KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

A Formal Language and Tool for QBF Family Definitions

Noel Arteche Echeverría

Thesis submitted for the degree of Master of Science in Engineering: Computer Science

Thesis supervisors:
Prof. dr. M. Denecker
Prof. dr. ir. G. Janssens
Prof. dr. M. Hermo

Assessors:
Matthias van der Hallen
Dr. J.T. Mühlberg

Mentor:
Matthias van der Hallen

Academic year 2019 – 2020

Preface

I would like to thank Marc Denecker for accepting to promote this thesis well before meeting me in person.

To Matthias van der Hallen, for his wise direction, constant supervision, comments and valuable insights during this time.

To my co-promoter back in Spain, Montserrat Hermo, for introducing me to complexity theory and welcoming me to work on it with her.

Finally, my sincere gratitude goes to Zoë for guiding me in Leuven and without whom the frustration would have been a lot more difficult to overcome.

Noel Arteche Echeverría

Contents

Abstract

After the generalized success of SAT solvers in recent times, interest has grown in QBF solvers, programs capable of solving the True Quantified Boolean Formula (TQBF) problem, a **PSPACE**-complete problem with practical applications in solving what lies beyond **NP**.

The research on QBF solvers is tightly related to proof complexity, where the length of proofs in different proof systems capable of dealing with QBF is studied. In the proof complexity literature, parameterized sets of formulae or *formula families* are used to show exponential lower bounds, separations and other proof-theoretical results on proof systems. Often, these formulae are used as benchmarks on QBF solvers to test whether the solvers built on top of the proof systems find the same difficulties proven in theory. Besides, given the growing variety of solving techniques for QBF, testing hard-to-solve formulae on different solvers is an interesting approach to empirical proof complexity research.

In this work, we propose a formal language to write definitions of classes of Quantified Boolean Formulae (QBF) in terms of —potentially— any type of parameters. A class of formulae provides an encoding in logic terms of some computational problem —often one belonging to **PSPACE**. These definitions of families of formulae usually depend on some parameters determining the size, structure, alternation patterns of quantifiers and complexity of the described formulae. These parameters and their relation to the structure of formulae can be easily encoded in this language. Additionally, we present QBDef, a computer tool capable of parsing these definitions and outputting the formulae in either the QCIR or QDIMACS formats to serve as input to a QBF solver. This aims to be both a framework and a tool for future empirical research on these topics.

Chapter 1

Introduction

Proof complexity is the field of mathematics and computer science studying the length of proofs in different proof systems. In the same way that computational complexity theory stems from computability theory, aiming at studying the tractability of computable problems, proof complexity goes beyond proof theory, focusing not on determining what can be proven, but on showing whether proofs are intractably long.

For a long time, complexity theory has focused on SAT, the satisfiability problem for Boolean formulae, while proof complexity has been working around the complementary problem, UNSAT, trying to give short proofs of universal statements like unsatisfiability. Surprisingly, today the SAT problem is somewhat under control. Despite the strong belief in the $\mathbf{P} \neq \mathbf{NP}$ conjecture, state-of-the-art tools known as SAT solvers employ very smart rules, heuristics and algorithmic techniques to check satisfiability very fast for most real-world instances. As a result, for some time now researchers have started looking beyond SAT and the domain of **NP** problems to focus on a bigger complexity class: **PSPACE**, and its canonical problem, an extension of SAT known as TQBF: the *True Quantified Boolean Formula* problem.

In the TQBF problem, we no longer look for a satisfying assignment for a Boolean formula φ. Now we want to check that a Boolean formula is satisfied for every variable configuration determined by some quantifiers. We are not only interested in knowing, say, if there exist some $x, y, z \in \{0, 1\}$ such that some $\varphi(x, y, z)$ is true, but whether, for instance, there *exists* some x such that *for all* y and *for all* z, the formula φ is satisfied. These are called *Quantified Boolean Formulae* (QBF).

This problem is significantly more complex than SAT and has forced the current research to go beyond the techniques and proof systems known so far. This quest for new proof systems and efficient QBF solvers is now in full expansion: new tools are in development and an active community is investigating their strengths, weaknesses and performance. In parallel, the theoretical research proceeds in the field of proof complexity by proving exponential lower bounds for the existing proof systems: showing that even in this domain too, there are limits and hurdles.

In this area of proof complexity, researchers often work using sets of parameterized QBF, often called *formula families.* In essence, researchers prove that a certain formula family is difficult for a given proof system: as the size of the parameters increase, the length of proofs in a certain proof system grows super-polynomially. And, on the practical side of things, researchers verify how fast these formulae are solved by QBF solvers, or whether whenever short proofs exist, solvers can find them quickly (a problem known as automatizability).

The formula families existent in the literature are many and varied, and it is often the case that we want to generate instances of them to serve as inputs to QBF solvers. This might be either to create benchmark problems, to guide particular research of a specific family or, more generally, to find difficult formulae for a given solver. Until now, the main approach to this problem was to write a script in some programming language, capable of printing files containing the specific formulae for the values of the parameters. This approach is feasible if the definition is fixed and well-known, if we are only interested in that single family and we do not plan to make modifications on it or if the translation from the natural-language mathematical definition to a valid format (like QCIR or QDIMACS) is straightforward. But, as soon as things begin to complicate, more general computer tools are needed for this purpose.

Scripts like that work directly with input/output writing statements and strings written in the formats accepted by solvers, which makes them too purpose-specific, very unreadable and non-reusable. Besides, given the current lack of conversion tools between formats, a script to output formulae in one format might be useless if we want to test the same family on a different solver. We believe that research of the QBF domain should be made easier, more convenient and flexible. Users should be able to write formulae with multiple parameters, with varied data-types; they should be able to generate random formulae; and playfully come up with new formulae in a convenient editor that can build them easily, without worrying about format, input and output files and other lower-level details holding them from easily playing with the formulae on a higher reasoning level.

Inspired by this, some attempts at developing modelling tools for QBF already exist. Most notably, Matthias van der Hallen's SOGrounder (see [36]). However, despite a certain overlap with the work we will be presenting here, SOGrounder is designed for a somewhat different purpose, as it aims at conveniently modelling problems in second-order logic to then perform tasks like grounding and model checking on those theories. Despite the convenience provided by SOGrounder for actual problem-solving, that tool cannot encode parameterized formula family definitions. For instance, the game of chess can be encoded in QBF and then written in SOGrounder, but a generalization of the game to an $n \times n$ board cannot, as this would imply defining the structure of quantifier prefixes in terms of this parameter n. Besides, while the goal of tools like SOGrounder is mainly modelling, QBDef is aimed more specifically at proof complexity, where formula families do not necessarily model intuitive but simply difficult to solve problems, whatever their nature, meaning or applicability.

In this work, such a tool for the proof complexity domain is presented: QBDef. It gets formula family definitions written in a formal language designed for this tool, capable of expressing potentially any computable formula family, and given specific values for the parameters outputs files in formats accepted by QBF solvers.

Outline of the thesis

This thesis tackles the problem of developing a tool for dealing with QBF families by first looking at a wide variety of existing definitions to note on required language features, then designing a formal language capable of expressing all the definitions encountered and, finally, developing a computer tool that can parse this language and output specific instances of the formulae for given values of the parameters.

Before going into the actual content, the next section introduces the notation and terminology used across this work, defines concepts and revises some common knowledge in complexity theory to serve as background to the reader.

In Chapter 2, the idea of formula families and their purpose is explored at length, looking at what formula families look like and drawing on the existing literature to present a brief yet representative selection of formula families to use as a reference for the rest of the work.

In Chapter 3, we design a formal language to write such definitions. We present its prominent features by means of examples and encode in it some of the formula families defined in Chapter 2. A complete formal grammar of this language is given in Appendix A.

In Chapter 4, we take a bit of a detour to dive into a case study of a specific problem that can be represented as a formula family and encoded in the language: the Generalized Geography game. We present the game and its generalized version on a graph, provide a polynomial-time algorithm for the particular case of directed acyclic graphs and describe how to model the game into QBF and then into our language.

In Chapter 5, we finally present QBDef, the computer tool working with the formal language described in Chapter 2. We present the tool and its features, discuss some minor technical details of the implementation and look into how formulae are built and encoded inside the system. Besides, we briefly discuss some sanity tests for the tool and give a taste of how QBDef could be used in proof complexity research by presenting an exponential separation given by the Chen Formulae of Type 2 as generated by our tool.

Finally, Chapter 6 concludes this thesis and points at future lines of work that would be interesting to follow.

Conceptual and notational preliminaries

We denote by $\mathbb{N}$ the set of natural numbers and we write $\mathbb{N}^*$ for $\mathbb{N} \setminus \{0\}$. For any $n \in \mathbb{N}^*$, $[n]$ denotes the set $\{1, \dots, n\}$ containing all positive natural numbers between 1 and n. Given $a, b \in \mathbb{N}$, $a \leq b$, $[a, b]_{\mathbb{N}}$ denotes the closed interval $[a, b]$ over the natural numbers (i.e. $[a, b]_{\mathbb{N}} = [a, b] \cap \mathbb{N}$).

(Quantified) Boolean formulae

Boolean variables are variables that can take value either 0 or 1. A *Boolean function* φ is a function that takes n Boolean inputs and produces one Boolean output, such that $\varphi : \{0,1\}^n \to \{0,1\}$. The inside of a Boolean function is often expressed as a *propositional formula.* A propositional formula is formed by *literals* (possibly negated Boolean variables) operated with negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), exclusive disjunction ($\oplus$)[1], implication ($\to$) and double implication ($\leftrightarrow$). Whenever a propositional formula φ is written as a conjunction of clauses, where these clauses are all disjunctions of literals, we say φ is written in *Conjunctive Normal Form* (CNF).

We employ the usual notation $\exists$ and $\forall$ to denote existential and universal quantification over Boolean variables. If $\varphi : \{0,1\}^n \to \{0,1\}$ is a Boolean function written as a propositional formula and we quantify the variables appearing in it, we get a *Quantified Boolean Formula* (QBF). If the variables are all quantified at the front, such as

$$\Phi = Q_1 x_1 \dots Q_n x_n : \varphi(x_1, \dots, x_n)$$

where each Q_i is either $\exists$ of $\forall$, we say that the QBF Φ is in *prenex* form and $Q_1 x_1 \dots Q_n x_n$ is called the *quantifier prefix.* If the *matrix*, $\varphi(x_1, \dots, x_n)$, is written in CNF, we say Φ is in *Prenex Conjunctive Normal Form* (PCNF). If, on the contrary, the matrix is not normalised, we sometimes say Φ is a quantified Boolean *circuit* (QBC).

The *satisfiability problem* is the problem of deciding whether a given Boolean formula φ is satisfiable: there exists an assignment $(b_1, \dots, b_n) \in \{0,1\}^n$ such that $\varphi(b_1, \dots, b_n) = 1$. We denote this problem by SAT: the set of binary strings encoding satisfiable formulae. A computer tool that computes SAT is called a SAT *solver.*

On the other hand, the problem of deciding whether a quantified Boolean formula is satisfiable is called the *True Quantified Boolean Formula* problem. We denote this problem by TQBF: the set of binary strings encoding satisfiable QBF. A computer tool that computes TQBF is called a QBF *solver.*

Complexity theory

We assume basic familiarity with formal language theory and basic computational complexity theory, but we review the main definitions and results that appear through this text. For a more consistent introduction to these topics see [5, 30].

The class **P** is the class of problems computable in polynomial time in the size of the input.

The class **NP** is the set of languages (the set of computational problems) that have polynomial-size certificates that can be deterministically verified in polynomial time. A problem L is **NP**-hard if all the problems in **NP** can be reduced to it in polynomial time. In other words, for every problem $L' \in \mathbf{NP}$, there exists a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$, $x \in L'$ if and only if $f(x) \in L$. If a problem is in **NP** and it is **NP**-hard, then it is **NP**-*complete.*

The satisfiability problem SAT is an **NP**-complete problem, a result known as the Cook-Levin theorem. As a result, any other **NP** problem (Independent Set,

[1]In this work, XOR is considered exclusively as an infix binary operator.

CLIQUE, VERTEX COVER, SUBSET SUM, SUDOKU, GRAPH k-COLORABILITY, etc.) can be solved by encoding them in a propositional formula and then checking for satisfiability on a SAT solver. Besides, $\mathbf{P} = \mathbf{NP}$ if and only if $\mathsf{SAT} \in \mathbf{P}$. This is known as the $\mathbf{P} =_? \mathbf{NP}$ question. It is often conjectured that $\mathbf{P} \neq \mathbf{NP}$.

On the other hand, TQBF is $\mathbf{NP}$-hard but it is likely not to be $\mathbf{NP}$-complete. This is because $\mathsf{SAT} \subseteq \mathsf{TQBF}$ (every propositional formula is an implicitly existentially quantified formula), so every $\mathbf{NP}$ problem can be reduced to TQBF, but we do not think short certificates exist for every quantified formula, so likely $\mathsf{TQBF} \notin \mathbf{NP}$.

The class **coNP** is the set of complementaries of **NP** languages:

$$\mathbf{coNP} = \{L \subseteq \{0,1\}^* : \overline{L} \in \mathbf{NP}\}$$

The class **coNP** contains the "negative" version of **NP**-problems, such as UNSAT. If $\mathbf{NP} \neq \mathbf{coNP}$, then $\mathbf{P} \neq \mathbf{NP}$. It is conjectured that $\mathbf{NP} \neq \mathbf{coNP}$.

The definition of **NP** (existence of a polynomial-size certificate) can be generalized. The class Σ_2^p is the class of languages L such that there exists a polynomial-time verifier Turing machine V and two polynomials p and q such that for all $x \in \{0,1\}^*$, $x \in L$ if and only if

$$\exists u_1 \in \{0,1\}^{p(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)} : V(x, u_1, u_2) = 1$$

The complementary of Σ_2^p is Π_2^p. If we generalize the previous definition, Σ_i^p is the class for which there exist i polynomials such that $x \in L$ if and only if

$$\exists u_1 \in \{0,1\}^{p_1(|x|)} \forall u_2 \in \{0,1\}^{p_2(|x|)} \dots Q u_i \in \{0,1\}^{p_i(|x|)} : V(x, u_1, u_2, \dots, u_i) = 1$$

where Q is $\exists$ or $\forall$ depending on whether i is odd or even. According to this generalization, $\mathbf{NP} = \Sigma_1^p$ and $\mathbf{coNP} = \Pi_1^p$, while $\mathbf{P} = \Sigma_0^p = \Pi_0^p$. The union of all the Σ_i^p classes is called the *Polynomial Hierarchy*, **PH**:

$$\mathbf{PH} = \bigcup_{i \in \mathbb{N}} \Sigma_i^p$$

It is conjectured that no complete problems exist for **PH**. If one existed, then it would be in some level i, and then the hierarchy would *collapse* to the i-th level: every other problem could be reduced to one in that level. In particular, if it collapses to the first level, then $\mathbf{P} = \mathbf{NP}$. Since it is conjectured that the hierarchy does not collapse, there are probably no complete problems for it.

However, each level of the hierarchy does have complete problems. These are bounded versions of the TQBF problem. The complete problem for Σ_i^p is the problem $\Sigma_i\mathsf{SAT}$: quantified formulae starting with $\exists$ and then alternating i times:

$$\exists x_1 \forall x_2 \exists x_3 \dots Q x_i : \varphi(x_1, \dots, x_i)$$

On the other hand, $\Pi_i\mathsf{SAT}$ is a complete problem for Π_i^p. These are formulae of the form:

$$\forall x_1 \exists x_2 \forall x_3 \dots Q x_i : \varphi(x_1, \cdots x_i)$$

If we do not impose restrictions on the bounded alternation of quantifiers, we get the TQBF problem, belonging to a class beyond **NP** and the Polynomial Hierarchy: **PSPACE**, the class of problems computable in polynomial space. The TQBF problem is **PSPACE**-complete, which means that TQBF $\in$ **PSPACE** (this is straightforward to see) and every other problem in **PSPACE** can be polynomial-time reduced to it (see [5, 27, 30] for a proof). It is conjectured that **NP** $\neq$ **PSPACE**.

It should be noted that despite TQBF is **PSPACE**-complete, this does not mean that QBF can only model those problems. It can model harder problems too; just probably not in polynomial size. The same remark applies for SAT.

The last two classes we mention are circuit complexity classes. The class **P**/poly is the class of problems solvable by polynomial-size Boolean circuits (although it can also be defined in terms of Turing machines that receive advice). Another interesting class of circuits is **BH**, the *Boolean Hierarchy.* This is the class of Boolean circuits over **NP** predicates. In particular, we remark the class **DP**, containing the intersection of problems belonging to **NP** and **coNP**. As we will later see, the Chromatic Number Problem is in this class.

Proof systems, solvers and formats

For a consistent introduction to proof complexity, see [28, 29].

Formally, a *proof system* for a language L is a binary relation $P(\tau, \rho)$ over L satisfying the following three conditions:

1. (*Soundness*) $\exists \rho : P(\tau, \rho) \Rightarrow \tau \in L$
2. (*Completeness*) $\tau \in L \Rightarrow \exists \rho : P(\tau, \rho)$
3. (*p-verifiability*) $P(\tau, \rho)$ can be decided in polynomial time

We read $P(\tau, \rho)$ as τ *is proven in system* P *by proof* ρ. Whenever a proof system can produce short (polynomial-size) proofs for every string in the language, we say it is *polynomially bounded* or p-bounded. More formally, a proof system P is *polynomially bounded* or *p-bounded* if and only if there exists $k \in \mathbb{N}^*$ such that for all $\tau, \rho \in \{0, 1\}^*$

$$P(\tau, \rho) \Rightarrow \exists \rho' : |\rho'| \leq (|\tau| + k)^k \text{ and } P(\tau, \rho')$$

Proof complexity is the field studying the length of proofs in different proof systems. It is conjectured that the language UNSAT does not have a p-bounded proof system and, therefore, that there are not always short proofs of unsatifiability. This amounts to **NP** $\neq$ **coNP**, the central conjecture of proof complexity.

An example of a proof system for the language UNSAT is *Resolution.* A *Resolution refutation* of a CNF propositional formula φ is a sequence of clauses

$$C_1, \dots, C_k, C_{k+1}, \dots, C_s$$

such that the first $C_1, \dots, C_k$ are the clauses in φ and for every $k < i \leq s$,

$$C_i = A \vee B$$

where $(A \vee x)$ and $(B \vee \neg x)$ are clauses that have already been derived.

The process ends when both the clauses x and $\neg x$ have been derived for some variable x (i.e. we have a contradiction). The size of the proof is the number of clauses, s.

The Resolution proof system can easily be modified to get a proof system for TQBF. Such an example is Q-Resolution, introduced in 1995 in [12]. The Q-Resolution proof system behaves exactly like Resolution but with an additional rule to derive new clauses. If $C = A \vee x$ is a derived clause, x appears in C and x is the outermost universally quantified variable, then A can be derived.

Amongst other things, proof complexity researchers prove that proof systems have exponential lower bounds: there are formulae that cannot be solved in polynomial size in that system. For example, in 1985 Haken proved that Resolution has an exponential lower bound, and because this is a particular case of Q-Resolution, this system has an exponential lower bound too.

SAT solvers and QBF solvers are computer tools that are built on top of these proof systems. They use heuristics to decide what rules to apply and when and are able to output proofs of satisfiability or unsatisfiability in their systems.

These tools get their input formulae written in a specific format. For SAT solvers this format is usually DIMACS, where formulae are written in CNF. An extension of this format is QDIMACS, allowing for PCNF QBF. QDIMACS is the format used by popular QBF solvers like DepQBF. On the other hand, if we are interested in writing quantified Boolean circuits instead of PCNF formulae, we can use the QCIR format, supported by solvers like QuAbS. Besides, the QCIR format, as presented in [25], also allows for the definition of non-prenex formulae, but this is generally not supported by current solvers.

Chapter 2

Formula Families

The purpose of the formal language and tool we are presenting in this work is to formally capture definitions of QBF families. But what are formula families? What do they look like? What are they used for? What role do formula families play in the field of proof complexity?

In this chapter, we answer the questions above and have a look at several formula families and their definitions. In Section 2.1, we introduce the concept and their uses as abstract mathematical objects and have a look at a famous example: the Pigeon Formulae used by Haken in 1985 to show the exponential lower bound of the Resolution proof system. This is an example where formulae are only existentially quantified. However, we aim to capture *quantified* formulae. In Section 2.2, we have a look at the additional elements we need to build quantified Boolean formulae and what types of formulae we can distinguish by looking at their syntactic properties. Finally, in Section 2.3, we offer a concise selection of relevant QBF families from the literature, paying special attention at the formal elements of their natural-language definitions as well as to the differences between them to distinguish what essential features should our language have to capture all of them effectively.

2.1 Formula families: what they are — and what for

In 1971, Stephen Cook proved that SAT is an **NP**-complete language. His famous result, today broadly known as the Cook-Levin theorem —what we could perhaps refer to as the Fundamental Theorem of Complexity Theory—, shows that every **NP**-language can be reduced to SAT instances and, therefore, that every **NP**-problem corresponds to a set of Boolean formulae. In the light of this theorem, much of the research around the $\mathbf{P} =_? \mathbf{NP}$ question was reduced to the domain of propositional formulae. In particular, instead of developing algorithms for each one fo the thousands of very different **NP**-problems, efforts have been directed to designing very fast SAT solvers. When presented with the challenge of solving an **NP**-problem, it is now customary to model the problem into SAT instances and then solve them via state-of-the-art SAT solvers, capable of quickly producing outputs in most situations.

When modelling a problem into logic, we describe what the formulae look like in terms of some parameters. For instance, if we are modelling a graph-based decision problem such as Independent Set, the input of the original problem would be a graph G and a natural number k. The abstract description of the formulae would be written in terms of these parameters, and when given specific values for them, we could obtain a concrete formula that could then be solved in a SAT solver. If $\mathcal{I}(G, k)$ is a propositional formula that is true if and only if G has an independent set of size k, then the set $\{\mathcal{I}(G, k) : G \text{ is a graph and } k \in \mathbb{N}^*\}$ is the formula family. The definition is just the formal description of what Boolean variables and clauses the formulae consist of.

Thus, we distinguish between the *formula family* and the formula family *definition*. The definition is the description of the family: it describes the syntactic and/or semantic features of the formulae in terms of the parameters, whilst the set with all the formulae described by the definition is the formula family, the actual mathematical object.

When looking at formula families this way, one may think the main purpose is to convert real-world problems into logic to then solve them through automated reasoning tools like SAT solvers. Though not completely false, that is not necessarily the main objective of formula families as presented from this conceptual perspective. When interested in solving real-world problems, we usually need more convenient tools for modelling of complex systems and knowledge representation. Formula families can theoretically capture any of those problems, but their definition as single propositional formulae is cumbersome and counterintuitive. Usually, we model these problems into theories, with several separate formulae and usually in a more convenient language in first-order logic with extensions, and the conversion to SAT-instances is hidden to the user. Therefore, if this approach is that inconvenient, what is the purpose of formula families?

The answer is proof complexity, a domain where formula families are extensively used in this manner because we are not that interested in solving actual problems, but, more pessimistically, in showing that certain formulae are hard to solve. Think of a proof system underlying some powerful SAT solver, such as Resolution. The system may be powerful in that it can prove the unsatisfiability of many formulae in polynomial size. However, proof complexity theorists are interested in showing that there exist formulae that cannot be succinctly proven in that system (as, otherwise, $\mathbf{NP} = \mathbf{coNP}$, which goes against the fundamental conjecture of proof complexity). The main task here is to find exponential lower bounds for specific proof systems: finding formula families that need super-polynomial-size proofs in a given system. And, naturally, when faced with this challenge, we are not interested in very complicated problems with very intricate encodings. We look for simple, succinct and easy to manipulate formulae, so that reasoning about proof-length lower bounds on them can be easy.

The most representative use of a formula family for this purpose is perhaps the encoding of the Pigeonhole Principle given by Cook and Reckhow in 1979 (see [16]) and later used by Haken to show an exponential lower bound for Resolution through the now-famous bottleneck method (see [20] for the original proof and [7] for a modern version by Beame and Pitassi, more recently reproduced in [5]). Haken showed that a certain propositional encoding of the Pigeonhole Principle needs exponential-size proofs in the Resolution proof system. Let us have a look at what this formula family looks like to get a first contact with them.

Example 2.1 (The Pigeonhole Formulae). The Pigeonhole Principle is a simple combinatorial property of sets, underlying many relevant theorems across mathematics: if we have m pigeons and n holes, and $m > n$, then there must be a hole with more than one pigeon. More formally, if $m, n \in \mathbb{N}^*$ and $m > n$, then there exists no bijection from the set $\{1, \dots, m\}$ to the set $\{1, \dots, n\}$.

Most undergraduate textbooks would say that the statement can be proven through straightforward induction, left as an exercise to the reader. But what happened if the proof system we worked with did not have induction? What if we tried to prove the Pigeonhole Principle in a system like Resolution? Very likely, the system would get lost in the locality of the formulae and it would need exponentially many steps to prove the tautology — it would assign some pigeons, see that it does not work, start over, and so on.

In order to study the Pigeonhole Principle under Resolution, we first need to encode the theorem into propositional logic formulae written in CNF. The following encoding is a simplified version of the original, summarised in [5]. We will work with the formula PHP^m_n stating that "there exists a bijection from $[m]$ to $[n]$". Clearly, if $m > n$, PHP^m_n is always false: $PHP^m_n \in \mathsf{UNSAT}$ and $\neg PHP^m_n \in \mathsf{TAUT}$.

Our formulae will be written in terms of variables $p_{i,j}$. Semantically, $p_{i,j}$ is true if and only if pigeon i is assigned to hole j.

These variables are arranged in two types of clauses:

1. Each pigeon is assigned to some hole:
$$(p_{i,1} \vee \cdots \vee p_{i,n})$$
2. The k-th hole doesn't get both the i-th and j-th pigeons:
$$(\neg p_{i,k} \vee \neg p_{j,k})$$

Thus, the complete Pigeonhole Formulae have the following form:

$$PHP^m_n = \bigwedge_{i \le m} (p_{i,1} \vee \cdots \vee p_{i,n}) \wedge \bigwedge_{\substack{i,j \le m \\ k \le n}} (\neg p_{i,k} \vee \neg p_{j,k})$$

The *Pigeonhole Formula Family* is the set

$$\mathcal{PH} = \{PHP^m_n : m, n \in \mathbb{N}^*\}$$

In particular, as shown in Haken's theorem, all formulae in the subfamily

$$\{PHP^n_{n-1} : n \in \mathbb{N} \text{ and } n \ge 2\} \subseteq \mathcal{PH}$$

need proofs of length at least $2^{n/20}$ in the Resolution proof system.

Clearly, the Pigeonhole Principle does not encode a very interesting property. After all, we could have convinced ourselves of the same fact by a simple proof by induction. But, again, that is fine for proof complexity: the propositional encoding is simple (it contains only two types of clauses and a small amount of variables, all with the same meaning), and thus it is simple to reason on them. In fact, the contemporary version of the proof of Haken's theorem, by Beame and Pitassi, makes a further simplification of these formulae by making them monotone in order to put their argument across.

It is therefore clear that when encoding computational problems into logic, the conceptual approach of formula families does not primarily cater for real-world problems or knowledge representation, but mainly for theoretical results on proof systems. This will become even more apparent when we have a look at different formula families from the literature and see that many of them do not even have intuitive meaning.

2.2 Formula families in the QBF domain

The example we just covered might seem strained in that it only needs propositional formulae without quantification (or implicitly existentially quantified). Since in this work we aim at covering quantified Boolean formulae, we must now look at further constructs that allow for quantification of variables. A formula family will now be capturing any **PSPACE** language, though in many cases it will not be obvious whether a certain formula family corresponds to a natural problem.

Before looking at actual formula family definitions from the literature, we will first discuss what shape can the basic components of a QBF take.

2.2.1 The matrix: CNF versus circuits

The matrix of a QBF is the propositional body, the formula written from propositional variables and connectives. For representing these, we have the same distinctions that we would have when working with SAT-formulae. Once variables have been defined, these are organised in either clauses or gates.

If we use clauses, then we aim at writing the formulae in clausal form, often in CNF. This is the case of the Pigeonhole Formulae, where no restriction is imposed on the number of disjuncts on a clause. Alternatively, the matrix can be written in Disjunctive Normal Form (DNF), a format that seems to have some interesting benefits for QBF solvers if properly balanced (see [32]).

Modelling using CNF is sometimes cumbersome, as we are forced to reformulate conditions that would otherwise seem natural. For this purpose, matrices can be written more freely in circuit format, where we allow any configuration of variables and propositional connectives and no normalised format restriction is imposed. In the domain of SAT solvers, most tools tend to accept inputs in CNF only, usually encoded in the DIMACS format, because it makes solving significantly easier. This has been no big issue because any propositional formula can be converted into normal form without increasing its size excessively via Tseitin transformations and other techniques (see [35, 31]).

Surprisingly though, in the domain of QBF solvers, the situation is somewhat different. Because formulae need to be verified for satisfiability under quantification, CNF seems to hide relevant and exploitable structural properties of formulae (see [25]). As a result, apart from the QDIMACS format to write PCNF QBF, there are solvers working under the QCIR format ([25]), that defines quantified Boolean *circuits*. It is thus very common to find family definitions in the literature directly written as circuits.

2.2.2 Quantification: prenex versus non-prenex formulae

Arguably, the main distinction between SAT solvers and QBF solvers is that now formulae are quantified. On this issue, two different approaches exist.

One could opt to quantify the formula at the beginning, preceding the matrix by a sequence of quantifiers and variables. Something like

$$\exists x \forall y \exists z \dots : \varphi(x, y, z, \dots)$$

where φ is the matrix and we assume no quantifiers to appear in there is known as *prenex form*, and it is the most usual way to write formulae.

Alternatively, if quantifiers appear in the matrix and not only in a quantifier prefix, we say that the formula is *non-prenex*. In the same way that circuits allow for a more natural encoding of problems, non-prenex is more intuitive than the prenex format in many situations. Unfortunately, until very recently not many efforts have been directed at studying proof complexity of non-prenex formulae. The QCIR format allows it, but solvers do not consistently support it at the time of writing this work. However, there have been some recent attempts at designing proof systems with nice properties beyond prenex forms (see [13]) and in [34] Tentrup used non-prenex formulae for some practical problem-solving.

2.2.3 Does format matter?

It would seem that non-prenex circuits are the ablest format, as they allow for the most natural representation. Though this is true, we shall recall that we are not particularly interested in easily writing complex formulae, but in having concise, easy-to-write formulae that are hard. As a result, in proof complexity, some format restriction can be useful to ease formula manipulation in proofs. In this regard, non-prenex forms are nowhere to be found in the current literature and we will not be seeing any examples in this chapter.

Besides, format does seem to have some influence in solving performance and thus in the complexity of the formulae. It has been shown that when converting matrices to CNF via a Tseitin transformation, adding the auxiliary variables at the end of the quantifier prefix can provoke an exponential blowup in the proofs. This exponential separation was theoretically proven in [9] and we have empirically observed it in the Chen Formulae of Type 2 as we will later explain in Section 5.4.

On the other hand, the potential additional hardness imposed by prenexing techniques (such as the ones discussed in [18]) have not been thoroughly studied theoretically, so it is not completely clear whether they have any influence complexity-wise.

All in all, it is safe to say that format does matter. On the one hand, restrictions are sometimes nice for proof complexity matters and, at the same time, it seems like the SAT and the TQBF problems are much more different than one might expect and that solving strategies may be different in nature and somewhat determined by format restriction.

For the rest of this work, we will focus mainly on prenex formulae, both in CNF and as circuits. The language we will present in Chapter 3 allows the definition of non-prenex QBF, but given the rudimentary —if not nonexistent— support in currently available solvers, that should be considered only an experimental feature and we will not discuss it at length.

2.3 Formula family definitions: a *tour d'horizon*

To get an idea of how formula families show up in the proof complexity literature, we now present a short collection of definitions. We start by having a look at the Chen Formulae of Type 1 and 2 (Definitions 2.1 and 2.2), two families that will let us have a closer look at the distinction between CNF and circuits. We then present the QParity formulae in their circuit version (Definition 2.3), a simple yet paradigmatic example of what a formula family definition looks like. In order to open the doors to modelling more complex **PSPACE** problems, we present the Chromatic Formulae (Definition 2.4), which take a graph as input, a very unusual yet interesting case in the literature. Finally, we present two more classes in CNF: the Janota Formulae (Definition 2.5), an interesting a approach to hard-to-prove contradictions based on two-player **PSPACE** games; and the KBKF formulae (Definition 2.6), a classic family in the early literature.

This selection of families is representative of both the definition styles as well as the tendencies to specific formats often used in the field of proof complexity of quantified Boolean formulae. Next chapter, we will use some of these formulae as examples of the expressive power of our formal language and in Chapter 4 we will study in detail an additional problem, not covered in this section: Generalized Geography.

2.3.1 Chen Formulae of Type 1

In [14], Chen showed that the QU-Resolution proof system is surprisingly more powerful than we might think. Although Haken's theorem also holds for this system, because for existentially quantified formulae QU-Resolution behaves just like Resolution, this system can find short proofs of formulae where intuitively more powerful systems cannot.

He showed this by introducing a formula family that we call the *Chen Formulae of Type 1.* These formulae have linear-size proofs in QU-Resolution and are defined in CNF. They constitute the paradigmatic example of a CNF definition by the explicit declaration of variables, quantifier blocks and conjunction of clauses.

Definition 2.1 (Chen Formulae of Type 1, [14])**.** Let $n \in \mathbb{N}^*$. For every $i \in \{0\} \cup [n]$, let X_i be the set of variables $\{x_{i,j,k} \mid j,k \in \{0,1\}\}$. Analogously, let $X'_i = \{x'_{i,j,k} \mid j,k \in \{0,1\}\}$, except for $i = 0$, when X'_i is not defined. Additionally, we have, for every $i \in \mathbb{N}^*$, a variable y_i. With these variables, we define P_n to be the quantifier prefix $\exists X_0 \exists X'_1 \forall y_1 \exists X_1 ... \exists X'_n \forall y_n \exists X_n$. Now, we define the following sets of clauses:

- $B = \{(\neg x_{0,j,k}) \mid j,k \in \{0,1\}\} \cup \{(x_{n,j,0} \vee x_{n,j,1}) \mid j \in \{0,1\}\}$
- For every $i \in [n]$ and every $j \in \{0,1\}$,
$$H_{i,j} = \{(\neg x'_{i,0,k} \vee \neg x'_{i,1,l} \vee x_{i-1,j,0} \vee x_{i-1,j,1}) \mid k,l \in \{0,1\}\}$$
- For every $i \in [n]$,
$$T_i = \{(\neg x_{i,0,k} \vee y_i \vee x'_{i,0,k} \mid k \in \{0,1\}\} \cup \{(\neg x_{i,1,k} \vee \neg y_i \vee x'_{i,1,k}) \mid k \in \{0,1\}\}$$

Then $\mathcal{C}_1(n) = P_n : \varphi$ is a *Chen Formula of Type 1*, where P_n is the prefix vector defined before and φ is the Boolean formula obtained from the conjunction of all the clauses in B, $H_{i,j}$ and T_i for every $i \in [n]$ and every $j \in \{0,1\}$.

2.3.2 Chen Formulae of Type 2

In that same paper ([14]), Chen also showed an exponential lower bound for a seemingly more powerful system he defined (Relaxing QU-Resolution). He showed that the *Chen Formulae of Type 2* need exponential-size proofs in that system.

Definition 2.2 (Chen Formulae of Type 2, [14])**.** Let $n \in \mathbb{N}^*$ and let P_n be the quantifier prefix $\exists x_1 \forall y_1 \dots \exists x_n \forall y_n$. Now, we consider Boolean circuits φ_n such that φ_n is true if and only if $\sum_{i=1}^{n}(x_i + y_i) \not\equiv n \pmod 3$. A QBF $\mathcal{C}_2(n) = P_n : \varphi_n$ is called a *Chen Formula of Type 2*.

In contrast with Definition 2.1, which provides a complete syntactic description of Type 1 formulae in CNF, Type 2 formulae are described in a much subtler way. The quantifier prefix is precisely described ($P_n = \exists x_1 \forall y_1 \dots \exists x_n \forall y_n$), but the formula's matrix is defined by an arithmetic property, a simple semantic condition, as that is all he needs to prove the exponential lower bound. As a result, it is not specified whether they should be built in CNF or as circuits.

However, if we try to write them in CNF without any additional variables, we will soon see that we need exponentially many clauses. We will now discuss how to build these Boolean formulae in a way that does not lead to a combinatorial explosion, which will invite us to use the more intuitive circuit representations.

We will first see the shortcomings of using CNF without any auxiliary variables.

Let n be a positive natural number, and let us consider the Boolean formulae φ_n, which we will simply refer to as φ. These are simply Boolean formulae over $2n$

variables, such that for all assignments $(x_1, y_1, \ldots, x_n, y_n)$ that make φ true, it must hold that

$$\sum_{i=1}^{n}(x_i + y_i) \not\equiv n \pmod 3$$

The question is: what formulae verify this property?

Let $S = \sum_{i=1}^{n}(x_i + y_i)$. It is clear that, since there are $2n$ terms in the sum, which can take value either 0 or 1, the total value of S must be between 0 and $2n$. Some values in the set $\{0, \ldots, 2n\}$ of possible values for S will be congruent with $n \pmod 3$ while others will not. Approximately one third of the values in the set of possible values for S will be congruent with $n \pmod 3$. The values in the set for which the congruence holds are what we call the *problematic values* and we want to rule out assignments adding up to one of those values: we want to build a formula such that whenever an assignment leads to S being a problematic value, the formula is falsified. On the other hand, all assignments that lead to a normal, non-problematic value of S, must make the formula true.

Let us consider, for instance, the case for $n = 2$, where we have a formula $\varphi(x_1, y_1, x_2, y_2)$. We have $S = (x_1 + y_1) + (x_2 + y_2) \in \{0, 1, 2, 3, 4\}$. There is a single problematic value in this case, $S = 2$. This means that we must rule out all assignments with two 1's, while the rest of them must make the formula true. For each problematic assignment we can build a specific clause such that only that assignment makes it false. For $n = 2$, the problematic assignments and their corresponding clauses are the following:

$$\begin{aligned}
(0,0,1,1) &\to (x_1 \lor y_1 \lor \neg x_2 \lor \neg y_2)\\
(0,1,0,1) &\to (x_1 \lor \neg y_1 \lor x_2 \lor \neg y_2)\\
(1,0,0,1) &\to (\neg x_1 \lor y_1 \lor x_2 \lor \neg y_2)\\
(0,1,1,0) &\to (x_1 \lor \neg y_1 \lor \neg x_2 \lor y_2)\\
(1,0,1,0) &\to (\neg x_1 \lor y_1 \lor \neg x_2 \lor y_2)\\
(1,1,0,0) &\to (\neg x_1 \lor \neg y_1 \lor x_2 \lor y_2)
\end{aligned}$$

The conjunction of those clauses is a Boolean formula written in Conjunctive Normal Form, verifying the desired modular property. In general, we can build Chen Formulae of Type 2 by looking for the problematic values in $\{0, \ldots, 2n\}$ and then adding a clause for each assignment that makes S add up to one of those values.

The reader may feel, however, that there are shortcomings to this representation of φ. The most urgent concern comes from the nagging question of *how many clauses will φ have when written this way?* The set of all possible assignments has cardinality $2^{2n} = 4^n$. Since, on average, one third of those assignments are related to problematic values of S, φ would end up having $\lceil \frac{4^n}{3} \rceil$ clauses. It is intuitive to see why this happens, though we can look for a more convincing proof in the fact that the exact number of clauses C is determined by the following sum, which turns out to be the mentioned $\lceil \frac{4^n}{3} \rceil$:

$$C = \sum_{i=0}^{\lfloor \frac{2n}{3} \rfloor} \binom{2n}{3i + (n \bmod 3)} = \left\lceil \frac{4^n}{3} \right\rceil$$

The previous equality can be easily checked by induction.

Fortunately, Chen Formulae of Type 2 admit a succinct representation when written in the form of a circuit, where we do not force φ to be in CNF and we are allowed some additional operators, such as XOR gates ($\oplus$). We now explain how to build these circuits.

For every $k \in \{1, \dots, n\}$ and every $m \in \{0, 1, 2\}$, we will now consider the auxiliary circuits μ_m^k over variables $(x_1, y_1, \dots, x_k, y_k)$, which are defined so that they verify the following property:

$$\mu_m^k = 1 \Leftrightarrow \sum_{i=1}^{k} (x_i + y_i) \equiv m \pmod 3$$

For $k = 1$, the μ-circuits are:

$$\mu_0^1 = \neg x_1 \wedge \neg y_1$$
$$\mu_1^1 = x_1 \oplus y_1$$
$$\mu_2^1 = x_1 \wedge y_1$$

If we have circuits μ_m^k for every k up to $n-1$, we can easily obtain the ones for n:

$$\mu_0^n = (\mu_0^{n-1} \wedge \neg x_n \wedge \neg y_n) \vee (\mu_1^{n-1} \wedge x_n \wedge y_n) \vee (\mu_2^{n-1} \wedge (x_n \oplus y_n))$$

$$\mu_1^n = (\mu_0^{n-1} \wedge (x_n \oplus y_n)) \vee (\mu_1^{n-1} \wedge \neg x_n \wedge \neg y_n) \vee (\mu_2^{n-1} \wedge x_n \wedge y_n)$$

$$\mu_2^n = (\mu_0^{n-1} \wedge x_n \wedge y_n) \vee (\mu_1^{n-1} \wedge (x_n \oplus y_n)) \vee (\mu_2^{n-1} \wedge \neg x_n \wedge \neg y_n)$$

Now, we can easily express our formula φ as

$$\varphi = \neg \mu_{n \bmod 3}^n$$

Since the size of every μ-circuit is constant and we have three μ-circuits for every $k \in \{1, \dots, n\}$, we have $3n$ auxiliary circuits plus the final one for φ, which means we can build Chen Formulae of Type 2 in size $\Theta(n)$ when building them as circuits.

2.3.3 QParity Formulae

The QParity formulae were first introduced in [11] and later used in [9] to show that Extended Q-Resolution can find short proofs of them, while Weak Extended Q-Resolution needs exponential size to show their unsatisfiability[1].

They are defined as follows:

[1]This is related to the question of complexity of CNF transformation. When writing these formulae in CNF, putting the auxiliary variables existentially quantified at the end of the existing

Definition 2.3 (QParity circuits, [9])**.** Let $n \in \mathbb{N}$, $n \geq 2$, and let $x_1, \dots, x_n$ and z be Boolean variables. We define the quantifier prefix $P_n = \exists x_1 \dots \exists x_n \forall z$. We define an auxiliary circuit t_2 as $t_2 = x_1 \oplus x_2$ and for $i \in \{3, \dots, n\}$ we define auxiliary t-circuits as $t_i = t_{i-1} \oplus x_i$ and the complete matrix as $\rho_n = t_n \oplus z$. The QBF instance will be $\text{QParity}_n = P_n : \rho_n$.

The formulae express that there exists an assignment to the x-variables such that $x_1 \oplus \cdots \oplus x_n$ is neither 0 nor 1, an obvious contradiction, which makes the QParity formulae always unsatisfiabiable.

In this case, it is possible to obtain a CNF version of these formulae considering the t-circuits to be auxiliary variables and expressing the $\oplus$ operation and assignment as the conjunction following conjunction, for $i \in \{3, \dots, n\}$:

$$(\neg t_{i-1} \vee \neg x_i \vee \neg t_i) \wedge (t_{i-1} \vee x_i \vee \neg t_i) \wedge (\neg t_{i-1} \vee x_i \vee t_i) \wedge (t_{i-1} \vee \neg x_i \vee t_i)$$

Then, we could define a new quantifier prefix with this additional variables:

$$\exists x_1 \dots \exists x_n \forall z \exists t_2 \dots \exists t_n$$

Slightly modifying the QParity formulae we can obtain a close family, the QInnerProduct formulae, where each variable x_i is now interchanged with the conjunction of two new variables, y_i and z_i.

The QParity definition is interesting in that it has all the defining characteristics of a formula family used for proof complexity purposes: they have no interesting meaning, they are easy to define and manipulate and they can be easily converted into CNF —and vice versa.

2.3.4 Chromatic Formulae

The *Chromatic Number Problem* is a well-known **DP**-complete problem: given a graph G and a natural number $k \geq 1$, decide whether k is the *chromatic number* of G, i.e. the minimum k such that G is k-colorable. Although this is close to k-colorability and can be encoded into a SAT formula, a more natural encoding is also possible using quantification: there *exists* a coloring of G with k colours and *for all* other coloring of size $k - 1$, these are not valid colorings for G.

We need to define a formula family for the problem, depending on two parameters: the graph G and the number k. The following definition or encoding for this problem was given by Sabharwal et al. in [32].

Definition 2.4 (General Chromatic Formulae, [32])**.** Let $G = (V, E)$ be a graph and let k be a positive natural number. Let $n = |V|$. We define variables $x_{i,j}$ for $i \in [n]$ and $j \in [k]$ and $y_{i,j}$ for $i \in [n]$ and $j \in [k - 1]$ (semantically, any of these variables is

prefix causes the formulae to be hard in the *weak* version of the proof system. It is an open question what should the optimal position of auxiliary quantifiers be when performing the transformation —or whether there exists such an optimal placement strategy.

set to 1 if and only if node i is set to have colour j). Then we define the following two subformulae:

$$\Gamma = \bigwedge_{i\in[n]} (x_{i,1} \vee \cdots \vee x_{i,k}) \wedge \bigwedge_{\substack{i\in[n] \\ j\neq j'\in[k]}} (\neg x_{i,j} \vee \neg x_{i,j'}) \wedge \bigwedge_{\substack{(i,i')\in E \\ j\in[k]}} (\neg x_{i,j} \vee \neg x_{i',j})$$

$$\Delta = \bigvee_{i\in[n]} (\neg y_{i,1} \wedge \cdots \wedge \neg y_{i,k-1}) \vee \bigvee_{\substack{i\in[n] \\ j\neq j'\in[k-1]}} (y_{i,j} \wedge y_{i,j'}) \vee \bigvee_{\substack{(i,i')\in E \\ j\in[k-1]}} (y_{i,j} \wedge y_{i',j})$$

Semantically, Γ is true when the x variables form a legal k-coloring, while Δ is true only when the y variables do not form a legal $(k-1)$-coloring. Thus, the *Chromatic Formula* is

$$\mathcal{K}(G,k) = \exists x_{1,1} \ldots x_{1,k} \ldots x_{n,1} \ldots x_{n,k} \forall y_{1,1} \ldots y_{1,k-1} \ldots y_{n,1} \ldots y_{n,k-1} : \Gamma \wedge \Delta$$

Regarding the definition's structure, it is interesting because it is the first family we encounter that has two parameters, and, besides, one of them is a mathematical object different from a natural number. The support for non-scalar parameters in formula family definitions is discussed at length next chapter.

2.3.5 Janota Formulae

As we will further discuss in Chapter 4, many **PSPACE** problems take the form of two-player games than can be encoded in QBF considering we have a *universal player* and an *existential player*.

The Janota formulae, first introduced in [24] and later used in [23], consider a very simple two-player game on over the Cartesian product of two sets $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$. We represent the Cartesian product as in Figure 2.1 and let the existential player start the game by deleting a cell from every column. Then, the universal player chooses one of the two rows and if the chosen row contains all the elements in A or all the elements in B, then the universal player wins. The universal player has a winning strategy for this game (see [23] for the argument).

a_1	$\ldots$	a_1	$\ldots$	a_n	$\ldots$	a_n
b_1	$\ldots$	b_n	$\ldots$	b_1	$\ldots$	b_n

FIGURE 2.1: Board for the Janota Formulae game (as depicted in [23]).

The game is encoded into a QBF formula over variables $x_{i,j}$ (these are true if and only if the existential player deletes the i-th cell from the j-th column) and a variable z denoting whether the universal player chooses the first or the second row. Finally, we add variables a_i and b_i used to make sure that if either of the sets is complete in one of the rows, the formula is falsified. Then, because there is a winning strategy for the universal player, the formula is always unsatifiable.

Definition 2.5 (Janota Formulae, [23])**.** Let n be a positive natural number. We consider the sets of variables $\mathcal{X} = \{x_{i,j} | i, j \in [n]\}$ and $\mathcal{L} = \{a_i | i \in [n]\} \cup \{b_i | i \in [n]\}$. The *Janota formula* of size n is defined with the conjunction of the following clauses:

$$\begin{array}{ll} x_{i,j} \lor z \lor a_i & \text{where } i, j \in [n] \\ \neg x_{i,j} \lor \neg z \lor b_i & \text{where } i, j \in [n] \\ \bigvee_{i \in [n]} \neg a_i & \\ \bigvee_{i \in [n]} \neg b_i & \end{array}$$

The QBF is

$$\mathcal{J}_n = \exists \mathcal{X} \forall z \exists \mathcal{L} : \varphi$$

where φ is the conjunction of the defined clauses clauses.

Janota formulae are interesting for two reasons. Firstly, because of the quantifier vector, as it provides a case were we define a vector based on a block and not on quantification over each one of the variables. Secondly, it presents a way to define unsatisfiable formulae from simple two-player games, a prolific source of hard-to-prove contradictions for proof complexity research in the **PSPACE** domain.

2.3.6 KBKF Formulae

The Kleine-Büning-Karpinski-Flögel formulae, or KBKF for short, were first introduced in [12] and have been later extensively used in proof complexity. There are several variants of this family depending on the proof system for which we want them to be difficult. Here we present a version of the original definition, as formulated in [11], as the variants do not present any particular differences regarding their definition style.

Definition 2.6 (KBKF Formulae, [11])**.** Let $t \in \mathbb{N}^*$. We define the prefix

$$P_t = \exists y_0 y_{1,0} y_{1,1} \forall x_1 \exists y_{2,0} y_{2,1} \forall x_2 \ldots \forall x_{t-1} \exists y_{t,0} y_{t,1} \forall x_t \exists y_{t+1} \ldots y_{t+t}$$

Now we define the following clauses:

$$\begin{array}{ll} C\_ = \neg y_0 & \\ C_0 = y_0 \lor \neg y_{1,0} \lor y_{1,1} & \\ C_i^0 = y_{i,0} \lor x_i \lor \neg y_{i+1,0} \lor \neg y_{i+1,1} & \text{where } i \in [t-1] \\ C_i^1 = y_{i,1} \lor x_i \lor \neg y_{i+1,0} \lor \neg y_{i+1,1} & \text{where } i \in [t-1] \\ C_t^0 = y_{t,0} \lor x_t \lor \neg y_{t+1} \lor \cdots \lor \neg y_{t+t} & \\ C_t^0 = y_{t,1} \lor x_t \lor \neg y_{t+1} \lor \cdots \lor \neg y_{t+t} & \\ C_{t+i}^0 = x_i \lor y_{t+i} & \text{where } i \in [t] \\ C_{t+i}^1 = \neg x_i \lor y_{t+i} & \text{where } i \in [t] \end{array}$$

The KBKF QBF $\mathcal{KBKF}_t$ is formed with the defined prefix P_t and the matrix obtained from the conjunction of all those clauses.

KBKF formulae are interesting because of their somewhat cumbersome structure, where a single parameter t defines a quite intricate quantifier vector, with some work on the prefix to perform. This is the first quantifier prefix were some of the subindices contain operations, such as $t + t$, $t - 1$ or $t + i$, rather than simple iterations over ranges.

Chapter 3

The Formal Language

In the previous chapter, we discussed what formula families are and what they look like. The main goal of this work is to present a formalism in which to easily write such definitions so that some computer program can later read and output instances. In this chapter, we turn our attention to the question of designing and specifying such a formal language.

In Section 3.1, we discuss the ideal features we would like our language to have, basing our decision on the definition styles and needs seen in the previous chapter. One of the main challenges in creating a language is to somewhat constrain the definitions to fundamental syntactic structures that can, at the same time, be as expressive as possible. Our formal language relies on the structure of *blocks*, sequences of literals that can be used to build both gates, clauses or quantifier prefixes. We present blocks —and its underlying components, *bricks*— in Section 3.2, and have a quick look at how they can be used to build formulae. We also look at how blocks written in the language are intended to be unfolded into actual sequences or literals (what we call the process of *fix-and-expand*). In Section 3.3, we give the formal encoding of two of the formula families defined in Chapter 2: the QParity and the Chromatic Formulae. Appendix A contains the full grammar of the language and Appendix B collects the formal version of all the other formula families covered so far. Finally, Section 3.4 briefly discusses the expressive power of the language and some potential formula family ideas that could be explored in this framework.

3.1 Some notes on desired features

Looking at the different examples of formula families in the previous chapter, we could say that the following features should at least be covered in our formalism:

- Both variables and clauses (or gates) should be easily grouped together and operated to build more complex structures, let that be either quantifier prefixes or matrices.

- Parameters could be something different from natural numbers (e.g. graphs).

- Parameters are not static (they are not used to define ranges as they are; they need to be operated and manipulated easily).
- Both CNF and circuits formats must be natively supported.
- Formal definitions should be as declarative as possible in nature.
- Formal definitions should be written via some fundamental construct that constrains the syntax of the definition while not restricting expressive power.
- It should be possible to encode a wide range of formula families, even those with parameter types not expected at this moment.

All these features boil down to two essential problems:

1. We need a basic declarative construct to build the matrix and the quantifier prefix.
2. The description of these constructs depends on the parameters of the formula family. These parameters can take any potential data type defined by the user and it must be easy to manipulate them.

To tackle the first goal we present the structure of *blocks*, which we discuss in the next section. The problem of having different data types and operations on them is solved embedding an external language (Python) into our own. We discuss this in Section 3.3.2 when we encode the Chromatic Formulae in our language.

3.2 The *block* structure

We have identified two main needs in the design of our language. First, a declarative construct to build the formulae. And second, effective ways to represent different parameter data types and operations on them. We now solve the former problem: we give a formalism in which to write the body of the definitions[1].

In this regard, our formal language relies on the basic concept of *blocks*. A block is a sequence of *bricks*, which are literals (input variables that may be negated) or references to other blocks (also possibly negated). A block can then be assigned a single *attribute*, i.e. a *quantifier* or a *logical operator* (conjunction, disjunction or exclusive disjunction).

We can define a block in a single line like this:

```
define block B := x, -y, A;
```

where `x` and `y` are Boolean variables, `y` is negated and `A` is a previously defined block. To add an attribute to a block, we specify if we operate it or quantify it (a block can only have one attribute). Some possible attributes for our block are:

[1]Appendix A contains the complete grammar or the language we are going to discuss. Here, we will only go through its basic syntax and features using some intuitive examples.

```
block B operated with OR;
block B operated with AND;
```

$\longrightarrow B = x \lor \neg y \lor A$

$\longrightarrow B = x \land \neg y \land A$

We also have the `XOR` operator whenever the block has only two bricks. On the other hand, if the block does not have negations (imagine `define block B := x, y, A`), then we can add quantifiers:

```
block B quantified with E;
block B quantified with A;
```

$\longrightarrow B = \exists x \exists y \exists A$

$\longrightarrow B = \forall x \forall y \forall A$

Let us now show how blocks can be combined to build a simple quantified Boolean formula.

Example 3.1 (Basic use of blocks). The formula $\varphi(x, y, z) = (x \lor y) \land z$ can be defined in our language using two blocks `B1` and `B2` (assuming that the variables x, y, z have already been declared):

```
define block B1 := x, y;            define block B2 := B1, z;
```

Blocks only declare ordering of bricks. Syntactically, they are just a sequential construction. Meaning is later given through an attribute, which will make the block into the component of a quantifier prefix or a gate in the matrix. In this case, we are encoding a simple propositional formula, so the attributes are the *and* and *or* operators. These operate between them all the bricks in the block.

```
block B1 operated with OR;          block B2 operated with AND;
```

However, note that inasmuch blocks only define ordering of literals, the structure captures simultaneously both the idea of gates on a Boolean circuit as well as the intricate nested patters of quantifier prefixes. Imagine that the previous formula is quantified as follows:

$$\forall x \exists y \exists z : \varphi(x, y, z)$$

We can define some blocks to obtain the structure of the quantifier prefix by first combining the blocks so that they order the variables appropriately and then specifying the attribute of the blocks:

```
define block Q1 := x;               block Q1 quantified with A;
define block Q2 := y, z;            block Q2 quantified with E;
define block Q := Q1, Q2;
```

Then, we can combine the quantifier prefix block `Q` with the block `B2` representing φ and indicate that this is our *output block*.

```
define block Phi := Q, B2;          output block: Phi;
```

The previous example showcased the use blocks, but it defined a single formula, while our goal is to describe the general structure of formulae belonging to a family. In other words, the structure will depend on some parameters. The next example introduces a more complex definition and uses two other languages features: *conditions* and *groupings*.

Example 3.2 (Defining a simple formula family). Let $n \in \mathbb{N}^*$. We will consider the formula family containing QBF over variables $x_1, \dots, x_n, y_1, \dots, y_n$ of the form

$$\Phi(n) = \exists x_1 \forall y_1 \exists x_2 \forall y_2 \dots \exists x_n \forall y_n : \varphi(x_1, \dots, x_n, y_1, \dots, y_n)$$

where φ is the matrix is given by

$$\varphi(x_1, \dots, x_n, y_1, \dots, y_n) = (\bigvee_{i=1}^{n} \neg x_i) \wedge \bigwedge_{i=1}^{n} (x_i \vee y_i)$$

For instance, in $n = 2$, the QBF is:

$$\Phi(2) = \exists x_1 \forall y_1 \exists x_2 \forall y_2 : (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee y_1) \wedge (x_2 \vee y_2)$$

We start by defining the parameters and the variables:

```
parameters: {                       variables: {
    n : int, ‘n >= 1‘;                  x(i)    where i in 1..n;
}                                       y(i)    where i in 1..n;
                                    }
```

As we can see, we specify the data type of the parameter `n` and impose the constraint that it must be positive. For the variables, whenever these have some indices, we must specify their ranges.

Now we define the quantifier prefix. First, define some blocks `QX(i)` and `QY(i)` that we will later quantify. These blocks are *grouped*. The grouping is just a collection of blocks, but it is not a block itself.

```
define blocks grouped in QX {       define blocks grouped in QY {
    QX(i) := x(i);                      QY(i) := y(i);
} where i in 1..n;                  } where i in 1..n;
```

The reason we want groupings is that it makes operating things easier. Now, to add the existential and universal quantifier, we just write:

```
all blocks in QX quantified with E;
all blocks in QY quantified with A;
```

This will add the corresponding attribute to every block in those groupings. We are now left with the task of composing those blocks into the quantifier prefix, `Q`:

```
define blocks {
    QXY(i) := QX(i), QY(i);
} where i in 1..n;
```

```
define blocks {
    Q := QXY(i);
} where i in 1..n;
```

Before we go on with the example, it is interesting to say a few words about the intended way in which these blocks are then supposed to unfold. For instance, the two following block definitions are different:

```
define blocks {
    X(i) := x(i);
} where i in 1..n;
```

```
define blocks {
    X := x(i);
} where i in 1..n;
```

The first piece of code defines n different blocks, each one containing a single variable. On the other hand, the second piece of code defines a single block, `X`, whose content is the sequence of all x-variables. If we were to expand `X`, we would see that it contains `x(1), x(2), ... , x(n)`. We refer to this way of depth-first unfolding of blocks as *fix-and-expand*. When going through the ranges, we first fix the values of the indices appearing in the left-hand side of the definition, and in the right-hand side, we *expand*: if a certain index does not have a fixed value, then we expand that brick by writing all its occurrences that meet the ranges defined in the `where` clause. Thus, in the definition of `X`, no index is fixed in the left-hand side, and therefore the `x(i)` brick is expanded, while in the definition of `X(i) := x(i)` we first fix the value of `i` in the left-hand side and then we only write one occurrence of `x(i)` in the right-hand side: `X(1) := x(1)`, `X(2) := x(2)` and so on.

Back to our example, we are left with the task of defining the matrix of the formula. The disjunction of all the negated x-variables can be expressed in a single block, while the clauses containing the pairs of x_i and y_i variables will be n different blocks. We write that as follows:

```
define blocks {
    X := -x(i);
} where i in 1..n;
```

```
define blocks grouped in XY {
    XY(i) := x(i), y(i);
} where i in 1..n;
```

Regarding operators, we add them as follows:

```
block X operated with OR;
all blocks in XY operated with OR;
```

Then we define the complete φ matrix in a block called `F` and we operate the bricks inside with conjunction:

```
define blocks {
    F := X, XY(i);
} where i in 1..n;

block F operated with AND;
```

Finally, we define the output block, which contains the quantifier prefix and the matrix:

```
define block Phi := Q, F;
```

3.3 Writing definitions in the language

We now offer the formal version of two of the formula families defined in Chapter 2: the QParity formulae (Definition 2.3) and the Chromatic Formulae (Definition 2.4). The formal versions of the rest of the families presented in the previous chapter can be found in Appendix B.

3.3.1 QParity Formulae

Let us recall the definition of the QParity circuits defined in the previous chapter. For some natural number $n \geq 2$, the circuit has the form

$$\text{QParity}_n = \exists x_1 \dots \exists x_n \forall z : \rho_n$$

where $\rho_n = t_n \oplus z$, and t_n is obtained by defining some auxiliary t-circuits: $t_2 = x_1 \oplus x_2$ and for $i \in \{3, \dots, n\}$, $t_i = t_{i-1} \oplus x_i$.

We start the formal definition by giving a name to the formula family and setting its format, `circuit-prenex`:

```
name: QParity;
format: circuit-prenex;
```

Then we declare *parameters*, followed by their type as well as possible constraints (in this case, $n \geq 2$). Using them, we can then declare the variables. Variables x_i are denoted `x(i)` and the range of indices must be specified.

```
parameters: {
    n : int, ‘n >= 2‘;
}
```

```
variables: {
    x(i)    where i in 1..n;
    z;
}
```

We now declare the blocks. We start with the ones used for quantifiers. All the x-variables are existentially quantified, followed by the variable z universally quantified.

```
define blocks {                      define block Z := z;
    X := x(i);                       define block Q := X, Z;
} where i in 1..n;

block X quantified with E;
block Z quantified with A;
```

In the same section, we define the blocks used to build the matrix of the formula. Here we use once again the feature of *groupings*: a set of blocks grouped under the same name, so that they can all be simultaneously operated or quantified.

```
define block T(2) := x(1), x(2);    define block Rho := T(n), z;
define blocks grouped in T {         block T(2) operated with XOR;
    T(i) := T(s), x(i);              all blocks in T operated with XOR;
} where i in 3..n, s = ‘i-1‘;        block Rho operated with XOR;
```

Finally we define the *output block*, `Phi`, and specify that that is the entrance to the whole formula i.e. `Phi` is $\text{QP{\small ARITY}}_n$:

```
define block Phi := Q, Rho;
output block: Phi;
```

3.3.2 Chromatic Formulae

In the previous chapter, the Chromatic Formulae were one of the most interesting examples we offered because the formulae had an intuitive, clear meaning, and because they had one parameter that was not a natural number: a graph.

When faced with the challenge of writing that definition in our formal language, we must address the second of the features needed mentioned at the beginning: our language should have support for basic integer arithmetic, but, additionally, we would also like to encode graphs, for formula families like the Chromatic Formulae. And not only graphs, e.g. if we encode a two-player game, we might need to encode the state of the board. These non-scalar objects may take the form of lists, matrices or other data structures. However, introducing native support for lists, matrices, sets and other data structures as well as operations on them would depart us from the goal of designing a language *for formula family definitions*. After all, we would be investing the time in defining an almost full-blown programming language just so that we can manipulate some simple expressions.

We tackle this issue embedding an external programming language into our system. In particular, Python, as this is the language we have used to build the tool we present in the next chapter. Thanks to embedded Python, definitions can contain any built-in Python data type and we can operate on them using any existing built-in Python function. This means that, for example, all evaluation of arithmetic expressions is left in the hands of the Python interpreter. For this to work, any complex expression will need to be enclosed in backticks. For example, if we want to indicate that the index i ranges from 1 to n, it will be enough to say `where i in`

`1..n`, but if, for instance, i ranges from 1 to $n^3 + 7$, that expression will need to be enclosed in backticks: `where i in 1..‘n**3 + 7‘`.

Now that we know how to encode and manipulate other parameters, we can give the formal definition of the Chromatic Formulae. Recall that they take two parameters, a graph $G = (V, E)$ and a number $k \in \mathbb{N}^*$, and are of the form:

$$\mathcal{K}(G,k) = \exists x_{1,1} \ldots x_{1,k} \ldots x_{n,1} \ldots x_{n,k} \forall y_{1,1} \ldots y_{1,k-1} \ldots y_{n,1} \ldots y_{n,k-1} : \Gamma \wedge \Delta$$

where

$$\Gamma = \bigwedge_{i\in[n]} (x_{i,1} \vee \cdots \vee x_{i,k}) \wedge \bigwedge_{\substack{i\in[n] \\ j\neq j'\in[k]}} (\neg x_{i,j} \vee \neg x_{i,j'}) \wedge \bigwedge_{\substack{(i,i')\in E \\ j\in[k]}} (\neg x_{i,j} \vee \neg x_{i',j})$$

and

$$\Delta = \bigvee_{i\in[n]} (\neg y_{i,1} \wedge \cdots \wedge \neg y_{i,k-1}) \vee \bigvee_{\substack{i\in[n] \\ j\neq j'\in[k-1]}} (y_{i,j} \wedge y_{i,j'}) \vee \bigvee_{\substack{(i,i')\in E \\ j\in[k-1]}} (y_{i,j} \wedge y_{i',j})$$

The main new feature we showcase when encoding these formulae is that the graph taken as a parameter is used to check whether a certain edge (i, j) is in the graph, $(i, j) \in_? E$. For this purpose, we encode the graph as and adjacency matrix, `edges`, and then the condition can be written as `‘edges[i-1][j-1] == 1‘` (using Python syntax). Note that, for the sake of readability, we also introduce an extra parameter, `n`, denoting the number of nodes of the graph.

We start by defining parameters and variables, as usual:

```
name: Chromatic formulae;
format: circuit-prenex;

parameters: {
    n     : int, ‘n >= 1‘;
    edges : list, ‘len(edges) == n‘;
    k     : int, ‘k >= 1‘;
}

variables: {
    x(i, j)    where i in 1..n, j in 1..k;
    y(i, j)    where i in 1..n, j in 1..‘k-1‘;
}
```

Regarding the blocks, we only show a small example where we use the embedded Python syntax. For instance, when defining the clauses

$$\bigwedge_{\substack{(i,j)\in E \\ l\in[k]}} (\neg x_{i,j} \vee \neg x_{i',j})$$

in the subformula Γ, we add the following block definition:

```
define blocks grouped in SubGamma3 {
    SG3(i, j, l) := -x(i, l), -x(j, l);
} where i in 1..n, j in 1..n, ‘edges[i-1][j-1] == 1‘, l in 1..k;
    ...
define block Gamma3 := all blocks in SubGamma3;
    ...
all blocks in SubGamma3 operated with OR;
block Gamma3 operated with AND;
```

The complete encoding of the Chromatic Formulae can be found in Appendix B.4.

3.4 Expressive power and potential use cases

A relevant question before we close the chapter is whether the language presented above can declare all possible families of formulae. Though this is a complicated matter and we have not explored it in depth, we could advance that this language should in principle be able to express virtually any set of QBF, given the fact that having access to Python makes the language Turing-complete. Therefore, our formal language should be able to express —albeit inconveniently— the definition of any computable family of QBF.

Note the emphasis on *computable*: there are formula families that exist as mathematical objects but cannot be effectively built, so to say, and therefore cannot be expressed in this language. Think of certain circuit families in non-uniform Boolean circuit complexity classes, like $\mathbf{P}/\mathsf{poly}$. In $\mathbf{P}/\mathsf{poly}$ we have languages like UHALT: the set of unary strings encoding a pair (M, x) of a Turing machine and an input x such that M halts on input x (see [5]). The formula family UHALT $= \{\mathcal{UH}_n : n \in \mathbb{N}^*\}$ exists, in that the Boolean circuit $\mathcal{UH}_n$ can be defined as the conjunction of the input bits if the input of length n halts, and as some trivially unsatisfiable circuit when the input of length n does not halt. This language is in $\mathbf{P}/\mathsf{poly}$ and it constitutes a formula family, but it cannot be computed and therefore it cannot be expressed in the language.

Amongst the things the language can express are non-prenex families. This is because the space in which blocks are defined is the same for both quantifier blocks and gate blocks. Therefore, they can be combined to put quantifiers before a gate, and so on. Nevertheless, as we will later discuss when talking about the computer tool implementing the language, this is more of an experimental feature, given the lack of non-prenex solvers currently available.

It is also interesting to come back now to a question mentioned in the introduction of this work. As we said, tools like SOGrounder are capable of modelling formula families and some of the potential uses may seem to overlap with the capabilities of the formal language we just introduced. Although arguably many problems can be modelled in both languages, SOGrounder cannot handle parameters that control alternation patterns of quantifiers. For instance, SOGrounder could be used to model the game of chess on a traditional 8×8 board, yet it cannot model the general case for $n \times n$ boards, used to show that *generalized* chess in **PSPACE**-complete (see [33]). If we want to play actual chess, on a normal board only, SOGrounder will be more convenient; if, on the other hand, we are interested in the hardness of chess and the length of proofs in QBF solvers as the size of the board increases, we will need to use a formalism like this one.

Finally, we could venture some other potential uses for this language. So far we have only encoded either simple formula families or existing sets defined in the literature. Though this is the main goal of this language, its capabilities should encourage users to play and easily define more inventive sets of hard-to-prove formula families. For instance, with some small modifications, it would be possible to make the language accept user-defined Python objects and packages. If one modelled a Turing machine and using that encoded the Cook-Levin formulae, something like the Sipser-Gács formulae could be expressed in this language, the set of QBF used in the Sipser-Gács theorem showing the class **BPP** is contained in the Polynomial Hierarchy ($\mathbf{BPP} \subseteq \Sigma_2^p \cap \Pi_2^p \subseteq \mathbf{PH}$, see [5]).

Another potential use, via Python's built-in functions for random number generation, is the encoding of existing models for random generation of hard-to-solve QBF, like the original Chen-Interian model (see [15]) and later modifications and extensions like the one proposed by Amendola et al. in [3].

Chapter 4

A Case Study: Encoding Geography in QBF

In the previous chapter, the Chromatic Formulae were presented, a QBF encoding of the CHROMATIC NUMBER problem, and we offered a formal version of them into our language. One may argue, however, that the Chromatic Formulae were a somewhat forced example: after all, that is a **DP**-complete problem that can be solved using the conjunction of two propositional formulae: one expressing that the graph is k-colorable and another one expressing it is not $k-1$-colorable. Therefore QBF solvers may incur unnecessary overheads in performance, while a SAT solver would do just fine.

In this chapter, we turn our attention to a proper **PSPACE** problem — at least as long as **NP** $\neq$ **PSPACE**. Amongst popular **PSPACE** problems are those of two-player games, where we try to decide whether one of the players (the *existential* player) has a forced win over its opponent (the *universal* player). Some popular examples of endgame decision problems based on board games are derived from Chess, Checkers or Go. We present a simpler game that is, nevertheless, as hard as the previous ones: GEOGRAPHY. In fact, GEOGRAPHY has shown to be a versatile problem, as it has been used to prove the **PSPACE**-completeness of endgame board games such as Checkers (see [19]).

We start the chapter by presenting the rules of the game in Section 4.1. In Section 4.2 we offer a generalized version of GEOGRAPHY on a graph, known as GENERALIZED GEOGRAPHY, which is a proper **PSPACE**-complete problem. Besides, we discuss a particular case in which the problem can be solved in polynomial time, an interesting feature to test the performance of QBF solvers and proof complexity. Finally, in Section 4.3, we provide an encoding of k-GENERALIZED GEOGRAPHY into QBF as well as its formal version into our language.

4.1 The Geography game

In its most basic form, GEOGRAPHY is a game played between two players, which we call player P and player Q, who try to test each other's geography knowledge by shouting the name of capital cities from all around the world.

Player P starts the game saying the name of the capital city of the country they are in. Let's say P is in Belgium and starts by saying *Brussels.* Now the second player, Q, must choose a city starting with the last letter of the city said by P. In this case, a world capital starting with *s*. Player Q may choose *Seoul*, starting with *s*, and now player P will have to find a city name starting with *l*.

Player P has now several options, such as *La Valeta* or *Luxembourg*, but in real life, no cities may spring to mind — after all, this is a game to test your geography knowledge! Thus, if P cannot come up with a city, they lose the game.

Presented like this it may seem like a frivolous game where players shout city names until nothing else comes to mind. We can make Geography more of a strategic endeavour by considering that the two players have access at any moment to a full list with the valid city names they are playing over, such as the list with all 197 capital cities in the world. Now the game is no longer about testing your memory, but about finding a perfect strategy: a sequence of cities such that no matter what the other player answers, they will eventually end up in a position where all the possible cities have already been used.

For a complete example, imagine again we are playing over world capitals, starting in Belgium. Then what follows is a valid game were player Q loses: no capital city in the world starts with *z* other then Zagreb, and that one has already been used by P in move 7.

1. P: Brussel**s** (Belgium)
2. Q: **S**ofi**a** (Bulgaria)
3. P: **A**msterda**m** (Netherlands)
4. Q: **M**ins**k** (Belarus)
5. P: **K**abu**l** (Afghanistan)
6. Q: **L**a Pa**z** (Bolivia)
7. P: **Z**agre**b** (Croatia)
8. Q: **B**erli**n** (Germany)
9. P: **N**uu**k** (Greeenland)
10. Q: **K**ie**v** (Ukraine)
11. P: **V**adu*z* (Liechtenstein)

This time P won Q in 11 moves, and we could say that P's strategy of securing *Zagreb* was clearly a master move. However, how particular is this strategy? Would it always work? After all, in their first move, Q had a long list of cities to choose from starting with *s*: Sarajevo, Seoul, Singapore, Stockholm... Likely, for one of those choices they might have been able to avoid ending up stuck in Vaduz.

Player P is thus forced to think even more generally: they want to anticipate every possible move player Q could make. They want to make sure that after each of Q's moves, they can find a city that will eventually make them win. In other words, P wants to make sure he has a *forced win.* Therefore, the question we may ask is the following: given a list of cities and a starting city c_0, whatever Q chooses in each turn, is there always a possible choice for player P such that Q will eventually be stuck?

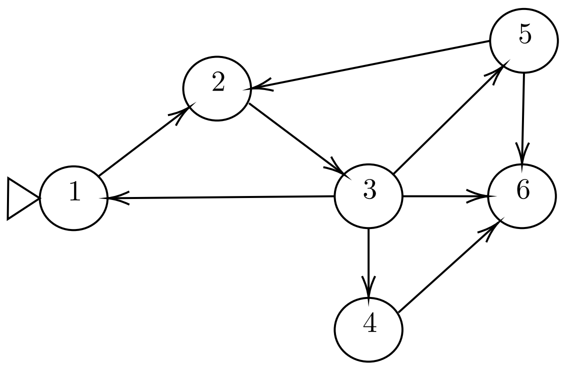

Figure 4.1: Example of a graph to play to Generalized Geography.

The **PSPACE**-like nature of the Geography should now become apparent, in that it very much resembles the quantifier alternation of a quantified Boolean formula: *for every* move Q makes, *there exists* a move for P, such that *for every* other move Q may take afterwards... Q always ends up in a position where they have no way out.

4.2 Generalized Geography

Under the conjecture $\mathbf{NP} \neq \mathbf{PSPACE}$, if a forced win like the one described above exists, there may not always be short certificates to prove it. Nevertheless, for many reasonable instances we could still ask if P has a forced win by sending an encoding of this problem to a QBF solver. In order to ease this task for computer tools we present a more abstract version of Geography based on graphs: Generalized Geography and k-Generalized Geography.

Instead of a list of cities, we will play over a graph $G = (V, E)$, where we can intuitively think of the nodes as the former cities of Geography. Whenever a city starts with the last letter of another city in the graph, those nodes will be adjacent and there will be a directed edge between them. That is, $(\text{Brussels}, \text{Seoul}) \in E$ but $(\text{Seoul}, \text{Brussels}) \notin E$. Additionally, we establish a starting node s, which we consider to be player P's initial move.

An example of Generalized Geography can be built using the graph in Figure 4.1, where we assume 1 to be the initial node. Q will be forced to choose 2 and P will only be able to choose 3. Now Q may choose either 4, 5 or 6 (note that $(3, 1) \in E$ but node 1 has already been used, so that is not a possible move). In the first two cases P wins because they can then move to 6 and lock Q in there. However, there is one scenario in which P loses: if Q goes from 3 to 6, player P is left without movements. We would conclude that P does not have a forced win in this graph.

We can define the game more formally as follows.

Definition 4.1 (Generalized Geography)**.** Let $G = (V, E)$ be a graph a let $s \in V$ be a node in that graph. Consider the following game between two players

P and Q: P starts the game by choosing s and then the two players take turns, each time choosing a node that extends the current path starting from s, without repeating nodes. We call this game Generalized Geography.

We say that player P has a *winning strategy in* Generalized Geography *over G starting at s* if when P starts the game at s, they can can always choose nodes in such a way that eventually Q is unable to extend the path. If, in particular, P has a winning strategy in k movements, we say they have a winning strategy in k-Generalized Geography.

We denote by GenGeo the set of all pairs (G, s) of a graph and a node such that player P has a winning strategy in G starting at s. In particular, we denote by k-GenGeo the pairs of graphs and nodes such that P has a winning strategy in k-Generalized Geography.

Generalized Geography no longer depends on cities but on graphs and, as such, we can study its complexity as the size of G and k increases. Under this formulation, Generalized Geography is **PSPACE**-complete (see [30] for a complete proof showing that TQBF can be polynomial-time reduced to GenGeo) and thus it will be hard to decide whether P has a forced win over Q for big enough graphs. The case with the 197 different cities will be, arguably, a difficult one.

Note that if we fix the number of moves k, then deciding whether a player has a winning strategy in k-Generalized Geography is Σ_k^p-complete, as the QBF encoding the decision would be an instance of Σ_kSAT.

4.2.1 The particular case of acyclic graphs

Interestingly enough, there exists a particular scenario in which deciding whether P has a forced win is not that hard: whenever the graph is acyclic, we can compute the decision in polynomial time. The existence of such a polynomial-time algorithm is suggested in [27]. We now briefly sketch such a procedure.

Algorithm 4.1 (Generalized Geography on a DAG)**.** Let $G = (V, E)$ be a directed acyclic graph, let $s \in V$ be the starting point and $n = |V|$. We want to decide if P has a forced win over player Q when playing over G.

1. Since G is acyclic, we can compute a topological ordering of the graph through any classical variation of depth-first search methods (see [17, 27]). This can be done in time $O(|V| + |E|) \subseteq O(n^2)$. Let $v_1, \ldots, v_n$ be such a topological ordering of the vertices and suppose node s, the starting point, corresponds to node v_s in the ordering.

2. Declare Boolean variables $w_1, \ldots, w_n$. The intended meaning of these variables is that $w_i = 1$ if and only if the player that is about to move has a forced win starting in v_i.

3. Since the last node has no outgoing edges, any player starting in v_n will lose, so we can safely set $w_n = 0$.

4. We compute the value of the remaining w_i through dynamic programming (see [17, 27]) in a descending order. Starting at w_{n-1}, we set $w_i = 1$ if and only if there exists $j > i$ such that $(v_i, v_j) \in E$ and $w_j = 0$, i.e. if the player that is about to move can move to v_j and we already know that the other player will have a forced loss starting at v_j. Each one of the w_i values can be computed in at most $O(n)$ time and thus the complete table can be obtained in $O(n^2)$.

5. Player P has a forced win starting at s if and only if $w_s = 1$.

The previous algorithm can compute whether P has a forced win over Q in time $O(n^2)$. Besides, it can be easily modified by adding a step counter such that we can also decide if P has a forced win specifically in k moves. Clearly, the previous procedure only works for acyclic graphs, for otherwise getting the topological ordering would be impossible.

The fact that the particular case of directed acyclic graphs can be computed in polynomial time is a very interesting property in the context of proof complexity and empirical research of QBF solvers. Once we encode Generalized Geography into QBF, we will be able to check whether the formulae obtained for acyclic graphs are solved significantly faster than those for very similar but cyclic graphs.

Going into the more theoretical domain of proof complexity, we may ask whether certain proof systems always prove in polynomial size the Generalized Geography formulae for acyclic graphs — or whether, perhaps, proof systems cannot see beyond the formula representation and cannot exploit this property. Recall that under the **NP** $\neq$ **PSPACE** conjecture, for every proof system for TQBF there will exist some QBF with exponentially long certificates. Analogously, since Generalized Geography is **PSPACE**-complete, we expect that some instances will also have exponentially long certificates, yet there might exist proof systems where the case for acyclic graphs can be efficiently proven.

4.3 The Geography Formulae

We are now ready to encode the k-Generalized Geography game into QBF formulae. Again, we are interested in deciding whether the existential player P has a forced win over the universal player Q on a graph $G = (V, E)$, where we denote $n = |V|$, starting at $s \in V$ in k moves (not counting the initial predefined move to s which we refer to as “move 0”).

4.3.1 Encoding the game

We start by defining the Boolean variables used in the formulae. We have variables

$$\begin{aligned}
&p_{i,m}, && \text{where } i \in \{1, \ldots, n\} \text{ and } m \in \{0, 2, 4, \ldots, k\} \\
&q_{i,m}, && \text{where } i \in \{1, \ldots, n\} \text{ and } m \in \{1, 3, 5, \ldots, k-1\} \\
&s_m, && \text{where } m \in \{1, 3, \ldots, k-1, k+1\}
\end{aligned}$$

Semantically, $p_{i,m}$ is true if and only if player P chooses node i at move m. The q-variables have the same meaning, but they are only defined for odd move numbers, as P can only move in even moves and Q in the odd ones. This already gives a condition for the parameters of the formulae: we will require that $k \bmod 2 = 0$. Besides, s_m is intended to be true if and only if the game is already stuck for player Q at time m or move m is the first time Q gets stuck.

The quantifier prefix of the formulae (we are building a prenex circuit) is straightforward: there exists a movement for P at time 0, such that there exists a stuckness state after that move, such that for all choices Q makes at move 1, there exists a move for P at time 2... and so on. We denote this prefix $P_{n,k}$:

$$\exists p_{1,0} \dots p_{n,0} \exists s_1 \forall q_{1,1} \dots q_{n,1} \dots \ \forall q_{1,k-1} \dots q_{n,k-1} \exists p_{1,k} \dots p_{n,k} \exists s_{k+1}$$

We are left with the task of modelling the problem constraints into propositional logic. Arguably, not all of the possible arrangements of variables will be valid, so we first need to encode what the validity of a move is. In particular, we want to avoid the possibility of either of the players trivially winning by choosing "not to play", selecting invalid assignments.

Now, what are the constraints we need to impose? We have four **validity constraints**, to which we give some names: P must choose s as initial movement (*initial condition*); at each move only one node must be chosen (*uniqueness of choice*); every node is only used at most once (*no overlapping*); and if two nodes are chosen consecutively, then there must exist an edge between them (*connectedness*).

Besides, we have some conditions for what it means to become stuck. This is the *stuckness condition*: if P chooses i at time m, Q has no way out, i.e. Q is stuck at node i and loses the game.

We now give the encoding of each one of these constraints.

1. **Validity**

   a) *Initial condition*
   P's initial choice (move 0) is node s. We add a single clause with variable $p_{s,0}$.

   b) *Uniqueness of choice*
   At each move m, only one node must be chosen. We add subformulae $\mathcal{U}_p(m)$ and $\mathcal{U}_q(m)$, each one making sure that at each move m, one and only one node is chosen.

$$\mathcal{U}_p(m) = \bigwedge_{i \in [n]} (p_{i,m} \leftrightarrow \bigwedge_{\substack{j \in [n] \\ j \neq i}} \neg p_{j,m})$$

$$\mathcal{U}_q(m) = \bigwedge_{i \in [n]} (q_{i,m} \leftrightarrow \bigwedge_{\substack{j \in [n] \\ j \neq i}} \neg q_{j,m})$$

c) *No overlapping*
Every node can be used at most once in the game by each player. Thefore, if a player chooses node i at time m, it must not have been chosen before.

$$\mathcal{O}_p(m) = \bigwedge_{i\in[n]} (p_{i,m} \to (\bigwedge_{m'\in\{0,2,\ldots,m-2\}} \neg p_{i,m'} \wedge \bigwedge_{m'\in\{1,3,\ldots,m-1\}} \neg q_{i,m'}))$$

$$\mathcal{O}_q(m) = \bigwedge_{i\in[n]} (q_{i,m} \to (\bigwedge_{m'\in\{0,2,\ldots,m-1\}} \neg p_{i,m'} \wedge \bigwedge_{m'\in\{1,3,\ldots,m-2\}} \neg q_{i,m'}))$$

d) *Connectedness*
If P chooses node i at step m, then it must be adjacent to the node chosen by Q at $m-1$. We add two subformulae $\mathcal{C}$ imposing the connectedness condition:

$$\mathcal{C}_p(m) = \bigwedge_{i\in[n]} (q_{i,m-1} \to \bigvee_{(i,j)\in E} p_{j,m})$$

$$\mathcal{C}_q(m) = \bigwedge_{i\in[n]} (p_{i,m-1} \to \bigvee_{(i,j)\in E} q_{j,m})$$

Now, let $\mathcal{V}_p(0) = p_{s,0} \wedge \mathcal{U}_p(0)$ and for $m \in \{2,4,\ldots,k\}$,

$$\mathcal{V}_p(m) = \mathcal{U}_p(m) \wedge \mathcal{O}_p(m) \wedge \mathcal{C}_p(m)$$

Analogously, for $m \in \{1,3,\ldots,k-1\}$, we define

$$\mathcal{V}_q(m) = \mathcal{U}_q(m) \wedge \mathcal{O}_q(m) \wedge \mathcal{C}_q(m)$$

We have that $\mathcal{V}_p(m)$ and $\mathcal{V}_q(m)$ are true if and only if player P (resp. Q) makes a valid assignment to their variables at move m.

Besides, for $m \in \{1,3,\ldots,k-1,k+1\}$ we denote

$$\mathcal{V}_<(m) = \bigwedge_{m'\in\{0,2,\ldots,m-1\}} \mathcal{V}_p(m') \wedge \bigwedge_{m'\in\{1,3,\ldots,m-2\}} \mathcal{V}_q(m')$$

such that $\mathcal{V}_<(m)$ is true if and only if all the movements up to but not including m were valid.

2. **Stuckness**
To express the stuckness we first encode a formula $\mathcal{S}(m)$ expressing that Q is stuck for the first time at time m. That is, if P chose some node i at time $m-1$, all the adjacent nodes are already used before. For $m \in \{1,3,\ldots,k-1,k+1\}$,

$$\mathcal{S}(m) = \bigvee_{i\in[n]} (p_{i,m-1} \wedge \bigwedge_{(i,j)\in E} (\bigvee_{m'\in\{0,2,\dots,m-3\}} p_{j,m'} \vee \bigvee_{m'\in\{1,3,\dots,m-2\}} q_{j,m'}))$$

Besides, we set $s_1 \leftrightarrow \mathcal{S}(1)$ and for every $m \in \{3,5,\dots,k-1,k+1\}$,

$$s_m \leftrightarrow (s_{m-2} \vee \mathcal{S}(m))$$

Now, winning the game is expressed as follows:

1. P makes a valid move at move 0.
2. At every odd move m, either:
   a) Q is already stuck or gets stuck for the first time.
   b) If Q makes a valid move, then P makes a valid move at $m+1$.
3. The game is stuck at move $k+1$.

For every odd movement we denote by $\mathcal{M}(m)$ the implication of making a valid move:

$$\mathcal{M}(m) = (\mathcal{V}_q(m) \wedge \mathcal{V}_<(m)) \rightarrow \mathcal{V}_p(m+1)$$

For movement $k-1$ we also impose that after making the move Q gets stuck:

$$\mathcal{M}(k-1) = (\mathcal{V}_q(k-1) \wedge \mathcal{V}_<(k-1)) \rightarrow (\mathcal{V}_p(k) \wedge \mathcal{S}(k+1))$$

We express this in the following formula $\mathcal{W}$:

$$\begin{aligned}\mathcal{W}(G,k,s) = \mathcal{V}_p(0) &\wedge (s_1 \vee \mathcal{M}(1))\\ &\wedge (s_3 \vee \mathcal{M}(3))\\ &\vdots\\ &\wedge (s_{k-1} \vee \mathcal{M}(k-1))\\ &\wedge (s_1 \leftrightarrow \mathcal{S}(1))\\ &\wedge \bigwedge_{m\in\{3,5,\dots,k+1\}} (s_m \leftrightarrow (s_{m-1} \vee \mathcal{S}(m)))\\ &\wedge s_{k+1}\end{aligned}$$

Definition 4.2 (Geography Formulae). Let $G=(V,E)$ be a graph, $n=|V|$, $s \in V$ a node in the graph and let $k \in \mathbb{N}$ such that $k \bmod 2 = 0$. The *Geography Formulae* are the family of formulae containing QBF of the form

$$\mathcal{G}(G,k) = P_{n,k} : \mathcal{W}(G,k,s)$$

where $P_{n,k}$ and $\mathcal{W}(G,k,s)$ are the quantifier prefix and matrix defined in the construction above.

4.3.2 Formal version of the Geography Formulae

As usual, we begin by giving a name to the family, determining its format and declaring variables and parameters. This is rather straightforward.

```
    name: Geography Formulae;
    format: circuit-prenex;

    parameters: {
        n       : int, ‘n >= 1‘;
        edges   : list, ‘len(edges) == n‘;
        k       : int, ‘k >= 0‘, ‘k % 2 == 0‘;
        s       : int, ‘s in range(1, n+1)‘;
    }

    variables: {
        p(i, m)    where i in 1..n, m in 0..k;
        q(i, m)    where i in 1..n, m in 1..‘k-1‘;
        s(m)       where m in 1..‘k+1‘;
    }
```

Next we define the quantifier prefix. Recall the prefix we are defining is:

$$\exists p_{1,0} \dots p_{n,0} \exists s_1 \forall q_{1,1} \dots q_{n,1} \dots \; \forall q_{1,k-1} \dots q_{n,k-1} \exists p_{1,k} \dots p_{n,k} \exists s_{k+1}$$

In the formal version we split the prefix in the following blocks:

$$\texttt{Qp(m)} \; \texttt{:=} \; \exists p_{1,m}, \dots, \exists p_{n,m} \exists s_{m+1}$$

$$\texttt{Qq(m)} \; \texttt{:=} \; \forall q_{1,m}, \dots, \forall q_{n,m}$$

Then we combine them as follows:

```
blocks: {

    define blocks grouped in Qp {
        Qp(m) := p(i, m), s(m1);
    } where m in 0..k, ‘m % 2 == 0‘, i in 1..n, m1 = ‘m + 1‘;

    define blocks grouped in Qq {
        Qq(m) := q(i, m);
    } where m in 1..‘k-1‘, ‘m % 2 != 0‘, i in 1..n;

    all blocks in Qp quantified with E;
    all blocks in Qq quantified with A;
```

```
define blocks grouped in Qm {
    Q(m) := Qp(m), Qq(m1);
} where m in 0..‘k-2‘, ‘m % 2 == 0‘, m1 = ‘m+1‘;

define block Q := all blocks in Qm, Qp(k);
```

Now we build the blocks encoding the validity conditions, which amounts to splitting up every formula into blocks. The process is lengthy but does not present any new difficulties so we do not go into the details. The full code can be found in Appendix B.7.

Using this encoding and the tool presented next chapter, a wide variety of instances could be produced to test on different QBF solvers. In particular, it would be possible to test if solvers see the difference between cyclic and acyclic graphs. Unfortunately, this further research is out of the scope of this thesis and must remain as future work (see Chapter 6).

Chapter 5

The Tool: Defining and Building Formulae with QBDef

In Chapter 3, we designed and described a formal language to write QBF family definitions. This formal language lets us define all the formula families seen so far (such as the ones defined in Chapter 2) and many more. The ultimate goal, however, is not to write definitions but to be able to generate actual instances of these formulae to evaluate the performance of QBF solvers and shed light on proof complexity related issues.

In this chapter, we introduce QBDef, a computer tool based on the formal language we just described that can read definitions and output instances of the given family in formats accepted by current QBF solvers (QCIR and QDIMACS).

In Section 5.1 we present the tool and its most prominent features, as well as some very general technical details about its construction. In Section 5.2 we discuss how we go from the parse tree to an internal representation of the formulae, as well as what this internal representation looks like. Besides, we give some bounds on the complexity of generating the formulae and describe the possible output formats and their limitations. Finally, we discuss the evaluation of the tool: Section 5.3 discusses how to run the tool and what simple tests we have carried out to check the correctness of the outputs, while Section 5.4 uses QBDef to get instances of the Chen Formula of Type 2 and give a taste of what type of proof complexity research can be carried out with the tool, empirically showing an exponential separation between circuit-based and CNF-based solvers.

5.1 A brief introduction to QBDef

QBDef is a tool designed to easily write parameterized QBF family definitions and get instances in formats accepted by QBF solvers.

Consider, for instance, the QParity formulae, defined in Chapter 2 (see Section 2.3.3) and later formally written in our language in Chapter 3 (see Section 3.3.1). The QParity formulae have a single natural number n as a parameter. QBDef takes as input a file with the formal definition written in our language and a specific value for the parameter n and outputs a file (in either QDIMACS or QCIR) containing the QParity$_n$ formula.

This tool is a command-line script written in Python 3. The definitions of formula families are parsed based on the formal grammar described in Chapter 3 (see Appendix A for the full grammar) using the implementation of the LALR(1) parsing algorithm offered by the Lark parsing library[1] for Python. Once the formula is parsed, an internal representation of the QBF is built into a Python object. This abstract internal representation can then be easily converted to QCIR, QDIMACS or even (experimentally) to Non-Prenex-QCIR.

Because the tool is written in Python, the embedded language features described in Chapter 3 are handled directly by the Python interpreter, giving the user access to its built-in data structures and functions, including floating-point arithmetic, Booleans, lists, sets, dictionaries, mathematical functions and much more.

5.2 Representing and building the formulae

Once a definition file is parsed by `QBDef`, we convert it into a Python object to manipulate it internally. We now describe what this internal representation looks like.

5.2.1 Representation

We represent formulae in Python `QBF` objects. Objects of this class contain eight fields[2] storing the information given in the definition plus the values. These are:

- `name` (*string*)
  Name of the formula family.
- `format` (*enmumerated type* `Format`)
  A value of the enumerated type `Format` that can take the values `CNF`, `prenex-circuit` and `non-prenex-circuit`.
- `values` (*dictionary*)
  A dictionary relating the names of parameters to its values.
- `parameters` (*list*)
  A list o `Parameter` objects (a `Parameter` object contains the name of the parameter, its assigned values, the constraints it must satisfy and a list of Booleans indicating the evaluation results of these constraints).

[1]See `https://lark-parser.readthedocs.io/en/latest/`: "LALR(1) is a very efficient parsing algorithm, incredibly fast and requiring very little memory, capable of parsing most programming languages (e.g. Python and Java). Lark comes with an efficient implementation that outperforms every other parsing library for Python (including PLY) and extends the traditional YACC-based architecture with a contextual lexer, which automatically provides feedback from the parser to the lexer". Its code is available at `https://github.com/lark-parser/lark`.

[2]The implementation contains some additional fields for technical purposes. We exclude them from the discussion to keep it more readable and clear.

- `variables` (*dictionary*)
  A dictionary relating variable names as strings to numerical identifiers.

- `blocks` (*dictionary*)
  A dictionary relating block names as strings to numerical identifiers.

- `block_contents` (*dictionary*)
  A dictionary relating block identifiers to `Block` objects; a `Block` object contains its name, a list of the bricks it is composed of (—possibly negative— numerical identifiers of other blocks or variables), and an attribute (an enumerated type for existential/universal quantifiers and the `OR/AND/XOR` operators).

- `groupings` (*dictionary*)
  A dictionary relating grouping names (strings) to a list of numerical identifiers of blocks included in that grouping.

- `output` (*integer*)
  Numerical identifier of the output block.

As we can see, in terms of data structures, we define some auxiliary enumerated types and Python classes. The enumerated types are used to encode formula *format* choices (`CNF`, `prenex-circuit`, `non-prenex-circuit`) or *attributes* (`exists`, `forall`, `AND`, `OR`, `XOR`), while we have two other Python classes to encode *parameters* (containing their name, constraints, and values) and *blocks* (containing name, bricks and attribute).

The data structures used are quite straightforward and intuitive. Except for those enumerated types and auxiliary classes, we use built-in Python types only (integers, lists and dictionaries). Dictionaries are used to easily access variable identifiers, block identifiers, contents and groupings. This is because we often have to go from the definition given by the user (with string identifiers) to the numerical identifiers employed in the internal representation. Thus, being able to quickly get the identifiers and its contents is vital to generate the formulae efficiently.

To have an intuitive idea of what formulae look like in this internal representation, we take a look at the QParity formulae for $n = 3$.

Example 5.1 (Iternal representation of QParity$_3$)*.* The Python `QBF` object of QParity$_3$ and its fields would look more or less as follows, where `<...>` represents objects and `{...}` represents dictionaries.

- `name` (*string*)
  `name = "QParity Formulae"`

- `format` (*enmumerated type* `Format`)
  `format = circuit_PRENEX`

- `values` (*dictionary*)
  `values = {'n' : 3}`

- `parameters` (*list*)
  ```
  parameters = [<'n', 'int', 3, ['n >= 2'], [True]>]
  ```
- `variables` (*dictionary*)

  ```
  variables = {'x(1)' : 1,
               'x(2)' : 2,
               'x(3)' : 3,
               'z'    : 4}
  ```

- `blocks` (*dictionary*)

  ```
  blocks = {'X()'   : 5,
            'Z()'   : 6,
            'Q()'   : 7,
            'T(2)'  : 8,
            'T(3)'  : 9,
            'Ro1()' : 10,
            'Ro2()' : 11,
            'F()'   : 12,
            'Phi()' : 13}
  ```

- `block_contents` (*dictionary*)

  ```
  block_contents = {6  : <'X()',   [1, 2, 3], exists>,
                    7  : <'Z()',   [4],       forall>,
                    8  : <'Q()',   [5,6],     None>,
                    9  : <'T(2)',  [1, 2],    XOR>,
                    10 : <'T(3)',  [8, 3],    XOR>,
                    12 : <'Ro1()', [4, 9],    OR>,
                    13 : <'Ro2()', [-4, -9],  OR>,
                    14 : <'F()',   [10, 11],  AND>,
                    15 : <'Phi()', [7, 12],   None>}
  ```

- `groupings` (*dictionary*)
  ```
  groupings = {'T' : [8, 9], 'Ro' :  [10, 11]}
  ```
- `output` (*integer*)
  ```
  output := 13;
  ```

5.2.2 Building the formulae

The internal representation we chose is not particularly complicated and therefore going from the parse tree of the definition to the actual object is not very difficult. For instance, setting the name and format of the family or declaring the variables is quite straightforward. We have setter methods in our `QBF` class that receive the strings parsed from the file and take care of them, filling the appropriate fields or creating the necessary identifiers for variables. Similarly, the information on parameters coming from the parse tree is sent to a setter method that creates a `Parameter` object and then adds it to the list of parameters, making sure that the values and expressions are evaluated using Python's interpreter (Python's `eval(...)` and `exec(...)` handle this).

The main difficulty in building the formulae is in adding the blocks to the `QBF` object. The definition given as input contains one or more of the following blocks definitions,

```
define blocks {
    $B_1$(...) := $x_{1,1}$(...), ..., $x_{1,b_1}$(...);
        ⋮
    $B_k$(...) := $x_{k,1}$(...), ..., $x_{k,b_k}$(...);
} where $i_1$ in $r_1$..$r'_1$, ..., $i_c$ in $r_c$..$r'_c$;
```

where we may have up to k *line-defintions*, each one containing up to b_i bricks inside (we denote by b the largest b_i), and up to c indices $i_1, \ldots, i_c$ ranging over intervals $[r_1, r'_1]_{\mathbb{N}}$, ..., $[r_c, r'_c]_{\mathbb{N}}$ respectively.

We denote by R the set containing all the index tuples defined by the `where` clause:

$$R \subseteq [r_1, r'_1]_{\mathbb{N}} \times \cdots \times [r_c, r'_c]_{\mathbb{N}}$$

Note that we write R as the *subset* of that Cartesian product because amongst the conditions imposed in the `where` clause there might be Boolean conditions that could cancel out certain tuples.

Algorithm 1 presents the pseudocode for the procedure that processes a block definition like the one above. It corresponds to the notion of depth-first unfolding that we referred to as *fix-and-expand* in Chapter 3 when describing the language features (see Example 3.2).

Looking at Algorithm 1 and using the notation we just defined, we can give some bounds on the complexity of that procedure and therefore a bound on the general complexity of generating the formulae, as the rest of the formula-building steps are straightforward and do not add any significant overheads.

Clearly the algorithm has four nested loops and the operations inside are performed (almost) in constant time, so the algorithm processes a `define block` collection of line-definitions like the one presented before in time $O(k \cdot |R| \cdot b \cdot |R|)$. Because in practice we only have a fixed and small number of line-blocks and bricks (in the examples we saw so far we never have more than two or three line-definitions

Algorithm 1 Procedure adding a collection of line-blocks to the `QBF` object.

```
for def := 1 to $k$ do
    for all $(i_1, \dots, i_c) \in R$ do
        if $\mathtt{B}_{\text{def}}(i_1, \dots, i_c)$ is not defined then
            Assign a new identifier to this block.
            for brick := 1 to $b_{\mathtt{def}}$ do
                for all $(j_1, \dots, j_c) \in R$ do
                    Add the brick $\mathtt{x}_{\text{def,brick}}$ to the block. When substituting the
                    indices in the brick use first the values $i_1, \dots, i_c$ if those appeared
                    amongst the indices of $\mathtt{B}_{\text{def}}$, otherwise use $j_1, \dots, j_c$.
                end for
            end for
            Save the block with the bricks that have been added.
        end if
    end for
end for
```

and three or four bricks) while R will grow in size as the values of the parameters increase, we can safely state that $k, b \ll |R|$ and therefore the time-complexity of Algorithm 1 is $O(|R|^2)$.

Let us look at an example.

Example 5.2 (*Fix-and-expand* procedure). Take the following block definition, where we assume n and m to be the parameters of the family:

```
define blocks {
    A := x(i), y(i);
    B(i) := x(i), y(j);
} where i in 1..n, j in 1..m;
```

In this case we have $k = 2$ line-definitions and $b = 2$ bricks per line-definition. The set R of valued index-tuples is the whole Cartesian product, $R = [n] \times [m]$. The algorithm goes through each line-definition and for each of them it iterates first over R, then over the two bricks and for each brick it iterates over R again. The complexity is $O(|R|^2) = O(n^2 \cdot m^2)$.

5.2.3 Giving values to the parameters

So far, we have only talked about definitions, assuming the parameters had some values. Before we go on, we say something about assigning values to parameters. QBDef receives as input two files: a file containing a definition and a file containing values for the parameters. Each value is written in a new line, preceded by the word `value:`.

For example, in the Chromatic Formulae (see Sections 2.3.4, 3.3.2 and Appendix B.4), we had three parameters: `n`, `edges` and `k`. A possible value assignment written in our syntax could be:

```
value: n = 4;
value: edges = ‘[[0, 1, 1, 1],
                 [1, 0, 0, 1],
                 [1, 0, 0, 0],
                 [1, 1, 0, 0]]‘;
value: k = 3;
```

5.2.4 Output formats

Once we have the internal representation of the formula, we can decide to print it in a certain format that can be used by QBF solvers. This process is quite straightforward too, mainly because we only output QCIR formulae and then perform conversions on that format.

QCIR

This is the most basic way of getting an output in QBDef. Using our formula representation, it is quite easy to traverse the object as a Boolean circuit. We know which block denotes the *output* of the formula, so we need to follow the trace starting there, exploring the blocks in a breath-first manner. During this traversal we write down the blocks we find and their attributes as the gates of a Boolean circuit using the QCIR syntax. When we finish, we have a string with a QCIR representation of the formula.

QDIMACS

The QCIR format is already enough for many users and, in fact, is seems like the circuit representation makes solvers more efficient. However, QBDef supports outputs in CNF via some third-party conversion tools. We opt for external conversion tools because although our representation is good in that it is intuitive to understand the structure of the represented QBF and traverse it as a circuit, it is rather cumbersome to manipulate it.

To get the QDIMACS output, first we obtain the QCIR version of the formula as described above. Then we put this into William Klieber's `qcir-to-qdimacs` conversion tool[3], which outputs a CNF version of the formula written in QDIMACS.

Non-Prenex-QCIR

As we mentioned in Chapters 2 and 3, the latest version of the QCIR format defines language features to write non-prenex formulae. This is, however, rather unsatisfactory, as writing a non-prenex formula using these mechanisms behaves as using a completely different language. So much so that, at the moment of writing this work, solvers do not fully support this format.

Because our formal language can build non-prenex formulae, we have added the possibility to output Non-Prenex-QCIR, but this should be considered an

[3]See `https://www.wklieber.com/ghostq/qcir-converter.html`.

experimental feature, because, after all, there are no solvers against which to test the correctness of these outputs. In terms of conversion, the process is analogous to the one used in the conversion to QCIR.

As a last remark, we shall say something about writing non-prenex formulae in our language. We do so by adding pairs of blocks of the form `Q, G`, where `Q` is a quantifier-block and `G` is a gate block. We present a simple example, but we will not come back to this issue.

Example 5.3 (Writing non-prenex formulae). We will write in our formal language the non-prenex QBF

$$\Phi = \forall z : (\exists x : (x \oplus z)) \wedge (z \oplus (\exists x : (x \oplus z)))$$

We have the following gates: `G1 :=` $x \oplus z$, `G2 :=` $\exists x : (x \oplus z)$, `G3 :=` $z \oplus \exists x : (x \oplus z)$, `G4 :=` $\exists x : (x \oplus z)) \wedge (z \oplus (\exists x : (x \oplus z))$ and `G1 :=` Φ.

In our language, this is written as follows:

```
name: Non-Prenex Example;
format: circuit-nonprenex;

variables: {
    x;
    z;
}

blocks: {

    define block G1 := x, z;           define block G2 := XQ, G1;
    block G1 operated with XOR;        define block G3 := z, G2;
                                       block G3 operated with XOR;
    define block XQ := x;
    block XQ quantified with E;        define block G4 := G2, G3;
                                       block G4 operated with AND;
    define block ZQ := z;
    block ZQ quantified with A;        define block G5 := ZQ, G4;


}

output block: G5;
```

Using QBDef, the output in Non-Prenex-QCIR looks like this:

```
#QCIR-G14
output(9)
3 = xor(1, 2)
7 = xor(2, 6)
```

```
6 = exists(1; 3)
8 = and(6, 7)
9 = forall(2; 8)
```

5.3 Evaluation of QBDef

QBDef is a command-line Python script. More information on installing and using the tool can be found in Appendix C. The appendix also contains the link to the complete source code of the application.

When running the tool on a terminal, we get four possible outputs: a formula in QCIR, a formula in QDIMACS, a formula in NON-PRENEX-QCIR or a readable version of the internal representation, like the one presented earlier in this chapter.

Of course, how do we know the output formulae are correct? In other words, how do we know they correspond to the actual definitions given as input?

Though we have not used any formal verification techniques for correctness, we have performed three types of tests to ensure that the tool behaves as expected. The tests rely on the formal versions of the seven formula families defined in Chapter 2. The three types of tests are these:

1. Small correctness tests
   It has been manually checked that small instances correspond to what we can build by hand.

2. General UNSAT tests
   The Chen Formulae of Type 1 and 2, the QPARITY formulae, the Janota formulae and the KBKF formulae are always unsatisfiable, ragardless of the values of the parameters. We have tested that the outputs for a wide range of parameter values are unsatisfiable both in QCIR and QDIMACS by feeding these files to the QuAbS[4] and DepQBF[5] solvers.

3. SAT-UNSAT switching tests
   The Chromatic Formulae have a very nice property: their satisfiability depends on the parameters. We check that the satisfiability and unsatisfiability results switch correctly when changing the values of the parameters.

We now explain these tests in a bit more detail.

5.3.1 Small correctness tests

As a rudimentary and first step towards correctness, we have checked that small instances of the formulae correspond to what could be expected by checking that they match the ones we could build on pen an paper. This has been performed for

[4]`https://github.com/ltentrup/quabs`
[5]`https://lonsing.github.io/depqbf/`

all small families of formulae: QParity, Chen Type 2, Janota and KBKF (the rest of the families produce very big outputs even for small values of the parameters).

As an example, we can show the output of the QParity formulae for $n = 3$ in QCIR as given by QBDef. Looking at it and at the internal representation provided earlier (see Example 5.1), it is straightforward to see that it corresponds to what we expect from Definition 2.3:

```
#QCIR-G14
exists(1, 2, 3)   → ∃x1∃x2∃x3
forall(4)         → ∀z
output(12)        → QParity_n = ∃x1∃x2∃x3∀z : ρ
8 = xor(1, 2)     → t2 = x1 ⊕ x2
9 = xor(8, 3)     → t3 = t2 ⊕ x3
11 = or(-4, -9)   → ¬z ∨ ¬t3
10 = or(4, 9)     → z ∨ t3
12 = and(10, 11)  → ρ = (¬z ∨ ¬t3) ∧ (z ∨ t3)
```

For all the cases covered, the tool behaved as desired.

5.3.2 General UNSAT tests

Most of the formula families covered so far are always unsatisfiable. This is the case of the Chen Formulae of Type 1 and 2, the QParity formulae, the Janota formulae and the KBKF.

All of these families depend on a single natural value as parameter, and we have checked that the outputs produced by QBDef were evaluated unsatisfiable both in QCIR (with the circuit-based solver QuAbS) and in CNF (with the CNF solver DepQBF) for values $\{5, 10, 30, 50, 100, 150, 200, 300, 400, 500, 750, 1000\}$. In all situations, the formulae produced by QBDef were unsatisfiable in both of those solvers.

5.3.3 SAT-UNSAT switching tests

The general UNSAT tests just described do not confirm much, given that the produced outputs might be unsatisfiable for the wrong reasons. Fortunately, we count with a formula family where the satisfiability switches depending on the values of the parameters: the Chromatic Formulae.

The formula $\mathcal{K}(G, k)$ is true if and only if k is the chromatic number of G (and obviously G has only one such number). In the tests, we have used complete graphs of varying sizes (1, 5, 10, 20 and 40 nodes), as well as some smaller non-complete graphs. For the complete graphs, the chromatic number coincides with the number of nodes, so we can check if the formulas produced by the system are satisfiable for that value of k and unsatisfiable elsewhere.

We have performed this test both in QCIR (with the circuit-based solver QuAbS) and in CNF (with the CNF solver DepQBF). In all situations, the formulae produced by QBDef were satisfiable only for the correct values of k.

5.4 QBDef in action: testing the Chen Type 2 formulae

Across this work, we have motivated the existence of QBDef arguing that it would serve as a unified tool to develop empirical research in proof complexity. Although conducting that research is of course out of the scope of this work, we now present a small example of how the tool could be used to get significant proof complexity insights.

Take the Chen Formulae of Type 2, defined in Section 2.3.2 (Definition 2.2), where we saw how to build them as circuits using the auxiliary μ-circuits. That definition was then encoded into our formalism (see Appendix B.2) and we can now use QBDef to generate instances of them.

The Chen Formulae of Type 2 were used in [14] to show that a seemingly very powerful system called Relaxing QU-Resolution had an exponential lower bound caused by this family. But is this exponential lower bound translated to the actual solvers? What about solvers built on top of a completely different proof system?

In particular, me may ask the following two questions:

1. How long does it take for solvers to find the unsatisfiability of Type 2 formulae? That is, how hard are they for them?

2. Is there a significant difference in running time between solvers designed for circuits and solvers designed for CNF?

The second question is not a trivial one: the Chen Formulae of Type 2 are built as circuits and their transformation into CNF will require auxiliary variables that will be existentially quantified at the end of the prefix. As we mentioned in Chapter 2, it has been theoretically proven that depending on the proof system this transformation can provoke an exponential blowup of the proof length (see [9]). Is this the case of the Chen Formulae of Type 2 for some CNF-based solvers?

We start by running QBDef to obtain instances of the formulae for values $n = 1, 100, 200, 300, \ldots, 1000$. Once we have them, we try them on two different circuit-based solvers: QuAbS and CQESTO[6]. We have performed these tests on an Intel Core i5-8250U processor with 8 GB of RAM. This is a rather limited machine for performing large scale tests, and thus the experimental results gathered are a bit limited, but they are enough to get a taste of how this tool could be used by researchers.

Table 5.1 shows the average running time in seconds after three tests on each of those solvers (it can be appreciated that for QuAbS we have data up to $n = 1000$, but CQESTO was significantly slower and we had to stop at 600). These results are displayed on a graph on Figure 5.1.

Clearly, QuAbS outperforms CQESTO by far. Can we get an estimate of the complexity of these formulae for QuAbS? Figure 5.2 displays polynomial regressions for degrees 1 to 4 done with values from $n = 1$ to $n = 600$ and then extended up to 1000 to see how they fit.

[6]`http://sat.inesc-id.pt/~mikolas/sw/cqesto/`

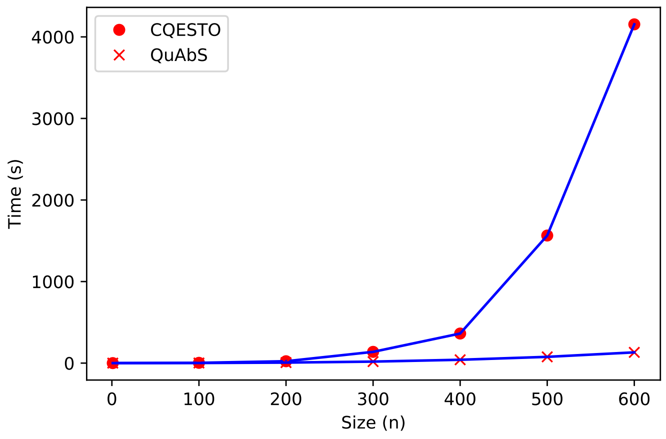

Figure 5.1: Chen Type 2 circuits' running times on QuAbS and CQESTO.

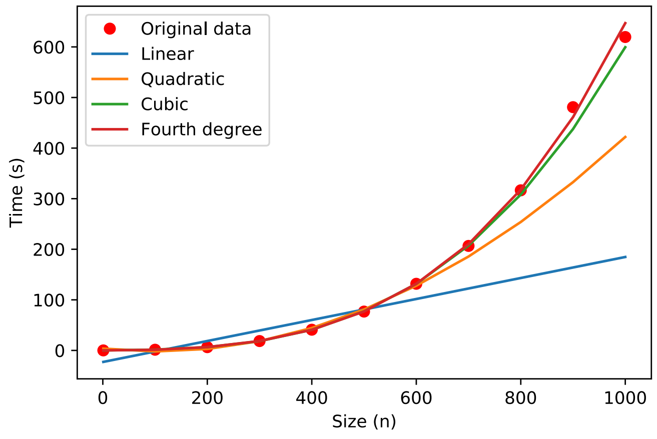

Figure 5.2: Polynomial regressions up to $n = 600$ on the QuAbS running times.

n	**QuAbS**	**CQESTO**
1	0.05061	0.12992
100	1.39349	3.48476
200	6.45269	23.03770
300	18.52676	137.92092
400	40.95977	362.74532
500	76.81426	1564.41455
600	131.73557	4153.90999
700	206.60374	-
800	316.54247	-
900	480.91833	-
1000	619.57376	-

TABLE 5.1: Average running time after three tests in seconds of the Chen Formulae of Type 2 on the QuAbS and CQESTO circuit-based solvers.

The linear and quadratic functions are easily discarded, as they are not very accurate. The cubic and the fourth degree ones are both close, but when we compare the latters' director coefficient we find that it is too small (10^{-7} versus 10^{-5}). Thus, we could conclude that the running times on QuAbS seem to grow at a cubic rate. Of course, the reader should note that deriving asymptotic growth rates based on empirical data is not precise at all. Too many factors interfere in the results and, besides, a very high-degree polynomial will always fit almost perfectly to any curve, yet that does not mean that running times follow that trend. Therefore, we must look for a balance between not too low director coefficients in the polynomials plus a quite accurate match with the data. That is why we discard the linear and quadratic regressions (too inaccurate) and we also discard the fourth-degree one (too low director coefficient). And yet, we should not consider this a proof that QuAbS is running in time $\Theta(n^3)$. However, this does let us say that despite the exponential lower bound proven for Relaxing QU-Resolution, when switching to a solver based on a different proof system, we can get very reasonable running times.

Let us now turn our attention to the second question: is there a significant difference in running time between solvers designed for circuits and solvers designed for CNF?

To answer this we try to perform a similar test on two CNF-based QBF solvers: DepQBF and CAQE[7]. Surprisingly, we will not get very far. In Table 5.2 and Figure 5.3 we can appreciate that running times are too high even for very small sizes. We cannot perform bigger tests.

As we see, it gets out of hand very quickly. When plotting the data, we see how an exponential curve fits neatly. For CAQE this curve is approximately $f(n) = 1.8538^n - 259.0903$, while for DepQBF it goes up a bit higher up to $f(n) = 1.9016^n - 393.3636$, almost a perfect match with a exponential curve of base 2. This clear exponential trend empirically confirms the results proven by Beyersdorf et al. in [9]: these

[7]`https://github.com/ltentrup/caqe`

n	**CAQE**	**depQBF**
1	0.10295	0.03602
2	0.04303	0.02145
3	0.04304	0.02372
4	0.04890	0.02330
5	0.05341	0.02648
6	0.08515	0.03009
7	0.22742	0.04562
8	0.62194	0.12410
9	1.87319	0.48458
10	7.42670	2.09502
11	31.11183	9.75327
12	108.41917	48.84733
13	355.90740	266.93340
14	1541.80189	1539.74230
15	6360.79237	9288.23431

Table 5.2: Chen Type 2 CNF formulae running times in seconds on CAQE and DepQBF.

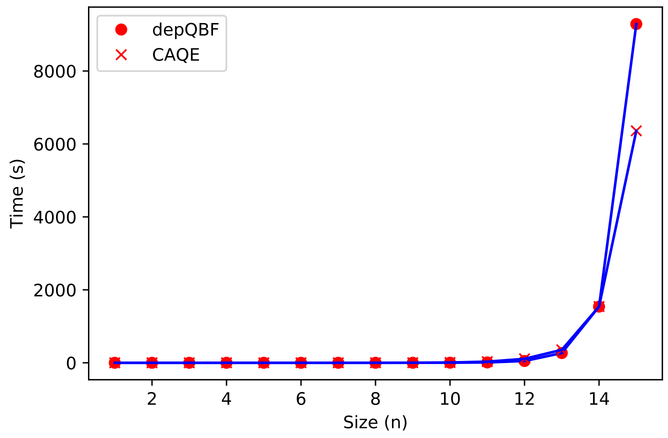

Figure 5.3: Average running time after three tests in seconds of the Chen Formulae of Type 2 on the DepQBF and CAQE CNF-based solvers.

CNF-based solvers cannot exploit certain patterns that are hidden in the Tseitin transformation.

We conclude the chapter by saying that QBDef helped greatly in this task: formula generation and format conversion was completely handled by the tool and we could focus on the data, the trends and the results. This is how the tool could be used in empirical research for proof complexity.

Chapter 6

Conclusion

In this work, we aimed at studying formula families in the QBF domain and ease their generation into popular computer formats through automated tools.

This goal has been achieved with a formal language in which to write formula family definitions and the implementation of QBDef, a computer tool capable of reading these definitions and outputting files with instances written in QCIR or QDIMACS.

We now summarise the results presented in this thesis.

Summary of the main results

In Chapter 2, we presented the concept of formula families and discussed it at length, providing examples of how these are used in the proof complexity literature to show exponential lower bounds, separations and other proof-theoretical results. We gave a short yet representative selection of formula families in the QBF domain, which covered a range of different formats, definitions styles and parameter data types. These were the Chen Formulae of Type 1 and 2 (Definitions 2.1 and 2.2), the QParity circuits (Definition 2.3), the Chromatic Formulae (Definition 2.4), the Janota Formulae (Definition 2.5) and the KBKF formulae (Definition 2.6).

Based on those definitions, in Chapter 3 we designed a formal language to capture them and potentially any other formula family. We explained its syntax, features and capabilities, and discussed some potential uses. We include the grammar of this language in Appendix A and the formal versions of all the formula families from Chapter 2 in Appendix B.

In Chapter 4 we looked at a **PSPACE**-complete two-player game: Generalized Geography, and we modelled it into QBF to then encode it into our language. This served to show how two-player games can be used to build interesting benchmarks for QBF solvers. The case of Generalized Geography is particularly attractive because of the way in which directed acyclic graphs behave. For these graphs, we provided an algorithm than can compute the existence of a winning strategy in polynomial time. This makes the game very attractive in that it would be interesting to see if QBF solvers can tell the difference between the formulae corresponding to an acyclic graph and the formulae corresponding to very similar yet cyclic graphs. Thanks to the encoding given in the language, generating the instances would be fairly easy.

Finally, in Chapter 5 we presented the main contribution of this work: QBDef, a tool parsing the language presented in Chapter 2 that can read formula family definitions and generate instances of them in the QCIR and QDIMACS formats. We briefly discussed the most interesting aspects of the implementation and described the evaluation and testing performed on the tool. Besides, we offered an example of real use based on the Chen Type 2 formulae, generating instances of them with QBDef to then try on four different solvers (DepQBF, CAQE, QuAbS and CQESTO). In particular, we showed how QBDef helped in getting certain insights on the complexity of this formulae, empirically pointing to an exponential separation between solvers depending on the format.

Final remarks and future work

We can safely conclude that this work achieved its main goal of providing a tool that could ease formula generation based on family definitions in the QBF domain. Given the interest in the tool showed by different people working in the field, we believe that this will be a useful addition to the toolkit of more empirically-oriented proof complexity theorists.

Naturally, this work pointed at many interesting lines of work that could be performed with QBDef. Besides, there exists a number of features the tool is still missing and certain lines of research that would be interesting to follow based on the work developed so far. We group these lines of future work in three main categories.

Additions to the language These include new operators and a more convenient support for non-prenex formula definitions.

- *New operators.* Currently the tool only allows conjunction, disjunction and exclusive disjunction as attributes for the blocks. The language would become a lot more convenient if implication, double implication, generalized XOR and other operators were included.

- *More support for non-prenex formulae.* Because of the way in which quantifier blocks and gate blocks are defined in the same space in the language, non-prenex formulae can be defined in our formalism. However, this is sometimes inconvenient and additional language constructs might ease this. Of course, this would need to go in hand with support of Non-Prenex-QCIR in the solvers, which will likely take some time.

New features for QBDef These include support for additional formats, better conversion tools, support for third-party Python packages, new testing techniques based on certificates and more detailed validity checking, as well as a more flexible implementation of the *fix-and-expand* procedure for better support of recursive nesting of blocks.

- *Support for additional formats.* Although QCIR and QDIMACS are the most popular format at the moment, other formalisms also exist. In particular, the DQDIMACS formats allows non-linear quantifier prefixes to write dependency quantified Boolean formulae (DQBF). Support for this would imply adding both language features and implementation of conversion tools to this new format.

- *More powerful conversion tools.* The QBF community is currently lacking strong and general conversion tools between formats. Although many solvers implement such conversion procedures as part of their code, they do not make them available as standalone applications. Developing these same features over and over again makes research slow and tedious. In particular, the main need is a consistent tool for QCIR-to-QDIMACS conversion, more efficient than the one developed by William Klieber and used in QBDef. Besides, development of such tools should go in hand with the theoretical research into the optimal placement of existential quantifiers at the end of prefixes in the conversion process (see next category).

- *Support for third-party Python packages.* With its built-in data types and functions, the embedded Python features of QBDef already make for an extremely powerful tool. However, even more could be done if we allowed external packages to be imported. This way, users could define their own data structures and functions and make some definitions easier.

- *Additional testing.* Given the scope of this thesis, the testing performed on the tool is limited. Although it is enough to convince us that the generated formulae correspond to the definitions in all formula families we covered, more testing would be desirable. Of course, once the tool is made available to the community, we expect to receive comments from users finding potential problems on formula families containing features we could not test. On the other hand, even with the formula families we have, more consistent testing could be carried out looking at certificates. Although not all solvers output certificates, some of them do, and this would be enough to check if the reasons for the unsatisfiability of formulae correspond to the definitions or not.

- *A more flexible implementation of* fix-and-expand. In Algorithm 1 we presented the procedure of *fix-and-expand* by which block definitions are unfolded and then saved into the internal representation. Some modifications could be made to the overall implementation to ideally make the running time less than quadratic in the size of $|R|$. Besides, because of the way in which blocks are added, some intricate recursive definitions might present problems. Certain adjustments to the procedure would be needed to make recursive nesting of blocks more convenient.

Research that could be conducted using QBDef This includes testing the Generalized Geography formulae, studying the separation provoked by normalisation techniques as well as improvements to conversion tools to avoid this.

- *Further study on* Generalized Geography. As we discussed in Chapter 4, the Generalized Geography formulae present a very nice property on acylic graphs that makes this problem an intersting one to test on solvers. Unfortuntaly, that research had to be left out of this thesis.
- *Research on the optimal placement of auxiliary variables in the quantifier.* As pointed out in [9], normalisation techniques when converting from circuits to CNF can provoke and exponential blowup in the proof lenghts in certain systems. We empirically observed this for the Chen Formulae of Type 2 generated with QBDef in Section 5.4. Currently, it is an open question what the optimal placement of existentially quantified auxiliary variables in the prefix should be when performing these conversions. It would be interesting to use QBDef to shed some light on this issue, by combining the tool with a versatile converter to see what placements present better results in different solvers.
- *Random generation of formulae.* Of course, we expect this tool will be used for generation of many different formula family ideas. In particular, it would be interesting to encode existing random generation models like the Chen-Interian model (see [15]) into our language, as well as some of the ideas we pointed at in Section 3.4.

Appendices

Appendix A

Formal grammar

The formal grammar of the language described in Chapter 3 and upon which QBDef is built, written in the Lark syntax for EBNF grammars.

```
start: value* formula_family? -> return_formula

value : "value:" NAME "=" expression ";" -> handle_value

formula_family : name format parameters? variables
                 blocks output_block

name: "name:" FAMILY_NAME ";" -> set_name

format: "format:" FORMAT ";" -> set_format

parameters: "parameters:" "{" parameter_declaration+ "}"

parameter_declaration: NAME ":" PARAM_TYPE
                       ("," expression)* ";" ->
                          add_parameter

variables : "variables:" "{" variable_declaration+ "}"

variable_declaration: NAME ("(" indices ")"
                      ("where" index_range
                          ("," index_range)*)?)? ";"
                      -> add_variable

indices : INDEX ("," INDEX)*

index_range : indices "in" (expression | INDEX | NAME)
                      ".." (expression | INDEX | NAME)
```

```
blocks: "blocks:" "{" block_definition (block_definition
                   | operator_declaration
                   |quantifier_declaration)+ "}"

block_definition:    "define block" single_block_def
                         -> add_blocks
                   | "define blocks" grouping?
                     "{" single_block_def+ "}" conditions?
                     ";" -> add_blocks

single_block_def: BLOCK_NAME ("(" indices ")")? ":="
                  block_body ";"

block_body: brick ("," brick)*

brick:  /all blocks in/ BLOCK_NAME
       | NEGATION? BLOCK_NAME ("(" indices ")")?
       | NEGATION? NAME ("(" indices ")")?

grouping: "grouped in" BLOCK_NAME

conditions: "where" condition ("," condition)*

condition: index_range | assignment | other_condition

assignment: INDEX "=" expression

other_condition: expression

quantifier_declaration: "block" BLOCK_NAME
                        ("(" indices ")")
                        "quantified with" QUANTIFIER ";"
                           -> add_attributes
                      | "blocks" BLOCK_NAME
                        ("(" indices ")")? (","
                           BLOCK_NAME
                        ("(" indices ")")?)+ "quantified
                           with"
                        QUANTIFIER ";"
                           -> add_attributes
                      | "all blocks in" BLOCK_NAME
                        "quantified with" QUANTIFIER ";"
                           -> add_attribute_to_grouping
```

```
operator_declaration:   "block" BLOCK_NAME
                        ("(" indices ")")?
                        "operated with" OPERATOR ";"
                         -> add_attributes
                      | "blocks" BLOCK_NAME ("(" indices
                         ")")?
                        ("," BLOCK_NAME ("(" indices ")")
                           ?)+
                        "operated with" OPERATOR ";"
                         -> add_attributes
                      | "all blocks in" BLOCK_NAME "
                         operated with"
                        OPERATOR ";"
                         -> add_attribute_to_grouping

output_block: "output block:" BLOCK_NAME
              ("(" indices ")")? ";"     ->
                 add_final_block

NAME : /(?!all blocks in)[a-z]([_?a-zA-Z0-9])*/
FAMILY_NAME : /[^;]+/ // old version: /[a-zA-Z]([ ?_?a-zA
   -Z0-9])*(?=;)/
FORMAT : "CNF" | "circuit-prenex" | "circuit-nonprenex"
PARAM_TYPE : "int" | "str" | "float" | "list" | "bool" |
   "other" // the names used by Python
?expression : /[0-9]+/ | "‘" /[^‘]+/ "‘" //| /[^‘,;\]]+/
INDEX : /[a-z][_?a-zA-Z0-9]*/ | /[0-9]+/
BLOCK_NAME : /[A-Z]([_?a-zA-Z0-9])*/
NEGATION : "-"
QUANTIFIER : "E" | "A"
OPERATOR: "AND" | "OR" | "XOR"
COMMENT: /\/\*((\*[^\/])|[^*])*\*\//


%import common.NUMBER
%import common.WS_INLINE
%import common.NEWLINE
%ignore WS_INLINE
%ignore NEWLINE
%ignore COMMENT
```

Appendix B

Encodings of the formula families

B.1 Chen Formulae of Type 1

```
name: Chen Type 1;
format: CNF;

parameters: {
    n : int, ‘n >= 1‘;
}

variables: {
    x1(i, j, k)       where i in 0..n, j, k in 0..1;
    x2(i, j, k)       where i in 1..n, j, k in 0..1;
    y(i)              where i in 1..n;
}


blocks:  {

    /*  ==== blocks for quantifers ==== */

    define blocks grouped in X1 {
        X1(i) := x1(i, j, j);
    } where i in 0..n, j, k in 0..1;

    define blocks grouped in X2 {
        X2(i) := x2(i, j, j);
    } where i in 1..n, j, k in 0..1;

    define blocks grouped in Y {
        Y(i) := y(i);
    } where i in 1.. n;

    define blocks grouped in X2YX1 {
        X2YX1(i) := X2(i), Y(i), X1(i);
    } where i in 1..n;

    define block Q := X1(0), all blocks in X2YX1;

    all blocks in X1 quantified with E;
    all blocks in X2 quantified with E;
    all blocks in Y  quantified with A;


    /* ==== blocks for formula ==== */

    define blocks grouped in B1 {
        B1(j, k) := -x1(0, j, k);
    } where j, k in 0..1;

    define blocks {
        B2 := x1(n, j, 0), x1(n, j, 1);
    } where j in 0..1;

    block B2 operated with OR;

    define blocks grouped in H {
        H(i, j) := -x2(i, 0, k), -x2(i, 1, l),
                        x1(s, j, 0), x1(s, j, 1);
    } where i in 1..n, j, k, l in 0..1, s = ‘i - 1‘;

    all blocks in H operated with OR;

    define blocks grouped in T {
        T1(i) := -x1(i, 0, k), y(i),  x2(i, 0, k);
        T2(i) := -x1(i, 1, k), -y(i), x2(i, 1, k);
    } where i in 1..n, k in 0..1;

    all blocks in T operated with OR;

    define block F := all blocks in B1, B2,
                      all blocks in H, all blocks in T;

    block F operated with AND;

    define block Phi := Q, F;
}

output block: Phi;
```

B.2 Chen Formulae of Type 2

```
name: Chen Type 2;
format: circuit-prenex;

parameters: {
    n : int, ‘n >= 1‘;
}

variables: {
    x(i)        where i in 1..n;
    y(i)        where i in 1..n;
}


blocks:  {

    /*  ==== blocks for quantifers ==== */

    define blocks grouped in X {
        X(i) := x(i);
    } where i in 1.. n;

    define blocks grouped in Y {
        Y(i) := y(i);
    } where i in 1.. n;

    all blocks in X quantified with E;
    all blocks in Y quantified with A;

    define blocks grouped in Qi {
        Q(i) := X(i), Y(i);
    } where i in 1..n;

    define block Q := all blocks in Qi;


    /* ==== blocks for formula ==== */

    define blocks grouped in Ones0 {
        Ones(0, i) := -x(i), -y(i);
    } where i in 1..n;

    define blocks grouped in Ones1 {
        Ones(1, i) := x(i), y(i);
    } where i in 1..n;

    define blocks grouped in Ones2 {
        Ones(2, i) := x(i), y(i);
    } where i in 1..n;

    all blocks in Ones0 operated with AND;
    all blocks in Ones1 operated with XOR;
    all blocks in Ones2 operated with AND;

    define blocks IsAndAdds {
        IsAndAdds(i, p, a) := Mu(i, p), Ones(a, k);
    } where i in 1..n, p, a in 0..2;

    all blocks in IsAndAdds operated with AND;

    define block Mu(0, 1) := Ones(0, 1);
    define block Mu(1, 1) := Ones(1, 1);
    define block Mu(2, 1) := Ones(2, 1);

    define blocks grouped in Mu {
        Mu(m, i) := IsAndAdds(k-1, p, a);
    } where m in 0..2, i in 2..n, p in 0..2,
                        a = ‘(m-p) % 3‘;

    define blocks {
        F := -Mu(s, n);
    } where s = ‘n % 3‘;

    define block Phi := Q, F;
}

output block: Phi;
```

B.3 QParity Formulae

```
name: QParity;
format: circuit-prenex;

parameters: {
    n : int, `n >= 1`;
}

variables: {
    x(i)    where i in 1..n;
    z;
}

blocks: {

    /* === Blocks for quantifers === */

    define blocks {
        X := x(i);
    } where i in 1..n;

    define block Z := z;

    define block Q := X, Z;

    block X quantified with E;
    block Z quantified with A;

    /* === Blocks for matrix === */

    define block T(2) := x(1), x(2);
    define blocks grouped in T {
        T(i) := T(s), x(i);
    } where i in 3..n, s = `i-1`;

    define block Rho := T(n), z;

    block T(2) operated with XOR;
    all blocks in T operated with XOR;
    block Rho operated with XOR;

    define block Phi := Q, Rho;
}

output block: Phi;
```

B.4 Chromatic Formulae

```
name: Chromatic Formulas;
format: circuit-prenex;

parameters: {
    n     : int, `n >= 1`;
    edges : list, `len(edges) == n`;
    k     : int, `k >= 1`;
}

variables: {
    x(i, j)    where i in 1..n, j in 1..k;
    y(i, j)    where i in 1..n, j in 1..`k-1`;
}

blocks: {

    /* === blocks for quantifers === */

    define blocks grouped in X {
        X(i) := x(i, j);
    } where i in 1..n, j in 1..k;

    define blocks grouped in Y {
        Y(i) := y(i, j);
    } where i in 1..n, j in 1..`k-1`;

    define block Q := all blocks in X,
                      all blocks in Y;

    all blocks in X quantified with E;
    all blocks in Y quantified with A;

    /* ==== blocks for matrix ==== */

    define blocks grouped in AllColored {
        Colored(i) := x(i, j);
    } where i in 1..n, j in 1..k;

    define blocks grouped in NotColored {
        NotColored(i) := -y(i, j);
    } where i in 1..n, j in 1..`k-1`;

    define block Gamma1 := all blocks in AllColored;

    define block Delta1 := all blocks in NotColored;

    define blocks grouped in SubGamma2 {
        SG2(i, j, l) := -x(i, j), -x(i, l);
    } where i in 1..n, j in 1..k,
                          l in 1..k, `j != l`;

    define blocks grouped in SubDelta2 {
        SD2(i, j, l) := y(i, j), y(i, l);
    } where i in 1..n, j in 1..`k-1`,
                          l in 1..`k-1`, `j != l`;

    define block Gamma2 := all blocks in SubGamma2;

    define block Delta2 := all blocks in SubDelta2;

    define blocks grouped in SubGamma3 {
        SG3(i, j, l) := -x(i, l), -x(j, l);
    } where i in 1..n, j in 1..n,
            `edges[i-1][j-1] == 1`, l in 1..k;

    define blocks grouped in SubDelta3 {
        SD3(i, j, l) := y(i, l), y(j, l);
    } where i in 1..n, j in 1..n,
            `edges[i-1][j-1] == 1`, l in 1..`k-1`;

    define block Gamma3 := all blocks in SubGamma3;

    define block Delta3 := all blocks in SubDelta3;

    define block Gamma := Gamma1, Gamma2, Gamma3;

    define block Delta := Delta1, Delta2, Delta3;

    define block F := Gamma, Delta;

    all blocks in AllColored operated with OR;
    all blocks in NotColored operated with AND;

    all blocks in SubGamma2 operated with OR;
    all blocks in SubDelta2 operated with AND;

    all blocks in SubGamma3 operated with OR;
    all blocks in SubDelta3 operated with AND;

    blocks Gamma1, Gamma2, Gamma3 operated with AND;
    block Gamma operated with AND;

    blocks Delta1, Delta2, Delta3 operated with OR;
    block Delta operated with OR;

    block F operated with AND;

    /* define the output block */
    define block Phi := Q, F;
}

output block: Phi;
```

B.5 Janota Formulae

```
name: Janota Formulae;
format: CNF;

parameters: {
    n : int, `n >= 1`;
}

variables: {
    x(i, j) where i, j in 1..n;
    a(i)    where i in 1..n;
    b(i)    where i in 1..n;
    z;
}

blocks: {

    /* === blocks for quantifers === */

    define blocks {
        X := x(i, j);
    } where i, j in 1..n;

    define block Z := z;

    define blocks {
        L := a(i), b(i);
    } where i in 1..n;


    define block Q := X, Z, L;

    block X quantified with E;
    block Z quantified with A;
    block L quantified with E;

    /* ==== blocks for matrix ==== */

    define blocks grouped in B_grp {
        B1(i, j) := x(i, j), z, a(i);
        B2(i, j) := -x(i, j), -z, b(i);
    } where i, j in 1..n;

    define blocks grouped in Or_grp {
        A := -a(i);
        B := -b(i);
    } where i in 1..n;


    define block F := all blocks in B_grp, all blocks in
       Or_grp;
    all blocks in Or_grp operated with OR;
    all blocks in B_grp operated with OR;

    block F operated with AND;

    /* define the output block */
    define block Phi := Q, F;

}

output block: Phi;
```

B.6 KBKF Formulae

```
name: FAST KBKF;
format: CNF;

parameters: {
    t : int, `t >= 1`;
}

variables: {
    x(i)        where i in 1..t;

    y(0);
    y(i, j)     where i in 1..t, j in 0..1;
    y(i)        where i in `t+1`..`t+t`;
}


blocks:  {

    /*  ==== blocks for quantifers ==== */

    define blocks grouped in X {
        X(i) := x(i);
    } where i in 1..t;

    define block Y(0) := y(0);

    define blocks grouped in Y {
        Y(i) := y(i, 0), y(i, 1);
    } where i in 1..t;

    define blocks {
        YRest := y(j);
    } where j in `t+1`..`t+t`;

    define blocks grouped in Pairs {
        Pair(i) := Y(i), X(i);
    } where i in 1..t;

    define block Q := Y(0), all blocks in Pairs, YRest;

    block Y(0) quantified with E;
    all blocks in X quantified with A;
    all blocks in Y quantified with E;
    block YRest quantified with E;

    /* ==== blocks for formula ==== */

    define block CMinus := -y(0);

    define block C(0) := y(0), -y(1, 0), -y(1,1);

    define blocks grouped in C1 {
        C(i, j) := y(i, j), x(i),
                   -y(s1, 0), -y(s1, 1);
    } where i in 1..‘t-1‘, j in 0..1, s1 = ‘i+1‘;

    define blocks grouped in C2 {
        C(t, j) := y(t, j), x(t), -y(k);
    } where j in 0..1, k in ‘t+1‘..‘t+t‘;


    define blocks grouped in C3 {
        C(s2, 0) := x(l), y(s2);
        C(s2, 1) := -x(l), y(s2);
    } where l in 1..t, s2 = ‘t+l‘;

    define block F := CMinus, C(0),
                      all blocks in C1,
                      all blocks in C2,
                      all blocks in C3;

    block C(0) operated with OR;
    all blocks in C1 operated with OR;
    all blocks in C2 operated with OR;
    all blocks in C3 operated with OR;

    block F operated with AND;

    define block Phi := Q, F;
}

output block: Phi;
```

B.7 Geography Formulae

```
name: Geography Formulae;
format: circuit-prenex;

parameters: {
    n       : int, ‘n >= 1‘;
    edges   : list, ‘len(edges) == n‘;
    k       : int, ‘k >= 0‘, ‘k % 2 == 0‘;
    s       : int, ‘s in range(1, n+1)‘;
}

variables: {
    p(i, m)     where i in 1..n, m in 0..k;
    q(i, m)     where i in 1..n, m in 1..‘k-1‘;
    s(m)        where m in 1..‘k+1‘;
}

blocks: {

        /* ==== Blocks for the quantifier === */

    define blocks grouped in Qp {
        Qp(m) := p(i, m), s(m1);
    } where m in 0..k, ‘m % 2 == 0‘, i in 1..n, m1 = ‘m +
        1‘;

    define blocks grouped in Qq {
        Qq(m) := q(i, m);
    } where m in 1..‘k-1‘, ‘m % 2 != 0‘, i in 1..n;

    all blocks in Qp quantified with E;
    all blocks in Qq quantified with A;

    define blocks grouped in Qm {
        Q(m) := Qp(m), Qq(m1);
    } where m in 0..‘k-2‘, ‘m % 2 == 0‘, m1 = ‘m+1‘;

    define block Q := all blocks in Qm, Qp(k);

    /* ----------------------------------------*/





    /* ==== Blocks for the matrix
       ========================================== */

    /* === 1. Validity conditions
       ========================================== */

    /* (a) Initial conditions (no need for blocks) */
    /* ----------------------------------------*/

    /* (b) Uniqueness of choice   */
    /* ---------------------------------------*/


    /* ---------------------------------------*/

    define blocks grouped in NotChosenOtherP {
        NotChosenOtherP(i, m) := -p(j, m);
    } where i in 1..n, m in 0..k, ‘m % 2 == 0‘,
                         j in 1..n, ‘j != i‘;

    all blocks in NotChosenOtherP operated with AND;

    define blocks grouped in LeftImpP {
        LeftImpP(i, m) := -p(i, m), NotChosenOtherP(i, m)
            ;
    } where i in 1..n, m in 0..k, ‘m % 2 == 0‘;

    define blocks grouped in RightImpP {
        RightImpP(i, m) := -NotChosenOtherP(i, m), p(i, m
            );
    } where i in 1..n, m in 0..k, ‘m % 2 == 0‘;

    all blocks in LeftImpP operated with OR;
    all blocks in RightImpP operated with OR;

    define blocks grouped in UniqueChoiceP {
        UniqueChoiceP(i, m) := LeftImpP(i, m), RightImpP(
            i, m);
    } where i in 1..n, m in 0..k, ‘m % 2 == 0‘;

    all blocks in UniqueChoiceP operated with AND;

    define blocks grouped in Up {
        Up(m) := UniqueChoiceP(i, m);
    } where m in 0..k, ‘m % 2 == 0‘, i in 1..n;

    all blocks in Up operated with AND;

    /* ---------------------------------------*/

    define blocks grouped in NotChosenOtherQ {
        NotChosenOtherQ(i, m) := -q(j, m);
    } where i in 1..n, m in 1..‘k-1‘, ‘m % 2 != 0‘,
                        j in 1..n, `j != i`;

    all blocks in NotChosenOtherQ operated with AND;

    define blocks grouped in LeftImpQ {
        LeftImpQ(i, m) := -q(i, m), NotChosenOtherQ(i, m)
            ;
    } where i in 1..n, m in 1..`k-1`, `m % 2 != 0`;

    define blocks grouped in RightImpQ {
        RightImpQ(i, m) := -NotChosenOtherQ(i, m), q(i, m
            );
    } where i in 1..n, m in 1..`k-1`, `m % 2 != 0`;

    all blocks in LeftImpQ operated with OR;
    all blocks in RightImpQ operated with OR;

    define blocks grouped in UniqueChoiceQ {
        UniqueChoiceQ(i, m) := LeftImpQ(i, m), RightImpQ(
            i, m);
    } where i in 1..n, m in 1..`k-1`, `m % 2 != 0`;

    all blocks in UniqueChoiceQ operated with AND;

    define blocks grouped in Uq {
        Uq(m) := UniqueChoiceQ(i, m);
    } where i in 1..n, m in 1..`k-1`, `m % 2 != 0`;

    all blocks in Uq operated with AND;

    /* (c) No Overlapping */
    /* ---------------------------------------*/

    /* ---------------------------------------*/

    define blocks grouped in NotUsedBeforeP_Op {
        NotUsedBeforeP_Op(i, m) := -p(i, m1);
    } where i in 1..n, m in 2..k, `m % 2 == 0`, m1 in
        0..`m-2`, `m1 % 2 == 0`;

    define blocks grouped in NotUsedBeforeQ_Op {
        NotUsedBeforeQ_Op(i, m) := -q(i, m1);
    } where i in 1..n, m in 2..k, `m % 2 == 0`, m1 in
        1..`m-1`, `m1 % 2 != 0`;

    all blocks in NotUsedBeforeP_Op operated with AND;
    all blocks in NotUsedBeforeQ_Op operated with AND;

    define blocks grouped in NotUsedBeforeP {
        NotUsedBeforeP(i, m) := NotUsedBeforeP_Op(i, m),
           NotUsedBeforeQ_Op(i, m);
    } where i in 1..n, m in 2..k, `m % 2 == 0`;

    all blocks in NotUsedBeforeP operated with AND;

    define blocks grouped in OverlapP {
        OverlapP(i, m) := -p(i, m), NotUsedBeforeP(i, m);
    } where i in 1..n, m in 2..k, `m % 2 == 0`;

    all blocks in OverlapP operated with OR;

    define blocks grouped in Op {
        Op(m) := OverlapP(i, m);
    } where m in 2..k, `m % 2 == 0`, i in 1..n;
    all blocks in Op operated with AND;

    /* ----------------------------------------*/

    define blocks grouped in NotUsedBeforeP_Oq {
        NotUsedBeforeP_Oq(i, m) := -p(i, m1);
    } where i in 1..n, m in 1..`k-1`, `m % 2 != 0`, m1 in
        0..`m-1`, `m1 % 2 == 0`;

    define blocks grouped in NotUsedBeforeQ_Oq {
        NotUsedBeforeQ_Oq(i, m) := -q(i, m1);
    } where i in 1..n, m in 1..`k-1`, `m % 2 != 0`, m1 in
        1..`m-1`, `m1 % 2 != 0`;

    all blocks in NotUsedBeforeP_Oq operated with AND;
    all blocks in NotUsedBeforeQ_Oq operated with AND;

    define blocks grouped in Special {
        NotUsedBeforeQ(i, 1) := NotUsedBeforeP_Oq(i, 1);
    } where i in 1..n;
    all blocks in Special operated with AND;

    define blocks grouped in NotUsedBeforeQ {
        NotUsedBeforeQ(i, m) := NotUsedBeforeP_Oq(i, m),
           NotUsedBeforeQ_Oq(i, m);
    } where i in 1..n, m in 3..`k-1`, `m % 2 != 0`;

    all blocks in NotUsedBeforeQ operated with AND;

    define blocks grouped in OverlapQ {
        OverlapQ(i, m) := -q(i, m), NotUsedBeforeQ(i, m);
    } where i in 1..n, m in 1..`k-1`, `m % 2 != 0`;

    all blocks in OverlapQ operated with OR;

    define blocks grouped in Oq {
        Oq(m) := OverlapQ(i, m);
    } where m in 1..`k-1`, `m % 2 != 0`, i in 1..n;
    all blocks in Oq operated with AND;


    /* (d) Connectedness */
    /* ---------------------------------------*/

    define blocks grouped in AdjacentP {
        AdjacentP(i, m) := p(j, m);
    } where i, j in 1..n, `edges[i-1][j-1] == 1`,
                  m in 2..k, `m % 2 == 0`;

    define blocks grouped in AdjacentQ {
        AdjacentQ(i, m) := q(j, m);
    } where i, j in 1..n, `edges[i-1][j-1] == 1`,
                  m in 1..`k-1`, `m % 2 != 0`;

    all blocks in AdjacentP operated with OR;
    all blocks in AdjacentQ operated with OR;

    define blocks grouped in ConnectedP {
        ConnectedP(i, m) := -q(i, m0), AdjacentP(i, m);
    } where i in 1..n, m in 2..k, `m % 2 == 0`, m0 = `m
       -1`;

    define blocks grouped in ConnectedQ {
        ConnectedQ(i, m) := -p(i, m0), AdjacentQ(i, m);
    } where i in 1..n, m in 1..`k-1`, `m % 2 != 0`, m0 =
       `m-1`;

    all blocks in ConnectedP operated with OR;
    all blocks in ConnectedQ operated with OR;

    define blocks grouped in Cp {
        Cp(m) := ConnectedP(i, m);
    } where m in 2..k, ‘m % 2 == 0‘, i in 1..n;

    define blocks grouped in Cq {
        Cq(m) := ConnectedQ(i, m);
    } where m in 1..‘k-1‘, ‘m % 2 != 0‘, i in 1..n;

    all blocks in Cp operated with AND;
    all blocks in Cq operated with AND;


    /* === 2. Stuckness
        ===============================================================
        */

    define blocks grouped in EdgeUsedP1 {
        EdgeUsedP(i, j, 1) := p(j, 0);
    } where i, j in 1..n, ‘edges[i-1][j-1] == 1‘;
    all blocks in EdgeUsedP1 operated with OR;

    define blocks grouped in EdgeUsedP {
        EdgeUsedP(i, j, m) := p(j, m1);
    } where i, j in 1..n, ‘edges[i-1][j-1] == 1‘,
                    m  in 3..‘k+1‘, ‘m  % 2 != 0‘,
                    m1 in 0..‘m-3‘, ‘m1 % 2 == 0‘;

    all blocks in EdgeUsedP operated with OR;

    define blocks grouped in EdgeUsedQ {
        EdgeUsedQ(i, j, m) := q(j, m1);
    } where i, j in 1..n, ‘edges[i-1][j-1] == 1‘,
                    m  in 1..‘k+1‘, ‘m  % 2 != 0‘,
                    m1 in 0..‘m-2‘, ‘m1 % 2 != 0‘;

    all blocks in EdgeUsedQ operated with OR;

    define blocks grouped in Special2 {
        EdgeUsed(i, j, 1) := EdgeUsedP(i, j, 1);
    } where i, j in 1..n, ‘edges[i-1][j-1] == 1‘;


    define blocks grouped in EdgeUsed {
        EdgeUsed(i, j, m) := EdgeUsedP(i, j, m),
           EdgeUsedQ(i, j, m);
    } where i in 1..n, j in 1..n, ‘edges[i-1][j-1] == 1‘,
                         m  in 1..‘k+1‘, ‘m  % 2 != 0‘;

    all blocks in EdgeUsed operated with OR;

    define blocks grouped in NoWayOutFrom {
        NoWayOutFrom(i, m) := EdgeUsed(i, j, m);
    } where i in 1..n, j in 1..n, ‘edges[i-1][j-1] == 1‘,
        m in 1..‘k+1‘, ‘m % 2 != 0‘;
    all blocks in NoWayOutFrom operated with AND;

    define blocks grouped in StuckIn {
        StuckIn(i, m) := p(i, m1), NoWayOutFrom(i, m);
    } where i in 1..n, m in 1..‘k+1‘, ‘m % 2 != 0‘, m1 =
       ‘m-1‘;

    all blocks in StuckIn operated with AND;

    define blocks grouped in S {
        S(m) := StuckIn(i, m);
    } where m in 1..‘k+1‘, ‘m % 2 != 0‘, i in 1..n;

    all blocks in S operated with OR;


    /* === 3. Final blocks
       =================================================
        */


    define block Vp(0) := p(s, 0), Up(0);
    block Vp(0) operated with AND;

    define blocks grouped in Vp {
        Vp(m) := Up(m), Op(m), Cp(m);
    } where m in 2..‘k‘, ‘m % 2 == 0‘;
    all blocks in Vp operated with AND;

    define blocks grouped in Vq {
        Vq(m) := Uq(m), Oq(m), Cq(m);
    } where m in 1..‘k-1‘, ‘m % 2 != 0‘;
    all blocks in Vq operated with AND;

    define blocks grouped in VpUpTo {
        VpUpTo(m) := Vp(m1);
    } where m in 1..`k-1`, `m % 2 != 0`, m1 in 0..`m -
        1`, `m1 % 2 == 0`;

    define blocks grouped in VqUpTo {
        VqUpTo(m) := Vq(m1);
    } where m in 2..`k-1`, `m % 2 != 0`, m1 in 0..`m -
        2`, `m1 % 2 != 0`;

    all blocks in VpUpTo operated with AND;
    all blocks in VqUpTo operated with AND;

    define block VUpTo(1) := VpUpTo(1);

    define blocks grouped in VUpTo {
        VUpTo(m) := VpUpTo(m), VqUpTo(m);
    } where m in 3..`k-1`, `m % 2 != 0`;

    all blocks in VUpTo operated with AND;

    define blocks grouped in ValidNow {
        ValidNow(m):= Vq(m), VUpTo(m);
    } where m in 1..`k-1`, `m % 2 != 0`;

    define blocks {
        ValidAndStuck := Vp(k), S(k1);
    } where k1 = `k + 1`;
    block ValidAndStuck operated with AND;

    define blocks grouped in Mk1 {
        M(k1) := -ValidNow(k1), ValidAndStuck;
    } where k1 = `k-1`;
    all blocks in Mk1 operated with OR;

    define blocks grouped in M {
        M(m) := -ValidNow(m), Vp(m1);
    } where m in 1..`k-3`, `m % 2 != 0`, m1 = `m + 1`;
    all blocks in M operated with OR;

    define blocks grouped in StuckOrMove {
        StuckOrMove(m) := s(m), M(m);
    } where m in 1..`k-1`, `m % 2 != 0`;
    all blocks in StuckOrMove operated with OR;


    define blocks grouped in NowOrBefore {
        NowOrBefore(m) := s(m0), S(m);
    } where m in 3..‘k+1‘, ‘m % 2 != 0‘, m0 = ‘m - 2‘;
    all blocks in NowOrBefore operated with OR;

    define blocks grouped in LeftS {
        LeftS(m) := -s(m), NowOrBefore(m);
    } where m in 3..‘k+1‘, ‘m % 2 != 0‘, m0 = ‘m - 1‘;
    define blocks grouped in RightS {
        RightS(m) := NowOrBefore(m), -s(m);
    } where m in 3..‘k+1‘, ‘m % 2 != 0‘, m0 = ‘m - 1‘;

    all blocks in LeftS operated with OR;
    all blocks in RightS operated with OR;

    define block S1ImpL := -s(1), S(1);
    define block S1ImpR := S(1), -s(1);
    block S1ImpL operated with OR;
    block S1ImpR operated with OR;

    define block S1Imp := S1ImpL, S1ImpR;
    block S1Imp operated with AND;

    define blocks {
        SImp := LeftS(m), RightS(m), S1Imp;
    } where m in 3..‘k+1‘, ‘m % 2 != 0‘;
    block SImp operated with AND;

    define blocks {
        SK1 := s(k1);
    } where k1 = ‘k + 1‘;

    define blocks {
        F := Vp(0), StuckOrMove(m), SK1, SImp;
    } where m in 1..‘k-1‘, ‘m % 2 != 0‘;
    block F operated with AND;


    /* --- Define the output block --- */
    define block G := Q, F;

}

output block: G;
```

Appendix C

Source code, installation and use of QBDef

QBDef is a command-line tool working on Python 3. The source code and information on how to install and run the tool can be found in the GitHub repository of this project:

`https://github.com/alephnoell/QBDef`

On Linux, the tool can be run by executing the `QBDef.py` script on a terminal:

```
python3 QBDeF.pyc input_file [-internal] [-verbose]
                             [-QDIMACS {file.qdimacs | [-stdIO]}]
                             [-QCIR {file.QCIR | [-stdIO]}]
                             [-non-prenex-QCIR {file.QCIR | [-stdIO]}]
```

For example, if `my_def.txt` is the QBF family definition and `values.txt` is the file with the values for the parameters,

```
python3 QBDeF.pyc my_def.txt values.txt -QCIR
```

outputs the QCIR format instance on terminal.
The possible options are:

- `-QCIR [output_file]`: outputs a QCIR, if no output file is provided, it is printed in the standard output.
- `-QDIMACS [output_file]`: outputs a QDIMACS, if no output file is provided, it is printed in the standard output.
- `-non-prenex-QCIR [output_file]`: outputs a non-prenex QCIR. This feature is experimental.

- `-internal`: outputs a human-readable version of the internal representation of the QBF.
- `-verbose`: prints messages while parsing and processing the definition.

Appendix D

Paper

In the following pages a summary paper of this project is attached. This paper was presented at the 2020 edition of the QBF Workshop, in the context of the 2020 SAT Conference, as a preliminary report on the work presented in this thesis.

A Formal Language for QBF Family Definitions

Noel Arteche Echeverría[1,2] and Matthias van der Hallen[2]

[1] University of the Basque Country, Faculty of Computer Science
Manuel Lardizabal 1, 20018 Donostia / San Sebastián, Spain
narteche002@ikasle.ehu.eus
[2] KU Leuven, Department of Computer Science
Celestijnenlaan 200A, 3001 Heverlee (Leuven), Belgium
noel.artecheecheverria@student.kuleuven.be
matthias.vanderhallen@kuleuven.be

Abstract. In this work we propose a formal language to write definitions of classes of Quantified Boolean Formulae (QBF) in terms of —potentially— any type of parameters. A class of formulae provides an encoding in logic terms of some computational problem, and these definitions of families of formulae usually depend on some parameters determining the size, structure, alternation patterns of quantifiers and complexity of the described formulae. These parameters and their relation to the structure of formulae can be easily encoded in this language. Additionally, we present QBDef, a computer tool capable of parsing these definitions and outputting the formulae in either the QCIR or QDIMACS formats to be fed to a QBF solver. This aims to be both a framework and a tool for future empirical research in these topics.

Keywords: QBF · Formal language · Proof complexity · PSPACE · QCIR · QDIMACS

1 Introduction

Many QBF solvers can be considered to perform a heuristic search for a proof in some proof system. Consequently, studying the proof complexity of these systems provides insights into their strengths and weaknesses. The theoretical research often proceeds by defining classes of formulae or *formula families* to then show proof-theoretic lower bounds on them. In practice, the definition of a formula family declares the structure of the formulae contained in that set. One can think of a formula family definition as the encoding of a PSPACE problem into QBF. Problems outside PSPACE might be encoded in QBF too, but (probably) not in polynomial time.

To check whether the theoretical results on proof systems apply to the QBF solvers built on top, a tool to ease the generation of instances from formula families has been missing. In this extended abstract, we present the work in progress for a formal language to define formula families and QBDef, a computer tool capable of reading them and, instantiating concrete parameter values, output a file that can be fed to a QBF solver.

Although most of the existing formula families in the literature depend on a single scalar value, the language presented in this paper supports virtually any data structure for parameter types via the use of embedded Python, e.g. graphs.

Naturally, any programming language could be used for this same purpose, by means of a script that generated instances of specific formula families. However, these family-specific scripts work directly with the formulae written in the final formats. Although this is acceptable if we are interested only in a particular family, the tool presented in this work (QBDef) lets users focus on the formulae, without having to worry about lower-level details and formats, and encourages to playfully come up with inventive hard-to-prove formulae while, at the same, brings QBF modelling tools closer to the non-QBF-expert.

2 Formula Families

In this context, a *formula family* or *class of formulae* is a set of QBF all presenting the same structure and meaning. In particular, we are interested in dealing with a formula family's *definition*.

The formula families used in the proof complexity literature are usually rather artificial. Such a case is that of the QParity formulae, first introduced in [2] and later used in [1] to show an exponential separation between proof systems. These formulae have a single parameter $n \in \mathbb{N}$. We present the more succinct version of circuits for this family, as an example of what a formula family definition looks like.

Definition 1 (QParity circuits [1]). *Let $n \in \mathbb{N}$, $n \geq 2$, and let $x_1, \ldots, x_n$ and z be Boolean variables. We define the quantifier prefix $P_n = \exists x_1 \ldots \exists x_n \forall z$. We define an auxiliary circuit t_2 as $t_2 = x_1 \oplus x_2$ and for $i \in \{3, \ldots, n\}$ we define auxiliary t-circuits as $t_i = t_{i-1} \oplus x_i$ and the complete matrix as $\rho_n = t_n \oplus z$. The* QBF *instance will be* $\text{QParity}_n = P_n : \rho_n$.

3 The Formal Language

We now give a formalism in which to write these definitions. This formal language relies on the basic concept of *blocks*. A block is a sequence of *bricks*, which are literals (input variables that may be negated) or references to other blocks (also possibly negated). A block can then be assigned a single *attribute*, i.e. a *quantifier* or a *logical operator* (conjunction, disjunction or exclusive disjunction).

Example 1 (Basic use of blocks). The formula $\varphi(x, y, z) = (x \vee y) \wedge z$ can be defined in our language using two blocks:

```
define block B1 := x, y;          define block B2 := B1, z;
```

Blocks only declare ordering of bricks. Meaning is later given through an attribute (e.g. the *and* and *or* operators). These operate between them all the bricks in the block.

```
block B1 operated with OR;          block B2 operated with AND;
```

The structure of blocks captures simultaneously both the idea of gates on a Boolean circuit as well as the intricate nested patters of quantifier prefixes. Imagine that the previous formula is quantified as follows:

$$\forall x \exists y \exists z : \varphi(x, y, z)$$

We can define some blocks to obtain the structure of the quantifier prefix:

```
define block Q1 := x;               block Q1 quantified with A;
define block Q2 := y, z;            block Q2 quantified with E;
define block Q := Q1, Q2;
```

Finally, we can combine the quantifier prefix block `Q` with the block `B2` representing φ and indicate that this is our *output block*.

```
define block Phi := Q, B2;          output block: Phi;
```

To showcase the language in a more realistic scenario, we present the formal version of the QParity formulae from the previous section.

Example 2 (Formal version of the QParity *formulae).* Firstly, we declare *parameters*, followed by their type as well as possible constraints (in this case, $n \geq 2$). We then declare the variables. Variables x_i are denoted `x(i)` and the range of indices must be specified.

```
parameters: {                       variables: {
    n : int, `n >= 2`;                  x(i)    where i in 1..n;
}                                       z;
                                    }
```

We now declare the blocks. We have blocks for quantifiers and blocks for gates, but they use the same space and syntax (this allows to define non-prenex circuits). When quantifying a block, the block is expanded, assigning the quantifier to variables in a depth-first manner.

```
define blocks {                     define block Q := X, Z;
    X := x(i);
} where i in 1..n;                  block X quantified with E;
                                    block Z quantified with A;
define block Z := z;
```

In the same section, we define the blocks used to build the matrix of the formula. An important feature is that of *groupings*: a set of blocks grouped under the same name, so that they can all be simultaneously operated or quantified.

```
define blocks grouped in T {        define block Ro := T(n), z;
    T(2) := x(1), x(2);             all blocks in T operated with XOR;
    T(i) := T(s), x(i);             block Ro operated with XOR;
} where i in 3..n, s = ‘i-1‘;       define block Phi := Q, F;
```

For an extra, less artificial example, the Appendix contains the case study of the *Chromatic Formulae*, a QBF encoding of the Chromatic Number Problem where the use of a graph as a parameter of the definition is showcased.

4 The Tool

Based on the language described above, QBDef, a computer tool to parse definitions and output files to be fed to a QBF solver has been developed in Python.

If the input definition declares a PCNF formula, the tool can print either a QDIMACS or a QCIR file. If the input is a circuit, it can natively output a QCIR file or convert it to CNF and output a QDIMACS file using William Klieber's conversion tool developed in the context of the GhostQ QBF solver[3]. If the formulae are non-prenex, they can be printed in the specific QCIR format for non-prenex formulae.

Complex arithmetic expressions must be written in Python syntax and enclosed in backticks. This is because they are interpreted and evaluated by Python itself. Embedded Python gives our language an immense expressive power, as virtually any condition or structured object can be written in the definitions using built-in Python data types and functions.

5 Ongoing Work

Currently only a prototype of QBDef exists and requires polishing and extensive testing[4].

Additionally, ongoing work includes the study of other families that could exploit the features of this language, such as encodings of PSPACE games.

As a long term goal, this work could lead to future empirical research in the performance of QBF solvers by easily translating both existing and new formula families into the language presented here.

Acknowledgements

Thanks to Marc Denecker and Montserrat Hermo for co-promoting the Master's thesis to which this work belongs. Thanks to Florian Lonsing for his useful comments and insights.

[3] `https://www.wklieber.com/ghostq/qcir-converter.html`

[4] The current prototype is available at `https://github.com/alephnoell/QBDef`. The final version of the tool and its source code will be made available in coming months through this same channel.

References

1. Beyersdorf, O., Chew, L., Janota, M.: Extension variables in QBF resolution. In: Workshops at the Thirtieth AAAI Conference on Artificial Intelligence (2016)
2. Beyersdorff, O., Chew, L., Janota, M.: Proof complexity of resolution-based QBF calculi. In: LIPI Symposium on Theoretical Aspects of Computer Science (STACS'15). vol. 30, pp. 76–89. Schloss Dagstuhl-Leibniz International Proceedings in Informatics (2015)
3. Sabharwal, A., Ansotegui, C., Gomes, C.P., Hart, J.W., Selman, B.: QBF modeling: Exploiting player symmetry for simplicity and efficiency. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 382–395. Springer (2006)

Appendix

Example 3 (Definition and formal version of the Chromatic Number Problem). The *Chromatic Number Problem* is a well-known DP-complete problem: given a graph G and a natural number $k \geq 1$, decide whether k is the *chromatic number* of G, i.e. the minimum k such that G is k-colorable. Although this is an NP-complete problem and, as such, can be encoded into a SAT formula, a more natural encoding is also possible using quantification: there *exists* a coloring of G with k colours and *for all* other coloring of size $k-1$, these are not valid colorings for G.

We need to define a formula family for the problem, depending on two parameters: the graph G and the number k. The following definition or encoding for this problem was given by Sabharwal, et al. in [**?**].

In what follows we denote by n the number of nodes in the graph $G = (V, E)$. We define variables $x_{i,j}$ for $i \in [n]$ and $j \in [k]$ and $y_{i,j}$ for $i \in [n]$ and $j \in [k-1]$. Semantically, any of these variables is set to 1 if and only if node i is set to have colour j.

We now define a subformula $\mathit{\Gamma}$ that is true whenever the x-variables form a legal k-coloring,

$$\mathit{\Gamma} = \bigwedge_{i\in[n]} (x_{i,1} \vee \ldots \vee x_{i,k}) \wedge \bigwedge_{\substack{i\in[n] \\ j\neq j'\in[k]}} (\neg x_{i,j} \vee \neg x_{i,j'}) \wedge \bigwedge_{\substack{(i,i')\in E \\ j\in[k]}} (\neg x_{i,j} \vee \neg x_{i',j})$$

and another subformula, $\mathit{\Delta}$, which is true only when the y-variables do not form a legal $(k-1)$-coloring

$$\mathit{\Delta} = \bigvee_{i\in[n]} (\neg y_{i,1} \wedge \ldots \wedge \neg y_{i,k-1}) \vee \bigvee_{\substack{i\in[n] \\ j\neq j'\in[k-1]}} (y_{i,j} \wedge y_{i,j'}) \vee \bigvee_{\substack{(i,i')\in E \\ j\in[k-1]}} (y_{i,j} \wedge y_{i',j})$$

Clearly, k will be the chromatic number of G if there exists an assignment for the x-variables that makes $\mathit{\Gamma}$ true and for any assignment to the y-variables, $\mathit{\Delta}$

is true. This gives us the full encoding of the Chromatic Number Problem into a QBF, that we call the *Chromatic Formula*, $K(G, k)$:

$$K(G,k) = \exists x_{1,1} \dots x_{1,k} \dots x_{n,1} \dots x_{n,k} \forall y_{1,1} \dots y_{1,k-1} \dots y_{n,1} \dots y_{n,k-1} : \Gamma \wedge \Delta$$

We can now give the formal version of this definition in the syntax of our language. The main new feature showcased by this example is that we need to check whether a certain edge (i, j) is in the graph, $(i, j) \in_? E$. For this purpose, we encode the graph as and adjacency matrix, `edges`, and then the condition can be written as ‘`edges[i-1][j-1] == 1`‘ (using Python syntax). The full code is given below.

Formal version of the Chromatic Formulae

```
name: Chromatic formulae;
format: circuit-prenex;

parameters: {
    n     : int, ‘n >= 1‘;
    edges : list;
    k     : int, ‘k >= 1‘;
}

variables: {
    x(i, j)    where i in 1..n, j in 1..k;
    y(i, j)    where i in 1..n, j in 1..‘k-1‘;
}

blocks: {

    /* === blocks for quantifers === */

    define blocks grouped in X {
        X(i) := x(i, j);
    } where i in 1..n, j in 1..k;

    define blocks grouped in Y {
        Y(i) := y(i, j);
    } where i in 1..n, j in 1..‘k-1‘;

    define block Q := all blocks in X, all blocks in Y;

    all blocks in X quantified with E;
    all blocks in Y quantified with A;
```

```
/* ==== blocks for matrix ==== */

define blocks grouped in AllColored {
    Colored(i) := x(i, j);
} where i in 1..n, j in 1..k;

define blocks grouped in NotColored {
    NotColored(i) := -y(i, j);
} where i in 1..n, j in 1..`k-1`;

define block Gamma1 := all blocks in AllColored;

define block Delta1 := all blocks in NotColored;

define blocks grouped in SubGamma2 {
    SG2(i, j, l) := -x(i, j), -x(i, l);
} where i in 1..n, j in 1..k, l in 1..k, `j != l`;

define blocks grouped in SubDelta2 {
    SD2(i, j, l) := y(i, j), y(i, l);
} where i in 1..n, j in 1..`k-1`, l in 1..`k-1`, `j != l`;

define block Gamma2 := all blocks in SubGamma2;

define block Delta2 := all blocks in SubDelta2;

define blocks grouped in SubGamma3 {
    SG3(i, j, l) := -x(i, l), -x(j, l);
} where i in 1..n, j in 1..n, `edges[i-1][j-1] == 1`, l in 1..k;

define blocks grouped in SubDelta3 {
    SD3(i, j, l) := y(i, l), y(j, l);
} where i in 1..n, j in 1..n, `edges[i-1][j-1] == 1`, l in 1..`k-1`;

define block Gamma3 := all blocks in SubGamma3;

define block Delta3 := all blocks in SubDelta3;

define block Gamma := Gamma1, Gamma2, Gamma3;

define block Delta := Delta1, Delta2, Delta3;

define block F := Gamma, Delta;

all blocks in AllColored operated with OR;
```

```
    all blocks in NotColored operated with AND;

    all blocks in SubGamma2 operated with OR;
    all blocks in SubDelta2 operated with AND;

    all blocks in SubGamma3 operated with OR;
    all blocks in SubDelta3 operated with AND;

    blocks Gamma1, Gamma2, Gamma3 operated with AND;
    block Gamma operated with AND;

    blocks Delta1, Delta2, Delta3 operated with OR;
    block Delta operated with OR;

    block F operated with AND;

    /* define the output block */
    define block Phi := Q, F;
}

output block: Phi;
```

Bibliography

[1] S. Aaronson. **P** $=_?$ **NP**. In *Open Problems in Mathematics*, pages 1–122. Springer, 2016.

[2] A. V. Aho, M. S. Lam, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, 2nd edition, 2007.

[3] G. Amendola, F. Ricca, and M. Truszczynski. New models for generating hard random Boolean formulas and disjunctive logic programs. *Artificial Intelligence*, 279:103185, 2020.

[4] C. Ansotegui, C. P. Gomes, and B. Selman. The Achilles' heel of QBF. In *AAAI*, volume 2, pages 2–1, 2005.

[5] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[6] V. Balabanov, M. Widl, and J.-H. R. Jiang. QBF resolution systems and their proof complexities. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 154–169. Springer, 2014.

[7] P. Beame and T. Pitassi. Simplified and improved resolution lower bounds. In *Proceedings of 37th Conference on Foundations of Computer Science*, pages 274–282. IEEE, 1996.

[8] P. Beame and A. Sabharwal. *Proof Complexity Lecture Notes*. 2014.

[9] O. Beyersdorf, L. Chew, and M. Janota. Extension variables in QBF resolution. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[10] O. Beyersdorff, L. Chew, J. Clymo, and M. Mahajan. Short proofs in QBF expansion. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 19–35. Springer, 2019.

[11] O. Beyersdorff, L. Chew, and M. Janota. Proof complexity of resolution-based QBF calculi. In *LIPI Symposium on Theoretical Aspects of Computer Science (STACS'15)*, volume 30, pages 76–89. Schloss Dagstuhl-Leibniz International Proceedings in Informatics, 2015.

[12] H. K. Buning, M. Karpinski, and A. Flogel. Resolution for quantified Boolean formulas. *Information and computation*, 117(1):12–18, 1995.

[13] H. Chen. Beyond Q-Resolution and prenex form: A proof system for quantified constraint satisfaction. *Logical Methods in Computer Science*, Volume 10, Issue 4, Dec. 2014.

[14] H. Chen. Proof complexity modulo the polynomial hierarchy: Understanding alternation as a source of hardness. *ACM Transactions on Computation Theory (TOCT)*, 9(3):1–20, 2017.

[15] H. Chen and Y. Interian. A model for generating random quantified Boolean formulas. In *IJCAI*, pages 66–71, 2005.

[16] S. A. Cook and R. A. Reckhow. The relative efficiency of propositional proof systems. *The journal of symbolic logic*, 44(1):36–50, 1979.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT press, 2009.

[18] 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, *Theory and Applications of Satisfiability Testing*, pages 214–228, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[19] A. S. Fraenkel, M. R. Garey, D. S. Johnson, T. Schaefer, and Y. Yesha. The complexity of checkers on an $N \times N$ board. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 55–64, 1978.

[20] A. Haken. The intractability of Resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[21] J. Heylen. *Intermediate Logic.* Compiled from the Open Logic Text, 2020.

[22] L. Hinde, J. Blinkhorn, and O. Beyersdorff. Size, cost, and capacity: A semantic technique for hard random QBFs. *Logical Methods in Computer Science*, 15, 2019.

[23] M. Janota. On Q-resolution and CDCL QBF solving. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 402–418. Springer, 2016.

[24] M. Janota and J. Marques-Silva. Expansion-based QBF solving versus Q-resolution. *Theoretical Computer Science*, 577:25–42, 2015.

[25] C. Jordan, W. Klieber, and M. Seidl. Non-CNF QBF solving with QCIR. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[26] A. Kfoury. Formal modeling with QBF. URL: http://www.cs.bu.edu/faculty/kfoury/UNI-Teaching/CS512/AK_Documents_Past_Semesters/modeling-with-QBF.pdf, last checked on 2020-05-12, 2017.

[27] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison Wesley, Boston, 2006.

[28] J. Krajíček. Proof complexity. In *European Congress of Mathematics (ECM)*, pages 221–231. Stockholm, Sweden, Zurich: European Mathematical Society, 2005.

[29] J. Krajíček. *Proof Complexity*. Cambridge University Press, 2019.

[30] C. H. Papadimitriou. *Computational Complexity*. John Wiley and Sons Ltd., 2003.

[31] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293304, Sept. 1986.

[32] A. Sabharwal, C. Ansotegui, C. P. Gomes, J. W. Hart, and B. Selman. QBF modeling: Exploiting player symmetry for simplicity and efficiency. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 382–395. Springer, 2006.

[33] J. A. Storer. On the complexity of chess. *Journal of Computer and System Sciences*, 27(1):77 – 100, 1983.

[34] L. Tentrup. Non-prenex QBF solving using abstraction. In N. Creignou and D. Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 393–401, Cham, 2016. Springer International Publishing.

[35] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

[36] M. van der Hallen and G. Janssens. SOGrounder: Modelling and solving second-order logic. In *Knowledge Representation and Reasoning Conference*, 2018.

[37] A. Van Gelder. Contributions to the theory of practical quantified Boolean formula solving. In *International Conference on Principles and Practice of Constraint Programming*, pages 647–663. Springer, 2012.

[38] Wikipedia. Boolean hierarchy. URL: https://en.wikipedia.org/wiki/Boolean_hierarchy, last checked on 2020-06-05.

[39] Wikipedia. Generalized geography. URL: https://en.wikipedia.org/wiki/Generalized_geography, last checked on 2020-05-12.