, \mathsf{Q} \in \{\forall, \exists\}. \end{cases}$$

The complexity of a QBF counts the number of occurrences of connectives and quantifiers. Parentheses, atoms, and truth constants are ignored.

**Example 2.1.5** Let $\phi$ be a QBF of the form

$$\forall x \exists y((\neg x \vee y) \wedge (\neg y \vee x))).$$

Then $\mathsf{k}(\phi) = 7$.

**Definition 2.1.17 (Scope of a Quantifier)**
The *scope* of a quantifier (occurrence) $\mathsf{Q}x$ in a QBF $\phi$ is defined as $\psi$, where $\mathsf{Q}x\,\psi$ is a subformula of $\phi$. An occurrence of a variable $x$ is called existential (resp. universal) if it is located within the scope of a quantifier $\exists x$ (resp. $\forall x$).

**Definition 2.1.18 (Free Variable Occurrences)**
The *free variable occurrences* $\mathsf{free}(\phi)$ of a QBF $\phi$ are defined as follows:

- If $\phi = \top$ or $\phi = \bot$ then $\mathsf{free}(\phi) = \emptyset$.
- If $\phi = x$ and $x \in \mathcal{P}$ then $\mathsf{free}(\phi) = \{x\}$.
- If $\phi = \neg\psi$ then $\mathsf{free}(\phi) = \mathsf{free}(\psi)$.
- If $\phi = \phi_1 \circ \phi_2$ and $\circ \in \{\vee, \wedge, \leftarrow, \leftrightarrow\}$ then $\mathsf{free}(\phi) = (\mathsf{free}(\phi_1) \cup \mathsf{free}(\phi_2))$.
- If $\phi = \mathsf{Q}x\,\psi$ and $\mathsf{Q} \in \{\forall, \exists\}$ then $\mathsf{free}(\phi) = (\mathsf{free}(\psi) \backslash \{x\})$.

An occurrence of a variable $x \in \mathcal{P}$ is called *bound* in a QBF $\phi$ if $x$ occurs within the scope of a quantifier $\mathsf{Q}x$ in $\phi$ ($\mathsf{Q} \in \{\forall, \exists\}$). By $\mathsf{bound}(\phi)$ we denote the set of bound variables of the QBF $\phi$.

The expression $\mathsf{vars}(\phi)$ denotes the set of all variables occurring in $\phi$. I.e., $\mathsf{vars}(\phi)$ is the union of $\mathsf{free}(\phi)$ and $\mathsf{bound}(\phi)$. A QBF $\phi$ is *closed* if $\mathsf{free}(\phi) = \emptyset$. We call a closed formula also a *sentence*.

As we will see in Section 2.2, closed formulas can be evaluated "in a more simple fashion". For example, the QBF $x$ evaluates to true or to false depending how $x$ is interpreted whereas the truth value of $\forall x\,x$ is definitely false.

**Definition 2.1.19 (Fresh Variable)**
A variable $x$ is *fresh* with respect to a QBF $\phi$ if $x \notin (\mathsf{bound}(\phi) \cup \mathsf{free}(\phi))$.

**Definition 2.1.20 (Polarity of a Variable)**
Let $\phi$ be a QBF. A specific occurrence $x'$ of a variable $x$ is *positive* (resp. *negative*) provided

- $\phi = x'$ (resp. $\phi = \neg x'$);
- $\phi = \neg\psi$ and $x'$ is negative (resp. positive) in $\psi$;
- $\phi = \psi_1 \circ \psi_2$, $\circ \in \{\wedge, \vee\}$ and $x'$ is positive (resp. negative) in $\psi_i$ ($i \in \{1, 2\}$);
- $\phi = (\mathsf{Q}y\,\psi)$, $\mathsf{Q} \in \{\forall, \exists\}$ and $x$ is positive (resp. negative) in $\psi$.

The function $\mathsf{lit} : \mathcal{L}_{\mathcal{P}} \times \mathcal{P} \rightarrow \{pos, neg, both, none\}$ returns in which polarity a variable occurs in a formula:

$$\mathsf{lit}(\phi, x) = \begin{cases} none & \text{if } x \notin \mathsf{vars}(\phi); \\ pos & \text{if all occurences of } x \text{ are positive in } \phi; \\ neg & \text{if all occurences of } x \text{ are negative in } \phi; \\ both & \text{otherwise.} \end{cases}$$

**Definition 2.1.21 (Pure Literal)**
A literal $l$ with $\mathsf{var}(l) = x$ is *pure* in a QBF $\mathsf{Q}x\,\phi$ if $\mathsf{lit}(\phi, x) = pos$ or $\mathsf{lit}(\phi, x) = neg$ ($\mathsf{Q} \in \{\forall, \exists\}$).

**Definition 2.1.22 (Local Unit Literal)**
A literal $l$ is *local unit* with respect to $\psi$ in a QBF $\phi$ if $\phi$ contains a subformula $(l \vee \psi)$ or $(l \wedge \psi)$.

**Definition 2.1.23 (Global Unit Literal)**
A literal $l$ is *global unit* with respect to a QBF $\phi$ if $\phi$ contains a subformula

$$\mathsf{Q}_1 X_1 \ldots \mathsf{Q}_n X_n \mathsf{Q}_{n+1} Y_1 \mathsf{Q}_{n+2} Y_2 \ldots \mathsf{Q}_{n+m} Y_m (\psi \circ l),$$

where $\mathsf{var}(l) = x$, $\circ \in \{\vee, \wedge\}$, and for all $1 \leq n \leq (n+m)$ and for all $1 \leq j < (n+m)$, $\mathsf{Q}_j \in \{\forall, \exists\}$, $\mathsf{Q}_i \neq \mathsf{Q}_{i+1}$.

**Definition 2.1.24 (Substitution)**
Let $\phi$, $\psi$ and $\gamma$ be QBFs. We define the substitution of $\phi$ by $\psi$ in $\gamma$ (written as $\gamma[\phi/\psi]$) as follows.

$$\gamma[\phi/\psi] = \begin{cases} \psi & \text{if } \gamma = \phi; \\ x & \text{if } \gamma = x,\ x \in \mathcal{P}; \\ \neg\gamma'[\phi/\psi] & \text{if } \gamma = \neg\gamma'; \\ \gamma_1[\phi/\psi] \circ \gamma_2[\phi/\psi] & \text{if } \gamma = \gamma_1 \circ \gamma_2; \\ \mathsf{Q}x\ \gamma'[\phi/\psi] & \text{if } \gamma = \mathsf{Q}x\ \gamma'. \end{cases}$$

Occurrences of $\phi$ in $\psi$ are not replaced by $\psi$ and therefore, the process of substitution terminates.

The last term introduced in this subsection is the notion of *propositional skeleton*. Roughly speaking, the propositional skeleton of a QBF $\phi$ is the propositional formula obtained by removing all quantifier occurrences from $\phi$.

**Definition 2.1.25 (Propositional Skeleton)**
The *propositional skeleton*, $\mathsf{psk}(\phi)$, of a QBF $\phi$ is given by

$$\mathsf{psk}(\phi) = \begin{cases} x & \text{if } \phi = x, x \in \mathcal{P}; \\ \neg(\mathsf{psk}(\psi)) & \text{if } \phi = \neg\psi; \\ \mathsf{psk}(\psi_1) \circ \mathsf{psk}(\psi_2) & \text{if } \phi = \psi_1 \circ \psi_2,\ \circ \in \{\vee, \wedge\}; \\ \mathsf{psk}(\psi) & \text{if } \phi = \mathsf{Q}x\ \psi,\ \mathsf{Q} \in \{\exists, \forall\}. \end{cases}$$

### 2.1.3 Normal Forms

Sometimes it is easier to establish results about QBFs when they are standardised with respect to a certain structure. In other words, not the whole set $\mathcal{L}_\mathcal{P}$ is of interest but a subset which contains formulas whose structure obey a certain schema. The usage of such a *normal form* is motivated for example by the better manageability and easier adaptability of formulas with restricted syntax and by the applicability of certain decision methods (e.g., resolution).

**Definition 2.1.26 (Standard Form)**
A QBF $\phi$ is in *standard form* if for every subformula $\psi_1 \circ \psi_2$ of $\phi$ ($\circ \in \{\vee, \wedge\}$), $x \notin \mathsf{vars}(\psi_1)$ (resp. $x \notin \mathsf{vars}(\psi_2)$) if $\psi_2$ (resp. $\psi_1$) contains a subformula of the form $\mathsf{Q}x\,\psi$ with $\mathsf{Q} \in \{\forall, \exists\}$.

**Definition 2.1.27 (Negation Normal Form)**
A QBF $\phi$ in standard form is in *negation normal form* (NNF) if

1. $\phi = \top$ or $\phi = \bot$;
2. $\phi = x$ or $\phi = \neg x$ with $x \in \mathcal{P}$;
3. $\phi = \psi_1 \circ \psi_2$ and $\psi_1$ and $\psi_2$ are in NNF with $\circ \in \{\vee, \wedge\}$;
4. $\phi = \mathsf{Q}x\,\psi$ and $\psi$ is NNF with $\mathsf{Q} \in \{\exists, \forall\}$.

Negation normal form does not impose a very strong restriction on the structure of a formula and, as we will see, the transformation into NNF is inexpensive in terms of structure loss, increase of formula size, introduction of new variables, etc. The transformation can be achieved deterministically by the rules which will be introduced in Theorem 2.2.1 below.

Unless stated otherwise, all considered QBFs are assumed to be closed, in standard form and in negation normal form which can be obtained deterministically and which only causes a neglectable increase of the formula size.

Note that special care has to be taken if equivalences are allowed connectives. If the equivalences are nested, an exponential blow-up can happen. It can be avoided by labels, which abbreviate the corresponding formula parts can be introduced (which results in an increase of the variable number). We will not use this connective in our encodings and therefore, we do not have this problem but for the sake of completeness, we included it in our language.

**Definition 2.1.28 (Prenex Normal Form)**
A QBF $\phi$ is said to be in *prenex normal form* (PNF) if $\phi$ is in standard form and of the structure

$$\mathsf{Q}_1 X_1 \mathsf{Q}_2 X_2 \ldots \mathsf{Q}_n X_n \psi,$$

where $\psi$ is a purely propositional formula, i.e., no quantifiers occur in $\psi$, the $X_i$ are disjoint sets of propositional variables ($1 \leq i \leq n$), $\mathsf{Q}_j \in \{\forall, \exists\}$ ($1 \leq j \leq n$), and $\mathsf{Q}_k \neq \mathsf{Q}_{k+1}$ for $1 \leq k < n$. The quantifier sequence $\mathsf{Q}_1 X_1 \mathsf{Q}_2 X_2 \ldots \mathsf{Q}_n X_n$ is called *prefix* and $\psi$ is called *matrix* of $\phi$.

**Definition 2.1.29 (Clause, Prenex Conjunctive Normal Form)**
A *clause* is a disjunction of literals. A QBF $\phi$ is in *prenex conjunctive normal form* if $\phi$ is in PNF and of the form

$$\mathsf{Q}_1 X_1 \mathsf{Q}_2 X_2 \ldots \mathsf{Q}_n X_n \bigwedge_{i=0}^{m} (\phi_m),$$

where the $\phi_i$ are clauses. Obviously, the matrix of a formula in PCNF consists of a conjunction of clauses.

Within this thesis, the normal form, the normal form transformation, and how to avoid it play a key role. A detailed discussion about this topic will be given in Chapter 4. Most state-of-the-art solvers are only able to evaluate formulas in PCNF. But as the encoding of practical problems does usually not result in a PCNF formula and as the transformation causes a severe change in the structure of the formula, we investigate if it is preferable to omit the transformation and to directly process the formula in its original structure. We develop decision methods which do not rely on restrictions on the formula structure stronger than NNF to overcome certain limitations and to avoid the costs induced by the normal form transformation (e.g., loss of structural information, increase of the formula size, etc.).

## 2.2 Semantics

In this section, we present the semantics of quantified Boolean formulas and establish some basic results about their semantical properties.

### 2.2.1 The Meaning of a QBF

Like the formulas of propositional logic, QBFs are two-valued: QBFs always evaluate to *true* or to *false*. We take the set $\{\mathsf{T}, \mathsf{F}\}$ as the set of truth values. $\mathsf{T}$ represents *truth*, $\mathsf{F}$ stands for *falsehood*. To define the semantics of a QBF, we need the concept of a *variable*

*assignment.*

**Definition 2.2.1 (Interpretation Function)**
Let $\mathcal{S}$ be a set of literals where $l \in \mathcal{S}$ implies that $\overline{l} \notin \mathcal{S}$. Then the *interpretation function* $\iota_{\mathcal{S}} : \mathcal{P} \to \{\mathsf{T}, \mathsf{F}, \mathsf{U}\}$ with respect to $\mathcal{S}$ is defined as follows:

$$\iota_{\mathcal{S}}(x) = \begin{cases} \mathsf{T} & \text{if } x \in \mathcal{S}; \\ \mathsf{F} & \text{if } \neg x \in \mathcal{S}; \\ \mathsf{U} & \text{otherwise.} \end{cases}$$

We often use the term "interpretation" instead of interpretation function. The letter $I$ denotes the set of all possible interpretations. If we talk about an arbitrary interpretation, the subscripted $\mathcal{S}$ can be omitted, i.e., we just write $\iota(l)$.

**Definition 2.2.2 (Evaluation Function for QBFs)**
Let $\phi$ be a QBF in standard form and let $\mathcal{S}$ be a set of literals such that for each $x \in \mathsf{free}(\phi)$, either $x \in \mathcal{S}$ or $\neg x \in \mathcal{S}$ (but not both and nothing else). The *evaluation function* $\upsilon_{\mathcal{S}} : \mathcal{L}_{\mathcal{P}} \to \{\mathsf{T}, \mathsf{F}\}$ is inductively defined on the structure of $\phi$ as follows.

- $\upsilon_{\mathcal{S}}(\top) = \mathsf{T}$ and $\upsilon_{\mathcal{S}}(\bot) = \mathsf{F}$
- $\upsilon_{\mathcal{S}}(x) = \iota_{\mathcal{S}}(x)$ for $x \in \mathcal{P}$
- $\upsilon_{\mathcal{S}}(\neg\phi') = \begin{cases} \mathsf{T} & \text{if } \upsilon_{\mathcal{S}}(\phi') = \mathsf{F} \\ \mathsf{F} & \text{if } \upsilon_{\mathcal{S}}(\phi') = \mathsf{T} \end{cases}$
- $\upsilon_{\mathcal{S}}(\phi_1 \vee \phi_2) = \begin{cases} \mathsf{T} & \text{if } \upsilon_{\mathcal{S}}(\phi_1) = \mathsf{T} \text{ or } \upsilon_{\mathcal{S}}(\phi_2) = \mathsf{T} \\ \mathsf{F} & \text{otherwise} \end{cases}$
- $\upsilon_{\mathcal{S}}(\phi_1 \wedge \phi_2) = \begin{cases} \mathsf{T} & \text{if } \upsilon_{\mathcal{S}}(\phi_1) = \mathsf{T} \text{ and } \upsilon_{\mathcal{S}}(\phi_2) = \mathsf{T} \\ \mathsf{F} & \text{otherwise} \end{cases}$
- $\upsilon_{\mathcal{S}}(\forall x\phi') = \begin{cases} \mathsf{T} & \text{if } \upsilon_{\mathcal{S}'}(\phi') = \mathsf{T} \text{ for all } \mathcal{S}' \in \{\mathcal{S} \cup \{x\},\ \mathcal{S} \cup \{\neg x\}\} \\ \mathsf{F} & \text{otherwise} \end{cases}$
- $\upsilon_{\mathcal{S}}(\exists x\phi') = \begin{cases} \mathsf{T} & \text{if } \upsilon_{\mathcal{S}'}(\phi') = \mathsf{T} \text{ for some } \mathcal{S}' \in \{\mathcal{S} \cup \{x\},\ \mathcal{S} \cup \{\neg x\}\} \\ \mathsf{F} & \text{otherwise} \end{cases}$

At the moment, our definition of the semantics of QBFs might seem very complicated, but as we need partial interpretations later on, it will facilitate many definitions and formulations of algorithms later in this thesis. For closed formulas, the subscripted $\mathcal{S}$ can be omitted. The evaluation function — and therefore the proof of a closed QBF — can be graphically represented by the *semantic tree.*

**Definition 2.2.3 (Semantic Tree)**
The *semantic tree* of a QBF $\phi$ is a binary tree whose nodes are labelled by the the variables of $\phi$, whose edges are labelled by $\top$ and $\bot$, and whose leaves are labelled by $\bot$ and $\top$, according to the evaluation function applied on $\phi$.

If we would strictly define the semantic tree according to the evaluation function, we would have to label the branches with $x$ (resp. $\neg x$). As the quantifier rules of the evaluation function could also be seen as a replacement of the quantified variable by $\top$ and/or $\bot$, and as this notation is more convenient for the proof procedures we will introduce, we prefer to use the truth constants.

A path from the root to one leaf of the tree describes one single assignment. The whole semantic tree contains all possible assignments necessary to obtain the truth value of a formula and therefore, a semantic tree depicts a proof of a QBF.

**Example 2.2.6** Let $\phi$ be a QBF of the form

$$\forall x_1 \exists y_1 (\forall x_2 \exists y_2 ((x_2 \vee \neg y_2) \wedge (\neg x_2 \vee y_2)) \wedge (x_1 \vee y_1)).$$

The semantic tree of $\phi$ is shown in Figure 2.2. Note that the branches which are not necessary for the evaluation of the formula are not included. For example, if an existential variable is assigned the truth constant $\bot$ and under this assignment the formula evaluates to true then it is not necessary to consider the case where $\top$ is assigned.

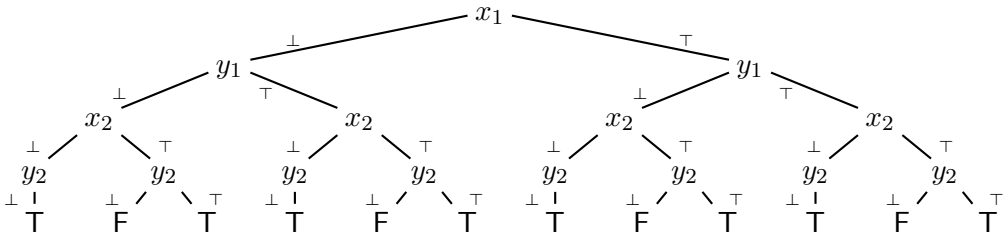



Figure 2.2: Semantic tree of a QBF.

**Definition 2.2.4 ((Un)Satisfiability, Model, Validity)**
A QBF $\phi$ is *satisfied* by an interpretation $\iota_{\mathcal{S}}$ if $\upsilon_{\mathcal{S}}(\phi) = \mathsf{T}$. Then $\iota_{\mathcal{S}}$ is called a *model* of $\phi$ and we write $\iota_{\mathcal{S}} \models \phi$. We denote the set of $\phi$'s models by $\mathsf{Mod}(\phi)$. If $\mathsf{Mod}(\phi) \neq \emptyset$ then $\phi$ is *satisfiable*. Otherwise, $\phi$ is *unsatisfiable*. If every interpretation of $\phi$ is a model, $\phi$ is said to be *valid*.

Note that closed QBFs are either valid or unsatisfiable as the evaluation is independent of any interpretation.

**Definition 2.2.5 (Contradiction, Tautology)**
A formula $\phi$ with $\mathsf{Mod}(\phi) = \emptyset$ (i.e., a formula without models) is called a *contradiction*. In contrast, a formula $\phi$ with $\mathsf{Mod}(\phi) = I$ (i.e., $\phi$ is valid) is called a *tautology*.

**Definition 2.2.6 (Equivalence)**
The QBFs $\phi$ and $\psi$ are *logically equivalent* (written as $\phi \Leftrightarrow \psi$) iff $\upsilon_{\mathcal{S}}(\phi) = \upsilon_{\mathcal{S}}(\psi)$ for every interpretation $\iota_{\mathcal{S}}$.

**Lemma 2.2.1 (Validity and Equivalence Checking)**
Let $\phi$ and $\psi$ be QBFs. Then the following holds.

- $\phi$ is valid iff $\neg\phi$ is unsatisfiable.
- $\phi$ is satisfiable iff $\neg\phi$ is not valid.
- $\phi$ is valid iff $\phi$ is equivalent to $\top$.
- $\phi$ and $\psi$ are logically equivalent iff $\phi \leftrightarrow \psi$ is valid.

**Lemma 2.2.2 (Satisfiability of a Theory)**
Let $\mathcal{T}$ be a theory, i.e., $\mathcal{T} = \{\phi_1, \ldots, \phi_n\}$. Then $\mathcal{T}$ is satisfiable iff $\phi_1 \wedge \cdots \wedge \phi_n$ is satisfiable.

### 2.2.2 Semantical Properties

In the following, we present important properties which will be used extensively in the solving process to simplify and to evaluate formulas. The following theorem introduces many useful equivalences and implications for reasoning. Then we state and prove two important theorems adapted for QBFs: (1) the Equivalence Replacement Theorem and (2) the Monotonic Replacement Theorem.

**Theorem 2.2.1 (Important Properties)**

| | | |
|---|---|---|
| (1) | $\neg\top \Leftrightarrow \bot$<br>$\neg\bot \Leftrightarrow \top$ | (truth constant negation) |
| (2) | $\neg\neg\phi \Leftrightarrow \phi$ | (removal of double negation) |
| (3) | $(\phi_1 \wedge \phi_2) \Leftrightarrow (\phi_2 \wedge \phi_1)$<br>$(\phi_1 \vee \phi_2) \Leftrightarrow (\phi_2 \vee \phi_1)$ | (commutativity) |
| (4) | $((\phi_1 \wedge \phi_2) \wedge \phi_3) \Leftrightarrow (\phi_1 \wedge (\phi_2 \wedge \phi_3))$<br>$((\phi_1 \vee \phi_2) \vee \phi_3) \Leftrightarrow (\phi_1 \vee (\phi_2 \vee \phi_3))$ | (associativity) |
| (5) | $\phi_1 \vee (\phi_2 \wedge \phi_3) \Leftrightarrow (\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3)$<br>$\phi_1 \wedge (\phi_2 \vee \phi_3) \Leftrightarrow (\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3)$ | (distributivity) |
| (6) | $\phi \vee \phi \Leftrightarrow \phi$<br>$\phi \wedge \phi \Leftrightarrow \phi$ | (idempotency) |
| (7) | $\phi \vee (\phi \wedge \psi) \Leftrightarrow \phi$<br>$\phi \wedge (\phi \vee \psi) \Leftrightarrow \phi$ | (adjunctivity) |
| (8a) | $\neg(\phi \vee \psi) \Leftrightarrow \neg\phi \wedge \neg\psi$<br>$\neg(\phi \wedge \psi) \Leftrightarrow \neg\phi \vee \neg\psi$ | (De Morgan's laws) |

| | | |
|---|---|---|
| (8b) | $\neg\exists x\,\phi \Leftrightarrow \forall x\,\neg\phi$<br>$\neg\forall x\,\phi \Leftrightarrow \exists x\,\neg\phi$ | (De Morgan's laws for quantifiers) |
| (9) | $\neg\phi \vee \phi \Leftrightarrow \top$ | (excluded middle) |
| (10) | $\phi \wedge \neg\phi \Leftrightarrow \bot$ | (contradiction) |
| (11) | $\top \wedge \phi \Leftrightarrow \phi$<br>$\bot \wedge \phi \Leftrightarrow \bot$<br>$\top \vee \phi \Leftrightarrow \top$<br>$\bot \vee \phi \Leftrightarrow \phi$ | (removal of truth constants) |
| (12) | $\forall x\,\top \Leftrightarrow \top$<br>$\forall x\,\bot \Leftrightarrow \bot$<br>$\exists x\,\top \Leftrightarrow \top$<br>$\exists x\,\bot \Leftrightarrow \bot$ | (constant quantification) |
| (13) | $\forall x\,x \Leftrightarrow \bot$<br>$\exists x\,x \Leftrightarrow \top$<br>$\forall x\,(\neg x) \Leftrightarrow \bot$<br>$\exists x\,(\neg x) \Leftrightarrow \top$ | (literal quantification) |
| (14) | $\forall x\,\phi \Leftrightarrow \phi[x/\bot] \wedge \phi[x/\top]$<br>$\exists x\,\phi \Leftrightarrow \phi[x/\bot] \vee \phi[x/\top]$ | (quantifier replacement) |
| (15) | $\forall x\forall y\,\phi \Leftrightarrow \forall y\forall x\,\phi$<br>$\exists x\exists y\,\phi \Leftrightarrow \exists y\exists x\,\phi$ | (quantifier commutability) |
| (16) | for $\mathsf{Q} \in \{\forall, \exists\}$ and $x \notin \mathsf{free}(\phi)$, $(\mathsf{Q}x\,\phi) \Rightarrow \phi$ | (quantifier elimination) |
| (17) | $\forall x\,(\phi \wedge \psi) \Rightarrow (\forall x\,\phi) \wedge (\forall x\,\psi)$<br>$\exists x\,(\phi \vee \psi) \Rightarrow (\exists x\,\phi) \vee (\exists x\,\psi)$<br>if $x \notin \mathsf{free}(\psi)$, $\forall x\,(\phi \vee \psi) \Rightarrow (\forall x\,\phi) \vee \psi$<br>if $x \notin \mathsf{free}(\psi)$, $\exists x\,(\phi \wedge \psi) \Rightarrow (\exists x\,\phi) \wedge \psi$ | (miniscoping) |
| (18) | $\forall x\,\phi \wedge \forall y\,\psi \Rightarrow \forall x\,(\phi \wedge \psi[y/x])$<br>$\exists x\,\phi \vee \exists y\,\psi \Rightarrow \exists x\,(\phi \vee \psi[y/x])$ | (quantifier fusion) |
| (19) | $\forall x\,(\phi \wedge \psi[y/x]) \Rightarrow \forall x\,\phi \wedge \forall y\,\psi$<br>$\exists x\,(\phi \vee \psi[y/x]) \Rightarrow \exists x\,\phi \vee \exists y\,\psi$ | (quantifier distribution) |

**Proof.** All the equivalences and implications above can be easily shown by the semantics of the QBFs. □

**Theorem 2.2.2 (Equivalence Replacement Theorem)**
Let $\phi[\psi]$ be a QBF $\phi$ and let $\psi$ be a subformula of $\phi$. Furthermore, let $\psi \Leftrightarrow \gamma$. Then it holds that $\phi[\psi] \Leftrightarrow \phi[\gamma]$.

**Proof.** We proof the Equivalence Replacement Theorem by induction on the complexity $\mathsf{k}(\phi)$ of $\phi$ minus the complexity of $\mathsf{k}(\psi)$ of $\psi$, i.e., by $|\mathsf{k}(\phi) - \mathsf{k}(\psi)|$.

**Basis Step.** In the first step, we show that the Equivalence Replacement Theorem holds for (1) $\phi = \psi$ and (2) for atomic formulas and truth constants. Then $(\mathsf{k}(\phi) - \mathsf{k}(\psi) = 0)$.

(1) Let $\phi = \psi$. Then $\phi[\psi] = \psi$ and $\phi[\gamma] = \gamma$. From $\psi \Leftrightarrow \gamma$, it follows that $\phi[\psi] \Leftrightarrow \phi[\gamma]$.
(2) Let $\phi' \in \mathcal{P} \cup \{\bot, \top\}$ and $\phi = \phi'$ with $\phi' \neq \psi$ and $\phi' \neq \gamma$. Then $\phi[\psi] = \phi'$ and $\phi[\gamma] = \phi'$ and therefore, $\phi[\psi] \Leftrightarrow \phi[\gamma]$. So the Equivalence Replacement Theorem holds for atomic formulas and truth constants.

**Induction Hypothesis.** Let $n \geq 0$ and assume that the Equivalence Replacement Theorem holds for the QBFs $\phi$ and $\psi$ with $(\mathsf{k}(\phi) - \mathsf{k}(\psi)) \leq n$.

**Induction Step.** We have to show that the Equivalence Replacement Theorem also holds for $(\mathsf{k}(\phi) - \mathsf{k}(\psi)) = n + 1$.
Five cases can be distinguished with respect to the main connective of $\phi$. We prove the Equivalence Replacement Theorem exemplarily for the disjunction and for the existential quantification — for the other connectives the proof is performed accordingly.

(1) *Disjunction*

Let $\phi[\psi] = \phi_1[\psi] \vee \phi_2[\psi]$ and $\phi[\gamma] = \phi_1[\gamma] \vee \phi_2[\gamma]$ and $\psi \Leftrightarrow \gamma$. By the induction hypothesis, $\phi_1[\psi] \Leftrightarrow \phi_1[\gamma]$ and $\phi_2[\psi] \Leftrightarrow \phi_2[\gamma]$. We show that the equivalence $\phi[\psi] \Leftrightarrow \phi[\gamma]$ holds by performing the following transformations.

$\upsilon(\phi[\psi]) = \upsilon(\phi_1[\psi] \vee \phi_2[\psi]) = \mathsf{T}$ if $\upsilon(\phi_1[\psi]) = \mathsf{T}$ or if $\upsilon(\phi_2[\psi]) = \mathsf{T}$.

Since $\phi_1[\psi] \Leftrightarrow \phi_1[\gamma]$ and $\phi_2[\psi] \Leftrightarrow \phi_2[\gamma]$, $\upsilon(\phi_1[\psi] \vee \phi_2[\psi]) = \mathsf{T}$ if
$\upsilon(\phi_1[\gamma]) = \mathsf{T}$ or if $\upsilon(\phi_2[\gamma]) = \mathsf{T}$.

This is equivalent to $\upsilon(\phi_1[\gamma] \vee \phi_2[\gamma])$ that again is equivalent to $\upsilon(\phi[\gamma])$.

(2) *Existential Quantification*

Assume that $\phi'[\psi] \Leftrightarrow \phi'[\gamma]$ holds by induction hypothesis. From the semantics of $\Leftrightarrow$ and $\exists$, it follows immediately that $\exists x\, \phi'[\psi] \Leftrightarrow \exists x\, \phi'[\gamma]$. □

**Theorem 2.2.3 (Monotonic Replacement Theorem)**
Let $\phi$ be a QBF and let $x$ be a variable, which is only positive or only negative in $\phi$. Let further $\psi$ and $\gamma$ be arbitrary QBFs.

1. If $\upsilon(\psi \rightarrow \gamma) = \mathsf{T}$ then $\upsilon(\phi[x/\psi] \rightarrow \phi[x/\gamma]) = \mathsf{T}$ if all occurrences of $x$ are positive in $\phi$, and

2. if $\upsilon(\psi \rightarrow \gamma) = \mathsf{T}$ then $\upsilon(\phi[x/\gamma] \rightarrow \phi[x/\psi]) = \mathsf{T}$ if all occurrences of $x$ are negative in $\phi$.

**Proof.** We proof the Monotonic Replacement Theorem by induction on the complexity $\mathsf{k}(\phi)$ of $\phi$ under the assumption that all occurrences of the variable $x$ are positive in $\phi$. The proof for 2. proceeds analogously.

**Basis Step.** In the first step, we show that the Monotonic Replacement Theorem holds for atomic formulas and truth constants. The $\mathsf{k}(\phi) = 0$.

(1) Let $\phi = \top$ (resp. $\phi = \bot$). Then $\phi[x/\psi] = \top$ (resp. $\phi[x/\psi] = \bot$) and also $\phi[x/\gamma] = \top$ (resp. $\phi[x/\gamma] = \bot$). Therefore, $\upsilon(\phi[x/\psi]) = \upsilon(\phi[x/\gamma])$ holds and consequently, the implication $\upsilon(\phi[x/\psi] \rightarrow \phi[x/\gamma])$ holds too.

(2) Suppose $\phi \in \mathcal{P}$. Then either $\phi = x$ or $\phi \neq x$. If $\phi = x$ then $\phi[x/\psi] = \psi$ and $\phi[x/\gamma] = \gamma$. Since $\upsilon(\psi \rightarrow \gamma) = \mathsf{T}$ by assumption, $\upsilon(\phi[x/\psi] \rightarrow \phi[x/\gamma]) = \mathsf{T}$ trivially holds. If $\phi \neq x$ then $\upsilon(\phi[x/\psi] \rightarrow \phi[x/\psi]) = \mathsf{T}$ because $\upsilon(\phi \rightarrow \phi) = \mathsf{T}$.

**Induction Hypothesis.** Let $n \geq 0$ be and assume that the Monotonic Replacement Theorem holds for every QBF with $\phi$ with $\mathsf{k}(\phi) \leq n$.

**Induction Step.** We have to show that the Monotonic Replacement Theorem also holds for every QBF with $\mathsf{k}(\phi) = n+1$. To prove the Monotonic Replacement Theorem we have to distinguish between four cases depending on the main connective of the formula $\phi$.

(1) *Disjunction*

Let $\phi = \phi_1 \vee \phi_2$. By induction hypothesis, it holds that $\upsilon(\phi_1[x/\psi] \rightarrow \phi_1[x/\gamma]) = \mathsf{T}$ and $\upsilon(\phi_2[x/\psi] \rightarrow \phi_2[x/\gamma]) = \mathsf{T}$. We must show that $\upsilon(\phi[x/\psi] \rightarrow \phi[x/\gamma]) = \mathsf{T}$, i.e., the following implication must hold: $\upsilon((\phi_1[x/\psi] \vee \phi_2[x/\psi]) \rightarrow (\phi_1[x/\gamma] \vee \phi_2[x/\gamma])) = \mathsf{T}$. We can rewrite this as $\upsilon(\neg(\phi_1[x/\psi] \vee (\phi_2[x/\psi]) \vee (\phi_1[x/\gamma] \vee \phi_2[x/\gamma])) = \mathsf{T}$.

By the application of De Morgan's laws and the associative and the commutative law, we obtain $\upsilon((\neg\phi_1[x/\psi] \vee \phi_1[x/\gamma] \vee \phi_2[x/\gamma]) \wedge (\neg\phi_2[x/\psi] \vee \phi_1[x/\gamma] \vee \phi_2[x/\gamma])) = \mathsf{T}$ which can be rewritten as $\upsilon(((\phi_1[x/\psi] \rightarrow \phi_1[x/\gamma]) \vee \phi_2[x/\gamma]) \wedge ((\phi_2[x/\psi] \rightarrow \phi_2[x/\gamma]) \vee \phi_1[x/\gamma])) = \mathsf{T}$. This holds because of the induction hypothesis.

(2) *Conjunction*

Let $\phi = \phi_1 \wedge \phi_2$. By induction hypothesis, it holds that $\upsilon(\phi_1[x/\psi] \rightarrow \phi_1[x/\gamma]) = \mathsf{T}$ and $\upsilon(\phi_2[x/\psi] \rightarrow \phi_2[x/\gamma]) = \mathsf{T}$. We have to show that $\upsilon(\phi[x/\psi] \rightarrow \phi[x/\gamma]) = \mathsf{T}$, i.e., the

following implication must hold: $\upsilon((\phi_1[x/\psi] \wedge \phi_2[x/\psi]) \rightarrow (\phi_1[x/\gamma] \wedge \phi_2[x/\gamma])) = \mathsf{T}$. We can rewrite this as $\upsilon(\neg(\phi_1[x/\psi] \wedge \phi_2[x/\psi]) \vee (\phi_1[x/\gamma] \wedge \phi_2[x/\gamma])) = \mathsf{T}$. This is the same as $\upsilon(((\phi_1[x/\psi] \rightarrow \phi_1[x/\gamma]) \vee \phi_2[x/\gamma]) \wedge ((\phi_2[x/\psi] \rightarrow \phi_2[x/\gamma]) \vee \phi_1[x/\gamma])) = \mathsf{T}$.
This holds because of the induction hypothesis.

(3) *Negation*

Let $\phi = \neg\phi'$. Then we have to show that the implication $\upsilon(\neg\phi'[x/\psi] \rightarrow \neg\phi'[x/\gamma])$ holds under the assumption that $\psi \rightarrow \gamma$. Now we distinguish between two cases:

(a) $\phi'$ is a variable or a truth constant. It can be shown similarly to the basis step that the Monotonic Replacement Theorem holds.

(b) Otherwise, we can shift the negation inside the formula by the application of De Morgan's laws. Now the main connective is either a conjunction, a disjunction, or a quantifier and the Monotonic Replacement Theorem holds as previously proven.

(4) *Quantification*

Assume that $\upsilon(\phi'[x/\psi] \rightarrow \phi'[x/\gamma]) = \mathsf{T}$ by induction hypothesis. Therefore, if it holds that $\upsilon(\phi'[x/\psi][y/\bot]) = \mathsf{T}$ and/or $\upsilon(\phi'[x/\psi][y/\top]) = \mathsf{T}$, also $\upsilon(\phi'[x/\gamma][y/\bot]) = \mathsf{T}$ and $\upsilon(\phi'[x/\gamma][y/\top]) = \mathsf{T}$. Therefore, $\upsilon(\mathsf{Q}y\,\phi'[x/\psi] \rightarrow \mathsf{Q}y\,\phi'[x/\gamma]) = \mathsf{T}$ with $\mathsf{Q} \in \{\forall, \exists\}$ (to obtain a QBF in standard form a suitable renaming has to be performed). □

**Theorem 2.2.4 (Pure Literal Elimination)**
Let the literal $l = x$ (resp. $l = \neg x$) be pure in the QBF $\phi$. Then

$$\phi \Leftrightarrow \begin{cases} \phi[x/\top] \text{ (resp. } \phi[x/\bot]) & \text{if } x \text{ is existential;} \\ \phi[x/\bot] \text{ (resp. } \phi[x/\top]) & \text{if } x \text{ is universal.} \end{cases}$$

**Proof.** We show that $\phi \Leftrightarrow \phi[x/\top]$ under the assumption that all occurrences of $x$ are positive in $\phi$ and that $x$ is an existential variable. Since $\phi$ is in standard form and due to the Equivalence Replacement Theorem, it suffices to prove that the subformula $\exists x\,\phi'$ of $\phi$ is equivalent to $\phi'[x/\top]$. We show that $\exists x\,\phi' \rightarrow \phi'[x/\top]$ and $\phi'[x/\top] \rightarrow \exists x\,\phi'$ hold.

(1) $\exists x\,\phi'[x/x] \rightarrow \exists x\,\phi'[x/\top]$ is valid due to the Monotonic Replacement Theorem (since, for any $\psi$, the formula $\psi \rightarrow \top$ is a tautology). The quantifier on the right-hand side of the implication can be omitted because all occurrences of $x$ are removed. Therefore, $\exists x\,\phi' \rightarrow \phi'[x/\top]$ holds.

(2) $(\phi'[x/\top] \rightarrow \exists x\,\phi') \Leftrightarrow (\phi'[x/\top] \rightarrow (\phi'[x/\top] \vee \phi'[x/\bot]))$, which is valid. □

**Theorem 2.2.5 (Local Unit Literal Elimination)**
Let the literal $l = x$ (resp. $l = \neg x$) and let $\psi$ be a subformula of the form $(l \circ \psi')$ in $\phi$. Then

$$\psi \Leftrightarrow \begin{cases} x \wedge \psi'[x/\top] & \text{if } \psi = x \wedge \psi'; \\ x \vee \psi'[x/\bot] & \text{if } \psi = x \vee \psi'; \\ \neg x \wedge \psi'[x/\bot] & \text{if } \psi = \neg x \wedge \psi'; \\ \neg x \vee \psi'[x/\top] & \text{if } \psi = \neg x \vee \psi'. \end{cases}$$

Due to the Equivalence Replacement Theorem $\psi$ can be substituted by the corresponding formula.

**Proof.** We show that, regardless whether we assign $x$ to $\bot$ or to $\top$, $x \wedge \psi' \Leftrightarrow x \wedge \psi'[x/\top]$ holds under this assignment. The other equivalences can be shown in a similar manner. We distinguish between the following two cases:

Case 1: $x$ is set to true.

Then $(x \wedge \psi')[x/\top] \Leftrightarrow (\top \wedge \psi'[x/\top]) \Leftrightarrow \top \wedge (\psi'[x/\top])[x/\top]$.

Case 2: $x$ is set to false.

Then $(x \wedge \psi')[x/\bot] \Leftrightarrow \bot \Leftrightarrow (\bot \wedge \psi'[x/\top]) \Leftrightarrow \bot \wedge (\psi'[x/\top])[x/\bot]$. □

**Theorem 2.2.6 (Global Unit Literal Elimination)**
Let the literal $l = x$ (resp. $l = \neg x$) be global unit in the QBF $\phi$. Then

$$\phi \Leftrightarrow \begin{cases} \phi[x/\top] \text{ (resp. } \phi[x/\bot]) & \text{if } x \text{ is existential;} \\ \phi[x/\bot] \text{ (resp. } \phi[x/\top]) & \text{if } x \text{ is universal.} \end{cases}$$

**Proof.** By the application of Theorem 2.2.4 and 2.2.5. □

## 2.3 Complexity

Quantified Boolean formulas are strongly connected to the *theory of computational complexity*. Computational complexity is the area of computer science that researches the reasons why some problems are so hard to solve by computers [82]. Complexity theory is the branch, where problems are not considered as entities which have to be solved but where problems become objects worth to be studied by themselves. Complexity theory provides a classification for problems which indicates the worst-case complexity for the solution of a particular problem instance.

The problem, we are interested in, is the decision problem for QBFs. It can be formulated as follows.

**Definition 2.3.1 (Decision Problem for QBFs (QSAT))**
Let $\phi$ be a QBF of the form

$$\mathsf{Q}_1 X_1 \ldots \mathsf{Q}_n X_n \psi$$

with $\mathsf{Q}_i \in \{\forall, \exists\}$ and $\psi$ being purely propositional. The *decision problem* for QBF is testing whether $\phi$ is satisfiable or not.

Note that the decision problem for QBFs is usually defined for formulas in prenex normal form because there is a close relation between the number of quantifier alternations in the prefix of the formula and the complexity classification of the corresponding decision problem.

To discuss the complexity of the decision problem, we have to introduce a machine model to provide "a computer" where we can run the implementation of our algorithm. A very prominent model for the simulation of arbitrary algorithms is the *Turing Machine*. We will provide a short, informal introduction sufficient for our purposes. For further information, we refer the reader to [82].

A *deterministic Turing machine* (DTM) consists of a tape with an infinite number of memory fields controlled by a finite automata. A read-write head can read symbols from the tape, write symbols on the tape from the finite alphabet $\Sigma$ and it can move one field to the left or to the right. The alphabet $\Sigma$ always contains one special symbol: the blank symbol $b$. Further, a Turing machine is specified by a set of states $S$. $S$ contains the special states $s_I$, the initial state, and the two end states $s_A$ and $s_R$ (the accepting state and the rejecting state). The actual program of a Turing machine is given by the state transition function $\delta$ which maps $(S \backslash \{s_A, s_R\} \times \Sigma)$ to $(S \times \Sigma \times \{\leftarrow, \rightarrow\})$. $\leftarrow$ and $\rightarrow$ indicate the direction of the next move.

The input of a program — a string consisting of symbols from $\Sigma$ — is initially written on the tape (one symbol per field). The other fields are initialised with the blank symbol. The execution of the program starts in the initial state $s_I$ and the head is positioned at the beginning of the input string. The calculation is performed stepwise according to $\delta$ until either no application of $\delta$ is possible or one of the end states is reached. E.g., if $\delta(s, 1) = (s', 0, \leftarrow)$ then the state of the finite automata changes from $s$ to $s'$, the "1" in the current field is overwritten by "0" and the read-write head moves one position to the left. If the program terminates with the state $s_A$ then the input string is said to be accepted, if the program terminates with the state $s_R$ then the the input string is said to be rejected.

The Turing machine we have presented is referred to as a *deterministic Turing machine* because this machine changes to exactly one state in each computation step. A probably more powerful machine model is the *nondeterministic Turing machine* (NDTM). A NDTM is defined similar to a DTM except that it does not have a single, uniquely defined next state. As far as it is known, DTMs can simulate NDTMs only with an exponential loss of efficiency — otherwise the prominent problem P = NP would be answered positively.

With NDTMs we are able to characterise the very famous class of NP, the set of decision problems solvable in polynomial time by a NDTM. NDTMs are also referred to as guess-and-check Turing machines. The non-deterministic part is used to guess a solution of the problem (like a variable assignment if we want to solve the SAT problem) and the deterministic part is used to check the correctness of the guessed solution.

**Definition 2.3.2 (NP)**
The class NP includes all decision problems for which all its instances can be solved by a NDTM in polynomial time with respect to the instance size.

**Definition 2.3.3 (Many-one Reducibility)**
Let $\Sigma$ be an alphabet and $\Sigma^*$ be the set of all strings over $\Sigma$. Given two languages $L_1$ and $L_2$, $L_1$ is called *polynomial time many-one reducible* to $L_2$ ($L_1 \leq_m^P L_2$) if there exists a function $f : \Sigma^* \to \Sigma^*$ such that $f(w)$ is computable in polynomial time with respect to the length $|w|$ of $w$ and $w \in L_1$ iff $f(w) \in L_2$ for all $w \in \Sigma^*$.

**Definition 2.3.4 (C-hardness, C-completeness)**

- Given a class $C$, a language $A$ is C-*hard* if $B \leq_m^P A$, for every $B \in$ C.
- Given a class $C$, a language $A$ is C-*complete* if $A$ is C-*hard* and $A \in$ C.

A very famous statement in complexity theory is the following: The *satisfiability problem* for propositional logic (SAT) is NP-complete [25].

We are not concerned with the decision problem for propositional logic but with the problem for quantified Boolean formulas. It is assumed that this problems is located outside NP, so we need to introduce complexity classes for probably harder problems.

The class PSPACE includes all problems calculated by a DTM using only polynomial space with respect to the size of the input. PSPACE includes naturally all problems which are solvable in polynomial time. The central problem shown to be PSPACE-complete is the satisfiability problem for QBFs [71].

PSPACE can be divided into several complexity classes itself. These classes span the so-called *polynomial hierarchy*. To set up the polynomial hierarchy, we have to extend the Turing Machines by the facilities of an oracle. The usage of an oracle corresponds to the call of a subroutine which is of unit cost and where we are interested neither in its time nor in its space usage. The classes $\mathrm{P}^C$ and $\mathrm{NP}^C$ refer to the classes of decision problems that can be solved in polynomial time by a deterministic (in the case of P) and nondeterministic (in the case of NP) Turing Machine using an oracle for problems in C where C is an arbitrary complexity class.

**Definition 2.3.5 (Polynomial Hierarchy [71])**
The *polynomial hierarchy* consists of classes $\Delta_k^p$, $\Sigma_k^p$, and $\Pi_k^p$, which are defined using the notion of oracles as follows:

$$\Delta_0^p = \Sigma_0^p = \Pi_0^p = \mathrm{P},$$

and, for all $k \geq 0$,

$$\Delta_{k+1}^p = \mathrm{P}^{\Sigma_k^p}, \quad \Sigma_{k+1}^p = \mathrm{NP}^{\Sigma_k^p}, \quad \Pi_{k+1}^p = \text{co-}\Sigma_{k+1}^p.$$

**Definition 2.3.6 (Levels of the Polynomial Hierarchy)**
A problem is located *at the kth level of the polynomial hierarchy*, if it is contained in $\Delta_{k+1}^p$ and if it is either $\Sigma_k^p$-hard or $\Pi_k^p$-hard.

Based on those classes, further classes can be derived. For instance,

$$\mathrm{D}_{k+1}^P = \{L_1 \times L_2 \mid L_1 \in \Sigma_k^p, L_2 \in \Pi_k^p\}$$

can be seen as the "conjunction" of $\Sigma_k^p$ and $\Pi_k^p$. The class $\mathrm{D}_2^P$ is also known as $\mathrm{D}^P$.

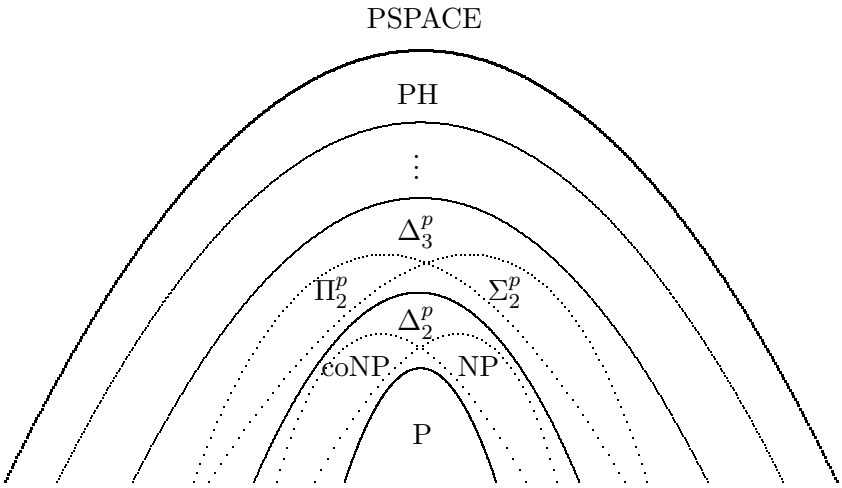



Figure 2.3: The polynomial hierarchy.

To conclude this chapter, we present two results on the complexity of QBFs due to Meyer and Stockmeyer [70] and Wrathall [94]: the complexity characterisation of QBFs. As we will see, QBFs span the whole polynomial hierarchy where many important problems can also be found.

**Theorem 2.3.1 (The SAT Problem for QBFs)**

1. The problem of satisfiability checking of a QBF of the form

$$\exists X_1 \forall X_2 \ldots \mathsf{Q}_k X_k \phi$$

is complete for $\Sigma_k^p$ with $\mathsf{Q}_k = \exists$ if $k$ is odd and $\mathsf{Q}_k = \forall$ if $k$ is even and $\phi$ is purely propositional.

2. The problem of satisfiability checking of a QBF of the form

$$\forall X_1 \exists X_2 \ldots \mathsf{Q}_k X_k \phi$$

is complete for $\Pi_k^p$ with $\mathsf{Q}_k = \forall$ if $k$ is odd and $\mathsf{Q}_k = \exists$ if $k$ is even and $\phi$ is purely propositional.

As Theorem 2.3.1 indicates, QBFs span the whole polynomial hierarchy. This makes QBFs a suitable tool for proving the complexity of many formalisms by translating them from and to QBFs. Therefore QBFs can serve as a powerful language to encode many problems. One generic solver — the QBF solver — can then be applied to solve formulas from very different fields. This is why QBFs became so popular during the last few years and why there is an enormous claim for efficient solvers.

# Chapter 3

# Decision Procedures

This chapter provides an overview of various ways to solve the QSAT problem. The basic concepts and methods used during the evaluation process will be reviewed including the following four decision procedures:

- the resolution calculus for QBFs;
- the decision procedure by Davis, Putnam, Loveland, and Lodgeman (`DPLL`);
- binary decision diagrams (BDDs);
- the sequent calculus for QBFs.

All of the decision methods listed above are extensions of procedures for propositional logic. In practice, only the first three are implemented in the state-of-the-art solvers presented at the end of this chapter. When discussing those algorithms, we will notice that resolution and `DPLL` do not work on arbitrary QBFs. The QBFs are expected to be transformed to a certain normal form, i.e., some assumptions about the structure of the formula are made. The impact of this transformation will be considered in the next chapter.

## 3.1 Resolution for QBFs

One of the most prominent and most widely used inference systems for first-order theorem proving is Robinson's *resolution calculus* [85]. A very appealing aspect of this decision method is its simplicity. Resolution can not only be applied to predicate logic, but a simpler version can also be used to evaluate formulas of propositional logic.

The extension of simple propositional logic's resolution to QBFs is called *Q-resolution* and was introduced in [59]. The main difference between ordinary resolution and Q-resolution is that universally quantified variables can be dropped in certain cases. It only works for QBFs in prenex conjunctive normal form, so we deal with formulas of the format

$$\mathsf{Q}_1 X_1 \mathsf{Q}_2 X_2 \ldots \mathsf{Q}_n X_n \bigwedge_{i=0}^{m} (\phi_i),$$

where the $\phi_i$ are clauses which contain literals from $X_1 \cup \cdots \cup X_n$. We assume that no clause is tautological (i.e., it does not contain $l$ and $\overline{l}$ at the same time), and we assume that no clause contains multiple occurrences of one literal. A literal $l_1$ is said to *precede* a literal $l_2$ if $\mathsf{var}(l_1) \in X_i$, $\mathsf{var}(l_2) \in X_j$ and $i < j$. The clauses $\phi_i$ are of the form $(l_1 \vee \cdots \vee l_{n_i})$. In the following, we consider clauses as *sets* of literals.

A clause is called a *pure* $\forall$*-clause* if it is a non-tautological clause, which contains only universally quantified literals. The *empty* clause $\emptyset$ is a pure $\forall$-clause.

**Definition 3.1.1 ($\forall$-Reduct)**
Let $\phi$ be a QBF in PCNF. The $\forall$*-reduct* of a clause $\psi$ in $\phi$ is the clause $\psi'$ resulting from the removal of any universal literal in $\psi$, which does not precede an existential literal of $\psi$ in the prefix of $\phi$.

**Definition 3.1.2 (Q-Resolution)**
The resolution operation for QBF — called *Q-resolution* — is defined as follows.

1. Every clause is replaced by its $\forall$-reduct. All clauses, which contain only universally quantified variables, are replaced by the empty clause.

2. Let $\phi_1$ be a clause, which contains an existentially quantified literal $l$ and let $\phi_2$ be a clause with the complementary literal $\overline{l}$. Furthermore, let $\phi_1'$ and $\phi_2'$ be the $\forall$-reduct of $\phi_1$ and $\phi_2$.

   The *Q-resolvent* $\phi_r$ is obtained from $\phi_1' \cup \phi_2'$ as follows:

   (a) $\phi_r' = (\phi_1' \cup \phi_2') \backslash \{l, \overline{l}\}$.

   (b) There exists no resolvent if $\phi_r'$ contains complementary literals. Otherwise, the Q-resolvent $\phi_r$ is the $\forall$-reduct of $\phi_r'$.

**Theorem 3.1.1 (Soundness and Completeness of Q-resolution)**

1. A QBF is false if the empty clause can be derived by repeated application of the resolution operator.

2. If a QBF is true then the empty clause cannot be derived.

The proof can be found in [59].

Although Q-resolution is theoretically complete, it is almost never used, because the single underlying decision method in an implementation as the number of clauses can grow exponentially. Usually, it is integrated to strengthen other decision methods, for example in his solver semprop, Letz [68] uses Q-resolution to calculate lemmas.

## 3.2 The DPLL Decision Method

The DPLL decision procedure for propositional logic was introduced at the beginning of the 1960s by Davis, Putnam, Logeman, and Loveland [26, 27, 28], and for a long time DPLL ranks among the most famous methods in the field of automatic theorem proving.

The original intention of M. Davis and P. Putnam was to provide a theorem proving technique suitable for satisfiability testing of propositional logic in order to develop an efficient proof procedure for first-order logic [27, 28]. The first practical implementation showed to be very memory consuming, and so M. Davis, G. Logeman, and D. Loveland proposed a modified variant in [26]. Today, `DPLL` is still among the fastest and most widely used methods for solving propositional formulas, and `DPLL` is implemented in many systems.

In the last few years, `DPLL` has been successfully adapted for QBFs, and it is used in many state-of-the-art solvers [41, 54, 68]. In the following, we present a version of `DPLL`, where we abstract from implementation details. We will also introduce some advanced pruning techniques. Without them, this method would not be practically applicable for bigger formulas. After discussing alternative approaches to solve a QBF, we will provide an overview of available solvers.

### 3.2.1 The Basic Decision Procedure

The `DPLL` decision procedure evaluates QBFs in prenex conjunctive normal form. As we will see in the next chapter, this is not a limitation, because every QBF can be transformed into PCNF. But we will also see that this transformation is bound up with some costs and trade-offs. This means that a preprocessing step is usually necessary to convert the input formula into an equivalent formula of the required format. At the moment, this preprocessing step is completely ignored in most implementations and all input formulas are assumed to be in PCNF (in fact, the language of QBFs is often introduced in such a manner that it contains exclusively formulas in PCNF). This distinguishes our solver qpro from most available implementations as we omit this preprocessing step by accepting formulas of nearly arbitrary structure. A trend in current QBF research is the attempt to reconstruct the structure of the original formula or to store the prefix as a tree, from which more information can be extracted. We focus on keeping and working with the original formula structure. Therefore, we can directly read off the information about the quantifier dependencies without performing any complicated analysis. Before we present the implementation of our solver (which is in fact also a variant of `DPLL`), it is necessary to obtain a rudimentary understanding of the "standard" `DPLL` decision procedure first.

Figure 3.1 shows the decision procedure in pseudo-code. We use a language similar to C. The language provides operators like `||` (logical or) and `&&` (logical and) to connect Boolean values. A comparison can be performed by the usage of `==`.

`DPLL`, also referred to as *splitting algorithm*, implicitly builds up the previously introduced *semantic tree* and realises a simple search-based backtracking algorithm. The simplicity of the algorithm was certainly critical to its success as it can almost be implemented as the pseudo-code proposes. Recursively, the structure tree is traversed and whenever a node containing a quantification is reached, the quantifier is eliminated by replacing the corresponding variable by $\bot$ and/or $\top$ depending on the type of the quantifier. Since we are dealing with closed formulas only, the formula evaluates to $\top$ or $\bot$ in any case.

The `simplify`-function performs — as the name indicates — simplifications on the formula. The truth constants are eliminated according to Theorem 2.2.1. Another op-

```
BOOLEAN dpll(QBF φ in PCNF) {

switch (simplify (φ)) {

  case ⊤:  return True;
  case ⊥:  return False;

  case (QXψ):  select x ∈ X;
      if Q = ∃ return (dpll(QX'ψ[x/⊥]) || dpll(QX'ψ[x/⊤]));
      if Q = ∀ return (dpll(QX'ψ[x/⊥]) && dpll(QX'ψ[x/⊤]));
  }
}
```

Figure 3.1: The DPLL decision procedure.

timisation consists of the removal of tautological clauses, i.e., clauses which contain two complementary literals $l$ and $\overline{l}$. During the solving process, tautological clauses can never be introduced. The size of the clauses can only decrease as no new literals are added at any time. So the removal of tautological clauses has to be applied only once.

To use the algorithm in practice, it is indispensable to implement the *unit propagation* and *pure literal elimination* rules based on Theorems 2.2.4 and 2.2.6. As we are dealing with QBFs in PCNF, we can define the term *unit clause*.

**Definition 3.2.1 (Unit Clause)**
Let $\phi$ be a QBF in PCNF. A clause $\psi$ of $\phi$ is called *unit* if it contains exactly one existentially quantified literal $l$ and the universal literals of $\psi$ do not precede any existential variable in the prefix of $\phi$.

The unit propagation rule and the pure elimination rule are as follows:

- *Unit Rule.* If a QBF $\phi$ in PCNF contains a unit clause $\psi$ with an existentially quantified atom $x$ (resp. $\neg x$) then $x$ can be replaced by $\top$ (resp. $\bot$) in $\phi$ immediately.
- *Pure Rule.* If a QBF $\phi$ contains a pure literal $x$ (resp. $\neg x$) then $x$ can be immediately replaced in $\phi$ by
  - $\top$ (resp. $\bot$) if $x$ is existentially quantified;
  - $\bot$ (resp. $\top$) if $x$ is universally quantified.

A refinement of the unit elimination rule is the removal of clauses, which contain only universally quantified literals.

- *Non-tautological All-Clause Rule.* A QBF $\phi$ in PCNF, which contains a non-tautological clause with only universally quantified literals, is unsatisfiable.

By every application of a variable substitution by $\top$ and $\bot$ and by the application of the `simplify`-function, the formula size decreases until it reduces to a single truth constant

if the QBF is closed. It is straight forward to see that the algorithm terminates but is exponential in time in the worst case and polynomial in space.

**Theorem 3.2.1 (Soundness, Completeness, Termination)**
`DPLL` is *sound*, *complete* and *terminating*, i.e., for every QBF $\phi$ in PCNF, `DPLL` returns

- "`True`" if $\upsilon(\phi) = \mathsf{T}$;
- "`False`" if $\upsilon(\phi) = \mathsf{F}$.

### 3.2.2 Improvements of `DPLL`

As we have seen, `DPLL` traverses the semantic tree in a depth-first fashion. Therefore, the necessary space is bounded polynomially by the size of the input formula. The solving time, however, is exponential in the worst case, because the number of variable assignments which has to be considered is exponential with respect to the number of variables occurring in the formula. Even for small formulas the worst case situation is not acceptable in practice. So optimisation techniques have to be integrated into the basic `split`-algorithm to prune the search space and to avoid the worst-case situation as much as possible. In the following, we will present some of the most important and successful techniques implemented in current state-of-the-art solvers.

#### Dependency-Directed Backtracking

As already discussed, `DPLL` chronologically traverses the semantic tree of a formula. Due to the semantics of the quantifiers, sometimes it is not necessary to build the second subtree of a node (e.g., if the variable of the node is existentially quantified and the first subproblem evaluates to true), and the procedure can backtrack to the node above. But what if the variable is existentially quantified (resp. universally quantified) and the first subproblem evaluates to false (resp. to true)? Is there a chance to omit the second branch without influencing the result? The answer is yes — the technique we need is *dependency-directed backtracking* (DDB).

DDB, also called *backjumping*, is for example described by Letz in [68]. The idea is very simple but nevertheless very effective. Letz distinguishes between two different kinds of dependency-directed backtracking: DDB *for false subproblems* and DDB *for true subproblems*.

Furthermore, Letz presents two versions of DDB for false subproblems, namely DDB *by labelling* and DDB *by relevance sets*. The first one causes almost no implementational overhead, but it does not prune the search space as much as the second one. When branching on a variable, say $x$, this variable is marked as irrelevant and assigned a truth constant, e.g., $\bot$. If during the search an assignment is found which sets the formula to true or to false, all variables which are responsible for the result are set to relevant. When returning to $x$ during backtracking it is checked if $x$ is irrelevant or not. If $x$ is still irrelevant then the second problem, where $x$ is set to $\top$ can be omitted in any case. Obviously, the less variables are marked as relevant the better.

Dependency-directed backtracking by relevance set can decrease the number of relevant variables if both subproblems at the branching point of a variable have to be considered. We will explain DDB in a more detailed manner later.

### The Use of Lemmas

*Lemma cashing* (also called nogood learning) is indispensable for solving certain unsatisfiable formulas (for examples, see [68]). The generation of the lemmas is achieved by the usage of the previously introduced Q-resolution. Since Q-resolution is sound, any resolvent of a QBF $\phi$ can be added to the matrix of $\phi$ without changing $\phi$'s truth value. The Q-resolvents are the lemmas and by the integration of them into the original formula, the proof of the QBF might be shortened.

In the following we will briefly describe how `DPLL` can be extended by lemma cashing.

**Definition 3.2.2 (`DPLL` with Lemmas)**
Let $N_i$ be a node in the semantic tree, which has evaluated to false. We associate $N_i$ with a clause as follows.

1. If $N_i$ is a leaf node, the associated clause is the $\forall$-reduct of one unsatisfiable clause.

2. If $N_i$ contains a variable, which is universally quantified, then it can be assumed that $N_i$ has exactly one false successor node, and we set the associated clause of $N_i$ equal to the associated clause of the successor node.

3. If $N_i$ contains a variable $x$, which is existentially quantified then its two successor nodes $N_i'$ and $N_i''$ must be false. Furthermore, let $C'$ be the clause associated with $N_i'$ and let $C''$ be the clause associated with $N_i''$. Then two cases can be distinguished.

   (a) $x \in C'$ and $\neg x \in C''$. Then the associated clause of $N_i$ is the Q-resolvent of $C'$ and $C''$ (if the resulting clause is non-tautological).

   (b) Otherwise, one of the clauses does not contain the respective literal — e.g., $\neg x \notin C''$. But this means that $x$ is irrelevant in $N_i''$ and therefore, we can set the associated clause of $N_i$ to $C'$.

It is important to note that the integration of lemmas is not as straightforward for QBFs as it is in the propositional case, because the resolvent lemma does not hold for QBFs containing universally quantified variables.

**Lemma 3.2.1 (Resolvent Lemma)**
Let $T$ be the semantic tree of a propositional formula $\phi$ in CNF. For any node $N$ of $T$, no associated clause is tautological.

As tautological lemmas are worthless, it is necessary to avoid the creation of them, so techniques to prevent them have to be necessarily included (for details see [68]).

### Model Cashing

Model cashing can be seen as the dual form of the employment of lemmas [68]. To perform model cashing, some technique has to be introduced, which corresponds to Q-resolution: *model resolution*.

**Definition 3.2.3 (Model Resolvent)**
Let $\phi$ be a QBF and let $\iota_{S_1}$ and $\iota_{S_2}$ be models of $\mathsf{psk}(\phi)$. Assume further that $S_1$ and $S_2$ contain two complementary literals — say $x$ with $x \in S_1$ and $\neg x \in S_2$. The *model resolvent* of $\iota_{S_1}$ and $\iota_{S_2}$ is $(S_1' \backslash \{\neg x\} \cup S_2' \backslash \{x\})$ where $S_1'$ (resp., $S_2'$) is obtained from $S_1$ (resp. $S_2$) by removing any existential literal, whose variable does not precede the variable of any universally quantified literal occurring in $S_1$ (resp., $S_2$) in the prefix.

**Lemma 3.2.2 (Model Resolvent Lemma)**
If a model resolvent $S$ of two models for a QBF $\phi$ is consistent then $\iota_S$ is also a model for $\mathsf{psk}(\phi)$.

The integration of model cashing into `DPLL` is similar to the integration of lemma cashing. At the leaves of the semantic tree, models are extracted. If a node contains a universally quantified variable, the minimal models of the two branches are combined by model resolution. If the model resolvent exists, it is added disjunctively to the formula. When a partial assignment satisfies the model resolvent then it *subsumes* the literal on the branch, and the subproblem is immediately solved.

### Trivial Truth and Trivial Falsity

In certain cases, the evaluation problem for QBFs can be reduced to the evaluation problem of propositional logic. The following lemmas provide two criterions.

**Lemma 3.2.3 (Trivial Truth)**
A QBF $\phi$ in PCNF with the matrix $\psi$ is true if the propositional formula $\psi'$ is satisfiable, where $\psi'$ is obtained from $\psi$ by simply deleting all universally quantified literals.

**Lemma 3.2.4 (Trivial Falsity)**
A QBF $\phi$ in PCNF is unsatisfiable if one of the two following conditions holds.

1. $\phi$ contains a non-tautological clause consisting only of universally quantified variables.
2. The propositional formula $\psi'$ consisting of the clauses, which contain only existentially quantified variables, is unsatisfiable.

## 3.3 Binary Decision Diagrams

An alternative approach to solve a QBF is the usage of *Binary Decision Diagrams* [19, 66]. BDDs are widely used in the model checking community but there are efforts to integrate

them into standard decision procedures. Before defining BDDs, we have to introduce the notion of Binary Decision Trees.

**Definition 3.3.1 (Binary Decision Tree)**
A *binary decision tree* is a tree such that the following holds.

- The internal nodes are labeled by variables.
- The leaves are labeled by $\mathsf{F}$ and $\mathsf{T}$.
- Every internal node has exactly two children. The edge to one node represents the assignment of the variable by $\mathsf{F}$ (and is therefore labeled by $\mathsf{F}$), whereas the other edge represents the assignment by $\mathsf{T}$ (and is therefore labeled by $\mathsf{T}$).
- Nodes on a path in the tree have always different labels.

At the first glance a Binary Decision Tree (BDT) looks like the previously introduced semantic tree. But in contrast to the semantic tree, which illustrates the steps performed by the splitting algorithm, the BDT represents the underlying data structure for those solvers.

Using BDTs directly would be extremely inefficient because of its memory consumption. Therefore, BDDs are introduced.

**Definition 3.3.2 (Binary Decision Diagram)**
A *binary decision diagram* (BDD) is a rooted dag with the same properties as the BDT plus the following two properties:

- for every node, its left and right subdag are non-isomorphic;
- every pair of subdags rooted at two different nodes are non-isomorphic.

There are implementations of QBFs based on BDDs and some extensions like OBDDS (Ordered BDDs) or ZBDDs (zero-supressed BDDS) [47], but we did not include them in our tests because either the solvers were not available or the memory consumption showed to be inherent in pretests.

## 3.4 The Sequent Calculus for QBFs

Gentzen's sequent calculus is one of the most prominent and elegant proof system for many logics including classical propositional and first-order logic. In this section, we develop a cut-free sequent calculus for proving QBFs [61]. In particular, we take a look at a variant called *Gqve*, which was introduced in [29].

**Definition 3.4.1 (Sequent, Antecedent, Succedent)**
A *sequent* is an ordered pair $\langle \Phi, \Psi \rangle$ of finite sets of formulas, written $\Phi \vdash \Psi$. $\Phi$ is called *antecedent* and $\Psi$ is referred to as *succedent*.

$$\phi, \Phi \vdash \Psi, \phi \quad Ax \qquad \bot, \Phi \vdash \Psi \quad \bot l \qquad \Phi \vdash \Psi, \top \quad \top r$$

Figure 3.2: Axioms $Ax$, $\top l$ and $\top r$ of *Gqve*.

By convention, $\vdash \Psi$ and $\Phi \vdash$ denote $\langle \emptyset, \Psi \rangle$ and $\langle \Phi, \emptyset \rangle$. $\Phi, \phi$ (resp. $\Psi, \psi$) abbreviates $\Phi \cup \{\phi\}$ (resp., $\Psi \cup \{\psi\}$).

**Definition 3.4.2 (Validity of Sequents)**
A sequent $\phi_1, \ldots, \phi_m \vdash \psi_1, \ldots, \psi_n$ is *valid* iff whenever $v(\phi_i) = \mathsf{T}$ for all $1 \leq i \leq m$ then there exists a $\psi_j$ with $v(\psi_j) = \mathsf{T}$ $(1 \leq \mathsf{j} \leq \mathsf{n})$.

Intuitively, Definition 3.4.2 semantically identifies a sequent $\top, \phi_1, \ldots, \phi_m \vdash \psi_1, \ldots, \psi_n, \bot$ with the implication $(\top \wedge \phi_1 \wedge \cdots \wedge \phi_m) \rightarrow (\psi_1 \vee \cdots \vee \psi_n \vee \bot)$. The empty sequent $\vdash$ is invalid. For techniqual reasons, we sometimes write the empty sequent as $\top \vdash \bot$.

**Definition 3.4.3 (Rules of the Sequent Calculus)**
A sequent calculus consists of three kinds of rules:

1. initial rules (axioms);
2. logical rules;
3. structural rules.

The axioms are summarised in Figure 3.2, the structural rules are shown in Figure 3.3 and the logical rules in Figure 3.4. The sequents above the line of a rule are called *premises*, the sequent below the line is called *conclusion*. Axioms do not have any premises and the line is omitted. Each logical rule introduces a connective or a quantifier either in the antecedent or in the succedent. The variable $c$ introduced in $\forall l, \exists r$ may be a truth constant or an arbitrary propositional variable, whereas $e$ of $\forall r, \exists l$ has to respect the eigenvariable condition, i.e., it does not occur in the conclusion of the rule and is neither $\top$ nor $\bot$. *Gqve* is a *cut-free* calculus, i.e., it does not include the *cut rule* shown in Figure 3.4. We used a set based formulation of the sequent calculus. In contrast to sequence based formulations, it is not necessary to state the structural rules explicitly.

When we read an instance of a logical rule from premise to conclusion, we call it an *inference*, if we read it the other way round (i.e., from conclusion to premise), we call it *reduction*.

**Definition 3.4.4 (Logical Complexity of a Sequent)**
Let S be a sequent of the form $\phi_1, \cdots, \phi_n \vdash \psi_1, \cdots, \psi_m$, then the *logical complexity* of S is defined by

$$\mathsf{k}(S) = \Sigma_{i=1}^{n} \mathsf{k}(\phi_i) + \Sigma_{j=1}^{m} \mathsf{k}(\psi_j).$$

A *derivation* in this calculus is a tree generated by the (bottom-up) application of the logical rules. The root of this tree is called endsequent. The leaves are sequents which contain only formulas with a complexity of zero.

$$\frac{\Phi \vdash \Psi}{\phi, \Phi \vdash \Psi} Wl \qquad \frac{\Phi \vdash \Psi}{\Phi \vdash \Psi, \phi} Wr$$

$$\frac{\phi, \phi, \Phi \vdash \Psi}{\phi, \Phi \vdash \Psi} Cl \qquad \frac{\Phi \vdash \Psi, \phi, \phi}{\Phi \vdash \Psi, \phi} Cr$$

$$\frac{\Phi_1, \phi, \psi, \Phi_2 \vdash \Psi}{\Phi_1, \psi, \phi, \Phi_2 \vdash \Psi} El \qquad \frac{\Phi \vdash \Psi_1, \phi, \psi, \Psi_2}{\Phi \vdash \Psi_1, \psi, \phi, \Psi_2} Er$$

Figure 3.3: Structural rules for *Gqve*.

$$\frac{\Phi \vdash \Psi, \phi}{\neg\phi, \Phi \vdash \Psi} \neg l \qquad \frac{\phi, \Phi \vdash \Psi}{\Phi \vdash \Psi, \neg\phi} \neg r$$

$$\frac{\phi, \psi, \Phi \vdash \Psi}{\phi \wedge \psi, \Phi \vdash \Psi} \wedge l \qquad \frac{\Phi \vdash \Psi, \phi \quad \Phi \vdash \Psi, \psi}{\Phi \vdash \Psi, \phi \wedge \psi,} \wedge r$$

$$\frac{\phi, \Phi \vdash \Psi \quad \psi, \Phi \vdash \Psi}{\phi \vee \psi, \Phi \vdash \Psi} \vee l \qquad \frac{\Phi \vdash \Psi, \phi, \psi}{\Phi \vdash \Psi, \phi \vee \psi} \vee r$$

$$\frac{\Phi \vdash \Psi, \phi \quad \psi, \Phi \vdash \Psi}{\phi \rightarrow \psi, \Phi \vdash \Psi} \rightarrow l \qquad \frac{\phi, \Phi \vdash \Psi, \psi}{\Phi \vdash \Psi, \phi \rightarrow \psi} \rightarrow r$$

$$\frac{\phi[x/c], \phi, \Phi \vdash \Psi}{\forall x\phi, \Phi \vdash \Psi} \forall l \qquad \frac{\Phi \vdash \Psi, \phi[x/e]}{\Phi \vdash \Psi, \forall x\phi} \forall r$$

$$\frac{\phi[x/e], \Phi \vdash \Psi}{\exists x\phi, \Phi \vdash \Psi} \exists l \qquad \frac{\Phi \vdash \Psi, \phi, \phi[x/c]}{\Phi \vdash \Psi, \exists x\phi} \exists r$$

Figure 3.4: Logical rules for *Gqve*.

$$\frac{\Phi \vdash \phi, \Psi \quad \phi, \Phi \vdash \Psi}{\Phi \vdash \Psi} \text{cut}$$

Figure 3.5: The cut rule.

A *proof* is a derivation, whose leaves are labelled with axioms. A proof of the endsequent $\vdash \phi$ can be constructed iff the QBF $\phi$ is valid.

Note that the axioms $\top l$ and $\bot r$ can be faithfully omitted, if we construct a proof for $\top \vdash \bot, \phi$ instead of $\vdash \phi$ to show a QBF $\phi$'s validity since $\neg\top \vee \bot \vee \phi$ is logical equivalent to $\phi$.

Based on this foundations we will present a more compact representation for proofs by eliminating certain redundancies. This method is known as the matrix characterisation of logical validity.

## 3.5 An Overview of State-of-the-Art Solvers

### 3.5.1 QuBE−BJ

The solver QuBE[1] [54, 52] developed by Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella is also based on DPLL. It is implemented on top of SIM [50], a library for SAT solving, in the programming language C.

At the moment, three versions of QuBE are publicly available:

1. QuBE−BT which implements just chronological backtracking;
2. QuBE−BJ which implements dependency-directed backtracking;
3. QuBE−REL which implements learning.

We included QuBE−BJ in our tests, because QuBE−BJ uses an algorithm similar to the one implemented in our solver, and because in previous tests, it proved to be the most stable solver of the three versions. Updated and extended versions QuBE have been presented in the last two years, but unfortunately they were not available to us.

### 3.5.2 quantor

The solver quantor[2] [16] by Armin Biere implements an expansion-based decision procedure where existentially quantified variables are resolved and universally quantified formulas are removed by expansion until the formula becomes propositional and can be solved by a SAT solver. At the moment, quantor includes the SAT Solver BooleForce[3] as backend.

Basically, quantor does the following:

1. **Eliminate the innermost existentially quantified variables by Q-resolution.** To avoid a huge increase in space consumption, the resource usage is carefully monitored and only the cheapest variable is eliminated such that the size of the resulting formula does not inherently increase.

[1] http://www.mrg.dist.unige.it/ qube/
[2] http://fmv.jku.at/quantor/
[3] http://fmv.jku.at/booleforce/

2. **Eliminate universally quantified variables by expansion.** To expand a universally quantified variable $x$ from the quantifier block $\forall X_{n-1}$ it is necessary to generate a copy of the last existential quantifier block $\exists X_n$. The resulting formula is of the following format:

$$\mathsf{Q}_1 X_1 \mathsf{Q}_2 X_2 \ldots \forall (X_{n-1} \backslash \{x\}) \exists (X_n \cup X'_n)(\phi[x/\top] \wedge \phi'[x/\bot] \wedge \psi),$$

   where $\phi[x/\top]$ denotes the result of substituting $x$ by $\top$ in $\phi$. Likewise, $\phi[x/\bot]$ denotes the result of replacing $x$ by $\bot$ in $\phi'$, a copy of $\phi$ where the variables of the literals quantified in $X_n$ are replaced by the variables from $X'_n$. All the clauses in $\psi$ do not contain variables from $X_n$.

3. **Repeat 1. and 2. until the formula is propositional and apply a SAT solver.**

Further pruning techniques like the detection of equivalences and subsumption are implemented. The author also realized "that there are situations in which a linear quantifier prefix is not optimal and the basic expansion step as described above copies to many clauses". In this context the notion of *locally connected variables* is introduced. A variable is called locally connected to another variable if both variables occur in the same clause. Then the relation *connected* is defined as the transitive closure of locally connected, ignoring variables which are left to the expanded variable in the prefix and all other universally quantified variables in the same scope. Then only these clauses are copied which contain a variable which is connected to the variable which will be expanded.

### 3.5.3 semprop

The solver semprop[4][68] developed by Reinhold Letz is also based on DPLL, i.e., a splitting algorithm combined with the unit and pure rules. Further it integrates the following features:

1. dependency-directed backtracking,
2. model and lemma caching, and
3. the sign abstraction method.

### 3.5.4 sKizzo

The solver sKizzo[5] [10, 11, 12, 13] by Marco Bendetti integrates multiple reasoning techniques: classical resolution-based reasoning, structure reconstruction, propositional skolemisation, BDDs, symbolic and search-based decision procedures. The evaluation process consists of six steps.

[4] http://www4.informatik.tu-muenchen.de/~letz/semprop/
[5] http://skizzo.info/

1. **Normalisation of the PCNF input formula.**
   Simplification rules such as unit an pure propagation are applied as extensively as possible.

2. **Extraction of a tree-shaped syntactic structure.**
   In this step, some of the information lost by the normal form transformation is reconstructed by the creation of a quantifier tree as the PCNF structure allows.

3. **Application of the Skolem theorem.**
   By this transformation, a universal but satisfiability-equivalent formula is obtained. Therefore, function symbols are introduced. Since the Skolem function symbols are not conformant with the language of QBFs, they must be eliminated. This is done by the expansion of their propositional meaning. Unfortunately, this results in an exponential blow-up which is dealt with a compact symbolic representation. The QBF is translated into an equivalent first-order logic formula by the introduction of a predicate symbol.

   Then the Skolem theorem for first-order logic is applied such that all existential quantifiers are eliminated. This formula is translated into an equivalent SAT instance.

4. **Solving.**
   After Step 3 the formula instance is completely propositional. A set of incomplete inference rules is applied.

5. **Devision of the problem into smaller subproblem.**
   Now the new problems can solved by the means of Step 4 or a SAT solver is used.

6. **Application of the SAT solver.**

### 3.5.5 Qubos

The solver Qubos was one of the first and quite efficiently working QBF solvers which dealt with formulas of arbitrary structure.

Qubos follows a different approach than ours.

The basic idea behind Qubos is to integrate some kind of normalisation using miniscoping with selective quantifier expansion and simplification. Qubos works as follows.

1. **Calculate the weight of the quantifiers.**

   Here it is determined whether the average quantifier weight is smaller for $\forall$ or $\exists$. If the weight is small for universally quantified variables then they are eliminated and a SAT solver is used to solve the SAT problem of the remaining propositional formula. Otherwise, the existentially quantified variables are deleted and the remaining propositional formula's validity has to be checked.

2. **Apply transformations until there is only one kind of quantifiers.**

   To eliminate either all universally or all existentially quantified variables, the following techniques are applied iteratively.

   - Push the quantifiers inside the formula as far as possible (miniscoping).
   - Expand the variables to eliminate by replacing them by $\top$ and $\bot$.
   - Apply semantic simplification rules.
   - Eliminate pure and unit literals.

3. **Apply a SAT solver.**

On first sight, Qubos performs very well. It would have been very interesting to compare Qubos to our solver. But tests showed that Qubos is not sound on certain (quite trivial) instances and therefore, we exclude this solver from our solver comparison.

## 3.6 Summary

To solve the satisfiability problem of QBFs, a variety of decision procedures has been developed. Some of them, like the sequent calculus, are only of theoretical interest, whereas others are implemented in practical systems. The algorithm of Davis, Putnam, Loveland, and Logeman is a simple search-based algorithm directly implementing the semantics of the QBFs. Solvers based on DPLL are the systems semprop and QuBE−BJ. Using only the plain algorithm, would not lead to an efficient implementation, so a diversity of pruning techniques like for example dependency-directed backtracking are included. The solvers quantor and sKizzo follow a different approach to solve the evaluation problem. But all solvers have one property in common: they are only capable of processing formulas in PCNF. In the next section, we will we see that using PCNF can be problematic.

# Chapter 4

# Towards Decision Methods for Arbitrary QBFs

Most solvers presented so far are only capable of processing formulas in *prenex conjunctive normal form*. This restriction on the formulas' structure is not only used in QBF solving, but represents a very common approach in automated deduction for many formalisms (e.g., see [67]). The motivation behind the usage of normal forms is twofold.

- Every closed QBF can be transformed into an equivalent QBF in normal form.
- Due to that less complex structure, a QBF in normal form is easier to process.

The transformation of such a closed formula to an (equivalent) formula in normal form allows to make certain assumptions about the structure. Then simpler decision procedures and calculi can be applied for the evaluation. But the simplification has its price. As we will see, the following drawbacks arise.

- There is an increase in the formula size.
- There is an increase in the number of variables.
- There is a loss of information about the formula's structure.
- In general, a formula's normal form is not unique.

Important normal forms (like the negation, the prenex, and the conjunctive normal form) were introduced in Chapter 2. In this chapter, we will consider the concept of normal form transformation in a more detailed manner; we will see different approaches to obtain certain normal forms of a given formula, and we will discuss the impacts and trade-offs resulting from the transformation. Based on this findings, we will develop decision procedures, which make this normal form transformation obsolete.

As mentioned before, the decision procedure of Davis, Putnam, Loveland and Logeman presented in Chapter 3 represents one of the most widely used methods for calculating the

truth values of propositional formulas. The adaption of this method for QBFs has been implemented very successfully in numerous state-of-the-art QBF solvers (e.g., [22, 24, 41, 54, 68]).

We have seen a variant of the `DPLL` algorithm realised in the function `split`, which evaluates QBFs in prenex conjunctive normal form. This facilitates the handling of the formula, because the structure is reduced to sets of literals. Therefore, the implementation becomes less complex. As argued before, the encodings of real-world problems often do not lead to formulas in this normal form and thus an extra transformation, which is not deterministic, and which destroys the original structure of the formula, has to be applied.

We present a generalisation of `split` in such a way that the transformation into prenex conjunctive normalform can be omitted, and arbitrary QBFs can be processed. In this chapter, we give a sequent-style version of `split` to show its soundness and correctness.

Although we focus mainly on `DPLL`, it is not the only decision procedure considered in this thesis. We will also discuss the *connection-based validity characterisation* for QBFs. In contrast to `DPLL` which is a search-based decision procedure, the connection-based validity characterisation makes use of proof theoretical properties of QBFs. In fact, this calculus represents a compact and more efficient version of the previously introduced sequent calculus.

## 4.1 Normal Form Transformation Revised

" *The* normal form"of a formula does not exist. Normalisations can be applied to a certain degree depending on the demands of the processing tool. For example, we have seen negation normal form, where the negation signs may only appear in front of atoms, we have seen prenex normal form, where the quantifiers are extracted from the formula, and where they are shifted to the left-hand side, and finally we have seen conjunctive normal form where only a very restricted formula structure is allowed.

There are often many ways and strategies to generate a normalised version of a formula. Unfortunately, not all variants of a formula's normal form turn out to be of the same quality with respect to the solving process. Below we will briefly discuss how the kind of the chosen transformation influences the behaviour of the solvers. Following [7], we introduce the concept of normal form transformation in the style of a term rewriting system.

**Definition 4.1.1 (Reduction)**
A *reduction* is a binary, irreflexive relation on the set of QBFs. We write a reduction as a set of pairs containing QBFs. Note that a rule of a reduction can be applied to any suitable subformula of a QBF $\phi$, not only to $\phi$ itself.

**Example 4.1.1** Let $\Rightarrow$ be a reduction which is given by $\{(\neg(\phi_1 \wedge \phi_2), \neg\phi_1 \wedge \neg\phi_2)\}$ and let $\neg(p_1 \wedge p_2)$ and $(q_1 \wedge q_2)$ be QBFs. Then the first formula can be rewritten as $\neg p_1 \wedge \neg p_2$. There exists no reduction rule which can be applied to the second formula.

In the example above, the formula on the left-hand side of $\Rightarrow$ is not equivalent to the formula on the the right-hand side. Considerations about the preservation of the semantics during the transformation process are omitted. Such transformations are not of any interest for our purposes, because the idea of normalisation is to simplify only the structure and not to alter the meaning. So we introduce the notion of correct reductions.

**Definition 4.1.2 (Correct Reductions)**
Let $\Rightarrow$ be a reduction on QBFs. It is called *correct* if for all $(\phi, \psi) \in \Rightarrow$, $\phi \Rightarrow \psi$ implies that $\phi$ is equivalent to $\psi$.

**Example 4.1.2** The reduction $\Rightarrow = \{(\neg(\phi_1 \wedge \phi_2), \neg\phi_1 \vee \neg\phi_2)\}$ is a correct one, because $\neg(\phi_1 \wedge \phi_2)$ is equivalent to $\neg\phi_1 \vee \neg\phi_2$ due to De Morgan's laws.

**Definition 4.1.3 (Termination)**
A reduction $\Rightarrow$ is called *terminating* if there exists no infinite chain $\phi_1 \Rightarrow \phi_2 \Rightarrow \ldots$

**Example 4.1.3** The reductions $\{(\neg\phi, \neg\neg\phi)\}$ and $\{(\phi \vee \psi, \psi \vee \phi)\}$ are not terminating. Note that the first reduction increases the logical complexity of the QBF with every application, whereas the second does not.

**Definition 4.1.4 (Normal Form, Irreducibility)**
Let $\Rightarrow$ be a reduction on QBFs. Furthermore, let $\phi$ be a QBF and let $\phi'$ be a subformula of $\phi$. The formula $\phi$ is in *normal form* with respect to $\Rightarrow$ if there exists no formula $\psi$ with $\phi \Rightarrow \psi$ or $\phi' \Rightarrow \psi$. Then $\phi$ is called irreducible under $\Rightarrow$.

To get a normal form of a formula, we iteratively apply the reduction on each subformula, until we obtain an irreducible formula. Therefore, we introduce the following definition.

**Definition 4.1.5 (Normal Form of a Formula)**
Let $\Rightarrow$ be a reduction and $\Rightarrow^*$ be the transitive closure of $\Rightarrow$. Let $\phi \Rightarrow^* \psi$ hold and let $\psi$ be irreducible under $\Rightarrow$. Then $\psi$ is the *normal form* of $\phi$.

Note that this definition does neither guarantee the uniqueness nor the existence of the normal form of a formula. The QBF $\neg\phi$ has no normal form with respect to the first reduction introduced in Example 4.1.2.

Now we define the reduction relation for *negation normal form* (NNF). Recall that a QBF $\phi$ is in NNF if the negation symbol solely occurs directly in front of atoms, and and if all conjunctively and disjunctively truth constants have been eliminated.

**Definition 4.1.6 (The Reduction Relation of the NNF Transformation)**
The *reduction relation of the negation normal form* $\Rightarrow_n$ of a QBF is given as follows.

1. $\top \wedge \phi \Rightarrow_n \phi, \quad \phi \wedge \top \Rightarrow_n \phi$
2. $\bot \wedge \phi \Rightarrow_n \bot, \quad \phi \wedge \bot \Rightarrow_n \bot$
3. $\top \vee \phi \Rightarrow_n \top, \quad \phi \vee \top \Rightarrow_n \top$
4. $\bot \vee \phi \Rightarrow_n \phi, \quad \phi \vee \bot \Rightarrow_n \phi$

5. $\neg\bot \Rightarrow_n \top, \quad \neg\top \Rightarrow_n \bot$
6. $\neg\neg\phi \Rightarrow_n \phi$
7. $\neg(\mathsf{Q}x\,\phi) \Rightarrow_n \overline{\mathsf{Q}}x\,(\neg\phi)$, $\mathsf{Q} \in \{\forall, \exists\}$
8. $\neg(\phi \vee \psi) \Rightarrow_n (\neg\phi \wedge \neg\psi)$
9. $\neg(\phi \wedge \psi) \Rightarrow_n (\neg\phi \vee \neg\psi)$

If the reduction is applied, until no application is possible then a formula is said to be in *negation normal form.*

**Theorem 4.1.1 (Properties of the NNF Reduction)**
The NNF reduction relation $\Rightarrow_n$ has the following properties.

1. $\Rightarrow_n$ is terminating.
2. $\Rightarrow_n$ is correct.
3. For $\phi \Rightarrow_n \psi$, $\mathsf{k}(\psi) \leq \mathsf{k}(\phi) + 1$.

**Proof.**

1. To show the termination of $\Rightarrow_n$, we have to introduce a measure $\mathsf{n}(\cdot)$, which decreases with every application of $\Rightarrow_n$, i.e., it holds that $\mathsf{n}(\phi) > \mathsf{n}(\psi)$ for $\phi \Rightarrow_n \psi$. Let $\mathsf{n}(\phi)$ (resp. $\mathsf{n}(\psi)$) be the sum of twice the number of all connectives and quantifiers occurring in the scope of any negation sign plus the number of negation signs plus the number of truth constants in the QBF $\phi$ (resp. in $\psi$). For any rule given in Definition 4.1.6 it holds that $\mathsf{n}(\phi) > \mathsf{n}(\psi)$. Therefore, $\Rightarrow_n$ is terminating.
2. Correctness can be easily shown by proving that $\phi$ is equivalent to $\psi$ for every rule $\phi \Rightarrow_n \psi$ in Definition 4.1.6. The equivalences hold according to Theorem 2.2.1.
3. Rule 1 and 4 of Definition 4.1.6 decrease the logical complexity of the QBF on the left-hand side by one, the logical complexity of the QBF resulting from the application of rule 2 and 3 is zero. Rules 5 and 6 decrease the logical complexity of the QBF on the left-hand side by one and two, the seventh rule has no impact on the logical complexity. Only the last two rules increase the logical complexity of the formula by one. □

**Theorem 4.1.2 (Negation Normal Form Transformation)**
Let $\phi$ be a QBF and $\psi$ a normal form of $\phi$ w.r.t. $\Rightarrow_n$ according to Definition 4.1.6. Then $\psi$ is (1) in negation normal form and (2) $\phi$ and $\psi$ are equivalent.

**Proof.** (1) Assume that $\psi$ is in normal form with respect to $\Rightarrow_n$, but not in negation normal form. If $\psi$ is not in NNF then it contains a subformula of the form $\neg\psi'$ with $\psi' \notin \mathcal{P}$, or for $\circ \in \{\wedge, \vee\}$, a subformula of the form $\top \circ \psi'$, $\psi' \circ \top$, $\top \circ \psi'$, or $\psi' \circ \top$. This means that one of the rules of Definition 4.1.6 is applicable. This contradicts the assumption that $\psi$ is in normal form with respect to $\Rightarrow_n$.

(2) Since $\psi$ is a normal form of $\phi$ w.r.t. $\Rightarrow_n$, a chain of reductions $\phi \Rightarrow_n \phi'_1 \Rightarrow_n \ldots \phi'_n \Rightarrow_n \psi$ exists. According to Theorem 4.1.1, $\phi \Leftrightarrow \phi'_1$, $\phi'_i \Leftrightarrow \phi'_{i+1}$, and $\phi'_n \Leftrightarrow \psi$ hold for $1 \leq i < n$. Since $\Leftrightarrow$ is an equivalence relation and therefore transitive, $\phi$ and $\psi$ are equivalent. □

**Theorem 4.1.3 (Uniqueness of the NNF Transformation)**
The negation normal form of a QBF is *unique*.

**Proof.** To prove the uniqueness of the NNF reduction, we need several results from term rewriting. We sketch how the proof can be done and refer the reader to [6] for the characterisations and properties of term rewriting systems.

1. If $\Rightarrow_n$ is normalising and confluent then every QBF has a unique normal form (see Lemma 2.1.8 from [6]).

   Normalising means that every QBF has a normal form with respect to $\Rightarrow_n$. We showed this in Theorem 4.1.2. What is left to prove is the confluence.

2. A reduction is *confluent* if multiple rules can be applied to rewrite a QBF and finally yield the same resulting QBF (i.e., if $\phi \Rightarrow_n^* \psi_1$ and $\phi \Rightarrow_n^* \psi_2$, there exists a QBF $\gamma$ with $\psi_1 \Rightarrow_n^* \gamma$ and $\psi_2 \Rightarrow_n^* \gamma$ for every QBF $\phi, \psi_1, \psi_2$). Confluence is a very strong property and cannot be shown easily. In fact, it is an undecidable problem to prove the confluence of a finite term rewriting system.

3. Fortunately, we are dealing with a terminating term rewriting system. Therefore, we can apply *Newman's Lemma* (Lemma 2.7.2 in [6]), which states the following: A terminating reduction is confluent if it is locally confluent.

4. *Local confluence* is a weaker property than confluence (a reduction is locally confluent if $\phi \Rightarrow_n \psi_1$ and $\phi \Rightarrow_n \psi_2$, there exists a QBF $\gamma$ with $\psi_1 \Rightarrow_n^* \gamma$ and $\psi_2 \Rightarrow_n^* \gamma$ for every QBF $\phi, \psi_1, \psi_2$). Local confluence can be shown by using the *Critical Pair Lemma* (Lemma 6.2.3 in [6]) which states the following: a reduction is locally confluent if all its critical pairs are joinable. The QBFs $\psi_1$ and $\psi_2$ are joinable if there exists a QBF $\gamma$ with $\psi_1 \Rightarrow_n^* \gamma$ and $\psi_2 \Rightarrow_n^* \gamma$.

5. So, we have to find all the critical pairs. Shortly speaking, a critical pair is the result of unifying the left-hand side of one rule with a non-variable subterm of the left-hand side of another rule and reducing the resulting QBF using both rules (for a detailed explanation see page 139 in [6]). In our case, the critical pairs result from the combination of the "elimination of double negation rule" with any of the other rules except the rules for the elimination of the truth constants. It can be shown easily that the critical pairs are joinable (i.e., that they can be reduced to the same formula) and therefore $\Rightarrow_n$ is locally confluent and therefore confluent. So the negation normal form of a QBF is unique. □

**Example 4.1.4** Let $\phi$ be a QBF of the form

$$\neg(\forall x_1 \exists y_1(\neg(\neg(x_1 \vee y_1) \wedge x_1) \wedge \forall x_2(x_1 \vee \neg x_2 \vee y_1))).$$

The negation normal form of $\phi$ is

$$\exists x_1 \forall y_1((\neg x_1 \wedge \neg y_1 \wedge x_1) \vee \exists x_2(\neg x_1 \wedge x_2 \wedge \neg y_1)).$$

To sum up, the negation normal form transformation has the following properties.

- It is deterministic.
- It does not introduce new variables.
- The structure of the QBF is almost retained.
- It increases the logical complexity of the formula at most linearly by the distribution of the negation signs.

Because of these properties, the changes of a QBF by the negation normal form transformation are very moderate. The NNF transformation removes the polarities from subformulas other than literals by shifting negations inside the formula. As the difference between the original QBF and its NNF are not very sever, we will develop our solver for QBFs in NNF.

For most solvers NNF is not enough. The next step to obtain prenex conjunctive normal form, which those solvers process, is the prenexing. Here the quantifiers are moved to the left of the formula in such a manner that we get a sequence of quantifiers (i.e., the prefix) and the quantifier-free propositional part of the formula (called matrix). In the following, we assume the QBFs to be in standard form; otherwise the renaming of bound variables would be necessary.

**Definition 4.1.7 (The Reduction Relation of Prenexing)**
The *reduction relation of the prenex normal form* $\Rightarrow_p$ for a QBF is given as follows (where $\mathsf{Q} \in \{\forall, \exists\}$).

1. $\neg(\mathsf{Q}x\phi) \Rightarrow_p \overline{\mathsf{Q}}x(\neg\phi)$;
2. $((\mathsf{Q}x\phi) \circ \psi) \Rightarrow_p \mathsf{Q}x(\phi \circ \psi), \circ \in \{\vee, \wedge\}$;
3. $(\phi \circ (\mathsf{Q}x\psi)) \Rightarrow_p \mathsf{Q}x(\phi \circ \psi), \circ \in \{\vee, \wedge\}$.

If the reduction is applied until no application is possible then the resulting formula is said to be *prenex normal form.*

Note that the first rule of the prenexing reduction relation can be omitted if we assume the QBF to be in negation normal form.

**Theorem 4.1.4 (Properties of the Prenex Reduction)**
The prenex reduction relation $\Rightarrow_p$ respects to the following properties.

1. $\Rightarrow_p$ is terminating.
2. $\Rightarrow_p$ is correct.

3. The formula size remains constant.

**Proof.**

1. As in the termination proof for the NNF reduction, we introduce a measure $\mathsf{n}(\cdot)$, which decreases with every application of a reduction rule from $\Rightarrow_p$. A possible measure is the sum of the depth of the quantifiers in the structure tree. For any rule given in Definition 4.1.7 it holds that $\mathsf{n}(\phi) > \mathsf{n}(\psi)$. Therefore, as the quantifiers are shifted upwards within the structure tree, $\Rightarrow_p$ is terminating.

2. Correctness can be easily be shown by proving that $\phi$ is equivalent to $\psi$ for every rule in Definition 4.1.7 for $\Rightarrow_p$. The equivalences hold according to Theorem 2.2.1.

3. For each $(\phi, \psi) \in \Rightarrow_p$, both formulas $\phi$ and $\psi$ have the same logical complexity. □

**Theorem 4.1.5**
Let $\phi$ be a QBF and $\psi$ a normal form of $\phi$ w.r.t. $\Rightarrow_p$ according to Definition 4.1.7. Then (1) $\psi$ is in prenex normal form and (2) $\phi$ and $\psi$ are equivalent.

**Proof.** (1) Assume that $\psi$ is in normal form with respect to $\Rightarrow_p$, but not in prenex normal form. If $\psi$ is not in PNF then it contains a subformula of the form $\mathsf{Q}x\,\psi'$, which is neither $\psi$ nor has an immediate superformula with a quantifier as its main connective. This means that one of the rules of Definition 4.1.7 is applicable. This contradicts the assumption that $\psi$ is in normal form.

(2) Since $\psi$ is a normal form of $\phi$ w.r.t. $\Rightarrow_p$, a chain of reductions $\phi \Rightarrow_p \phi'_1 \Rightarrow_p \ldots \phi'_n \Rightarrow_p \psi$ exists. According to Theorem 4.1.4, $\phi \Leftrightarrow \phi'_1$, $\phi'_i \Leftrightarrow \phi'_{i+1}$, and $\phi'_n \Leftrightarrow \psi$ hold for $1 \leq i < n$. Since $\Leftrightarrow$ is an equivalence relation and therefore transitive, $\phi$ and $\psi$ are equivalent. □

**Example 4.1.5** Let $\phi$ be a QBF of the form

$$\exists x_1 \forall y_1((\forall y_2(\neg y_2 \wedge \neg y_1) \wedge x_1) \vee \exists x_2(\neg x_1 \wedge x_2 \wedge y_1)).$$

A prenex normal form of $\phi$ is

$$\exists x_1 \forall y_1 \forall y_2 \exists x_2((\neg y_2 \wedge \neg y_1 \wedge x_1) \vee (\neg x_1 \wedge x_2 \wedge y_1)).$$

Now we encounter the first severe problem on the way to PCNF. The transformation to PNF is not deterministic. Consider the very simple QBF $(\forall x x) \vee (\exists y y)$. The application of $\Rightarrow_p$ results in

$$\forall x \exists y(x \vee y),$$

or

$$\exists y \forall x(x \vee y),$$

depending on the order the rules are applied. This simple example illustrates that the prenexing is not deterministic, and therefore, the prenex normal form of a QBF is not unique in general. Note that we are only interested in the prenex normal forms of a formula with the smallest possible number of quantifier alternations.

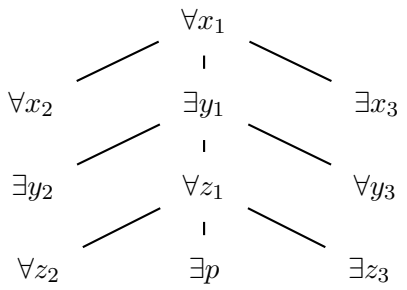



Figure 4.1: A quantifier dependency tree.

Also note that there is only a freedom of choice to order two quantifiers if they appear not on the same path in the structure tree. We can reduce the structure tree to the *quantifier dependency tree*, which imposes an order on the quantifiers which has to be respected during the prenexing.

**Definition 4.1.8 (Quantifier Dependency Tree)**
The *quantifier dependency tree* of a QBF $\phi$ is the structure tree of $\phi$, where only the nodes, which contain a quantifier, are included.

**Example 4.1.6** Let $\phi$ be a QBF of the form

$$\forall x_1((\forall x_2 x_2 \vee \exists x_3 x_3) \vee \exists y_1(\exists y_2 y_2 \wedge \forall y_3 y_3) \vee \forall z_1(\forall z_2 z_2 \vee \exists z_3 z_3) \vee \exists p p).$$

The quantifier dependency tree of $\phi$ is shown in Figure 4.1. The formula $\phi$ has eight different prenex normal forms minimal with respect to the number of quantifier alternations. Three of them are shown in Figure 4.2. Details about prenexing can be found in [96].

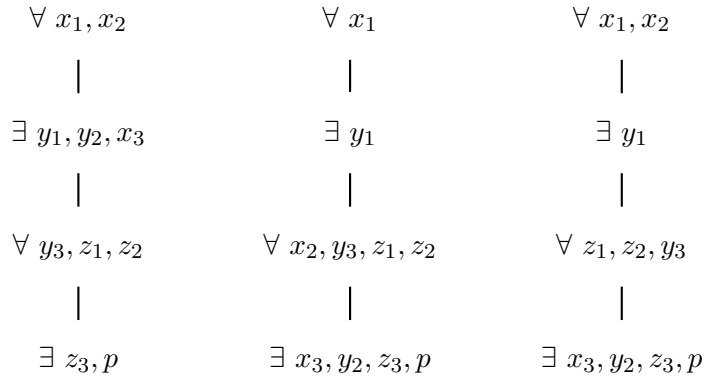



Figure 4.2: A quantifier dependency tree of formulas in PNF.

The situation becomes even worse as we continue. The last missing step is the transformation of the matrix into conjunctive normal form. Shortly speaking, two different kinds of CNF transformations are possible, namely a *nonstructural normal form transformation*

and a *structure-preserving normal form transformation*. As the name indicates, the non-structural transformation results in a loss of structural information of the input formula since parts of subformulas are disrupted by the application of the distributive law. Those parts are distributed into different clauses of the resulting formula.

**Definition 4.1.9 (The Reduction Relation of the CNF Transformation)**
Let $\phi$ be a QBF in prenex negation normal, form, i.e., $\phi = \mathsf{Q}_1 X_1 \ldots \mathsf{Q}_n X_n \psi$ where $\psi$ is purely propositional and in negation normal form. The reduction relation of the *conjunctive normal form* is given as follows.

1. $(\psi_1 \wedge \psi_2) \vee \psi_3 \Rightarrow_{c_1} (\psi_1 \vee \psi_3) \wedge (\psi_2 \vee \psi_3)$
2. $\psi_3 \vee (\psi_1 \wedge \psi_2) \Rightarrow_{c_1} (\psi_3 \vee \psi_1) \wedge (\psi_3 \vee \psi_1)$

If the reduction is applied on every subformula until no application is possible then the resulting formula is said to be in *conjunctive normal form*.

**Theorem 4.1.6 (Properties of the CNF Reduction)**
The CNF Reduction Relation $\Rightarrow_{c_1}$ has the following properties.

1. $\Rightarrow_{c_1}$ is terminating.
2. $\Rightarrow_{c_1}$ is correct.

**Proof.**

1. Again, we introduce a measure $\mathsf{n}(\cdot)$, which decreases with every application of a reduction rule of $\Rightarrow_{c_1}$.

$$n(\phi) = \begin{cases} 2 & \text{if } \phi \text{ is a literal or a (negated) truth constant;} \\ n(\phi_1) + n(\phi_2) & \text{if } \phi = \phi_1 \wedge \phi_2; \\ 2^{n(\phi_1)} * 2^{n(\phi_2)} & \text{if } \phi = \phi_1 \vee \phi_2. \end{cases}$$

   Therefore, $\Rightarrow_{c_1}$ terminates.

2. Correctness can be easily be shown by proving that $\phi$ is equivalent to $\psi$ for every rule in Definition 4.1.9 for $\Rightarrow_{c_1}$. The equivalences hold due to Theorem 2.2.1. □

**Theorem 4.1.7**
Let $\phi$ be a QBF in NNF and in prenex normal form and $\psi$ a normal form of $\phi$ w.r.t. $\Rightarrow_{c_1}$ according to Definition 4.1.9. Then $\psi$ is (1) in conjunctive normal form and (2) $\phi$ and $\psi$ are equivalent.

**Proof.** (1) Assume that $\psi$ is in normal form with respect to $\Rightarrow_{c_1}$ but not in conjunctive normal form. If $\psi$ is not in CNF then it contains a subformula of the form $(\psi_1 \wedge \psi_2) \vee \psi_3$. Therefore, a rule of the reduction given in Definition 4.1.9 is applicable. But this contradicts the assumption that $\psi$ is in normal form.

(2) Since $\psi$ is a normal form of $\phi$ w.r.t. $\Rightarrow_{c_1}$, a chain of reductions $\phi \Rightarrow_{c_1} \phi'_1 \Rightarrow_{c_1} \ldots \phi'_n \Rightarrow_{c_1} \psi$ exists. According to Theorem 4.1.6, $\phi \Leftrightarrow \phi'_1$, $\phi'_i \Leftrightarrow \phi'_{i+1}$, and $\phi'_n \Leftrightarrow \psi$ hold for $1 \leq i < n$. Since $\Leftrightarrow$ is an equivalence relation and therefore transitive, $\phi$ and $\psi$ are equivalent. □

Unfortunately, $\Rightarrow_{c_1}$ can result in an exponential blow-up with respect to the formula size. Consider the following example.

**Example 4.1.7** Let $\phi$ be a QBF in prenex negation normal form of the structure

$$\mathsf{Q}_1 X_1 \ldots \mathsf{Q}_m X_m \bigvee_{i=1}^{n} (l_{(i,1)} \wedge l_{(i,2)}),$$

where $l_{i,j}$ are pairwise distinct literals, i.e., $\phi$ contains $2n$ literal occurrences. The conjunctive normal form of $\phi$ is

$$\mathsf{Q}_1 X_1 \ldots \mathsf{Q}_m X_m \bigwedge_{j_1,\ldots,j_n \in \{1,2\}} (l_{(1,j_1)} \vee \cdots \vee l_{(n,j_n)}).$$

The normal form contains $2^n$ literal occurrences.

A transformation, which might increase the formula size exponentially, is problematic in general. We introduce a second reduction to conjunctive normal form, which behaves better. The idea is very simple: we abbreviate a complex subformula $\psi$ in a QBF $\phi$ by a label — a new existentially quantified variable $p$, which is defined to be equivalent to $\psi$. We replace every occurrence of $\psi$ by $p$ and add the formula $(p \leftrightarrow \psi)$ conjunctively to the matrix. Then we append $\exists p$ at the end of the prefix.

Before providing an algorithm to perform the structure-preserving CNF transformation, we have to introduce the notations of simple formulas and short definitions.

**Definition 4.1.10 (Simple Formula, Short Definition)**
A formula is called *simple* if all of its proper subformulas are either literals or truth constants. A formula is called *short definition* if it is of the form $p \rightarrow \phi$ where $p$ is a variable and $\phi$ is simple.

**Definition 4.1.11 (Structure-Preserving CNF Transformation)**
Let $\phi$ be a QBF in NNF and PNF. To obtain $\phi$'s PCNF by *structure-preserving CNF transformation* the following steps have to be performed.

1. Rewrite the formula by the application of the associativity law to the form

   $$\mathsf{Q}_1 X_1 \ldots \mathsf{Q}_n X_n \phi',$$

   such that $\phi' = \bigwedge_{i=0}^{m} \psi_i$ and no $\psi_i$ is a conjunction (for $1 \leq i \leq n$).

2. If $\phi'$ is in conjunctive normal form then terminate.

3. If some $\psi_i$ is a short definition, apply the nonstructural CNF transformation on $\psi_i$.

4. Choose one $\psi_i$ which is neither a clause nor a short definition and a simple subformula $\psi_i'$ of $\psi_i$. Substitute all occurrences of $\psi_i'$ by a fresh variable $p$ in $\phi'$. Let $\phi''$ be the result of the replacement. The variable $p$ is called a *label*. Replace the matrix $\phi'$ by

$$(\exists p\,(\phi'' \wedge (p \rightarrow \psi_i'))).$$

5. Continue with 1.

**Theorem 4.1.8 (Complexity of Structure-Preserving NFT)**
The time complexity (and therefore also the space complexity) of the transformation of a given QBF $\phi$ to PCNF is polynomial with respect to the size of $\phi$.

The newly introduced labels encode the structure of the original formula. If the prefix of a QBF in PNF ends with a block of universally quantified variable then the quantifier depth is increased by one. The new variables can have a great impact on the solving process depending on the solver used, even though theoretically, it should not. In practice, however, it can happen that thousands of new variables will be introduced, because of the complexity of the formula's structure. Most solvers are not able to distinguish between labels and "normal" variables. This matters enormously, because in most decision methods, it is not necessary to assign truth values to the labels as they get eliminated by the simplification rules automatically (as in the non-normal form case) under the assumption that the quantifications of the labels are appended at the end of the prefix. It is also possible to place the labels at another place in the prefix. The dependencies to the other variables in the formula have to be considered in order to decide where the quantifier is placed.

In what follows, we will propose two decision methods, which directly work on formulas of arbitrary structure, and thus are independent of any problematic normal form transformation.

## 4.2 The Basic DPLL Algorithm

The algorithm developed by Davis, Putnam, Logeman, and Loveland (DPLL) presented in the previous chapter is one of the most successful methods to decide the satisfiability of propositional formulas as well as the satisfiability of QBFs.

Since this method can be easily written down in a programming language, it is usually presented in a procedural manner. Because of its close relation to the semantics of the formalisms, discussions about soundness and correctness are omitted. We will follow this informal approach too, but we will wait until the next chapter to present our variant of DPLL in pseudo-code. This section is dedicated to a declarative description of DPLL by presenting the algorithm in a sequent calculus like style. By doing so, we abstract from implementational aspects as well as from practical concerns like memory requirements and control flow. This allows for a stronger focus on the discussion of theoretical properties, especially on soundness and completeness.

We characterise DPLL for closed but otherwise unrestricted QBFs by the set of axioms and rules shown in Figures 4.3 to 4.6 by providing rules as in the sequent calculus. Due to

$$\bot \vdash \quad (\mathrm{Ax}_\bot) \qquad \vdash \top \quad (\mathrm{Ax}_\top)$$

Figure 4.3: Axioms of DPLL.

$$\frac{\vdash \phi}{\neg\phi \vdash}\ (\neg_l) \qquad \frac{\phi \vdash}{\vdash \neg\phi}\ (\neg_r)$$

Figure 4.4: The negation rules.

$$\frac{\phi \vdash}{\phi \wedge \psi \vdash}\ (\wedge_{l'}) \quad \frac{\psi \vdash}{\phi \wedge \psi \vdash}(\wedge_{l''}) \qquad \frac{\vdash \phi \quad \vdash \psi}{\vdash \phi \wedge \psi}\ (\wedge_r)$$

$$\frac{\phi \vdash \quad \psi \vdash}{\phi \vee \psi \vdash}\ (\vee_l) \qquad \frac{\vdash \phi}{\vdash \phi \vee \psi}\ (\vee_{r'}) \quad \frac{\vdash \psi}{\vdash \phi \vee \psi}\ (\vee_{r''})$$

Figure 4.5: The conjunction and disjunction rules.

$$\frac{\phi[x/\bot] \vdash}{\exists x\phi \vdash}\ (\forall_{l'}) \ \frac{\phi[x/\top] \vdash}{\exists x\phi \vdash}\ (\forall_{l''}) \qquad \frac{\vdash \phi[x/\bot] \quad \vdash \phi[x/\top]}{\vdash \forall x\phi}\ (\forall_r)$$

$$\frac{\phi[x/\bot] \vdash \quad \phi[x/\top] \vdash}{\exists x\phi \vdash}\ (\exists_l) \qquad \frac{\vdash \phi[x/\bot]}{\vdash \exists x\phi}\ (\exists_{r'}) \ \frac{\vdash \phi[x/\top]}{\vdash \exists x\phi}\ (\exists_{r''})$$

Figure 4.6: The quantifier rules.

this similarity, we also use the notion of sequents in this context. Recall that a sequent is an ordered pair of the form $\langle \Phi, \Psi \rangle$, where $\Phi$ and $\Psi$ are sets of QBFs. $\Phi$ is called antecedent, $\Psi$ is called succedent. Figure 4.3 shows the two axioms or initial sequents, namely $\mathrm{Ax}_\bot$ and $\mathrm{Ax}_\top$, which indicate that (a part of) the formula has been evaluated to a truth value. For all possible main connectives of a formula, i.e., for the negation, the conjunction, the disjunction and the quantifications, rules are provided.

As in the previously introduced sequent calculus, the sequents above the line are called the *premises* and the sequents below are called the *conclusion*. By the term *reduction* we refer to a rule, which is read from the conclusion to the premise(s). A proof in DPLL is a tree constructed by the application of the rules, where the end sequent (i.e., the root of the tree) is the formula, which has to be proven and where the leaves are axioms. In the next two subsections, we will prove the rules to be sound and correct. Before doing so, we will illustrate how to apply DPLL.

**Example 4.2.8** Let $\phi$ be a QBF of the form

$$\forall x \exists y((\neg x \vee y) \wedge (x \vee \neg y)).$$

In the following, we are able to show that $\phi$ is satisfiable by the following DPLL-proof:

$$\frac{\alpha_1 \quad \alpha_2}{\vdash \forall x \exists y((\neg x \vee y) \wedge (x \vee \neg y))}\ (\forall_r)$$

The subproofs $\alpha_1$ and $\alpha_2$ are as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\bot \vdash}{\vdash \neg\bot}\ (\neg_r)}{\vdash (\neg\bot \vee \bot)}\ (\vee_{r'}) \quad \dfrac{\dfrac{\bot \vdash}{\vdash \neg\bot}\ (\neg_r)}{\vdash (\bot \vee \neg\bot)}\ (\vee_{r''})}{\vdash (\neg\bot \vee \bot) \wedge (\bot \vee \neg\bot)}\ (\wedge_r)}{\vdash \exists y(\neg\bot \vee y) \wedge (\bot \vee \neg y)}\ (\exists_{r'})}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash \top}{\vdash (\neg\top \vee \top)}\ (\vee_{r''}) \quad \dfrac{\vdash \top}{\vdash (\top \vee \neg\top)}\ (\vee_{r'})}{\vdash (\neg\top \vee \top) \wedge (\top \vee \neg\top)}\ (\wedge_r)}{\vdash \exists y(\neg\top \vee y) \wedge (\top \vee \neg y)}\ (\exists_{r''})}$$

Shortly speaking, DPLL consists of two parts: (1) the part, which "processes" the closed formula, and which does the search, and (2) the part, which does the simplifications. In our characterisation of DPLL, we omit the simplification part for the moment. The simplifications are of minor interest for now as they are equivalence-preserving transformations which decrease the formula's complexity, and therefore the search space. They can be applied at any time during the prove and will not change the result. We also do not care about the space used by the decision method. Obviously, if we do not impose the constraint that we only perform a depth-first and not a breath-first search, the amount of necessary space would not remain polynomial with respect to the input formula.

### 4.2.1 Soundness

In this subsection, we prove the soundness of DPLL for closed QBFs.

**Theorem 4.2.1 (Soundness of DPLL)**
DPLL is sound (i.e., every closed QBF provable in DPLL is valid).

**Proof.** We prove that the axioms and rules are sound (i.e., by the application of a rule on one (resp. two) valid sequents, only a valid sequent can be derived). In other words, we show that that valid premises lead to valid conclusions.

Recall the semantics of a sequent: A sequent $\phi_1, \ldots, \phi_m \vdash \psi_1, \ldots, \psi_n$ is valid iff whenever $v(\phi_i) = \mathsf{T}$ for all $1 \leq i \leq m$ then there exists a formula $\psi_j$ with $v(\psi_j) = \mathsf{T}$ $(1 \leq \mathsf{j} \leq \mathsf{n})$.

a) *The axioms are sound.*

The soundness of the axioms

$$\bot \vdash \quad (\mathrm{Ax}_\bot) \qquad \vdash \top \quad (\mathrm{Ax}_\top)$$

follows immediately from the semantics of the truth constants and from the semantics of the sequent.

b) *The rules are sound.*

- The negation rules.

$$\frac{\vdash \phi}{\neg\phi \vdash}(\neg_l)$$

Assume $\vdash \phi$ is valid. Then for each possible evaluation function $\upsilon$, $\upsilon(\phi) = \mathsf{T}$ and $\upsilon(\neg\phi) = \mathsf{F}$. Therefore, it holds that the sequent $\neg\phi \vdash$ is valid.

$$\frac{\phi \vdash}{\vdash \neg\phi}(\neg_r)$$

Assume $\phi \vdash$ is valid. Then for each possible evaluation function $\upsilon$, $\upsilon(\phi) = \mathsf{F}$ and $\upsilon(\neg\phi) = \mathsf{T}$. Therefore, it holds that the sequent $\vdash \neg\phi$ is valid.

- The conjunction rules.

$$\frac{\phi \vdash}{\phi \wedge \psi \vdash}\ (\wedge_{l'}) \qquad \frac{\psi \vdash}{\phi \wedge \psi \vdash}(\wedge_{l''})$$

Assume $\phi \vdash$ or $\psi \vdash$ or both are valid. Then for each possible evaluation function $\upsilon$, $\upsilon(\phi) = \mathsf{F}$ or $\upsilon(\psi) = \mathsf{F}$. Therefore, it holds that the sequent $\phi \wedge \psi \vdash$ is valid.

$$\frac{\vdash \phi \quad \vdash \psi}{\vdash \phi \wedge \psi}(\wedge_r)$$

Assume $\vdash \phi$ and $\vdash \psi$ are valid. Then for each possible evaluation function $\upsilon$, $\upsilon(\phi) = \mathsf{T}$ and $\upsilon(\psi) = \mathsf{T}$. Therefore, it holds that $\vdash \phi \wedge \psi$ is valid.

- The disjunction rules.

$$\frac{\phi \vdash \quad \psi \vdash}{\phi \vee \psi \vdash}(\vee_l)$$

Assume $\phi \vdash$ and $\psi \vdash$ are valid. Then for each possible evaluation function $\upsilon$, $\upsilon(\phi) = \mathsf{F}$ and $\upsilon(\psi) = \mathsf{F}$. Therefore, it holds that $\phi \vee \psi \vdash$ is valid.

$$\frac{\vdash \phi}{\vdash \phi \vee \psi}\ (\vee_{r'}) \qquad \frac{\vdash \psi}{\vdash \phi \vee \psi}(\vee_{r''})$$

Assume $\vdash \phi$ or $\vdash \psi$ or both are valid. Then for each possible evaluation function $\upsilon$, $\upsilon(\phi) = \mathsf{T}$ or $\upsilon(\psi) = \mathsf{T}$. Therefore, it holds that $\vdash \phi \vee \psi$ is valid.

- The existential quantifier rules.

$$\frac{\phi[x/\top] \vdash \quad \phi[x/\bot] \vdash}{\exists x\, \phi \vdash}(\exists_l)$$

Assume $\phi[x/\top] \vdash$ and $\phi[x/\bot] \vdash$ are valid. Then for each possible evaluation function $\upsilon$, $\upsilon(\phi[x/\top]) = \mathsf{F}$ and $\upsilon(\phi[x/\bot]) = \mathsf{F}$. Therefore, it holds that $\exists x\, \phi \vdash$ is valid.

$$\frac{\vdash \phi[x/\top]}{\vdash \exists\, x\phi}\ (\exists_{r'}) \qquad \frac{\vdash \phi[x/\bot]}{\vdash \exists x\, \phi}(\exists_{r''})$$

Assume $\vdash \phi[x/\top]$ or $\vdash \phi[x/\bot]$ or both are valid. Then for each possible evaluation function $\upsilon$, $\upsilon(\phi[x/\top]) = \mathsf{T}$ or $\upsilon(\phi[x/\bot]) = \mathsf{T}$. Therefore, it holds that $\vdash \exists x\, \phi$ is valid.

- The universal quantifier rules.

$$\frac{\phi[x/\top] \vdash}{\forall\, x\phi \vdash}\ (\forall_{l'}) \qquad \frac{\phi[x/\bot] \vdash}{\forall x\ \phi \vdash}(\forall_{l''})$$

Assume $\phi[x/\top] \vdash$ or $\phi[x/\bot] \vdash$ or both are valid. Then for each possible evaluation function $\upsilon$, $\upsilon(\phi[x/\top]) = \mathsf{F}$ or $\upsilon(\phi[x/\bot]) = \mathsf{F}$. Therefore, it holds that $\forall x\, \phi \vdash$ is valid.

$$\frac{\vdash \phi[x/\top] \quad \vdash \phi[x/\bot]}{\vdash \forall x\, \phi}(\forall_r)$$

Assume $\vdash \phi[x/\top]$ and $\vdash \phi[x/\bot]$ are valid. Then for each possible evaluation function $\upsilon$, $\upsilon(\phi[x/\top]) = \mathsf{F}$ and $\upsilon(\phi[x/\bot]) = \mathsf{F}$. Therefore, it holds that $\vdash \forall x\, \phi$ is valid. □

### 4.2.2 Completeness

In this subsection, we prove the completeness of DPLL. For this, we need the following lemma.

**Lemma 4.2.1 (Validity of Premises)**
If the conclusion of a sequent is valid then all premises are valid in the case of the rules $(\neg_l)$, $(\neg_r)$, $(\vee_l)$, $(\wedge_r)$, $(\exists_l)$, and $(\forall_r)$. For the other rules, at least one of the premises is valid.

**Proof.** This lemma can be easily shown by proving for each rule that valid conclusions imply valid premises. This can be done like in the soundness proof, but in the other direction. □

**Theorem 4.2.2 (Completeness of DPLL)**
DPLL is complete for QBFs (i.e., every valid QBF is provable in DPLL).

**Proof.** We show that every valid sequent $S$ has a proof in DPLL by induction over the complexity $\mathsf{k}(S)$ of $S$.

**Basis Step.** Assume $S$ is a valid sequent with $\mathsf{k}(S) = 0$. Then $S$ is of the form $x \vdash$ or $\vdash y$, where $x$ and $y$ are truth constants since we are only dealing with closed formulas. The variable $x$ must be $\bot$ and $y$ must be $\top$, because of the validity of $S$. Therefore, $S$ is an axiom and provable in DPLL.

**Induction Hypothesis.** If $S$ is a sequent with $\mathsf{k}(S) \leq n$ and if $S$ is valid then $S$ is provable in DPLL.

**Induction Step.** We have to show that a sequent $S$ with $\mathsf{k}(S) = n+1$, which is valid, is provable in DPLL.

Since the complexity of $S$ is greater than zero, $S$ contains a formula $\phi$ which is not a truth constant. Therefore a rule of DPLL according to the main connective of $\phi$ can be applied. The resulting premise(s) are (i) valid sequents because of Lemma 4.2.1 (or at least one is valid in the case of disjunction and existential quantification) and (ii) the complexity of the sequent of the premise(s) is not bigger than $n$. By induction hypothesis it follows that the premises are provable in DPLL. Hence $S$ has a proof in DPLL. Note that in the case of $(\wedge_l)$, $(\vee_r)$, $(\forall_l)$, and $(\exists_r)$ it might happen that only one of the premises is valid. Then this premise has to be chosen to continue in order to obtain a proof. □

## 4.3 The Connection Calculus for QBFs

In this section, we present another approach to evaluate QBFs: the *connection calculus* for QBFs. This decision procedure is also known as the matrix-characterisation of logical validity, originally developed for full classical first-order formulas in normal form by Bibel [15] and extended for arbitrary formulas in classical and non-classical first-order logic by Wallen [92]. A similar method has been developed by Andrews [1]. Unlike DPLL, the connection calculus is based on a proof-theoretical characterisation of the underlying formalisms and not on an almost brute-force search algorithm.

The sequent calculus as presented in Chapter 3 is a very prominent decision method not only for QBFs, but for many other formal languages too. Nevertheless, only few implemented proof system use it as the underlying decision method, because the sequent calculus is not suited for direct implementation due to redundancies identified by Wallen [92]. These redundancies can be divided into three groups:

1. notational redundancy;
2. irrelevance;
3. non-permutability of inference rules.

In the following, we will discuss what these redundancies in the context of QBF solving are, how they influence the proving process and how they can be eliminated. Before we present the connection calculus for QBFs, we have to introduce some basic terminology used for the new validity characterisation.

**Definition 4.3.1 (Signed Formula, Uniform Notation)**
A *signed formula* is a pair $\langle \phi, n \rangle$, where $\phi$ is a QBF and $n \in \{0, 1\}$. If $\phi \in (\mathcal{P} \cup \{\top, \bot\})$ then the signed formula is said to be *atomic*.

Non-atomic signed formulas can be classified as follows.

1. A signed formula of the form $\langle \phi_1 \vee \phi_2, 0 \rangle$, $\langle \phi_1 \wedge \phi_2, 1 \rangle$, $\langle \neg\phi, 0 \rangle$ or $\langle \neg\phi, 1 \rangle$ is of *$\alpha$-type* (also conjunctive type).

2. A signed formula of the form $\langle \phi_1 \vee \phi_2, 1\rangle$ or $\langle \phi_1 \wedge \phi_2, 0\rangle$ is of *β-type* (also disjunctive type).

3. A signed formula of the form $\langle \exists x\phi, 0\rangle$ or $\langle \forall x\phi, 1\rangle$ is of *γ-type* (also universal type).

4. A signed formula of the form $\langle \forall x\phi, 0\rangle$ or $\langle \exists x\phi, 1\rangle$ is of *δ-type* (also existential type).

### 4.3.1 Formula Trees and Notational Redundancies

Wallen [92] identified *notational redundancy* as the first redundancy to eliminate. Notational redundancy concerns the representation of intermediate states, which arise during the search. A proof in a sequent calculus contains many intermediate states (i.e., the intermediate derivations, which lead to the axioms). Especially disjunctive choices (like $\vee r$ or $\exists r$) demand to keep multiple states in memory, because it might be necessary to duplicate the current formula. In theory, this is no problem, but in practice, memory consumption has a great impact on the efficiency of the proof system. Alternative search paths make it often impossible to abandon the auxiliary sequents. Intermediate derivations can become very large, especially when the formula to evaluate is very large. This prohibitive overhead is one reason, why the sequent calculus is not suitable for implementation. To overcome this problem, we present a method to capture shared structures between sequents. Therefore, we need the notion of *formula tree* and *multiplicity*.

The cut-free sequent calculus as presented earlier possesses the *subformula property*. This means that a formula occurring during the derivation of an endsequent, is also a subformula of that endsequent. In this context, the notion of subformula is extended as follows. If $\phi = \mathsf{Q}x\,\psi$ is a QBF and $y$ is an arbitrary propositional variable then $\psi[x/y]$ is a subformula of $\phi$. Premises of each rule are completely formed from subformulas of the conclusion of the rule. This observation can be used to eliminate the notational redundancy. Each subformula is assigned a name — a *position* — which can be considered as a pointer to a concrete representation of the subformula. This method has been proposed in Bibel's connection calculus [15].

**Definition 4.3.2 (Formula Tree)**
A *formula tree* is the representation of the structure of a QBF $\phi$, where each node of the tree contains a label (called *position*). A label is associated with one of $\phi$'s subformulas. The formula tree obeys the following ordering: a position $p$ is below a position $q$ in the tree (written as $p \ll q$) if the subformula associated with $p$ is a proper subformula of the formula associated with the formula of $q$. By $\mathsf{lab}(k)$, we denote the subformula associated with the position $k$.

**Example 4.3.9** The formula tree of $\forall x \exists y((\neg x \vee y) \wedge (x \vee \neg y))$ is shown in Figure 4.7. The assignment of positions to the subformulas is given in Table 4.1.

$\Gamma$ and $\Delta$ denote the positions of type $\gamma$ (resp. $\delta$). By $\Gamma'$ (resp. $\Delta'$) we denote the set of positions, whose direct preceding positions are of type $\gamma$ (resp. $\delta$). A position labelled by an atom or a truth constant is called *atomic position*. Positions alone are not powerful enough to identify the occurrence of a subformula, because no differences between formulas,

| position $p$ | $\mathsf{lab}(p)$ |
|---|---|
| $a_0$ | $\forall x \exists y((\neg x \vee y) \wedge (x \vee \neg y))$ |
| $a_1$ | $\exists y((\neg x \vee y) \wedge (x \vee \neg x))$ |
| $a_2$ | $(\neg x \vee y) \wedge (x \vee \neg y)$ |
| $a_3$ | $(\neg x \vee y)$ |
| $a_4$ | $x$ |
| $a_5$ | $y$ |
| $a_6$ | $(x \vee \neg y)$ |
| $a_7$ | $y$ |
| $a_8$ | $x$ |

Table 4.1: The positions of the QBF $\forall x \exists y((\neg x \vee y) \wedge (x \vee \neg y))$.

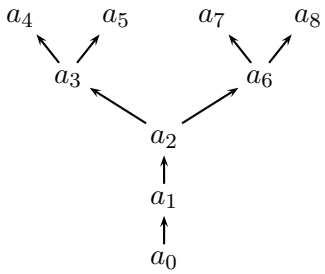



Figure 4.7: Formula tree of $\forall x \exists y((\neg x \vee y) \wedge (x \vee \neg y))$.

which appear in the antecedent and formulas, which appear in the succedent of the proof, are made. The second problem is the loss of information about the quantification of the variable in an atomic position. To identify the side, where a subformula occurs in an endsequent, we have to introduce the notion of polarity of a subformula.

**Definition 4.3.3 (Polarity of a Subformula or a Sequent)**
The *polarity of a subformula or a sequent* is inductively given as follows.

1. A QBF $\phi$ is always a positive subformula of $\phi$.
2. If $\psi$ is a positive (resp. negative) subformula of a QBF $\phi$ then it is
   - a positive (resp. negative) subformula of $\phi \vee \phi'$, $\phi \wedge \phi'$, $\forall x\, \phi$, $\exists x\, \phi$, and $\Phi \vdash \phi, \Psi$;
   - a negative (resp. positive) subformula of $\neg\phi$ and $\Phi, \phi \vdash \Psi$.

A cut-free sequent calculi has the well-known property that a subformula can only occur as a succedent (resp. antecedent) formula if it is a positive (resp. negative) subformula of the formula in the endsequent. To capture this property, we define the *formula tree for signed formulas*. A signed formula can be used for the representation of a formula in the proof, where 0 indicates that the formula is positive and 1 indicates that the formula is negative.

- $\langle \phi, 0 \rangle$ represents a formula in the succedent of a sequent;
- $\langle \phi, 1 \rangle$ represents a formula in the antecedent of a sequent.

**Definition 4.3.4 (Polarities of Position)**
The polarity $\mathsf{pol}(\cdot)$ of a position $p$ in a formula tree of a formula $\langle \phi, n \rangle$ ($n \in \{0, 1\}$) is given as follows.

- If $\mathsf{lab}(p)$ occurs positively in $\phi$ then $\mathsf{pol}(p) = n$.
- Otherwise, $\mathsf{pol}(p) = (n + 1) \bmod 2$.

**Example 4.3.10** Table 4.2 shows the polarities of the positions of the positive QBF

$$\forall x \exists y((\neg x \vee y) \wedge (x \vee \neg y))$$

shown in Table 4.1.

| **position** | $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ | $a_8$ |
|---|---|---|---|---|---|---|---|---|---|
| $\mathsf{pol}(p)$ | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

Table 4.2: The polarities of the positions of $\forall x \exists y((\neg x \vee y) \wedge (x \vee \neg y))$.

The given notions are still not enough to completely capture a proof in the sequent calculus. What happens if a subformula has to be duplicated, i.e., if the contraction rule of the sequent has to be applied? Consider the following example.

**Example 4.3.11** Let $\phi$ be a QBF of the form

$$\forall y \forall z(\neg \forall x x \vee (y \wedge z)).$$

It can be easily shown that $\phi$ is satisfiable (i.e., by the usage of the semantics). Our first trial to prove the formula in the sequent calculus is shown in Figure 4.8. Even though the derivation is obviously not finished, it will not work, i.e., we will not get a proof. So what went wrong? We would have got a proof if we had duplicated $\forall x x$ or if we had applied the $\forall r$ at the latest possible moment when the splitting due to the $\wedge r$ had been already done.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{y \vdash y \wedge z}{\forall x x \vdash y \wedge z}\,\forall l}{\vdash \neg \forall x x, (y \wedge z)}\,\neg r}{\vdash (\neg \forall x x \vee (y \wedge z)}\,\vee r}{\vdash \forall z(\neg \forall x x \vee (y \wedge z))}\,\forall r}{\vdash \forall y \forall z(\neg \forall x x \vee (y \wedge z))}\,\forall r$$

Figure 4.8: The proof of $\forall y \forall z(\neg \forall x x \vee (y \wedge z))$.

The problematic rule of the sequent calculus are the $\forall l$ and the $\exists r$ rules, where arbitrary constants and variables can be introduced, which might be necessary multiple times. Those

formulas are also called generative formulas. Without loss of completeness, it is only necessary to duplicate the generative formulas during a derivation. So we have to identify the positions, where such a duplication may occur. To indicate how many instances of a particular subformula may occur in a derivation, we introduce a property called *multiplicity*.

Shortly speaking, a multiplicity is a function encoding the number of different subformulas which have to be considered during the proof search. Let $p$ be a position, which represents a subformula $\mathsf{lab}(p)$. Different instances of this formulas can be distinguished by indexing the position, i.e., we write $p^\kappa$ where $\kappa$ is a sequence of positive integers. Instances of subformulas are only equal if they are equally indexed, i.e.,

$$\mathsf{lab}(k^\kappa) = \mathsf{lab}(k^\tau) \text{ if } \kappa = \tau.$$

**Definition 4.3.5 (Multiplicity)**
Let $\Phi = \langle \phi, 0 \rangle$ be a signed formula. A function $\mu$ which maps formulas of type $\Gamma'$ to the natural numbers is called multiplicity for $\Phi$.

**Definition 4.3.6 (Indexed Formula)**
$\Phi^\mu$ is called an *indexed formula* if $\Phi$ is a signed formula and $\mu$ is a multiplicity function.

**Definition 4.3.7 (Indexed Formula Tree, Indexed Position)**
An *indexed formula tree* for an indexed formula $\Phi^\mu$ is the formula tree of $\Phi$ with indexed positions of the form $p^\kappa$, where $p$ is a position of $\Phi$ and $\kappa$ is a sequence of positive integers defined as follows.

Let $p_1 \ll \cdots \ll p_n \ll p$ all those positions with $(p_i \in \Gamma')$ that precede $p$ in the formula tree of $\Phi$. By $p^\kappa$ we denote a position of the formula tree of the indexed formula $\Phi^\mu$ if the following conditions hold.

- $p$ is a position of $\Phi$,
- $\mu(p_i) \neq 0$ $(1 \leq i \leq n)$, and
- $\kappa = m_1 m_2 \ldots m_n$ where $1 \leq m_i \leq \mu(p_i)$, $1 \leq i \leq n$.

The expression $\kappa \prec \tau$ denotes that $\kappa$ is a proper initial sequence of $\tau$. The tree ordering $\ll$ is extended to the indexed tree as follows. An indexed position $p^\kappa$ is below an indexed position $q^\tau$ (written as $p \ll^\mu q$) if $p \ll q$ and $\kappa \preceq \tau$. The polarities are defined as in the unindexed case. The label $\mathsf{lab}(p^\kappa)$ of an indexed position $p^\kappa$ of a QBF $\phi$ is defined inductively as follows.

1. $\mathsf{lab}(p) = \phi$ if p is the root position of the tree.
2. If $\mathsf{lab}(p^\kappa) = \neg\psi$ and $p_1^\kappa$ is the child of $p^\kappa$ then $\mathsf{lab}(p_1^\kappa) = \psi$.
3. If $\mathsf{lab}(p^\kappa) = \psi_1 \circ \psi_2$ and $p_1^\kappa$ and $p_2^\kappa$ are the children of $p^\kappa$ then $\mathsf{lab}(p_i^\kappa) = \psi_i$.
4. If $\mathsf{lab}(p^\kappa) = \mathsf{Q}x\,\psi$ and $p'^\tau$ is the child of $p^\kappa$ for some sequence $\tau$ with $\kappa \preceq \tau$ then $\mathsf{lab}(p_1^\tau) = \psi[x/p_1^\tau]$.

### 4.3.2 Paths, Connections and Irrelevance

Let us illustrate the problem of *irrelevance* by an example.

**Example 4.3.12** Assume we obtained the following derivation during the backward proof search of a QBF.

$$\frac{x \vdash (x \vee \phi)\psi}{x \vdash (x \vee \phi) \vee \psi,} \vee r$$

The formulas $\phi$ and $\psi$ are arbitrary QBFs, and $x$ is as usual a variable. At this point, we are forced to make a choice: which of the two possible formulas shall we choose to continue? Is it preferable to choose $x \vee \phi$ or is it better to continue with $\psi$? Probably $\psi$ would be the wrong decision. If we choose $\psi$, it could happen that we will spend much effort into the decomposition of this formula and its subformulas (and this maybe for nothing). If we choose $x \vee \psi$ instead, we immediately obtain an axiom, and we are finished.

Those disjunctive choices are very problematic during the proof search. On the one hand, they could cause an enormous waste of resources. On the other hand, they also provide a high potential for optimisations. If the correct choice is made at such a decision point, it could prune the search space drastically. Unfortunately, it is not always so obvious as in the previous example, which formula is better, and which one is not.

We call this problem *irrelevance*. The sequent calculus has the problem of irrelevance, because only the main connectives of the formulas are considered, and because the internal structure of the formulas is neglected. Irrelevance can be prevented by introducing the concepts of *paths* and *connections*.

**Definition 4.3.8 ($\alpha$- and $\beta-$related Atomic Positions)**
Two atomic positions $p_1$ and $p_2$ are *$\alpha$-related* (denoted by $p_1 \sim_\alpha p_2$), resp. *$\beta$-related* (denoted by $p_1 \sim_\beta p_2$) if $p_1 \neq p_2$ and their greatest common ancestor w.r.t. $<$ is a formula of type $\alpha$ (resp. $\beta$).

**Definition 4.3.9 (Path)**
A *path* is the union of the maximal set of mutually $\alpha$-related atomic positions and $\{\top, \bot\}$. The polarity of $\top$ (resp. $\bot$) is 1 (resp. 0).

As mentioned above, logically, it is of no importance if we prove a QBF $\phi$ or $(\neg\top \vee \bot \vee \phi)$. Since the polarity of the formula to prove is 0, and because the truth constants are connected by $\vee$ to $\phi$, $\neg\top$ and $\bot$ are $\alpha$-related to every other position in $\phi$, so we include them in every path by definition.

**Example 4.3.13** $\{\top, \bot, a_4, a_5\}$, $\{\top, \bot, a_7, a_8\}$ are the paths of $\phi$ of the formula tree depicted in Figure 4.7.

**Lemma 4.3.1 (Paths and *Gqve*)**
Let $S$ be the set of all paths of a QBF $\phi$. For every leaf of a derivation of the QBF $(\neg\top \vee \ \bot \vee \phi)$ in *Gqve* of the form $(\Phi \vdash \Psi)$, $S$ contains a path $\mathsf{P}$ with

$$\Phi = \{\ \mathsf{lab}(p) \mid p \in \mathsf{P}, \mathsf{pol}(p) = 1\ \},$$

and

$$\Psi = \{\ \mathsf{lab}(p) \mid p \in \mathsf{P}, \mathsf{pol}(p) = 0\ \}.$$

**Definition 4.3.10 (Connection)**
A pair of $\alpha$-related atomic positions $\langle u, v\rangle$ is called *connection* if their polarities are different.
The connection is complementary under a substitution $\sigma$ iff

- $\mathsf{lab}(u) = \mathsf{lab}(v)$ where $\mathsf{lab}(u)$, $\mathsf{lab}(v) \in \Delta'$;
- $\sigma(\mathsf{lab}(u)) = \sigma(\mathsf{lab}(v))$ where $\mathsf{lab}(u)$, $\mathsf{lab}(v) \in \Gamma'$;
- $\sigma(\mathsf{lab}(u)) = \mathsf{lab}(v)$ where $\mathsf{lab}(u) \in \Gamma'$, $\mathsf{lab}(v) \in \Delta'$;
- $\mathsf{lab}(u) = \sigma(\mathsf{lab}(v))$ where $\mathsf{lab}(v) \in \Gamma'$, $\mathsf{lab}(u) \in \Delta'$.

A substitution is a mapping $\sigma : \Gamma' \mapsto \Gamma' \cup \Delta'$. It induces an equivalence relation $\sim \subseteq \Gamma' \times \Gamma'$ as follows. If $\sigma(p) = q$ and $q \in \Gamma'$ then $p \sim q$. Furthermore, a binary relation $\sqsubset \subseteq \Delta \times \Gamma$ is induced as follows. If $\sigma(p) = q$ and $q \in \Delta'$ then $p \sqsubset q$, and if $p \sqsubset q$ and $q \sim p'$ then $p \sqsubset p'$.

### 4.3.3 Reduction Orderings and Permutability

Wallen [92] states the third and last redundancy as the most fundamental and severe problem within the sequent-based proof search. This redundancy does not occur in the sequent calculus of propositional logic but only in the sequent calculus of first-order logic. Unfortunately, the sequent calculus of QBFs is also affected by the redundancy of *non-permutability*.

Again, we illustrate the problem with an example:

**Example 4.3.14** Assume we obtained the following sequent during a derivation:

$$\forall x(x \vee (\neg x \wedge \bot)) \vdash \forall yy.$$

If we continue with $\forall l$ then we get

$$\frac{y \vee (\neg y \wedge \bot) \vdash \forall yy}{\forall x(x \vee (\neg x \wedge \bot)) \vdash \forall yy}\ \forall l$$

It can be checked easily that we have lost at this point, and that we will not obtain a proof even though the sequent is satisfiable. The situation would have been different if we had applied the $\forall r$ rule before the $\forall l$ rule because we could have chosen the variable replacement of $x$ accordingly to the eigenvariable introduced in the succendent of the sequent.

Although it could have been easily detected in which order to apply the rules in this simple example, this *non-permutability* has a great impact and influence on the solving process in practice. To overcome this problem, we introduce the notion of *admissible substitution* which expresses *Skolemization*, well-known from the normal form transformation in first order logic, where the quantifier dependencies are expressed in terms of functions.

**Definition 4.3.11 (Admissible Substitution)**
A substitution $\sigma$ is called *admissible* with respect to a QBF $\phi$ iff the reduction ordering $\lhd := (\sqsubset \cup \ll)^+$ is irreflexive.

Note that the check if a substitution is admissible corresponds to the occur check in the unification.

**Lemma 4.3.2**
Let $P'$ denote the set of paths with respect to a QBF $\phi$ after the application of the substitution $\sigma$ (i.e., the labels with type $\Gamma'$ of some positions may have been changed). $\sigma$ is admissible iff there is a reduction in *Gqve*, which corresponds to $P'$ (i.e., $\sigma$ respects to the eigenvariable condition).

### 4.3.4 The Connection-Based Validity Characterisation

Now we have provided enough formal background to apply the connection-based validity characterisation to QBFs.

**Theorem 4.3.1 (Connection-Based Validity Characterisation)**
A QBF $\phi$ is valid iff there is an admissible substitution $\sigma$ and a set of $\sigma$-complementary connections such that every path through $\phi$ contains a connection from this set.

Note that we use unification not only to express substitutions but to ensure the existence of a sequent proof of the formula. Moreover, no single concrete order needs to be preferred. Redundant sequent derivations are removed from the search space by the identification. Now the problem of notational redundancies is solved by the usage of positions, the problem of irrelevance is void due to the notion of path and non-permutability is dealt with admissible substitutions.

**Example 4.3.15** Figure 4.9 shows the reduction ordering of

$$\forall x \exists y((\neg x \vee y) \wedge (x \vee \neg y)).$$

The graph is acyclic and therefore the substitution is irreflexive and according to Theorem 4.3.1 the formula is valid, which is obviously the case.

**Example 4.3.16** Figure 4.10 shows the reduction ordering of the unsatisfiable formula of

$$\exists x \forall y((\neg x \vee y) \wedge (x \vee y)).$$

As the reduction ordering is not irreflexive (i.e., it contains a cycle), according to Theorem 4.3.1, the formula is not valid.

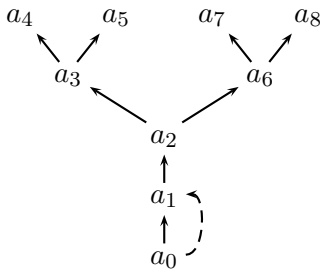



Figure 4.9: The reduction ordering of $\forall x \exists y((\neg x \vee y) \wedge (x \vee \neg y))$.

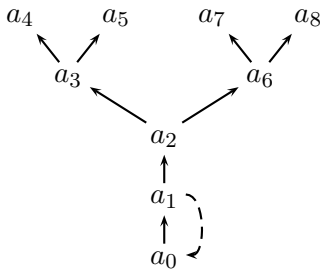



Figure 4.10: The reduction ordering of $\exists x \forall y((\neg x \vee y) \wedge (x \vee \neg y))$.

## 4.4 Discussion

In this chapter, we presented two different proof procedures for the evaluation of QBFs: (i) the search-based `DPLL` method, and (ii) the connection-based validity characterisation based on results from proof theory. Both methods have in common that they can be used to evaluate QBFs of arbitrary structure, i.e., no transformation to a specific normal form like prenex conjunctive normal form is necessary.

For both decision method, we implemented a prototype in the programming language Haskell and compared them. Tests indicated that the `DPLL` is the more promising decision method in practice. The implementation of the connection calculus performs well, as long as the multiplicities do not come into the play. But without multiplicities, the calculus is not complete any more, and therefore, not interesting for our purposes. Although the connection-based validity characterisation is the more elegant decision method of these two, we decided to focus on `DPLL`.

# Chapter 5

# The Solver qpro

So far, we have presented `split` in a very formal and abstract manner to analyse properties like soundness and completeness. In this chapter, we rewrite the decision method in a more operational manner (as it is usually done in the literature) and include some very important pruning techniques. Even if we get closer, we are still very far from a practical implementation as we do not consider the data structure and other details like how to realise the backtracking yet (this topic will be discussed in the next chapter). But still, the pseudo-code we present in the following, is the basis for our solver qpro.

```
BOOLEAN split(QBF in NNF φ) {
/* In:   closed QBF φ in negation normal form */
/* Out:  {T, F} (i.e., the truth value of φ)    */

 switch (simplify (φ)) {

  case ⊤:  return T;
  case ⊥:  return F;

  case (φ1 ∨ φ2): return (split(φ1) || split(φ2));
  case (φ1 ∧ φ2): return (split(φ1) && split(φ2));

  case (∃xψ):  return (split(ψ[x/⊥]) || split(ψ[x/⊤]));
  case (∀xψ):  return (split(ψ[x/⊥]) && split(ψ[x/⊤]));

 }

}
```

Figure 5.1: The basic algorithm of qpro.

## 5.1 The Basic Algorithm

Figure 5.1 shows the basic procedure `split` for the evaluation of QBFs in negation normal form. The pseudo-code we use consists of a mixture of elements found in standard procedural programming languages like C or Pascal and formal logics. The symbols of negation, disjunction, and conjunction are denoted by "`!`", "||", and "&&", a comparison is expressed by "==".

Note that our algorithm works in a deterministic manner: (i) we always take the left-most variable from quantifier block when it is necessary to choose a variable, (ii) we always replace the chosen variable first by $\bot$ and only if necessary by $\top$, and (iii) we process the immediate subformulas of a conjunction or disjunction in the order they are given. It is possible to integrate selection heuristics.

We restrict our attention to the deterministic version of `split` as given above, since tests showed that the improvement gained by heuristics is not very promising and the impact is not as much as one might suspect. Also in the QBF literature, the heuristics are not emphasised and nearly never mentioned, although implemented in most solvers.

```
QBF simplify(QBF in NNF φ) {
/* In:   QBF φ in negation normal form         */
/* Out:  simplified QBF in NNF equivalent to φ */

  switch(φ) {
   case (ψ1 ∧ ψ2) : φ' = (simplify(ψ1) ∧ simplify(ψ2));
   case (ψ1 ∨ ψ2) : φ' = (simplify(ψ1) ∨ simplify(ψ2));
   case (Qxψ)     : φ' = simplify(ψ);
                    if x ∈ free(φ') then φ' = Qxφ';
   otherwise      : φ' = φ;
  }

  switch(φ') {

   case (¬⊤)   : return(⊥);
   case (¬⊥)   : return(⊤);
   case (l ∧ l̄)  : return(⊥);
   case (l ∨ l̄)  : return(⊤);
   case (⊥ ∧ ψ) : return(⊥);
   case (⊤ ∨ ψ) : return(⊤);
   case (⊤ ∧ ψ) : return(ψ);
   case (⊥ ∨ ψ) : retunr(ψ);
   otherwise    : return(φ');
  }

}
```

Figure 5.2: The function `simplify`.

Every call of `split` induces an application of `simplify` which is shown in Figure 5.2. This function removes every occurrence of the truth constants $\bot$ and $\top$ according to Theorem 2.2.1. It eliminates simple tautologies and contradictions as well. In the normal form case, the check if a clause contains a literal $l$ and a literal $\overline{l}$ has to be done only once at the beginning, because the formula structure never changes — the size of a clause can only decrease, but no literal is added to a clause at any time during the solving process. In the nonclausal case, the structure of the formula changes very dynamically.

For the ease of readability, we write the disjunction and the conjunction only as binary connectives. But in an actual implementation, it is preferable to consider them as $n$-ary connectives (which does not change the semantics as the associative and commutative laws hold). Much more tautologies and contradictions can be detected this way.

For the moment, we only consider this very simple version of `simplify`, but during this chapter, we will extend it to a very powerful component of our algorithm and discuss it in a more detailed manner. Note that `simplify` is linear in space and time w.r.t. the size of a given QBF $\phi$ under the assumption that the check if a variable $x$ occurs freely in $\phi$.

Obviously, `split` is a direct application of the QBFs' semantics. The formula is split into subproblems, whose return values are included in an adequate manner, the bound variables are replaced by the truth constants and simplifications are applied until a truth value is obtained. The basic decision procedure is a simple search-based backtracking algorithm which is polynomial in space.

**Theorem 5.1.1 (Space and Time Complexity)**
Deciding the satisfiability of a QBF with the procedure `split` has polynomial space complexity w.r.t. the formula size and exponential time complexity w.r.t. the number of variables in the worst case.

## 5.2 Dependency-Directed Backtracking

Assume that `split` has to be applied on a QBF $\phi$, which has a quantifier as its main connective, i.e., $\phi$ is of the form $\mathsf{Q}x\,\psi$ ($\mathsf{Q} \in \{\forall, \exists\}$). Then the application of `split` on $\phi$ results in two subproblems, namely $\psi[x/\bot]$ and $\psi[x/\top]$. At least one of them has to be solved in any case. Under certain circumstances, the solution of the first subproblem allows for the omission of the second subproblem. If the quantifier is existential (resp. universal) and if the solution of the first treated subproblem is true (resp. false) then, due to the semantics of the quantifier, the second subproblem can be neglected. Suppose that the variable $x$ does not influence the evaluation result (i.e., $x$ is *irrelevant*) for the result of the first subproblem (see the example below). Obviously, it is not necessary to solve the second subproblem in any case. This technique to avoid to solve the second subproblem in this situation is called *dependency-directed backtracking* (also *backjumping* or *level cut*), and it is implemented by many state-of-the-art solvers [51, 52, 54, 68].

In the following, we present a generalisation of dependency-directed backtracking (DDB) in such a manner that we can integrate it into `split`. Before we start with the technical details, we give an example in order to illustrate how this technique prunes the search space.

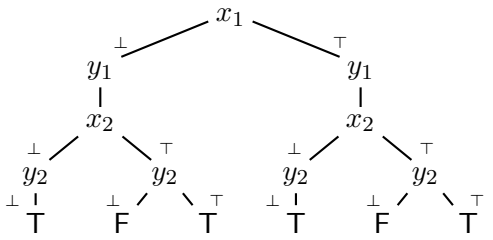



Figure 5.3: The splitting tree of $\forall x_1 \exists y_1 (\forall x_2 \exists y_2 ((x_2 \vee \neg y_2) \wedge (\neg x_2 \vee y_2)) \vee (x_1 \wedge y_1))$.

**Example 5.2.1** Let $\phi$ be the formula

$$\forall x_1 \exists y_1 (\forall x_2 \exists y_2 ((x_2 \vee \neg y_2) \wedge (\neg x_2 \vee y_2)) \vee (x_1 \wedge y_1)).$$

The semantic tree of $\phi$ is shown in Figure 5.3. Observe that the same subtree occurs on the left and on the right below $x_1$. The algorithm performs as follows.

1. Branch on the variable $x_1$ and start with $\phi'[x_1/\bot]$ ($\phi'$ denotes $\phi$ without $\forall x_1$). When we replace $x_1$ by $\bot$ in $\phi'$ and simplify $\phi'$ afterwards, we obtain the formula $\forall x_2 \exists y_2 ((x_2 \vee \neg y_2) \wedge (\neg x_2 \vee y_2))$. Note that `simplify` eliminates the quantifier $\exists y_1$, since $y_1$ does not occur in the formula anymore.

2. Branch on the variable $x_2$, and set it to $\bot$, which results in $\exists y_2 (\neg y_2)$ after the simplification.

3. If we replace $y_2$ by $\bot$, the formula evaluates to $\mathsf{T}$. We realize that only the variables $x_2$ and $y_2$ are responsible that the formula becomes true, i.e., $x_1$ does not influence the evaluation result. This is because the subformula $((x_2 \vee \neg y_2) \wedge (\neg x_2 \vee y_2))$ evaluates to true, which suffices that the whole formula evaluates to true. But this subformula contains no occurrence of $x_1$. Only $x_2$ is of interest for us, since it is universally quantified. If we return to the node of $y_2$ during backtracking, we can skip the second subproblem anyway when the first one has evaluated to true.

4. Since $x_2$ is universally quantified, we have also to consider the subproblem, where the variable $x_2$ is replaced by $\top$.

5. Setting $y_2$ to $\bot$ results in $\mathsf{F}$, but setting $y_2$ to $\top$ yields $\mathsf{T}$. Again, the relevant variables with respect to the result of this branch in the semantic tree are $x_2$ and $y_2$.

6. As $x_1$ is universally quantified and the solution of the first subproblem is true, we should also consider the second subproblem. As $x_1$ is never relevant for making $\phi'[x_1/\bot]$ true, it is unnecessary to consider the other problem.

In the following, we present two versions of dependency-directed backtracking quite similar to the algorithms implemented in the solver semprop for PCNF formulas by Letz [68]. The first one is a weaker version but with less implementational overhead, whereas the second one has higher potential in decreasing the search space.

Our algorithm works in the same way for a universal variable where the solution of the first subproblem is true (as in the example above), and for an existential variable where the solution of the first subproblem is false. If the QBF is in prenex clausal normal form then the handling of true and false subproblems must be distinguished. When the subproblem is false, the set of relevant variables is obviously given by the set of literals of one single false clause. But which variables shall be chosen as relevant if the subproblem is true? However, the complex formula structure allows us for a dual handling of true and false subproblems.

### 5.2.1 DDB by Labelling

To apply dependency-directed backtracking, we have to determine which variables are relevant for the solution of the first considered subproblem and which are not, such that we can decide if we shall solve the second subproblem during splitting or not. Therefore, we have to introduce the notion of the set of relevant variables with respect to a partial interpretation, which represents the assignment of variables during the search.

**Definition 5.2.1 (Set of Relevant Variables)**
Let $\phi$ be a QBF, $\iota_{\mathcal{S}}$ be an interpretation and $\phi' = \mathsf{psk}(\phi)$ be the propositional skeleton of $\phi$.

1. If $\upsilon_{\mathcal{S}}(\phi') = \mathsf{T}$ then the *set of relevant variables* $\mathsf{RV}_{\phi}(\mathcal{S})$ is defined as follows.

   - If $\phi'$ is a literal (say $l$) then

   $$\mathsf{RV}_{\phi}(\mathcal{S}) = \begin{cases} \{\mathsf{var}(l)\} & \text{if } \mathsf{var}(l) \text{ is universal in } \phi; \\ \{\} & \text{otherwise.} \end{cases}$$

   - Let $\phi'$ be a disjunction of the form $\phi_1 \vee \phi_2$ and let $\mathsf{RV}^1_{\phi}(\mathcal{S})$ (resp. $\mathsf{RV}^2_{\phi}(\mathcal{S})$) be the set of relevant variables for $\phi_1$ (resp. $\phi_2$). Then, for some $i$ from $\{1, 2\}$, the set of relevant variables is defined as $\mathsf{RV}_{\phi}(\mathcal{S}) = \mathsf{RV}^i_{\phi}(\mathcal{S})$ such that $\upsilon_{\mathcal{S}}(\phi_i) = \mathsf{T}$.
   - Let $\phi'$ be a conjunction of the form $\phi_1 \wedge \phi_2$ and let $\mathsf{RV}^1_{\phi}(\mathcal{S})$ (resp. $\mathsf{RV}^2_{\phi}(\mathcal{S})$) be the set of relevant variables for $\phi_1$ (resp. $\phi_2$). Then $\mathsf{RV}_{\phi}(\mathcal{S}) = \mathsf{RV}^1_{\phi}(\mathcal{S}) \cup \mathsf{RV}^2_{\phi}(\mathcal{S})$.

2. If $\upsilon_{\mathcal{S}}(\phi') = \mathsf{F}$, then $\mathsf{RV}_{\phi}(\mathcal{S})$ is defined as follows.

   - If $\phi'$ is a literal (say $l$) then

   $$\mathsf{RV}_{\phi}(\mathcal{S}) = \begin{cases} \{\mathsf{var}(l)\} & \text{if } \mathsf{var}(l) \text{ is existential in } \phi; \\ \{\} & \text{otherwise.} \end{cases}$$

   - Let $\phi'$ be a conjunction of the form $\phi_1 \wedge \phi_2$ and let $\mathsf{RV}^1_{\phi}(\mathcal{S})$ (resp. $\mathsf{RV}^2_{\phi}(\mathcal{S})$) be the set of relevant variables for $\phi_1$ (resp. $\phi_2$). Then, for some $i$ from $\{1, 2\}$, the set of relevant variables is defined as $\mathsf{RV}_{\phi}(\mathcal{S}) = \mathsf{RV}^i_{\phi}(\mathcal{S})$ such that $\upsilon_{\mathcal{S}}(\phi_i) = \mathsf{T}$.

- Let $\phi'$ be a disjunction of the form $\phi_1 \vee \phi_2$ and let $\mathsf{RV}^1_\phi(\mathcal{S})$ (resp. $\mathsf{RV}^2_\phi(\mathcal{S})$) be the set of relevant variables for $\phi_1$ (resp. $\phi_2$). Then $\mathsf{RV}_\phi(\mathcal{S}) = \mathsf{RV}^1_\phi(\mathcal{S}) \cup \mathsf{RV}^2_\phi(\mathcal{S})$.

Note that the set of relevant variables is not necessarily unique for a propositional skeleton of a QBF and an interpretation. Not all rules are deterministic, e.g., when $\phi' = \phi_1 \vee \phi_2$ and both $\phi_1$ and $\phi_2$ evaluate to true then a choice has to be made. In fact, it would not be incorrect to include the relevant variables of both formulas, but later we will need the set of relevant variables to decide if we must consider the second subproblem of a certain variable (as demonstrated in the previous example). The smaller the relevance set is, the better, because a variable missing in this set indicates that the second problem can be omitted. We also distinguish between variables of different quantification: if we calculate the set of relevant variables for a true subproblem, we only collect universally quantified variables, otherwise we only collect existentially quantified variables. We skip the second branch in the semantic tree of a universally quantified variable only if the subproblem has been evaluated to true — otherwise we would omit the second problem anyway.

In Figure 5.4, we present our algorithm extended by *dependency-directed backtracking by labelling*. The idea is very simple: when we branch on a variable $x$, we mark $x$ as irrelevant. If we reach a leaf of the semantic tree, we calculate the set of relevant variables, and we label all variables contained in this set as relevant. If we return to the variable $x$ during the backtracking process, we check whether $x$ has been set to relevant or not. If $x$ is still irrelevant, the second subproblem can be omitted and we can continue to backtrack.

The function `split` in Figure 5.4 has got further arguments now. In $\mathcal{S}$, the current variable assignments are stored, i.e., if the variable $x$ is substituted by $\top$ then $x \in \mathcal{S}$, if $x$ is substituted by $\bot$ then $\neg x \in \mathcal{S}$, otherwise $x$ has not got a truth value yet, and therefore $x \notin \mathcal{S}$. We need $\mathcal{S}$ to calculate the set of relevant variables. Note that we distinguish between the formulas $\Phi$ and $\phi$. The third argument $\Phi$ stands for the original input QBF, which is never altered. Therefore, $\Phi$ can be seen as some kind of "global" variable and $\phi$ as a local variable to the procedure representing the currently processed formula, i.e., the result of simplifications, variable substitutions, etc.

Obviously, dependency-directed backtracking allows for the removal of whole branches from the semantic tree. As we will see later, it turns out that DDB is absolutely necessary for obtaining competitive runtimes. In fact, an implementation of `split` without further pruning techniques except the inclusion of unit and pure rules is worthless for larger problem instances. But the important question we have to ask now is the question for soundness and completeness: is our algorithm still sound and complete? If not, the pruning technique would be worthless for us, because we are not interested to create a solver which does not work for either true or false instances in any case.

**Lemma 5.2.1**
Let $\phi$ be a QBF, $\phi' = \mathsf{psk}(\phi)$ the propositional skeleton of $\phi$, and let $x$ be a universally quantified variable in $\phi$. Further assume that $\upsilon_{\mathcal{S}}(\phi') = \mathsf{T}$, $\neg x \in \mathcal{S}$, and let $x$ be irrelevant, i.e., $x \notin \mathsf{RV}_\phi(\mathcal{S})$. Then $\upsilon_{\mathcal{S}'}(\phi') = \mathsf{T}$ with $\mathcal{S}' = (\mathcal{S} \setminus \{\neg x\}) \cup \{x\}$.

**Proof.** We prove this lemma by induction over the logical complexity $\mathsf{k}(\phi')$ of $\phi'$.

**Basis Step.** Assume that $\phi'$ is a literal or a truth constant, i.e., $\phi' = l$, with $\mathsf{var}(l) = y$ or $\phi' \in \{\top, \bot, \neg\top, \neg\bot\}$. Therefore, $\mathsf{k}(\phi') \leq 1$. For ordinary literals, we have to distinguish between three cases; the (negated) truth constants are treated accordingly.

Case 1: $y \neq x$.

From $\upsilon_{\mathcal{S}}(l) = \mathsf{T}$, immediately follows that $\upsilon_{\mathcal{S}'}(l) = \mathsf{T}$ holds because $\mathcal{S}\backslash\{\neg x\} = \mathcal{S}'\backslash\{x\}$.

Case 2: $l = x$.

Since $\neg x \in \mathcal{S}$, $\upsilon_{\mathcal{S}}(l) = \mathsf{F}$ holds. But this contradicts the assumption that $\upsilon_{\mathcal{S}}(\phi') = \mathsf{T}$.

Case 3: $l = \neg x$.

From $\upsilon_{\mathcal{S}}(l) = \mathsf{T}$, it follows that $x \in \mathsf{RV}_{\phi}(\mathcal{S})$ because $x$ is responsible that the formula evaluates to true. But this contradicts the assumption that $x \notin \mathsf{RV}_{\phi}(\mathcal{S})$.

**Induction Hypothesis.** Let $\phi$ be a QBF. For all propositional formulas $\phi'$ with $\mathsf{k}(\phi') \leq n$ which are subformulas of $\mathsf{psk}(\phi)$, it holds that, if $\upsilon_{\mathcal{S}}(\phi') = \mathsf{T}$, $\neg x \in \mathcal{S}$, and $x$ is irrelevant with repsect to $\phi$, i.e., $x \notin \mathsf{RV}_{\phi}(\mathcal{S})$ then $\upsilon_{\mathcal{S}'}(\phi') = \mathsf{T}$ with $\mathcal{S}' = (\mathcal{S} \setminus \{\neg x\}) \cup \{x\}$.

**Induction Step.** Let $\phi$ be a QBF and let $\phi'$ be a propositional formula which is a subformula of $\mathsf{psk}(\phi)$ with $\mathsf{k}(\phi') = n + 1$. If $\upsilon_{\mathcal{S}}(\phi') = \mathsf{T}$ and the variable $x$ with $\neg x \in \mathcal{S}$ is irrelevant with respect to $\phi$ then $\upsilon_{\mathcal{S}'}(\phi') = \mathsf{T}$ with $\mathcal{S}' = \mathcal{S}\backslash\{\neg x\} \cup \{x\}$. We have to show all cases for $\phi'$.

(1) Let $\phi' = \phi_1 \vee \phi_2$.

Then either $\upsilon_{\mathcal{S}}(\phi_1) = \mathsf{T}$ or $\upsilon_{\mathcal{S}}(\phi_2) = \mathsf{T}$ or both are true. It holds that for at least one $i$ for $i \in \{1, 2\}$, $x \notin \mathsf{RV}^i_{\phi}(\mathcal{S})$ with $\upsilon_{\mathcal{S}}(\phi_i) = \mathsf{T}$ where $\mathsf{RV}^i_{\phi}(\mathcal{S})$ denotes the set of relevant variables with respect to $\phi_i$; otherwise $x$ would not be irrelevant. By the induction hypothesis $\upsilon_{\mathcal{S}'}(\phi_i) = \mathsf{T}$ holds and so $\upsilon_{\mathcal{S}'}(\phi') = \mathsf{T}$.

(2) Let $\phi' = \phi_1 \wedge \phi_2$.

Then $\upsilon_{\mathcal{S}}(\phi_1) = \mathsf{T}$ and $\upsilon_{\mathcal{S}}(\phi_2) = \mathsf{T}$ hold. As $x$ is irrelevant in $\phi$, it holds that for all $i \in \{1, 2\}$, $x$ is not included in the set of relevant variables with respect to $\phi_i$. By the induction hypothesis $\upsilon_{\mathcal{S}'}(\phi_i) = \mathsf{T}$ holds and so $\upsilon_{\mathcal{S}'}(\phi') = \mathsf{T}$. □

**Definition 5.2.2 (Assignments for Bound Variables)**
Let $\phi$ be a closed QBF and $\mathcal{S}$ be a variable assignment. By $\phi_{\mathcal{S}}$ we denote the formula obtained from $\phi$ by substituting all occurrences of a variable $x$ by $\bot$ if $\neg x \in \mathcal{S}$ and by substituting $x$ by $\top$ if $x \in \mathcal{S}$.

**Definition 5.2.3 (Satisfying Assigments)**
Let $b$ be the semantic tree of a QBF $\phi$ with $\upsilon(\phi) = \mathsf{T}$. Then the set of *satisfying assignments* contains the sets of literals corresponding to the paths of $b$ with leaves labelled by $\mathsf{T}$.

**Theorem 5.2.1**
Let $\Phi$ and $\forall x\, \psi$ be QBFs, such that $\Phi_{\mathcal{V}} = \forall x\, \psi$ (where $\mathcal{V}$ is the current assignment). Furthermore, let $\upsilon(\psi[x/\bot]) = \mathsf{T}$ and let $\overline{\mathcal{A}}$ be the set of satisfying assignments of $\psi[x/\bot]$.

Let $\mathcal{A} \in \overline{\mathcal{A}}$. If $\upsilon_{\mathcal{S}}(\mathsf{psk}(\Phi)) = \mathsf{T}$ (with $\mathcal{V} \cup \mathcal{A} \cup \{\neg x\} \subseteq \mathcal{S}$) and $x \notin \mathsf{RV}_{\Phi}(\mathcal{S})$ then (i) $\upsilon_{\mathcal{S}'}(\mathsf{psk}(\Phi)) = \mathsf{T}$ (with $\mathcal{V} \cup \mathcal{A} \cup \{x\} \subseteq \mathcal{S}'$). (ii) the subproblem $\upsilon(\psi[x/\top]) = \mathsf{T}$.

**Proof.** (i) follows immediately by the application of Lemma 5.2.1 on each element of $\overline{\mathcal{A}}$.
(ii) $\overline{\mathcal{A}}$ contains every assignment $\mathcal{A}$ necessary to satisfy $\mathsf{psk}(\psi[x/\bot])$. Furthermore, there exists an assignment $\mathcal{S}$ with $\mathcal{A} \cup \{\neg x\} \subseteq \mathcal{S}$ such that $\upsilon_{\mathcal{S}}(\mathsf{psk}(\Phi)) = \mathsf{T}$. According to (i), $\upsilon_{\mathcal{S}'}(\mathsf{psk}(\Phi)) = \mathsf{T}$ with $\mathcal{A} \cup \{x\} \subseteq \mathcal{S}$. So the variable $x$ has no influence on the satisfiability of $\Phi$ with respect to the assignment $\mathcal{S}$. Since this holds for all $\mathcal{A} \in \overline{\mathcal{A}}$, the subproblem $\psi[x/\top]$ also evaluates to $\top$. □

**Theorem 5.2.2**
Let $\Phi$ and $\exists x\, \psi$ be QBFs, such that $\Phi_{\mathcal{V}} = \exists x\, \psi$ (where $\mathcal{V}$ is the current assignment). Furthermore, let $\upsilon(\psi[x/\bot]) = \mathsf{F}$ and let $\overline{\mathcal{A}}$ be the set of satisfying assignments of $\psi[x/\bot]$. Let $\mathcal{A} \in \overline{\mathcal{A}}$. If $\upsilon_{\mathcal{S}}(\mathsf{psk}(\Phi)) = \mathsf{F}$ (with $\mathcal{V} \cup \mathcal{A} \cup \{\neg x\} \subseteq \mathcal{S}$) and $x \notin \mathsf{RV}_{\Phi}(\mathcal{S})$ then (i) $\upsilon_{\mathcal{S}'}(\mathsf{psk}(\Phi)) = \mathsf{F}$ (with $\mathcal{V} \cup \mathcal{A} \cup \{x\} \subseteq \mathcal{S}'$). (ii) the subproblem $\upsilon(\psi[x/\top]) = \mathsf{F}$.

### 5.2.2 DDB by Relevance Sets

The smaller the set of variables labelled as relevant, the better it is for the efficiency of the solving process. If a variable is irrelevant then the second subproblem can be skipped under any circumstances. Consider the following case: we branch on an existentially quantified variable and the first subproblem evaluates to false. As $x$ is labelled as relevant, we have to consider the second subproblem too. But also the second subproblem evaluates to false, but now $x$ is not relevant. We have labelled the variables of both subproblems as relevant. If we had considered the second subproblem as the first one, the situation would have been very different: (1) the other problem could have been omitted, (2) the set of the variables labelled as relevant would have been smaller.

We cannot undo our decision to avoid the processing of the wrong subproblem, but we can reduce the set of relevant variables at least. We collect the relevant variables of the subproblems in *relevance sets*, make some case distinctions with respect to the main connective of the currently processed formula and construct the smaller set of relevant variables accordingly. Before we describe how to construct and treat such relevance sets, we motivate the approach by a short example.

**Example 5.2.2** Let $\phi$ be the formula

$$\forall z \forall x_1 \exists y_1 (\forall x_2 \exists y_2 ((x_2 \vee \neg y_2) \wedge (\neg x_2 \vee y_2)) \vee (\neg x_1 \wedge \neg y_1 \wedge \neg z)).$$

A semantic tree of $\phi$ is shown in Figure 5.5.

1. Branch on the variable $z$ and set $z$ to $\bot$.

   We get $\forall x_1 \exists y_1 (\forall x_2 \exists y_2 ((x_2 \vee \neg y_2) \wedge (\neg x_2 \vee y_2)) \vee (\neg x_1 \wedge \neg y_1))$ after simplification.

2. Branch on the variable $x_1$ and set $x_1$ to $\bot$.

   We get $\exists y_1 (\forall x_2 \exists y_2 ((x_2 \vee \neg y_2) \wedge (\neg x_2 \vee y_2)) \vee (\neg y_1))$ after simplification.

```
BOOLEAN split(φ,S,Φ) {
/* In:   closed QBF φ in NNF, set S of assignments, the input QBF Φ */
/* Out:  {T,F} (i.e., the truth value of φ)                          */

  φ' = simplify(φ);

  switch(φ')
   case ⊤        :  calculate RV_Φ(S);
                    for all x ∈ RV_Φ(S) setRelevant(x);
                    return T;
   case ⊥        :  calculate RV_Φ(S);
                    for all x ∈ RV_Φ(S) setRelevant(x);
                    return F;

   case (φ1 ∨ φ2) : return (split(φ1,S,Φ) || split(φ2,S,Φ));
   case (φ1 ∧ φ2) : return (split(φ1,S,Φ) && split(φ2,S,Φ));

   case ∃xψ      : setIrrelevant(x)
                   if ((split(ψ[x/⊥],S ∪ {¬x},Φ) == F) {

                     if isIrrelevant(x) return F;
                     else return split(ψ[x/⊤],S ∪ {x},Φ);

                   }
                   return T;

   case ∀xψ      : setIrrelevant(x)
                   if ((split(ψ[x/⊥],S ∪ {¬x},Φ) == T) {

                     if isIrrelevant(x) return T;
                     else return split(ψ[x/⊤],S ∪ {x},Φ);

                   }
                   return F;
}
```

Figure 5.4: The algorithm with dependency-directed backtracking.

3. Branch on the variable $y_1$ and set $y_1$ to $\bot$. The formula evaluates to $\mathsf{T}$. Obviously, the relevant (universal) variables are $z$ and $x_1$, so we mark them as relevant.

4. As $x_1$ is universally quantified, we have also to consider the subproblem, where $x_1$ is set to $\top$. We obtain $\forall x_2 \exists y_2((x_2 \vee \neg y_2) \wedge (\neg x_2 \vee y_2))$ after simplification.

5. If we replace $x_2$ and $y_2$ by $\bot$, the formula evaluates to $\mathsf{T}$. We realize that $x_2$ is the only universally quantified variable responsible that the formula $\phi$ becomes true, i.e., $x_1, y_1$ and $z$ do not influence the result.

6. Since $x_2$ is universally quantified, we have also to consider the second subproblem of $\forall x_2 \exists y_2((x_2 \vee \neg y_2) \wedge (\neg x_2 \vee y_2))$, where the variable $x_2$ is replaced by $\top$. We obtain $\exists y_2 y_2$ after simplification.

7. Setting $y_2$ to $\bot$ results in $\mathsf{F}$, but setting $y_2$ to $\top$ yields $\mathsf{T}$. Again, the only relevant universal variable with respect to the result of this branch in the semantic tree is $x_2$.

8. When we return to $x_1$ during backtracking, we notice that $x_1$ is not relevant in the second subproblem, where the variable has been replaced by $\top$. That is bad luck — if we had chosen the second subproblem as the first one to consider, we could have omitted setting $x_1$ to $\bot$. The set of relevant variables would have been a different one: $z$ would not have been included (recall that $z$ is still labelled as relevant because of 3.).

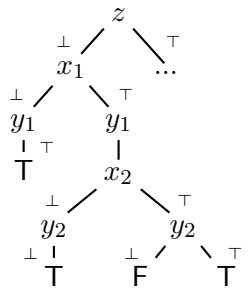



Figure 5.5: The semantic tree of $\forall x_1 \exists y_1 (\forall x_2 \exists y_2 ((x_2 \vee \neg y_2) \wedge (\neg x_2 \vee y_2 \vee \neg x_1)) \vee (x_1 \wedge y_1))$.

**Definition 5.2.4 (Relevance Set for False Subproblems)**
Let $\mathsf{R}_1$ and $\mathsf{R}_2$ be the relevance sets of the QBFs $\phi_1$ and $\phi_2$. Let $\phi$ be a QBF and $\phi'$ be a formula obtained during the splitting. The relevance set $\mathsf{R}$ of $\phi'$ with respect to an interpretation $\iota_{\mathcal{S}}$ with $\upsilon_{\mathcal{S}}(\mathsf{psk}(\phi)) = \mathsf{F}$ is obtained as follows.

- If $\phi' = \bot$ then $\mathsf{R} = \mathsf{RV}_{\phi}(\mathcal{S})$.
- If $\phi' = \phi_1 \vee \phi_2$ then $\mathsf{R} = \mathsf{R}_1 \cup \mathsf{R}_2$.
- If $\phi' = \phi_1 \wedge \phi_2$ then, for some $i$ from $\{1, 2\}$ and for $\upsilon_{\mathcal{S}}(\phi_i) = \mathsf{F}$, $\mathsf{R} = \mathsf{R}_i$.
- If $\phi = \forall x\, \psi$ with $\phi_1 = \psi[x/\bot]$ and $\phi_2 = \phi[x/\top]$ then, for some $i$ from $\{1, 2\}$ and for $\upsilon_{\mathcal{S}}(\phi_i) = \mathsf{F}$, $\mathsf{R} = \mathsf{R}_i$.
- If $\phi = \exists x\, \psi$, we get two subproblems $\phi[x/\bot]$ and $\phi[x/\top]$ — let us call them $\phi_1$ and $\phi_2$. Let further $\mathsf{R}$ denote the relevance set of $\phi$, $\mathsf{R}_1$ denote the relevance set of $\phi_1$ and $\mathsf{R}_2$ denote the relevance set of $\phi_2$.
  - If $x \notin \mathsf{R}_1$ then $\phi_2$ can be considered as false and $\mathsf{R} = \mathsf{R}_1$.
  - Otherwise $\phi_2$ has to be solved. If $\phi_2$ is also false, then
    - if $x \notin \mathsf{R}_2$ then $\mathsf{R} = \mathsf{R}_2$;
    - if $x \in \mathsf{R}_1 \cap \mathsf{R}_2$ then $\mathsf{R} = \mathsf{R}_1 \cup \mathsf{R}_2$.

**Definition 5.2.5 (Relevance Set for True Subproblems)**
Let $\mathsf{R}_1$ and $\mathsf{R}_2$ be the relevance sets of the QBFs $\phi_1$ and $\phi_2$. Let $\phi$ be a QBF and $\phi'$ be a formula obtained during the splitting. The relevance set $\mathsf{R}$ of $\phi'$ with respect to an interpretation $\iota_{\mathcal{S}}$ with $\upsilon_{\mathcal{S}}(\mathsf{psk}(\phi)) = \mathsf{T}$ is obtained as follows.

- If $\phi' = \top$ then $\mathsf{R} = \mathsf{RV}_{\phi}(\mathcal{S})$.
- If $\phi' = \phi_1 \wedge \phi_2$ then $\mathsf{R} = \mathsf{R}_1 \cup \mathsf{R}_2$.
- If $\phi' = \phi_1 \vee \phi_2$ then, for some $i$ from $\{1, 2\}$ and for $\upsilon_{\mathcal{S}}(\phi_i) = \mathsf{T}$, $\mathsf{R} = \mathsf{R}_i$.
- If $\phi = \exists x\, \psi$ with $\phi_1 = \psi[x/\bot]$ and $\phi_2 = \phi[x/\top]$ then, for some $i$ from $\{1, 2\}$ and for $\upsilon_{\mathcal{S}}(\phi_i) = \mathsf{T}$, $\mathsf{R} = \mathsf{R}_i$.
- If $\phi' = \forall x\, \psi$, we get two subproblems $\phi[x/\bot]$ and $\phi[x/\top]$ — let us call them $\phi_1$ and $\phi_2$. Let further $\mathsf{R}$ denote the relevance set of $\phi$, $\mathsf{R}_1$ denote the relevance set of $\phi_1$ and $\mathsf{R}_2$ denote the relevance set of $\phi_2$.
  - If $x \notin \mathsf{R}_1$ then $\phi_2$ can be considered as true and $\mathsf{R} = \mathsf{R}_1$.
  - Otherwise $\phi_2$ has to be solved. If $\phi_2$ is also true, then
    - if $x \notin \mathsf{R}_2$ then $\mathsf{R} = \mathsf{R}_2$;
    - if $x \in \mathsf{R}_1 \cap \mathsf{R}_2$ then $\mathsf{R} = \mathsf{R}_1 \cup \mathsf{R}_2$.

**Theorem 5.2.3**
Dependency-directed backtracking by relevance sets is sound and complete.

**Proof.** Follows immediately from Theorem 5.2.1 and Theorem 5.2.2 as the order the subproblems are processed is of no importance for the truth value of a QBF. □

Figure 5.6 shows the implementation of `split` extended by dependency-directed backtracking by relevance sets. It has the same arguments as `split` extended by labelling but the return value is different. Besides the truth value, the procedure returns the current relevance set.

## 5.3 Improving the Function `simplify`

The efficiency of `split` relies heavily on the functionality of `simplify`. This function not only removes the truth constants from the formula but it is also responsible for the application of more sophisticated simplification rules like local and global unit and pure. The pseudo-code of the improved version is given in Figure 5.7. The function `simplify` is applied to the QBF $\phi$ until no more simplifications can be performed.

One important task of `simplify` is the removal of truth constants occurring in the formula. This is achieved by making use of the rules given in Theorem 2.2.1. The last feature we included in Figure 5.7 is the application of pure and unit when possible (see Figure 5.8, 5.9, and 5.10) as well as miniscoping (see Figure 5.11). Note that all functions have a variable assignment $\mathcal{S}$ as argument, that they manipulate the current assignment

```
(BOOLEAN, relset) split(φ,S,Φ) {
/* In:   closed QBF φ in NNF φ, set S of assignments,
         the input QBF Φ */
/* Out:  ({T, F}, set of relevant variables) */

  φ' = simplify(φ,S);

  switch(φ')
   case ⊤        : return (T, RV_Φ(S));
   case ⊥        : return (F, RV_Φ(S));

   case (φ1 ∨ φ2) : (r1,R1) = split(φ1,S,Φ);
                    if (r1 == T) then return (T,R1);
                    (r2,R2) = split(φ2,S,Φ);
                    if (r2 == F) then return (F,R1 ∪ R2);
                    else return (T,R2);

   case (φ1 ∧ φ2) : (r1,R1) = split(φ1,S,Φ);
                    if (r1 == F) then return (F,R1);
                    (r2,R2) = split(φ2,S,Φ);
                    if (r2 == T) then return (T,R1 ∪ R2);
                    else return (F,R2);

   case ∃xψ       : (r1,R1) = (split(ψ[x/⊥],S ∪ {¬x},Φ);
                    if (r1 == T) then return (T, R1);
                    if (x ∉ R1) then return (F, R1);

                    (r2,R2) = (split(ψ[x/⊤],S ∪ {x},Φ);
                    if (r2 == T) then return (T, R2);
                    if (x ∉ R2) then return (F,R2);

                    return (F,R1 ∪ R2);

   case ∀xψ       : (r1,R1) = (split(ψ[x/⊥],S ∪ {¬x},Φ);
                    if (r1 == F) then return (F, R1);
                    if (x ∉ R1) then return (T, R1);

                    (r2,R2) = (split(ψ[x/⊤],S ∪ {x},Φ);
                    if (r2 == F) then return (F, R2);
                    if (x ∉ R2) then return (T,R2);

                    return (T,R1 ∪ R2);
}
```

Figure 5.6: DPLL extended by DDB by relevance sets.

$\mathcal{S}$ and that they return the altered set. As we will see soon, this is necessary to combine unit and pure reduction with DDB.

Note that `simplify`, as well as `pure`, `gunit`, and `lunit` make use of auxiliary functions which we describe only verbally. The function *isUnit* (resp. *isPure*) checks whether a variable in the second argument of the function is unit (resp. pure) in the QBF in the first argument according to Definitions 2.1.21 and 2.1.22. The function *quant* determines the kind of quantification of a variable, i.e., it determines whether the variable is universal or existential. The function $\mathsf{lit}(\phi, x)$ is applied on a QBF $\phi$ and a variable $x$. The return value is *positive* if all occurrences of $x$ in $\phi$ are positive, the return value is *neg* if all occurrences are negative and *both* in the remaining case. In `gunit` we use the function *getPolarityOfUnitVar* which determines the polarity of the literal where it is unit in the formula.

The function `miniscope` shifts quantifiers inside the formula which is in contrast to prenexing where the quantifiers are shifted outside. Miniscoping is done to increase the probability that a variable becomes unit or pure.

### 5.3.1 DDB with Unit and Pure

What we have presented so far, is a simplified version of our algorithm: we have omitted the unit (local as well as global) and the pure rule (recall Definitions 2.1.21, 2.1.22, and 2.1.23). Dependency-directed backtracking also works together with them, in fact, it is very important to include all of the three rules but the description of the algorithm becomes more complex. The following example illustrates how the algorithm becomes incorrect if we do not handle this optimisation rules in a special manner.

**Example 5.3.3** Assume we want to solve the QBF

$$\forall x \exists y((x \vee y) \wedge (\neg x \vee \neg y) \wedge (\neg x \vee y)).$$

We use a variant of `split` which only implements the removal of truth constants, pure, and DDB by labelling. The algorithm performs as follows.

1. Replace the variable $x$ by $\bot$ and remove the truth constants. We obtain $\exists y\, y$.
2. Now we can apply the pure rule on $y$ and the formula evaluates to true.
3. We calculate the relevant variables for the labelling. The only important variable is $y$, so we set $y$ relevant.
4. Then we return to $x$ during the backtracking. As $x$ is still irrelevant, we can stop — the formula evaluates to true.

Obviously, our result is wrong — if we had considered the second subproblem (i.e., if we had set $x$ to $\top$), we would have obtained $\exists y(y \wedge \neg y)$ which is contradictory.

What went wrong in the last example? We only considered $y$ as the relevant variable. The variable $y$ was not eliminated in the usual manner (i.e., during splitting) but it was

```
(QBF, assign) simplify(φ, S) {
/* In:   QBF φ in NNF, set S of assignments */
/* Out:  simplified QBF equivalent to φ, set of assignments */

  switch(φ) {

   case (l ∧ ψ)    : (φ, S') = simplify(lunit(ψ, l, ∧, S));
   case (l ∨ ψ)    : (φ, S') = simplify(lunit(ψ, l, ∨, S));
   case (ψ1 ∧ ψ2) :
                     (ψ'1, S1) = simplify(ψ1, S);
                     (ψ'2, S2) = simplify(ψ2, S1);
                     (φ', S') = (ψ'1 ∧ ψ'2, S2);
   case (ψ1 ∨ ψ2) :
                     (ψ'1, S1) = simplify(ψ1, S);
                     (ψ'2, S2) = simplify(ψ2, S1);
                     (φ', S') = (ψ'1 ∨ ψ'2, S2);
   case (Qxψ)      : if isunit(ψ, x) then (ψ, S1) = gunit(φ, x, S);
                     if ispure(ψ, x) then (ψ, S2) = pure(φ, x, S1);
                     (ψ', S3) = simplify(ψ, S2);
                     (φ', S') = (miniscope(Qxψ'),S3)

   otherwise       : (φ', S') = (φ, S)
  }

  switch(φ') {

   case (¬⊤)    : return (⊥, S');
   case (¬⊥)    : return (⊤, S');
   case (⊥ ∧ ψ) : return (⊥, S');
   case (⊤ ∨ ψ) : return (⊤, S');
   case (⊤ ∧ ψ) : return (ψ, S');
   case (⊥ ∨ ψ) : return (ψ, S');
   otherwise    : return (φ', S');
  }

}
```

Figure 5.7: The improved function `simplify`.

removed because of a special rule. The special rule is not applicable at any time, only under certain circumstances. The question, we have to ask, is: which part of the formula and the truth assignment of which variables was responsible that we are allowed to apply this rule. For example, the assignment of which variables removed instances of $y$, such that it became unit? It was $x$. And if we label $x$ as relevant then everything goes fine as we cannot skip the second subproblem and therefore, we obtain the correct result.

```
(QBF, assign) pure(φ,x,S) {
/* In:   QBF φ, pure variable x, set S of assignments*/
/* Out:  QBF φ with x substituted by a truth constant,
         set of variable assignments */

  if (quant(x) == ∀) then {

   if (lit(φ, x) == pos) then return (φ[x/⊥], S ∪ {¬x});
   else                       return (φ[x/⊤], S ∪ {x});

  }

  if (lit(φ, x) == pos) then return (φ[x/⊤], S ∪ {x});
  else                       return (φ[x/⊥], S ∪ {¬x});
}
```

Figure 5.8: The function pure.

```
(QBF, assign) gunit(φ,x,S) {
/* In:   QBF φ, unit variable occurrence x,
         set S of assignments */
/* Out:  QBF φ where x has been substituted by a truth constant,
         set of variable assignment */

  pol = getPolarityOfUnitVar(φ, x);

  if (quant(x) == ∀) then {

  if (pol == pos) then return (φ[x/⊥], S ∪ {¬x});
  else                 return (φ[x/⊤], S ∪ {x});

  }

  if (pol == pos) then return (φ[x/⊤], S ∪ {x});
  else                 return (φ[x/⊥], S ∪ {¬x});
}
```

Figure 5.9: The function global unit.

To combine unit and pure with DDB we have to do the following.

1. We have to collect the dependencies to be able to figure out why unit or pure was applicable.

2. We have to remember why a variable was eliminated (because of common splitting or because of unit or pure).

3. When we label a variable as relevant which was removed because of unit or pure,

```
(QBF, assign) lunit(φ,l,c,S) {
/* In:   QBF φ, literal l local unit w.r.t. φ,
         operator c which connects l to φ, set S of assignments */
/* Out:  QBF φ with x substituted by a truth constant,
         set of variable assignments */

  if (c == ∧) then {

   if (l == var(l)) then return (l ∧ φ[l/⊤], S ∪ {l});
   else                  return (l ∧ φ[var(l)/⊥], S ∪ {l});

  }

  if (l == var(l)) then return (l ∨ φ[l/⊥], S ∪ {¬l});
  else                  return (l ∨ φ[var(l)/⊤], S ∪ {var(l)});

}
```

Figure 5.10: The function `local unit`.

```
QBF miniscope(φ) {
/* In:   QBF φ */
/* Out:  QBF which is euqivalent to φ */

   if (φ = ∀x(φ1 ∧ φ2)) && x ∉ φ2 then return ((∀xφ1) ∧ φ2);
   if (φ = ∀x(φ1 ∨ φ2)) && x ∉ φ2 then return ((∀xφ1) ∨ φ2);
   if (φ = ∃x(φ1 ∧ φ2)) && x ∉ φ2 then return ((∃xφ1) ∧ φ2);
   if (φ = ∃x(φ1 ∨ φ2)) && x ∉ φ2 then return ((∃xφ1) ∨ φ2);

   /* x' is a fresh variable */
   if (φ = ∃x(φ1 ∨ φ2)) then return (∃xφ1 ∨ ∃x'φ2[x/x']);

   /* x' is a fresh variable */
   if (φ = ∀x(φ1 ∧ φ2)) then return ((∀xφ1 ∧ ∀x'φ2[x/x']);

}
```

Figure 5.11: The function `miniscope`.

we have to label the variables too, which were responsible that one of the rules was applicable.

## 5.4 Discussion

In this section, we anticipate the content of Chapter 7, which is about the benchmarks. We do so to discuss the impact of the different pruning techniques. Therefore, we consider two benchmark sets: (i) encodings of reasoning on nested counterfactuals, and (ii) encodings

of the modal logic $\mathcal{K}$. We enabled/disabled the following options:

| solver name | unit | pure | labelling | rel. sets |
|---|---|---|---|---|
| qpro | | | | |
| qproup | X | X | | |
| qprol | | | X | |
| qpros | | | | X |
| qpropl | X | X | X | |
| qproups | X | X | | X |

The formuals of the nested counterfactual encodings are grouped in six sets according to their nesting depth, the encodings from the modal formulas are grouped in 18 sets, where each contains 21 formulas. The half of the set evaluates to true, the other half to false. The formulas of both sets differ strongly in their structure: the nested counterfactuals yield QBFs of a very complex quantifier structure, whereas the structure tree of the encodings from the modal formulas is linear.

Table 5.1 shows the number of solved formulas according to the result (0 is false, 1 is true, t is timeout) and Table 5.2 shows the average runtime for each set of the nested counterfactual encodings. Table 5.3 shows the number of solved formulas for each set of the modal formula encodings and Table 5.4 contains the average runtimes.

We see that the integration of the optimisation techniques has an enormous impact on the number of (un)solved formulas and on the average runtime. The variant qpro, which just implements plain chronological backtracking is not very successful. We also see that more advanced pruning techniques have their price and the enabling of the most sophisticated techniques does not alway yield the best result. In the case of the nested counterfactuals, the disabling of unit and pure in combination with dependency-directed backtracking by labelling improves the runtimes. Furthermore, dependency-directed backtracking by relevance sets is indeed better for these formulas than dependency-directed backtracking by labelling. This is not the case for the encodings of the modal formulas. DDB by labelling or DDB by relevance sets behave almost the same.

| **depth** | **truth value** | qpro | qproup | qprol | qpros | qpropl | qproups |
|---|---|---|---|---|---|---|---|
| 2 | 0 | 18 | 28 | 28 | 28 | 28 | 28 |
| 2 | 1 | 5 | 22 | 22 | 22 | 22 | 22 |
| 2 | t | 27 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 19 | 19 | 19 | 19 | 19 |
| 3 | 1 | 3 | 31 | 31 | 31 | 31 | 31 |
| 3 | t | 47 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 21 | 21 | 21 | 21 | 21 |
| 4 | 1 | 0 | 29 | 29 | 29 | 29 | 29 |
| 4 | t | 50 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 19 | 17 | 19 | 19 | 19 |
| 5 | 1 | 0 | 31 | 31 | 31 | 31 | 31 |
| 5 | t | 50 | 0 | 2 | 0 | 0 | 0 |
| 6 | 0 | 0 | 16 | 16 | 17 | 16 | 17 |
| 6 | 1 | 0 | 31 | 31 | 33 | 31 | 33 |
| 6 | t | 50 | 3 | 3 | 0 | 3 | 0 |

Table 5.1: Results of the nested counterfactual set.

| **depth** | qpro | qproup | qprol | qpros | qpropl | qproups |
|---|---|---|---|---|---|---|
| 2 | 80.30 | 0.42 | 0.59 | 0.00 | 0.60 | 0.40 |
| 3 | 98.42 | 2.17 | 1.13 | 0.00 | 1.53 | 1.05 |
| 4 | 99.98 | 4.27 | 3.76 | 0.01 | 3.04 | 2.06 |
| 5 | 99.98 | 4.64 | 7.33 | 0.04 | 3.49 | 2.34 |
| 6 | 99.98 | 11.01 | 11.84 | 0.10 | 9.65 | 6.81 |

Table 5.2: Average runtimes of nested counterfactual set.

| | qpro | qproup | qprol | qpros | qpropl | qproups |
|---|---|---|---|---|---|---|
| 1 | 2 | 9 | 2 | 3 | 11 | 11 |
| 2 | 1 | 4 | 2 | 5 | 18 | 18 |
| 3 | 2 | 7 | 4 | 5 | 7 | 7 |
| 4 | 3 | 11 | 7 | 21 | 10 | 12 |
| 5 | 6 | 21 | 7 | 8 | 21 | 21 |
| 6 | 4 | 21 | 7 | 10 | 21 | 21 |
| 7 | 0 | 21 | 3 | 9 | 21 | 21 |
| 8 | 0 | 21 | 1 | 9 | 21 | 21 |
| 9 | 2 | 21 | 5 | 5 | 21 | 21 |
| 10 | 2 | 8 | 3 | 17 | 21 | 21 |
| 11 | 3 | 14 | 3 | 8 | 12 | 12 |
| 12 | 4 | 17 | 4 | 10 | 14 | 15 |
| 13 | 5 | 21 | 6 | 10 | 21 | 21 |
| 14 | 4 | 6 | 4 | 5 | 6 | 6 |
| 15 | 2 | 21 | 3 | 3 | 21 | 21 |
| 16 | 2 | 21 | 5 | 21 | 21 | 21 |
| 17 | 61 | 4 | 1 | 2 | 4 | 4 |
| 18 | 61 | 8 | 2 | 4 | 6 | 8 |

Table 5.3: Number of solved formulas of the modal formulas.

| | qpro | qproup | qprol | qpros | qpropl | qproups |
|---|---|---|---|---|---|---|
| 1 | 90.85 | 62.28 | 90.46 | 86.78 | 54.16 | 52.26 |
| 2 | 95.21 | 81.02 | 90.48 | 79.46 | 24.58 | 23.06 |
| 3 | 90.79 | 69.28 | 84.42 | 80.64 | 68.41 | 68.46 |
| 4 | 85.85 | 50.46 | 72.01 | 0.23 | 54.51 | 45.90 |
| 5 | 72.99 | 0.00 | 68.30 | 62.75 | 0.03 | 0.03 |
| 6 | 83.27 | 0.41 | 69.91 | 57.08 | 0.73 | 0.51 |
| 7 | 99.98 | 0.00 | 89.81 | 60.54 | 0.00 | 0.00 |
| 8 | 99.98 | 0.00 | 95.76 | 64.50 | 0.00 | 0.00 |
| 9 | 90.52 | 0.01 | 80.62 | 78.98 | 0.02 | 0.02 |
| 10 | 90.73 | 67.35 | 85.84 | 32.844 | 0.00 | 0.00 |
| 11 | 86.45 | 36.72 | 85.88 | 68.65 | 46.73 | 45.63 |
| 12 | 83.36 | 25.10 | 81.49 | 58.61 | 37.00 | 35.34 |
| 13 | 77.99 | 0.20 | 74.45 | 55.73 | 0.26 | 0.23 |
| 14 | 81.13 | 72.70 | 80.94 | 76.31 | 72.32 | 72.34 |
| 15 | 90.46 | 6.22 | 86.39 | 86.45 | 7.59 | 7.75 |
| 16 | 90.86 | 0.27 | 79.40 | 1.94 | 0.28 | 0.28 |
| 17 | 95.59 | 81.56 | 95.23 | 90.95 | 81.81 | 81.53 |
| 18 | 96.98 | 66.63 | 94.78 | 85.17 | 68.32 | 68.42 |

Table 5.4: Average runtimes of the modal formulas.

# Chapter 6

# qpro's Insides

Until now, we have provided a description of the algorithms used in the solver qpro in a very formal manner (Chapter 4) to prove certain properties, and we have considered the applied techniques in an abstract way, which proposes an implementation (Chapter 5). But still one important point is missing: how can all these parts be composed in one whole program which is executable on a computer? The pseudo-code given in the previous chapter can be written down almost directly in any imperative and/or declarative programming language, but the result would be very disappointing. When a variable is assigned a truth value, the structure tree is searched for the variable occurences in a top-down manner. It is preferable to use data structures which allow for the direct access of the occurrences of a certain variable.

Anticipating the next chapter, it can be stated already here that qpro performs very competitive with respect to current state-of-the-art solvers in many situations (and sometimes even outperforms them). But this competiveness does not only result from the presented algorithms, but also from the usage of clever implementation techniques and data structures. This chapter is dedicated to the internal structure of the solver and shows how it actually works in very concrete manner (i.e., we take a closer look at the code of the solver).

Our solver qpro was developed in the programming language C and consists of about 10,000 lines of code. First prototypes were developed in the declarative languages Haskell and Prolog. The development of the prototypes was done in a very short amount of time and the solvers worked. Unfortunately, these implementation could only be considered to be proof of concepts, because they were inherently inefficient. Large QBF instances could not be dealt with. So the reimplementation was necessary to get a well performing solver. The language of choice was C because of its flexibility and because of the availability of compilers which deliver efficient code. This did not come for free — the flexibility (i.e., absolute control over the data structures) demands also for an accurate testing.

This chapter reviews the concrete components necessary to build an efficient solver like qpro. We present the used data structure, then we will consider the parsing and preprocessing. Finally, we will describe how the actual solving process runs off.

## 6.1 The Data Structure

As we do not make any assumption about the structure of the formula (except the formerly mentioned restriction that negations only occur directly in front of atoms), we have to be able to represent an arbitrary closed QBF. The restriction to conjunctive normal form allows to use a list of list as basic data structure. In the general case, a QBF is naturally represented as a structure tree. We could use a binary tree, because the disjunction and the conjunction are binary connectives by definition.

For the implementation, it is more convenient to use $n$-ary instead of binary connectives. So the data structure of a single formula is very close to the input format which is specified in the appendix.

```
struct formula {
  char type;                          /* CONJ, DISJ */
  char *val;                          /* TRUE, FALSE, UNKN */

  unsigned int relp, relh, reln;     /* infos for backjumping */
  unsigned int moved_up;
  struct formula *dep;

  struct quant *q;                    /* quantifier list */

  struct var *plits;                  /* list of pos literals */
  struct var *nlits;                  /* list of neg literals */

  struct formulas *fs;                /* list of direct subformulas */
  struct formulas *fs_fix;            /* list of subformulas
                                         of orig. formula */

  struct formulas *superfs;           /* position in superformula  */
  struct formula *super;              /* link to the super formula */
  struct formula *super_fix;          /* link to super formula
                                         of orig. formula */
};
```

Figure 6.1: The data structure for a formula.

Note that the code given in Figure 6.1 is just a simplified version of the original code which contains even more information used during the solving process.

A formula has a `type` which is either a disjunction or a conjunction (globally defined constants). The value `val` of a formula can be either `TRUE`, `FALSE`, `UNKN` which represent the evaluation status of a (sub)formula. The structure `formulas` is a doubly-linked list

```
struct quant {
 char type;                              /* EX or FA             */
 struct vinfo *var;                      /* the var's name       */
 struct quant *prev;                     /* previous quantifier  */
 struct quant *next;                     /* next quantifier      */
};
```

Figure 6.2: The data structure for a quantifier.

containing elements of type `formula`.

Note that we deal with a pointer to a truth value. This is due to the fact that the truth value of a formula can depend on the truth value of either one of its literals or of one of its subformulas. If during the solving process all literals and subformulas except one have been eliminated, the remaining literal or subformula is shifted up and inserted into the direct superformula. The shift may allow the application of simplification rules like unit, which can accelerate the solving process. But for the dependency-directed backtracking, it can be necessary to know the truth value of the formula and therefore, it is coupled to the assignment of the shifted literal or subformula.

The variables used for dependency directed backtracking will be explained later. A formula may have a doublyx-linked list of quantifiers which can be empty. An entry of this list is given in Figure 6.2. The structure and the intentions behind `vinfo` will be considered soon — it contains all necessary information about a variable.

In the code shown in Figure 6.1, `plits` and `nlits` are pointers to doubly-linked lists containing the positive and negative literals of a formula (see Figure 6.3 for details). `next` and `prev` point to the predecessor and to the successor node of a literal.

The string "`_fix`" in a variable's name indicates that this value (usually a pointer) is only set once — namely during the parsing process. Such a pointer may not be altered. These pointers are used to maintain the information about the original input formula's structure as the structure is altered during the solving process. Dependency-directed backtracking makes use of these variables to identify the relevant parts of a formula for a solved subproblem.

A literal contains a second pair of pointers, `nextv` and `prevv`, besides the two pointers for the literal list. These pointers link the literal to a list contained in the data structure `vinfo` which administrates a list of all occurrences of a variable. This is necessary in order to find them efficiently during the branching when a variable is assigned a truth value. The alternative would be a top-down search within the structure tree to locate every occurrence of a certain variable but this would be computationally very costly and should be avoided. Before the parsing, an array of type `vinfo` is created where every entry represents one variable (the name of a variable is given by an integer and is therefore used as the corresponding index in the array). The size of the array is given by the the number of variables indicated in the input file given by the qpro-format (see Appendix A for a detailed description).

Subformulas are also represented by a doubly-linked list very similar to the literal lists.

```
struct var {
  struct formula *f;                /* position in formula      */
  struct formula *f_fix;            /* pos. in orig. formula    */

  char value;                       /* TRUE, FALSE, UNKN        */
  struct vinfo *name;               /* variable name            */

  struct var *next;                 /* next literal             */
  struct var *next_fix;             /* next lit. in orig. form. */
  struct var *prev;                 /* previous literal         */

  struct var *prevv;                /* previous var in vinfo   */
  struct var *nextv;                /* next var in vinfo       */
  struct var *nextv_fix;            /* nextv in orig. formula  */

  struct formula *dep;              /* dependency for backjumping */
};
```

Figure 6.3: The data structure of a literal.

Now we have considered all components necessary to build a formula in the program. There are many other data structures used in the implementation (for example very specialised kinds of stacks), but we do not describe them here as they are just auxiliary elements to the program and not so important for understanding the basic functionality.

```
struct vinfo {
  char quant;                   /* EX, FA                       */
  struct quant *qp;             /* quantifier position          */
  unsigned int name;            /* internal var name            */

  char value;                   /* TRUE, FALSE, UNKN            */

  struct var *p;                /* list of pos. instances      */
  struct var *n;                /* list of neg. instances       */

};
```

Figure 6.4: The data structure of a variable.

## 6.2 The Preprocessing of the Formula

The main function of qpro is very compact and initates the following three processes:

1. the setting of the options,
2. the preprocessing of the formula, and
3. the actual solving.

The options enable/disable the various techniques described in the previous chapter and are mainly motivated to test the impact of switching on and off different features like unit and pure elimination. Internally, flags are set to indicate which operations have to be performed and which not. More interesting and very important is the preprocessing step of the formula. qpro's preprocessing of a formula is tightly coupled with parsing and building-up the data structure of the formula.

By preprocessing, we do not refer to the same kind of preprocessing necessary for PCNF solvers (i.e., the normalisation of the input formula). We include certain optimisation directly into the parsing process. At this early stage, we try to identify and remove tautologies and contradictions, we try to apply the unit and pure literal elimination rules as extensively as possible to reduce the formula as far as possible.

As we directly integrate those optimisations into the parsing process and as the structure of the input formula is not very complicated, we decided to refrain from the usage of a lexical analyser like FLEX and a parser generator like YACC or BISON and to implement the parsing by ourselves. Therefore, we developed the parser by the well-known technique of recursive descendance, where the single components of a formula are described by procedures which call each other accordingly.

The main function of the parser is to initialise the data structure which contains the formula. The first task is to allocate an array (a global variable which can be accessed at any time and at any point within the whole program) and which is not altered any more during the program execution (except if miniscoping is enabled where new variables are dynamically introduced). The input formula has to be checked if its syntax is correct and contains only variables named according to the name convention (i.e., they are integers ranging from 0 to the maximum value according to the input format (see Appendix A). If the input file contains an incorrect formula, an error message is provided, and the program execution is stopped. When the literal lists are build, trivial tautologies and contradictions are detected and are immediately removed. As a consequence, it sometimes can happen that very large subformulas become obsolete and can be abandoned. This can start an avalanche of further reduction opportunities and in the optimal case, the formula is evaluated without even starting the actual solving process.

Before a variable is inserted into the respecting literal list, all superformulas are searched for occurrences of this variable. If the search succeeds then the local unit literal elimination rule can be applied and the variable has either not to be inserted or the whole subformula (and if we are lucky maybe some superformulas) can be eliminated.

These reductions are not for free, much searching has to be done and many checks have to be performed. But this preparatory work proved to be crucial for the later program

execution as the number of variables (and therefore the number of splittings) can be reduced.

In the following, let $\phi$ be the input formula, and let $\phi'$ be the reduced formula. Having a small $\phi'$ is very important for the dependency-directed backtracking as, whenever a variable assignment is found which makes the whole formula true or false, it has to be checked which variables are responsible and therefore which variables have to marked as relevant. And this search for relevant variables is always done on $\phi$.

At the end of the parsing, we check for every variable whether it is pure. Note that it is only necessary during the parsing process to test all variables for purity. During the solving process, it can be done on-the-fly: whenever an occurrence of a variable is removed, it is checked whether the remaining instances of this variable are of the same polarity.

If a variable is pure, we can remove every occurrence in the formula accordingly, and we check if the application of further optimisations is possible (for example the removal of a pure literal could cause that another literal has to be shifted up in the formula and can also be removed which could cause another variable to become pure and so on). We apply all optimisation rules until no application is possible any more and the formula structure is stable. If the whole formula has not been assigned a truth value yet, the actual solving process has to be started.

## 6.3 The Basic Solving Process

The implementation of the splitting (i.e., the solving of subproblems by the assignment of variables) represents the heart of the solver. The splitting function is responsible for five tasks.

1. Select the next splitting variable or the next subformula.
2. Save the data structure of the formula for backtracking.
3. Assign the truth value if a splitting variable was selected.
4. Continue recursively with the evaluation until a truth value for the whole formula is found.
5. Restore the data structure during backtracking.

The variable selection is done accordingly to the chosen heuristic. In the simplest case, always the first variable of a formula's quantifier list is taken. Before starting to solve the subproblem by assigning the selected variable a truth value, it is necessary to store the current state of the formula. Otherwise, the other subproblem (i.e., the complementary assignment of the variable) has to be considered and the formula has to be restored as it has been at this point during the program execution before the first subproblem has been considered. If the quantifier list of the current formula is empty, then a subformula has to be chosen, which is processed next.

Generally speaking, there are two ways to save the formula:

1. copy the whole formula, and

2. track the changes.

The first proposed method is not very sophisticated and can be implemented easily, but obviously it is very memory consuming. In the worst case, the memory consumption for storing the QBF almost doubles at every branching step. So we implemented the second method: every change of the formula's structure is reported and recorded in the so-called "UNDO-stack". Every stack entry contains a type with the information what has to be undone and one or more pointers which reference the locations in the formula where the changes have taken place and where they should be undone.

Before solving a new subproblem, a special item is pushed on the stack to indicate that a new block of tracking elements will follow. When calling the UNDO-function (in practice it is a macro) then all elements are popped from the stack and are processed until this special item is reached. The elements left on the stack are subject to an other call of the UNDO-function. With the increase of the solvers functionality also the numbers of operations to undo grows. The implementation of the current UNDO-function is able to distinguish between 27 operations.

When a variable has been chosen and the special item to indicate a new block has been pushed on the UNDO-stack, every occurrence of the variable can be replaced by the according truth value. As it would be very inefficient to directly search for the variable in the formula tree in a top-down manner, now we make use of the information stored in the `vinfo` data structure. Recall that `vinfo` contains a list of pointers to all occurrences of the variable. The only action we have to perform is to process the lists and replace the literals by true or false. Instantly, we remove them from the formula and also propagate the values to sub- and/or superformulas if possible. We also apply pruning techniques like unit and pure literal elimination if possible. In contrast to the application of those rules during the parsing process, we do not check for every variable if the rules can be used. We assume that all possible applications of the rules have been made before, but when we change the structure of the formula, we get new candidates for the application of those rules.

For example, if a literal is shifted up into its direct subformula because it is the only component left in a formula, it is possible that

1. it has been shifted directly in front of the quantifier block containing its quantification. Then global unit can be applied;

2. at least the local unit rule becomes applicable.

If all reductions have been performed, and no truth value for the formula has been obtained then the splitting process must be repeated until a truth value is found.

If a truth value has been found, the changes on the formula structure can be undone and depending on the type of quantifier and the calculated truth value, the second subproblem (i.e., the complementary assignment of the variable) has to be considered or can be omitted.

## 6.4 Dependency Directed Backtracking

Even though the idea behind dependency-directed backtracking is simple, the implementation is not so straightforward as it may seem at the first sight. The important aspect is to maintain the correctness of the solver and therefore, it is necessary to carefully keep track of the dependencies of the variable elimination (i.e., to remember if one literal has been removed because another variable has been assigned a truth value).

The function where the splitting is performed has to be extended as follows:

1. Set a variable to *irrelevant* before it is assigned a truth value.
2. Check if the variable is still irrelevant when returning from backtracking.

This setting and checking of the (ir)relevance of a variable is the easiest part about the integration of the dependency-directed backtracking. Two more aspects are left to do:

1. extend the `simplify` function to get the order on the elimination times of the variables;
2. implement a function, which calculates the set of relevant variables.

Especially when pruning techniques like unit and pure are involved, when whole subformulas are removed by one single literal, or when subformulas are shifted upwards in the formula tree, many more details have to be considered for getting the correct set of relevant variables. Every variable has a pointer `dep` to the formula where the reason can be found why it has been removed (note that the pointer is not set if the variable has been assigned a truth value in the usual manner).

The set of relevant variables is calculated top-down with respect to the formula tree of the original input formula (the pointer containing "`_fix`" provides the necessary information).

The difference between the implementation of dependency-directed backtracking by labelling and the implementation of dependency-directed backtracking by relevance sets is not so severe. In the second case, efficient operations on sets have to be implemented, in the other case only a flag has to be set if a variable is irrelevant.

# Chapter 7

# Experimental Evaluation

In this chapter, we report on QBF encodings "from practice" and about the behaviour of the previously presented solver qpro compared to current state-of-the-art systems when evaluating those formulas. With other words, we dedicate this chapter to the testing of our implementation. The motivation behind testing is twofold: (1) we want to verify whether qpro works correctly by experiments and (2) we are interested how qpro performs with respect to other state-of-the-art solvers.

One of the most severe problems a solver programmer has to face during the development process is to detect and correct programming errors. It is possible to prove soundness and completeness of the algorithm on which the implementation is based (what we have done previously). But how can thousands of lines of code be shown to be correct? When using an imperative language with side effects and with pointers, this task becomes very hard and cumbersome. And a buggy solver is absolutely useless: for example consider a program, which always returns true. In some cases, this solver yields correct results, in others, it does not but it always needs a constant amount of time and space. This drastic example illustrates very clearly the fact that an error (here it concerns an algorithmic and not an implementational bug) can prune the search space enormously. The elimination of this problem would result in an implementation with a very different behaviour with respect to time and space consumption.

In the last years, the QBF community has collected a large number of various benchmark sets, which are available at the `QBFLIB` [53]. These benchmarks are used in the QBF solver evaluations and competitions [65, 64, 72] and by many solver developers to test and to evaluate their systems.

Roughly speaking, there are two different kinds of formulas: (i) randomly generated instances and (ii) formulas stemming from encodings of "real-world" applications. As most solvers only process formulas in PCNF, the random formulas are artificially generated PCNF formulas parametrised by the number of variables, the number of quantifier alternations, the size of the clause, etc. The real-wold problem instances are, unfortunately, translated to the `qdimacs` format, which is the standard format for PCNF formulas.

Therefore, the sets are not interesting for our purposes. To run our tests, we had to create our on benchmark sets and we adapted QBFs encodings from the literature except

the encodings of modal logic $\mathcal{K}$ in QBFs, which were kindly provided to us by G. Pan.

We have chosen three different sets of formulas:

1. Encodings of reasoning with nested counterfactuals.
2. Encodings of formulas from modal logic $K$.
3. Encodings of correspondence-tests from answer-set programming.

The formulas of these benchmarks are originally not encoded in PCNF and they differ in the complexity of their structure and depth of quantifier prefix. We will describe the nested counterfactuals and the correspondence check problems in a very detailed manner, as we created this benchmark sets by ourselves. Translated to `qdimacs`, we submitted them to the `QBFLIB`. A subset of the formulas is now publicly available at the `QBFLIB` website [53] and they were used in the QBF solver evaluation. For the encoding of the modal formulas, we refer to [80].

For the comparison of our implementation qpro to other solvers, we have chosen the well-established systems QuBE−BJ [51] (v1.2), sKizzo [10] (v0.4), semprop [68] (rel. 24/02/02), and quantor [16] (rel. 25/04/01). We selected those solvers because they are publicly available, showed to be very competitive in previous solver evaluations, and so far, they did not deliver wrong results on our tests. Moreover, QuBE−BJ and semprop implement the dependency-directed backtracking technique similar to qpro. The solvers sKizzo and quantor try to extract information about the quantifier dependencies in the original formula from the quantifier prefix of the formula already transformed into PCNF. As already mentioned, we did not include the solver Qubos [5] because we encountered some problems in our pretests but we hope to be able to compare qpro and Qubos as soon as a bug fix is available.

For all solvers, we used their predefined standard options and for qpro, we enabled the standard simplification techniques together with dependency-directed backtracking by relevance sets.

As all solvers except qpro are only capable to process formulas in PCNF, we applied the following testing strategy: Given a QBF $\phi$ from the benchmark set not in PCNF, we (i) provided $\phi$ as input to qpro; (ii) translated $\phi$ into PCNF and provided the outcome as input to the other solvers. For normal form transformation, we used the tool `qst` [96], which applies structure preserving normal form transformation (see Chapter 4) and which implements fourteen different prenexing strategies. Which strategy we have chosen for each test set, is described below. We ran our test on an Intel Xeon 3 GHz with 4 GB of RAM unless reported differently below. We set the timeout to 100 seconds for each formula.

## 7.1 Nested Counterfactuals

A *counterfactual* is an expression of the form $A > B$ which can informally be read as the conditional query "If $A$ held, would $B$ necessarily hold too?". $A$ is called the *premise* and

$B$ is called the *conclusion* of the counterfactual. If either the premise or the conclusion of a counterfactual contains a counterfactual too, we speak about a *nested counterfactual* (NCF). For illustration consider the following example from [39].

**Example 7.1.1** Let a theory $\mathcal{T}$ encode the structure and the functionality of a car's electric system. We are interested in the following question: "If the headlight switch was turned on ($h$) and the light did not shine ($\neg s$), would it shine ($s$) if the fuse protecting the light ($f$) was changed by a new one?".

In the context of nested counterfactuals, a theory $\mathcal{T}$ encodes the background knowledge in terms of purely propositional formulas in our case. A nested counterfactual can express the question above as follows:

$$(h \wedge \neg s) > (f > s)$$

Eiter and Gottlob showed in [39] that the problem of deciding *right nested counterfactuals* (i.e., only the conclusion of a counterfactual contains further counterfactuals) is $\Pi^P_{k+2}$-complete for a nesting depth bound by $k$ and PSPACE-complete if the nesting depth is unbound. Therefore, the number of quantifier alternations in the QBF encoding is parametrised by the nesting depth of the corresponding nested counterfactual. We presented a translation of right nested counterfactuals to QBFs in [35] and we used them as a case study to investigate the impact of different prenexing strategies on the solving process. A refinement and extension of this work can be found in [96].

As the encoding of right nested counterfactuals results in QBFs of a complex structure which is far from being in PCNF, this type of formulas are very interesting for this thesis as they can show clearly if the avoidance of the normal form transformation and the usage of a solver like qpro results in better runtimes.

**Definition 7.1.1 (Right Nested Counterfactuals)**
The set $\mathcal{C}$ of *right nested counterfactuals* is defined inductively as follows:

1. if $A, B$ are formulas from propositional logic then $A, B, (A > B) \in \mathcal{C}$;
2. if $A$ is a formula from propositional logic and $N \in \mathcal{C}$ then $(A > N) \in \mathcal{C}$;
3. if $N \in \mathcal{C}$ then $(\neg N) \in \mathcal{C}$.

Note that we write nested counterfactuals of the form $(\neg(A > N))$ as $(A \not> B)$. Further, we assume the operators $>$ and $\not>$ to be right associative. As mentioned before, we presented a translation of right nested counterfactuals to QBFs in [35].

**Definition 7.1.2 (Nesting Depth)**
The *nesting depth* $\mathsf{nd}(N)$ of a nested counterfactual $N$ is defined as follows.

$$\mathsf{nd}(N) = \begin{cases} \mathsf{nd}(A > N') & \text{if } N = (A \not> N'); \\ 1 + \mathsf{nd}(N') & \text{if } N = (A > N'); \\ 0 & \text{otherwise.} \end{cases}$$

**Definition 7.1.3 (Maximal Consistent Subtheories)**
Let $\mathcal{T}$ be a theory. The set of maximal $A$-consistent subtheories of $\mathcal{T}$ is defined by

$$\mathsf{mct}(A, \mathcal{T}) = \{\mathcal{S} \mid \mathcal{S} \subseteq \mathcal{T}, \mathcal{S} \nvdash \neg A, \mathcal{S} \subseteq \mathcal{S}' \subseteq \mathcal{T} \rightarrow \mathcal{S}' \vdash \neg A\}.$$

**Definition 7.1.4 (Semantics of Nested Counterfactuals)**
The *evaluation function* $v_{ncf}$ for a nested counterfactual is defined as follows.

- $v_{ncf}(\mathcal{T}, A \not> N) = \begin{cases} \mathsf{T} & \text{if } v_{ncf}(\mathcal{T}, A > N) = \mathsf{F}; \\ \mathsf{F} & \text{otherwise.} \end{cases}$
- $v_{ncf}(\mathcal{T}, A > N) = \begin{cases} \mathsf{T} & \text{if } \mathsf{nd}(N) > 0 \text{ and for all } \mathcal{S} \in \mathsf{mct}(A, \mathcal{T}) \\ & v_{ncf}(\mathcal{S} \cup A, N) = \mathsf{T}; \\ \mathsf{T} & \text{if } \mathsf{nd}(N) = 0 \text{ and for all } \mathcal{S} \in \mathsf{mct}(A, \mathcal{T}) \\ & \mathcal{S} \cup A \vdash N; \\ \mathsf{F} & \text{otherwise.} \end{cases}$

**Definition 7.1.5 (Validity of Nested Counterfactuals)**
A nested counterfactual $N$ is *valid* w.r.t. a theory $\mathcal{T}$, if $v_{ncf}(\mathcal{T}, N) = \mathsf{T}$. Otherwise, it is said to be unsatisfiable w.r.t. the theory $\mathcal{T}$.

**Theorem 7.1.1 (The Complexity of Nested Counterfactuals)**

1. The problem of evaluating a right nested counterfactual $N$ with $\mathsf{nd}(N) \leq k$ is $\Pi^P_{k+2}$-complete.
2. The problem of evaluating a right nested counterfactual is PSPACE-complete if the nesting depth is unbound.

**Proof.** The proof can be found in [39]. □

### 7.1.1 The Encoding

In the following, we describe how reasoning about nested counterfactuals can be expressed in terms of QBFs. Therefore, we introduce two shorthands. The expression $\mathcal{S} \leq \mathcal{T}$ abbreviates $\{\phi_i \rightarrow \varphi_i \mid 1 \leq i \leq n\}$ and is identified with $\bigwedge_{i=1}^{n}(\phi_i \rightarrow \varphi_i)$. Moreover, $\mathcal{S} < \mathcal{T}$ stands for $(\bigwedge_{i=1}^{n}(\phi_i \rightarrow \varphi_i)) \wedge \neg(\bigwedge_{i=1}^{n}(\varphi_i \rightarrow \phi_i))$.

**Definition 7.1.6 (Encodings of NCFs in QBFs)**
Let $N$ be a nested counterfactual with $\mathsf{nd}(N) = k$ and of the structure

$$A_0 \succ_0 (A_1 \succ_1 (\ldots \succ_{k-1} (A_k \succ_k A_{k+1}) \ldots)),$$

where $\succ_i \in \{>, \not>\}$. Let $\mathcal{T} = \{\varphi_1, \ldots, \varphi_n\}$ be a propositional theory. Let $\mathcal{S} = \{\phi_1, \ldots, \phi_n\}$ be an indexed set of formulas.

Furthermore, let $S_i = \{s_{i,1}, \ldots, s_{i,n+i}\}$ and $U_i = \{u_{i,1}, \ldots, u_{i,n+i}\}$ be sets of new atoms $(0 \leq i \leq k)$.

Now we define $\mathcal{M}_0[\mathcal{T}, N] = S_0 \leq \mathcal{T}$ and, for $0 < j \leq k$,

$$\mathcal{M}_j[\mathcal{T}, N] = S_j \leq \Big(\mathcal{M}_{j-1}[\mathcal{T}, N] \cup \{A_{j-1}\}\Big).$$

For $0 \leq i \leq k$, we furthermore define

$$\Phi_i = \exists V_i\Big(\mathcal{M}_i[\mathcal{T}, N] \wedge A_i\Big) \wedge \forall U_i\Big((S_i < U_i) \rightarrow \forall V_i((\mathcal{M}_i[\mathcal{T}, N])[S_i/U_i] \rightarrow \neg A_i)\Big),$$

where each $V_i$ denotes the set of atoms occurring in $\mathcal{T} \cup \{A_0, \ldots, A_i\}$, as well as

$$\Psi_i = \begin{cases} \forall S_i(\Phi_i \rightarrow \Psi_{i+1}) & \text{if } \succ_i = >; \\ \exists S_i(\Phi_i \wedge \neg\Psi_{i+1}) & \text{if } \succ_i = \not> , \end{cases}$$

with $\Psi_{k+1} = \forall W((\mathcal{M}_k[T, N] \wedge A_k) \rightarrow A_{k+1})$, where $W$ contains all atoms from $V_k$ and those occurring in $\psi$. Then the desired encoding, $\mathcal{E}[\mathcal{T}, N]$, is given by $\Psi_0$.

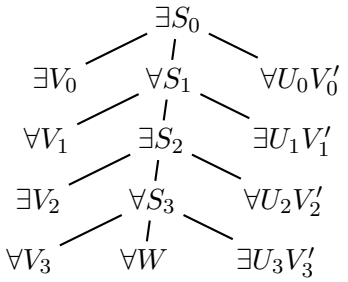
**Theorem 7.1.2 (Correctness of the Encoding)**
Let $\mathcal{T}$ be a propositional theory and let $N$ be a nested counterfactual. It holds that

$$v_{ncf}(\mathcal{T}, N) = v(\mathcal{E}[\mathcal{T}, N]).$$

For illustration, we consider the encoding $\mathcal{E}[\mathcal{T}, N]$ with $\mathsf{nd}(N) = 3$ which can be written as

$$\exists S_0(\Phi_0 \wedge \forall S_1(\Phi_1 \rightarrow \exists S_2(\Phi_2 \wedge \forall S_3(\Phi_3 \rightarrow \forall W((\mathcal{M}_3[\mathcal{T}, N] \wedge p_3) \rightarrow q))))).$$

The quantifier ordering for the QBF $\mathcal{E}[\mathcal{T}, N]$ is graphically represented as follows:




The quantifier tree of this nested counterfactual is quite complex, even though the nesting depth is very small. This fact makes the nested counterfactuals very interesting and promising formulas for our tests.

Obviously, there are multiple possibilities to linearise the quantifier dependency tree to obtain a quantifier prefix. For illustration, we apply four different (and correct) shifting strategies and we obtain a quantifier prefix of the form $\exists P_0 \forall P_1 \exists P_2 \forall P_3 \exists P_4 \psi$ where the sets $P_i$ are given as follows:

| # | Strategy | $P_0$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ |
|---|---|---|---|---|---|---|
| 0 | $\exists^{\downarrow}\forall^{\downarrow}$ | $S_0$ | $S_1$ | $S_2$ | $S_3U_0V_0'V_1U_2V_2'V_3W$ | $V_0U_1V_1'V_2U_3V_3'$ |
| 1 | $\exists^{\uparrow}\forall^{\uparrow}$ | $S_0V_0$ | $S_1U_0V_0'V_1$ | $S_2U_1V_1'V_2$ | $S_3U_2V_2'V_3W$ | $U_3V_3'$ |
| 2 | $\exists^{\uparrow}\forall^{\downarrow}$ | $S_0V_0$ | $S_1$ | $S_2U_1V_1'V_2$ | $S_3U_0V_0'V_1U_2V_2'V_3W$ | $U_3V_3'$ |
| 3 | $\exists^{\downarrow}\forall^{\uparrow}$ | $S_0$ | $S_1U_0V_0'V_1$ | $S_2$ | $S_3U_2V_2'V_3W$ | $V_0U_1V_1'V_2U_3V_3'$ |

The second column of the table contains a short description for each strategy. The expression $\mathsf{Q}^{\uparrow}$ (resp. $\mathsf{Q}^{\downarrow}$) indicates that the quantifier $\mathsf{Q}$ is shifted as left-most (resp. as right-most) within the quantifier prefix as possible. A detailed description of prenexing may be found in Chapter 4.

### 7.1.2 The Generator

We developed a simple random generator for right nested counterfactuals. The generator accepts the following input parameters:

- the number of clauses $n$ in the theory,
- the number of literals per clause $m$ in the theory,
- the total number of variables $o$, and
- the nesting depth $k$.

The generator produces a theory and a single nested counterfactual of the following structure:

1. $o_0 \not> (o_1 \not> (\cdots \not> (o_k \not> o_{k+1}))\ldots))$ for an odd nesting depth;
2. $o_0 > (o_1 \not> (\cdots \not> (o_k \not> o_{k+1}))\ldots))$ for an even nesting depth.

The $o_i$ is a variable randomly selected from a set of $o$ variables. The QBF generated from a nested counterfactual with an odd nesting depth is in $\Sigma^P_{k+2}$ and a QBF generated from a nested counterfactual with an even nesting depth is in $\Pi^P_{k+2}$.

The associated theory is a set of $n$ clauses with length $m$ consisting of the literals chosen from the same set as the variables in the nested counterfactuals. The literals are negated with a probability of 0.5.

### 7.1.3 The Experiments

We present two different sets of benchmarks containing right nested counterfactuals. The first set is a part of the set used by Zolda [96] to explore the impact of different prenexing

| **Nested Counterfactuals** | number of instances | 10000 |
|---|---|---|
| nesting depth | | 6 |
| number of clauses | | 6–30 |
| number of variables | | 3–22 |
| **Encoding in QBF** | | |
| number of clauses | average | 2293 |
| number of variables | average | 1106 |

Table 7.1: Information about the benchmarks by Zolda.

strategies. Some information about the 1000 formulas contained in this set is given in Table 7.1. It is important to note that this set contains very easy formulas as well as very hard formulas even though the nesting depth of the underlying nested counterfactuals is restricted to 6. The reason why some formulas cannot be solved within the time bounds is due to wide range of the number of chosen variables and on the number of clauses. Most formulas can be solved within milliseconds, whereas others have to be abandoned due to the timeout.

| solver | strategy | timeouts | average runtimes | median | 75% | 80% |
|---|---|---|---|---|---|---|
| qpro | - | 445 | 6.67 | 0.03 | 0.16 | 0.43 |
| QuBE−BJ | $\exists\uparrow\forall\downarrow$ | 447 | 5.64 | 0.04 | 0.19 | 0.36 |
| | $\exists\downarrow\forall\uparrow$ | 642 | 7.33 | 0.03 | 0.17 | 0.35 |
| semprop | $\exists\downarrow\forall\uparrow$ | 2189 | 24.45 | 0.08 | 31.84 | 100 |
| | $\uparrow$ | 2270 | 25.34 | 0.08 | 47.77 | 100 |

Table 7.2: Results of the benchmark set by Zolda.

As this test was performed on a cluster of older machines (namely Intel Pentium with 256 MB of RAM), it was not possible to run all of the presented solvers as the available memory was insufficient and the most time was spent by writing to the swap space. Therefore, we excluded sKizzo and quantor. The solver qpro performs very well even if only small memory resources are available. If the formula is in memory and the data structure is built up, almost no memory is allocated during the solving process any more.

Table 7.2 shows the number of timeouts and the average runtime of the solvers qpro, QuBE−BJ, and semprop. Zolda [96] ran semprop and QuBE−BJ on each formula to test the fourteen different shifting strategies. We show only the best two shifting strategies for the two normal form solvers. Without going into details, $\exists\uparrow\forall\downarrow$ denotes the strategy where the existential quantifiers are shifted as leftmost in the prefix as possible whereas the universal quantifiers are shifted as rightmost as possible. The strategy $\exists\downarrow\forall\uparrow$ is defined dually and when $\uparrow$ is applied, all quantifiers are shifted as leftmost as possible (i.e., at the highest possible position in the linearised quantifier dependency tree).

Our solver qpro is the solver which solved the most formulas, i.e., with the smallest number of timeouts. Nevertheless, qpro is not the solver with the best average runtime. This is QuBE−BJ with the strategy $\exists\uparrow\forall\downarrow$. To analyse the behaviour in more detail, we take a look at the quantiles to find out at which time QuBE−BJ outperforms qpro. A quantile indicates at which time a certain percentage of the formulas has been solved. E.g., the 50% quantile (also called median) is the time when half of the formulas has been solved. The value of the median is almost the same for both solver (the 0.01 – 0.03 second difference is neglectable due to inprecisions of the CPU clock in the area of milliseconds). At the 75% quantile, both solvers still perform equally well but the point where QuBE−BJ shakes off qpro is around the 80% quantile. This indicates that there are a few formulas which both solvers can hardly solve anymore. These are the formulas which influence the average runtime such that QuBE−BJ is better than qpro even though qpro is able to solve more formulas.

Interestingly, the second best shifting strategy for QuBE−BJ increases the number of

formulas which could not be solved by about two hundred. The average solving time is increased by about two seconds. But the increase of average runtime is only caused by very few formulas — if we take a look at the 80% quantile, we see that the two strategies are almost equally good at this point.

The solver semprop is far behind. It is not able to solve about one quarter of the formulas and this also can be seen in the average runtime. The choice of a different shifting strategy influences the runtimes minimal. The other two solvers are capable to solve 80% of the formals within less than a second whereas the 80% quantile value for semprop is already the timeout.

| **Nested Counterfactuals** | number of instances | 250 |
|---|---|---|
| | true | 60% |
| | false | 40% |
| number of variables | | 8 |
| number of clauses | | 20 |
| clause size | | 3 |
| nesting depth | | 2 – 6 |
| instances per nesting depth | | 50 |
| **Encoding in QBF** | | |
| number of variables | nesting depth | |
| | 2 | 183 |
| | 3 | 245 |
| | 4 | 309 |
| | 5 | 375 |
| | 6 | 433 |
| number of variables after NFT | nesting depth | |
| | 2 | 464 |
| | 3 | 600 |
| | 4 | 786 |
| | 5 | 934 |
| | 6 | 1132 |

Table 7.3: Information about the second NCF benchmark set.

The second set of nested counterfactuals has been chosen very differently. The clause size and the number of variables are fixed but the nesting depth of the counterfactuals ranges from 2 to 6 (see Table 7.3). Therefore, the resulting QBFs have a quantifier depth ranging from 4 to 8. The parameters have been chosen to create hard formulas for normal form solvers and to test how qpro behaves in this case.

Table 7.4 shows the number of timeouts and the average runtimes for each solver. This time we only include the outcome for the best strategy for each solver. And again, we want to emphasise that qpro does not need the preprocessing of the formula (i.e., it is not necessary to select any shifting strategy).

For this benchmark set, qpro is clearly the winner. Our solver qpro is able to solve all formulas whereas the others have enormous problems with the increasing quantifier depth.

| | number of timeouts | | | | | average runtimes (in sec.) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | qpro | QuBE−BJ ∃↑∀↓ | semprop ↑ | sKizzo ∃↓∀↑ | quantor ∃↓∀↑ | qpro | QuBE−BJ ∃↑∀↓ | semprop ↑ | sKizzo ∃↓∀↑ | quantor ∃↓∀↑ |
| 4 | 0 | 1 | 8 | 9 | 31 | **0.41** | 5.10 | 39.39 | 22.30 | 86.27 |
| 5 | 0 | 3 | 10 | 13 | 30 | **1.06** | 9.35 | 28.69 | 32.72 | 88.62 |
| 6 | 0 | 4 | 34 | 26 | 42 | **2.06** | 11.66 | 69.01 | 57.20 | 82.87 |
| 7 | 0 | 8 | 32 | 28 | 41 | **2.34** | 20.45 | 63.34 | 60.06 | 82.87 |
| 8 | 0 | 12 | 45 | 38 | 41 | **6.81** | 32.08 | 79.72 | 78.55 | 90.47 |

Table 7.4: Results of the second NCF benchmark set.

The "expand and resolve" solver quantor is the only solver which cannot deal with those formulas in any depth. In the tests it could be observed that quantor solves a formula either within the first few seconds or never within the timeout.

But also the other solvers have their problems with these formulas. Because the complex quantifier dependency tree of nested counterfactuals allows for many ways to obtain the prefix and due to the complex formula structure, qpro benefits enormously from the omission of the normal form transformation. Finally, Figure 7.1 illustrates the behaviour of the solvers graphically.

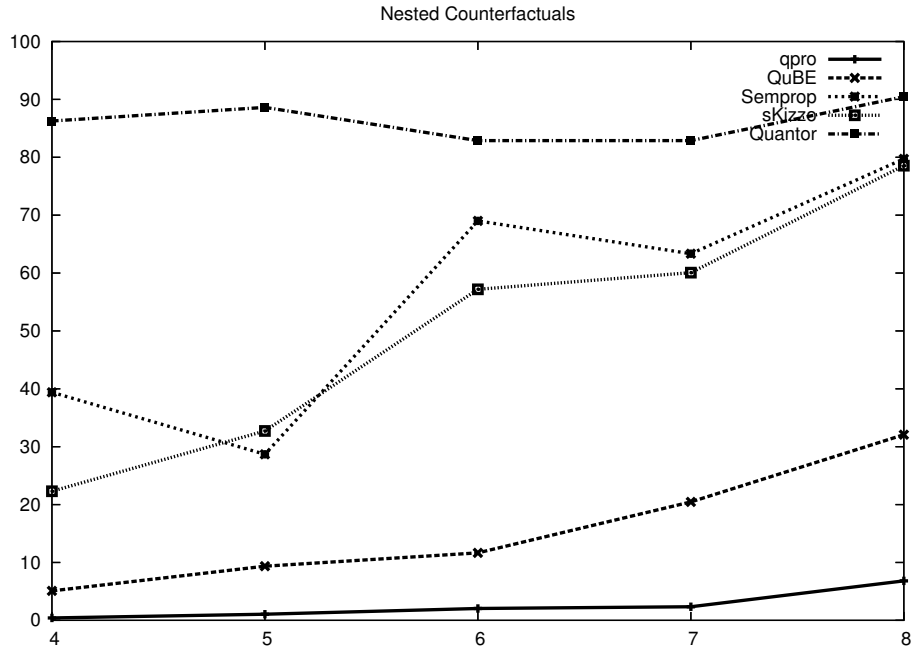



Figure 7.1: Runtimes for nested counterfactuals.

## 7.2 Correspondence Checking in Answer-Set Programming

Logic programming under the answer-set semantics (ASP) [44] represents an important and significant paradigm for encoding and solving a wide range of problems in AI like planning, diagnosis, inheritance reasoning, etc. (for an overview see [43]). A very relevant issue in answer-set programming consists in supporting the check of the equivalence of two different programs which should encode the same problem. In [76], we presented a tool to reduce the problem of correspondence checking to the satisfiability problem of QBFs.

In this section, we provide a short introduction of answer-set programming, introduce some notions of equivalence and give an overview about the encoding.

**Definition 7.2.1 (Propositional Disjunctive Logic Program)**
A *propositional disjunctive logic program* (DLP) is a finite set of rules of the form

$$a_1 \vee \cdots \vee a_l \leftarrow a_{l+1}, \ldots, a_m, not\, a_{m+1}, \ldots, not\, a_n,$$

$n \geq m \geq l \geq 0$, where all $a_i$ are propositional atoms from some fixed universe $\mathcal{U}$ and *not* denotes default negation. Rules of the form $a \leftarrow$ are called *facts* and are also identified by the atom $a$, itself. If all atoms occurring in a program $P$ are from a given set $A \subseteq \mathcal{U}$ of atoms, we say that $P$ is a program *over* $A$.

A rule $r$ of the form $a_1 \vee \cdots \vee a_l \leftarrow a_{l+1}, \ldots, a_m, not\, a_{m+1}, \ldots, not\, a_n$ is said to be *true* under an interpretation $I$, symbolically $I \models r$, iff $\{a_1, \ldots, a_l\} \cap I \neq \emptyset$, whenever it holds that $\{a_{l+1}, \ldots, a_m\} \subseteq I$ and $\{a_{m+1}, \ldots, a_n\} \cap I = \emptyset$. If $I \models r$ holds, then $I$ is also said to be a *model* of $r$. An interpretation $I$ is a model of a program $P$ iff $I \models r$, for all $r \in P$.

Following Gelfond and Lifschitz [44], an interpretation $I$ is an *answer set* of a program $P$ iff it is a minimal model of the *reduct* $P^I$, resulting from $P$ by

- deleting all rules containing default negated atoms *not a* such that $a \in I$; and
- deleting all default negated atoms in the remaining rules.

The collection of all answer sets of a program $P$ is denoted by $\mathcal{AS}(P)$.

**Definition 7.2.2 (Notions of Equivalences)**
Two programs, $P$ and $Q$, are *ordinarily equivalent* iff $\mathcal{AS}(P) = \mathcal{AS}(Q)$. $P$ and $Q$ are *strongly equivalent* iff, for any program $R$, $\mathcal{AS}(P \cup R) = \mathcal{AS}(Q \cup R)$.

In abstracting from these equivalence notions, Eiter et al. [40] introduce the notion of a *correspondence problem* which allows to specify, on the one hand, a *context*, i.e., a class of programs used to be added to the programs under consideration and, on the other hand, the relation that has to hold between the collections of answer sets of the extended programs. Following Eiter et al. [40], we focus here on correspondence problems where the context is parametrised in terms of alphabets and the comparison relation is a projection of the standard subset or set-equality relation.

**Definition 7.2.3 (Inclusion and Equivalence Problems)**
Let $\mathcal{U}$ denote a universe. A correspondence problem, $\Pi$ over $\mathcal{U}$ is a quadruple of form $(P, Q, \mathcal{P}_A, \rho_B)$, where $P, Q \in \mathcal{P}_\mathcal{U}$. The sets $A, B \subseteq \mathcal{U}$ contain atoms.

By $\rho_B$ we denote either $\subseteq_B$ or $=_B$, which are defined as follows: for any sets $\mathcal{S}, \mathcal{S}'$, $\mathcal{S} \subseteq_B \mathcal{S}'$ iff $\mathcal{S}|_B \subseteq \mathcal{S}'|_B$, and $\mathcal{S} =_B \mathcal{S}'$ iff $\mathcal{S}|_B = \mathcal{S}'|_B$, where $\mathcal{S}|_B = \{I \cap B \mid I \in \mathcal{S}\}$.
We say that $\Pi$ *holds* iff, for all $R \in \mathcal{P}_A$, $(\mathcal{AS}(P \cup R), \mathcal{AS}(Q \cup R)) \in \rho_B$. We call $\Pi$ an *equivalence problem* if $\rho_B$ is given by $=_B$, and an *inclusion problem* if $\rho_B$ is given by $\subseteq_B$, for some $B \subseteq \mathcal{U}$.

Note that $(P, Q, \mathcal{P}_A, =_B)$ holds iff $(P, Q, \mathcal{P}_A, \subseteq_B)$ and $(Q, P, \mathcal{P}_A, \subseteq_B)$ jointly hold.

**Theorem 7.2.1 (Complexity of Correspondence Checking)**
Given programs $P$ and $Q$, sets of atoms $A$ and $B$, and $\rho \in \{\subseteq_B, =_B\}$, deciding whether a correspondence problem $(P, Q, \mathcal{P}_A, \rho)$ holds is:

1. $\Pi_4^p$-complete, in general;
2. $\Pi_3^p$-complete, for $A = \emptyset$;
3. $\Pi_2^p$-complete, for $B = \mathcal{U}$;
4. coNP-complete for $A = \mathcal{U}$.

While Case 1 provides the result in the general setting, for the other cases we have the following: Case 2 amounts to *ordinary equivalence with projection*, i.e., the answer sets of two programs relative to a specified set $B$ of atoms are compared. Case 3 amounts to *strong equivalence relative to* $A$ and includes, as a special case, viz. for $A = \emptyset$, *ordinary equivalence*. Finally, Case 4 includes *strong equivalence* (for $B = \mathcal{U}$) as well as strong equivalence with projection.

The $\Pi_4^p$-hardness result shows that, in general, checking the correspondence of two programs cannot (presumably) be efficiently encoded in terms of ASP, which has its basic reasoning tasks located at the second level of the polynomial hierarchy (i.e., they are contained in $\Sigma_2^p$ or $\Pi_2^p$).

### 7.2.1 The Experiments

In our experiments, we consider two different reductions from inclusion problems to QBFs, $\mathsf{S}[\cdot]$ and $\mathsf{T}[\cdot]$, where $\mathsf{T}[\cdot]$ can be seen as an explicit optimisation of $\mathsf{S}[\cdot]$. Recall that equivalence problems can be decided by the composition of two inclusion problems. Thus, a composed encoding for equivalence problems is easily obtained via a conjunction of two particular instantiations of $\mathsf{S}[\cdot]$ (or $\mathsf{T}[\cdot]$). The QBF encodings are described in [73].

For our tests, we created 1000 of small QBFs which can be solved very easily by any of the QBF solvers. Then we encoded these QBFs in terms of answer-set correspondence checks. We translated those encodings back to QBFs which resulted in large and not so easily solvable formulas. For this, we used the tool $\mathsf{cc}\top$ developed by Oetsch at our departement [75]. This approach to generate test formulas might seem a little bit complicated but now we know for sure whether a complex QBF evaluates to true or to false

| **problem instances** | | 1000 |
|---|---|---|
| | true | 465 |
| | false | 535 |
| number of atoms in P, Q | | 40 |
| rules in program P | | 620 |
| rules in program Q | | 280 |
| atoms in set A | | 16 |
| atoms in set B | | 16 |
| **encoding S** | | |
| atoms in corresponding QBF | | 200 |
| | after NFT | 2851 |
| clauses after NFT | average | 6577 |
| | minimum | 6391 |
| | maximum | 6631 |
| **encoding T** | | |
| atoms in corresponding QBF | | 152 |
| | after NFT | 2555 |
| clauses after NFT | average | 6217 |
| | minimum | 6031 |
| | maximum | 6271 |

Table 7.5: Information about the correspondence-check encodings.

which is not always given. By evaluating the original, simple QBF, we can verify the output of the solvers. Information about the formulas is shown in Table 7.5.

Table 7.6 shows the number of timeouts and the average runtimes ordered by the different shifting strategies (two are possible) and the different encodings. Figures 7.2 and 7.3 illustrate the average runtimes as a diagrams. As quantor could not deal with these formulas, we did not include this solver in the pictures.

This test set illustrates the enormous influence of the shifting strategy. With the correct shifting strategy and the optimised encoding, the solvers suddenly perform very well. If the wrong strategy with the unoptimised encoding is chosen then all solvers have the biggest problems with the formulas. But not so qpro. Again we emphasise that our solver

| | number of timeouts | | | | | average runtimes (in sec.) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | qpro | QuBE−BJ | semprop | sKizzo | quantor | qpro | QuBE−BJ | semprop | sKizzo | quantor |
| S↑ | – | 842 | 117 | 527 | 1000 | – | 87.34 | 81.82 | 74.67 | 100 |
| S↓ | – | 90 | 6 | 0 | 1000 | – | 43.21 | 27.60 | **2.67** | 100 |
| T↑ | – | 43 | 38 | 0 | 1000 | – | 20.85 | 54.86 | 2.97 | 100 |
| T↓ | – | 0 | 0 | 0 | 1000 | – | 9.26 | 16.90 | **1.13** | 100 |
| S | 29 | – | – | – | – | 33.37 | – | – | – | – |
| T | 0 | – | – | – | – | 17.87 | – | – | – | – |

Table 7.6: Results from answer-set correspondence checking.

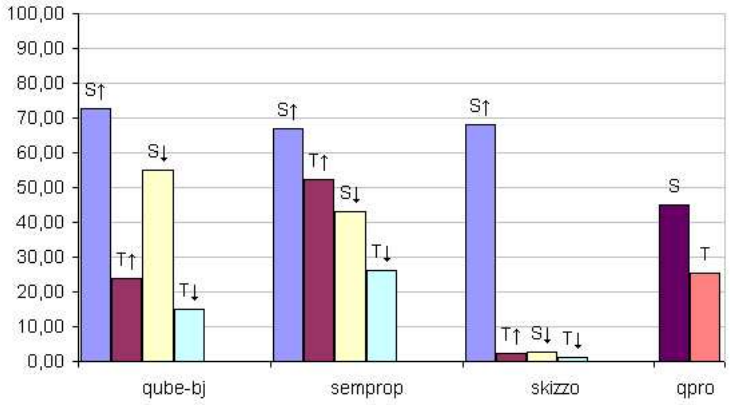



Figure 7.2: Runtimes from answer-set correspondence checking for true instances.

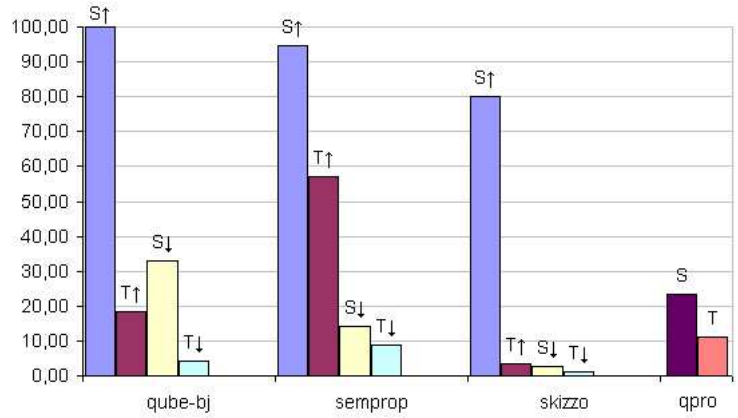



Figure 7.3: Runtimes from answer-set correspondence checking for false instances.

is independent of any shifting strategy. But more remarkable is the fact, that qpro is not as depended as the other solvers on the optimisations performed during the encoding. Overall, qpro's performance is competitive with the other solvers especially considering the low number of quantifier blocks.

We also considered a second benchmark set of encodings from answer set correspondence checking problems. Based on randomly generated $(2, \exists)$-QBFs according to Model A [46] the reduction of every formula following Eiter and Gottlob [38] yields a program which posses an answer set iff the original QBF is valid. To simulate a "sloppy" programmer a randomly selected line of each program is extinguished and the "buggy" program is compared to the "correct" one in terms of ordinary equivalence.

Figure 7.4 shows the average running times parameterised according to the number of variables of the original input QBF which ranges from 10 to 24. For each data point 100 instances where generated (details about the benchmarks can be found in [78]). Due to the fact that the underlying QBFs are set on the second level of the polynomial hierarchy,

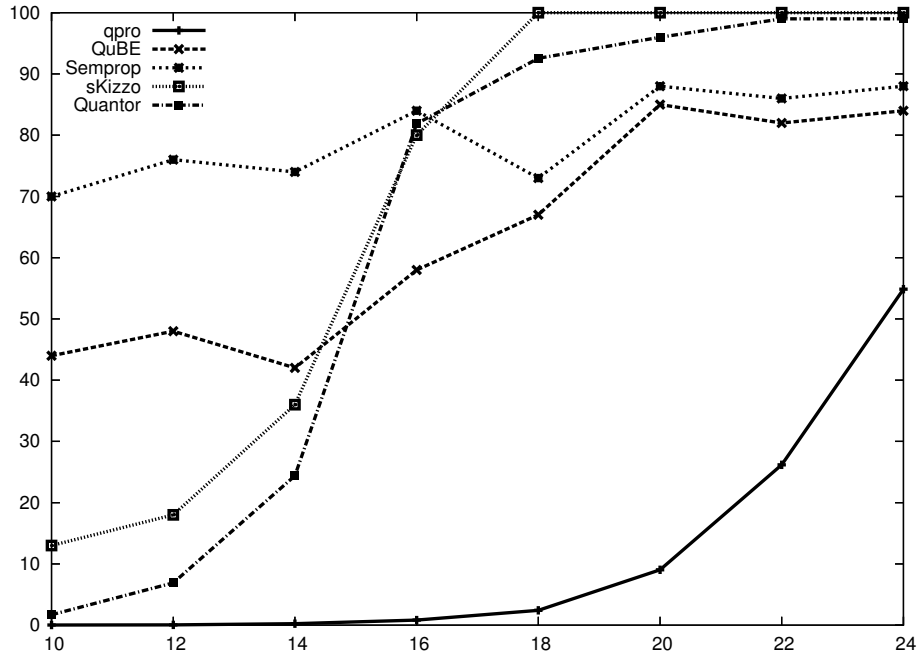



Figure 7.4: Runtimes of answer-set correspondence checks.

there is only one prenexing strategy.

Interestingly, qpro performs best although the linear quantifier tree. This is due to the complex formula structure on the one side, on the other side it is because of the dual implementation of DDB for false and for true subproblems. The separated runtimes are shown in the Figures 7.5 and 7.6. It is interesting to observe the different behaviours of the solvers depending on the truth value of the formula. For example, QuBE−BJ and semprop are absolutely not able to handle the true instances of this benchmark set in contrast to quantor which even outperforms sKizzo in this case. On the other hand, quantor performs not so well on the false instances in contrast to QuBE−BJ.

| | qpro | QuBE−BJ | semprop | sKizzo | quantor |
|---|---|---|---|---|---|
| 10 | **0.01** | 0.29 | 56.00 | 12.27 | 1.15 |
| 12 | **0.02** | 1.49 | 65.06 | 18.24 | 4.01 |
| 14 | **0.07** | 5.35 | 69.35 | 33.17 | 17.73 |
| 16 | **0.23** | 25.48 | 86.53 | 100 | 83.59 |
| 18 | **0.50** | 46.10 | 65.74 | 100 | 100 |
| 20 | **1.95** | 100 | 90.34 | 100 | 100 |
| 22 | **6.11** | 100 | 86.95 | 100 | 100 |
| 24 | **14.81** | 100 | 92.43 | 100 | 100 |

Figure 7.5: Runtimes for correspondence checks which evaluate to F.

| | qpro | QuBE−BJ | semprop | sKizzo | quantor |
|---|---|---|---|---|---|
| 10 | **0.05** | 100 | 100 | 14.71 | 2.37 |
| 12 | **0.17** | 100 | 100 | 18.45 | 10.05 |
| 14 | **0.51** | 100 | 100 | 48.70 | 35.38 |
| 16 | **1.54** | 100 | 100 | 100 | 100 |
| 18 | **4.85** | 100 | 100 | 100 | 100 |
| 20 | **15.07** | 100 | 100 | 100 | 100 |
| 22 | **46.23** | 100 | 100 | 100 | 100 |
| 24 | **100** | 100 | 100 | 100 | 100 |

Figure 7.6: Runtimes for correspondence checks which evaluate to F.

## 7.3 Modal Formulas by Pan and Vardi

The benchmark set due to Pan and Vardi [80, 81] contains encodings of the modal logic $\mathcal{K}$. This modal logic extends propositional logic by the two unary propositional operators $\Box$ and $\Diamond$. The semantics of formulas in $\mathcal{K}$ is given by a *Kripke structure* $K = \langle D, W, R, L\rangle$. The letter $D$ denotes a non-empty set, the domain $W$ is the set of possible worlds, $R \subseteq W^2$ is the accessibility relation on worlds and the function $L : W \to 2^D$ maps every world to a subset of the domain, i.e., $L(w)$ is interpreted as the domain of world $w$ ($w \in W$).

A Kripke structure $K = \langle D, W, R, L\rangle$ satisfies a formula $\mu$ (written as $K, w \models \mu$) if, for every $w \in W$,

- $K, w \models \Box\mu$ if for all $(w, v) \in R, K, v \models \mu$ holds;
- $K, w \models \Diamond\mu$ if for some $(w, v) \in R, K, v \models \mu$ holds.

The semantics of the other connectives is the same as in propositional logic. The problem of checking the satisfiability of formulas from the modal logic $\mathcal{K}$ is PSPACE-complete.

In [80], Pan and Vardi provide a translation of $\mathcal{K}$ to QBF. They use their BDD-based modal solver $\mathcal{KBDD}$ to generate models which they encode in terms of QBFs.

Even though the structure of the resulting QBFs is quite complex with respect to nestings of conjunctions and disjunctions, the quantifier dependency tree is linear. This means that there exists only one single quantifier prefix for each formula and that all different prenexing strategies yield this prefix. Considering this property of the encoding, it will be very interesting to observe the behavior of qpro with respect to the other solvers to watch how qpro handels formulas with a linear quantifier dependency tree.

### 7.3.1 The Experiments

The benchmark set consists of 18 scalable classes from modal logics $\mathcal{K}$ constructed by Heuerding and Schwendimann [58]. The modal properties are nested to construct successively harder formulas. Each class consists of 21 formulas. The resulting QBFs are also

grouped in the same classes. The names of the formula classes indicate the truth values of its formulas: the formulas contained in classes whose names start with an odd number evaluate to true, the other formulas evaluate to false. The quantifier depth of the formulas scales linearly within a class and the formulas successively get harder.

Table 7.7 shows the the number of timeouts as well as the average runtime for each solver grouped by the different formula classes. Obviously, some sets are very easy for all solvers whereas others have an average runtime which is quite close to the timeout (e.g., the average runtime of set 17-* for quantor is 82.57 seconds). For the solver quantor, the whole benchmark set is very difficult to handle. There are only two classes of which all its members could be solved without a timeout.

The solvers QuBE−BJ and semprop perform best on those formulas in most cases. If the formula class is a class easy to solve — like the classes 05-* to 10-* — qpro is among the best solvers. As soon as the classes contain harder formulas which cannot be solved within milliseconds, qpro is thrown back but still the number of timeouts remains moderate.

| | number of timeouts | | | | | average runtimes (in sec.) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | qpro | QuBE−BJ | semprop | sKizzo | quantor | qpro | QuBE−BJ | semprop | sKizzo | quantor |
| 01-* | 10 | 7 | 7 | 0 | 18 | 52.34 | 37.43 | 38.29 | **12.65** | 67.90 |
| 02-* | 2 | 0 | 0 | 0 | 18 | 23.90 | 6.92 | **0.08** | 0.89 | 71.67 |
| 03-* | 14 | 10 | 14 | 17 | 17 | 70.30 | **52.92** | 68.27 | 80.72 | 80.69 |
| 04-* | 9 | 8 | 0 | 0 | 17 | 45.06 | 45.22 | **0.01** | 0.18 | 79.79 |
| 05-* | 0 | 0 | 0 | 7 | 13 | **0.01** | **0.00** | 0.11 | 37.25 | 63.09 |
| 06-* | 0 | 0 | 0 | 0 | 12 | **0.00** | **0.00** | 0.32 | 0.29 | 54.90 |
| 07-* | 0 | 0 | 0 | 0 | 8 | **0.00** | **0.00** | **0.00** | 0.17 | 43.10 |
| 08-* | 0 | 0 | 0 | 4 | 8 | **0.00** | **0.00** | **0.01** | 0.12 | 40.24 |
| 09-* | 0 | 0 | 0 | 1 | 2 | **0.00** | **0.00** | **0.01** | 2.33 | 15.09 |
| 10-* | 0 | 0 | 0 | 0 | 0 | **0.00** | **0.00** | **0.01** | **0.00** | **0.07** |
| 11-* | 9 | 8 | 0 | 0 | 16 | 45.09 | 43.63 | **0.14** | 0.22 | 76.02 |
| 12-* | 6 | 6 | 0 | 0 | 15 | 34.28 | 32.67 | **0.07** | 0.21 | 72.10 |
| 13-* | 0 | 4 | 0 | 0 | 1 | 0.22 | 26.74 | **0.02** | 1.00 | 6.79 |
| 14-* | 13 | 13 | 12 | 10 | 11 | 72.26 | 64.57 | 58.89 | **50.13** | 56.22 |
| 15-* | 0 | 0 | 0 | 0 | 0 | 3.90 | **0.01** | 0.39 | **0.04** | **0.05** |
| 16-* | 0 | 0 | 0 | 0 | 19 | 0.27 | 0.28 | **0.01** | 0.57 | 73.96 |
| 17-* | 16 | 0 | 12 | 4 | 20 | 78.74 | **0.72** | 61.29 | 47.83 | 82.57 |
| 18-* | 13 | 0 | 0 | 6 | 18 | 65.63 | 0.26 | **0.04** | 41.46 | 79.95 |

Table 7.7: Number of solved formulas and average runtimes of modal Logic $\mathcal{K}$.

To conclude, our solver qpro performs very well considering the fact that the quantifier tree of the non-prenex formulas are simply linear lists and that no information about the structure of the quantifier dependencies is lost during prenexing. The shifting only increases the scope of the quantifiers but in this case the impact is minimal.

## 7.4 Summary

We tested our solver qpro on thousands of formulas from three different sets of benchmarks. On the one hand, we wanted to test the correctness of our solver empirically, on the other hand, we wanted to know how qpro performs compared to current PCNF solvers.

The results of the tests are very encouraging. They indicate clearly that not only the prenexing but also the transformation to CNF has an enormous impact on the runtimes of the solvers. The tests strongly confirm our assumption that the omission of the normal form transformation can positively influence the solving process. We could show that our implementation is very efficient and even though we have to build more complex data structures than the other solvers, the more powerful optimisation techniques compensate the overhead due to the more complex structure in many cases.

# Chapter 8

# Conclusion

From a historical point of view, the most efficient and successful theorem provers and solvers attacking the evaluation problems of various formalisms are those, which rely on (prenex) clausal normal form. This holds e.g., for propositional, as well as for the more expressive classical first-order logic. Following this approach, many efficient QBF solvers have been implemented during the last decade. Although QBFs are still far from gaining the attention propositional logic is experiencing at the moment for large scale industrial domains, much effort is spent on researching QBFs. Those formulas present themselves as promising host language for the efficient encoding of problems located at some level of the polynomial hierarchy which includes many reasoning tasks from artificial intelligence and knowledge representation.

Usually, real world instances of QBFs are not directly available in prenex conjunctive normal form (PCNF). Before they can be passed on to the solvers for evaluation, they have to be transformed to an equivalent formula of the demanded structure which is easier to handle and to process. Naturally, this simplification comes at a price. Not only is the formula structure disrupted and information that could have been used for the solving process is lost, but the size of the formula and the number of propositional variables increase as well. Further, the transformation to PCNF is not deterministic in general.

In previous work [35, 96], it was empirically shown that all these arguments against PCNF transformation are well founded in practice and that the PCNF transformation has an enormous influence on the behaviour of the utilised solver. We were able to confirm and extend the results in this thesis. Consequently, we developed a solver independent of the preprocessing tools which transform the input formulas to PCNF.

In this thesis, we presented the non-prenex, non-conjunctive normalform solver qpro, which we completely developed from scratch. We first proposed two decision procedures to evaluate quantified Boolean formulas: (i) the connection calculus and (ii) a generalisation of the DPLL algorithm. The connection calculus is a proof-theoretical characterisation of the validity of QBFs, whereas DPLL is a depth-first search method on the formula tree used in most implementations of current state-of-the-art solvers. The special feature of our variant of DPLL, on which we focused, is that we abandoned the assumption that the formula to solve is available in PCNF. We only imposed one restriction on the formula structure: negation signs are only allowed to appear in front of propositional variables

which is far less restrictive. We could show that, in contrast to prenexing and the transformation to conjunctive normal form, the transformation to the negation normal form is harmless, but very advantageous for the implementation.

We analysed DPLL and proved important properties like soundness and correctness using a sequent calculus style notation. Then we refined the basic algorithm and included optimisation techniques like dependency-directed backtracking to prune the search space.

Finally, we presented our implementation qpro and compared our solver to publicly available state-of-the-art systems in numerous tests. We included formulas from different areas like reasoning on nested counterfactuals, encodings of a modal logic and answer-set correspondence checks, in our set of benchmarks. The results were very encouraging: qpro performed very competitive in comparison to well established systems. As soon as the structure of the formulas of the test sets became more complicated, qpro easily outperformed the other solvers and was able to find solutions to formulas, which were out of scope for any available solver until now. Even for formulas with a structure close to PCNF and with few quantifier alternations, qpro provided solutions of formulas which have never been solved.

Nevertheless, many improvements and extensions are possible, like the refinement of the data structures on the implementational side and the integration of further pruning techniques like more specialised kinds of learning. It would also be very interesting to compare qpro with solvers based on BDDs, which are not available at the moment, and then maybe consider an integration of BDD techniques into DPLL.

To conclude, non-normal form solvers will gain a lot of importance in the future and that the solver development will follow this direction.

# Appendix A

# qpro's Input Format

As almost all current solvers are only able to process formulas in normal form, the quasi standard input syntax is — not surprisingly — restricted to formulas in PCNF. This format, called `qdimacs` [2], allows for the representation of a QBF in ASCII. One file which contains exactly one QBF is divided into three sections:

- the preamble,
- the quantifier prefix, and
- matrix.

The *preamble* contains arbitrary many, human readable comments which are ignored by the solver followed by exactly one *problem line* of the form

```
p cnf VARIABLES CLAUSES
```

In the expression above, `VARIABLES` and `CLAUSES` are integers, where `VARIABLES` stands for the maximal numbers of variables occurring in the formula and accordingly, `CLAUSES` denotes the number of the formula's clauses.

Information about the quantifier blocks is encoded in the *quantifier prefix*. Every line of the prefix starts either with an "`e`" (for an existential quantifier) or an "`a`" (for an universal quantifier) or an "`r`" (for an random quantifier) which indicates the type of quantification. The letter is followed by a sequence of variables separated by a space and terminated by "`0`". The variables are represented as positive integers between 1 and the number `VARIABLES` provided in the problem line. Note that each number may occur only once in the prefix.

The last section — the *matrix* — contains the clauses (one clause per line). Every line contains a sequence of integers, now positive as well as negative ones. Positive integers stand for positive literals, whereas negative integers represent negative literals.

For our purposes `qdimacs` is insufficient as we need a format which is general enough to represent QBFs in NNF. The input syntax used by `qpro` is given in Figure A.1.

*qbf* → `(`*comment*`)*` *problemLine formula*

*comment* → `c` *string* `\n`

*problemLine* → `QBF` *posInteger* `\n`

*formula* → *qformula* `\n` |
*cformula* `\n` |
*dformula* `\n`

*qformula* → `q \n`
`(`*aQBlock* | *eQBlock*`)`
`(`*dformula* | *cformula*`)`
`/q \n`

*aQBlock* → *aQuant* `(`*eQuant aQuant* `)* (`*eQuant*`)?`
*eQBlock* → *eQuant* `(`*aQuant eQuant* `)* (`*aQuant*`)?`

*aQuant* → `a` *posInteger posIntSeq*
*eQuant* → `e` *posInteger posIntSeq*

*dformula* → `d \n`
*posIntSeq*
*posIntSeq*
`(`*qformula* | *cformula*`)*`
`/d \n`

*cformula* → `c \n`
*posIntSeq*
*posIntSeq*
`(`*qformula* | *dformula*`)*`
`/c \n`

*posIntSeq* → *posInteger posIntSeq* `| \n`

Figure A.1: The input syntax for qpro.

Our format is not very different from `qdimacs`. In fact, it can be seen as a generalisation of it. The main difference is that `qpro`'s format has only two sections instead of three, namely the preamble and the formula. There is no special area for the quantifier blocks as they are included directly in the formula.

According to our format, a QBF file may start with human-readable comments which begin with "`c`" and which are ignored by `qpro`. The line starting with `QBF` which is followed by a positive integer is the first line of real relevance. The number indicates how many variables occur in the formula. If the file respects the format then it can be assumed that the names of the variables are numbers between 1 and the given integer.

In the next line the actual formula starts. It can either be a conjunction, a disjunction or a quantified formula. We consider conjunction and disjunction as n-ary connectives and we do not allow that a direct subformula of a conjunction is a conjunction and that the direct subformula of a disjunction is a disjunction. The same holds for a QBF with a quantifier as the main connective: it may contain an arbitrary number of alternating quantifier blocks, but it is not allowed to have a direct subformula with starts with a quantifier. We denote a formula with a quantifier as main connective a *qformula*, accordingly disjunction and conjunction are called *dformula* and *cformula*.

A *qformula* starts with a line containing the single letter "`q`" and ending with a line "`/q`". For the disjunction (resp. for the conjunction) the "`q`" is exchanged by a "`d`" (resp. a "`c`").

The lines after the "`q`" contain the quantifications. They start either with an "`a`" or an "`e`" indicating whether quantifier is universal ("`a`") or existential ("`e`"). A line starting with "`e`" may not be followed by a line starting with an "`a`" and vice versa. The letters at the beginning of the line are followed by an ordered list of positive integers - the variable names which are separated by blanks. So the "`q`" and the closing "`/q`" define the scope of a variable. After the alternating quantifier blocks a *cformula* or a *dformula* follows.

The conjunctions and disjunctions, i.e., *cformulas* and *dformulas*, are constructed similarly: the first two lines contain ordered lists of positive integers. In the first line all positive literals are collected whereas we find all the negative literals in the second one. If a formula contains no positive or negative integers, the lines remain blank. Then arbitrary many subformulas (which are not literals) follow before the formula is closed either by "`/c`" or by "`/d`".

By this means we are able to describe any formula in NNF. An example is given in Figure A.2.

```
QBF
10
q
a 2
e 3 4
c
4

q
a 5 6
d


c
5
6
/c
q
e 7 8 9 10
c
7 8 9 10

/c
/q
/d
/q
/c
/q
QBF
```

Figure A.2: The QBF $\forall a_2 \exists e_2 e_4 (e_4 \wedge \forall a_5 a_6 ((a_5 \wedge \neg a_6) \vee \exists e_7 e_8 e_9 e_{10} (e_7 \wedge e_8 \wedge e_9 \wedge e_{10})))$ in qpro's input format.

# Bibliography


[1] P. B. Andrews. Theorem Proving via General Matings. *Journal of the Association for Computing Machinery*, 28(2):193–214, 1981.

[2] Anonymous. Quantified Boolean Formulas Satisfiability — Suggested Format. Available in electronic form at `http://www.qbflib.org/Draft/qDimacs.ps.gz`, July 2001.

[3] W. Aspray. *Computing before Computers*. Iowa State University Press, 1990.

[4] G. Audemard and L. Sais. A Symbolic Search Based Approach for Quantified Boolean Formulas. In F. Bacchus and T. Walsh, editors, *Proceedings of the 8th International Conference on the Theory and Applications of Satisfiability Testing (SAT 2005)*, volume 3569 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2005.

[5] A. Ayari and D. A. Basin. QUBOS: Deciding Quantified Boolean Logic Using Propositional Satisfiability Solvers. In M. Aagaard and J. W. O'Leary, editors, *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2002)*, volume 2517 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 2002.

[6] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[7] M. Baaz, U. Egly, and A. Leitsch. Normal Form Transformations. In J. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 5. The MIT Press, 2001.

[8] M. Baaz and A. Leitsch. On Skolemization and Proof Complexity. *Fundamenta Informaticae*, 20(4):353–379, 1994.

[9] BDDLib: A BDD Library with Extensions for Sequential Verification. Software available at `http://www.cs.cmu.edu/~modelcheck/bdd.html`.

[10] M. Benedetti. Evaluating QBFs via Symbolic Skolemization. In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2004)*, volume 3452 of *Lecture Notes in Computer Science*, pages 285–300. Springer, 2005.

[11] M. Benedetti. Extracting Certificates from Quantified Boolean Formulas. In L. P. Kaelbling and Al. Saffiotti, editors, *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 47–53. Professional Book Center, 2005.

[12] M. Benedetti. Quantifier Trees for QBFs. In F. Bacchus and T. Walsh, editors, *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, volume 3569 of *Lecture Notes in Computer Science*, pages 378–385. Springer, 2005.

[13] M. Benedetti. sKizzo: a Suite to Evaluate and Certify QBFs. In R. Nieuwenhuis, editor, *Proceedings of the 20th International Conference on Automated Deduction (CADE 2005)*, volume 3632 of *Lecture Notes in Computer Science*, pages 369–376. Springer, 2005.

[14] P. Besnard, T. Schaub, H. Tompits, and S. Woltran. Representing Paraconsistent Reasoning via Quantified Propositional Logic. In *Inconsistency Tolerance*, volume 3300 of *Lecture Notes in Computer Science*, pages 84–118. Springer, 2005.

[15] W. Bibel. *Automated Theorem Proving.* Vieweg, 1987.

[16] A. Biere. Expand and Resolve. In H. H. Hoos and D. G. Mitchell, editors, *Proceedings of the 7th International Conference on the Theory and Applications of Satisfiability Testing (SAT 2004)*, volume 3542 of *Lecture Notes in Computer Science*, pages 59–70. Springer, 2005.

[17] R. Bosch. Theorembeweisen für QBF und die Anwendung zum Planen. Master's thesis, Universität Ulm, 1998.

[18] T. Boy de la Tour. An Optimality Result for Clause Form Translation. *Journal of Symbolic Computation*, 14(4):283–302, 1992.

[19] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[20] S. Buss. *Handbook of Proof Theory.* Elsevier, 1998.

[21] M. Cadoli, T. Eiter, and G. Gottlob. Complexity of Nested Circumscription and Abnormality Theories. In B. Nebel, editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 169–174. Morgan Kaufmann, 2001.

[22] M. Cadoli, A. Giovanardi, and M. Schaerf. An Algorithm to Evaluate Quantified Boolean Formulae. In B. Buchanan and R. Uthurusamy, editors, *Proceedings of the 15th National Conference on Artificial Intelligence and the 10th Innovative Applications of Artificial Intelligence Conference (AAAI 1998/IAAI 1998)*, pages 262–267. AAAI Press / The MIT Press, 1998.

[23] M. Cadoli, Andrea Giovanardi, and Marco Schaerf. Experimental Analysis of the Computational Cost of Evaluating Quantified Boolean Formulae. In M. Lenzerini,

editor, *Proceedings of the 5th Congress of the Italian Association for Artificial Intelligence: Advances in Artificial Intelligence (AI*IA97)*, volume 1321 of *Lecture Notes in Computer Science*, pages 207–218. Springer, 1997.

[24] M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi. An Algorithm to Evaluate Quantified Boolean Formulae and its Experimental Evaluation. *Journal of Automated Reasoning*, 28(2):101–142, 2002.

[25] S. Cook. The Complexity of Theorem-Proving Procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing (STOC71)*, pages 151–158, 1971.

[26] M. Davis, G. Logeman, and D. Loveland. A Machine Program for Theorem-Proving. *Communications of the Association for Computing Machinery*, 5:394–397, 1962.

[27] M. Davis and H. Putnam. Computational Methods in the Propositional Calculus. Technical report, Rensselaer Polytechnic Institute, 1958. Unpublished Report.

[28] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[29] U. Egly. New Calculi for QBFs. Unpublished Draft.

[30] U. Egly. On Different Structure-preserving Translations to Normal Form. *Journal of Symbolic Computation*, 22(2):121–142, 1996.

[31] U. Egly. Quantifiers and the System KE: Some Surprising Results. In G. Gottlob, E. Grandjean, and K. Seyr, editors, *Proceedings of the 12th International Workshop for Computer Science Logic (CSL 1998)*, volume 1584 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 1999.

[32] U. Egly, T. Eiter, V. Klotz, H. Tompits, and S. Woltran. Experimental Evaluation of the Disjunctive Logic Programming Module of the System QUIP. In F. Bry, U. Geske, and D. Seipel, editors, *Proceedings of the Workshop on Logic Programming (WLP 2000)*, pages 113–122, 2000.

[33] U. Egly, T. Eiter, H. Tompits, and S. Woltran. Solving Advanced Reasoning Tasks Using Quantified Boolean Formulas. In R. Engelmore and H. Hirsh, editors, *Proceedings of the 17th Conference on Artificial Intelligence and of the 12th Conference on Innovative Applications of Artificial Intelligence (AAAI 2000/IAAI 2000)*, pages 417–422. AAAI Press / The MIT Press, 2000.

[34] U. Egly and T. Rath. On the Practical Value of Different Definitional Translations to Normal Form. In M. A. McRobbie and J. K. Slaney, editors, *International Conference on Automated Deduction (CADE 1996)*, volume 1104 of *Lecture Notes in Computer Science*, pages 403–417. Springer, 1996.

[35] U. Egly, M. Seidl, H. Tompits, S. Woltran, and M. Zolda. Comparing Different Prenexing Strategies for Quantified Boolean Formulas. In E. Giunchiglia and A. Tacchella, editors, *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2004.

[36] U. Egly, M. Seidl, and S. Woltran. A Solver for QBFs in Nonprenex Form. In G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, editors, *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI 2006)*, pages 477–481. IOS Press, 2006.

[37] T. Eiter and G. Gottlob. On the Complexity of Propositional Knowledge Base Revision, Updates, and Counterfactuals. *Artifical Intelligence*, 57(2-3):227–270, 1992.

[38] T. Eiter and G. Gottlob. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. In *Annals of Mathematics and Artificial Intelligence 15(3/4)*, pages 289–323, 1995.

[39] T. Eiter and G. Gottlob. The Complexity of Nested Counterfactuals and Iterated Knowledge Base Revisions. *Journal of Computer and System Sciences*, 53(3):497–512, 1996.

[40] T. Eiter, H. Tompits, and S. Woltran. On Solution Correspondences in Answer Set Programming. In L. P. Kaelbling and A. Saffiotti, editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 97–102. Professional Book Center, 2005.

[41] R. Feldmann, B. Monien, and S. Schamberger. A Distributed Algorithm to Evaluate Quantified Boolean Formulas. In R. Engelmore and H. Hirsh, editors, *Proceedings of the 17th Conference on Artificial Intelligence and of the 12th Conference on Innovative Applications of Artificial Intelligence (AAAI 2000/IAAI 2000)*, pages 285–290. AAAI Press / The MIT Press, 2000.

[42] M. Fitting. *First–Order Logic and Automated Theorem Proving.* Springer, 1990.

[43] M. Gelfond and N. Leone. Logic Programming and Knowledge Representation - The A-Prolog Perspective. *Artificial Intelligence*, 138(1-2):3–38, 2002.

[44] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:364–385, 1991.

[45] I. Gent and A. G. D. Rowley. Encoding Connect-4 using Quantified Boolean Formulae. Technical Report APES-68-2003, APES Research Group, 2003. Available in electronic form at `http://www.dcs.st-and.ac.uk/~apes/apesreports.html`.

[46] I. Gent and T. Walsh. Beyond NP: The QSAT Phase Transition. In R. Uthurusamy and B. Hayes-Roth, editors, *Proceedings of the 16th Conference on Artificial Intelligence and of the 11th Conference on Innovative Applications of Artificial Intelligence (AAAI 1999/IAAI 1999)*, pages 648–653. AAAI Press / The MIT Press, 2000.

[47] M. GhasemZadeh. A New Algorithm for the QSAT Problem Based on ZBDDs. Dissertation, University of Potsdam, 2004.

[48] M. GhasemZadeh, V. Klotz, and C. Meinel. Embedding Memoization to the Semantic Tree Search for Deciding QBFs. In G. I. Webb and X. Yu, editors, *Proceedings of the 17th Australian Joint Conference on Artificial Intelligence (AUSAI 2004)*, volume 3339 of *Lecture Notes in Computer Science*, pages 681–693. Springer, 2004.

[49] M. L. Ginsberg. Counterfactuals. *Journal of Artifical Intelligence*, 30(1):35–79, 1986.

[50] E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin. Evaluating Search Heuristics and Optimization Techniques in Propositional Satisfiability. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR 2001)*, volume 2083 of *Lecture Notes in Computer Science*, pages 347–362. Springer, 2001.

[51] E. Giunchiglia, M. Narizzano, and A. Tacchella. An Analysis of Backjumping and Trivial Truth in Quantified Boolean Formulas Satisfiability. In F. Esposito, editor, *Proceedings of the 7th Congress of the Italian Association for Artificial Intelligence, Advances in Artificial Intelligence (AI*AI 2001)*, volume 2175 of *Lecture Notes in Computer Science*, pages 111–1121. Springer, 2001.

[52] E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for Quantified Boolean Logic Satisfiability. In B. Nebel, editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 275–281. Morgan Kaufmann, 2001.

[53] E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2001. http://www.qbflib.org.

[54] E. Giunchiglia, M. Narizzano, and A. Tacchella. System Description: QuBE A system for Deciding Quantified Boolean Formulas Satisfiability. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR 2001)*, volume 2083 of *Lecture Notes in Computer Science*, pages 364–369. Springer, 2001.

[55] E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for Quantified Boolean Logic Satisfiability. In S. Chien and J. Riedl, editors, *Proceedings of the 18th National Conference on Artificial Intelligence and the 13th Innovative Applications of Artificial Intelligence Conference (AAAI 2002/IAAI 2002)*, pages 649–654. AAAI Press / The MIT Presss, 2002.

[56] E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantifier Structure in Search Based Procedures for QBFs. In G. G. E. Gielen, editor, *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2006)*, pages 812–817. European Design and Automation Association, Leuven, Belgium, 2006.

[57] G. Gottlob. Complexity Results for Nonmonotonic Logics. *Journal of Logic and Computation*, 2(3):397–425, 1992.

[58] A. Heuerding and S. Schwendimann. A Benchmark Method for the Propositional Modal Logic K. Technical report, Universität Bern, 1996.

[59] H. Kleine Büning, M. Karpinski, and A. Flögel. Resolution for Quantified Boolean Formulas. *Information and Computation*, 117(1):12–18, 1995.

[60] H. Kleine Büning and T. Lettmann. *Aussagenlogik: Deduktion und Algorithmen*. B. G. Teubner, 1994.

[61] J. Krajicek. *Bounded Arithmetic, Propositional Logic and Complexity Theory*. Cambridge University Press, 2004.

[62] C. Kreitz and J. Otten. Connection-based Theorem Proving in Classical and Non-classical Logics. *Journal of Universal Computer Science*, pages 88–112, 1999.

[63] R. E. Ladner. The Computational Complexity of Provability in Systems of Modal Propositional Logic. *SIAM Journal on Computing*, 6(3):467–480, 1977.

[64] D. Le Berre, M. Narizzano, L. Simon, and A. Tacchella. The Second QBF Solvers Comparative Evaluation. In H. H. Hoos and D. G. Mitchell, editors, *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, volume 3542 of *Lecture Notes in Computer Science*, pages 376–392. Springer, 2005.

[65] D. Le Berre, L. Simon, and A. Tacchella. Challenges in the QBF Arena: The SAT'03 Evaluation of QBF Solvers. In E. Giunchiglia and A. Tacchella, editors, *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *Lecture Notes in Computer Science*, pages 468–485. Springer, 2004.

[66] L. C. Lee. Representation of Switching Circuits by Binary Decision Diagrams. *Bell Syst. Tech. Journal*, 38:985–999, 1959.

[67] A. Leitsch. *The Resolution Calculus*. Springer, 1997.

[68] R. Letz. Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In U. Egly and C. G. Fermüller, editors, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2002)*, volume 2381 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2002.

[69] J. Łukasiewicz and A. Tarski. Untersuchungen über den Aussagenkalkül. *Comptes Rendus Séances Société des Sciences et Lettres Varsovie*, 23(Cl. III):30–50, 1930.

[70] A. R. Meyer and L. J. Stockmeyer. The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space. In *13th Annual Symposium on Switching and Automata Theory*, pages 125–129. IEEE, 1972.

[71] A. R. Meyer and L. J. Stockmeyer. Word Problems Requiring Exponential Time. In *ACM Symposium on Theory of Computing (STOC-73)*, pages 1–9. ACM Press, 1973.

[72] M. Narizzano, L. Pulina, and A. Tacchella. Report of the Third QBF Solvers Evaluation. *Journal of Satisfiability, Boolean Modeling and Computation*, 2:145–164, 2006.

[73] J. Oetsch, M. Seidl, H. Tompits, and S. Woltran. A Tool for Advanced Correspondence Checking in Answer-Set Programming. In J. Dix and A. Hunter, editors, *Proceedings on the Workshop on Non-Monotonic Reasoning (NMR 2006)*, pages 20–29, 2006.

[74] J. Oetsch, M. Seidl, H. Tompits, and S. Woltran. A Tool for Advanced Correspondence Checking in Answer-Set Programming: Preliminary Experimental Results. In M. Fink, H. Tompits, and S. Woltran, editors, *Proceedings of the 20th Workshop on Logic Programming (WLP 2006)*, pages 200–205, 2006.

[75] J. Oetsch, M. Seidl, H. Tompits, and S. Woltran. ccT: A Correspondence-Checking Tool for Logic Programs under the Answer-Set Semantics. In M. Fisher, W. v. d. Hoek, B. Konev, and A. Lisitsa, editors, *Proceedings of the 10th European Conference on Logics in Artificial Intelligence (JELIA 2006)*, volume 4160 of *Lecture Notes in Computer Science*, pages 502–505. Springer, 2006.

[76] J. Oetsch, M. Seidl, H. Tompits, and S. Woltran. ccT: A Tool for Checking Advanced Correspondence Problems in Answer-Set Programming. In L. O'Conner, editor, *Proceedings of the 15th International Conference on Computing (CIC 2006)*, pages 3–11. IEEE Computer Society, 2006.

[77] J. Oetsch, M. Seidl, H. Tompits, and S. Woltran. ccT: A Tool for Checking Advanced Correspondence Problems in Answer-Set Programming. In *Proceedings of the LaSh'06 Workshop*, pages 77–92, 2006.

[78] E. Oikarinen and T. Janhunen. Verifying the Equivalence of Logic Programs in the Disjunctive Case. In V. Lifschitz and I. Niemelä, editors, *Proceddings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2004)*, volume 2923 of *Lecture Notes in Computer Science*, pages 180–193. Springer, 2004.

[79] J. Otten. On the Advantage of a Non-Clausal Davis-Putnam Procedure. Technical report, TU Darmstadt, 1997.

[80] G. Pan and M.Y. Vardi. Optimizing a BDD-Based Modal Solver. In F. Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction (CADE 2003)*, volume 2741 of *Lecture Notes in Computer Science*, pages 75–89. Springer, 2003.

[81] G. Pan and M.Y. Vardi. Symbolic Decision Procedures for QBF. In M. Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 453–467. Springer, 2004.

[82] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1995.

[83] J. Rintanen. Constructing Conditional Plans by a Theorem-Prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.

[84] J. Rintanen. Improvements to the Evaluation of Quantified Boolean Formulae. In T. Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 1999)*, pages 1192–1197. Morgan Kaufmann, 1999.

[85] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[86] B. Russel. The Theory of Implication. *American Journal of Mathematics*, 28(2):159–202, 1906.

[87] S. Schamberger. Ein paralleler Algorithmus zum Lösen von Quantifizierten Boolschen Formeln. Master's thesis, Universität Gesamthochschule Paderborn, 2000.

[88] Richard Statman. Intuitionistic propositional logic is polynomial-space complete. *Theor. Comput. Sci.*, 9:67–72, 1979.

[89] L. J. Stockmeyer. The Polynomial-Time Hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1976.

[90] C. Thiffault, F. Bacchus, and T. Walsh. Solving Non-clausal Formulas with DPLL search. In M. Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *Lecture Notes in Computer Science*, pages 663–678. Springer, 2004.

[91] H. Turner. Polynomial-Length Planning Spans the Polynomial Hierarchy. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Proceedings of European Conference on Logics in Artificial Intelligence (JELIA 2002)*, volume 2424 of *Lecture Notes in Computer Science*, pages 111–124. Springer, 2002.

[92] L. A. Wallen. *Automated Deduction in Nonclassical Logics.* MIT Press, 1990.

[93] S. Woltran. Quantified Boolean Formulas - From Theory to Practice. Dissertation, Vienna University of Technology, 2003.

[94] C. Wrathall. Complete Sets and the Polynomial-Time Hierarchy. *Theoretical Computer Science*, 3(1):23–33, 1976.

[95] L. Zhang and S. Malik. Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. In P. Van Hentenryck, editor, *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP 2002)*, volume 2470 of *Lecture Notes in Computer Science*, pages 200–215. Springer, 2002.

[96] M. Zolda. Comparing Different Prenexing Strategies for Quantifed Boolean Formulas. Diplomarbeit, Vienna University of Technology, 2004.

## Curriculum Vitae

**Name**
DI Martina Seidl

**Address**
Wienerstr. 230
3400 Klosterneuburg, Austria

**Date and Place of Birth**
February 4th, 1980, Vienna, Austria

**Education**

| | |
|---|---|
| 1986 - 1990 | Volksschule Anton-Brucknergasse, Klosterneuburg |
| 1990 - 1998 | BG/BRG Klosterneuburg |
| 1998 - 2003 | Studies in Computer Science, University of Technology, Vienna |

**Employment**

| | |
|---|---|
| 1999 - 2004 | Teaching Assistant<br>- Introduction to Programming<br>- Introduction to Logic Oriented Programming<br>Vienna University of Technology |
| 2005 - 2006 | University Assistant<br>Institute of Software Technology and Interactive Systems<br>Vienna University of Technology |
| since 2006 | Research Assistant (Project ModelCVS)<br>Institute of Software Technology and Interactive Systems<br>Vienna University of Technology |

**Languages**
German (native), English (fluently),
French (basic), Spanish (basic), Swedish (beginner)

**Selected Publications**

- U. Egly, M. Seidl, H. Tompits, S. Woltran, M. Zolda, *Comparing Different Prenexing Strategies for Quantified Boolean Formulas*, Proc. of SAT 2003, pages 2214–228, Springer, 2004
- J. Oetsch, M. Seidl, H. Tompits, S. Woltran, *ccT: A Correspondence-Checking Tool for Logic Programs under the Answer-Set Semantics*, Proc. of JELIA 2006, pages 502–505, Springer, 2006
- U. Egly, M. Seidl, S. Woltran, *A Solver for QBFs in Nonprenex Form*, Proc. of ECAI 2006, pages 477–481, IOS Press, 2006