TU

Technische Universität Wien

Diplomarbeit

Comparing Different Prenexing Strategies for Quantified Boolean Formulas

ausgeführt am Institut für
Informationssysteme
der Technischen Universität Wien

unter der Anleitung von
Ao.Univ.Prof. Dipl.-Ing. Dr.rer.nat. Uwe Egly

sowie der begleitenden Betreuung von
Dipl.-Ing. Dr.techn. Stefan Woltran

durch
Michael Zolda
Widerinstraße 14/II/10, A-3100 St. Pölten

Datum Unterschrift

This page is dedicated to the people who supported the author throughout the creation of this work through their help and encouragement.

Vorwort

Das Erfüllbarkeitsproblem der Aussagenlogik wird gerne als Prototyp eines NP-vollständigen Problems herangezogen. In Bezug auf die Klasse der PSPACE-vollständigen Probleme findet indes das Erfüllbarkeitsproblem der *Quantifizierten Aussagenlogik*, welches unter dem Namen *QSAT* bekannt ist, als Standardproblem Verwendung. *QSAT*, welches eine Verallgemeinerung des Erfüllbarkeitsproblems der Aussagenlogik darstellt, ist jedoch nicht nur zur Charakterisierung der Klasse PSPACE geeignet. Vielmehr ermöglichen einfache syntaktische Einschränkungen des Problems die Beschreibung einer ganzen Hierarchie von Komplexitätsklassen, welche sich von P bis PSPACE erstreckt. Diese Hierarchie, welche in [38] eingeführt wird, ist als *Polynomielle Hierarchie* bekannt.

Neben den offensichtlichen Vorteilen eines kanonischen Standardproblems für Betrachtungen theoretischer Natur wird die Sprache der *quantifizierten booleschen Formeln* seit einiger Zeit als Zielsprache für die Übersetzung unterschiedlichster Probleme aus dem Bereichen der Artificial Intelligence und des Automatischen Beweisens herangezogen [6, 18, 20, 41].

Es kann gezeigt werden, dass das allgemeine QSAT Problem ohne Beschränkung der Allgemeinheit auf eine syntaktische Teilmenge von Formeln eingeschränkt werden kann, nämlich auf die Menge der Formeln in *konjunktiver Pränex-Normalform*. Es läßt sich auch zeigen, dass eine entsprechende Normalformtransformation in polynomieller Zeit durchgeführt werden kann.

Die Anwendung dieses Resultats hatte zur Folge, dass heute eine Vielzahl moderner Entscheidungsprozeduren für QSAT nur Formeln in konjunktiver Prenex-Normalform verarbeiten können. Während gegenwärtig viel Aufwand für die Optimierung aktueller Entscheidungsprozeduren betrieben wird, halten sich die Investitionen in das Verständnis von Normalformtransformationen in Grenzen. Das Wissen über das Verhalten solcher Transformationen in praktischen Anwendungen kann sicherlich als unzureichend bezeichnet werden. Insbesondere sind die Auswirkungen der Transformationen auf die praktische Durchführbarkeit von Erfüllbarkeitstests mittels konkreter Entscheidungsprozeduren unklar, was nicht zuletzt auf die hohe Komplexität solcher Prozeduren zurückzuführen ist.

In [20] wurden bereits erste Forschungsergebnisse präsentiert, die einen deutlichen Zusammenhang zwischen der Anwendung unterschiedlicher Strategien während der Normalformtransformation und der Laufzeit aktueller Entscheidungsprozeduren erkennen lassen. Bei den dort verwendeten Formeln handelte es sich um Übersetzungen von *Nested Counterfactuals* [21].

In der vorliegenden Arbeit versuchen wir, die Resultate aus [20] durch eine ge-

nauere Analyse des Zusammenspiels zwischen unterschiedlichen Strategien zur Normalformtransformation und einer Auswahl moderner Entscheidungsprozeduren [23, 31, 32, 33, 37, 43] zu untermauern.

Preface

Whereas the satisfiability problem of propositional logic is widely recognized as the prototypical problem for the class of NP-complete problems, a generalization of that problem has gained similar significance as a representative of the class of PSPACE-complete problems: The satisfiability problem of *quantified propositional logic*, *QSAT*. Even more importantly, simple syntactic restrictions of QSAT lead to a class of decision problems that are prototypical of an entire hierarchy of complexity classes spanning from P to PSPACE, which is commonly known as the *polynomial hierarchy* [38].

Apart from this value as a canonical problem for theoretical works, efforts have recently been made to harness the language of *quantified boolean formulas* as target language for the compilation of various problems from the fields of artificial intelligence and automated reasoning [6, 16, 18, 19, 20, 22, 26, 27, 41, 48].

It can be shown that the general QSAT problem for arbitrary formulas can, without loss of generality, be restricted to a certain syntactic subset of formulas—namely those in *prenex conjunctive normal form*—and that a corresponding translation can be performed in polynomial time.

It is for this reason that many contemporary decision procedures for QSAT assume their input to be in prenex conjunctive normal form. However, less effort than into optimizing state of the art decision procedures has been put into the apprehension of normal form transformations. The practical behavior of these translations is little understood, and even less are the effects they have on the tractability of the problems with respect to existing decision procedures. However, the latter aspect is not very surprising, considering the high complexity of modern QSAT decision procedures.

We have presented some preliminary experimental results in [20], which show that the utilization of different strategies during normal form transformation has a considerable impact on the running time of contemporary decision procedures w.r.t. a particular class of formulas that represent encodings of *nested counterfactuals* [21, 29].

In the present work we confirm the results from [20] by performing a broad analysis of the interaction between various normal form transformation strategies on a number of contemporary decision procedures [23, 31, 32, 33, 37, 43].

Contents

List of Figures

List of Tables

And so it begins. You have forgotten something.
– Kosh

Chapter 1

Basics of Quantified Propositional Logic

In this chapter, we formally introduce *quantified propositional logic*. We define the set of *quantified boolean formulas*, defining the syntax and the semantics of these objects in Sections 1.1 and 1.2, respectively. In Section 1.3, we consider the basic reasoning tasks associated with quantified propositional logic. Next, in Section 1.4, we introduce the concept of a *normal form*, along with the most common normal forms for quantified boolean formulas, most importantly *prenex conjunctive normal form*. The section concludes with a presentation of some simple normal form transformations.

1.1 The Syntax of QBFs

The principal object of interest in quantified propositional logic is the set of *quantified boolean formulas (QBFs)*. We introduce this set and present many useful notions and notations that capture syntactic properties of such formulas.

1.1.1 Formulas and their Components

Definition 1.1.1 (Quantified Boolean Formulas)

1. *Quantified boolean formulas* are words over the following alphabet:
   (a) $v_0, v_1, \ldots$ (*propositional variables* or *atoms*);
   (b) $\top, \bot$ (*propositional constants*);
   (c) $\neg$ (*unary connective*);
   (d) $\wedge, \vee$ (*binary connectives*);
   (e) $\exists$ (*existential quantifier symbol*);
   (f) $\forall$ (*universal quantifier symbol*);
   (g) $(,)$ (*parentheses*).
2. The set of propositional variables is written as $\mathcal{V}$.

3. The set $\mathcal{Q}$ of *quantifiers* is defined by $\mathcal{Q} = \{(\exists q), (\forall q) \mid q \in \mathcal{V}\}$.
4. The set $\mathcal{F}$ of *quantified boolean formulas* is inductively defined by
   (a) $\top, \bot \in \mathcal{F}$;
   (b) $q \in \mathcal{F}$, if $q \in \mathcal{V}$;
   (c) $\neg A \in \mathcal{F}$, if $A \in \mathcal{F}$;
   (d) $(Qq)A \in \mathcal{F}$, if $(Qq) \in \mathcal{Q}$ and $A \in \mathcal{F}$;
   (e) $(A_1 \wedge A_2) \in \mathcal{F}$, if $A_1 \in \mathcal{F}$ and $A_2 \in \mathcal{F}$;
   (f) $(A_1 \vee A_2) \in \mathcal{F}$, if $A_1 \in \mathcal{F}$ and $A_2 \in \mathcal{F}$.

The readings of the symbols $\top, \bot, \neg, \wedge, \vee, \exists$, and $\forall$, introduced above, are (in order): *verum*, *falsum*, *not*, *and*, *or*, *exists*, and *for all*. Formulas of the form $A \wedge B$ (resp. $A \vee B$) are called *conjunctions* (resp. *disjunctions*). A formula of the form $(Qq)A$ is called a *quantifier application* of (Qq) on A.

Many authors introduce additional connectives *implies* and *equivalent* (which are usually written as $\rightarrow$ and $\leftrightarrow$, respectively). Semantically, these connectives can be explained in terms of *not*, *or*, and *and*. We prefer to view them as mere shorthand notation.

Notation 1.1.1 (Shorthands)
We introduce the following shorthands:

1. The notation $A \rightarrow B$ as a shorthand for the formula $\neg A \vee B$;
2. The notation $A \leftrightarrow B$ as a shorthand for the formula $(A \rightarrow B) \wedge (B \rightarrow A)$;
3. The notation $\bigwedge_{i=k}^{n} A_i$ as a shorthand for the formula $\top$, if $k > n$, or
$$A_k \wedge (A_{k+1} \wedge (\ldots \wedge (A_{n-1} \wedge A_n))),$$
if $k \leq n$;
4. The notation $\bigvee_{i=k}^{n} A_i$ as a shorthand for the formula $\bot$, if $k > n$, or
$$A_k \vee (A_{k+1} \vee (\ldots \vee (A_{n-1} \vee A_n))),$$
if $k \leq n$.

As a notational convention, we establish that parentheses may be omitted from a formula, if the syntactic structure arising from Definition 1.1.1 remains clear. In particular, this means that we may always omit the outermost parentheses of a formula. To further simplify the handling of largish formulas, we introduce the following precedence rules.

Convention 1.1.1 (Precedence Rules for Formulas)
1. Shorthands have a higher priority than all genuine operators, except for the *not* operator and quantifier application;
2. The *not* operator and quantifier application have a higher priority than shorthands.

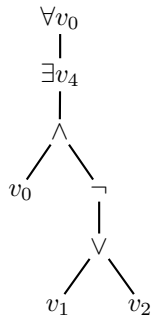

Figure 1.1: The structure of Formula (3) from Example 1.1.1.

Example 1.1.1 (Formulas)
The following strings are examples of formulas:

1. v_7;
2. $v_0 \rightarrow (v_1 \leftrightarrow v_2)$;
3. $(\forall v_0)(\exists v_4)(v_0 \wedge \neg(v_1 \vee v_2))$.

Formula (1) is simply a variable and thus is one of the smallest possible formulas. Formula (2) contains shorthands from Notation 1.1.1. If we expand these shorthands, we obtain

$$(\neg v_0 \vee ((\neg v_1 \vee v_2) \wedge (v_1 \vee \neg v_2))).$$

The structure of Formula (3) is visualized in Figure 1.1.

Convention 1.1.2 (Variable Naming)

1. Variable symbols $A, \ldots, P$ are used to denote arbitrary formulas.[1]
2. Variable symbols $Q, \ldots, U$ are used to denote arbitrary quantifier symbols.
3. Variable symbols $q, \ldots, u$ are used to denote arbitrary propositional variables.
4. The symbol $\circ$ is used to denote an arbitrary binary connective.

The symbols $q, \ldots, u$ are propositional metavariables. It is very easy to confuse these with the concrete propositional variables $v_0, v_1, \ldots$. Propositional variables are part of the language of QBFs, whereas propositional metavariables are introduced because we want to make statements about QBFs that are independent of their concrete propositional variables. Following this line of reasoning, propositional metavariables are to be understood as universally quantified inside such statements. For example, after having introduced the semantics of QBFs in Section 1.2, we might observe that the (schematic) formula $q \vee \neg r$ is not valid, because some concrete formulas, e.g., $v_5 \vee \neg v_7$, are not valid, even though some special ones, like $v_1 \vee \neg v_1$, may be.

[1] Of course, conventions like this one always include derivative symbols, e.g., indexed variants.

We sometimes collect several interesting formulas in a set. Such a set is usually called a *theory*.

Definition 1.1.2 (Theory)
A *theory* $\mathcal{T}$ is any arbitrary subset of $\mathcal{F}$.

Definition 1.1.3 (Complementary Quantifier Symbol)
The *complementary quantifier symbol* (notation $\overline{Q}$) of a quantifier symbol Q is defined by

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

Definition 1.1.4 (Root Symbol)
The *root symbol* $\mathrm{root}(A)$ of a formula A is defined by

$$\mathrm{root}(A) = \begin{cases} A & \text{if } A \in \{\top, \bot\} \cup \mathcal{V}; \\ \neg & \text{if } A = \neg B; \\ Qq & \text{if } A = (Qq)B; \\ \circ & \text{if } A = B_1 \circ B_2. \end{cases}$$

We often want to speak about *sub-formulas*, i.e., those parts of a formula that are formulas in turn. To be able to precisely point at sub-formulas, we need a notion of *position* in a formula.

Positions are words over the alphabet $\{1, 2\}$, with ε denoting the empty word. We associate positions with their corresponding sub-formulas.

Definition 1.1.5 (Positions in a Formula)

1. The set $\mathrm{pos}(A)$ of *positions in a formula* A is recursively defined by:

$$\mathrm{pos}(A) = \begin{cases} \{\varepsilon\} & \text{if } A \in \{\top, \bot\} \cup \mathcal{V}; \\ \{\varepsilon, 1\pi \mid \pi \in \mathrm{pos}(B)\} & \text{if } A = \neg B \text{ or } A = (Qq)B; \\ \{\varepsilon, i\pi_i \mid \pi_i \in \mathrm{pos}(B_i), i \in \{1, 2\}\} & \text{if } A = B_1 \circ B_2. \end{cases}$$

2. The position ε is called *root position*, or simply *root*.

Convention 1.1.3 (Position Naming)
The variables π and ϱ are used to denote arbitrary positions.

It can easily be shown that the set of all positions is closed under the usual string concatenation operation, and that ε is unit w.r.t. that operation. Therefore, we usually drop the symbol ε when writing down positions, unless, of course, the position is the root position.

We can even order positions through the usual prefix order on the set $\{1, 2\}^\star$ of words over $\{1, 2\}$.

Definition 1.1.6 (Order over Positions, Parallelism of Positions)

1. A position ϱ is said to be *below* position π ($\pi \preceq \varrho$), if there exists a position π', such that $\pi\pi' = \varrho$.

2. A position ϱ is said to be *strictly below* position π ($\pi \prec \varrho$), if $\pi \preceq \varrho$, but $\pi \neq \varrho$.

3. Two positions π and ϱ are said to be *parallel* ($\pi \parallel \varrho$), if $\pi \not\preceq \varrho$ and $\varrho \not\preceq \pi$.

Definition 1.1.7 (Sub-Formula, Occurrence)

1. Let π be a position in a formula A, i.e., $\pi \in \mathrm{pos}(A)$. Then the *sub-formula of A at position π*, written as $A|_\pi$, is recursively defined by

$$A|_\pi = \begin{cases} A & \text{if } \pi = \varepsilon; \\ B|_\varrho & \text{if } \pi = 1\varrho \text{ and either } A = \neg B \text{ or } A = (Qq)B; \\ B_i|_\varrho & \text{if } \pi = i\varrho \text{ and } A = B_1 \circ B_2. \end{cases}$$

2. A formula B is said to *occur in A at position π*, if $A|_\pi = B$. In that case, we also speak of the *occurrence of B in A at position π*.

3. A quantifier Qq is said to *occur in A at position π*, if $A|_\pi = (Qq)B$. In that case, we also speak of the *quantifier occurrence of Qq in A at position π*.

4. A formula B is called a *sub-formula of A* if $A|_\pi = B$, for some position π. We then write $A \preceq B$.

5. A formula B is called a *proper sub-formula of A* if $A|_\pi = B$, for some position $\pi \neq \varepsilon$. We then write $A \prec B$.

6. A formula B is called an *immediate sub-formula of A* if $A|_\pi = B$, for some position $\pi \in \{1, 2\}$.

7. If $A \preceq B$, and B is a special kind of formula (e.g., a variable, a disjunction, etc.), we say that B is, respectively, a *variable*, a *disjunction*, etc. ***in*** A.

Example 1.1.2 (Positions)
Consider the formula

$$A = (\forall q)(\exists r)(q \leftrightarrow r),$$

which, by Notation 1.1.1, is a shorthand for the actual formula

$$(\forall q)(\exists r)((\neg q \vee r) \wedge (\neg r \vee q)).$$

Then the set of positions of A is

$$\mathrm{pos}(A) = \{\varepsilon, 1, 11, 111, 1111, 11111, 1112, 112, 1121, 11211, 1122\}.$$

The positions 111 and 1121 are parallel, and 112 is strictly below 1. The sub-formula at position 112 is $A|_{112} = \neg r \vee q$. It is a proper, but not an immediate sub-formula, of A.

We sometimes want to introduce new variables that are independent of some previously mentioned formulas.

Definition 1.1.8 (Fresh Variable)
When a variable q is introduced in a context (a definition, an algorithm, etc.) where some formulas $A_0, \ldots, A_n$ occur, then q is called a *fresh variable* under the premise that it does not occur in any of these formulas, i.e., none of of the conditions $A_0 \preceq q$, $\ldots$, $A_n \preceq q$ is satisfied.

Example 1.1.3 (Fresh Variable)
Consider the formulas

$$A = v_0 \vee (\forall v_1)(v_2 \vee v_1) \text{ and } B = (v_3 \vee \neg v_0) \wedge (v_4).$$

In this context, v_5 is a fresh variable. The variables v_0 through v_4 are not fresh, as they already occur in at least one of the formulas A and B.

For complexity discussions, we need a measure for the size of a formula. Depending on what we regard a formula as (a string, a tree, a DAG[2], etc.), different measures suggest themselves, and their choice has indeed an influence on the complexity analysis of algorithms or problems.

However, for "natural" measures, this impact is small and typically manifests itself as a polynomial factor in complexity terms (see, e.g., Section 2.2. of [39]). Note, however, that DAGs are an exception here, in the sense that they can efficiently encode trees, i.e., for certain trees the corresponding DAG is exponentially smaller.

We choose a measure that is based on the tree notion of formulas that is suggested by the inductive definition of formulas. Our measure simply counts the number of nodes in such a representation, following the intuition that each node can be represented by a memory segment of a fixed size k, containing a representation of the node name and a maximum of two pointers to its successor nodes. More precisely, we define the following standard machine representation of formulas.

Definition 1.1.9 (Standard Machine Representation)
By the *standard representation of a formula* A, we mean a machine representation of A that consists of $|\mathrm{pos}(A)| + 1$ memory segments of some fixed size k, which we sometimes call *node*s. For each position $\pi \in A$, there is exactly one segment that encodes the value of $\mathrm{root}(A|_\pi)$. Furthermore, each segment contains a maximum of two pointers to the segments that represent the immediate sub-formulas of $A|_\pi$. Lastly, there is one special, marked segment, called *root segment* (or *root node*), which merely holds an initial pointer to the segment corresponding to position ε.

One might object that the assumption of memory segments of fixed size disregards the fact that there is an infinite number of propositional variables. However, since

1. any real machine is constrained by a limited amount of memory, and
2. binary numbers can efficiently encode a huge range of variables,

we can neglect this aspect for all cases with practical relevance.

Definition 1.1.10 (Cardinality of Formulas)
The cardinality $|A|$ of a formula A is defined by $|A| = |\mathrm{pos}(A)|$.

[2]DAG: directed acyclical graph.

Example 1.1.4 (Cardinality of Formulas)
Consider the formula

$$A = (\forall q)(\exists r)((\neg q \vee r) \wedge (q \vee r)).$$

Then the set of positions in A is

$$\mathrm{pos}(A) = \{\varepsilon, 1, 11, 111, 1111, 11111, 1112, 112, 1121, 1122\},$$

so the cardinality of A is 10.

We now turn towards certain classes of formulas that are so interesting that they have been given special names in literature. In particular, we consider *literals*, *simple formulas*, *short definitions*, *clauses*, and *c-conjunctions*[3].

Definition 1.1.11 (Literal)
A formula L is called a *literal*, if it is either a propositional variable q (called a *positive literal*) or its negation $\neg q$ (called a *negative literal*).

Definition 1.1.12 (Relative Literal Variable Position)
Let L be a literal. Then the *relative literal variable position* $\mathrm{lvp}(L)$ of L is defined by

$$\mathrm{lvp}(L) = \begin{cases} \varepsilon & \text{if } L = q; \\ 1 & \text{if } L = \neg q. \end{cases}$$

Definition 1.1.13 (Literal Variable)
Let L be a literal. Then the *literal variable* $\lfloor L \rfloor$ of L is defined by

$$\lfloor L \rfloor = L|_{\mathrm{lvp}(L)}.$$

For any given set $\mathcal{A}$ of literals, we define

$$\lfloor \mathcal{A} \rfloor = \{\lfloor L \rfloor \mid L \in \mathcal{A}\}.$$

Definition 1.1.14 (Complementary Literal)
The *complementary literal* $\overline{L}$ of a literal L is defined by

$$\overline{L} = \begin{cases} \neg q & \text{if } L = q; \\ q & \text{if } L = \neg q. \end{cases}$$

Definition 1.1.15 (Simple Formula)
A formula is called *simple*, if each of its proper sub-formulas is either a literal or a constant.

Definition 1.1.16 (Partial Short Definition)
A formula is called a *(partial) short definition*, if it has one of the forms $q \to A$ or $A \to q$, where A is a simple formula.

Definition 1.1.17 (Clause)
A formula is called a *clause*, if it is of the form $\bigvee_{i=1}^{n} L_i$ where $L_1, \ldots, L_n$ are literals. The clause $\bigvee_{i=1}^{0} L_i = \bot$ is called the *empty clause*.

[3] Our non-standard name for conjunctions of clauses.

Definition 1.1.18 (C-Conjunction)
A formula is said to be a *c-conjunction*, if it is of the form $\bigwedge_{i=1}^{n} C_i$, where $C_1, \ldots, C_n$ are clauses. The c-conjunction $\bigwedge_{i=1}^{0} C_i = \top$ is called the *empty c-conjunction*.

Definition 1.1.19 (Counts for C-Conjunctions)
Let

$$A = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{k_i} L_{i,j}$$

be a c-conjunction. Then

1. the *clause count* $\operatorname{ccnt}(A)$ of A is defined by

$$\operatorname{ccnt}(A) = m;$$

2. the *literal count* $\operatorname{lcnt}(A, L, n)$ of a literal L in clauses of A with a length of exactly n is defined by

$$\operatorname{lcnt}(A, L, n) = \sum_{\substack{1 \le i \le m \\ k_i = n}} \sum_{\substack{1 \le j \le k_i \\ L_{i,j} = L}} 1;$$

3. the *maximal clause length* $\operatorname{mcl}(A)$ in A is defined by

$$\operatorname{mcl}(A) = \max_{1 \le i \le m} (k_i).$$

Example 1.1.5 (Counts for C-Conjunctions)
Consider the formula

$$A = (q \vee r \vee \neg s) \wedge (q \vee \neg r \vee \neg s) \wedge (q \vee r) \wedge (\neg q \vee s).$$

Then $\operatorname{ccnt}(A) = 4$, $\operatorname{lcnt}(A, r, 3) = 1$, $\operatorname{lcnt}(A, \neg s, 3) = 2$, and $\operatorname{mcl}(A) = 3$.

Convention 1.1.4 (Special Formula Naming)
1. The variable C denotes an arbitrary clause.
2. The variable D denotes a short definition.
3. The variable L denotes an arbitrary literal.[4]

Example 1.1.6 (Special Formulas)
Consider the following formulas:

1. $\bot$;
2. $\neg r$;
3. $\neg q \wedge \neg r$;

[4] Many authors like to use lower case letters for literals, because they generalize the concept of variables. However, we find upper case letters more appropriate, since literals actually *are* formulas.

4. $\neg q \vee (s \vee \neg r)$;
5. $(q \vee \neg r) \wedge (r \vee s)$.

Of these, only formula (2) is a literal. Formulas (1), (2) and (4) are clauses, whereas (3) and (5) are not. Just formula (3) is a simple formula. Only formula (4) is a short definition, and all six formulas are c-conjunctions.

Polarity is an important syntactic property that tells us something about the semantic qualities of sub-formulas. In particular, it characterizes the semantically interesting case of pure literals, which is given below.

Definition 1.1.20 (Polarity)
1. Let π be a position in A. Then the *polarity* $\mathrm{pol}(A, \pi)$ of the sub-formula of A at position π is recursively defined by

$$\mathrm{pol}(A,\pi) = \left\{ \begin{array}{ll} +1 & \text{if } \pi = \varepsilon; \\ (-1)\cdot \mathrm{pol}(A,\varrho) & \text{if } A|_\varrho = \neg B \\ (+1)\cdot \mathrm{pol}(A,\varrho) & \text{otherwise} \end{array} \right\} \text{if } \pi = \varrho i, i \in \{1,2\}.$$

2. Let A be a formula with $A|_\pi = B$. If $\mathrm{pol}(A, \pi) = +1$, we say that B *occurs positively* at position π of A. Otherwise, B is said to *occur negatively* at position π of A.

Definition 1.1.21 (Pure Literal)
Let L be a literal that occurs at position π of some formula A. If

$$|\{\mathrm{pol}(A,\varrho) \mid A|_\varrho = \lfloor L \rfloor, \varrho \in \mathrm{pos}(A)\}| = 1,$$

i.e., $\lfloor L \rfloor$ occurs either only positively or only negatively in A, then we say that L is *pure in* A with polarity $x = \mathrm{pol}(A, \pi)$.

Example 1.1.7 (Polarity and Pure Literal)
Consider the formula $A = (q \vee r) \wedge \neg(\neg q \vee r)$. Then

1. The formula $q \vee r$ occurs positively at position 1;
2. The formula q occurs positively at positions 11 and 2111;
3. The formula $\neg q$ occurs negatively at position 211;
4. The formula r occurs positively at position 12, but negatively at position 212;
5. The literals q and $\neg q$ are pure in A, whereas r is not.

1.1.2 Variable Binding

For literals, we distinguish between *free* and *bound* occurrences.

Definition 1.1.22 (Bound and Free Occurrence)
Let L be a literal that occurs at some position π of a formula A.

1. If there exist positions π_1, π_2 such that $\pi = \pi_1\pi_2$ and $\mathrm{root}(A|_{\pi_1}) = Q\lfloor L \rfloor$, we say that L occurs *bound* at position π of A. Otherwise, we say that L occurs *free* at position π of A.

2. If L has only bound (resp. only free) occurrences in A, we say that L *is bound* (resp. *is free*) in A.

3. We denote the set of all literals that have free occurrences in A by $\mathrm{free}(A)$. Likewise, $\mathrm{bound}(A)$ denotes the set of all literals that have bound occurrences in A. A formula A with $\mathrm{free}(A) = \emptyset$ is called *closed*. Otherwise, it is called *open*. For sets $\mathcal{A}$ of formulas, we define

$$\mathrm{free}(\mathcal{A}) = \bigcup_{A \in \mathcal{A}} \mathrm{free}(A)$$

and

$$\mathrm{bound}(\mathcal{A}) = \bigcup_{A \in \mathcal{A}} \mathrm{bound}(A).$$

From a semantic point of view (c.f. Section 1.2), a literal L that occurs bound at a position π of a formula is associated with exactly one quantifier occurrence. That quantifier occurrence is the one which is nearest to L, i.e., the one for which π_2 from Definition 1.1.22 is minimal w.r.t. $\preceq$. This leads us to the definition of the associated quantifier position and the associated quantifier of a bound literal.

Definition 1.1.23 (Associated Quantifier)
Let L be a literal that occurs bound at position π of a formula A.

1. By Definition 1.1.22, there exist positions π_1, π_2 such that $\pi = \pi_1\pi_2$ and $\mathrm{root}(A|_{\pi_1}) = Q\lfloor L \rfloor$. Choose the minimal π_2 (w.r.t. $\preceq$) that satisfies the above condition. We then call the position π_1 the *associated quantifier position* of the literal occurrence at position π of the formula A, and designate this position by $\mathrm{qpos}(A, \pi)$.

2. The *associated quantifier* $\mathrm{quant}(A, \pi)$ of a literal that occurs bound at position π of a formula A is defined by

$$\mathrm{quant}(A, \pi) = Q\lfloor L \rfloor, \text{ where } Q\lfloor L \rfloor = \mathrm{root}(A|_{\mathrm{qpos}(A,\pi)}).$$

Definition 1.1.24 (Existential and Universal Binding)

1. Let L be a literal that occurs bound at position π of a formula A and let $\varrho = \mathrm{qpos}(A, \pi)$. We say that L *occurs existentially bound* at position π, if either $\mathrm{root}(A|_\varrho) = \exists\lfloor L \rfloor$ and $\mathrm{pol}(A, \varrho) = +1$, or $\mathrm{root}(A|_\varrho) = \forall\lfloor L \rfloor$ and $\mathrm{pol}(A, \varrho) = -1$. Otherwise, we say that L *occurs universally bound* at position π. If L has only existentially bound (resp. universally bound) occurrences in A, we say that L *is existentially bound* (resp. *is universally bound*) in A.

2. The set of all literals that are existentially bound in A, is written as $\mathrm{lit}_\exists(A)$.

3. The set of all literals that are universally bound in A, is written as $\mathrm{lit}_\forall(A)$.

Example 1.1.8 (Bound and Free Occurrence)
Consider the formula

$$A = \neg((\exists q)(\neg q \wedge r)) \vee (\exists s)(q \vee s).$$

The variable q occurs bound at position 11111 and free at position 211. Furthermore, with $\varrho = \mathrm{qpos}(A, 11111) = 11$, it can be seen that $\mathrm{root}(A|_\varrho) = \exists q$ and $\mathrm{pol}(A, \varrho) = -1$, and therefore the variable q (and also the literal $\neg q$) *occurs* universally bound at position 11111.

However, it is not true that q *is* universally bound in A, because there is a free occurrence of q too. For s, on the other hand, all occurrences are existentially bound, so s is existentially bound in A. Furthermore, A is open, because $\mathrm{free}(A) = \{q, r\}$.

Example 1.1.9 (Bound Literals in Formula)
Consider the formula

$$(\exists q)(\forall r)(\exists t)((\neg q \vee r \vee \neg s) \wedge (q \vee \neg r \vee t) \wedge (q \vee r \vee \neg u)).$$

Then $\mathrm{lit}_\exists(A) = \{q, \neg q, t\}$ and $\mathrm{lit}_\forall(A) = \{r, \neg r\}$.

The framework presented for dealing with the binding properties of variables so far is a bit cumbersome to use. Therefore, most literature introduces the notion of a *cleansed* formula.

Definition 1.1.25 (Cleansed Formula)

1. A formula A is said to be *cleansed*, if the following conditions hold:
   (a) If $(Qq)B$ and $(Rr)C$ are sub-formula occurrences at different positions in A, then $q \neq r$.
   (b) If $(Qq)B$ is a sub-formula of A, then $q \notin \mathrm{free}(A)$ and $q \in \mathrm{free}(B)$.
2. The set of all cleansed formulas is written as $\dot{\mathcal{F}}$.

Example 1.1.10 (Cleansed Formula)
Consider the following formulas:

1. $q \vee (\forall q)q$;
2. $(\exists r)q \vee (\forall q)q$;
3. $(\exists q)q \vee (\forall q)q$;
4. $(\exists r)r \vee (\forall q)q$.

Of these formulas, only the last one is a cleansed formula.

Theorem 1.1.1 (Cleansed Formula)
Let A be a cleansed formula. Then the following propositions hold:

1. If L is a literal in A, then L is either free in A, or bound in A.
2. If L is a literal that occurs bound at position π of A, then there exists exactly one position π_1 with $\mathrm{root}(A|_{\pi_1}) = Q\lfloor L \rfloor$.

3. If $\mathrm{root}(A|_\pi) = Qq$ for some π, then A contains a bound occurrence of the variable q.

Proof.

1. Assume that L is a literal in A that is neither free nor bound in A, i.e., there is at least one free and at least one bound occurrence of L in A. Then A certainly contains a sub-formula of the form $(Q\lfloor L\rfloor)B$. Since A is cleansed, it follows that $\lfloor L\rfloor \notin \mathrm{free}(A)$, i.e., there cannot be any free occurrence of L in A, which contradicts our assumption.

2. Assume that L is a literal that occurs bound at position π of A. The proof that there is at least one position π_1 with $\mathrm{root}(A|_{\pi_1}) = Q\lfloor L\rfloor$ is trivial, because $\pi_1 = \mathrm{quant}(A, \pi)$ is such a position.

   Now assume another position π_2 such that $\mathrm{root}(A|_{\pi_2}) = Q\lfloor L\rfloor$. Because A is cleansed, we can conclude, by the contraposition of the first part of the definition of a cleansed formula that $\pi_1 = \pi_2$, i.e., there is at most one position π_1 with $\mathrm{root}(A|_{\pi_1}) = Q\lfloor L\rfloor$. Altogether, we have shown that there is exactly one position π_1 with $\mathrm{root}(A|_{\pi_1}) = Q\lfloor L\rfloor$.

3. Assume that $\mathrm{root}(A|_{\pi_1}) = Qq$ for some π_1. Then $(Qq)B$ is a sub-formula of A, for some B. Since A is cleansed, it follows that $q \in \mathrm{free}(B)$. Therefore, q is clearly a literal that occurs at some position π_2 in B. The corresponding position of q in A is then $\pi = \pi_1\pi_2$, and, by our assumption, $\mathrm{root}(A|_{\pi_1}) = Qq$, i.e., q occurs bound at position π of A.

□

The first part of this theorem tells us that we never need to refer to particular positions when talking about the binding status of a literal in a cleansed formula. The second part tells us that, within a cleansed formula, there always exists a single associated quantifier position for all bound occurrences of the literals L and $\overline{L}$.

Taking both these parts of the theorem together, it follows that we can simply talk about *the* associated quantifier position of a given literal that is bound in a cleansed formula, without referring to a particular occurrence.

As a consequence we may—for cleansed formulas only—overload the definitions of $\mathrm{qpos}(\cdot, \cdot)$ and $\mathrm{quant}(\cdot, \cdot)$, such that they accept literals in their second argument. This will simplify our handling of cleansed formulas.

Definition 1.1.26 (Associated Quantifier for Cleansed Formulas)
Let L be a literal that occurs bound in a cleansed formula A.

1. The *associated quantifier position of the literal* L in A is defined by

$$\mathrm{qpos}(A, L) = \mathrm{qpos}(A, \pi),$$

   where π is chosen such that $A|_\pi = L$.

2. The *associated quantifier of the literal* L in A is defined by

$$\mathrm{quant}(A, L) = A|_{\mathrm{qpos}(A,L)}.$$

Example 1.1.11 (Associated Quantifier for Cleansed Formulas)
Consider the formula

$$A = (\exists q)(q \wedge \neg q) \wedge (\forall r)(r \wedge s).$$

We can see that $\mathrm{qpos}(A, \neg q) = 1$ and $\mathrm{quant}(A, \neg q) = (\exists q)$. The expressions $\mathrm{qpos}(A, s)$ and $\mathrm{quant}(A, s)$ are undefined, because s is not bound in A.

The third part of Theorem 1.1.1 tells us that cleansed formulas never contain superfluous quantifiers that could immediately be removed from a formula without affecting its semantics. If some quantifier occurs in a given formula, we can assume that formula also contains a bound occurrence of the corresponding variable. In other words, the relation between bound literals and quantifiers is a surjection, for any fixed cleansed formula.

These properties make cleansed formulas very appealing for day-to-day work. Later, in Section 1.2, we will see that it is, without loss of generality, possible to restrict oneself to cleansed formulas in discussions of quantified propositional logic.

1.1.3 Modifying Formulas

Sometimes we want to replace a sub-formula by another (arbitrary) formula. This can be achieved by *substituting in* a given *position*.

Definition 1.1.27 (Substitution at Position)
Let π be a position in A. Then $A[B]_\pi$ denotes the formula obtained from A by replacing the sub-formula at position π of A by B. Formally:

$$A[B]_\pi = \begin{cases} B & \text{if } \pi = \varepsilon; \\ \neg(C[B]_\varrho) & \text{if } \pi = 1\varrho \text{ and } A = \neg C; \\ (Qq)(C[B]_\varrho) & \text{if } \pi = 1\varrho \text{ and } A = (Qq)C; \\ (C_1[B]_\varrho) \circ C_2 & \text{if } \pi = 1\varrho \text{ and } A = C_1 \circ C_2; \\ C_1 \circ (C_2[B]_\varrho) & \text{if } \pi = 2\varrho \text{ and } A = C_1 \circ C_2. \end{cases}$$

Another useful form of replacement that complements substitution at positions is substitution of free occurrences of a variable.

Definition 1.1.28 (Substitution of Free Variables)
Let A be a formula and q a propositional variable. $A[q/B]$ denotes the result of replacing each free occurrence of q in A by B, and is defined as follows:

$$A[q/B] = \begin{cases} A & \text{if } A \in \{\top, \bot\} \cup \mathcal{V} \text{ and } A \neq q; \\ B & \text{if } A = q; \\ \neg(C[q/B]) & \text{if } A = \neg C; \\ (Qq)C & \text{if } A = (Qq)C; \\ (Qr)(C[q/B]) & \text{if } A = (Qr)C \text{ and } r \neq q; \\ C_1[q/B] \circ C_2[q/B] & \text{if } A = C_1 \circ C_2. \end{cases}$$

Notation 1.1.2 (Shorthand for Sequences of Substitutions)
As a shorthand for a sequence $(\ldots((A[q_1/B_1])[q_2/B_2])\ldots[q_n/B_n])$ of substitutions, we simply write $A[q_1/B_1, q_2/B_2, \ldots, q_n/B_n]$.

Example 1.1.12 (Substitutions)
Consider the formulas

$$A = (\exists s)((\exists r)(q \wedge ((\neg q \vee r) \wedge s)) \vee (\forall q)(\neg q)) \text{ and } B = (\exists q)(q \vee s).$$

Then the formula obtained from A by replacing the sub-formula at position 1211 by B is

$$A[B]_{1211} = (\exists s)((\exists r)(q \wedge ((\neg q \vee r) \wedge s)) \vee (\forall q)(\neg(\exists q)(q \vee s))).$$

On the other hand, the formula obtained from A by replacing each free occurrence of q by B is

$$A[q/B] = (\exists s)((\exists r)((\exists q)(q \vee s) \wedge ((\neg(\exists q)(q \vee s) \vee r) \wedge s)) \vee (\forall q)(\neg q)).$$

Obviously, only substituting at a position allows us to replace bound variables. Note that formulas resulting from either kind of substitution need not be cleansed, even if both arguments of the substitution are. Also note how s, which is free in B, becomes bound during substitution. Usually, we want to avoid such effects by working with fresh variables (c.f. Definition 1.1.8) and by performing appropriate changes of bound variable (see below).

Definition 1.1.29 (Change of Bound Variables)
We say that a formula A' is obtained from a formula A by a *change of variables bound at position π into r*, if $A|_\pi = (Qq)B$ and $A' = A[(Qr)(B[q/r])]_\pi$, where $r \notin \text{free}(B)$.

Definition 1.1.30 (Elimination of Superfluous Quantifier)
We say that a formula A' is obtained from a formula A by an *elimination of a superfluous quantifier at position π*, if there exists a position π such that $A|_\pi = (Qq)B$ with $q \notin \text{free}(B)$ and $A' = A[B]_\pi$.

Example 1.1.13 (Change of Bound Var., Elim. of Sup. Quantifier)
Consider, once again, the formula

$$A[B]_{1211} = (\exists s)((\exists r)(q \wedge ((\neg q \vee r) \wedge s)) \vee (\forall q)(\neg(\exists q)(q \vee s)))$$

from Example 1.1.12. Then the formula

$$(\exists s)((\exists t)(q \wedge ((\neg q \vee t) \wedge s)) \vee (\forall q)(\neg(\exists q)(q \vee s)))$$

is obtained from $A[B]_{1211}$ by a change of bound variables. Furthermore,

$$(\exists s)((\exists r)(q \wedge ((\neg q \vee r) \wedge s)) \vee (\neg(\exists q)(q \vee s)))$$

is obtained from $A[B]_{1211}$ by the elimination of a superfluous quantifier. If we repeatedly apply both operations, we can derive the cleansed formula

$$(\exists s)((\exists r)(q \wedge ((\neg q \vee r) \wedge s)) \vee (\neg(\exists t)(t \vee u))).$$

As we have already mentioned in the above example, changes of bound variables and elimination of superfluous quantifiers can serve as basic operations in the conversion of formulas into cleansed form.

In the following section, we will, amongst others, see that both operations preserve the semantics of formulas.

1.2 The Semantics of QBFs

1.2.1 Basic Semantic Definitions

The semantics of QBFs is defined through the notions of *truth*, *interpretation*, and *model*. We formally introduce these notions along with the important semantic concepts of *tautology*, *contradiction*, *logical entailment*, and *logical equivalence*. Then we present some important results concerning the semantic properties of QBFs.

Definition 1.2.1 (Interpretation, Truth Value)
An *interpretation* is a function $\phi : \mathcal{V} \to \{0,1\}$, which maps every propositional variable $q \in \mathcal{V}$ to a *truth value* $\phi(q)$. We denote the set of all possible interpretations by $\mathcal{I}$.

Definition 1.2.2 (Evaluation Function)
The evaluation function $v_{\text{qbf}} : \mathcal{F} \times \mathcal{I} \to \{0,1\}$ is recursively defined by:

$$v_{\text{qbf}}(A,\phi) = \begin{cases} 1 & \text{if } A = \top; \\ 0 & \text{if } A = \bot; \\ \phi(A) & \text{if } A \in \mathcal{V}; \\ 1 - v_{\text{qbf}}(B,\phi) & \text{if } A = \neg B; \\ \min(v_{\text{qbf}}(B,\phi), v_{\text{qbf}}(C,\phi)) & \text{if } A = B \wedge C; \\ \max(v_{\text{qbf}}(B,\phi), v_{\text{qbf}}(C,\phi)) & \text{if } A = B \vee C; \\ v_{\text{qbf}}(B[q/\top] \wedge B[q/\bot], \phi) & \text{if } A = (\forall q)B; \\ v_{\text{qbf}}(B[q/\top] \vee B[q/\bot], \phi) & \text{if } A = (\exists q)B. \end{cases}$$

Definition 1.2.3 (Satisfiability, Model, Validity)

1. A formula A is *satisfied* by an interpretation ϕ, if $v_{\text{qbf}}(A,\phi) = 1$. In that case, we write $\phi \models A$ and call ϕ a *model* of A. We denote the set of all models of A by $\mathcal{M}(A)$. A formula A with $\mathcal{M}(A) \neq \emptyset$ is said to be *satisfiable*. Otherwise, it is said to be *unsatisfiable*. An unsatisfiable formula is also called a *contradiction*. A formula A, for which $\mathcal{M}(A) = \mathcal{I}$, is said to be *valid*. Such a formula is also called a *tautology*.

2. A theory $\mathcal{T}$ is *satisfied* by an interpretation ϕ (notation $\phi \models \mathcal{T}$), if $v_{\text{qbf}}(A,\phi) = 1$ for every $A \in \mathcal{T}$. The definitions of *model*, *satisfiability*, *unsatisfiability*, *validity* and $\mathcal{M}$ for theories are then analogous to the corresponding definitions for formulas. Satisfiable theories are also called *consistent*, unsatisfiable ones are called *inconsistent*.

3. The denotation of a formula is the set of its models.

Example 1.2.1 (Satisfiability, Model, Validity)
Consider the formulas

$$A = (\forall q)(\exists r)((\neg q \vee r) \wedge (q \vee \neg r)) \text{ and } B = s \vee (t \wedge \neg t).$$

Then $v_{\text{qbf}}(A,\phi) = 1$ for every interpretation $\phi \in \mathcal{I}$, i.e., $\mathcal{M}(A) = \mathcal{I}$, and therefore A is a valid formula. It is easy to see that every valid formula is also a satisfiable formula, so A is also a satisfiable formula. On the other hand, $v_{\text{qbf}}(B,\phi)$ evaluates to 1 for every ϕ with $\phi(s) = 1$, but to 0, otherwise. Therefore, B is satisfiable, but not valid. The theory $\mathcal{T} = \{A, B\}$ is consistent.

Definition 1.2.4 (Maximal Consistent Subtheories)
Let $\mathcal{T}$ be a theory. Then the set of maximal A-consistent subtheories of $\mathcal{T}$, $\mathrm{mct}(A, \mathcal{T})$, is defined by

$$\mathrm{mct}(A, \mathcal{T}) = \{\mathcal{A} \mid \mathcal{A} \subseteq \mathcal{T}, \mathcal{A} \nvdash \neg A, \mathcal{A} \subset \mathcal{A}' \subseteq \mathcal{T} \Rightarrow \mathcal{A}' \vdash \neg A\}.^5$$

By the given semantics, it is clear that ordinary propositional logic can be viewed as a fragment of quantified propositional logic, so that we can define the set of boolean formulas as a subset of $\mathcal{F}$:

Definition 1.2.5 (Boolean Formulas)
The set $\mathcal{B}$ of *boolean formulas* is the set of all quantified boolean formulas that do not contain any quantifier symbol.

1.2.2 Entailment and Equivalence

Definition 1.2.6 (Entailment and Equivalence)

1. A formula A *logically entails* a formula B, if $\mathcal{M}(A) \subseteq \mathcal{M}(B)$. We then write $A \vdash B$.

2. A theory $\mathcal{T}$ *logically entails* a formula B, if $\mathcal{M}(\mathcal{T}) \subseteq \mathcal{M}(B)$. We then write $\mathcal{T} \vdash B$.

3. Two formulas A and B are *logically equivalent*, if $\mathcal{M}(A) = \mathcal{M}(B)$. We then write $A \equiv B$.

4. Two formulas A and B are *satisfiability equivalent*, if either both, $\mathcal{M}(A) = \emptyset$ and $\mathcal{M}(B) = \emptyset$, or both, $\mathcal{M}(A) \neq \emptyset$ and $\mathcal{M}(B) \neq \emptyset$. We then write $A \overset{\text{sat}}{\equiv} B$.

Example 1.2.2 (Entailment, Equivalence)
Consider the formulas

$$A = (\forall r)(s \wedge (q \vee r)), B = (\forall r)(s \wedge (\neg q \vee r)) \text{ and } C = s \wedge \neg q.$$

We can see that $\phi \models A$, for every ϕ with $\phi(s) = 1$ and $\phi(q) = 1$, but not for any other ϕ. Moreover, $\phi \models B$ for every ϕ with $\phi(s) = 1$ and $\phi(q) = 0$, but not for any other ϕ. Finally, only interpretations ϕ with $\phi(s) = 1$ and $\phi(q) = 0$ are models of C. Therefore, by comparing the models of B and C, we can see that these two formulas are logically equivalent.

On the other hand, we can see that A and B are not logically equivalent. We can perform the same test once more to conclude that A and C are not logically equivalent, but this follows more easily from the fact that the relation $\equiv$ is reflexive.

Reflexivity and the rest of the equivalence properties are stated in the below. Next, we can see that all three formulas are pairwise satisfiability equivalent, since each of them has at least one model. Concerning entailment, we see that B entails C and vice versa. This is generally true for logically equivalent formulas.

[5] Please note the (typographically subtle) difference between $\mathcal{A}$ and A.

It is easy to see that $\equiv$ and $\overset{\text{sat}}{\equiv}$ inherit the three defining properties of an equivalence relation—reflexivity, symmetry, and transitivity—from the corresponding properties of the set equivalence relation, on which the definition of either relation is based.

Theorem 1.2.1 (Equivalence Properties)
The relations $\equiv$ and $\overset{\text{sat}}{\equiv}$ are reflexive, symmetric and transitive.

The following theorem allows us to replace equals by equals, even within a formula. This enables the kind of reasoning known as "reasoning by substitution".

Theorem 1.2.2 (Equivalent Replacement)
Let A be a formula such that $A|_\pi = B$ and $B \equiv B'$. Then $A[B']_\pi \equiv A$.

Proof. We only give a sketch of a complete proof. Assume that B and B' are logically equivalent, i.e., for every ϕ, it holds that $v_{\text{qbf}}(B, \phi) = v_{\text{qbf}}(B', \phi)$.

For $\pi = \varepsilon$, the proof follows directly from the symmetry property of the relation $\equiv$, considering that $A|_\varepsilon = A$ and $A[B']_\varepsilon = B'$.

Next, by using Definition 1.2.2, it is then easy to show, via a simple case distinction, that $v_{\text{qbf}}(A[B]_i, \phi) = v_{\text{qbf}}(A[B']_i, \phi)$, i.e., that $A[B]_i \equiv A[B']_i$, for $i \in \{1, 2\}$.

For formulas A with $A|_i = B$, it is clear that $A = A[B]_i$, so we have shown the theorem for the case where $\pi \in \{1, 2\}$. The general case can now be shown by a simple inductive argument. □

There is another important replacement theorem, which allows replacement between formulas B and B' where $B \vdash B'$, under an additional premise concerning polarities. The resulting formula is not necessarily logically equivalent to the original one, but when we are concerned with satisfiability, the theorem can serve as a sufficient (un)satisfiability criterion.

Theorem 1.2.3 (Monotonic Replacement)
Let A, B, and B' be formulas such that $A|_\pi = B$ and $\mathcal{M}(B) \subseteq \mathcal{M}(B')$ (i.e., $B \vdash B'$). Then it holds that

1. A entails $A[B']_\pi$, provided that $\text{pol}(A, \pi) = +1$;
2. $A[B']_\pi$ entails A, provided that $\text{pol}(A, \pi) = -1$.

There are many useful equivalences, which we might use for reasoning by substitution. We get even more comfort, if we use them as left-to-right oriented *replacement rules* that perform *formula rewriting* in just the same way that term rewrite rules perform term rewriting.[6] Note however, that we won't usually use commutativity as a rewrite rule, because it could lead to looping when applied unrestrictedly.

[6] The major difference is that we cannot easily interpret QBFs as terms over a finite signature. We would either have to introduce an infinite number of function symbols (one for each quantifier) or resort to more intricate vehicles, like higher order terms.

Theorem 1.2.4 (Useful Equivalences)
The following equivalences hold:

1. $\left.\begin{array}{l} A \wedge B \equiv B \wedge A \\ A \vee B \equiv B \vee A \end{array}\right\}$ (commutativity);
2. $\left.\begin{array}{l} (A \wedge B) \wedge C \equiv A \wedge (B \wedge C) \\ (A \vee B) \vee C \equiv A \vee (B \vee C) \end{array}\right\}$ (associativity);
3. $\left.\begin{array}{l} A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C) \\ (A \vee B) \wedge C \equiv (A \wedge C) \vee (B \wedge C) \end{array}\right\}$ (distributivity of $\wedge$);
4. $\left.\begin{array}{l} A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C) \\ (A \wedge B) \vee C \equiv (A \vee C) \wedge (B \vee C) \end{array}\right\}$ (distributivity of $\vee$);
5. $\left.\begin{array}{l} A \wedge A \equiv A \\ A \vee A \equiv A \end{array}\right\}$ (idempotency);
6. $\left.\begin{array}{l} A \wedge (A \vee B) \equiv A \\ (A \vee B) \wedge A \equiv A \\ A \vee (A \wedge B) \equiv A \\ (A \wedge B) \vee A \equiv A \end{array}\right\}$ (adjunctivity);
7. $\left.\begin{array}{l} \neg(A \wedge B) \equiv (\neg A) \vee (\neg B) \\ \neg(A \vee B) \equiv (\neg A) \wedge (\neg B) \end{array}\right\}$ (De Morgan's law);
8. $\left.\begin{array}{l} \neg\top \equiv \bot \\ \neg\bot \equiv \top \end{array}\right\}$ (constant negation);
9. $\left.\begin{array}{l} A \wedge \bot \equiv \bot \\ \bot \wedge A \equiv \bot \end{array}\right\}$ (least element);
10. $\left.\begin{array}{l} A \vee \top \equiv \top \\ \top \vee A \equiv \top \end{array}\right\}$ (greatest element);
11. $\left.\begin{array}{l} A \wedge \top \equiv A \\ \top \wedge A \equiv A \end{array}\right\}$ (1-element);
12. $\left.\begin{array}{l} A \vee \bot \equiv A \\ \bot \vee A \equiv A \end{array}\right\}$ (0-element);
13. $\left.\begin{array}{l} \neg A \vee A \equiv \top \\ A \vee \neg A \equiv \top \end{array}\right\}$ (excluded middle);
14. $\left.\begin{array}{l} \neg A \wedge A \equiv \bot \\ A \wedge \neg A \equiv \bot \end{array}\right\}$ (contradiction);
15. $\neg\neg A \equiv A$ (double negation);
16. $\left.\begin{array}{l} (\forall q)\top \equiv \top \\ (\exists q)\top \equiv \top \\ (\forall q)\bot \equiv \bot \\ (\exists q)\bot \equiv \bot \end{array}\right\}$ (constant quantification);

17. $\left.\begin{array}{l}(\forall q)q \equiv \bot \\ (\exists q)q \equiv \top \\ (\forall q)\neg q \equiv \bot \\ (\exists q)\neg q \equiv \top\end{array}\right\}$ (literal quantification);

18. $\left.\begin{array}{l}(\exists q)B \equiv B[q/\top] \vee B[q/\bot] \\ (\forall q)B \equiv B[q/\top] \wedge B[q/\bot]\end{array}\right\}$ (quantifier elimination);

19. $(Qq)(Qr)A \equiv (Qr)(Qq)A$ (quantifier commutability);

20. $(Qq)(Qq)A \equiv (Qq)A$ (quantifier idempotency);

21. $\left.\begin{array}{l}((Qq)A) \wedge B \equiv (Qq)(A \wedge B) \\ B \wedge ((Qq)A) \equiv (Qq)(A \wedge B) \\ ((Qq)A) \vee B \equiv (Qq)(A \vee B) \\ B \vee ((Qq)A) \equiv (Qq)(A \vee B)\end{array}\right\} q \notin \text{free}(B)$ (binary quantifier shifting);

22. $\neg((Qq)A) \equiv (\overline{Q}q)(\neg A)$ (unary quantifier shifting);

23. $\left.\begin{array}{l}((\forall q)A) \wedge ((\forall r)B) \equiv (\forall q)(A \wedge B[r/q]) \\ ((\exists q)A) \vee ((\exists r)B) \equiv (\exists q)(A \vee B[r/q])\end{array}\right\}$ (quantifier fusion).

Proof. All equivalences from the above theorem can be easily proved by using the semantic definitions. □

One important insight that can be obtained from Equivalence 18 is that, compared with ordinary propositional logic, quantified propositional logic does not offer any more expressive power, since any QBF can be rewritten to a logically equivalent boolean formula by using the mentioned equivalences. Some formulas do, however, grow exponentially during this process (see Example 1.2.3). There exists no translation with polynomial time complexity, unless NP=PSPACE.

Example 1.2.3 (Exponential Growth during Quantifier Elimination) Consider the formula

$$A_n = (\forall q_0) \ldots (\forall q_n)(q_0 \vee \ldots \vee q_n),$$

for $n \in \mathbb{N}$. Furthermore, let $B_0 = \bot$ and $B_1 = \top$. Then iterated applications of the quantifier elimination rule eventually produces the formula

$$\bigwedge_{i=0}^{2^n} (B_{a_{i,1}} \vee \ldots \vee B_{a_{i,n}}),$$

where the concatenation $a_{i,1} \ldots a_{i,n}$ is the binary representation of the natural number i, for $0 \leq i \leq 2^n$. Put in another way, the resulting formula contains all 2^n clause-like disjunctions[7] that can be generated by selecting n times one of the formulas $\top$ or $\bot$.

As we have seen in Theorem 1.2.4, both $\vee$ and $\wedge$ are commutative, associative and idempotent. It therefore makes sense to abstract over these properties and allow an interpretation of clauses as sets of literals and of c-conjunctions as sets of clauses.

[7] Note that clauses cannot have $\top$ as a sub-formula, and the only clause that has $\bot$ as a sub-formula is the empty clause.

Convention 1.2.1 (Set Notion for Clauses and C-Conjunctions)
Unless noted otherwise, we regard clauses as sets[8] of literals and c-conjunctions as sets of clauses.

When writing down such sets, we would be well advised to disambiguate clauses, c-conjunctions and general sets of formulas.

Notation 1.2.1 (Set Notation for Clauses and C-Conjunctions)

1. We write $\langle L_1, \ldots, L_n \rangle$ to denote clauses that consist of (possibly duplicate) occurrences of the literals $L_1, \ldots, L_n$.
2. We write $[C_1, \ldots, C_n]$ to denote c-conjunctions that consist of (possibly duplicate) occurrences of the clauses $C_1, \ldots, C_n$.
3. Apart from the special parentheses, both of these constructs are to be regarded as sets.

As can easily be seen by the quantifier shifting and quantifier idempotency rules of Theorem 1.2.4, we may represent certain adjacent quantifiers as sets of variables that share the same quantifier symbol.

Notation 1.2.2 (Set Notation for Quantifiers)
We write $(Q\{q_1, \ldots, q_n\})A$ to denote formulas $(Qf(q_1)) \ldots (Qf(q_n))A$, where f is some permutation of the variables $q_1, \ldots, q_n$.

Example 1.2.4 (Set Notations)
Consider the formula

$$A = (\forall v_1)(\forall v_2)(\exists v_3)((v_1 \vee (\neg v_2 \vee v_1)) \wedge ((\neg v_2 \vee v_2) \vee v_3))$$

Under the application of Notations 1.2.1 and 1.2.2, A would be represented as

$$(\forall\{v_1, v_2\})(\exists\{v_3\})[\langle v_1, \neg v_2 \rangle, \langle \neg v_2, v_2, v_3 \rangle].$$

1.2.3 Formula Cleansing

We have already introduced the notions of a *change of bound variables* and of an *elimination of superfluous quantifiers*. Now, after we have introduced the semantics of QBFs, we will see how these notions, viewed as operations, can be used to cleanse formulas. First, we show that either operation leaves the semantic value of a formula unaffected.

Theorem 1.2.5 (Change of Bound Variables)
Let A' be a formula that is obtained from another formula A by a change of variables bound in some position π into r. Then it holds that $A' \equiv A$.

Proof. By Definition 1.1.29, $A|_\pi = (Qq)B$ and $A' = A[(Qr)(B[q/r])]_\pi$, where $r \notin \text{free}(B)$. Since $r \notin \text{free}(B)$, it is easy to see that

$$\begin{aligned} v_{\text{qbf}}((Qr)(B[q/r])) &= v_{\text{qbf}}(B[q/r, r/\top] \circ B[q/r, r/\bot], \phi) = \\ &= v_{\text{qbf}}(B[q/\top] \circ B[q/\bot], \phi) = v_{\text{qbf}}((Qq)B), \end{aligned}$$

[8] We mean genuine sets, not multi-sets.

i.e., $(Qq)B$ and $(Qr)(B[q/r])$ are equivalent. Therefore, by way of the Equivalent Replacement Theorem, it follows that $A' \equiv A$. □

Theorem 1.2.6 (Elimination of Superfluous Quantifier)
Let A' be a formula that is obtained from another formula A by elimination of a superfluous quantifier. Then $A' \equiv A$ holds.

Proof. By Definition 1.1.30, there is some position π, such that $A|_\pi = (Qq)B$ with $q \notin \text{free}(B)$ and $A' = A[B]_\pi$. Since $q \notin \text{free}(B)$, it is easy to see that

$$v_{\text{qbf}}((Qq)B) = v_{\text{qbf}}(B[q/\top] \circ B[q/\bot], \phi) = v_{\text{qbf}}(B \circ B, \phi) = v_{\text{qbf}}(B, \phi),$$

i.e., $(Qq)B$ and B are equivalent. Therefore, by way of the Equivalent Replacement Theorem, it follows that $A' \equiv A$. □

By a simple inductive argument, it is now easy to show that the semantics of a formula is preserved by any finite number of replacements of bound variables and eliminations of superfluous quantifiers. This allows us to construct a simple cleansing algorithm.

Algorithm 1.2.1 (Formula Cleansing)
Signature: $\mathcal{F} \to \dot{\mathcal{F}}$
Let A be a formula. Perform the following operations on A:

1. For all positions π in A' with $A'|_\pi = (Qq)B$, for some q and B, perform a change of variables bound at position π into a fresh variable.
2. While applicable, perform eliminations of superfluous quantifiers. Call the resulting formula A'.

Theorem 1.2.7 (Formula Cleansing Algorithm)
Algorithm 1.2.1 always terminates. If A is a formula, then the formula obtained by applying the algorithm to it is cleansed and logically equivalent to A.

Proof.

Termination The first step of Algorithm 1.2.1 must terminate, because any formula has only a finite number of positions. In Step (2) of the algorithm, each elimination of a superfluous quantifier reduces the cardinality of the formula. But cardinalities of formulas are natural numbers, so this step must eventually terminate too.

Correctness Since the formula is only modified by steps of changes of bound variables and elimination of superfluous quantifiers, the resulting formula must, by Theorems 1.2.3 and 1.2.6, be equivalent to the original formula. Step (1) assures that all quantifiers are different from each other and from any free variables in the formula. Therefore, Part 1(a) and the first half of Part 1(b) of Definition 1.1.25 are satisfied.

Because all bound variables are renamed along with their associated quantifiers in Step (1), and because any quantifiers without associated bound variables are eliminated in Step (2), the second half of Part (1b) of Definition 1.1.25 is satisfied too. Hence the application of Algorithm 1.2.1 to a formula produces an equivalent cleansed formula.

□

The fact that there are cleansing transformations with polynomial time complexity, which, moreover, leave the syntactic structure of a formula essentially unchanged, allows us to do most of our work using cleansed formulas. Therefore, we impose the following convention.

Convention 1.2.2 (Assumption of Cleansed Formulas)
When talking about formulas, we always assume that appropriate cleansing has been performed.

1.3 Reasoning Tasks

Looking at the semantic definitions of satisfiability, validity, logical entailment, and logical equivalence that were given in the previous section, we can identify at least four basic reasoning tasks that mirror these semantic concepts. There may be more, but for our purpose, these four tasks will suffice. We state them as decision problems.

1. $\mathcal{L}_{sat} = \{A \mid A \in \mathcal{F}, \mathcal{M}(A) \neq \emptyset\}$ (all satisfiable QBFs);
2. $\mathcal{L}_{valid} = \{A \mid A \in \mathcal{F}, \mathcal{M}(A) = \mathcal{I}\}$ (all valid QBFs);
3. $\mathcal{L}_{unsat} = \{A \mid A \in \mathcal{F}, \mathcal{M}(A) = \emptyset\}$ (all unsatisfiable QBFs);
4. $\mathcal{L}_{\equiv} = \{(A, B) \mid A, B \in \mathcal{F}, A \equiv B\}$ (Equality Theory of QBFs).

The problems of deciding the languages $\mathcal{L}_{unsat}$ and $\mathcal{L}_{\equiv}$ can be reduced to the problem of deciding language $\mathcal{L}_{valid}$, by pushing the semantic negation (resp. equivalence) down to the object level.

Theorem 1.3.1 (Object Level Negation and Equivalence)
1. A is unsatisfiable, iff $\neg A$ is valid;
2. $A \equiv B$, iff $(A \leftrightarrow B)$ is valid.

Proof.

1. By the semantics of QBFs, A is unsatisfiable, iff it has no model, i.e., $v_{\text{qbf}}(A, \phi) = 0$, for every $\phi \in \mathcal{I}$. In that case, however, $v_{\text{qbf}}(\neg A, \phi) = 1 - v_{\text{qbf}}(A, \phi) = 1$, for every $\phi \in \mathcal{I}$, i.e., $\neg A$ is valid. If, on the other hand, A is satisfiable, then there is some $\phi \in \mathcal{I}$ such that $v_{\text{qbf}}(A, \phi) = 1$, and $v_{\text{qbf}}(\neg A, \phi) = 1 - v_{\text{qbf}}(A, \phi) = 0$, hence A is not valid.
2. The formula $C = (A \leftrightarrow B)$ is a shorthand for $(\neg A \vee B) \wedge (\neg B \vee A)$. If we expand $v_{\text{qbf}}(C, \phi)$, we obtain

   $$\min(\max(1 - v_{\text{qbf}}(A, \phi), v_{\text{qbf}}(B, \phi)), \max(1 - v_{\text{qbf}}(B, \phi), v_{\text{qbf}}(A, \phi))).$$

   A closer inspection of this term shows that it evaluates to 1, iff $v_{\text{qbf}}(A, \phi) = v_{\text{qbf}}(B, \phi)$. Therefore, A and B share the same set of models, if, and only if, C is valid.

□

Furthermore, we may decide $\mathcal{L}_{sat}$ and $\mathcal{L}_{valid}$ by deciding corresponding sublanguages of closed formulas, as can be seen by the last two parts of the next theorem.

Lemma 1.3.1 (Constant Substitution)
Let A be a formula, q a variable and ϕ an interpretation. Furthermore, let ϕ_1 denote the interpretation with $\phi_1(q) = 1$ and $\phi_1(r) = \phi(r)$ for all $r \neq q$. Likewise, let ϕ_0 denote the interpretation with $\phi_0(q) = 0$ and $\phi_0(r) = \phi(r)$ for all $r \neq q$. Then

1. $\phi \models A[q/\top]$, iff $\phi_1 \models A$;
2. $\phi \models A[q/\bot]$, iff $\phi_0 \models A$.

Proof.

1. For ϕ_1, it holds that $v_{\text{qbf}}(q, \phi_1) = v_{\text{qbf}}(\top, \phi)$, for any interpretation ϕ. By considering the evaluation function for formulas, it is then easy to see that $v_{\text{qbf}}(A, \phi_1) = v_{\text{qbf}}(A[q/\top], \phi)$, for any interpretation ϕ, i.e., $\phi \models A[q/\top]$, iff $\phi_1 \models A$.
2. The proof of the second part of the lemma is analogous.

□

Theorem 1.3.2 (Formula Closing)
Let A be a formula with $\text{free}(A) = \{q_1, \ldots, q_n\}$, and let q be a variable. Then

1. A is satisfiable, iff $(\exists q)A$ is satisfiable;
2. A is valid, iff $(\forall q)A$ is valid;
3. A is satisfiable, iff $(\exists\{q_1, \ldots, q_n\})A$ is satisfiable;
4. A is valid, iff $(\forall\{q_1, \ldots, q_n\})A$ is valid.

Proof.

1. The formula $(\exists q)A$ is satisfiable, iff there exists some interpretation ϕ such that

$$v_{\text{qbf}}((\exists q)A) = \max(v_{\text{qbf}}(A[q/\top], \phi), v_{\text{qbf}}(A[q/\bot], \phi)) = 1,$$

i.e., iff at least one of the terms $v_{\text{qbf}}(A[q/\top], \phi)$ and $v_{\text{qbf}}(A[q/\bot], \phi)$ evaluates to 1. By Lemma 1.3.1, this is exactly the case, iff at least one of the statements, $\phi_1 \models A$ and $\phi_0 \models A$ is true, which only happens, iff A is satisfiable.

2. The formula $(\forall q)A$ is valid, iff for any interpretation ϕ, it holds that

$$v_{\text{qbf}}((\forall q)A) = \min(v_{\text{qbf}}(A[q/\top], \phi), v_{\text{qbf}}(A[q/\bot], \phi)) = 1,$$

   i.e., iff both of the terms $v_{\text{qbf}}(A[q/\top], \phi)$ and $v_{\text{qbf}}(A[q/\bot], \phi)$ evaluate to 1. By Lemma 1.3.1, this is exactly the case, iff both of the statements, $\phi_1 \models A$ and $\phi_0 \models A$ are true, which only happens, iff A is valid.

3. Part (3) of the theorem can be show easily by induction on the number n of quantifiers, using Part (1), which has been proven above.

4. Part (4) of the theorem can be show easily by induction on the number n of quantifiers, using Part (2), which has been proven above.

□

For closed QBFs, however, the notions of satisfiability and validity coincide.

Theorem 1.3.3 (Coincidence of Satisfiability and Validity)
Any closed formula A is valid, iff it is satisfiable.

Proof. It is an immediate consequence of Definition 1.2.1 that $\mathcal{I} \neq \emptyset$, i.e., that any formula has at least one interpretation. From this, it follows, by Definition 1.2.3, that any valid formula is also satisfiable.

For the second direction of the equivalence, we give an informal argument. Assume that A is closed. Then any occurrence of a variable q at position π of A has an associated quantifier position $\rho = \text{quant}(A, \pi)$. It follows from Definition 1.1.23 that $\rho \preceq \pi$.

It is now easy to see, by the definition of the evaluation functions for formulas, that every variable occurrence eventually gets substituted by constants. Therefore, the semantic value of the formula must be independent of the interpretation under which it is evaluated. If a formula A is satisfiable, then $v_{\text{qbf}}(A, \phi) = 1$, for some interpretation ϕ, but by the above argument, this implies that $v_{\text{qbf}}(A, \phi) = 1$, for any interpretation ϕ. □

From Theorems 1.3.1, 1.3.2 and 1.3.3, the construction of a polynomial-time algorithm that reduces any instance of our four basic reasoning problems into the problem of deciding the satisfiability of a closed formula, should be obvious. So we define a single standard decision problem.

Definition 1.3.1 (QSAT)
By *QSAT* we mean the problem of deciding the language

$$\mathcal{L}_{csat} = \{A \mid A \in \mathcal{F},\ \text{free}(A) = \emptyset,\ \mathcal{M}(A) \neq \emptyset\}.$$

QSAT is certainly decidable. To check a given closed formula A for satisfiability, the evaluate A under one arbitrary "dummy" interpretation[9] ϕ is sufficient. But what complexity is associated with QSAT? Such results are usually stated w.r.t. formulas in *Prenex Normal Form* (c.f. Section 1.4).

[9] It follows from Theorem 1.3.3 that the choice of ϕ is not important.

Theorem 1.3.4 (Complexity of QSAT)
Consider the set of formulas

$$\mathcal{A}_n^{Q_1} = \{\Gamma_1 \dots \Gamma_n B \mid \Gamma_i = (Q_i\{q_{i_1}, \dots, q_{i_m}\}), Q_{i+1} = \overline{Q_i}, B \in \mathcal{B} \text{ for } 1 \leq i < n\},$$

for $n \in \mathbb{N}$. Then

1. the problem of deciding $\mathcal{L}_{csat} \cap \mathcal{A}_n^{\exists}$ is Σ_n^P-complete;[10]
2. the problem of deciding $\mathcal{L}_{csat} \cap \mathcal{A}_n^{\forall}$ is Π_n^P-complete;
3. the problem of deciding $\mathcal{L}_{csat}$ is PSPACE-complete.

Proof. See, e.g., [44] and [54]. □

Morale 1.3.1 (Minimization of Quantifier Alternations)
When given the choice of a set $\mathcal{A}$ of equivalent formulas as candidates for a satisfiability test, we shall *ceteris paribus* choose a formula with a minimum number of quantifier alternations, i.e., which is member of $\mathcal{A}_n^{\exists}$ (resp. $\mathcal{A}_n^{\forall}$), with n minimal w.r.t. $\mathcal{A}$.

1.4 Normal Forms

1.4.1 Common Normal Forms

When working with formulas, it is sometimes practical to consider only formulas that are in a normal form, i.e., which are irreducible w.r.t. a certain set of reduction rules or, more generally, w.r.t. a certain algorithm that performs some reduction. Formally, we define the concept of a normal form w.r.t. to a reduction relation that describes all allowed reductions.

Definition 1.4.1 (Normal Form)
1. Let A be a formula and $\rightarrowtail$ a binary relation over $\mathcal{F}$. Then A is said to be in *normal form* w.r.t. $\rightarrowtail$, if there is no formula B such that $A \rightarrowtail B$.
2. Let A be a formula such that $A \rightarrowtail^* B$, where B is in normal form.[11] We then call B *a normal form of* A.

We introduce normal forms in terms of their syntactic properties rather than via their corresponding reduction relation.

The normal form transformations which we are interested in, either

1. preserve the semantics of formulas completely by producing formulas that are *logically equivalent* to the original formula, or
2. preserve the semantics of formula partially by producing formulas that are *satisfiability equivalent* to the original formula.

[10] Appendix A.2.2 contains a short introduction to the complexity classes Σ_n^P and Π_n^P.
[11] See Appendix A.1.1 for an explanation of the notation $A \rightarrowtail^* B$.

Definition 1.4.2 (Constant Normal Form)
A formula A is said to be in *constant normal form (CoNF)*, if neither $\top$ nor $\bot$ are proper sub-formulas of A.

Constant normal form restricts the usage of the propositional constants to just the two formulas $\top$ and $\bot$. In all other cases, these symbols can be eliminated with little effort at the prospect of substantially shorter formulas.

The simplest normal form that is commonly introduced in literature is the *negation normal form*. Its main advantage is that it simplifies reasoning about the polarities of sub-formulas.

Definition 1.4.3 (Negation Normal Form)
A formula A is said to be in *negation normal form (NNF)*, if every sub-formula B of A with $\mathrm{root}(B) = \neg$ is a literal.

Another useful normal form is *prenex normal form*, which is characterized by its isolation of a quantifier prenex from a purely propositional formula part.

Definition 1.4.4 (Prenex, Matrix, Prenex Normal Form)
A formula A is said to be in *prenex normal form (PNF)*, if it has the structure

$$(Q_1 q_1) \ldots (Q_n q_n) M,$$

where $M \in \mathcal{B}$. In that case, we call $(Q_1 q_1) \ldots (Q_n q_n)$ the *prenex* and M the *matrix* of A and denote these components by $\mathrm{pren}(A)$ and $\mathrm{matr}(A)$, respectively.

The next normal form is taken from propositional logic.

Definition 1.4.5 (Conjunctive Normal Form)
A (quantified boolean) formula is said to be in *conjunctive normal form (CNF)*, if it is a c-conjunction.

Obviously, formulas in conjunctive normal form are purely propositional. The normal form we are most interested in is the special case of those formulas in prenex normal form, where the matrix is in conjunctive normal form:

Definition 1.4.6 (Prenex Conjunctive Normal Form)
A PNF formula A is in *prenex conjunctive normal form (PCNF)*, if $\mathrm{matr}(A)$ is in CNF.

1.4.2 Simple Transformations

Having introduced the most important normal forms for formulas in terms of structural properties, we are, of course, interested in effective methods to transform arbitrary formulas[12] into logically or satisfiability equivalent formulas in normal form.

[12] Recall that we hereby mean—by way of Convention 1.2.2—arbitrary *cleansed* formulas.

Algorithm 1.4.1 (Standard CoNF Transformation)
Signature: $\dot{\mathcal{F}} \to \dot{\mathcal{F}}$
An algorithm for transforming a given cleansed formula into a logically equivalent CoNF formula is given by the following system of rewrite rules: least element, greatest element, constant negation, constant quantification, 1-element and 0-element.

Algorithm 1.4.2 (Standard NNF Transformation)
Signature: $\dot{\mathcal{F}} \to \dot{\mathcal{F}}$
An algorithm for transforming a given cleansed formula into a logically equivalent NNF formula is given by the union of the rewrite rules from Algorithm 1.4.1 and the following set of rules: De Morgan's law, double negation and unary quantifier shifting.

Algorithm 1.4.3 (Standard PNF Transformation)
Signature: $\dot{\mathcal{F}} \to \dot{\mathcal{F}}$
An algorithm for transforming a given cleansed formula into a logically equivalent PNF formula is given by the following system of rewrite rules: binary and unary quantifier shifting.

Algorithm 1.4.4 (Standard CNF Transformation)
Signature: $\dot{\mathcal{F}} \to \dot{\mathcal{F}}$
An algorithm for transforming a given boolean formula into a logically equivalent CNF formula is given by the following system of rewrite rules: associativity, distributivity of $\vee$, De Morgan's law, 1-element, 0-element, least element, greatest element, constant negation and double negation.

Algorithm 1.4.5 (Standard PCNF Transformation)
Signature: $\dot{\mathcal{F}} \to \dot{\mathcal{F}}$
An algorithm for transforming a given cleansed formula into a logically equivalent PCNF formula is given by the union of the rewrite systems from Algorithms 1.4.3 and 1.4.4.

Unfortunately, the application of the Standard (P)CNF Transformation is limited by the possibility of an exponential formula growth, as follows easily from Example 1.4.1.

Example 1.4.1 (Exponential Growth during Distribution)
Consider the formula

$$A_n = \bigvee_{i=1}^{n} (q_{i,0} \wedge q_{i,1}),$$

for $n \in \mathbb{N}$. Then iterated application of the distributivity rule eventually produces, modulo commutativity and associativity, the formula

$$\bigwedge_{i=0}^{2^n} (q_{1,a_{i,1}} \vee \ldots \vee q_{n,a_{i,n}}),$$

where the concatenation $a_{i,1} \ldots a_{i,n}$ is the binary representation of the natural number i, for $0 \leq i \leq 2^n$. Put in another way, the resulting formula contains all 2^n clauses that can, modulo commutativity and associativity, be generated by selecting one literal from each of the n formulas $q_{i,0} \wedge q_{i,1}$ in A_n.

The method of *Structure Preserving PCNF Transformation* avoids the exponential formula growth illustrated by the above example by replacing complex proper sub-formulas by new variables (called *labels*) and adding appropriate short definitions.

A propositional version of this method is given in [46], whereas [40] presents a version for first order logic. Eder investigates an alternative structure preserving normal form transformation in [17]. In [4], the authors discuss the normal form transformations for first oder logic, laying special emphasis on the their influence on proof lengths.

Algorithm 1.4.6 (Structure Preserving PCNF Transformation)
Signature: $\dot{\mathcal{F}} \rightarrow \dot{\mathcal{F}}$
An algorithm for transforming a given cleansed formula A into a satisfiability equivalent PCNF formula is given by performing the following steps:

1. Apply the Standard CoNF and Standard PNF Transformations.
2. Rewrite the formula using the associativity rule. The formula is now in PNF and has the form
$$(Q_1 q_1) \ldots (Q_n q_n) M,$$
with $M = \bigwedge_{i=1}^{m} A_i$, where $\mathrm{root}(A_i) \neq \wedge$ for all $1 \leq i \leq m$.
3. If M is in CNF, then terminate and return the transformed formula.
4. If some A_i is a simple formula or short definition, transform A_i by applying the Standard CNF transformation.
5. If some A_i is neither a clause, nor a simple formula, nor a short definition, choose a simple sub-formula A of A_i, i.e., let $A = A_i|_\pi$ such that A is simple. If $\mathrm{pol}(A) = +1$ (resp. $\mathrm{pol}(A) = -1$) then replace all positive (resp. negative) occurrences of A in M by a fresh variable q (called a *label* in this context). Replace M by $(\exists q)((q \rightarrow A) \wedge M)$ (resp. $(\exists q)((A \rightarrow q) \wedge M)$).
6. Continue with Step 2.

The algorithm can be optimized, e.g., by integration of the literal quantification rules from Theorem 1.2.4. However, for our purpose, the above version is sufficient.

Theorem 1.4.1 (Complexity of PCNF Transformation)
The time complexity of computing, for a given formula, a satisfiability equivalent formula in PCNF is polynomial in the cardinality of A.

This result justifies the use of decision procedures that are restricted to formulas in PCNF for deciding QSAT, using a PCNF transformation algorithm with polynomial time complexity as preprocessor.

Notation 1.4.1 (Special Sets of Formulas)
1. We use the symbol $\check{\mathcal{F}}$ to designate the set of closed, cleansed formulas in PCNF.[13]

[13] We choose the accent "˘" because of its resemblance to the symbol"$\bigvee$", which should remind the reader of the clauses in a PCNF matrix.

2. We use the symbol $\tilde{\mathcal{F}}$ to designate the set of closed, cleansed formulas in NNF.[14]

[14] We choose the accent "~" because of its resemblance to the overstriking operator "‾", which should remind the reader of the literals in a NNF formula.

Chapter 2

The QDLL Procedure

In Section 1.3 we have identified QSAT as a fundamental reasoning problem of quantified propositional logic. Now we turn our attention to algorithms that are capable of deciding QSAT practically.

Many decision procedures for the *satisfiability problem* of propositional logic (SAT)—like semantic tableaux or propositional resolution [24]—have extensions for quantified propositional logic [7, 35]. The most famous algorithm, however, has its roots in the work of M. Davis and H. Putnam [14, 15], who were looking for a procedure for efficient propositional satisfiability testing as part of their effort to devise an efficient proof procedure for first order logic.

The original procedure was modified as soon as the first actual implementation was found to require too much memory. It is this modified procedure, proposed by M. Davis, G. Logemann and D. Loveland in [13], which has been adopted as a standard decision procedure by the SAT community and which is, amongst its members, known as *Davis-Putnam*, *DLL* and *DPLL Procedure*. This procedure has recently been extended for QSAT. In this work, we will refer to this extended procedure as QDLL procedure.

We first present an abstract version of the procedure in Section 2.1. Then we describe the behavior of practical implementations of QDLL in Section 2.2. In the same section, we also present some advanced techniques that are used in modern QDLL procedures. We conclude the section with a presentation of the techniques that are integrated into the contemporary decision procedures used in the experimental evaluation of our prenexing strategies (c.f. Chapter 4).

2.1 The Basic Procedure

The QDLL procedure is a QSAT decision procedure for closed, cleansed quantified boolean formulas in prenex conjunctive normal form (recall that we write the set of all such formulas as $\check{\mathcal{F}}$). We have already seen in Section 1.3 that the restriction to cleansed, closed formulas does not amount to any real limitation.

In Section 1.4.2, we have described an efficient algorithm that achieves a translation of such formulas into PCNF. It is thus easy to conceive a simple preprocessor that puts an arbitrary formula into a form that QDLL can deal with.

Being aware of this simple way to obtain a general QSAT decision procedure, many scientists completely abstract from the necessary preprocessing step and commit themselves to the exclusivity of formulas in PCNF. In this work, however, we are, on the contrary, more concerned with the preprocessing step, in particular with the implementational options for prenexing. For this, however, it is nonetheless necessary to obtain a rudimentary understanding of the QDLL procedure first.

The QDLL procedure is fundamentally based on the quantifier elimination rules from Theorem 1.2.4, but does, of course, also implicitly rely on many other rules and theorems, most notably perhaps on the Monotonic Replacement Theorem. One central idea of the procedure is given by the observation that the substitutions $A[p/\bot]$ and $A[p/\top]$ that are introduced by the quantifier elimination rules, can be performed very efficiently on CNF matrices by the following three primitive operations.

Definition 2.1.1 (Functions for Removing Parts of a Formula)
Let A be a formula in PCNF, L a literal in A and q a variable. Also, recall that may view clauses and c-conjunctions as sets (c.f. Convention 1.2.1).

1. The function $\mathrm{lrem}(\,\cdot,\cdot\,)$ (remove literal) is defined by
$$\mathrm{lrem}(A, L) = \mathrm{pren}(A)[C \setminus \langle L\rangle \mid C \in \mathrm{matr}(A)].$$
2. For sets $\mathcal{A} = \{L_1, \ldots, L_n\}$ of literals, we define
$$\mathrm{lrem}(A, \mathcal{A}) = \mathrm{lrem}(\ldots \mathrm{lrem}(A, L_1) \ldots, L_n).$$
3. The function $\mathrm{crem}(\,\cdot,\cdot\,)$ (remove clause) is defined by
$$\mathrm{crem}(A, L) = \mathrm{pren}(A)[C \mid C \in \mathrm{matr}(A), L \notin C].$$
4. For sets $\mathcal{A} = \{L_1, \ldots, L_n\}$ of literals, we define
$$\mathrm{crem}(A, \mathcal{A}) = \mathrm{crem}(\ldots \mathrm{crem}(A, L_1) \ldots, L_n).$$
5. The function $\mathrm{qrem}(\,\cdot,\cdot\,)$ (remove quantifier) is defined by
$$\mathrm{qrem}(A, q) = (Q_1(\mathcal{A}_1 \setminus \{q\})) \ldots (Q_n(\mathcal{A}_n \setminus \{q\}))\mathrm{matr}(A),$$
where $\mathrm{pren}(A) = (Q_1\mathcal{A}_1) \ldots (Q_n\mathcal{A}_n)$.
6. For sets $\mathcal{A} = \{q_1, \ldots, q_n\}$ of variables, we define
$$\mathrm{qrem}(A, \mathcal{A}) = \mathrm{qrem}(\ldots \mathrm{qrem}(A, q_1) \ldots, q_n).$$

Theorem 2.1.1 (Semantics of Removing Literals and Clauses)
Let A be a formula in PCNF. Then

1. $\mathrm{lrem}(A, L) \equiv \mathrm{pren}(A)(\mathrm{matr}(A)[L/\bot])$;
2. $\mathrm{crem}(A, L) \equiv \mathrm{pren}(A)(\mathrm{matr}(A)[L/\top])$.

Proof. Let A be a formula in PCNF, and let L be some literal. If $L \notin \mathrm{lit}(A)$, then the theorem holds trivially. So let us consider the interesting case where $L \in \mathrm{lit}(A)$.

1. Consider the formula

$$B = \mathrm{pren}(A)(\mathrm{matr}(A)[L/\bot]).$$

   By application of the 0-element rule from Theorem 1.2.4, we can remove all occurrences of the formula $\bot$ from B. Call the new formula B'. Since A is in PCNF, it cannot contain any occurrences of the formula $\bot$, so all occurrences of $\bot$ in B correspond to occurrences of the literal L in A. So B' is the formula A, with all occurrences of the literal L removed, i.e., $B' = \mathrm{lrem}(A, L)$.

2. Consider the formula

$$B = \mathrm{pren}(A)(\mathrm{matr}(A)[L/\top]).$$

   By iterated application of the greatest element rule from Theorem 1.2.4, we can replace any clause that contains an occurrence of the formula $\top$ in B with the formula $\top$. Call the new formula B'.

   Next, by iterated application of the 1-element rule, we can remove any occurrence of the formula $\top$ from the formula (unless $\mathrm{matr}(B') = \top$). Call the new formula B''.

   Since A is in PCNF, it cannot contain any occurrence of the formula $\top$ inside clauses, so all occurrences of $\top$ in clauses of B correspond to occurrences of the literal L in A. For the same reason, A cannot contain any occurrence of the formula $\top$ (unless $\mathrm{matr}(A) = \top$). Therefore, all occurrences of the formula $\top$ in B' correspond to clauses in A that contain the literal L (unless $\mathrm{matr}(A) = \top$). So B'' is the formula A, with all occurrences of the literal L removed, i.e., $B'' = \mathrm{crem}(A, L)$.

□

Note that all of the rules from Definition 2.1.1 achieve a shortening of formulas, which qualifies them as useful reduction rules.

From Theorem 2.1.1, it is easy to see that, given a formula A in PCNF with

$$\mathrm{pren}(A) = (Q_1 q_1) \ldots (Q_{i-1} q_{i-1})(Q_i q_i)(Q_{i+1} q_{i+1}) \ldots (Q_n q_n)$$

and $Q_i = \exists$ (resp. $Q_i = \forall$), the formula $B_0 \vee B_1$ (resp. $B_0 \wedge B_1$) with the two immediate sub-formulas

$$B_0 = \mathrm{qrem}(\mathrm{lrem}(\mathrm{crem}(A, \overline{q_i}), q_i), q_i), \text{and}$$
$$B_1 = \mathrm{qrem}(\mathrm{lrem}(\mathrm{crem}(A, q_i), \overline{q_i}), q_i).$$

is exactly the formula obtained applying the quantifier elimination rule (for the variable q_i) on A.

The above observation is cast into the form of the two *elimination rules* (c.f. Definition 2.1.4), which provide the constant driving force behind the procedure.

Whenever no better mechanism is applicable, these rules unwaveringly chop the problem down into two simpler subproblems.

However, whereas the elimination rules are in fact indispensable, the heart of the procedure can be found in the mechanism of *unit propagation*, i.e., the successive reduction of a formula by the *unit rule*. Lastly, the two *pure rule*s, which allow QDLL to handle another special case efficiently, are usually also considered as part of the core procedure.

We will present all of these rules shortly, after some preparatory work.

Definition 2.1.2 (Tautological, Contradictory and Unit Clause)
A clause C of a closed formula A is called

1. *tautological*, if it contains a literal L and also its complement $\overline{L}$;
2. *contradictory* in A, if it is neither tautological, nor contains an existentially bound literal;
3. *unit* in A, if it contains exactly one existentially bound literal L, and for every literal $L' \neq L$ of C (which is, consequently, universally bound), it holds that $\mathrm{qpos}(A, L') \prec \mathrm{qpos}(A, L)$.

We define the basic operations of the QDLL procedure as rules that operate on a *semantic tree*. A semantic tree is essentially a vehicle which captures the structure of a search space that combines existential and universal branching. To understand the operations, it is useful to abstract from the concrete search strategy. However, in Section 2.2, we will discuss how actual implementations of QDLL perform a traversal of such a search space.

Definition 2.1.3 (Semantic Tree)
A *semantic tree* is a tree $(\mathcal{O}, \mathcal{E}, \mu, \nu)$, consisting of the following constituents.

1. a set of nodes $\mathcal{O} \subseteq \{1, 2\}^\star$;
2. a set of edges $\mathcal{E} \subseteq (\mathcal{O} \times \mathcal{O})$;
3. a node marking function $\mu : \mathcal{O} \to (\{\exists, \forall\} \cup \check{\mathcal{F}})$;
4. an edge marking function $\nu : \mathcal{E} \to \{v, \neg v \mid v \in \mathcal{V}\}$.

For the node marking function ν, we require that $\mu(\pi) \in \check{\mathcal{F}}$, for any leaf $\pi \in \mathcal{O}$, and $\mu(\pi) \in \{\exists, \forall\}$, for any inner node $\pi \in \mathcal{O}$. We denote the set of all semantic trees by $\mathcal{Y}$.

Example 2.1.1 (Semantic Tree)
Let

1. $\mathcal{O} = \{\varepsilon, 1, 11, 12, 2\}$,
2. $\mathcal{E} = \{(\varepsilon, 1), (\varepsilon, 2), (1, 11), (1, 12)\}$,
3. $\mu = \{(\varepsilon, \forall), (1, \exists), (11, [\langle\,\rangle]), (12, [\,]), (2, (\exists r)[\langle r \rangle])\}$, and

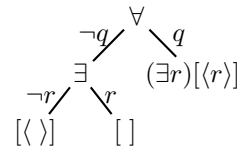

Figure 2.1: Graphical representation of the semantic tree from Example 2.1.1.

4. $\nu = \{((\varepsilon, 1), \neg q), ((\varepsilon, 2), q), ((1, 11), \neg r), ((1, 12), r)\}$.

Then $(\mathcal{O}, \mathcal{E}, \mu, \nu)$ is a semantic tree. A graphical representation of this tree is given in Figure 2.1.

Definition 2.1.4 (Fundamental QDLL Rules)
Let $\omega = (\mathcal{O}, \mathcal{E}, \mu, \nu)$ be a semantic tree and $\pi \in \mathcal{O}$ a node with $\mu(\pi) \in \check{\mathcal{F}}$. The each of the following rules can be used obtain, from ω, a new semantic tree $\omega' = (\mathcal{O}', \mathcal{E}', \mu', \nu')$.

1. *Tautology Rule:* If $\mu(\pi)$ contains a tautological clause C, then let

$$\mu'(\pi) = \mathrm{pren}(\mu(\pi))(\mathrm{matr}(\mu(\pi)) \setminus \{C\});$$
$$\mu'(\varrho) = \mu(\varrho), \text{ for any node } \varrho \neq \pi.$$

   All other parts of the tree remain unchanged, i.e., $\mathcal{O}' = \mathcal{O}$, $\mathcal{E}' = \mathcal{E}$ and $\nu' = \nu$.

2. *Contradiction Rule:* If $\mu(\pi)$ contains a contradictory clause C, then let

$$\mu'(\pi) = [\langle\ \rangle];$$
$$\mu'(\varrho) = \mu(\varrho), \text{ for any node } \varrho \neq \pi.$$

   All other parts of the tree remain unchanged, i.e., $\mathcal{O}' = \mathcal{O}$, $\mathcal{E}' = \mathcal{E}$ and $\nu' = \nu$.

3. *Unit Rule:* If $\mu(\pi)$ contains a unit clause C with the existentially bound literal L, then let

$$\mu'(\pi) = \mathrm{qrem}(\mathrm{lrem}(\mathrm{crem}(\mu(\pi), L), \overline{L}), \lfloor L \rfloor);$$
$$\mu'(\varrho) = \mu(\varrho), \text{ for any node } \varrho \neq \pi.$$

   All other parts of the tree remain unchanged, i.e., $\mathcal{O}' = \mathcal{O}$, $\mathcal{E}' = \mathcal{E}$ and $\nu' = \nu$.

4. *Existential Pure Rule:* If $\mu(\pi)$ contains an existentially bound pure literal L, then let

$$\mu'(\pi) = \mathrm{qrem}(\mathrm{crem}(\mu(\pi), L), \lfloor L \rfloor);$$
$$\mu'(\varrho) = \mu(\varrho), \text{ for any node } \varrho \neq \pi.$$

   All other parts of the tree remain unchanged, i.e., $\mathcal{O}' = \mathcal{O}$, $\mathcal{E}' = \mathcal{E}$ and $\nu' = \nu$.

5. *Universal Pure Rule:* If $\mu(\pi)$ contains a universally bound pure literal L, then let
$$\mu'(\pi) = \mathrm{qrem}(\mathrm{lrem}(\mu(\pi), L), \lfloor L \rfloor);$$
$$\mu'(\varrho) = \mu(\varrho), \text{ for any node } \varrho \neq \pi.$$
All other parts of the tree remain unchanged, i.e., $\mathcal{O}' = \mathcal{O}$, $\mathcal{E}' = \mathcal{E}$ and $\nu' = \nu$.

6. *Existential Elimination Rule:* If $\mu(\pi) = (\exists\{q_1, \ldots, q_n\})B$, then choose some i with $1 \leq i \leq n$, and let
$$\begin{array}{l}\mathcal{O}' = \mathcal{O} \cup \{\pi 1, \pi 2\};\\ \mathcal{E}' = \mathcal{E} \cup \{(\pi, \pi 1), (\pi, \pi 2)\};\\ \mu'(\pi) = \exists;\\ \mu'(\pi 1) = \mathrm{qrem}(\mathrm{lrem}(\mathrm{crem}(\mu(\pi), \overline{q_i}), q_i), q_i);\\ \mu'(\pi 2) = \mathrm{qrem}(\mathrm{lrem}(\mathrm{crem}(\mu(\pi), q_i), \overline{q_i}), q_i);\\ \mu'(\varrho) = \mu(\varrho) \text{ for } \varrho \notin \{\pi, \pi 1, \pi 2\};\\ \nu'(\pi, \pi 1) = \neg q_i;\\ \nu'(\pi, \pi 2) = q_i;\\ \nu'(\varrho, \sigma) = \nu(\varrho, \sigma) \text{ for } (\varrho, \sigma) \notin \{(\pi, \pi 1), (\pi, \pi 2)\}.\end{array}$$
In this context, q_i is called the *splitting variable*.

7. *Universal Elimination Rule:* If $\mu(\pi) = (\forall\{q_1, \ldots, q_n\})B$, then choose some i with $1 \leq i \leq n$, and let
$$\begin{array}{l}\mathcal{O}' = \mathcal{O} \cup \{\pi 1, \pi 2\};\\ \mathcal{E}' = \mathcal{E} \cup \{(\pi, \pi 1), (\pi, \pi 2)\};\\ \mu'(\pi) = \forall;\\ \mu'(\pi 1) = \mathrm{qrem}(\mathrm{lrem}(\mathrm{crem}(\mu(\pi), \overline{q_i}), q_i), q_i);\\ \mu'(\pi 2) = \mathrm{qrem}(\mathrm{lrem}(\mathrm{crem}(\mu(\pi), q_i), \overline{q_i}), q_i);\\ \mu'(\varrho) = \mu(\varrho) \text{ for } \varrho \notin \{\pi, \pi 1, \pi 2\};\\ \nu'(\pi, \pi 1) = \neg q_i;\\ \nu'(\pi, \pi 2) = q_i;\\ \nu'(\varrho, \sigma) = \nu(\varrho, \sigma) \text{ for } (\varrho, \sigma) \notin \{(\pi, \pi 1), (\pi, \pi 2)\}.\end{array}$$
Again, q_i is called the *splitting variable*. Note that the universal elimination rule differs from the existential elimination rule in the value of $\mu'(\pi)$.

Example 2.1.2 (Fundamental QDLL Rules)
Consider a semantic tree $\omega = (\mathcal{O}, \mathcal{E}, \mu, \nu)$, which contains some node π, with
$$\begin{array}{l}\mu(\pi) = (\forall\{q_1\})(\exists\{q_2\})\forall\{q_3, q_4\})(\exists\{q_5\})\\ \qquad [\langle \neg q_1, q_2, q_3, q_5\rangle, \langle \neg q_1, q_2, q_3, q_4, \neg q_5\rangle, \langle \neg q_1, \neg q_2, \neg q_3, q_5\rangle].\end{array}$$
We can then reduce ω by applying the *universal elimination rule* at π. The new tree[1] contains two additional nodes, π_0 and π_1, with $\nu'((\pi, \pi_0)) = \neg q_1$ and $\nu'((\pi, \pi_1)) = q_1$. Now
$$\mu'(\pi_0) = (\exists\{q_2\})\forall\{q_3, q_4\})(\exists\{q_5\})[\,],$$

[1] As usual, we designate derived structures by adding a prime symbol to the original names. In particular, we use this convention for trees and all their components.

and

$$\mu'(\pi_1) = (\exists\{q_2\})\forall\{q_3, q_4\})(\exists\{q_5\})[\langle q_2, q_3, q_5\rangle, \langle q_2, q_3, q_4, \neg q_5\rangle, \langle \neg q_2, \neg q_3, q_5\rangle].$$

Next, we reduce ω' by applying the *existential elimination rule* at π_1. The new tree contains two additional nodes, π_2 and π_3, with $\nu'((\pi, \pi_2)) = \neg q_2$ and $\nu'((\pi, \pi_2)) = q_1$. Now

$$\mu''(\pi_2) = (\forall\{q_3, q_4\})(\exists\{q_5\})[\langle q_3, q_5\rangle, \langle q_3, q_4, \neg q_5\rangle],$$

and

$$\mu''(\pi_3) = \forall\{q_3, q_4\})(\exists\{q_5\})[\langle \neg q_3, q_5\rangle].$$

We reduce ω'' by applying the *universal pure rule* at π_2 to eliminate the variable q_3, resulting in

$$\mu'''(\pi_2) = (\forall\{q_4\})(\exists\{q_5\})[\langle q_5\rangle, \langle q_4, \neg q_5\rangle].$$

We reduce ω''' by applying the *unit rule* at π_2, resulting in

$$\mu^{4\prime}(\pi_2) = (\forall\{q_4\})[\langle\,\rangle, \langle q_4\rangle].$$

Finally, we reduce $\omega^{4\prime}$ by applying the *contradiction rule* at π_2, resulting in

$$\mu^{5\prime}(\pi_2) = [\langle\,\rangle].$$

Definition 2.1.5 (Evaluation Function for Semantic Trees)
The evaluation function $v_{\mathrm{ao}} : (\mathcal{Y} \times \{0,1\}^\star) \to \{0,1\}$ is recursively defined by:

$$v_{\mathrm{ao}}((\mathcal{O}, \mathcal{E}, \mu, \nu), \pi) = \begin{cases} \min_{(\pi,\pi')\in\mathcal{E}} v_{\mathrm{ao}}((\mathcal{O}, \mathcal{E}, \mu, \nu), \pi') & \text{if } \mu(\pi) = \forall; \\ \max_{(\pi,\pi')\in\mathcal{E}} v_{\mathrm{ao}}((\mathcal{O}, \mathcal{E}, \mu, \nu), \pi') & \text{if } \mu(\pi) = \exists; \\ 1 & \text{if } \mu(\pi) = [\,]; \\ 0 & \text{if } \mu(\pi) = [\langle\,\rangle]; \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Example 2.1.3 (Evaluation Function for Semantic Trees)
Consider the semantic tree from Example 2.1.1. Then $v_{\mathrm{ao}}((\mathcal{O}, \mathcal{E}, \sigma, \mu), 1) = 1$, but $v_{\mathrm{ao}}((\mathcal{O}, \mathcal{E}, \sigma, \mu), \varepsilon)$ is undefined.

It is not hard to see that semantic trees bear a strong resemblance to the computational trees of alternating Turing machines, which are explained in Section A.2.1, and the evaluation function $v_{\mathrm{ao}}(\,\cdot, \cdot\,)$ for semantic trees mirrors the semantic concept of *acceptance*. In fact, the complexity results for QSAT can be used to establish a lower bound for the APTIME, the polynomial time complexity class for alternating Turing machines [49].

As we can see, there is no semantic value for leaves that are marked with formulas other that $[\,]$ or $[\langle\,\rangle]$, i.e., $v_{\mathrm{ao}}(\,\cdot, \cdot\,)$ is only partially defined. The definedness for the inner nodes depends on that of the leaves, e.g., the semantic value of a node that is marked by $\vee$ is defined, if either both of its successors have a semantic value of 0, or if least one of them has a semantic value of 1, because $\min(1, x) = \min(x, 1) = 1$, if x is restricted to $\{0, 1\}$.

Algorithm 2.1.1 (QDLL Procedure)
Signature: $\check{\mathcal{F}} \to \{0, 1\}$
From the input formula A, construct an initial semantic tree

$$\omega = (\{\varepsilon\}, \emptyset, \{(\varepsilon, A)\}, \emptyset).$$

Then keep on expanding ω by applying the QDLL rules from Definition 2.1.4, until the value of $v_{\mathrm{ao}}(\omega, \varepsilon)$ is defined. At that point, return that value.

The correctness proof for the QDLL procedure is complex and goes beyond the scope of this work, so we give the corresponding theorem without proof.

Theorem 2.1.2 (QDLL Procedure)
Algorithm 2.1.1 terminates and returns the value 1, if its input formula is satisfiable, and 0 otherwise.

2.2 Advanced Techniques and Implementation

Semantic trees are a conceptual tool to explain the behavior of the QDLL procedure, but they are not suitable for a direct implementation of the above algorithm. Such trees can become huge, and it is no use keeping them stored in memory anyway.

Rather, implementations of QDLL usually perform the expansion of nodes in a depth first fashion. After some branch has been completely expanded, such implementations can backtrack and then follow another branch, if necessary. The corresponding semantic trees can then be seen as a map of the search space that has been explored at any given time.

Observe that the exploration of the search space in a depth-first fashion enables the computational complexity of the proof procedures to located in PSPACE. Retaining complete semantic trees, or even performing a depth first search, would cause an exponential space complexity.

2.2.1 Dependency Directed Backtracking

If, upon backtracking, the semantic value of a node has already become defined after exploring the first branch, then it is not necessary to explore the second branch, and the procedure can backtrack further.

The simplest way to prune the search space in this way follows from the semantics of existential (resp. universal) nodes. If the semantics of the first branch can be determined to be 1 (resp. 0), then the whole subtree necessarily evaluates to 1 (resp. 0).

However, there are also more intricate methods that are capable of eliminating the need to check the second branch, even where the simple semantic method cannot help, and the technique of *Dependency Directed Backtracking* is one such method.

This technique, which is also known as Backjumping, is explained in [37] (but see also [33]). We can distinguish two different forms of this technique, one

that is applicable to the existential nodes of a semantic tree, and one that is applicable to the universal nodes. In both cases, the successful application of the technique eliminates the need to further expand the second successor of a given node, if the first successor has already been analyzed.

In the case of an existential node, the technique works by extracting, from applications of the contradiction rule, sets of variables that have certainly contributed to the contradiction. These are the existentially bound variables of one of the initial clauses, from which a contradictory clause was derived.

In [37], two different methods to pass this information upwards upon backtracking are proposed. When backtracking from a node with a semantic value of 0 to an existential node, the examination of the second branch can be dropped, if the splitting variable of that node turns out to be irrelevant to the generation of the contradiction.

For the case of a universal node, the technique works by extracting the model corresponding to leaves π with $\mu(\pi) = [\,]$, whenever such a node is reached. The model corresponding to such a node π is the interpretation ϕ, with

$$\phi(q) = \begin{cases} 1 & \text{if } \nu(\rho_1, \rho_2) = q \text{ for some edge } (\rho_1, \rho_2) \text{ in the path from } \varepsilon \text{ to } \pi; \\ 0 & \text{otherwise.} \end{cases}$$

On backtracking along some edge $x = (\pi_1, \pi_2)$ with $\lfloor \nu(x) \rfloor = r$, a modified model ϕ' is created, with $\phi'(r) = 1 - \phi(r)$, and $\phi'(s) = \phi(s)$, for $s \neq r$. That new model can then be tried on similar subtrees.

2.2.2 Quantifier Inversion

Another technique that is also targeted at the elimination of unnecessary backtracking is *Quantifier Inversion*. Rintanen observes the following theorem in [42].

Theorem 2.2.1 (Quantifier Inversion)
Let A be a formula with $\text{free}(A) = \{q_1, \ldots, q_n, r_1, \ldots, r_m\}$. If, for all models ϕ of A, it also holds that $\phi(r_i) = 0$ (resp. $\phi(r_i) = 1$), for some $1 \leq i \leq m$, then $\psi(r_i) = 0$ (resp. $\psi(r_i) = 1$) for any model ψ of $(\forall\{q_1, \ldots, q_n\})A$.

Proof. Let A be some formula with $\text{free}(A) = \{q_1, \ldots, q_n, r_1, \ldots, r_m\}$. It is easy to show by the semantics of QBFs, that any model ϕ of the formula $(\forall\{q_1, \ldots, q_n\})A$ is also a model of A. However, by assumption, $\phi(r_i) = 0$ (resp. $\phi(r_i) = 1$), for any model ϕ of A. Since ϕ was chosen as an arbitrary model of $(\forall\{q_1, \ldots, q_n\})A$, it follows that $\phi(r_i) = 0$ (resp. $\phi(r_i) = 1$) for any model of that formula. □

The application of this theorem in decision procedures works as follows. Whenever a formula of the form

$$A = (\exists\{r_1, \ldots, r_m\})(\forall\{q_1, \ldots, q_n\})B$$

is encountered, the procedure may try to identify a variable $r = r_i$, with $1 \leq i \leq m$, which must be 0 (resp. 1) under any model of $(\forall\{q_1, \ldots, q_n\})B$. In that case, it would suffice to check the satisfiability of

$$(\exists\{r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_m\})(\forall\{q_1, \ldots, q_n\})A',$$

with $A' = A[q_i/\bot]$ (resp. $A' = A[q_i/\top]$) to determine the satisfiability of A. In practice, the following application of the above lemma can yield such a variable. Reduce the formula $(\forall\{q_1, \ldots, q_n\})B$ with the universal elimination rule along some (possibly randomly chosen) path, until some $r \in \{r_1, \ldots, r_m\}$ becomes unit. In that case, r is a variable with the mentioned property. There is, of course, no guarantee that this will happen, and it might be a good idea to try several different paths, but an exhaustive search will probably be too expensive.

2.2.3 Lemma Caching

Kleine-Büning et al. introduce Q-resolution, a generalization of propositional resolution for quantified propositional logic in [7]. In [37], Letz describes a technique that uses Q-resolution for the generation of lemmas, i.e., additional clauses that can be added to the formula matrices of other leaves without changing the semantics of that leaf (and thus maintaining the semantics of the entire tree). The hope is that such additional clauses limit the size of the subtrees that can be produced during the expansion of such nodes, by yielding a contradictionary clause early.

Such lemmas are generated as follows. For applications of the contradiction rule, one of the initial clauses from which a contradiction was derived, is identified. Upon backtracking from a node with a semantic value of 0 to a universal node, this information is simply passed upwards, but if, upon backtracking from the second successor of an existential node π, the semantic value of π turns out to be 0, Q-resolution is used to resolve the clauses from the subtrees.

2.2.4 Model Caching

The technique of model caching, another technique that is introduced in [37], is the dual form of lemma caching. Here, models are extracted from leaves π with $\mu(\pi) = [\,]$ just as it is done in dependency directed backtracking. *Model resolution*, the dual form of q-resolution, is then used to combine models in universal nodes. These models may be used in other nodes π', where the path from ε to π' contains all the literals of a model. In that case, π' is said to *subsume* that model, and the semantics of π' can be safely assumed to be 1.

2.2.5 Trivial Truth and Trivial Falsity

The technique of *Trivial Truth* [8, 9] employs a sufficient criterion for the satisfiability of a formula to reduce the computations costs in cases where a formula is indeed satisfiable.

Theorem 2.2.2 (Trivial Truth)
Let A be a formula in PCNF. Then

$$\mathrm{qrem}(\mathrm{lrem}(A, \mathrm{lit}_\forall(A)), \lfloor \mathrm{lit}_\forall(A) \rfloor) \vdash A$$

holds.

Proof. We show the theorem for single literals. The general result is obtained by induction on the number $n = |\mathrm{lit}_\forall(A)|$ of literals involved. So let A be some formula in PCNF, and let $L \in \mathrm{lit}_\forall(A)$ be some universally bound literal in A. Recall that $\mathrm{lrem}(A, L)$ semantically corresponds to $\mathrm{pren}(A)(\mathrm{matr}(A)[L/\bot])$, and note that $\bot \vdash C$, for any formula C. Therefore, by the monotonic replacement theorem, it is easy to see that

$$\mathrm{pren}(A)(\mathrm{matr}(A)[L/\bot]) \vdash A.$$

Since $\mathrm{pren}(A)(\mathrm{matr}(A)[L/\bot])$ does not contain an occurrence of L any more, the quantifier $(Q\lfloor L \rfloor)$ is superfluous in $\mathrm{pren}(A)(\mathrm{matr}(A)[L/\bot])$, and therefrom it follows that

$$\mathrm{qrem}(\mathrm{pren}(A)(\mathrm{matr}(A)[L/\bot]), \lfloor L \rfloor) \vdash A,$$

which is equivalent to

$$\mathrm{qrem}(\mathrm{lrem}(A, \mathrm{lit}_\forall(A)), \lfloor \mathrm{lit}_\forall(A) \rfloor) \vdash A.$$

□

When trying to determine the satisfiability of some formula A, an advanced QDLL procedure might resolve to checking the satisfiability of the simpler formula

$$A' = \mathrm{qrem}(\mathrm{lrem}(A, \mathrm{lit}_\forall(A)), \lfloor \mathrm{lit}_\forall(A) \rfloor),$$

and if that formula is indeed satisfiability, it can conclude that A is also satisfiable.

However, the implication works only in one direction, and it is very well possible that A is satisfiable, whereas A' is not. In that case, the procedure might waste a lot of resources on checking A', without gaining any new information about the satisfiability of A. To accommodate this problem, some procedures employ a stateful heuristic that activates Trivial Truth in an adaptive way.

Analogously to the technique of Trivial Truth, the authors of [8] also present two techniques of *Trivial Falsity*, which employ a *necessary* satisfiability criterion to detect the unsatisfiability of a formula. The first of these techniques is already implicitly built into our variant of the basic QDLL procedure through our notion of a tautological clause (c.f. Definition 2.1.2). The second technique, which works by removing all clauses that contain any universally bound literal, has, to our knowledge, not found any widespread adoption.

Feldman et al. present a completely different necessary satisfiability criterion [23], which they also call *Trivial Falsity*. This naming is rather unfortunate, not only because it conflicts with the above techniques from [8], but also because this newer technique is—in our view—far from trivial.

technique	QuBE-BJ	SEMPROP	SSOLVE
backjumping	yes	yes	no
lemma caching	no	yes	no
model caching	no	yes	no
quantifier inversion	no	no	yes
Trivial Truth	yes	special case	yes
Trivial Falsity [8]	no[a]	no	no
Trivial Falsity [23]	no[a]	no	yes

[a] [32] mentions that QuBE employs Trivial Falsity, but we did not find any reference to the technique in later works, like [30] and [31].

Table 2.1: Comparison chart of techniques applied in different decision procedures.

The application of this technique in a QSAT decision procedure is complementary to Trivial Truth. The procedure might resolve to check A', which, being a pure boolean formula is potentially easier than checking A, and if that formula turns out to be unsatisfiable, it can conclude that A is also unsatisfiable.

All of the mentioned "Trivial" techniques—except for the first technique of Trivial Falsity from [8], which is even more trivial—yield a boolean formula that is simpler than the original QBF. That boolean formula needs to be checked for satisfiability, in order to enable possible conclusions w.r.t. the satisfiability of the original formula. The checking can be done by a separate SAT decision procedure, or recursively, by interpreting the problem as a QSAT problem.

Beedless to say all of the "Trivial" techniques do merely increase the total computational overhead in the case where the respective criterion does not allow any conclusion to the satisfiability of the original formula. However, experimental results show the practical value of these techniques (see, e.g., [30, 23]).

2.2.6 Availability of Techniques

For this work, we have chosen three advanced QDLL implementations for an empirical study of the behavior of different prenexing strategies on the observable performance of these decision procedures: QuBE-BJ [31, 32], SEMPROP [37] and SSOLVE [23, 43]. Since all three of these procedures are based on the same basic procedure, we have reason to assume that their difference in behavior is essentially founded in the different subsets of advanced techniques that each implementation employs.

Table 2.1 gives an overview of which techniques have been integrated into these procedures. Note that we chose the classical QuBEinstead of the more advanced of QuBE-Rel, because the latter happened to occasionally terminate with an internal error message, in our setup.

Chapter 3

Prenexing Strategies

In the previous chapter, we have discussed QDLL as a procedure for deciding the satisfiability of QBFs. The necessary PCNF transformation can be performed by the structure preserving PCNF transformation that was presented in Section 1.4.2. However, until now we have omitted some important details that determine the exact behavior of the transformation. In particular, the procedure presented makes use of the standard PNF Transformation, which was defined in terms of a non-confluent rewrite system [3].

In this chapter, we investigate different prenexing strategies, i.e., prescriptions that specify the behavior of a prenexing algorithm. We first present a theoretical framework for the description of prenexing strategies in Section 3.1. In Section 3.2, we consider quantifiers paths as a representation of quantifier precedence. We continue with the presentation of 14 different prenexing strategies in Section 3.3. The different behavior of these strategies is demonstrated in an example in Section 3.3.4.

3.1 A Theoretical Framework

Since we are interested in deciding satisfiability, it suffices to consider transformations which preserve satisfiability. The problem of computing, for a given formula A, a satisfiability equivalent PCNF can be naturally separated into two steps. Given a formula A,

1. transform A into a PNF formula A', and then
2. transform the matrix of A' into PCNF, using a structure preserving normal form translation.

We have already seen this approach in Algorithm 1.4.6.

The latter step will typically employ a structure preserving normal form transformation, like the one seen in Algorithm 1.4.6, which produces a PCNF formula with a prenex consisting of existential quantifiers binding the introduced labels. The combined effect of both phases does, however, still produce a proper PCNF

formula, by the implicit appending of the prenex from the second step to that of the first step.

Considering Morale 1.3.1, this procedure might not always be optimal, since it may introduce one unnecessary quantifier alternation, whenever the prefix produced by the first step ends in an universal quantifier. Fortunately, this case can easily be identified by analyzing *critical paths*, which are presented in Section 3.2, and we suggest the use of the following procedure for dealing with such situations. Given a formula A,

1. transform $\neg A$ into a PCNF formula A' as above,
2. decide the satisfiability of A', and
3. invert the result obtained from the satisfiability test.[1]

We want to investigate into the first step, and along the way present a theoretical framework for the description of such transformations. For this framework, it is practical to assume formulas in NNF to avoid certain technical difficulties concerning the polarity of sub-formulas, which would obscure the general ideas by awkward sidesteps. The resulting limitation is marginal, since the usual NNF transformations, like, e.g., Algorithm 1.4.2, do preserve most of the structural properties of a formula. We therefore introduce a new convention.

Convention 3.1.1 (Assumption of NNF Formulas)
In this section, when talking about formulas, we always assume that these formulas are in NNF (unless stated otherwise).

The prenexing step can essentially be captured by a function that maps each formula A to a suitable permutation of the quantifiers that occur in A. We first define the set of quantifiers in a formula. For formulas in NNF, this is straightforward:

Definition 3.1.1 (Quantifiers in a Formula)
For any formula A in NNF, the *set* $\mathrm{quant}(A)$ *of quantifiers in* A is defined by

$$\mathrm{quant}(A) = \{Qq \mid A|_\pi = (Qq)B, \text{ for some position } \pi\}.$$

On the other hand, the propositional skeleton of a formula is what remains of the formula after removing all quantifiers:

Definition 3.1.2 (Propositional Skeleton)
Let A be a formula. The *propositional skeleton* $\mathrm{sk}(A)$ *of* A is recursively defined as follows:

$$\mathrm{sk}(A) = \begin{cases} A & \text{if } A = \mathcal{V} \cup \{\top, \bot\}; \\ \neg\mathrm{sk}(B) & \text{if } A = \neg B; \\ \mathrm{sk}(B) & \text{if } A = (Qq)B; \\ \mathrm{sk}(B) \circ \mathrm{sk}(C) & \text{if } A = B \circ C. \end{cases}$$

[1] Remember that QSAT is defined on closed formulas. So this last step is justified by Theorem 1.3.1.

Example 3.1.1 (Quantifiers in a Formula, Propositional Skeleton)
Consider the formula

$$A = \neg(\exists q)(q \wedge ((\forall r)(\exists s)((\neg r) \vee s))).$$

From A, we can extract the set of quantifiers $\{(\exists q), (\forall r), (\exists s)\}$ and the propositional skeleton $\neg(q \wedge ((\neg r) \vee s))$.

Next, we introduce the notion of *prenexing functions*, which are functions that map formulas to quantifier prenexes (recall that $\tilde{\mathcal{F}}$ denotes the set of closed, cleansed formulas in NNF).

Definition 3.1.3 (Prenexing Function)
A function $f : \tilde{\mathcal{F}} \to \mathcal{Q}^\star$ is called a *prenexing function*, if, for every $A \in \tilde{\mathcal{F}}$, it holds that $f(A) = (Q_1 q_1) \ldots (Q_n q_n)$ where $\text{quant}(A) = \{(Q_i q_i) \mid 1 \leq i \leq n\}$ and $(Q_i q_i) \neq (Q_j q_j)$, for all $1 \leq i < j \leq n$.

Every prenexing function has an associated transformation function, which maps formulas to formulas.

Definition 3.1.4 (Transformation Function)
Let f be a prenexing function. Then the *transformation function* $\hat{f}$ *for* f is defined by $\hat{f}(A) = f(A)\text{sk}(A)$.

Transformation functions are thus suitable for the description of the important class of PNF transformations that

1. preserve the set of quantifiers, and
2. do not change the propositional skeleton of a formula.

Such transformations can be seen as an abstraction of a series of applications of the binary *quantifier shifting* rules. Our definition does, however, not encompass transformations that rely on the *quantifier fusion* rule, which is somewhat similar to *binary quantifier shifting*, but has the beneficial effect of eliminating some quantifiers. The transformations that we describe by transformation functions never drop any quantifiers.

A prenexing function is correct, iff its corresponding transformation function preserves the semantics of formulas.

Definition 3.1.5 (Correct Prenexing Function)
A prenexing function f is called *correct w.r.t. logical equivalence*, or simply *correct*, if $\hat{f}(A) \equiv A$ for all $A \in \tilde{\mathcal{F}}$. Likewise, we say that f is *correct w.r.t. satisfiability equivalence*, if $\hat{f}(A) \overset{\text{sat}}{\equiv} A$ for all $A \in \tilde{\mathcal{F}}$.

As we mentioned before, the weaker notion of satisfiability equivalence will suffice for our purpose, since we are merely concerned about satisfiability. When we simply speak of equivalence in this context, we mean *satisfiability equivalence*.

It is not hard to see that every correct transformation function represents a PNF transformation that is correct in the sense that the original formula and the resulting formula are equivalent. However, the definition *per se* does not have

much practical value, since it makes direct use of the semantic notion of equivalence. What we would like to get are syntactic transformations. Accordingly, we would like a simple syntactic criterion that describes correct transformation functions (or their underlying prenexing functions).

If we can provide an easily decidable order relation $\mathcal{R}$, then we can also easily check the following positional criterion.

Definition 3.1.6 (Prenexing Function Obeying Relation)
Let A be a formula, f a prenexing function and $\mathcal{R} \subseteq (\mathcal{Q} \times \mathcal{Q})$ a relation on quantifiers. We then say that f *obeys* $\mathcal{R}$ for A, if $f(A) = (Q_1 q_1) \dots (Q_n q_n)$ with $(Q_i q_i)\mathcal{R}(Q_j q_j)$ implies $i < j$, for all $0 \leq i \leq n$ and $0 \leq j \leq n$.

That is, wherever two quantifiers can be ordered w.r.t. $\mathcal{R}$, then the prenexing function must honor that relation for the image of A by arranging these quantifiers in a corresponding left-to-right sequence. This way, we can view $\mathcal{R}$ as a constituent of a partial specification of a prenexing function. The order determines the positional dependencies that should be respected by the prenexing function, if such a function exists.

We said that a relation on quantifiers is a *constituent* of a partial specification. An entire partial specification must provide a relation on quantifiers for each formula. Technically, we define a specification as a function, but intuitively it can be thought of as a collection of relations on quantifiers, one for each formula.

Definition 3.1.7 (Specification)
A *(partial) specification* is a total function $\theta : \tilde{\mathcal{F}} \to (\mathcal{Q} \times \mathcal{Q})$ that maps every formula to a relation on quantifiers.

Definition 3.1.8 (Prenexing Function Obeying Specification)
Let f be a prenexing function and θ a specification. Then f is said to *obey* θ, if f obeys $\theta(A)$, for any formula A.

We are, of course, mostly interested in specifications that induce correct prenexing functions. We call these *correct specifications*.

Definition 3.1.9 (Correct Specification)
Let θ be a specification. If every prenexing function f that obeys θ is correct, then θ is said to be *correct*.

If we already have some specification θ, we can easily construct a more special specification by extending relations.

Definition 3.1.10 (Specialization)
Let θ be a specification. A specification θ' with $\theta(A) \subseteq \theta'(A)$ for every $A \in \tilde{\mathcal{F}}$ is called a *specialization* of θ.

Theorem 3.1.1 (Inheritance of Obedience)
Let θ be some specification, let θ' be a specialization of θ, and let f be a prenexing function that obeys θ'. Then f obeys θ.

Proof. We assume that the prenexing function f obeys θ', i.e., for any formula A, it holds that $f(A) = (Q_1 q_1) \dots (Q_n q_n)$, such that $(Q_i q_i)\theta'(A)(Q_j q_j)$ implies $i < j$, for all $0 \leq i \leq n$ and $0 \leq j \leq n$. But θ' is a specialization of θ, therefore $(Q_i q_i)\theta(A)(Q_j q_j)$ implies $(Q_i q_i)\theta'(A)(Q_j q_j)$. It is then easy to see that $(Q_i q_i)\theta(A)(Q_j q_j)$ implies $i < j$, for all $0 \leq i \leq n$ and $0 \leq j \leq n$, i.e., f obeys θ. □

Our interest in the notion of specialization is founded in the following central theorem.

Theorem 3.1.2 (Specialization of Correct Specifications)
Any specialization of a correct specification is correct.

Proof. Let θ be a correct specification and θ' a specialization of θ. We show the theorem by contradiction, and thus assume that θ' is incorrect, i.e., there is some prenexing function f such that f obeys θ', but f is incorrect. However, by Theorem 3.1.1, f must also obey θ. But θ is a correct specification, so f must be correct, which contradicts the above argument that f is incorrect. □

Our idea should now be fairly evident: If we can find some minimal correct specification, then we can very easily construct correct prenexing functions, and consequently new correct PNF transformations through specialization. A prescription on how to perform this specialization is called *prenexing strategy*.

The only problem in this idea lies in finding such a minimal specification. Let us approach this problem by contemplating possibly harmful behavior in a transformation.

The quantifier commutability rule of Theorem 1.2.4 shows that adjacent quantifiers containing the same quantifier symbol may be swapped, without affecting the semantics. For quantifiers containing complementary quantifier symbols, this is not generally true.

Example 3.1.2 (Swapping of Adjacent Quantifiers)

1. The equivalence $(\forall q)(\exists r)(q \vee r) \equiv (\exists r)(\forall q)(q \vee r)$ does hold, but
2. the equivalence $(\forall q)(\exists r)(q \leftrightarrow r) \equiv (\exists r)(\forall q)(q \leftrightarrow r)$ does not hold.

We may thus consider a specification that disallows swapping that would affect the semantics of a formula.

Of course, in a sequence of "illegal" swapping operations, the adverse effects are not unlikely to cancel each other and such a transformation might, as a whole, eventually produce formulas that are equivalent to the original ones. However, such interactions are presumably hard to capture in a syntactic way.

If we leave aside such intricate interactions, there still remains the question of how to distinguish illegal swapping operations from harmless ones in a syntactic way. This is still an intrinsically hard problem, for which we cannot offer a completely satisfactory solution. Instead, we suggest the usage of a weaker condition that sorts out all *potentially* dangerous swapping operations. In our terminology, the condition is given through the following relation on quantifiers.

Definition 3.1.11 (Quantifier Precedence Relation)

1. Let A be a formula. The *Quantifier Precedence Relation (QPR)* $\blacktriangleleft_A$ *of* A is defined as the transitive closure of the following relation $\mathcal{R}_A$, defined by

$$Qq \; \mathcal{R}_A \; Rr \iff \mathrm{qpos}(A, q) \prec \mathrm{qpos}(A, r) \text{ and } Q \neq R,$$

   for all $\{Qq, Rr\} \subseteq \mathrm{quant}(A)$.

2. The *quantifier precedence specification* $\blacktriangleleft$ is defined by

$$\blacktriangleleft (A) = \blacktriangleleft_A, \text{ for any } A \in \tilde{\mathcal{F}}.$$

Example 3.1.3 (Quantifier Precedence)
Consider the formula

$$(\exists q)(\forall r)((\exists s)(q \vee (r \vee s)) \wedge (\forall t)(q \vee (r \vee t))).$$

The quantifier precedence relation of this formula is

$$\{(\exists q, \forall r), (\exists q, \exists s), (\exists q, \forall t), (\forall r, \exists s)\}.$$

Theorem 3.1.3 (Correctness of Quantifier Precedence Specification)
The quantifier precedence specification is correct.

Proof. We want to show that the quantifier precedence specification is correct, i.e., that every prenexing function f that obeys $\blacktriangleleft_A$ for any formula A is correct. So let A be some formula, and let f be a prenexing function that obeys $\blacktriangleleft_A$. To prove the correctness of f, we have to show that $f(A)\mathrm{sk}(A) \equiv A$.

We do this by showing how the formula

$$f(A)\mathrm{sk}(A) = (Q_1 q_1 \ldots Q_n q_n)\mathrm{sk}(A)$$

can be constructed from A, through a series of applications of *quantifier shifting* rules from Theorem 1.2.4. In order to construct the prefix $(Q_1 q_1 \ldots Q_n q_n)$, we move each quantifier into its place, one after the other, starting with $(Q_1 q_1)$, then $(Q_2 q_2)$, and so on.

If two quantifiers $(Q_i q_i)$ and $(Q_j q_j)$ with $i \neq j$ occur A at parallel positions, then they can be moved independently of each other. The only situation that would require special attention is the case, where some quantifier $(Q_i q_i)$ must be moved into its proper place, but is blocked by some other quantifier $(Q_j q_j)$. However, f obeys $\blacktriangleleft_A$, therefore $\mathrm{qpos}(A, q_j) \prec \mathrm{qpos}(A, q_i)$ implies $j < i$, i.e., the situation can never occur, because blocking quantifiers are always moved before the quantifiers they block are moved.

It is clear that the construction of the prefix $f(A)$ just described leaves behind the skeleton of A, so we have actually constructed $f(A)\mathrm{sk}(A)$ from A in an equivalence preserving way. □

3.2 Quantifier Paths and Alternations

As we have seen in Theorem 1.3.4, the theoretical complexity of the problem of deciding the satisfiability of a formula A in PNF depends on the number of alternations of quantifier symbols in the prenex of A. It therefore makes sense to require that, for a prenex function f, the image $f(A)$ of A should contain a minimal number of alternations of quantifier symbols, for any formula A.

Definition 3.2.1 (Paths, Alternations, Critical Paths)

1. Let A be a formula. The set $\mathrm{pth}(A) \subseteq \mathcal{Q}^\star$ of *quantifier paths* of A is defined by

$$\mathrm{pth}(A) = \begin{cases} \mathrm{pth}(B_1) \cup \mathrm{pth}(B_2) & \text{if } A = B_1 \circ B_2; \\ \{Qq\gamma \mid \gamma \in \mathrm{pth}(B)\} & \text{if } A = (Qq)B; \\ \{\varepsilon\} & \text{otherwise.} \end{cases}$$

   Recall that we consider formulas in NNF only.

2. Let γ be a quantifier path. The number $\mathrm{alt}(\gamma)$ of *quantifier alternations* in γ is recursively defined by

$$\mathrm{alt}(\gamma) = \begin{cases} 0 & \text{if } \gamma \in \mathcal{Q} \cup \varepsilon; \\ \mathrm{alt}(Qr\delta) & \text{if } \gamma = (Qq)(Qr)\delta; \\ \mathrm{alt}(\overline{Q}r\delta) + 1 & \text{if } \gamma = (Qq)(\overline{Q}r)\delta. \end{cases}$$

3. Let $\mathcal{G}$ be a set of quantifier paths. The *maximal number of quantifier alternations* $\mathrm{malt}(\mathcal{G})$ in $\mathcal{G}$ is defined by

$$\mathrm{malt}(\mathcal{G}) = \max_{\gamma \in \mathcal{G}} \mathrm{alt}(\gamma).$$

4. Let A be a formula. The *maximal number of quantifier alternations* $\mathrm{malt}(A)$ in A is defined by

$$\mathrm{malt}(A) = \mathrm{malt}(\mathrm{pth}(A)).$$

5. Let $\mathcal{G}$ be a set of quantifier paths. The set of *critical paths* $\mathrm{crit}(\mathcal{G})$ in $\mathcal{G}$ are defined by

$$\mathrm{crit}(\mathcal{G}) = \{\gamma \mid \mathrm{alt}(\gamma) = \mathrm{malt}(\mathcal{G}), \gamma \in \mathcal{G}\}.$$

6. Let A be a formula. The set of *critical paths* $\mathrm{crit}(A)$ of A is defined by

$$\mathrm{crit}(A) = \mathrm{crit}(\mathrm{pth}(A)).$$

It is easy to see that, for formulas in PNF, there is exactly one quantifier path, and that the value of $\mathrm{malt}(\cdot)$ of such formulas is identical to the number of quantifier symbol alternations.

Notation 3.2.1 (Shorthand for Quantifier Paths)
As a shorthand for a quantifier path $Qq_1 \ldots Qq_n$, we simply write $Q(q_1 \ldots q_n)$.

Example 3.2.1 (Shorthand for Quantifier Paths)
The quantifier path $\forall q_1 \forall q_2 \exists q_3 \exists q_4 \exists q_5 \forall q_6$ may be written in a more concise form as $\forall(q_1 q_2)\exists(q_3 q_4 q_5)\forall(q_6)$.

Theorem 3.2.1 (Bound for Quantifier Alternations)
Let A be some formula with $\mathrm{malt}(A) = n$.

1. If all paths $\gamma \in \mathrm{crit}(A)$ start with the same quantifier symbol Q, then we can obtain from A a formula A' in PNF such that $\mathrm{malt}(A') = n$ and the prenex of A' starts with the quantifier symbol Q.
2. Otherwise, i.e, if $\mathrm{crit}(A)$ contains two or more paths, some of which start with the quantifier symbol Q and some of which start with the quantifier symbol $\overline{Q}$, then we can obtain from A a formula A' in PNF such that $\mathrm{malt}(A') = n + 1$ and the prenex of A' starts with the quantifier symbol Q.[2]

Proof. We give a proof sketch. Let A be some formula with $\mathrm{malt}(A) = n$. We show the theorem by induction on the structure of A.

Case $A \in \mathcal{V} \cup \{\top, \bot\} \cup \{v, \neg v \mid v \in \mathcal{V}\}$**:** Then A is in PNF, i.e., $A' = A$, and the theorem trivially holds.

Case $A = (Qq)A_1$**:** We distinguish two sub-cases:

1. $\mathrm{crit}(A_1)$ contains only paths that start with the same quantifier symbol R.
   By the induction hypothesis, we may replace A_1 with a formula A_1' in PNF such that $\mathrm{malt}(A_1') = \mathrm{malt}(A_1)$ and the prenex of A_1' starts with the quantifier symbol R. Clearly, the prenex of A' starts with the quantifier symbol Q.
   Concerning the length of the prenex, we make the following observation. If $Q = R$, then
   $$\mathrm{malt}(A') = \mathrm{malt}(A_1') = \mathrm{malt}(A_1) = \mathrm{malt}(A) = n.$$
   On the other hand, if $Q \neq R$, then
   $$\mathrm{malt}(A') = \mathrm{malt}(A_1') + 1 = \mathrm{malt}(A_1) + 1 = \mathrm{malt}(A) = n.$$
2. $\mathrm{crit}(A_1)$ contains either paths that start with Q and also paths that start with the quantifier symbol $\overline{Q}$.
   By the induction hypothesis, we can replace A_1 with a formula A_1' in PNF such that $\mathrm{malt}(A_1') = \mathrm{malt}(A_1) + 1$ and the prenex of A_1' starts with the quantifier symbol Q. Clearly, the prenex of A' starts with the quantifier symbol Q.
   Concerning the length of the prenex, we make the following observation.
   $$\mathrm{malt}(A') = \mathrm{malt}(A_1') = \mathrm{malt}(A_1) + 1 = \mathrm{malt}(A) + 1 = n + 1.$$

[2] Note that, for this part of the definition, we can likewise obtain a prenex that starts with $\overline{Q}$, because of the symmetry of Q and $\overline{Q}$.

Case $A = A_1 \circ A_2$**:** We distinguish three sub-cases:

1. $\mathrm{crit}(A_1) \cup \mathrm{crit}(A_2)$ contains only paths that start with the same quantifier symbol Q.

   By the induction hypothesis, we may replace A_1 with a formula A_1' in PNF, such that $\mathrm{malt}(A_1') = \mathrm{malt}(A_1)$ and the prenex of A_1' starts with the quantifier symbol Q. Likewise, we may replace A_2 with a formula A_2', with $\mathrm{malt}(A_2') = \mathrm{malt}(A_2)$ and with a prenex that starts with the quantifier symbol Q.

   In the prenexing steps, we first try to move out as many quantifiers containing the quantifier symbol Q as possible by applying the *binary quantifier shifting* rules. Next, we try to move out as many quantifiers containing the quantifier symbol $\overline{Q}$ as possible, then we start over again with the quantifiers that contain the quantifier symbol Q, etc.

   It is not hard to see that this procedure eventually yields a formula A' that is in PNF, with

   $$\mathrm{malt}(A') = \max(\mathrm{malt}(A_1'), \mathrm{malt}(A_2'))(= n)$$

   and with a prenex that starts with the quantifier symbol Q.

2. $\mathrm{crit}(A_1)$ and $\mathrm{crit}(A_2)$ each contain either paths that start with the quantifier symbol Q and also paths that start with $\overline{Q}$.

   This sub-case is similar to the first one. We first replace the two immediate sub-formulas A_1 and A_2 with A_1' and A_2', respectively, with the difference that now $\mathrm{malt}(A_1') = \mathrm{malt}(A_1) + 1$ and $\mathrm{malt}(A_2') = \mathrm{malt}(A_2) + 1$. The prenex of either sub-formula starts with the quantifier symbol Q, and we can merge these in exactly the same way as in the first case, only now

   $$\mathrm{malt}(A') = \max(\mathrm{malt}(A_1'), \mathrm{malt}(A_2'))(= n + 1).$$

3. $\mathrm{crit}(A_1)$ contains either paths that start with Q and also paths that start with the quantifier symbol $\overline{Q}$, but $\mathrm{crit}(A_2)$ contains only paths that start with the quantifier symbol R.

   By the induction hypothesis, we can replace A_1 with a formula A_1' in PNF such that $\mathrm{malt}(A_1') = \mathrm{malt}(A_1) + 1$ and the prenex of A_1' starts with the quantifier symbol Q. Likewise, we can replace A_2 with a formula A_2' in PNF such that $\mathrm{malt}(A_2') = \mathrm{malt}(A_2)$ and the prenex of A_2' starts with the quantifier symbol R.

   We proceed to merging the prenexes just like in the first case, starting with quantifiers that contain the quantifier symbol Q and finally obtain a formula A' that is in PNF, with

   $$\mathrm{malt}(A') = \max(\mathrm{malt}(A_1'), \mathrm{malt}(A_2'))(= n + 1)$$

   and a prenex that starts with the quantifier symbol Q.

□

Any set of quantifier paths implicitly describes a relation on quantifiers.

Definition 3.2.2 (Associated Relation of a Set of Paths)
Let $\mathcal{G}$ be a set of quantifier paths. Then the *associated relation on quantifiers* $\mathrm{aqr}(\mathcal{G})$ of $\mathcal{G}$ is defined as the transitive closure of the following relation $\mathcal{R}_A$, defined by

$$Qq\ \mathrm{aqr}(\mathcal{G})\ Rr \iff \gamma_1 Q q \gamma_2 R r \gamma_3 \in \mathcal{G} \text{ and } Q \neq R,$$

for some quantifiers paths γ_1, γ_2, and γ_3.

Note the similarity of this definition to Defintion 3.1.11.

The connection between quantifier paths and quantifier relations is established by the following theorem.

Theorem 3.2.2 (Associated Relation of Paths of a Formula)
For any formula A in NNF, it holds that $\mathrm{aqr}(\mathrm{pth}(A)) = \blacktriangleleft_A$.

Theorem 3.2.3 (Path Augmentation)
Let $\mathcal{G}$, $\mathcal{G}_1$, and $\mathcal{G}_2$ be sets of paths, with

$$\begin{array}{l}\mathcal{G}_1 = \mathcal{G} \cup \{\gamma\} \text{ with } \gamma = \gamma_1\gamma_2\ldots\gamma_n, \text{and}\\ \mathcal{G}_2 = \mathcal{G} \cup \{\gamma'\} \text{ with } \gamma' = \gamma_1 Q_1 q_1 \gamma_2 Q_2 q_2 \ldots \gamma_{n-1} Q_{n-1} q_{n-1} \gamma_n.\end{array}$$

Then it holds that $\mathrm{aqr}(\mathcal{G}_1) \subseteq \mathrm{aqr}(\mathcal{G}_2)$.

Notation 3.2.2 (Extension of Relations on Paths)
Let $\mathcal{R} \subseteq (\mathcal{Q} \times \mathcal{Q})$ be some relation on quantifiers. Furthermore, let

$$\begin{array}{l}\gamma = Q_1(q_{1,1}\ldots q_{1,k_1})\ldots Q_n(q_{n,1}\ldots q_{n,k_n}), \text{ and}\\ \delta = R_1(r_{1,1}\ldots r_{1,l_1})\ldots R_m(r_{m,1}\ldots r_{m,l_m})\end{array}$$

be quantifier paths, with $Q_{i+1} = \overline{Q_i}$, for $1 \leq i < n$, and $R_{i+1} = \overline{R_i}$, for $1 \leq i < m$. Then $\gamma\,\mathcal{R}\,\delta$ is a shorthand for the following proposition:

$$Q_i q_{i,j}\,\mathcal{R}\,R_i r_{i,k}, \text{for any } 1 \leq i \leq \min(n,m), 1 \leq j \leq n_i \text{ and } 1 \leq k \leq m_i.$$

3.3 Prenexing Strategies

3.3.1 Simple Symbol Based Strategies

This class of strategies is best described operationally, as an algorithm on quantifier paths.

Algorithm 3.3.1 (Simple Symbol Based Path Merging)
Signature: $\{\mathsf{adeu}, \mathsf{aued}, \mathsf{d}, \mathsf{edau}, \mathsf{euad}, \mathsf{u}\} \times \tilde{\mathcal{F}} \rightarrow \mathcal{Q}^\star$

The algorithm takes an input formula A and computes a quantifier path, by performing the following steps:

1. If $\mathrm{pth}(A) = \emptyset$, then return the empty path ε as result.

2. Choose any path $\gamma_0 \in \mathrm{crit}(A)$. If possible, choose one that ends in an existential quantifier symbol[3]. Furthermore, let $\mathcal{G}_0 = \mathrm{pth}(A) \setminus \{\gamma_0\}$.

3. Iterate the following steps, successively producing paths $\gamma_0, \gamma_1, \ldots$ and sets $\mathcal{G}_0, \mathcal{G}_1, \ldots$, until $\mathcal{G}_i$ is empty. For the i-th iteration, perform the following steps.

   (a) Choose any path $\delta_i \in \mathrm{crit}(\mathcal{G}_i)$, and let $\mathcal{G}_{i+1} = \mathcal{G}_i \setminus \{\delta_i\}$.

   (b) Generate a new path γ_{i+1} by merging the paths

   $$\gamma_i = \zeta_i Q_1 \alpha_1 \ldots Q_n \alpha_n \text{ and}$$
   $$\delta_i = \zeta_i R_1 \beta_1 \ldots R_m \beta_m,$$

   where

   i. ζ_i is the longest common prefix of γ_i and δ_i,

   ii. α_j is a sequence

   $$\alpha_j = (q_{j,1}, q_{j,2}, \ldots, q_{j,k_j})$$

   of variables, for $1 \leq j \leq n$,

   iii. β_j is a sequence

   $$\beta_j = (r_{j,1}, r_{j,2}, \ldots, r_{j,l_j})$$

   of variables, for $1 \leq j \leq m$,

   iv. $Q_{j+1} = \overline{Q_j}$, for $1 \leq j < n$, and

   v. $R_{j+1} = \overline{R_j}$, for $1 \leq j < m$.

   (c) If $n = m$ and $Q_1 \neq R_1$, then generate the new path

   $$\gamma_{i+1} = R_1 \beta_1 Q_1 \alpha_1 \ldots R_n \beta_n Q_n \alpha_n.$$

   Otherwise, generate the new path in the following way. First determine an index d, depending on the prenexing strategy.

   Strategy adeu: If $R_1 = \exists$, then let $d = 1$, else let $d = 0$.

   Strategy aued: If $R_m = \exists$, then let $d = m - 1$, else let $d = m$.

   Strategy edau: If $R_1 = \forall$, then let $d = 1$, else let $d = 0$.

   Strategy euad: If $R_m = \forall$, then let $d = m - 1$, else let $d = m$.

   Strategy d: Let $d = 0$.

   Strategy u: Let $d = m$.

   Let $c = m - n + 1$. Given the index d, create a new path

   $$\begin{aligned}\gamma_{i+1} = \zeta_i & \\ & Q_1 \alpha_1 R_1 \beta_1 \ldots Q_d \alpha_d R_d \beta_d \\ & Q_{d+1} \alpha_{d+1} \ldots Q_{d+c-1} \alpha_{d+c-1} \\ & R_{d+c} \beta_{d+c} Q_{d+c} \alpha_{d+c} \ldots R_m \beta_m Q_n \alpha_n.\end{aligned}$$

[3] If γ_0 ends in an existential quantifier symbol, then the prenex (without labels) will end in an existential quantifier symbol too. The existential quantifier bindings obtained from the labeling process can then be appended to this path, without introducing any additional quantifier alternation (c.f. the beginning of Chapter 3.1).

4. Return the path γ_w as result, where w is the number of iterations that were performed.

It is clear that the algorithm terminates, as $\mathcal{G}_0$ is finite and each iteration removes one element from $\mathcal{G}_i$ to produce $\mathcal{G}_{i+1}$, until the empty set is reached.

Example 3.3.1 (Simple Symbol Based Path Merging)
Assume that formula A yields the following set of paths.

$$\begin{aligned}\mathrm{pth}(A) = \{&\exists q_1 \forall q_2 \forall r_1 \exists r_2 \forall r_3, \\ &\exists q_1 \forall q_2 \exists s_1 \forall s_2 \exists s_3 \forall s_4 \exists s_5 \forall s_6\}.\end{aligned}$$

The set $\mathrm{crit}(A)$ contains a single path, so the algorithm chooses

$$\begin{aligned}\gamma_0 &= \exists q_1 \forall q_2 \exists s_1 \forall s_2 \exists s_3 \forall s_4 \exists s_5 \forall s_6, \text{ and} \\ \delta_0 &= \exists q_1 \forall q_2 \forall r_1 \exists r_2 \forall r_3.\end{aligned}$$

The algorithm then merges these two paths into a new path γ_1, depending on the selected strategy.

Strategy adeu: $\gamma_1 = \exists q_1 \forall q_2 \exists s_1 \forall s_2 \exists s_3 \underline{\forall r_1} \forall s_4 \underline{\exists r_2} \exists s_5 \underline{\forall r_3} \forall s_6$;

Strategy aued: $\gamma_1 = \exists q_1 \forall q_2 \exists s_1 \underline{\forall r_1} \forall s_2 \underline{\exists r_2} \exists s_3 \underline{\forall r_3} \forall s_4 \exists s_5 \forall s_6$;

Strategy edau: $\gamma_1 = \exists q_1 \forall q_2 \exists s_1 \underline{\forall r_1} \forall s_2 \exists s_3 \forall s_4 \underline{\exists r_2} \exists s_5 \underline{\forall r_3} \forall s_6$;

Strategy euad: $\gamma_1 = \exists q_1 \forall q_2 \exists s_1 \underline{\forall r_1} \forall s_2 \underline{\exists r_2} \exists s_3 \forall s_4 \exists s_5 \underline{\forall r_3} \forall s_6$;

Strategy d: $\gamma_1 = \exists q_1 \forall q_2 \exists s_1 \forall s_2 \exists s_3 \underline{\forall r_1} \forall s_4 \underline{\exists r_2} \exists s_5 \underline{\forall r_3} \forall s_6$;

Strategy u: $\gamma_1 = \exists q_1 \forall q_2 \exists s_1 \underline{\forall r_1} \forall s_2 \underline{\exists r_2} \exists s_3 \underline{\forall r_3} \forall s_4 \exists s_5 \forall s_6$.

Since $\mathcal{G}_1$ is empty, the algorithm terminates and returns γ_1.

Definition 3.3.1 (Simple Symbol Based Strategies)
The *simple symbol based* strategies θ_x, for $x \in \{\mathsf{adeu}, \mathsf{aued}, \mathsf{edau}, \mathsf{euad}, \mathsf{d}, \mathsf{u}\}$, are given by the following definition.

$\theta_x(A) = \mathrm{aqr}(\{\gamma_{x,A}\})$, where $\gamma_{x,A}$ is the output of Algorithm 3.3.1 for the input (x, A).

Theorem 3.3.1 (Correctness of Simple Symbol Based Strategies)
The *simple symbol based* strategies θ_x, for $x \in \{\mathsf{adeu}, \mathsf{aued}, \mathsf{edau}, \mathsf{euad}, \mathsf{d}, \mathsf{u}\}$, are correct.

Proof. We give a proof sketch. For formula A with $\mathrm{pth}(A) = \emptyset$, the algorithm returns the empty path ε, and it is trivial that

$$\blacktriangleleft_A = \emptyset \subseteq \mathrm{aqr}(\varepsilon) = \theta_x(A).$$

We note this result, and turn to the more interesting case where $\mathrm{pth}(A) \neq \emptyset$. So we assume any formula A with $k \geq 0$, where $k = |\mathrm{pth}(A)| - 1$. We can see that the algorithm works by merging all paths into a single path. During the i-th iteration of the loop, the paths γ_i and δ_i are merged into a new path γ_{i+1}

in such a way that γ_{i+1} augments both, γ_i and δ_i. Therefore, it follows from Theorem 3.2.3 that

$$\begin{array}{l} \mathrm{aqr}(\mathcal{H} \cup \{\gamma_i\}) \subseteq \mathrm{aqr}(\mathcal{H} \cup \{\gamma_{i+1}\}), \text{ and} \\ \mathrm{aqr}(\mathcal{H}' \cup \{\delta_i\}) \subseteq \mathrm{aqr}(\mathcal{H}' \cup \{\gamma_{i+1}\}), \end{array}$$

for any sets of paths $\mathcal{H}$ and $\mathcal{H}'$. If we now let $\mathcal{H} = \mathcal{G}_i \cup \{\delta_i\}$, and $\mathcal{H}' = \mathcal{G}_i \cup \{\gamma_{i+1}\}$, we obtain

$$\begin{array}{l} \quad \mathrm{aqr}((\mathcal{G}_i \cup \{\delta_i\}) \cup \{\gamma_i\}) \subseteq \mathrm{aqr}((\mathcal{G}_i \cup \{\delta_i\}) \cup \{\gamma_{i+1}\}) = \\ = \mathrm{aqr}((\mathcal{G}_i \cup \{\gamma_{i+1}\}) \cup \{\delta_i\}) \subseteq \mathrm{aqr}((\mathcal{G}_i \cup \{\gamma_{i+1}\}) \cup \{\gamma_{i+1}\}) = \\ = \mathrm{aqr}(\mathcal{G}_i \cup \{\gamma_{i+1}\}). \end{array}$$

By the definition of the algorithm $\mathrm{pth}(A) = \mathcal{G}_0 \cup \{\gamma_0\}$, and $\mathcal{G}_i = \mathcal{G}_{i+1} \cup \{\delta_{i+1}\}$, for $i < k$. It is also clear that $\mathcal{G}_k = \emptyset$. By a simple inductive argument, we can thus obtain

$$\begin{array}{rl} \mathrm{aqr}(\mathrm{pth}(A)) & = \mathrm{aqr}((\mathcal{G}_0 \cup \{\delta_0\}) \cup \gamma_0) \subseteq \ldots \subseteq \mathrm{aqr}(\mathcal{G}_{i-1} \cup \{\gamma_i\}) = \\ & = \mathrm{aqr}((\mathcal{G}_i \cup \{\delta_i\}) \cup \{\gamma_i\}) \subseteq \ldots \subseteq \mathrm{aqr}(\mathcal{G}_k \cup \{\gamma_{k+1}\}) = \\ & = \mathrm{aqr}(\{\gamma_{k+1}\}). \end{array}$$

$$\begin{array}{rl} \mathrm{aqr}(\mathrm{pth}(A)) & = \mathrm{aqr}((\mathcal{G}_0 \cup \{\delta_0\}) \cup \gamma_0) \subseteq \ldots \subseteq \mathrm{aqr}(\mathcal{G}_{i-1} \cup \{\gamma_i\}) = \\ & = \mathrm{aqr}((\mathcal{G}_i \cup \{\delta_i\}) \cup \{\gamma_i\}) \subseteq \ldots \subseteq \mathrm{aqr}(\mathcal{G}_k \cup \{\gamma_{k+1}\}) = \\ & = \mathrm{aqr}(\{\gamma_{k+1}\}). \end{array}$$

Therefore, we have shown, for any formula A, that

$$\blacktriangleleft_A = \mathrm{aqr}(\mathrm{pth}(A)) \subseteq \mathrm{aqr}(\{\gamma_{k+1}\}) = \theta_x(A),$$

where $x \in \{\mathsf{adeu}, \mathsf{aued}, \mathsf{edau}, \mathsf{euad}, \mathsf{d}, \mathsf{u}\}$, i.e., all these strategies are specializations of the quantifier precedence specification $\blacktriangleleft$. Therefore, they are correct, by Theorems 3.1.2 and 3.1.3. □

3.3.2 Heuristic Based Strategies

We first introduce the notion of a *heuristic function* in our context.

Definition 3.3.2 (Heuristic Function)
Let $\mathcal{A}$ be a set on which a total order $\sqsubset$ has been defined. Then any function $h : (\mathcal{F} \times \mathcal{V}) \to \mathcal{A}$ is called a *heuristic function.*

In our heuristic based strategies, we employ three heuristic functions that were originally designed for use in QDLL for the selection of splitting variables. The intended use of these heuristics is the choice of appropriate splitting variables in applications of the elimination rules from Definition 2.1.4.

Such heuristics are typically based on the CNF matrix of the formula, and the heuristics we choose are no exception in this aspect. Since the QDLL procedure applies the eliminations rules on different formulas within the semantic tree, the heuristic values for the same literal may be different for different nodes. During prenexing, however, we are confronted with a single static matrix, so we have to expect a globally different behavior.

We can, however, expect a similar behavior w.r.t. the topmost quantifiers. For the topmost quantifiers, the heuristics will return essentially the same result, since the matrices are the same. However, the deeper QDLL climbs down the semantic tree, the more deviations will occurs.

Our intuition is that our application of the heuristics might produce a prenex formula that has to offer attractive sets of variables for the topmost quantifiers. This is especially desirable, considering the fact that the topmost choices in reduction are potentially most critical for the runtime behavior of QDLL, as the early availability of the semantic value of the first branch can sometimes eliminate the need to consider large subtrees.

Definition 3.3.3 (Common Heuristic Functions)
Recall the literal counts that were introduced in Definition 1.1.19. We define the following heuristic functions.

Largest Combined Sum simply counts the number of occurrences of a variable. It is defined by

$$h_{\mathrm{lcs}}(A,q) = \sum_{i=1}^{\mathrm{mcl}(A)} (\mathrm{lcnt}(A,q,i) + \mathrm{lcnt}(A,\neg q,i)).$$

As the total order relation $\sqsubset$ from Definition 3.3.2, we choose the greater-than relation $<$ on $\mathbb{N}$.

Jeroslow–Wang is similar, but gives more weight to small clauses. It is defined by

$$h_{\mathrm{jw}}(A,q) = \sum_{i=1}^{\mathrm{mcl}(A)} (\mathrm{lcnt}(A,q,i) + \mathrm{lcnt}(A,\neg q,i)) \cdot 2^{-i}.$$

Here we choose $<$ on $\mathbb{Q}$ as the total order relation.

Boehm is the most intricate of our three heuristic functions. Just like the Jeroslow–Wang measure, it prefers small clauses, but instead of using different weights for clauses of different length, a vector is created, where each component corresponds to a specific clause length. A lexicographic order is then used to compare these vectors.

The heuristic function offers two tuning parameters, α and β, which determine the relative influence of positive literal occurrences (expectation: removal of a large number of clauses) versus negative ones (expectation: size reduction of a large number of clauses). We use the values $\alpha = 1$ and $\beta = 2$, which are suggested in literature [55, 43]. Given the auxiliary definition of

$$\begin{aligned} h_{\mathrm{bo}}(A,q,i) = {} & \alpha \cdot \max(\mathrm{lcnt}(A,q,i), \mathrm{lcnt}(A,\neg q,i)) + \\ & + \beta \cdot \min(\mathrm{lcnt}(A,q,i), \mathrm{lcnt}(A,\neg q,i)), \end{aligned}$$

the heuristic function is defined by

$$h_{\mathrm{bhm}}(A,q) = \begin{pmatrix} h_{\mathrm{bo}}(A,q,1) \\ \vdots \\ h_{\mathrm{bo}}(A,q,\mathrm{mcl}(A)) \end{pmatrix}.$$

As the total order relation $\sqsubset$ from Definition 3.3.2, we choose the lexicographic order $<_{lex}$, which is defined by

$$\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} <_{lex} \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} \quad \begin{array}{l} \text{if there exists some } k \leq n, \\ \text{such that } a_i = b_i, \text{ for all } i < k, \\ \text{and } a_k < b_k. \end{array}$$

Algorithm 3.3.2 (Heuristic Based Path Merging)
Signature: $\{\mathsf{bhmmax}, \mathsf{bhmmin}, \mathsf{lcsmax}, \mathsf{lcsmin}, \mathsf{jwmax}, \mathsf{jwmin}\} \times \tilde{\mathcal{F}} \rightarrow \mathcal{Q}^\star$

The algorithm takes an input formula A and computes a quantifier path, by performing the following steps:

1. If $\mathrm{pth}(A) = \emptyset$, then return the empty path ε as result.
2. Let $\gamma_0 = \varepsilon$. Choose any path $\delta_0 \in \mathrm{crit}(A)$, if possible one that ends in an existential quantifier[4]. and let $\mathcal{G}_0 = \mathrm{pth}(A)$.
3. Iterate the following steps, successively producing paths $\gamma_1, \gamma_2, \ldots$ and sets $\mathcal{G}_0, \mathcal{G}_1, \ldots$, until some set $\mathcal{G}_i$ is empty. For the i-th iteration, perform the following steps.
   (a) Considering the path $\delta_i = Rr\delta$, create a new path
   $$\gamma_{i+1} = \gamma_i Rr.$$
   (b) Generate a new set of paths $\mathcal{G}_{i+1}$, by stripping the leading quantifier Rr from all paths in $\mathcal{G}_i$, i.e., let
   $$\mathcal{G}_{i+1} = (\mathcal{G}_i \setminus \{Rr\delta \mid Rr\delta \in \mathcal{G}_i\}) \cup \{\delta \mid \delta \neq \varepsilon, Rr\delta \in \mathcal{G}_i\}.$$
   (c) If $\mathcal{G}_{i+1}$ is empty, then return the path γ_{i+1} as a result.
   (d) Let $\mathcal{H}$ the set of all paths from $\mathcal{G}_{i+1}$ with a maximal number of quantifier alternations, i.e.,
   $$\mathcal{H} = \{\delta \mid \mathrm{alt}(\delta) = \max\{\mathrm{alt}(\delta') \mid \delta' \in \mathcal{G}_{i+1}\}, \delta \in \mathcal{G}_{i+1}\}.$$
   If $\mathcal{H}$ contains a path that starts with the quantifier symbol $\overline{R}$, then let $\mathcal{H}'$ be the set of all paths from $\mathcal{G}_{i+1}$ that start with the quantifier symbol R, i.e.,
   $$\mathcal{H}' = \{\delta \mid \delta = Rs\delta', s \in \mathcal{V}, \delta \in \mathcal{G}_{i+1}\}.$$
   Otherwise, let $\mathcal{H}' = \mathcal{G}_{i+1}$.
   Now choose a new path $\delta_{i+1} = Rr\delta$ from the set $\mathcal{H}'$, depending on the prenexing strategy, as follows.
   Strategy bhmmax: Choose δ, such that $h_{\mathrm{bhm}}(A, r)$ is maximal.
   Strategy bhmmin: Choose δ, such that $h_{\mathrm{bhm}}(A, r)$ is minimal.
   Strategy lcsmax: Choose δ, such that $h_{\mathrm{lcs}}(A, r)$ is maximal.
   Strategy lcsmin: Choose δ, such that $h_{\mathrm{lcs}}(A, r)$ is minimal.

[4] For the same reason as in Algorithm 3.3.1.

Strategy jwmax: Choose δ, such that $h_{\mathrm{jw}}(A, r)$ is maximal.
Strategy jwmin: Choose δ, such that $h_{\mathrm{jw}}(A, r)$ is minimal.

Definition 3.3.4 (Heuristic Based Strategies)
The *simple symbol based* strategies θ_x, for

$$x \in \{\mathsf{bhmmax}, \mathsf{bhmmin}, \mathsf{lcsmax}, \mathsf{lcsmin}, \mathsf{jwmax}, \mathsf{jwmin}\},$$

are given by the following definition.

$\theta_x(A) = \mathrm{aqr}(\{\gamma_{x,A}\})$, where $\gamma_{x,A}$ is the output of Algorithm 3.3.2 for the input (x, A).

3.3.3 Dumb Strategies

Although Morale 1.3.1 tells us that we should prefer prenexing strategies that minimize the number of quantifier alterations in the prenex, such strategies need not necessarily be optimal in all cases. We can conceive certain classes of formulas for which rather simple strategies yield formulas for which the satisfiability problem is empirically easier to decide. Also, simple strategies introduce a smaller computational overhead into the translation process.

Considering that the set of quantifier paths of a formula forms a forest structure, we can easily derive a linearization of each tree along a depth or breadth first search. It is obvious that such a procedure preserves any previous precedences.

Definition 3.3.5 (Dumb Right Depth First Strategy)
The strategy *dumb right depth first* θ_{drdf} is given by the following definition.

1. $Q_1 q_1 \ \theta_{\mathsf{drdf}}(A) \ Q_2 q_2$, if $Q_1 q_1 \blacktriangleleft_A Q_2 q_2$;
2. $Q_1 q_1 \ \theta_{\mathsf{drdf}}(A) \ Q_2 q_2 \begin{cases} \text{if } A|_{\varrho 1 \pi_1} = (Q_1 q_1) B_1; \\ \text{if } A|_{\varrho 2 \pi_2} = (Q_2 q_2) B_2. \end{cases}$

Definition 3.3.6 (Dumb Right Breadth First Strategy)
The strategy *dumb right breadth first* θ_{drbf} is given by the following definition.

1. $Q_1 q_1 \ \theta_{\mathsf{drbf}}(A) \ Q_2 q_2$, if $Q_1 q_1 \blacktriangleleft_A Q_2 q_2$;
2. If $A_1 \circ A_2$ is a sub-formula of A, $\gamma_1 \in \mathrm{pth}(A_1)$ and $\gamma_2 \in \mathrm{pth}(A_2)$, then $\gamma_1 \, \theta_{\mathsf{drbf}}(A) \, \gamma_2$.

Theorem 3.3.2 (Correctness of Dumb Strategies)
The strategies θ_{drdf} and θ_{drbf} are correct.

Proof. For any formula A, $\theta_{\mathsf{drdf}}(A)$ and $\theta_{\mathsf{drbf}}(A)$ are, by definition, supersets of $\blacktriangleleft_A$, i.e., both strategies are specializations of the quantifier precedence specification $\blacktriangleleft$. Therefore, they are correct, by Theorems 3.1.2 and 3.1.3. □

3.3.4 An Example of Strategy Applications

Example 3.3.2 (Strategy Application)
We illustrate the effects of the various strategies on the formula

$$\begin{aligned} A = (\forall q_1)(((\forall r_1)(\exists r_2)B) \wedge \\ ((\exists s_1)(\forall s_2)(\exists s_3)(\forall s_4)(\exists s_5)C) \wedge \\ ((\exists t_1)(\forall t_2)D))), \end{aligned}$$

with the sub-formulas

$$\begin{aligned} B &= [\langle r_1\rangle, \langle r_2\rangle], \\ C &= [\langle q\rangle, \langle s_1\rangle, \langle s_2\rangle, \langle q, s_3, s_4, s_5\rangle, \langle q, s_3, s_4, \neg s_5\rangle, \\ &\quad \langle q, s_3, \neg s_4, s_5\rangle, \langle q, s_3, \neg s_4, \neg s_5\rangle, \langle q, \neg s_3, s_4, s_5\rangle, \\ &\quad \langle q, \neg s_3, s_4, \neg s_5\rangle]; \\ D &= [\langle q, t_1, t_2\rangle, \langle q, t_1, \neg t_2\rangle, \langle q, \neg t_1, t_2\rangle, \langle q, \neg t_1, \neg t_2\rangle, \\ &\quad \langle \neg q, t_1, t_2\rangle]. \end{aligned}$$

Since $\mathrm{sk}(A)$ is already in CNF, the heuristic values for the variables can be easily obtained directly from A. Table 3.1 summarizes these values.

Figure 3.1 represents the relation $\blacktriangleleft_A$ as a graph. The graph can be essentially interpreted like a HASSE diagram, with two specialties. Firstly, we have collected quantifiers $(Qq_1)\ldots(Qq_n)$ that

1. are pairwise not related,
2. share the same predecessor, and
3. share the same set of successors

into single nodes. Secondly, the nodes in the diagram are marked by the respective quantifier symbol and (in parentheses) the number quantifiers that each node represents.

Figures 3.2 through 3.9 augment Figure 3.1 with additional edges, drawn in a dashed style. Any of these edges indicates that the representation of the specialized relation $\theta_x(A)$ is obtained from Figure 3.1 by merging the corresponding nodes.

x	$h_{\mathrm{lcs}}(A,x)$	$h_{\mathrm{jw}}(A,x)$	$h_{\mathrm{bhm}}(A,x)$
q	12	24	(1,0,6,5)
r_1	1	8	(1,0,0,0)
r_2	1	8	(1,0,0,0)
s_1	1	8	(1,0,0,0)
s_2	1	8	(1,0,0,0)
s_3	6	6	(0,0,0,8)
s_4	6	6	(0,0,0,8)
s_5	6	6	(0,0,0,9)
t_1	5	10	(0,0,7,0)
t_2	5	10	(0,0,7,0)

Table 3.1: Heuristic values for formula A from Example 3.3.2.

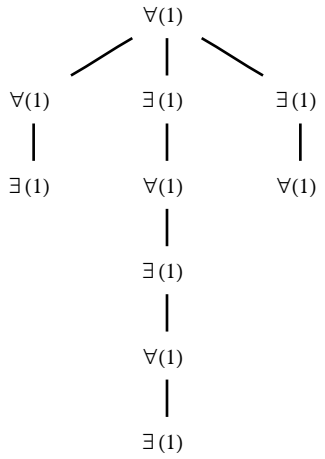

Figure 3.1: Structure of the quantifier precedence relation for formula A from Example 3.3.2.

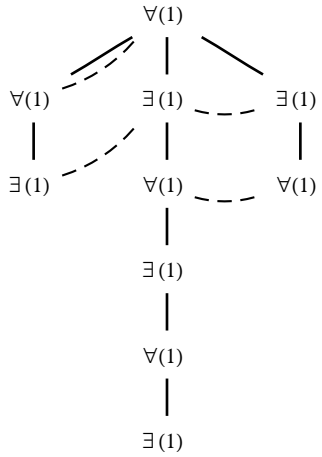

Figure 3.2: Application of strategy u (Example 3.3.2).

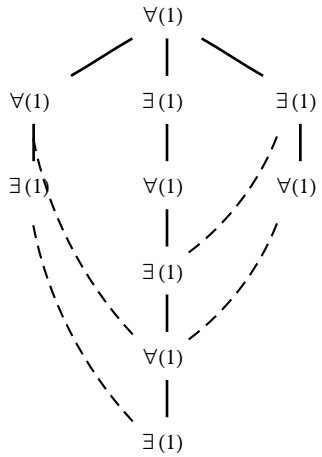

Figure 3.3: Application of strategy d (Example 3.3.2).

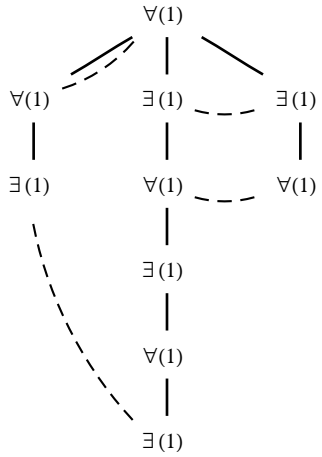

Figure 3.4: Application of strategy aued (Example 3.3.2).

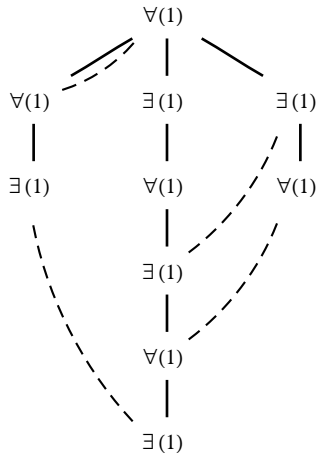

Figure 3.5: Application of strategy edau (Example 3.3.2).

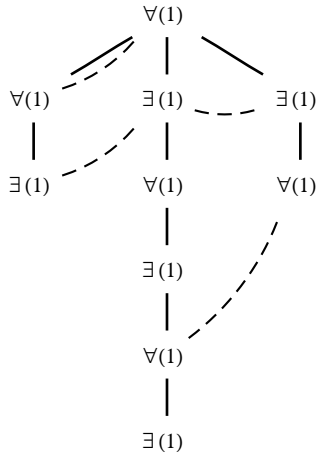

Figure 3.6: Application of strategy euad (Example 3.3.2).

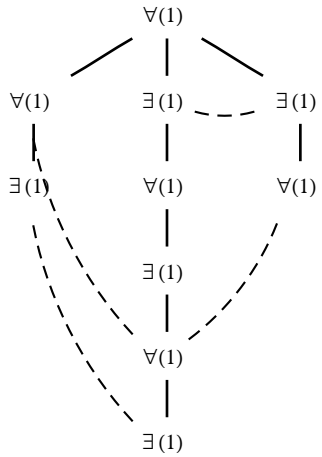

Figure 3.7: Application of strategy adeu (Example 3.3.2).

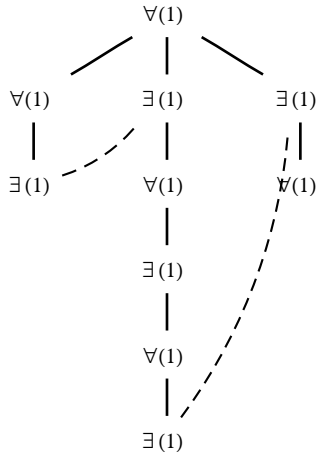

Figure 3.8: Application of strategy drdf (Example 3.3.2).

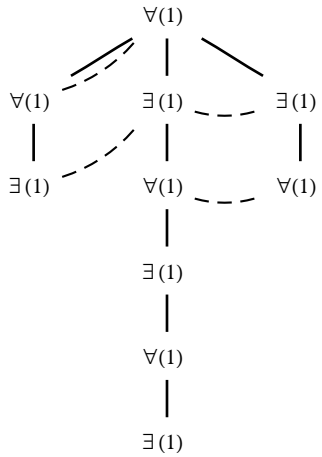

Figure 3.9: Application of strategy drbf (Example 3.3.2).

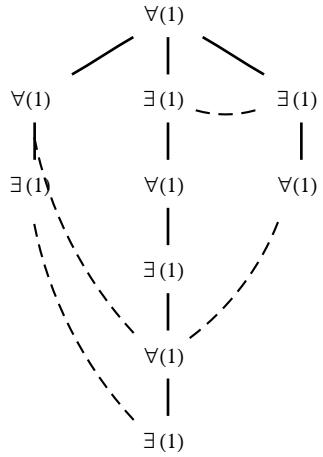

Figure 3.10: Application of strategy lcsmax (Example 3.3.2).

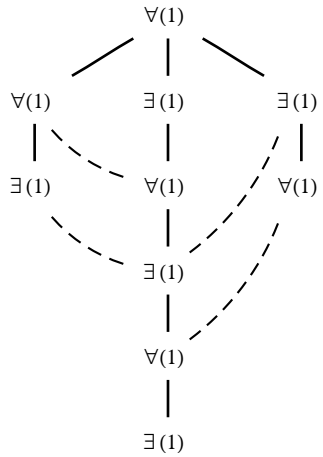

Figure 3.11: Application of strategy lcsmin (Example 3.3.2).

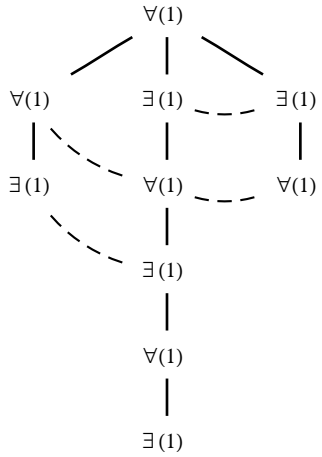

Figure 3.12: Application of strategy jwmax (Example 3.3.2).

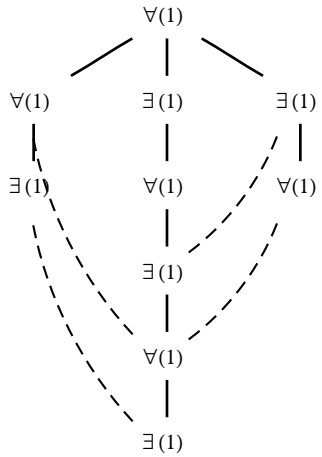

Figure 3.13: Application of strategy jwmin (Example 3.3.2).

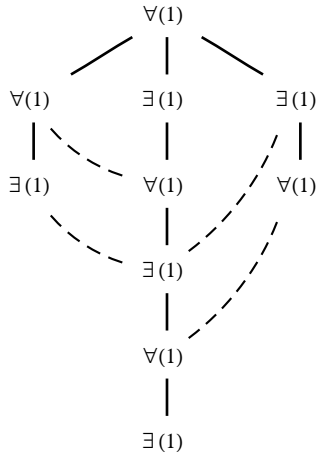

Figure 3.14: Application of strategy bhmmax (Example 3.3.2).

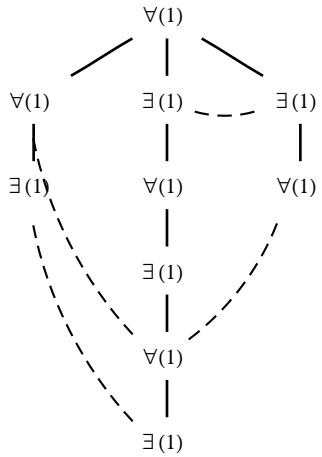

Figure 3.15: Application of strategy bhmmin (Example 3.3.2).

Chapter 4

Case Study: Nested Counterfactuals

In this chapter, we present a statistical experiment that was carried out with the objectives of

1. establishing statistical evidence that the choice of a prenexing strategy has an effect on the observable behavior of QSAT decision procedures and of quantifying such effects w.r.t. the strategies presented in Section 3.3, and

2. of obtaining preliminary insight into the connection between strategy choices and their effects.

For this, we consider the class of expressions known as *nested counterfactuals* [21, 29].

A counterfactual $A > B$ can be interpreted as a conditional query with an informal reading like "If A held, would then B necessarily hold too?" A nested counterfactual (NCF) is an expression obtained by placing (possibly negated and possibly nested) counterfactuals in the premise and/or conclusion of a counterfactual. Evidently, nested counterfactuals represent an interesting class of inferential reasoning problems, and their application, e.g., in diagnosis problems, should be quite obvious.[1]

Concerning the complexity of deciding the semantic value of nested counterfactuals, Eiter and Gottlob [21] have presented a wealth of interesting results. Notably, they have shown that the problem of deciding right nested counterfactuals is Π^P_{k+2}-complete for a nesting depths $< k$, and PSPACE-complete, if the nesting depth is unbounded.

In [20], we have presented a translation of right nested counterfactuals to QBFs. The QBFs obtained from this translation show enough structure to form a reasonable population for evaluating the behavior of the various prenexing strategies we have considered in Section 3.3.

[1] We will soon present a tiny example from this domain for illustrative purposes.

We are thus able to present quantitative empirical results about the effect of our prenexing strategies on the observable behavior of several QSAT decision procedures, w.r.t. an interesting population of formulas.

The outline of this chapter is as follows. In Section 4.1 we first introduce nested counterfactuals. In Section 4.2 we present a simple generator for random synthetic nested counterfactuals, which was used in our experiment. In Section 4.3 we present the QBF translation for right nested counterfactuals, which was first published in [20]. Section 4.4 explains how preliminary observations were used to work out sane ranges for the parameters for the nested counterfactual generator. These parameter ranges are discussed in detail in Section 4.5. An unpleasant complication is introduced into our experiment through the necessity to limit the maximal running time of the decision procedures used, through a timeout mechanism. We discuss the problems that may arise from this necessity in Section 4.6. In Section 4.7 we present the distribution of the empirical running times of the decision procedures that were used in our experiment, in order to obtain a first impression of our experimental data. Section 4.8 argues that we need a special measure in order to obtain a useful interpretation of our raw data. In Section 4.9, we propose such a measure, which is based on the notion of quantiles. In Section 4.10 we consider the running time overhead introduced by prenexing. Finally, we give an interpretation of our experiment in Section 4.11.

4.1 Nested Counterfactuals

In this work, we restrict ourselves to right nested counterfactuals, which are defined as follows.

Definition 4.1.1 (Right Nested Counterfactuals)
The set $\mathcal{C}$ of *right nested counterfactuals* is defined inductively by

1. $A, B \in \mathcal{B} \Rightarrow (A > B) \in \mathcal{C}$;
2. $A \in \mathcal{B}, N \in \mathcal{C} \Rightarrow (A > N) \in \mathcal{C}$;
3. $N \in \mathcal{C} \Rightarrow (\neg N) \in \mathcal{C}$.

Notation 4.1.1

1. We usually abbreviate nested counterfactuals of the form $(\neg(A > N))$ as $(A \not> N)$.
2. As usual, we are allowed to omit outermost parentheses. Furthermore, we assume the operators $>$ and $\not>$ to be right associative and may, as a consequence, drop some more parentheses.

Informally, a counterfactual $A > B$ can be read as "If A held, would then B necessarily hold too?" What distinguishes the counterfactual $A > B$ from the material implication $A \rightarrow B$ is the fact that, given a theory $\mathcal{T}$ implying $\neg A$, $A \rightarrow B$ is necessarily true, whereas $A > B$ need not be, because only A-consistent subtheories of $\mathcal{T}$ are considered in the latter case. The following example of a nested counterfactual, which considers a scenario taken from the realm of diagnosis, is due to [21].

Example 4.1.1
Assume that the structure and the functionality of the electric system of a car are encoded in a theory $\mathcal{T}$. We consider the following question: If the headlight switch was turned on (q_0) and the light did not shine ($\neg q_1$), then would it shine (q_1) if the fuse protecting the light would had been replaced (q_2) by a new one? We can formulate this question as the nested counterfactual

$$(q_0 \wedge \neg q_1) > (q_2 > q_1).$$

Definition 4.1.2 (Nesting Depth)
The *nesting depth* $\mathrm{nd}(N)$ of a nested counterfactual N is recursively defined by:

$$\mathrm{nd}(N) = \begin{cases} \mathrm{nd}(A > N') & \text{if } N = (A \not> N'); \\ 1 + \mathrm{nd}(N') & \text{if } N = (A > N') \text{ and } N' \notin \mathcal{B}; \\ 0 & \text{otherwise.} \end{cases}$$

Definition 4.1.3 (Evaluation Function for Nested Counterfactuals)
The evaluation function $v_{ncf} : \mathfrak{P}(\mathcal{B}) \times \mathcal{C} \to \{0, 1\}$ is recursively defined by

1. $v_{ncf}(\mathcal{T}, A \not> N) = 1 - v_{ncf}(\mathcal{T}, A > N)$;
2. $v_{ncf}(\mathcal{T}, A > N) = \begin{cases} 1 & \text{if } \mathrm{nd}(N) > 0 \text{ and for every } \mathcal{A} \in \mathrm{mct}(A, \mathcal{T}) \\ & \text{it holds that } v_{ncf}(\mathcal{A} \cup \{A\}, N) = 1; \\ 1 & \text{if } \mathrm{nd}(N) = 0 \text{ and for every } \mathcal{A} \in \mathrm{mct}(A, \mathcal{T}) \\ & \text{it holds that } \mathcal{A} \cup \{A\} \vdash N; \\ 0 & \text{otherwise.} \end{cases}$

Definition 4.1.4 (Validity of Nested Counterfactuals)
A nested counterfactual N is *valid* w.r.t. a theory $\mathcal{T}$, if $v_{ncf}(\mathcal{T}, N) = 1$. Otherwise, it is said to be *unsatisfiable* w.r.t. theory $\mathcal{T}$.

Theorem 4.1.1 (Complexity of Nested Counterfactual Evaluation)
1. The computational problem of evaluating right nested counterfactuals is Π^P_{k+2}-complete, if the nesting depth is bound by k.
2. The computational problem of evaluating right nested counterfactuals is PSPACE-complete, if the nesting depth is unbounded.

Proof. See [21]. □

4.2 The NCF Generator

At the early stage of our experiment, we developed a simple generator for nested counterfactuals (c.f. Section B.4), which is parameterized by the following variables: the number n of clauses, the number m of literals per clause, the number o of variables and the nesting depth k. Upon invocation, the formula generator produces a single nested counterfactual with the following structure:

1. $q_0 \not> (q_1 \not> (\ldots \not> (q_k \not> q_{k+1}) \ldots))$ for odd k, yielding QBFs with a decision problem in Σ^P_{k+2} and

2. $q_0 > (q_1 \not> (\ldots \not> (q_k \not> q_{k+1})\ldots))$ for even k, yielding QBFs with a decision problem in Π^P_{k+2}.

In both cases, the sequence $q_0, \ldots, q_n$ of propositional variables is a random combination (with repetitions) from a set of o distinct variables.

Moreover, an associated theory

$$\{l_{1,1} \vee \ldots \vee l_{1,m}, \ldots, l_{n,m} \vee \ldots \vee l_{n,m}\}$$

is generated, where the sequence

$$\lfloor l_{1,1} \rfloor, \ldots, \lfloor l_{1,m} \rfloor, \ldots, \lfloor l_{n,m} \rfloor, \ldots, \lfloor l_{n,m} \rfloor$$

is, again, a random combination (with repetitions) from the above set of variables. Each of these literals is negative with a probability of 0.5.

4.3 Translation of NCFs

What follows is a translation of right nested counterfactuals to QBFs, as presented in [20], with some minor modifications to improve the readability and, of course, the necessary adoptions to the notations used in the present work.

Definition 4.3.1 (NCF Translation)
Let N be a nested counterfactual with the structure

$$N = A_0 \succ_0 (A_1 \succ_1 (\ldots \succ_{k-1} (A_k \succ_k A_{k+1})\ldots)),$$

with $\succ_i \in \{>, \not>\}$. Let $k = \mathrm{nd}(N)$, and let

$$\mathcal{T} = \{B_0, \ldots, B_n\}$$

be a propositional theory. Furthermore, we assume the following sets of variables. Let

$$\mathcal{S}_i = \{s_{i,1}, \ldots, s_{i,n+i}\} \text{ and } \mathcal{U}_i = \{u_{i,1}, \ldots, u_{i,n+i}\},$$

for $0 \leq i \leq k$ be sets of fresh propositional variables, and let

$$\mathcal{W}_i = \mathrm{free}(\mathcal{T} \cup \{A_0, \ldots, A_i\}),$$

for $0 \leq i \leq k+1$. Now define formulas E_i, M_i, G_i, for $0 \leq i \leq k$, by

$$E_i = \Big(\bigwedge_{j=0}^{n+i}(s_{i,j} \rightarrow u_{i,j})\Big) \wedge \Big(\bigvee_{j=0}^{n+i}(\neg s_{i,j} \wedge u_{i,j})\Big);$$

$$M_i = \Big(\bigwedge_{j=0}^{n}\Big(\bigwedge_{l=0}^{i} s_{l,j}\Big) \rightarrow B_l\Big) \wedge \Big(\bigwedge_{j=1}^{i}\Big(\bigwedge_{l=j}^{i} s_{i,n+l}\Big) \rightarrow A_{l-1}\Big);$$

$$\begin{aligned} G_i = {} & (\exists \mathcal{W}_i)\Big(M_i \wedge A_i\Big) \wedge \\ & \wedge (\forall \mathcal{U}_i)\Big(E_i \rightarrow (\forall \mathcal{W}_i)(\neg M_i[s_{i,1}/u_{i,1}, \ldots, s_{i,n+i}/u_{i,n+i}] \vee \neg A_i)\Big). \end{aligned}$$

Lastly, we define formulas $H_i^{\mathcal{T},N}$, for $0 \leq i \leq k$, by

$$H_i^{\mathcal{T},N} = \begin{cases} (\forall \mathcal{S}_i)(G_i \rightarrow H_{i+1}^{\mathcal{T},N}) & \text{if } \succ_i = > ; \\ (\exists \mathcal{S}_i)(G_i \wedge \neg H_{i+1}^{\mathcal{T},N}) & \text{if } \succ_i = \not> , \text{ and} \end{cases}$$

$$H_{k+1}^{\mathcal{T},N} = (\forall \mathcal{W}_{k+1})((M_k \wedge A_k) \rightarrow A_{k+1}).$$

Then the desired encoding is given by the QBF $H_0^{\mathcal{T},N}$.

Theorem 4.3.1 (Correctness of NCF Translation)
Let $\mathcal{T}$ be a propositional theory and N a right nested counterfactual. It then holds that $v_{ncf}(\mathcal{T}, N) = v_{\mathrm{qbf}}(H_0^{\mathcal{T},N})$.

See [20] for the original presentation of this translation.

Example 4.3.1 (NCF Translation)
Consider the translation of a nested counterfactual

$$q_0 > (q_1 \not> (q_2 \not> (q_3 \not> (q_4 \not> (q_5 \not> (q_6 \not> q_7)))))).$$

The corresponding QBF has the structure

$$\begin{array}{l} (\forall S_0)(\neg G_0 \vee \\ \quad (\exists S_1)(G_1 \wedge \neg \\ \quad\quad (\exists S_2)(G_2 \wedge \neg \\ \quad\quad\quad (\exists S_3)(G_3 \wedge \neg \\ \quad\quad\quad\quad (\exists S_4)(G_4 \wedge \neg \\ \quad\quad\quad\quad\quad (\exists S_5)(G_5 \wedge \neg \\ \quad\quad\quad\quad\quad\quad (\exists S_6)(G_6 \wedge \neg((\forall \mathcal{W}_7)(M_6 \wedge q_6) \rightarrow q_7))))))), \end{array}$$

where

$$\begin{aligned} G_i = {} & (\exists \mathcal{W}_i)\Big(M_i \wedge A_i\Big) \wedge \\ & \wedge (\forall \mathcal{U}_i)\Big(E_i \rightarrow (\forall \mathcal{W}_i)(\neg M_i[s_{i,1}/u_{i,1}, \ldots, s_{i,n+i}/u_{i,n+i}] \vee \neg A_i)\Big), \end{aligned}$$

for $0 \leq i \leq 6$. At this point we should cleanse the formulas G_i, for $0 \leq i \leq 5$, replacing, the universally bound variables in $\mathcal{W}_i$ by fresh ones. We thus obtain

$$\begin{aligned} G_i' = {} & (\exists \mathcal{W}_i)\Big(M_i \wedge A_i\Big) \wedge \\ & \wedge (\forall \mathcal{U}_i)\Big(E_i \rightarrow (\forall \mathcal{W}_i')(\neg M_i'[s_{i,1}/u_{i,1}, \ldots, s_{i,n+i}/u_{i,n+i}] \vee \neg A_i')\Big), \end{aligned}$$

for $0 \leq i \leq 6$. Next we apply *unary quantifier shifting*, *double negation* and De Morgan*'s law*, in order to render the polarity of all quantified sub-formulas positive. This unveils the structure of the associated quantifier precedence relation (c.f. Definition 3.1.11), which is visualized in Figure 4.1:

$$\begin{array}{l} (\forall S_0)(G_0'' \wedge \\ \quad (\exists S_1)(G_1' \vee \\ \quad\quad (\forall S_2)(G_2'' \wedge \\ \quad\quad\quad (\exists S_3)(G_3' \wedge \\ \quad\quad\quad\quad (\forall S_4)(G_4'' \wedge \\ \quad\quad\quad\quad\quad (\exists S_5)(G_5' \wedge \\ \quad\quad\quad\quad\quad\quad (\forall S_6)(G_6'' \vee ((\forall \mathcal{W}_7)(M_6 \wedge q_6) \rightarrow q_7))))))), \end{array}$$

where

$$G_i'' = (\forall \mathcal{W}_i)\Big(\neg M_i \vee \neg A_i\Big) \vee \\ \vee\ (\exists \mathcal{U}_i)\Big(E_i \wedge (\exists \mathcal{W}_i')(M_i'[s_{i,1}/u_{i,1}, \ldots, s_{i,n+i}/u_{i,n+i}] \wedge A_i')\Big),$$

for even i with $0 \leq i \leq 6$.

Next, we elaborate our example by assuming

$$N = v_3 > (v_6 \not> (v_8 \not> (v_0 \not> (v_4 \not> (v_{10} \not> (v_{12} \not> v1))))))),$$

and a theory

$$\begin{aligned}\mathcal{T} = \{&v_1 \vee \neg v_7 \vee \neg v_{12} \vee v_8 \vee v_{11}, v_9 \vee \neg v_6 \vee \neg v_7 \vee \neg v_{13} \vee v_8,\\ &v_{10} \vee v_2 \vee \neg v_7 \vee v_{13} \vee v_5, v_{13} \vee v_2 \vee v_3 \vee v_{12} \vee \neg v_5,\\ &v_8 \vee v_7 \vee \neg v_0 \vee \neg v_4 \vee v_{12}, v_{10} \vee \neg v_9 \vee \neg v_5 \vee \neg v_{12} \vee \neg v_1,\\ &v_7 \vee v_1 \vee \neg v_0 \vee \neg v_3 \vee \neg v_8, \neg v_8 \vee v_0 \vee v_{10} \vee \neg v_{12} \vee v_{12},\\ &\neg v_1 \vee \neg v_4 \vee \neg v_{12} \vee \neg v_1 \vee \neg v_5, v_{12} \vee v_0 \vee v_{11} \vee \neg v_{13} \vee v_6,\\ &v_3 \vee \neg v_5 \vee \neg v_3 \vee \neg v_8 \vee \neg v_7, \neg v_5 \vee v_6 \vee \neg v_5 \vee \neg v_5 \vee v_8,\\ &v_{13} \vee v_6 \vee v_4 \vee v_7 \vee \neg v_3, v_6 \vee \neg v_{12} \vee v_5 \vee \neg v_{13} \vee \neg v_6,\\ &\neg v_{10} \vee \neg v_1 \vee \neg v_0 \vee \neg v_{12} \vee \neg v_{11}, v_3 \vee v_{13} \vee v_9 \vee v_{11} \vee v_4,\\ &\neg v_{13} \vee \neg v_{12} \vee v_9 \vee \neg v_8 \vee \neg v_2, v_1 \vee \neg v_{11} \vee \neg v_8 \vee v_5 \vee \neg v_2\}.\end{aligned}$$

We have already seen the quantifier precedence relation of the corresponding QBF in Figure 4.1. Now we will take a look at the sizes of each quantifier block. In Figure 4.2, these numbers are given in parentheses after each quantifier symbol (c.f. Example 3.3.2 for the semantics of this and subsequent figures). Figures 4.3 through 4.7 indicate the sizes of each quantifier block in the formulas after the application our prenexing strategies[2]. Note that, for the translations of nested counterfactuals, the strategies aued and edau yield the same result, as is easily demonstrated by the example of Figure 4.1. The same is, of course, true of the strategy pair adeu and euad, therefore we do not consider edau and euad through the remainder of this chapter.

Also note that all strategies, except for the *dumb* strategies, maintain the number malt($\cdot$) of quantifier alternations from the original formula. For the *dumb* strategies, this number increases, yielding a potentially harder formula (c.f. Theorem 1.3.4).

4.4 Initial Observations

The formula generator allowed us to perform preliminary exploratory tests to get a first real live impression of our NCF translation's and various decision procedures' behavior on translated NCFs. These tests eventually led to the preliminary demarcation of "reasonable" parameter ranges that were used to obtain the experimental results in [20]. In the latter work, the parameter ranges were then deliberately chosen rather wide—within the constraints imposed by

[2]Note that these figures were generated by the actual software, which might show some minor implementation specific differences from the explanations in this text.

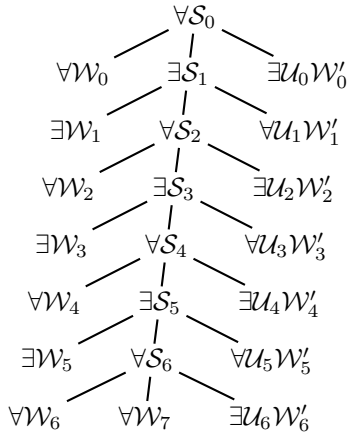

Figure 4.1: Quantifier order for the QBF from Example 4.3.1.

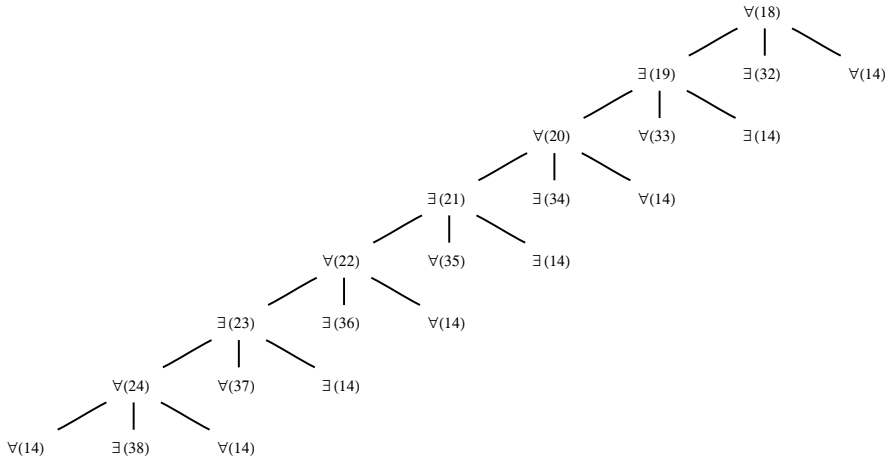

Figure 4.2: Structure of the quantifier precedence relation (Example 4.3.1).

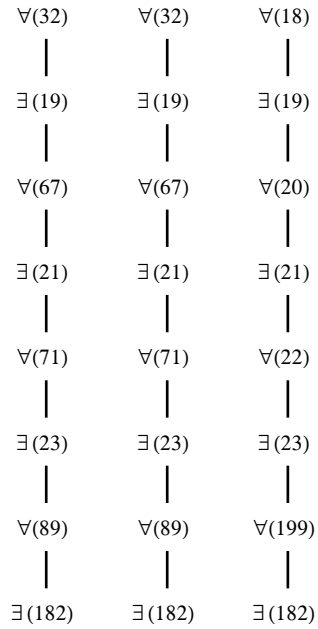

Figure 4.3: Prenex for strategies aued, edau and d (Example 4.3.1).

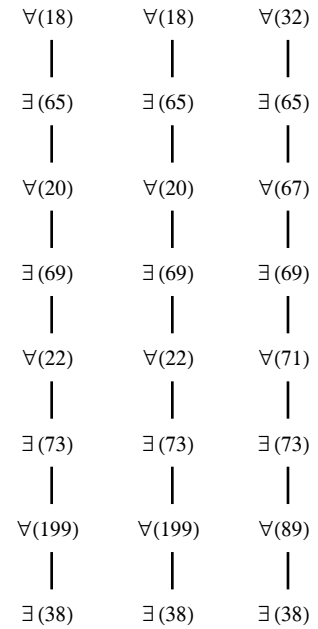

Figure 4.4: Prenex for strategies adeu, euad and u (Example 4.3.1).

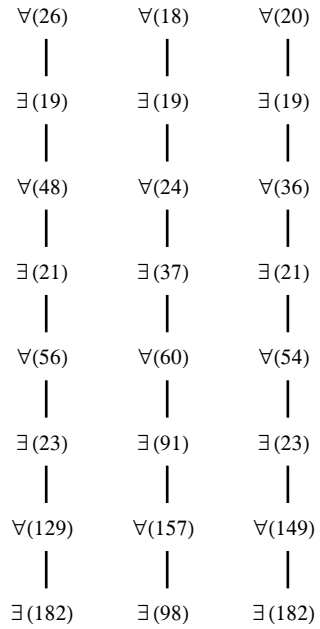

Figure 4.5: Prenex for strategies bhmmax, lcsmax and jwmax (Example 4.3.1).

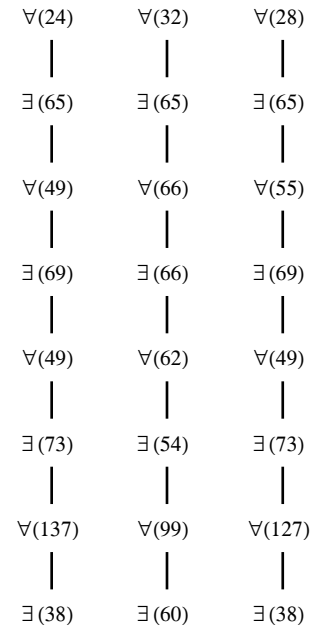

Figure 4.6: Prenex for strategies bhmmin, lcsmin and jwmin (Example 4.3.1).

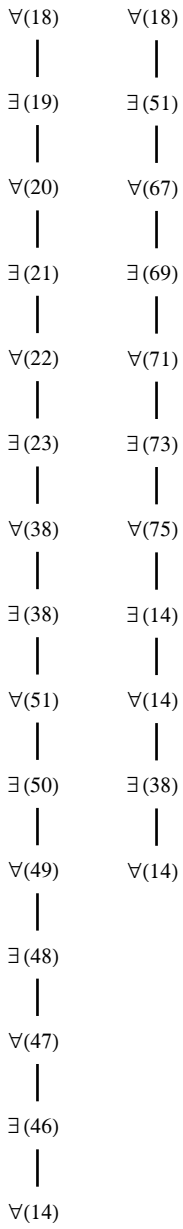

Figure 4.7: Prenex for strategies drdf and drbf (Example 4.3.1).

the availability of computing power—in order to obtain a general picture. For the present work, however, we revised our choice of ranges to obtain what we believe to be a very interesting and balanced population of formulas. We will explain the rationale behind our parameter choice shortly, after setting out some basic connections between the parameters in our NCF model and the corresponding QBFs obtained via our translation.

4.5 Parameter Ranges

In our experiment, the theories of the original NCFs have a range of 3 to 22 variables and a range of 6 to 30 clauses.[3] These numbers may seem tiny on the scale of quantified propositional logic, but when we look at the corresponding QBFs,[4] we find an average number of 1106 variables and 2293 clauses, which puts the potential hardness of the NCFs into a proper perspective.

The sharp increase in these numbers can be explained by the recursive introduction of a large number of variables during NCF translation, followed by a massive introduction of labels and short definitions during structure preserving normal form transformation.

More about the relation between the NCF variables and clauses versus their QBF counterparts can be learned from the conditioning plot in Figure 4.8. As not all readers will be acquainted with conditioning plots, we provide a short idea of these graphical representations of stochastic dependencies. For a complete introduction, see [10, 11].

Given two variables, α and β, we can form (possibly overlapping) subsets $\mathcal{A}_1^\alpha, \ldots \mathcal{A}_n^\alpha$ and $\mathcal{A}_1^\beta, \ldots \mathcal{A}_m^\beta$ of their corresponding ranges. A two-dimensional conditioning plot consists of a matrix of $n \times m$ panels containing individual subplots. Each subplot that is located in some row i and column j of the matrix can be conceived in its normal interpretation, with the additional constraint that $\alpha \in \mathcal{A}_i^\alpha$ and $\beta \in \mathcal{A}_j^\beta$. E.g., Figure 4.8 contains a 3×3 matrix of scatter plots of the number of QBF variables versus the number of QBF clauses. Each scatter plot is constrained to a certain interval $\mathcal{A}_i^\alpha$, for the number of NCF variables and a certain interval $\mathcal{A}_j^\beta$, for the number of NCF clauses. These intervals are indicated by they grey bars near the left hand side and near top of the figure, respectively.

Returning to our topic, we can see that the plot in Figure 4.8 indicates, besides a substantial correlation between QBF variables and clauses that an increase in NCF clauses is accompanied by an increase in both QBF clauses and variables. An increase in the number of NCF variable is associated with an increase in QBF variables either, albeit the influence is considerably smaller. On the other hand, the plot does not indicate any strong influence of the number of NCF variables on the number of QBF clauses.

Eventually, for completeness' sake, we should also consider the influence of the number of literals per clause. The plot in Figure 4.9 indicates that an increase of

[3] We will explain the choice of these ranges shortly.

[4] We hereby mean the final QBFs that are in PCNF and can be used as input for the decision procedures.

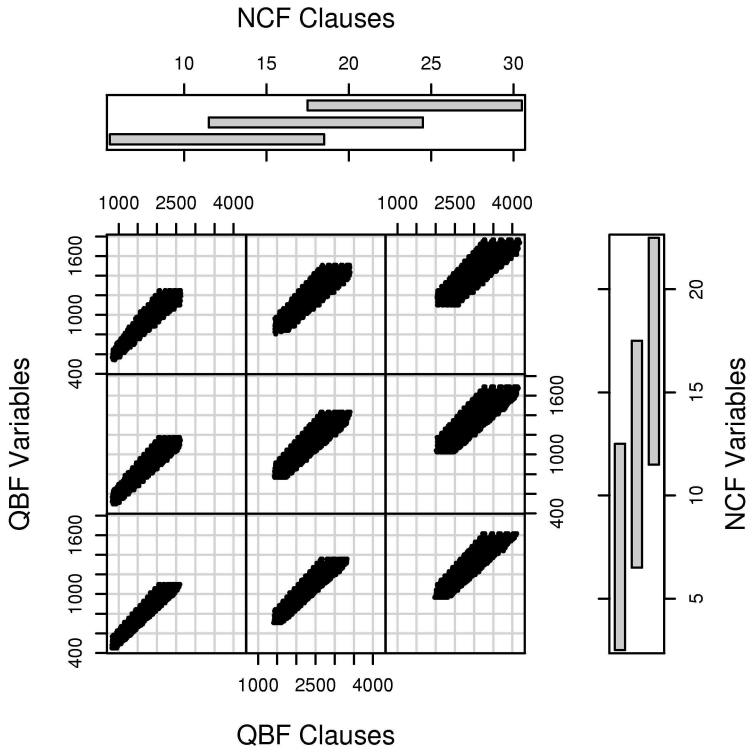

Figure 4.8: Relationship between NCF and QBF parameters.

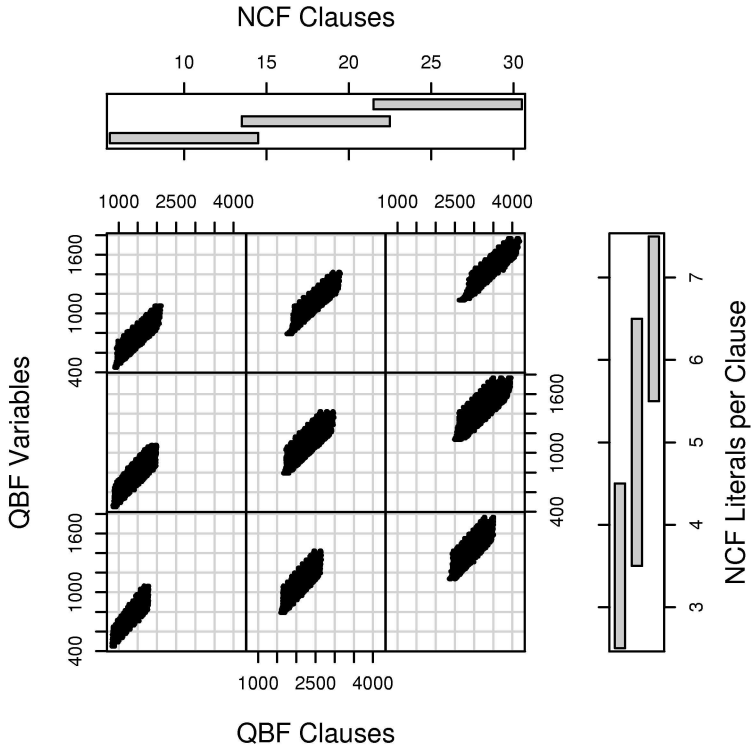

Figure 4.9: Relationship between NCF and QBF parameters.

this parameter is answered by a moderate expansion of the sample cloud in the dimension of QBF clauses (with a fixed point below the cloud's lower boundary in that dimension).

In a nutshell, it seems that the number of NCF clauses is the most influential parameter w.r.t. the basic QBF parameters of clause and variable number. However, these basic QBF parameters are not immediately suitable as a criterion for choosing parameter ranges.

Let us explain our choice of parameter ranges in detail. First, we decided to use, unlike in [20], a fixed nesting depth of 6. The rationale behind this choice was that the preliminary results in [20] indicate an exponential relationship between nesting depth of NCFs and the average empirical hardness[5], a fact that is consistent with the theoretical results of [21]. As this relationship is likely to considerably widen the gap between extremely easy and extremely hard formulas—a gap that can be huge even for a single nesting depth—it was decided that it was wiser to stick with a fixed value.

Next came the choice of the numbers of clause, variables and literals per clause. Our approach towards an interesting subsets of formulas[6] was to select the

[5]By *empirical hardness* we mean the hardness suggested by the observed running time of the various employed decision procedures.

[6]As the authors of [28] point out, it is especially important to avoid trivially insoluble problems.

range of these parameters, such as to obtain a population of formulas with a satisfiability ratio of roughly 50%, with an average hardness that would neither completely overwhelm, nor sub-challenge the chosen decision procedures. The rationale behind this choice is that a satisfiability ratio of 50% corresponds to minimal knowledge about the satisfiability of any particular formula of this class, making the class as a whole particularly interesting w.r.t. the amount of information obtainable by deciding the satisfiability of a given formula of that class.

Preliminary tests, starting with the lowest possible parameters, suggested that a higher number of clauses increases the average empirical hardness of translated NCF, up to a point where current decision procedures are brought to their knees. We thus cut off the number of clauses at a maximum of 30 clauses.

Next, it was observed that, for growing numbers of variables and literals per clause, starting with the lowest sensible number for those parameters, the empirical hardness of formulas would appear rather high up to a certain "critical" parameter value, at which it would drop rather sharply. Consequently, it was decided to set the maximum numbers of these parameter accordingly. Concretely, we chose 22 as maximum number of variables, and 7 as maximum number of literals per clause. We also cut off the less interesting and metrologically problematic easy formulas, setting either of the minimums to 3.

Eventually, we also cut of the easy formulas arising from a low number of clauses below 6.

Interestingly, this procedure already provided us with a class of formulas that met our criterion of having a empirical satisfiability ratio of roughly 50%,[7].

During the experiment, instances were selected by simple random sampling (SRS), allowing repetitions. Considering the huge population of formulas, this is practically identical to the possibly more common simple random sampling without repetitions.

We generated a total of 20000 nested counterparts. With 12 different strategies, this yielded a total of 240000 formulas.

4.6 Timeouts

Having fixed the formula parameters, we faced the problem of choosing an adequate timeout value for limiting the maximum time a given decision procedure would be allowed to spend on a given instance. We already knew that running time distributions of QBF decision procedures on typical formula classes are marked by an extremely long right tail, whereas a large part of the total mass of the distribution is concentrated near a running time of zero. In other words, most formulas are (empirically) fairly easy, but those few that are hard contribute mostly to the average running time. This is an extremely unpleasant

[7]The actual empirical satisfiability ratio is 48.07% which was obtained as follows. During our experiment, any given structured QBF was checked several times, using combinations of different prenexing strategies and decision procedures. To obtain the empirical satisfiability ratio, we did compute the ratio of satisfiable structured QBFs versus the total number of those structured QBFs, for which at least one prenexing strategy/procedure combination had produced a result.

situation, since it means that we may not neglect hard formulas. To obtain the associated data, we would have to use a very high timeout value (at least in the order of thousands of seconds, possibly even higher). However, resource constraints would then force us to limit ourselves to a rather modestly sized sample, which would in turn taint the quality of our results. One way out would be to use easier formulas, but this would force the large part of the formulas even tighter near a running time of zero, where running time measurement is problematic due to a high relative measuring error rooted, amongst others, in a lack of granularity of the timers available on our testing environment.

We eventually decided in favor of a big sample, settling the timeout value at 100 seconds. In more precise terms, this means that any formula that could be solved within the time interval $[0, 100)$ was recorded as normal data point, whereas all other formulas were timeouts.

The existence of a timeout condition raises the question of how to deal with situations in which a timeout actually arises. One solution would be to consider the corresponding samples as undefined. However, we know that these data points should actually correspond to some definite, albeit unknown, running time that is at least as big as the the timeout value. Therefore, recording these samples with a running time equal to the timeout value is sometimes advantageous. Besides, many standard definitions of statistics usually assume totally defined data. We will therefore assume a data set that was made total as was just sketched, up to Section 4.9, which, by its very nature, calls for a distinction of data points that are affected by timeouts.

As a very rough measure of the quality of a timeout parameter, we might consider the solved ratio of a sample:

Definition 4.6.1 (Solved Ratio)
Let $x_1, \ldots, x_n$ be a sample of the stochastic variable X of solving time. The *solved ratio* of that sample is the quotient between the number $m(\leq n)$ of defined sample points x_i (i.e., those that were not subject to timeouts) and the size n of the sample.

The solved ratio for our entire data set is 82.26%. However, we again emphasize that this measure can only be seen, at most, as a very rough indication of quality. As a rough consideration, we might, for example, regard a timeout value that leads to a solved ratio smaller than 0.5 as too low. However, it would make little sense to favor a solved ratio of, say 0.95 over 0.94, since there are many other influential parameters that should be considered, like, e.g., the concentration of the running time distribution or the total sample size. The latter is, of course, indirectly proportional to the chosen timeout value, given a fixed amount of total available CPU time.

4.7 Distribution of Running Times

Figure 4.10 shows the empirical cumulative distribution function of the solving time, factored by decision procedure. It immediately becomes clear that there is a very high concentration of mass near the left hand side, i.e., most of the

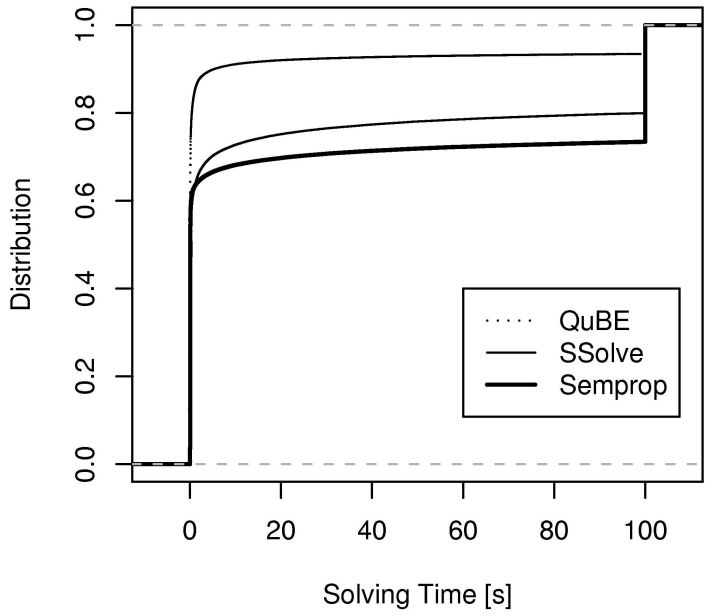

Figure 4.10: Empirical cumulative distribution function of the solving time, factored by decision procedure.

formulas were decided within a very short time. On the other hand, the manifestation of timeouts in the form of a jump of considerable height on the right hand side urges us to consider that the long right hand tail of the distribution still contains a fair share of the total mass.

We can see that the situation is essential the same for all three procedures. Most of the formulas are decided within milliseconds, but beyond a certain quantile—say, roughly at the 75%-quantile for SEMPROP, or the 90%-quantile for QuBE-BJ–the distribution function becomes almost completely flat, until it jumps to 1 at 100 seconds. This jump is, of course, due to timeouts (which were recorded with a running time of 100 seconds, the highest value that is a lower bound with a probability of 1).

It is quite clear that, in an ideal world without timeout, all three curves should slowly converge to 1, while extending towards the right a very, very far way, eventually hitting 1 at the running time needed to solve the very hardest formula in our huge, but yet finite population. The ridiculously flat slope beyond 90 seconds, for all three decision procedures, confirms us in our decision for a rather small timeout of 100 seconds, since even very generous timeouts–say one hour–could only catch a modest fraction of all timeouts. At this view, it seems more reasonable to use a smaller timeout and collect more data below that threshold, in order to obtain an accurate picture below that threshold. All the

more is this true, as the smoothness of various empirical distribution functions like the ones in Figure 4.10 offer no indication of any extraordinary behavior below the cutoff point.

When it comes to interpreting the data obtained by this process, it is clear that the timeout cutoff seriously distorts many of the positional parameters that are used in classical descriptive statistics, like the arithmetic mean and variance. On the other hand, the value of the α-quantiles remains completely unaffected by the cutoff, for suitably small values of α. Whereas the 0.5-quantile, the median, is valued, even in classical descriptive statistics, for its low susceptibility to outlier values, the use of quantiles to describe distributions is prevalent in the EDA[8] approach. In that approach, it is common practice to use the *summary point* (which correspond to the classical median) as measure for the center of a distribution. We use a variant of this measure, which is adapted to suit the specific peculiarities of our problem (like the missing data points that are due to timeout and performance differences between the decision procedures that were used).

4.8 Finding a Measure

Our goal is to quantitatively compare the behavior of each procedure/strategy combination. In a first approach, we might consider comparing the individual running time medians. However, the data of the corresponding contingency table (c.f. Table 4.1) is not too useful. Firstly, the data columns corresponding to different procedures are differently scaled, e.g., the medians for SSOLVE can be seen to be roughly 5 to 6 times higher than those of QUBE-BJ. Whereas this would not be such a big problem in itself, since we rather want to compare the effects of different strategies for each procedure individually that directly compare data for different procedures, what is worse is that all medians are located quite near to 0 seconds. The measurement of such small temporal intervals on traditional PC hardware[9] is notoriously prone to measurement errors, therefore we believe that it would be rather unwise to attempt a comparison based on positional measures in this range. Also, the different medians for each individual decision procedure are very close to each other, making them especially unattractive for a comparative analysis.

4.9 Working with Quantiles

If we give up the idea of obtaining a measure of center, there is no real reason to restrain ourselves to the 0.5-quantile. Instead, we might use any other quantile that seems reasonable. If we take a look at Table 4.2, we can see that higher quantile parameters tend to lead to more distinguished measures. From a practical point of view, a higher parameter seems interesting too, since the real

[8] Exploratory Data Analysis, see, e.g., [47].

[9] We hereby refer to the legacy 8254 hardware timers. Only very recently, widespread alternatives have been deployed (see, e.g., [12]).

	QuBE-BJ	SEMPROP	SSOLVE
adeu	0.04	0.08	0.2
aued	0.04	0.08	0.2
bhmmax	0.04	0.08	0.2
bhmmin	0.04	0.08	0.2
d	0.04	0.08	0.2
drbf	0.05	0.08	0.2
drdf	0.04	0.08	0.2
jwmax	0.04	0.08	0.2
jwmin	0.04	0.08	0.2
lcsmax	0.04	0.08	0.2
lcsmin	0.04	0.08	0.2
u	0.04	0.08	0.2

Table 4.1: Median running time of each procedure/strategy combination.

	0.2	0.3	0.4	0.5	0.6	0.7	0.8
adeu	0.03	0.05	0.07	0.10	0.18	0.84	30.07
aued	0.03	0.05	0.07	0.10	0.17	0.75	12.54
bhmmax	0.03	0.05	0.07	0.10	0.17	0.85	18.55
bhmmin	0.03	0.05	0.07	0.10	0.18	1.12	68.37
d	0.03	0.05	0.07	0.11	0.19	0.98	29.05
drbf	0.04	0.05	0.07	0.10	0.17	0.93	100.00
drdf	0.03	0.05	0.07	0.10	0.17	0.70	29.19
jwmax	0.03	0.05	0.07	0.10	0.18	0.94	21.01
jwmin	0.03	0.05	0.07	0.10	0.18	1.16	60.36
lcsmax	0.03	0.05	0.07	0.10	0.18	0.95	22.60
lcsmin	0.03	0.05	0.07	0.10	0.18	1.32	83.12
u	0.03	0.05	0.07	0.10	0.17	0.84	23.36

Table 4.2: Divergence of α-quantiles for increasing α.

benefit of a suitable prenexing strategy should manifest itself in the optimization not of the trivial cases, but of those with upscale hardness.[10]

Clearly, if we intend to compare the behavior of strategies, the quantile parameter must not itself depend on strategies, but, as was pointed out before, it may depend on the procedure used. Our idea is to normalize each data column such that the "worst" strategy determines the amount of scaling performed. Motivated by Table 4.2, we are eager to maximize the quantile parameters. However, we must take care not to push these parameters to high. For a suitably high parameter α, the α-quantile will have the timeout value of 100 seconds, which is outside of the range $[0, 100)$ of meaningful time parameters. In order to find

[10]The attentive reader might have observed that, in the above discussion about quantiles, we have highlighted their restricted susceptibility to outlier values. It is true that, considering the extreme concentration of running times near zero, we might consider almost everything else as an "unusual case". However, the real outliers in this case can be found very far from zero, much further than we we will push the quantiles. Perhaps the most expensive two or even less percent of all formulas could be considered as outliers.

	α_x
QuBE-BJ	0.92
SEMPROP	0.67
SSOLVE	0.76

Table 4.3: Quantile parameters used in Table 4.4.

the maximal meaningful parameter, reconsider the definition of the solved ratio (Definition 4.6.1). The following theorem can seen to be true.

Theorem 4.9.1 (Cutoff)
Let $x_1, \ldots, x_n$ be a sample of the stochastic variable X of solving time that was obtained with a timeout value of t, and let a be the solved ratio of that sample. Then exactly $a \cdot n$ samples are smaller than t, and, conversely, $(1-a) \cdot n$ samples correspond to timeouts.

Proof. Assuming a solved ratio of a and a sample size of n, the number of defined sample point is $a \cdot n$, by Definition 4.6.1. We know that the undefined sample points are exactly those data point, where timeouts occurred. So there must be $a \cdot n$ samples that are smaller than the timeout value t, because otherwise they could not be defined. Likewise, any of the $n - (a \cdot n) = (1-a) \cdot n$ undefined data points correspond to timeouts. □

The essence of this theorem is that the a-quantile of a sample $x_1, \ldots, x_n$ that was obtained without any timeout constraint is equal to the a-quantile of the sample $x'_1, \ldots, x'_n$ that is obtained after "cutting off" sample points that are greater or equal to a given timeout value t, where a is the solved ratio of $x'_1, \ldots, x'_n$. This process corresponds exactly to the cutoff experienced through the use of a timeout during the experiment. Therefore, we can see that the a-quantile remains completely unaffected by timeouts, a very desirable property. It is known that quantiles, viewed as functions on their parameter α, are monotonically rising. It is therefore not hard to see that the above unaffectedness property holds for any α with $0 \leq \alpha \leq a$. Also note that the same needs not be true for an α with $\alpha > a$.

Let $a_{1,1}, a_{1,m}, \ldots, a_{n,m}$ be the solved ratio for each of the $n \cdot m$ combinations of one of n procedures and one of m strategies. We will then use $\alpha_i = \min_{1 \leq j \leq m} a_{n,j}$, for $1 \leq i \leq n$ as quantile parameter for the i-th procedure, i.e., we choose, for each procedure, the greatest possible quantile parameter α_i such that the corresponding quantile is unaffected by timeouts (for any strategy).

These optimal quantile parameters are presented in Table 4.3. Considering Figure 4.10, it should be no surprise that the quantile values are very sensitive w.r.t. these parameters, so that small numerical errors in the parameters might show up in the quantile values. However, this does not affect the quality of our result, as the figures for each decision procedure are still equally scaled, and therefore comparable, as they share the same parameter. Finally, note that the data in Tables 4.3 and 4.4 were rounded for the presentation in the work (however, we used full precision for the calculations, of course).

	QuBE-BJ	SEMPROP	SSOLVE
adeu	6.50	5.11	48.50
aued	19.62	1.84	10.89
bhmmax	24.73	3.38	12.01
bhmmin	33.69	9.92	52.19
d	72.32	4.55	16.11
drbf	37.06	99.53	32.67
drdf	2.31	20.89	6.30
jwmax	40.23	3.64	12.92
jwmin	41.67	6.48	51.59
lcsmax	16.98	4.60	15.58
lcsmin	98.57	5.66	99.95
u	17.48	2.40	31.06

Table 4.4: α_x-quantiles of the running times of each procedure/strategy combination. The corresponding parameters are shown in Table 4.3.

	QuBE-BJ	SEMPROP	SSOLVE
adeu	2	7	9
aued	5	1	2
bhmmax	6	3	3
bhmmin	7	10	11
d	11	5	6
drbf	8	12	8
drdf	1	11	1
jwmax	9	4	4
jwmin	10	9	10
lcsmax	3	6	5
lcsmin	12	8	12
u	4	2	7

Table 4.5: Ranks of the α_x-quantiles of the running times of each procedure/strategy combination. The corresponding parameters are shown in Table 4.3.

Table 4.4 presents our summarizing table of quantile values. Each column contains the quantile values for one decision procedure, w.r.t. the corresponding optimal quantile value from Table 4.3, factored by strategy. We can now clearly identify which strategies did yield the best results w.r.t. the running time spent by the corresponding decision procedure. However, we note that a comparison between different decision procedures does not make any sense at all, based on this table. Also, the actual quantile values are not too interesting, what counts are the ranks of these values. So address both issues by turning to ranks. We thus obtain Table 4.5, which is still a bit complex. We can simplify the presentation by giving, for each decision procedure, a vector of all strategies, sorted by quantile rank (Table 4.6).

	QuBE-BJ	SEMPROP	SSOLVE
1	drdf	aued	drdf
2	adeu	u	aued
3	lcsmax	bhmmax	bhmmax
4	u	jwmax	jwmax
5	aued	d	lcsmax
6	bhmmax	lcsmax	d
7	bhmmin	adeu	u
8	drbf	lcsmin	drbf
9	jwmax	jwmin	adeu
10	jwmin	bhmmin	jwmin
11	d	drdf	bhmmin
12	lcsmin	drbf	lcsmin

Table 4.6: Strategies, sorted by their α_x-quantile rank, for each procedure.

4.10 Translation Overhead

Up to this point, we were only concerned about the running times of decision procedures, but we have not considered the overhead produced by the normal form translation. However, that extra expense in running time might be an important factor to the actual usefulness of a strategy. After all, a seemingly excellent prenexing strategy, or more precisely a specific implementation of it, would be of little use w.r.t. a certain decision procedure, in case its running time overhead turned out to compensate the speedup observed in that decision procedure.

We have to mention that the main focus of our implementation was the development of an easily-extensible research vehicle for testing various (possibly fundamentally different) prenexing strategies. On the other hand, speed was not a main objective, a fact that needs to be considered when interpreting these results. We suspect that an optimized implementation will perform considerably better, although it is very hard to give any precise estimation.[11]

Even though we did not consider speed as a main goal, we did nonetheless try to avoid any unnecessary waste of CPU time. To this end, we did also perform several profiling analyses. Besides providing us with many useful insights that were used to improve our implementation, the parser component, which is implemented using the Parsec [36] parser combinator library was identified as a major devourer of CPU time. Considering the fact that real life applications of our strategies would probably tightly integrate prenexing into decision procedures, we do not consider the time spent on parsing as a relevant factor. Consequently, Table 4.7, which gives the median measured translation times for each strategy, contains three columns: the time spent on reading the input formula (parsing), the time spent for the actual calculation and printing and the total running time—which is of course the sum of the previous two columns. The only column we consider relevant is thus the middle one, whereas the other

[11] To give an idea of the speedup potential, consider the fact that our implementation is given in Haskell 98 [34], and was compiled using a current version of GHC [45].

	parsing	rest	total
adeu	0.49	0.53	1.02
aued	0.49	0.53	1.02
bhmmax	0.49	1.77	2.26
bhmmin	0.49	1.36	1.85
d	0.49	0.53	1.02
drbf	0.49	0.53	1.02
drdf	0.49	0.53	1.02
jwmax	0.49	2.22	2.71
jwmin	0.49	1.82	2.31
lcsmax	0.49	1.69	2.18
lcsmin	0.49	1.28	1.77
u	0.49	0.53	1.02

Table 4.7: Median translation times for each strategy.

two were given for completeness' sake. Unfortunately, the lazy evaluation-based functional design of the software did not allow us to easily factor out the time spent on printing, which we likewise regard as non-relevant factor. The performance loss through printing is much lower than the loss through parsing, although not altogether negligible.

We can see that running time expense of the heuristic based strategies is considerably higher than that of the rest of the strategies. During our profiling analysis, it turned out that a big extra penalty is introduced by the calculation of the various heuristic values. However, the current implementation of these calculations is rather wasteful and leaves ample room for optimization.

4.11 Interpretation

When we take a look at Table 4.6, the first thing that we can see is that the aptness of any given strategy is highly dependent on the decision procedure used. This is not an unexpected result, considering the unlike results for different decision procedures that were given in [20]. It is reasonable to presume that this diverse sensitivity is essentially founded in the different subsets of advanced techniques that each decision procedure employs (c.f. Section 2.2).

Whereas the data in [20] already clearly indicated that the quality of a strategy depends on the decision procedure used, we now get a clearer look at the complexity of this dependency. It seems hard to identify any regular pattern in the connection between the two factors: strategy and decision procedure.

One thing that catches the eye is the fact that the simple drdf strategy wins the performance race for two out of the three decision procedures. For the third one (SEMPROP), however, the two Dumb strategies turn out the most unsuitable strategies. So, although drdf is a very desirable strategy for two decision procedures, it is not very robust.

When it comes to terms of robustness w.r.t. to the decision procedures, jwmin

excels[12], but unfortunately it is located at the lower end of our performance ranking.

If we were to choose a strategy that yield good performance, but also be reasonably robust, we would probably choose aued, which yields excellent performance for SEMPROP and SSOLVE. For QUBE-BJ, the performance ranking of aued at least falls among the better half of all strategies.

If we take a look at the heuristic based strategies, we see that, any -max strategy always yields a better performance than its -min counterpart. More generally, with a single exception, the -max strategies always yield a better performance that *any* of the -min strategies.

[12]Of course, we must be careful not to generalize this property over other decision procedures.

Chapter 5

Conclusion and Future Work

In this work we confirm the results from [20] by performing a broad analysis of the interaction between various normal form transformation strategies on a number of contemporary decision procedures [23, 31, 32, 33, 37, 43].

Chapter 1, gives a concise introduction into *quantified propositional logic*. We introduce the syntax and semantics of *quantified boolean formulas* and discuss the most important reasoning tasks associated with these formulas, most importantly *QSAT*, the satisfiability problem of quantified propositional logic. We introduce a number of useful notions and notations that are used throughout this work. We consider some important normal forms for quantified boolean formulas and some corresponding transformations, most importantly *prenex conjunctive normal form (PCNF)*.

Whereas virtually all beginners' textbooks on mathematical logic introduce propositional and first order logic, there is hardly any introductory level treatise on quantified propositional logic (one exception is [35], which contains a chapter on the topic). We believe that the first chapter of this work can serve as a short, self-contained introduction into quantified propositional logic that includes many proofs of simpler theorems that are typically omitted in similar treatises.

Chapter 2 introduces *QDLL*, which is one of the most important QSAT decision procedures for quantified boolean formulas in PCNF. Our approach is distinguished by our attempt to obtain a "pure" QDLL procedure by abstracting from implementational details, especially those concerning the concrete traversal of the search space. Such details, which obscure the look on the core procedure, are typically found in treatises of contemporary decision procedures, e.g., [32].

Our treatise of the QDLL procedure concludes with an overview of advanced techniques that are used to improve the runtime behavior of contemporary QSAT decision procedures. Although we only include those techniques that are relevant to the case study in Chapter 4, our presentation does, to the best of our knowledge, form the most comprehensive overview of such techniques.

In Chapter 3, we introduce a novel theoretical framework for the description of *prenexing strategies*, i.e., prescriptions that specify the behavior of a prenexing algorithm. Furthermore, we present three distinct classes of prenexing strategies, yielding a total of fourteen different strategies.

The framework presented is suitable for the description of PNF transformations that preserve the set of quantifiers, and do not change the propositional skeleton of a formula.

With such restrictions, the framework is able to capture many important transformations, but if we want to capture more advanced strategies, like those that merge quantifiers by application of the *quantifier fusion rule* from Chapter 1, we need to loosen the constraints. The corresponding generalization of the framework represents promising future work.

In Chapter 4, we present a case study that we conduct with the goal of testing the real life behavior of the prenexing strategies from Chapter 3. We first introduce nested counterfactuals, along with a translation of such formulas to quantified boolean formulas as are not in normal form. We then describe in detail the settings and results of an experiment, in which the runtime behavior of various contemporary QSAT decision procedures was tested for QBFs in PCNF that arise from the application of the different prenexing strategies on the translations of randomly generated nested counterfactuals.

As the main result of our experiment, we present a quality ranking of prenexing strategies. Interestingly, the assessment of most strategies turns out to be completely different for different decision procedures. As a likely explanation for the phenomenon of diverse sensitivity, we point out that the different decision procedures that were used in our experiment, although essentially based on the same basic procedure, employ different subsets of advanced techniques.

The big challenge in future work will be the development of an understanding of the structure of the interaction between prenexing strategies and decision procedures. Ideally, each of the advanced techniques described in Section 2.2 and each prenexing strategy would be associated with some characteristic interaction pattern. However, while a detailed analysis of each such interaction is already far from trivial, it is extremely unlikely that the overall behavior of the system could be described as the simple sum of these interactions. It is quite certain that the interaction of different techniques will lead to completely novel behavior when such features are combined. For a combination of n advanced techniques, we have to expect $(n^2 - n)/2$ interactions, where n is constantly rising, as contemporary decision procedures keep on integrating new techniques.

Given such prospects, a more controlled environment for experimentation would be highly desirable. For this, the implementation of a plain testbed procedure might turn out be practical.

In this work, we consider QBF translations of nested counterfactuals as an interesting class of structured formulas, but there are, of course, many other classes that deserve close inspection w.r.t. prenexing. In particular, we see that the translations of nested counterfactuals feature a rather simple quantifier precedence relation, whereas other classes of formulas might offer a wealth of more intricate structures.

Appendix A

Adopted Concepts

A.1 Mathematical Concepts and Notations

A.1.1 Operators on Sets

For any set $\mathcal{A}$,

1. $\mathfrak{P}(\mathcal{A})$ denotes the power set of $\mathcal{A}$, and
2. $\mathcal{A}^\star$ denotes the Kleene closure of $\mathcal{A}$ (see below).

The Kleene closure of a set $\mathcal{A}$ is defined as the set $\bigcup_{i=0}^{\infty} \mathcal{A}^i$, where $\mathcal{A}^n$ for $n \geq 1$ denotes the n-ary cartesian product $\mathcal{A} \times \ldots \times \mathcal{A}$ and $\mathcal{A}^0 = \{\varepsilon\}$. $\mathcal{A}^\star$ can be interpreted as the set of strings over $\mathcal{A}$, where ε denotes the empty word. We usually assume an implicit concatenation operator, i.e., if $x, y \in \mathcal{A}^\star$, then xy denotes the concatenation of x and y, and therefore, of course, $xy \in \mathcal{A}^\star$.

A.1.2 Operators on Binary Relations

For any binary relation $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{A}$,

1. $\mathcal{R}^+$ denotes the transitive closure of $\mathcal{R}$, and
2. $\mathcal{R}^*$ denotes the reflexive and transitive closure of $\mathcal{R}$.

A.2 Machines and Complexity

A.2.1 Alternating Turing Machines

The alternating Turing machine (ATM) is an extension of the standard Turing machine model that is perfectly suited for explaining the complexity of QSAT. We give a short informal treatment of the ATM model, adopted from [49] and [52]. For a thorough treatment, the reader is referred to [49].

As with nondeterministic Turing machines, a computation of an ATM is represented as a computation tree, where every node corresponds to a particular configuration and different branches rooted in one node represent different possible transitions. Every state is tagged with exactly one of the labels *existential*, *universal*, *accepting* and *rejecting*, where *accepting* and *rejecting* are terminal states in which the machine halts. The tag of a configuration is defined as the tag of its state.

Each node of the computation tree has a quality *accept*, *reject* or *undef*, which is recursively determined by the following set of rules:

1. the quality of a leaf node corresponding to an accepting (resp. rejecting) configuration is *accept* (resp. *reject*);
2. the quality of an internal node corresponding to an existential configuration is *accept*, if at least one of its successors has the quality *accept*;
3. the quality of an internal node corresponding to an existential configuration is *reject*, if all of its successors have the quality *reject*;
4. the quality of an internal node corresponding to a universal configuration is *reject*, if at least one of its successors has the quality *reject*;
5. the quality of an internal node corresponding to a universal configuration is *accept*, if all of its successors have the quality *accept*;
6. the quality of all other nodes is *undef*.

By definition, an ATM accepts its input, if the root node of the corresponding computation tree has the quality *accept*.

Associated with ATMs is the following hierarchy of complexity classes:

$$\mathsf{ALOGSPACE} \subseteq \mathsf{APTIME} \subseteq \mathsf{APSPACE} \subseteq \mathsf{AEXPTIME}$$

A.2.2 The Polynomial Hierarchy

The polynomial hierarchy, introduced in [38, 44], is a hierarchy of complexity classes spanning from P to PSPACE.

The structure of the hierarchy w.r.t. set inclusion is indicated in Figure A.1. Unfortunately, it is unknown whether these inclusions are proper ones (see, e.g., [39] and [52]).

In algorithmic complexity theory, these classes are defined through the notion of oracle Turing machines (see, e.g., [52]). Roughly speaking, an oracle Turing machine "with an oracle of complexity C" (where C is some complexity class) is allowed to consult its oracle to solve any problem in C in constant time. E.g., an oracle Turing machine with an oracle of complexity NP might solve any NP-hard problem in constant time (usually as part of solving a more complex problem). Σ_{n+1}^P is then defined, for any n, as the class of problems that can be solved in polynomial time on a non-deterministic Turing machine with an oracle of complexity Σ_n^P and Σ_0^P is defined to equal P. Δ_n^P is defined analogously to

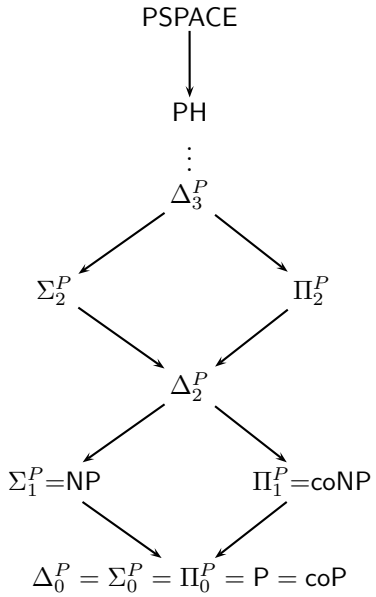

Figure A.1: The $\subseteq$ inclusions of the complexity classes that form the polynomial hierarchy.

Σ_n^P, but for deterministic Turing machines, i.e., Δ_{n+1}^P is the class of problems that can be solved in polynomial time on a deterministic Turing machine with an oracle of complexity Σ_n^P and Δ_0^P is again defined to equal P. Finally, Π_n^P is defined as $\mathsf{co}\Sigma_n^P$ (see, e.g., [52]).

A more pleasing, but equivalent, characterization of the classes of the polynomial hierarchy is perhaps given by the following Definition (see Appendix A.2.1 for a definition of alternating Turing machines).

Definition A.2.1 (Polynomial Hierarchy)

1. Σ_n^P (resp. Π_n^P) is the class of all languages that are accepted in polynomial time by an alternating Turing machine, where the number of alternations in the computation trees is bounded by n and the root node is existential (resp. universal);

2. $\mathsf{PH} = \bigcup_{k=0}^{\infty} \Sigma_k^{\mathsf{P}}$.

Appendix B

The Traquasto Software Package

This Appendix provides a user oriented documentation for the major components of the TRAQUASTO[1] software package. The package was developed by Martina Seidl and Michael Zolda, and was first applied in the production of the benchmarks for [20]. Since then, the software has been maintained and extended by Michael Zolda. This documentation refers to version 0.5 of TRAQUASTO.

B.1 Development Status

The development of all our tools and especially our translation tool QST has been continued since the publication of [20]. Besides the implementation of new prenexing strategies, QST has most notably been improved in terms of speed, strengthening the implicit assumption that many of our prenexing strategies can be of practical value when implemented with serious speed considerations in mind. Despite our success in making our current implementation faster, we currently see QST more as an experimental vehicle that sometimes sacrifices speed for code simplicity.

B.2 Traqla: Translation Tool

TRAQLA[2] is an implementation of the nested counterfactual translation presented in [20] and Chapter 4 of this work. It was implemented in SWI Prolog [53] and is available as a command line tool. It reads nested counterfactuals in *ebf* format, and outputs the corresponding QBFs in BOOLE format. See Section B.5 for a description of these formats.

[1] TRAQUASTO is an acronym for Translation and Quantifier Shifting Toolkit.
[2] TRAQLA (pronounced "Dracula") is an acronym for Translate to QBF Language.

B.2.1 Command Line Interface

Synopsis

`traqla [`*`OPTION`*`...]`

Options

`-b`
Generate a complete BOOLE program for evaluating that formula.

`--help`
Display a brief command line help and terminate.

`-i` *`FILE`*
Read input from the designated file instead of standard input. The input must be in *ebf* format (c.f. Section B.5.1).

`-o` *`FILE`*
Write output to the designated file instead of standard output.

`-q`
Generate QUBOS [2] format instead of BOOLE format output.

B.3 Qst: Prenexing Tool

QST[3] is an implementation of the prenexing strategies presented in Chapter 3 of this work. It was implemented in Haskell 98 [34] and is available as a command line tool.

B.3.1 Command Line Interface

Synopsis

`qst [`*`OPTION`*`...]`

Options

`-c`
`--comments`
Enrich output with useful meta information about the translation process, embedded as comments. The format of this output has not been formally specified, but should be pretty self-explanatory.

`-d` *`STRUCTURE`*
`--dump=`*`STRUCTURE`*
Dump the designated intermediate compiler structure and terminate. See Section B.3.2 for a description of available structures.

[3] QST is an acronym for Quantifier Shifting Tool.

`-f` *FORMAT*
`--format=`*FORMAT*
Write output using the designated format, which must be either BOOLE or QDIMACS. See Section B.5 for an explanation of these formats.

`-h`
`--help`
Display a brief command line help and terminate.

`-i` *FILE*
`--input=`*FILE*
Read input from the designated file instead of standard input. The input must be in BOOLE format (c.f. Section B.5).

`-l`
`--fewlabels`
Usually, QST introduces at most one label for each simple formula. With this switch, QST tries to reduce the total number of labels by taking commutativity into account. This is quite costly in terms of compilation speed.

`-n`
`--invert`
Reserved.

`-o` *FILE*
`--output=`*FILE*
Write output to the designated file instead of standard output.

`-r`
`--norename`
Usually, QST performs a complete renaming of all variables in its input formula to obtain a cleansed formula. This switch turns renaming off, which will probably cause problems with uncleansed formulas. It is occasionally useful for inspecting the behavior of the translation and for debugging.

`-s` *STRATEGY*
`--strategy=`*STRATEGY*
Apply the designated merging strategy. Available merging strategies are: adeu, aued, edau, euad, u, d, bhmmax, lcsmax, jwmax, bhmmin, lcsmin, jwmin, drbf and drdf, as described in Section 3.3.

`-u`
`--fusion`
Use quantifier fusion to simplify formula.

`-v`
`--version`
Display version information and terminate.

B.3.2 Compiler Structures

btree The intermediate quantifier forest in DOT [25] format.

prefix The intermediate quantifier prefix in DOT [25] format.

mstat Statistical information that is used in the calculation of heuristic values. Intended for debugging.

final The compiled formula.

B.4 Mkncf: Nested Counterfactual Generator

MKNCF[4] is a simple generator for random synthetic nested counterfactual. Formulas are generated in the *EBF* format, which is described in Section B.5.1. The model of these formulas is explained is Section 4.2. The software was implemented in Python [51] and is available as a command line tool.

Synopsis

`mkncf` *VAR CLS LPC DEP*

Four arguments are required, which have the following meaning: *VAR* is the maximum number of distinct variables used; *CLS* is the number of clauses in the theory; *LPC* is the number of literals per theory clause; *DEP* is the nesting depth. For details, please refer to Section 4.2.

B.5 Formats

Boole This is the formula format used by the BOOLE QBF calculator, which is included in the BDDLIB software package [5]. Unlike the standard QDIMACS format, it supports structured QBFs. As an output format for QST, it is especially useful for inspecting the behavior of the translation in combination with the `-r` switch. For details about the format, see BDDLIB. As an undocumented feature, BOOLE offers support for line comments that are introduced by the ASCII sequence `--`. The `-c` option of QST takes advantage of this feature to embed useful meta information into formulas. On the other hand, our tools disallow the use of the symbol > within identifiers, for the sake of simplicity. It is reserved for use in new *ebf* operators.

EBF This is a modification of the BOOLE format that provides support for right nested counterfactuals and explicitly allows for embedded comments. It is mostly compatible with the traditional BOOLE format. A specification is given in Section B.5.1.

QDimacs The QDIMACS [1] format, an extension of the DIMACS format for propositional formulas, has recently gained significance as a standard format for QBFs in PCNF. Filtering QDIMACS output through various lightweight conversion tools[5] effectively augments the range of available output formats.

[4] MKNCF is an acronym for Make Nested Counterfactual.

[5] Such filters were available on the WWW for some time. Unfortunately, it seems that they have recently been removed. These filters, which had been distributed without any copyright information, are included in the TRAQUASTO package.

Operator	Priority	Semantics
x `>` y	6	counterfactual implication
x `!>` y	6	negated counterfactual implication
`exists [ ... ] ( x )`	5	exist. quantification
`forall [ ... ] ( x )`	5	univ. quantification
`subst [ ... ] ( x )`	5	substitution of variables[6]
`!` x, `~` x	5	negation
x `=` y	4	equivalence
x `&` y, x `*` y	3	conjunction
x `|` y, x `+` y	2	disjunction
x `?` y `:` z	1	conditional

Table B.1: Priority and semantics of operators.

B.5.1 Extended Boole Format

The lexical structure of EBF is given by the following EBNF grammar, where the top level structure is an *ebfstring*.

ebfstring = { *lexeme* | *whitespace* }
lexeme = *grouping* | *id* | *keyword* | *opsym*
grouping = `(` | `)` | `[` | `]`
id = *idchar* {*idchar* | *digit*}
idchar = `A` | ... | `Z` | `a` | ... | `z` | `_` | `$` | `%` | `#` | `@` | `/` | `\` | `<`
digit = `0` | ... | `9`
keyword = `exists` | `forall` | `ncfact` | `subst`
opsym = `>` | `!>` | `|` | `+` | `&` | `*` | `!` | `~` | `=` | `:=` | `?` | `:` | `0` | `1`
whitespace = *whitestuff* {*whitestuff*}
whitestuff = *comment* | *space* | *newline* | *tab*
comment = `--` *linechar* *newline*

A *linechar* is any character other than *newline*.

The following EBNF grammar specifies the syntax of EBF expressions. The usual top level structure is a *ncf* expression.

ncf = `ncfact [` { *qbf* }] `(` *qbf* *opncf* { *qbf* *opncf* } *qbf* `)`
qbf = `0` | `1` | *id* | *qsym* `[` { *id* } `]` `(` *qbf* `)` | *op1* *qbf* | *qbf* *op2* *qbf*
| *qbf* `?` *qbf* `:` *qbf* | `subst [` { *id* `:=` *qbf* } `]` `(` *qbf* `)`
qsym = `exists` | `forall`
op1 = `!` | `~`
op2 = `|` | `+` | `&` | `*` | `=`
opncf = `>` | `!>`

For a description of operator priorities and semantics, please refer to Table B.1. All operators are right associative, as far as a notion of associativity is applicable. In particular, a term like `a?b:c?d:e` is semantically equivalent to `a?b:(c?d:e)`, and *not* `(a?b:c)?d:e`.

[6] Currently not supported by QST.

Bibliography

[1] Anonymous. Quantified Boolean Formulas Satisfiability—Suggested Format. Available in electronic form at `http://www.qbflib.org/Draft/-qDimacs.ps.gz`, July 2001.

[2] 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*, volume 2517 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 2002.

[3] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[4] M. Baaz, U. Egly, and A. Leitsch. Normal Form Transformations. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 5, pages 273–333. The MIT Press, 2001.

[5] BDDLib: A BDD Library with Extensions for Sequential Verification. Software available at `http://www.cs.cmu.edu/~modelcheck/bdd.html`.

[6] R. Bosch. Theorembeweisen für QBF und die Anwendung zum Planen. Master's thesis, Universität Ulm, 1998.

[7] H. Kleine Büning, M. Karpinski, and A. Flögel. Resolution for Quantified Boolean Formulas. *Information and Computation*, 117(1):12–18, 1995.

[8] M. Cadoli, A. Giovanardi, and M. Schaerf. An Algorithm to Evaluate Quantified Boolean Formulae. In *Proceedings of the 15th National Conference on Artificial Intelligence*, pages 262–267. AAAI Press/The MIT Press, 1998.

[9] 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.

[10] J. M. Chambers. *Data for Models*. Wadsworth & Brooks/Cole, 1992.

[11] W. S. Cleveland. *Visualizing Data*. Summit Press, New Jersey, 1993.

[12] Intel Corporation. IA-PC HPET (High Precision Event Timers) Specification. Available in electronic form at `ftp://download.intel.com/labs/-platcomp/hpet/download/hpetspec_1.pdf`, June 2004.

[13] M. Davis, G. Logeman, and D. Loveland. A Machine Program for Theorem-Proving. *Communications of the Association for Computing Machinery*, 5:394–397, 1962.

[14] M. Davis and H. Putnam. Computational Methods in the Propositional Calculus. Technical report, Rensselaer Polytechnic Institute, 1958. unpulished report.

[15] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[16] J. Delgrande, T. Schaub, H. Tompits, and S. Woltran. On Computing Solutions to Belief Change Scenarios. *Journal of Logic and Computation*, 14(6):801–826, 2004.

[17] E. Eder. *Relative Complexities of First Order Calculi*. Verlag Vieweg, Wiesbaden, 1992.

[18] U. Egly, T. Eiter, H. Tompits, and S. Woltran. Solving Advanced Reasoning Tasks Using Quantified Boolean Formulas. In *Proceedings of the 17th Conference on Artificial Intelligence and of the 12th Conference on Innovative Applications of Artificial Intelligence*, pages 417–422, Menlo Park, July 2000. AAAI Press.

[19] U. Egly, R. Pichler, and S. Woltran. On Deciding Subsumption Problems. *Annals of Mathematics and Artificial Intelligence*, 43(1–4):255–294, 2005.

[20] 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*, volume 2919 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2004.

[21] 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.

[22] T. Eiter, V. Klotz, H. Tompits, and S. Woltran. Modal Nonmonotonic Logics Revisited: Efficient Encodings for the Basic Reasoning Tasks. In U. Egly and Ch. Fermüller, editors, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference (TABLEAUX-02)*, volume 2381 of *LNCS*, pages 100–114. Springer-Verlag, 2002.

[23] R. Feldmann, B. Monien, and S. Schamberger. A Distributed Algorithm to Evaluate Quantified Boolean Formulas. In *Proceedings of the 17th National Conference on Artificial Intelligence*, pages 285–290. AAAI Press / The MIT Press, 2000.

[24] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, New York, 1990.

[25] E. Gansner, E. Koutsofios, and S. North. Drawing Graphs with dot. Technical report, AT&T Bell Laboratories, Murray Hill, February 2002.

[26] I. Gent and A. G. D. Rowley. Encoding connect-4 using quantified boolean formulae. Technical Report APES-68-2003, APES Research Group, July 2003. Available in electronic form at `http://www.dcs.st-and.ac.uk/~apes/-apesreports.html`.

[27] I. P. Gent, P. Nightingale, and A. G. D. Rowley. Encoding quantified csps as quantified boolean formulae. Technical Report APES-79-2004, APES Research Group, February 2004. Available in electronic form at `from http://-www.dcs.st-and.ac.uk/~apes/apesreports.html`.

[28] I. P. Gent and T. Walsh. Beyond NP: The QSAT Phase Transition. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99)*, pages 648–653. AAAI Press/MIT Press, 1999.

[29] M. I. Ginsberg and M. L. Ginsberg. Counterfactuals. *Journal of Artifical Intelligence*, 30(1):35–80, 1986.

[30] 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*, volume 2175 of *Lecture Notes in Computer Science*, pages 111–122. Springer, 2001.

[31] E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for Quantified Boolean Logic Satisfiability. In B. Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 275–281. Morgan Kaufmann Publishers, Inc., 2001.

[32] E. Giunchiglia, M. Narizzano, and A. Tacchella. QUBE: A System for Deciding Quantified Boolean Formulas Satisfiability. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings*, volume 2083 of *Lecture Notes in Computer Science*, pages 364–369. Springer, 2001.

[33] E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for Quantified Boolean Logic Satisfiability. In *Proceedings of the 18th National Conference on Artificial Intelligence*, pages 649–654. AAAI Press / The MIT Press, 2002.

[34] S. P. Jones and J. Hughes, editors. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, April 2003.

[35] H. Kleine-Büning and T. Lettmann. *Propositional Logic: Deduction and Algorithms*. Cambridge University Press, 1999.

[36] D. Leijen. *Parsec, a fast Combinator Parser*. Dept. of Computer Science, University of Utrecht, October 2001. Available in electronic form at `http://www.cs.uu.nl/~daan/download/parsec/parsec.pdf`.

[37] R. Letz. Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In U. Egly and C. G. Fermüller, editors, *Proceedings of the 11th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, International Conference*, volume 2381 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2002.

[38] A. R. Meyer and L. J. Stockmeyer. The Equivalence Problem for Regular Expressions with Squaring requires Exponential Time. In *Proceedings 13th Annual IEEE Symposium on Switching and Automata Theory*, pages 125–129, 1972.

[39] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[40] D. A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

[41] J. Rintanen. Constructing Conditional Plans by a Theorem-Prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.

[42] J. Rintanen. Improvements to the Evaluation of Quantified Boolean Formulae. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 1192–1197. Morgan Kaufmann Publishers, Inc., 1999.

[43] S. Schamberger. Ein paralleler Algorithmus zum Lösen von Quantifizierten Boolschen Formeln. Master's thesis, Universität Gesamthochschule Paderborn, January 2000.

[44] L. J. Stockmeyer. The Polynomial-Time Hierarchy. *Journal of Theoretical Computer Science*, 3(1):1–22, 1977.

[45] The GHC Team. *GHC Documentation*. Available in electronic form at `http://www.haskell.org/ghc/documentation.html`.

[46] G. S. Tseitin. On the Complexity of Derivation in Propositional Calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967–1970*, pages 466–483. Springer, Berlin, Heidelberg, 1983.

[47] J. W. Tukey. *Exploratory Data Analysis*. Addison-Wesley series in Behavioral Science. Addison-Wesley, Reading, Massachusetts, 1977.

[48] H. Turner. Polynomial-Length Planning Spans the Polynomial Hierarchy. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Logics in Artificial Intelligence, European Conference, JELIA 2002, Proceedings*, volume 2424 of *LNCS*, pages 111–124. Springer-Verlag, 2002.

[49] P. van Emde Boas. Machine Models and Simulations. In van Leeuwen [50].

[50] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*. Elsevier Science Publishers B.V., Amsterdam, 1990.

[51] G. van Rossum. *Python Reference Manual*. PythonLabs, May 2004. Available in electronic form at `http://docs.python.org/ref/ref.html`.

[52] G. Wechsung. *Vorlesungen zur Komplexitätstheorie*. B. G. Teubner, Stuttgart, 2000.

[53] J. Wielemaker. SWI-Prolog Reference Manual. Available in electronic form at `http://www.swi.psy.uva.nl/projects/SWI-Prolog/Manual/`.

[54] C. Wrathall. Complete Sets and the Polynomial-Time Hierarchy. *Journal of Theoretical Computer Science*, 3(1):23–33, October 1976.

[55] L. Zhang. SAT-Solving: From Davis-Putnam to Zchaff and Beyond. Available in electronic form at `http://research.microsoft.com/users/lintaoz/-SATSolving/sat_course[123].pdf`.