
TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

M A G I S T E R A R B E I T

# Counter Lattice Generation for Non-provable Formulas

conducted at the Institute of Information Systems
Knowledge-Based Systems Group
at the Vienna University of Technology

under the supervision of
A.o.Univ.Prof. Dipl.-Ing. Dr.rer.nat. Uwe Egly

by

Bakk.techn. Andreas Zugaj
Leibnizg. 13/25
A-1100 Wien

Ort, Datum

Unterschrift

**Abstract**

Orthologic is the logic defined over lattices in which an unary operation called orthocomplementation is defined. It is similar to classical propositional logic, which is defined as orthologic but with the extension that the distributive laws must hold. What makes orthologic so interesting is that, unlike for classical logic, the validity problem in orthologic is known to be polynomially solvable.

Given some formula, it is interesting to determine whether it is valid or not in orthologic. This question can be answered using proof systems like Gentzen systems for orthologic. For these systems, it is known from the literature that they are sound and complete. In the positive case that a formula is valid, these proof systems result in a proof. But in the negative case that a formula is not valid, these systems give us no proof. They fail to find a proof and because of the soundness and correctness, we can conclude that the formula is not valid. What is missing is a certificate that the given formula does not hold. One possibility to present such a certificate is to construct a counter example. Such a counter example is sufficient, because validity of a formula in orthologic means that it holds in *all* ortholattices. This means that, if we find at least one lattice where the formula does not hold, this lattice is a witness for the invalidity of the formula.

In this master thesis, we describe an implementation, CGOL, of a program written in the programming language C that is capable of testing an orthologic formula for validity. If the formula is not valid, the program is able to generate a counter example, which convinces the user that this formula is indeed not valid. The thesis handles the theoretic aspects necessary for generating such counter examples and it also gives details about the concrete implementation.

# Contents

# Chapter 1

# Introduction

When studying some logic, we are always interested in a method to determine whether some given formula is valid or not with respect to some semantics. For proof systems formal soundness and completeness proofs usually exist, but how credible is one specific instance of a proof for a certain formula. Of course the proof can be done by hand, which is very credible, because every inference step and its applicability is known to the user, but doing proofs by hand is limited to very small formulas. In real world applications we rather use computer-generated proofs instead, but how should the user know the computer is really doing its job right when just outputting "valid" or "invalid" at the end of a possibly long lasting proof-search? The answer is that it is obviously quite hard to check the result if one has only the answer "valid" or "invalid". What is needed is a kind of certificate for the answer. For valid formulas, the certificate is the proof, found by the proof system. Such a proof can be easily verified using (random) proof checking for example. But in case of an invalid formula, we only see that our inference procedure is not able to derive a proof, but that is all the information we get. So there is not much difference in just saying: "invalid". But a formula is only valid, if it holds in all structures, where the specific structure depends on the logic. For instance, in classical propositional logic, a formula is valid if it is true in all

interpretations. That means if we find one structure for which the formula does not hold, this example is the certificate for the invalidity of the formula. Such a counter example is constructed from a failed proof attempt.
In this work, we will focus on a logic called *orthologic* which is, due to historical reasons, also known as *minimum quantum logic*. Orthologic is the logic defined over lattices in which an unary operation called orthocomplementation is defined. But we will also turn our attention to the logic defined over pure lattices, because this logic is a subset of orthologic and many concepts we need for orthologic are easier to understand in the simpler environment of pure lattices.
Why do we concentrate on orthologic? One reason is its similarity to classical propositional logic, having the same syntax, but of course different semantics. The difference is that, in orthologic, the distributive laws do not hold. Just the distributive laws do not seem to make a crucial difference, but taking a closer look on the complexity for automatic proof-search as is done in [2] shows the real difference: The validity problem for orthologic is polynomially solvable unlike in classical logic where it is NP-complete.
Obtaining polynomially bounded Gentzen proof procedures is a delicate matter. Using Gentzen-like calculi with backward search for orthologic results only in an exponential algorithm. However, using a Gentzen system in forward search (also known as Maslov's inverse method) gives us a polynomially bounded proof procedure if the subformula property is taken into account, when axiom schemata are generated. This is interesting, because in classical logic backward search is usually considered faster than forward search.
Another reason why orthologic is interesting is that there is not much work about the generation of counter examples for non-provable formulas in orthologic. For first order logic there is various work on model generation, e.g., by Leitsch et al. in [6]. There is a work in progress by Egly [1] that deals with the theory of generating counter examples for orthologic, but there is no implementation as computer program yet. With this work we are trying to fill this gap.

The aim of this master thesis is implementing a program in C based on the Gentzen-like calculus GOL for orthologic that determines for some given formula, whether it is valid or not, and if it is valid prints out the proof. If it is not valid it generates a counter example that gives evidence of the invalidity of this formula.
In Chapter 2, we will first discuss the theoretical background needed to understand orthologic and the theory we need for the construction of counter examples. In orthologic the counter examples have the form of ortholattices, that is why we will call them counter lattices in the following. In Chapter 3, we will discuss the basic ideas on which the implementation in C is based. The program is called CGOL for counter example GOL. We will also discuss the performance of the implementation on test cases. Chapter 4 will be dedicated to the reduction of the size of the counter lattices. We will see that the size of a counter lattice can be quite large, especially when the formula is large. Because our algorithm is polynomial, we can handle big formulas in reasonable time. Since the purpose of the counter lattice is to convince the user that the given input formula does not hold for all lattices, we will spend some effort on a way to decrease the size of the counter lattice, because small counter ortholattices are easier to depict and they are therefore easier to comprehend. The approach we will present is implemented in CGOL and proves quite useful. In the last chapter, we will conclude with a summary of the results and give some outlook on open problems and problems currently under work.

# Chapter 2

# Theoretical Background

This chapter is dedicated to the theoretical background we need throughout this thesis. We will first review what lattices and ortholattices are and discuss their important properties needed in the following. Afterward we will discuss the logics defined over lattices respectively ortholattices.
Then we introduce Gentzen systems for lattices (GL) and ortholattices (GOL). It is well-known that Gentzen proof systems deliver proofs in case the formula is valid. However, the generation of counter lattices for non-provable formulas is not obvious. We apply an approach presented in [1], which shows that forward search in Gentzen systems for lattices can simulate the effect of Skolem's procedure. Since for the result of the Skolem's procedure, it is possible to generate a counter lattice, it is also possible to generate such a lattice from failed proof attempts in Gentzen systems. Moreover, as shown in [1], the construction can be extended to ortholattices.

## 2.1 Lattices and Ortholattices

In this section, we recall the properties of lattices and ortholattices. This exposition is based on the discussion of lattices and ortholattices in [4].

### 2.1.1 Lattices and Posets

There are two ways of defining lattices. The first one is an equational characterization of lattices, the other one uses *partially ordered sets* (*posets*). We will start with the equational definition:
A *lattice* is an algebra $(L, \wedge, \vee)$ with $L$ being a set closed under the binary operations $\wedge$ called "meet" and $\vee$ called "join". The algebra must satisfy the following laws.

| | | |
|---|---|---|
| 1a. | $a \vee (b \vee c) = (a \vee b) \vee c$ | (associativity) |
| 1b. | $a \wedge (b \wedge c) = (a \wedge b) \wedge c$ | |
| 2a. | $a \vee b = b \vee a$ | (commutativity) |
| 2b. | $a \wedge b = b \wedge a$ | |
| 3a. | $a \vee (a \wedge b) = a$ | (absorption law) |
| 3b. | $a \wedge (a \vee b) = a$ | |

We show that the idempotence laws

| | | |
|---|---|---|
| 4a. | $a = a \wedge a$ | (idempotence) |
| 4b. | $a = a \vee a$ | |

can be derived from the absorption laws.
We start with $a \wedge (a \vee (a \wedge b')) = a$ which is an instance of 3b. with $b = a \wedge b'$. Then $a = a \wedge (a \vee (a \wedge b')) =_{3a.} a \wedge a$. Therefore $a = a \wedge a$. The proof of $a = a \vee a$ is similar.

These laws mentioned above hold in every lattice. There are two important laws that *do not hold in every lattice*, namely distributivity and modularity. We define these two laws in the following.

5a. $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$ (distributivity)

5b. $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$

6. $a \vee (b \wedge (a \vee c)) = (a \vee b) \wedge (a \vee c)$ (modularity)

An alternative approach for the definition of lattices is based on posets. A *poset* is a pair $(P, \leq)$ where $P$ is a set and $\leq$ is a binary relation on $P$. The relation $\leq$ must satisfy the following conditions:

1. $a \leq a$ for all $a \in P$ (reflexivity)

2. $a \leq b$ and $b \leq a$ imply $a = b$ for all $a, b \in P$ (anti-symmetry)

3. $a \leq b$ and $b \leq c$ imply $a \leq c$ for all $a, b, c \in P$ (transitivity)

Note that the equivalence between $a$ and $b$ is defined as follows:

$$a = b \text{ holds iff } a \leq b \text{ and } b \leq a \text{ holds.}$$

Let $P$ be a poset and let $X$ be a subset of $P$. An *upper bound* $c$ of $X$ is an element $c$ in $P$ such that $x \leq c$ holds for all $x \in X$. A *lower bound* $d$ of $X$ is an element $d$ in $P$ such that $d \leq x$ holds for all $x \in X$. The *supremum* of $X$, denoted by $\bigvee X$, is an upper bound of $X$ which is smaller or equal than every other upper bound of $X$. A supremum might not exist for every subset. The *infimum* of $X$, denoted by $\bigwedge X$ is a lower bound of $X$ which is larger or equal than every other lower bound of $X$. An infimum might not exist for every subset.

*Lattices are exactly those posets, for which every two-element subset has a supremum and an infimum.*

For elements $a, b$ of a lattice $L$ we define:

$$a \leq b \text{ holds if } a = a \wedge b \text{ or } b = a \vee b \text{ holds.}$$

We will need the term *bounded poset* later on.

A *bounded poset* is a poset $P$ with a smallest element $\bot$ with $\bot \leq x$ for all $x \in P$ and a largest element $\top$ with $x \leq \top$ for all $x \in P$.

### 2.1.2 Ortholattices and Orthoposets

We enhance posets by an unary operation (orthocomplementation) in order to get orthoposets, which are used as a basis for ortholattices.
Let $P$ be a bounded poset. We define an unary operation $\neg$ on $P$ called *orthocomplementation* which satisfies the following conditions:

1. if $a \leq b$ then $\neg b \leq \neg a$,
2. $\neg\neg a = a$,
3. the supremum of $a \vee \neg a$ and the infimum of $a \wedge \neg a$ exist and $a \vee \neg a = \top$ and $a \wedge \neg a = \bot$.

A poset with orthocomplementation is called *orthoposet*. An *ortholattice* is an orthoposet which is also a lattice, i.e., for which every two-element subset has a supremum and an infimum.
For ortholattices, the *de Morgan's laws* hold:

$$\neg(a \vee b) = \neg a \wedge \neg b$$
$$\neg(a \wedge b) = \neg a \vee \neg b$$

A map $\varphi$ from a poset $P$ into a poset $Q$ is called a *homomorphism* if it is order preserving, i.e., if $x \leq y$ implies $\varphi(x) \leq \varphi(y)$.

### 2.1.3 Logical Aspects

Throughout this thesis, the attention will be restricted to finitely presented lattices and ortholattices to be defined below.

**Definition 2.1** *Let $X$ be a countably infinite set disjoint from the set $\{\wedge, \vee, \neg, \top, \bot\}$. We define the set of* lattice terms *over $X$ inductively as follows:*

1. *Every element $x$ of the set $X \cup \{\top, \bot\}$ is a lattice term.*

2. *If $a, b$ are lattice terms, then $a \wedge b$ and $a \vee b$ are lattice terms.*

A *lattice presentation* is a pair $< X, R >$, where $X$ is a set of variables and $R$ is a set of relations. A *relation* is an equation $s(x_1, \ldots, x_n) \leq t(x_1, \ldots, x_n)$ where $s$ and $t$ are lattice terms over $X$. A *finite presentation* is a lattice presentation where $X$ and $R$ are finite. A *finitely presented lattice* is a lattice which is presented by a finite presentation.
We will now extend the above for orthologic.

**Definition 2.2** *The set of* ortholattice terms *over $X$ is defined inductively as follows:*

1. *Every lattice term over $X$ is an ortholattice term.*

2. *If $a$ is an ortholattice term, then $\neg a$ is an ortholattice term.*

A finitely presented ortholattice is defined as above in the case of a lattice, but using the notion of ortholattice term instead of lattice term. The syntax of an ortholattice term is equivalent to the syntax of a propositional logic *formula*. That is why we will use formula synonymously for lattice term and ortholattice term.
We use the notation $A \models_{LL} B$ to denote that a formula $B$ can be inferred from a formula $A$ in the logic of pure lattices. Analogously we use the notation $A \models_O B$ to denote that a formula $B$ can be inferred from a formula $A$ in orthologic. A *valuation* $v$ of a formula $F$ is a map from $F$ into an (ortho)lattice, where $\neg$, $\wedge$ and $\vee$ are interpreted as the operations orthocomplementation, lattice meet and lattice join.
For the logic of pure lattices we define that $A \models_{LL} B$ holds iff $v(A) \leq v(B)$ holds for all valuations $v$ into all lattices. Analogously we define for orthologic that $A \models_O B$ holds iff $v(A) \leq v(B)$ holds for all valuations $v$ into all ortholattices.

A formula $F$ is said to be *valid* in orthologic if $\models_O F$ holds. A formula that is not valid is said to be *invalid*. The empty left side of $\models_O$ is interpreted as $a \vee \neg a$. For pure lattices a similar statement cannot be made. It only allows us validity considerations on $A \models_{LL} B$ for non-empty $A$ and $B$.

## 2.2 Gentzen Systems

In this section, we will introduce the sequent (or Gentzen) calculi GL for the logic of pure lattices and GOL for orthologic. For sequent systems we need some definitions first.
Gentzen calculi operate on sequents instead of formulas. A sequent $M \vdash N$ is an ordered pair of sets $M, N$ of formulas, where the sets can have restrictions depending on the calculus. $M$ is called the *antecedent* and $N$ the *succedent*. In our case we need n-restricted sequents.

**Definition 2.3** *A* n-restricted *sequent is an ordered pair $M \vdash N$, where $M, N$ are sets and $|M| + |N| \leq n$ holds. $M$ is called the antecedent and $N$ is called the succedent of the sequent. $|X|$ denotes the cardinality of the set $X$.*

For GL and GOL-1-1 we will need an even more restricted version of a sequent.

**Definition 2.4** *A* 1-1-sequent *is an ordered pair of the form $\phi \vdash \psi$, where $\phi$ and $\psi$ are formulas. The formula $\phi$ is the only formula of the antecedent and $\psi$ is the only formula of the succedent of the 1-1-sequent.*

A Gentzen system consists of a set of axioms and a set of inference rules. For GL and GOL the axioms are split into logical and non-logical axioms. The logical axioms are a set of sequents that are known to be valid. The non-logical axioms are assumptions that some sequents hold. For GL and GOL without non-logical axioms a cut-elimination theorem exists. Non-logical

axioms are discussed in more detail in Chapter 4. The inference rules are of the form:

$$\frac{P}{C}\,\alpha \qquad \text{and} \qquad \frac{P \qquad Q}{C}\,\beta$$

where $P, Q$ and $C$ are sequents. $P$ and $Q$ are called *premises* and $C$ is called *conclusion*. Inference rules of the form of $\alpha$ are called *unary-rules* or *$\alpha$-rules*, whereas the other type is called *binary-rules* or *$\beta$-rules*.
We want to focus our attention on two common proof-search strategies, the widely-used *backward search* and the, in classical logic, seldomly applied *forward search* (also known as Maslov's inverse method). Forward search in classical logic is often considered to be inefficient, because it is not goal oriented. In our system, backward search generates *tree proofs*, whilst forward search generates *sequence proofs*.

**Definition 2.5** *A* tree proof *of a sequent $S$ is a rooted tree with $S$ as its root. The inner nodes are sequents derived by applications of the inference rules on the direct predecessor nodes and all leaves are axioms.*

**Definition 2.6** *A* sequence proof *of a sequent $S$ from a set of non-logical axioms $A$ in GL or GOL is a sequence $S_1, \ldots, S_n$ of sequents such that $S = S_n$ and for $1 \leq m \leq n$ and $1 \leq o \leq n$ and $k, l < m$,*

1. *$S_m$ is a logical or non-logical axiom, or*
2. *$S_m$ is the conclusion of a unary inference with premise $S_k$, or*
3. *$S_m$ is the conclusion of a binary inference with premises $S_k$ and $S_l$.*

**Definition 2.7** *A* saturation $\sigma(X)$ *from a set of non-logical axioms $A$ and restricted by a set $X$ of formulas in GL or GOL is a sequence $S_1, \ldots, S_n$ of sequents such that for $1 \leq m \leq n$ and $k, l < m$,*

1. *$S_m$ is a logical or non-logical axiom, or*

2. *$S_m$ is the conclusion of a unary inference with premise $S_k$, or*

3. *$S_m$ is the conclusion of a binary inference with premises $S_k$ and $S_l$, and*

4. *for all formulas $x, y$ occurring in $S_m$, $x, y$ are members of the set $X$, and*

5. *there exists no sequent $S_{n+1}$ where $S_{n+1}$ is the conclusion of a unary inference with premise $S_m$ without violating the restriction in 4, and*

6. *there exists no sequent $S_{n+1}$ where $S_{n+1}$ is the conclusion of a binary inference with premises $S_m$ and $S_o$ without violating the restriction in 4.*

Informally a *saturation* is the sequence of all sequents that are derivable by GL or GOL from a set of non-logical axioms A restricted by a set $X$ of formulas that are allowed to appear in the sequents.
A sequent $S$ is provable if there exists a sequence of applications of inference rules starting from a set of axioms, such that it derives the sequent $S$. If a sequent $M \vdash N$ is provable in a sequent calculus for a logic $L$, then $M \models_L N$ holds.
The usual backward search starts with the sequent $S$ as the root, and builds up a tree with axioms and non-logical axioms as leaves. Backward search is a goal-oriented approach, whereas forward search is a saturation-based approach which requires a subformula property to be efficient. GOL (GL respectively) has such a subformula property, i.e., forward search can be implemented efficiently. In GOL forward search is a polynomial decision procedure whilst backward search is exponential in the worst case! A detailed analysis on the properties for the different proof-search strategies for GOL is given in [2]. We will further on refer to the sequent S as the *input sequent* or synonymously the *end sequent*.

### 2.2.1 GL

GL is a sequent calculus for lattices. GL is a real subset of GOL, which we will introduce in the next subsection. Let $A \models_{LL} B$ be the inference relation of the logic for pure lattices. The relation $A \models_{LL} B$ means that $B$ is derivable from $A$ in the logic of pure lattices.

**Theorem 2.1** *The sequent $A \vdash B$ is provable in GL iff $A \models_{LL} B$ holds. This means that GL is sound and correct for the logic of pure lattices.*

The proof of the theorem can be found in [4]. Note that [4] gives only a proof of soundness and correctness of GOL. But GL is a real subset of GOL, thus a soundness and correctness proof of GOL can be easily adapted for GL. Also note that the version of GOL we will introduce later on, is an adapted version of the one in [4]. The cut-elimination theorem is also shown in [4]. For more properties of GOL see [2] and [1].
GL operates on 1-1-sequents as defined above. Note that for GL the subformula property holds in a strict sense. This means given a sequent $\phi \vdash \psi$ we are trying to prove, all sequents we are deriving during the proof-search have only subformulas of $\phi$ on the antecedent side and only subformulas of $\psi$ on the succedent side. Logical axioms in GL are sequents of the form $a \vdash a$. In the case of a proof-search for a specific sequent $\phi \vdash \psi$, the axiom's formulas $a$ can be restricted to subformulas of $\phi, \psi$. In the following $\phi$, $\theta$ and $\psi$ denote single formulas. The inference rules consist of $\alpha$-rules which are unary rules requiring only a single premise:

$$\frac{\theta \vdash \phi}{\theta \vdash \phi \vee \psi} \vee r_1 \qquad \frac{\theta \vdash \psi}{\theta \vdash \phi \vee \psi} \vee r_2$$

$$\frac{\phi \vdash \theta}{\phi \wedge \psi \vdash \theta} \wedge l_1 \qquad \frac{\psi \vdash \theta}{\phi \wedge \psi \vdash \theta} \wedge l_2$$

and $\beta$-rules which are binary rules requiring two premises:

$$\frac{\phi \vdash \theta \quad \psi \vdash \theta}{\phi \vee \psi \vdash \theta} \vee l$$

$$\frac{\theta \vdash \phi \quad \theta \vdash \psi}{\theta \vdash \phi \wedge \psi} \wedge r$$

We will need a restricted version of the cut rule. The system GL+acut is the system GL extended by the *analytic cut* rule. Analytic means that all formulas appearing in the rule are subformulas of the formulas of the end sequent. The acut rule is:

$$\frac{\theta \vdash \phi \quad \phi \vdash \psi}{\theta \vdash \psi} acut$$

GL in forward search is a polynomial decision method for determining whether $s \leq t$ holds in all lattices. This is a direct consequence of Theorem 19 in [2].

### 2.2.2 GOL

GOL can be considered as an extension of GL for orthologic. Let $A \models_O B$ be the inference relation of orthologic. The relation $A \models_O B$ means that $B$ is derivable from $A$ in orthologic.

**Theorem 2.2** *The sequent $A \vdash B$ is provable in GOL iff $A \models_O B$ holds. This means that GOL is sound and correct for orthologic.*

The proof of the theorem can be found in [4].
Instead of 1-1-sequents, GOL requires 2-restricted sequents, i.e., sequents $M \vdash N$ for which $|M| + |N| \leq 2$ holds. It consists of $\alpha$-rules and $\beta$-rules similar to GL:

$$\frac{M \vdash \phi, N}{M \vdash \phi \vee \psi, N} \vee r_1 \qquad \frac{M \vdash \psi, N}{M \vdash \phi \vee \psi, N} \vee r_2$$

$$\frac{M, \phi \vdash N}{M, \phi \wedge \psi \vdash N} \wedge l_1 \qquad \frac{M, \psi \vdash N}{M, \phi \wedge \psi \vdash N} \wedge l_2$$

$$\frac{M, \phi \vdash N \quad M, \psi \vdash N}{M, \phi \vee \psi \vdash N} \vee l$$

$$\frac{M \vdash \phi, N \quad M \vdash \psi, N}{M \vdash \phi \wedge \psi, N} \wedge r$$

Additionally GOL has rules for the orthocomplementation (also called negation):

$$\frac{M, \phi \vdash N}{M \vdash \neg\phi, N} \neg r \qquad \frac{M \vdash \phi, N}{M, \neg\phi \vdash N} \neg l$$

GOL also needs structural rules for weakening:

$$\frac{M \vdash N}{M, \phi \vdash N} wl \qquad \frac{M \vdash N}{M \vdash \phi, N} wr$$

Note that the structural rules of exchange and contraction are not required, because $M$, $N$ are sets.
The system GOL+acut is the system GOL extended by the analytic cut rule.

Since sequents are not restricted to 1-1-sequents in GOL, there are additional rules for analytic cut of the following form:

$$\frac{x \vdash y \quad y, z \vdash}{x, z \vdash} \qquad \frac{\vdash y, z \quad y \vdash x}{\vdash x, z}$$

$$\frac{\vdash y, z \quad x, y \vdash}{x \vdash z} \qquad \frac{\vdash x, y \quad y, z \vdash}{z \vdash x}$$

GOL has the *subformula property*, which means that all occurrences of formulas in all derived sequents in a proof for some sequent $M \vdash N$ are subformulas of $M$ and $N$. Furthermore GOL has the *strict subformula property*, which means that all occurrences of formulas are subformulas of $M$ and $N$ in the corresponding *polarity*. We will refer to this property further on as the *polarity restriction*. Given a sequent $M \vdash N$ we define *polarity* $\mathrm{pol}(x)$ inductively:

1. $\mathrm{pol}(M) = -$;
2. $\mathrm{pol}(N) = +$;
3. $\mathrm{pol}(\neg x) = \begin{cases} + & \text{if } \mathrm{pol}(x) = -; \\ - & \text{if } \mathrm{pol}(x) = +; \end{cases}$

For example in the sequent $\neg x \vdash \neg\neg y$ $\mathrm{pol}(x) = +$ and $\mathrm{pol}(y) = +$, because $x$ appears in the antecedent side thus $\mathrm{pol}(x) = -$ but $x$ is negated thus its polarity is changed to $+$.

The *strict subformula property* means that formula with positive polarity occurrences of the end sequent occur in the succedent and formula with negative polarity occurrences of the end sequent occur in the antecedent.

This subformula property together with 2-restricted sequents is very useful for forward search, making GOL a polynomial decision method for the validity of an orthologic formula. It is also necessary for the termination of forward search. Forward search terminates if no new sequent can be generated.

Thus, the set of possible sequents must be finite, otherwise the algorithm would not terminate. The idea is, that the number of subformulas is finite and the possibilities to build 2-restricted sequents out of them is also finite. This leads to a finite number of possible sequents, to be more precise the number is not only finite but polynomially bounded (see [2] for the details and an approximation of the number of possible sequents).

### 2.2.3 GOL-1-1

GOL-1-1 is a special version of GOL that operates on 1-1-c sequents instead of 2-restricted sequents. It also needs a certain form of sequent formulas, so called *c-formulas* which are formulas with an indicator $c$ which is interpreted as "the formula occurs on the complementary side of the sequent".
Let $\phi$ be a formula. Then $\phi$ as well as an expression of the form $\phi^c$ is a *c-formula*. In the former case, we say that the indicator is $\epsilon$, in the latter case, it is $c$.

**Definition 2.8** *A* 1-1-c-sequent *is a 1-1-sequent, where all formulas of the sequent are c-formulas and at most one sequent formula is indicated with the indicator c. Therefore, 1-1-c-sequents have the form*

$$x^{c_1} \vdash y^{c_2} \text{ where } c_1, c_2 \in \{\epsilon, c\} \text{ and } c_1 \neq c \text{ or } c_2 \neq c.$$

The system of GOL-1-1 is given by the following inference rules:

$$\frac{\phi^{c_1} \vdash \psi_1}{\phi^{c_1} \vdash \psi_1 \vee \psi_2} \vee r_1 \qquad \frac{\phi^{c_1} \vdash \psi_2}{\phi^{c_1} \vdash \psi_1 \vee \psi_2} \vee r_2 \qquad \frac{\phi_1 \vdash \psi^{c_1} \quad \phi_2 \vdash \psi^{c_1}}{\phi_1 \vee \phi_2 \vdash \psi^{c_1}} \vee l$$

$$\frac{\phi_1 \vdash \psi^{c_1}}{\phi_1 \wedge \phi_2 \vdash \psi^{c_1}} \wedge l_1 \qquad \frac{\phi_2 \vdash \psi^{c_1}}{\phi_1 \wedge \phi_2 \vdash \psi^{c_1}} \wedge l_2 \qquad \frac{\phi^{c_1} \vdash \psi_1 \quad \phi^{c_1} \vdash \psi_2}{\phi^{c_1} \vdash \psi_1 \wedge \psi_2} \wedge r$$

$$\frac{\phi \vdash \psi^{c_1}}{\phi \vdash (\neg\psi)^{\overline{c_1}}} \neg r^{1-1} \qquad \frac{\phi^{c_1} \vdash \psi}{(\neg\phi)^{\overline{c_1}} \vdash \psi} \neg l^{1-1} \qquad \frac{\phi^{c_1} \vdash \psi^{c_2}}{\psi^{\overline{c_2}} \vdash \phi^{\overline{c_1}}} toggle \quad c_1 \neq c_2$$

$$\frac{\phi^c \vdash \phi}{\psi^{c_1} \vdash \phi} wl^{1-1} \qquad \frac{\phi \vdash \phi^c}{\phi \vdash \psi^{c_1}} wr^{1-1}$$

Egly gives in [1] a proof of the equivalence between GOL-1-1 and GOL. Interesting for our purpose is the translation between arbitrary 2-restricted sequents and 1-1-c-sequents, because, as we will see later on, we can construct our counter orthoposet only out of 1-1-c-sequents and not out of 2-restricted ones. Thus we need a translation that enables us to transform all sequents we derived in GOL into 1-1-c-sequents.

$$\delta_{1-1}(S) = \begin{cases} x \vdash y & \text{if } S : x \vdash y; \\ x \vdash x^c & \text{if } S : x \vdash; \\ y^c \vdash y & \text{if } S : \ \vdash y; \\ x \vdash y^c \text{ and } y \vdash x^c & \text{if } S : x, y \vdash; \\ x^c \vdash y \text{ and } y^c \vdash x & \text{if } S : \ \vdash x, y; \end{cases}$$

Note that the translation of $x, y \vdash$ yields two 1-1-c-sequents. This means the translation is not injective and thus the translation given here is only used in one direction, i.e., from GOL to GOL-1-1. The reason why we use this translation is that for generating the counter lattice we only need the translation from GOL to GOL-1-1.

## 2.3 Generation of Counter (Ortho)Lattices

We first show how to generate a counter lattice and later on, we extend to counter ortholattices.

### 2.3.1 Counter Lattices

Let us suppose that we have a failed proof-search for some $s \leq t$ where we used forward search. Then by the correctness of GL we can conclude that $s \leq t$ does not hold for all lattices. As a witness for the fact that $s \leq t$ does not hold for all lattices, we want to generate a lattice which illustrates that fact. The procedure is the following. With the calculus GL in forward

search, we derive all $\leq$-relations that hold for the set of subformulas $S = \mathrm{sf}(s) \cup \mathrm{sf}(t)$. Egly gives in [1] a proof that GL in forward search, without the polarity restriction derives for a sequent $s \vdash t$ a set of sequents, which are sufficient for finding all the relations that hold on $S$. It is sufficient to translate all derived sequent $x \vdash y$ with $x \leq y$. The resulting poset can be extended to a lattice, in which $s \leq t$ does not hold, which gives us a counter example. We illustrate the generation of the counter example with an example.

**Example 2.1** *Let $s = a \wedge (b \vee c)$ and $t = (a \wedge b) \vee c$. The sequent $s \vdash t$ is not derivable in GL, i.e., $s \leq t$ does not hold for all lattices. Using GL with subformula property but without polarity restriction yields the following set of sequents:*
*First we derive the logical axioms 0.-6. Note that only axioms $A \vdash A$ are interesting where $A$ is a subformula of the end sequent.*

*0.* $a \vdash a$

*1.* $b \vdash b$

*2.* $c \vdash c$

*3.* $a \wedge b \vdash a \wedge b$

*4.* $b \vee c \vdash b \vee c$

*5.* $(a \wedge b) \vee c \vdash (a \wedge b) \vee c$

*6.* $a \wedge (b \vee c) \vdash a \wedge (b \vee c)$

*From 0.-6. we derive the following sequents by applying the inference rules of GL.*

*7.* $a \wedge b \vdash a$ *from 0 using rule* $\wedge l_1$

8. $a \wedge (b \vee c) \vdash a$ *from 0 using rule* $\wedge l_1$

9. $a \wedge b \vdash b$ *from 1 using rule* $\wedge l_2$

10. $b \vdash b \vee c$ *from 1 using rule* $\vee r_1$

11. $c \vdash (a \wedge b) \vee c$ *from 2 using rule* $\vee r_2$

12. $c \vdash b \vee c$ *from 2 using rule* $\vee r_2$

13. $a \wedge b \vdash (a \wedge b) \vee c$ *from 3 using rule* $\vee r_1$

14. $a \wedge (b \vee c) \vdash b \vee c$ *from 4 using rule* $\wedge l_2$

*From 7., 9., 12. and 15. we derive the following sequents by applying the inference rules of GL.*

15. $a \wedge b \vdash b \vee c$ *from 9 using rule* $\vee r_1$

16. $(a \wedge b) \vee c \vdash b \vee c$ *from 12 together with 15 using binary rule* $\vee l$

17. $a \wedge b \vdash a \wedge (b \vee c)$ *from 15 together with 7 using binary rule* $\wedge r$

*After the generation of 0. – 17., no more new sequents are derivable and the proof-search terminates. As it can easily be seen,* $a \wedge (b \vee c) \vdash (a \wedge b) \vee c$ *is not derived, which means that* $a \wedge (b \vee c) \leq (a \wedge b) \vee c$ *does not hold for every lattice. The sequents derived above with* $\vdash$ *interpreted as* $\leq$ *give us a poset. Figure 2.1 represent the poset as Hasse-Diagram with reflexivity and transitivity omitted.*

Figure 2.1 illustrates the drawback of our approach. Our approach derives a poset with the desired property that $s \leq t$ does not hold, but as can be seen, it is only a poset *not* a lattice! Recall that a lattice is a poset where every two-element subset has a supremum and an infimum. In this example, $a \wedge b$ and $c$ have no infimum and $a$ and $b \vee c$ have no supremum. This means that we have only a "partial lattice", which we have to extend somehow to a

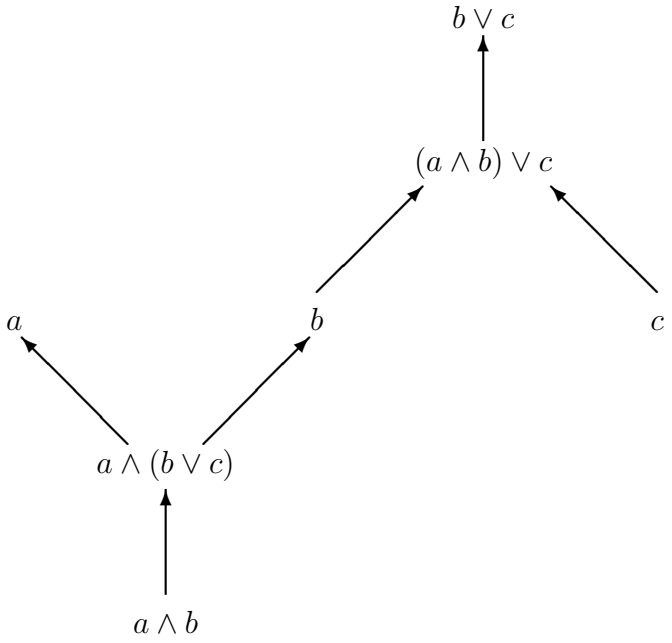


Figure 2.1: The Poset generated in Example 2.1.

lattice, without violating the order relation i.e., we have to find a homomorphism.

We will solve this problem by artificially introducing two elements $\top$ and $\bot$, which results in a bounded poset. A bounded poset is always a lattice. Assume that we have greatest elements $o_1, \ldots, o_k$ and smallest elements $u_1, \ldots, u_l$ with $k, l > 1$ in a poset generated by the procedure above. We add an artificial greatest element $\top$ and an artificial smallest element $\bot$ to the poset together with $o_i \leq \top (1 < i \leq k)$ and $\bot \leq u_j (1 < j \leq l)$. The resulting poset is a bounded lattice where the old order relation between two terms $u, v$ with $u, v \notin \{\top, \bot\}$ is preserved. The preservation of the order relation is obvious, because we only add some $\leq$-relations which have no influence on the already existing relations. This bounded lattice is the counter lattice we

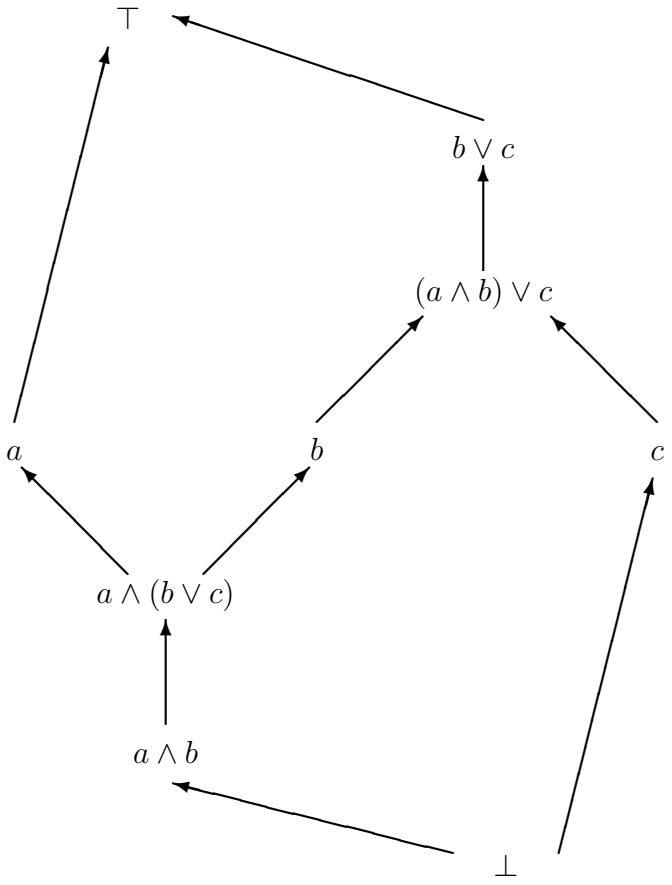


Figure 2.2: The poset generated in Example 2.1 with $\top, \bot$-introduction.

have been looking for. Figure 2.2 shows the extended poset of Example 2.1. We will further on call this extension method $\top, \bot$-introduction.

The $\top, \bot$-introduction has one drawback: it increases the size of the counter lattice, which makes it less convincing for a human user. This problem and its solutions will be discussed more detailed in Chapter 4.

### 2.3.2 Counter Ortholattices

In [1] Egly proves that GOL is sufficient for building an orthoposet as above, which can be extended with $\top, \bot$-introduction as above to a counter ortholattice. Recall that for proof-search, the strict subformula property is admissible. But for the counter ortholattice generation it is not. The polarity restriction must be omitted as in GL. For counter ortholattices we have to introduce weaker forms of the subformula property. In the following we will define the *weak subformula property* and the *highly weakened subformula property*. The reason for this can be depicted by the following problem:
Given some orthologic formula $f$. Not necessarily all subformulas $x$ of $f$ also appear as complemented subformula $\neg x$. This means there might exist some subformula $x$ for which $\neg x$ is not a subformula of $f$. Assume this $x$ appears in some sequents, then we know its relations to other elements of the counter ortholattice we are trying to generate. But because of the subformula property $\neg x$ never appears in a sequent we will derive. If we never derive a sequent with $\neg x$, we know nothing about its relation to other elements. Recall that for an ortholattice it must always hold that whenever $x \leq y$ holds then $\neg y \leq \neg x$ must hold. This means if we just translate the derived sequents as for GL, we will have some relations $x \leq y$ in our counter lattice, but not $\neg y \leq \neg x$. Thus, we would not have an ortholattice.
That is why we have to extend the set of allowed subformulas to appear in sequents. Let $\mathrm{sf}(x)$ denote the set of all subformula of a formula $x$. Note that we skip all polarity considerations for the counter ortholattice generation. Let $s \vdash t$ be the non-provable input sequent for which we want to build a counter ortholattice. We define the set $\Sigma$ to be $\mathrm{sf}(s) \cup \mathrm{sf}(t)$. The set $\Sigma$ is the set of possible subformulas, that might appear in a sequent during the proof-search using the subformula property without polarity restriction. We now have to extend this set in order to generate a counter ortholattice. It is important to note that we want to extend this set as minimal as possible, because the efficiency of the forward proof-search depends on the restric-

tion of this set. We extend $\Sigma$ to the set $\Theta(\Sigma)$ which is defined as follows: $\Theta(\Sigma) = \Sigma \cup \{\neg x, \neg\neg x | x \in \Sigma\}$. The set $\Theta(\Sigma)$ defines us the set of allowed subformulas to appear in sequents for the *highly weakened subformula property*. Recall that for ortholattices $\neg\neg x = x$ holds. Thus, extending $\Sigma$ by more than $\neg\neg x$ would be senseless, it would only decrease the efficiency of the proof-search. An important question is, do we even need the double negation? To answer this question we need some definitions first.
We define the set $\Psi(\Sigma)$ as $\Psi(\Sigma) = \Sigma \cup \{\neg x | x \in \Sigma\}$. This means we extend $S$ only by adding the negated form of each subformula. The set $\Psi(\Sigma)$ defines us the set of allowed subformulas to appear in sequents for the *weak subformula property*. Recall the definition of a saturation $\sigma(X)$ on page 10, where the set $X$ determines the formulas that are allowed to occur in the sequents. Let $\epsilon(Y)$ be a function defined on a set $Y$ of sequents, which replaces all occurrences of a formula $\neg\neg x$ by $x$. For example applying $\epsilon(\{\neg\neg x \vdash y\})$ results in $\{x \vdash y\}$.
With the above definitions the following lemma expresses that we only need the weak subformula property. This is because all sequents $\neg\neg x \vdash y$ derived using the highly weakened subformula property, are already derived as $x \vdash y$ by the weak subformula property.

**Lemma 2.1** $\{\sigma(\Psi(\Sigma))\} = \epsilon(\{\sigma(\Theta(\Sigma))\})$.

*Proof.* The only way to derive $\neg\neg x$ is to derive it from $x$ using the negation rules. Thus, when generating the saturation $\neg\neg x$ is first derived from the axiom $x \vdash x$ in the sequents $\neg\neg x \vdash x$ and $x \vdash \neg\neg x$. But no rules can be applied on $\neg\neg x$, because neither $\neg\neg x \vee a$ nor $\neg\neg x \wedge a$ are in $\Theta(\Sigma)$. If they are, then they are also in $\Sigma$, because $\Theta(\Sigma)$ is constructed from $\Sigma$ by only adding negations, not $\vee$ or $\wedge$. This means all sequents that are derivable from $\neg\neg x \vdash x$ and $x \vdash \neg\neg x$ are only derived by rule application on $x$. But these rule applications can also be applied on $x \vdash x$. Thus, for every sequent $\neg\neg x, M \vdash N$ also $x, M \vdash N$ is derived. This also holds for the acut rule. $\square$

There is also another argument why the highly weakened subformula property is not required. If we allow $\neg\neg x$ to appear in sequents, then $\neg\neg x \vdash x$ and $x \vdash \neg\neg x$ are always derived. But if we interpret this sequents as $\leq$-relations, we get the equivalence of $x$ and $\neg\neg x$. Thus, when we check for relational equivalences, $x$ and $\neg\neg x$ are members of the same equivalence class and thus, only one prototype is used for both of them.

Again as in GL the idea for generating the counter ortholattice is generating a saturation with GOL and then translate it into $\leq$-expressions. Recall GOL is only restricted to 2-restricted sequents, which are not necessarily 1-1-sequents. This means we can encounter sequents of the form $(\vdash a, b), (a, b \vdash), (a \vdash)$ and $(\vdash a)$. Thus just translating the sequents with $\vdash$ interpreted as $\leq$ does not work. Here the previously introduced translation $\delta_{1-1}$ helps us. It allows us to translate arbitrary 2-restricted sequents into 1-1-c-sequents. This means before we want to interpret the sequents of the saturation as $\leq$-expressions, we first have to apply $\delta_{1-1}$ on all sequents, such that all sequents of the saturation are 1-1-c-sequents. Such 1-1-c-sequent can be translated into $\leq$-expressions. We define the following translation $\tau(s)$ which translates a sequent into a $\leq$-relation.

$$\tau(s) = \begin{cases} x \leq y & \text{if } s = x \vdash y; \\ x \leq \neg x & \text{if } s = x \vdash x^c; \\ \neg y \leq y & \text{if } s = y^c \vdash y; \\ x \leq \neg y & \text{if } s = x \vdash y^c; \\ \neg x \leq y & \text{if } s = x^c \vdash y; \end{cases}$$

With the above we would already be able to generate our counter ortholattice, but recall that we stated that the subformula property is important for the efficiency of GOL. Thus the weak subformula property might lead to a decrease in speed, because more subformulas and therefore more possible sequents must be considered for building the saturation. This problem could be avoided, if we find a way to efficiently translate a GOL-saturation with subformula restriction into a GOL-saturation with weak subformula restriction, such that the $\leq$-expressions defined by both saturations are equal. This

is done by adding the sequent $\neg y \vdash \neg x$ for every sequent $x \vdash y$. We define the following extension $\gamma(S)$ which extends a saturation $S$ as follows:

$$\gamma(S) = S \cup \{\neg x \vdash \neg y | y \vdash x \in S\}.$$

**Theorem 2.3** *Let $f$ be an invalid orthologic formula. Let $A = \{\sigma(\Psi(\Sigma))\}$ be the set of sequents of the saturation generated by GOL with weak subformula restriction for $f$ and let $B = \{\sigma(\Sigma)\}$ be the set of sequents of the saturation generated by GOL with subformula restriction for $f$ as above. Let $S$ be the set of 1-1-c sequents that results from applying $\delta_{1-1}$ on all sequents of $A$ and let $T$ be the set of 1-1-c sequents that results from applying $\delta_{1-1}$ on all sequents of $B$. Let $U$ be the set of sequents that results from applying $\gamma$ on all sequents of $T$. Let $R$ be the set of $\leq$-relations that results from applying $\tau$ on all sequents of $S$ and let $V$ be the set of $\leq$-relations that results from applying $\tau$ on all sequents of $U$, then $R = V$ holds.*

*Proof.* Obviously $R$ contains at least the $\leq$-expressions of $V$ because $B \subseteq A$ holds. We proof the other direction by showing for all cases of sequents $s$ with $s \in A$ and $s \notin B$ that the $\leq$-relations that result from s in $R$, are also in $V$.

1: $s = \neg x \vdash \neg y$ with $\neg x, \neg y \notin \Sigma$ but $x, y \in \Sigma$ and $\neg x, \neg y \in \Psi(\Sigma)$. The sequent $s$ results in $\{(\neg x \leq \neg y)\}$ for $R$. Because $x, y$ are in $\Sigma$, all relations that hold for $x$ and $y$ must be derived. Recall that for all ortholattices must hold that if $\neg x \leq \neg y$ holds than also $y \leq x$ must hold, thus $y \leq x$ must also be in $R$. This means that the sequent $y \vdash x$ must occur in $A$. But recall that $x, y$ are in $\Sigma$, thus the sequent $y \vdash x$ must also occur in $B$. Therefor it is in $T$ and when applying $\gamma$ on $T$ we add $\neg x \vdash \neg y$ to $U$. Applying $\tau$ on this sequent results in $\neg x \leq \neg y$ for $V$.

2: s $= \neg x \vdash y$ with $\neg x \notin \Sigma$ but $x, y \in \Sigma$. The sequent $s$ results in $\{(\neg x \leq y)\}$ for $R$. If $\neg x \vdash y$ is in $A$ then also $\vdash x, y$ must be in $A$,

otherwise $\neg x \vdash y$ would not have been derivable. But then $\vdash x, y$ is also in $B$. Applying $\delta_{1-1}(\vdash x, y)$ results in the 1-1-c-sequents $x^c \vdash y$ and $y^c \vdash x$. The application of $\gamma$ has no effect, but $\tau(x^c \vdash y)$ results in $\{(\neg x \leq y)\}$.

3: $x \vdash \neg y$ with $\neg y \notin \Sigma$ but $x, y \in \Sigma$. This is analogous to case 2.

4: $\neg x, y \vdash$ with $\neg x \notin \Sigma$ but $x, y \in \Sigma$. Applying $\delta_{1-1}(\neg x \vdash y)$ results in the 1-1-c-sequents $\{(y \vdash \neg x^c), (\neg x \vdash y^c)\}$. Applying $\tau$ on them results in $\{(y \leq \neg\neg x), (\neg x \leq \neg y)\}$. But if $\neg x$ is not in $\Sigma$ then $\neg\neg x$ is even not in $\Psi$. This means that $s$ results only in $\{(\neg x \leq \neg y)\}$ for $R$. If $\neg x, y \vdash$ is in $A$ then also $y \vdash x$ is in $A$ otherwise $\neg x, y \vdash$ would not have been derivable. But this means that $y \vdash x$ is also in $B$ and thus in $T$. It is easy to see that when applying $\gamma$ on $T$, $\{(\neg x \vdash \neg y)\}$ is added to $U$ and thus $\{(\neg x \leq \neg y)\}$ is in $V$.

5: $\vdash \neg x, y$ with $\neg x \notin \Sigma$ but $x, y \in \Sigma$. This is analogous to case 4. □

To illustrate the generation of a counter ortholattice we give an example.

**Example 2.2** *Given a sequent $\vdash s$ with $s = (a \wedge b) \wedge \neg(a \vee b)$, $\vdash s$ is not provable. We derive the following sequents using GOL in forward search with subformula restriction (logical axioms of the form $x \vdash x$ are omitted):*

1. $a \wedge b \vdash a$
2. $a \vdash a \vee b$
3. $a \wedge b \vdash b$
4. $b \vdash a \vee b$
5. $\vdash \neg(a \vee b), a \vee b$
6. $\neg(a \vee b), a \vee b \vdash$

7. $(a \wedge b) \wedge (\neg(a \vee b)) \vdash a \wedge b$

8. $(a \wedge b) \wedge (\neg(a \vee b)) \vdash \neg(a \vee b)$

9. $(a \wedge b) \wedge (\neg(a \vee b)) \vdash a$

10. $a \wedge b \vdash a \vee b$

11. $\neg(a \vee b), a \vdash$

12. $(a \wedge b) \wedge (\neg(a \vee b)) \vdash b$

13. $\neg(a \vee b), b \vdash$

14. $(a \wedge b) \wedge (\neg(a \vee b)), a \vee b \vdash$

15. $(a \wedge b) \wedge (\neg(a \vee b)) \vdash a \vee b$

16. $\neg(a \vee b), a \wedge b \vdash$

17. $(a \wedge b) \wedge (\neg(a \vee b)), a \vdash$

18. $(a \wedge b) \wedge (\neg(a \vee b)), b \vdash$

19. $\neg(a \vee b), (a \wedge b) \wedge (\neg(a \vee b)) \vdash$

20. $(a \wedge b) \wedge (\neg(a \vee b)), a \wedge b \vdash$

21. $(a \wedge b) \wedge (\neg(a \vee b)), (a \wedge b) \wedge (\neg(a \vee b)) \vdash$

22. $(a \wedge b) \wedge (\neg(a \vee b)) \vdash$

*We now apply* $\delta_{1-1}$ *and* $\tau$ *on the sequents (for the rest of the example we will call this* translation*). Sequents 5. and 6. yield* $\neg(a \vee b) \vdash \neg(a \vee b)$ *which is already an axiom. Note that the other translation* $\neg\neg(a \vee b) \vdash (a \vee b)$ *respectively* $(a \vee b) \vdash \neg\neg(a \vee b)$ *gives us the equivalence of* $\neg\neg(a \vee b)$ *and* $(a \vee b)$*, thus after searching for equivalence classes, we will use* $(a \vee b)$ *as prototype for this equivalence class. Thus in the following translations* $\neg\neg(a \vee b)$ *will be replaced by* $(a \vee b)$ *when building the counter ortholattice. The other sequents not of the form* $x \vdash y$ *are translated to:*

*t11.* $\neg(a \vee b) \leq \neg a$ *;* $a \leq \neg\neg(a \vee b)$ *from 11*

*t13.* $\neg(a \vee b) \leq \neg b$ *;* $b \leq \neg\neg(a \vee b)$ *from 13*

*t14.* $(a \wedge b) \wedge (\neg(a \vee b)) \leq \neg(a \vee b)$ *;* $a \vee b \leq \neg(a \wedge b) \wedge (\neg(a \vee b)))$ *from 14*

*t16.* $\neg(a \vee b) \leq \neg(a \wedge b)$ *;* $a \wedge b \leq \neg\neg(a \vee b)$ *from 16*

*t17.* $(a \wedge b) \wedge (\neg(a \vee b)) \leq \neg a$ *;* $a \leq \neg((a \wedge b) \wedge (\neg(a \vee b)))$ *from 17*

*t18.* $(a \wedge b) \wedge (\neg(a \vee b)) \leq \neg b$ *;* $b \leq \neg((a \wedge b) \wedge (\neg(a \vee b)))$ *from 18*

*t19.* $(a \wedge b) \wedge (\neg(a \vee b)) \leq \neg\neg(a \vee b)$ *;* $\neg(a \vee b) \leq \neg((a \wedge b) \wedge (\neg(a \vee b)))$ *from 19*

*t20.* $(a \wedge b) \wedge (\neg(a \vee b)) \leq \neg(a \wedge b)$ *;* $a \wedge b \leq \neg((a \wedge b) \wedge (\neg(a \vee b)))$ *from 20*

*t21.* $(a \wedge b) \wedge (\neg(a \vee b)) \leq \neg((a \wedge b) \wedge (\neg(a \vee b)))$ *from 21*

*Sequent 22. has the same translation as sequent 21. The translation of the* $x \vdash y$ *sequents into* $\neg y \vdash \neg x$ *sequents is trivial and we do not write them down explicitly. Taking all* $\leq$*-expressions together, omitting reflexivity and transitivity yields the orthoposet illustrated as a Hasse-diagram in Figure 2.3.*

It is interesting to note that in the above example, a smallest and largest element already exist and therefore this orthoposet is already an admissible ortholattice. Also note the symmetry between the elements and their complements, which is typical for ortholattices. As for lattices, in Chapter 4 we discuss how to reduce the size of the counter ortholattice in Chapter 4.

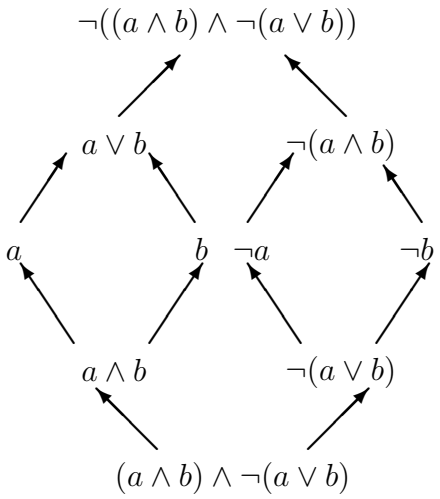


Figure 2.3: The Orthoposet generated in Example 2.2.

# Chapter 3

# Implementation

After discussing the theoretical background of counter lattices in the previous chapter, this chapter is dedicated to the implementation part. That is the implementation of a modified GOL version for the generation of counter (ortho)lattices. This implementation is called CGOL for counter example GOL. We will first focus on the goals and basic ideas of CGOL. Afterward we will discuss the details and the important modules CGOL consists of. The chapter will be concluded with a section about experimental results of CGOL.

## 3.1 Goals and Basic Ideas

The main goal is to implement a computer program, which reads arbitrary orthologic formulas and $\leq$-expressions of lattice theory and then proves whether they are valid or invalid. If the result is valid, it shall print the generated proof. If the formula is invalid, it shall generate a counter lattice (the term counter lattice respectively lattice will be synonymously used for both pure lattices and ortholattices). The counter example should convince a human user that the formula does not hold at least in this lattice.

Due to the requirement of handling pure lattice $\leq$-expressions, CGOL reads the input formula in sequent notation $x \vdash y$ with $x, y$ being formulas where $x$ can be empty, thus enabling it to handle $\leq$-expressions for orthologic too, except for $\vdash x, y$ and $x, y \vdash$. Because the syntax and rules of GL are a real subset of GOL and the possibility of inputing $\leq$-expressions, CGOL is simultaneously a proof system for pure lattice and orthologic.

Efficiency issues are of high priority for an automatic proof system. This is very important for the choice of the programming language, but portability issues are also of great importance. CGOL should be portable enough to run on the three major OS-platforms: Linux/Unix, Windows and Mac OS. The demand for speed and portability led to the decision to use the programming language C, or more precisely, to use the strict ANSI-C standard. All functionality is implemented using only the standard library functions as defined in [5] and the open source library popt.lib, which is freely available for all three major platforms. This library is used to simplify the handling of command line arguments.

CGOL is a command line program only. A user interface with sophisticated drawing of lattices is currently under work in the course of a master thesis of Georg Ziegler [8] at the Knowledge-Based Systems Group at the Vienna University of Technology. This is also the reason why CGOL is not intended to draw the counter lattices. It only generates a file with the information needed for drawing the counter lattice. This file is used as an interface to Ziegler's user interface. It has the suffix `.lat` and the grammar looks as follows in an EBNF like language:

$$
\begin{aligned}
\textit{File} &\rightarrow \text{“(”}\ (\textit{Vertex})^*\ \text{“)”}\ (\textit{Formulalist}) \\
\textit{Vertex} &\rightarrow \text{“(”}\ \textit{Index}\ \text{“(”}\ \textit{Cover}\ \text{“)”}\ \text{“)”} \\
\textit{Index} &\rightarrow (\ (\mathbf{c})?\ ([\mathbf{0\text{-}9}])+) \mid\mid \mathbf{top} \mid\mid \mathbf{bot} \\
\textit{Cover} &\rightarrow \textit{Index}\ (<\text{blank}>\ \textit{Index})^* \mid\mid <\text{blank}>
\end{aligned}
$$

$$
\begin{aligned}
\textit{Formulalist} &\rightarrow \mathbf{Subformula:}\ \textit{Index}\ \mathbf{is:}\ \textit{Formula}\ \textit{Formulalist} \mid\mid \epsilon \\
\textit{Formula} &\rightarrow \textit{Atom} \mid\mid \text{”(”}\,\textit{Formula}\ \textit{Con}\ \textit{Formula}\,\text{”)”} \mid\mid \text{”}\sim\text{(”}\,\textit{Formula}\,\text{”)”} \\
\textit{Con} &\rightarrow \text{”}\&\text{”} \mid\mid \text{”}|\text{”} \\
\textit{Atom} &\rightarrow ([\mathbf{a\text{-}z}])+
\end{aligned}
$$

Because user-interaction is not a requirement, CGOL is always triggered once with the according command line arguments, makes one run, generates the output and then terminates execution. This is also the reason why CGOL does not read an input formula during the runtime through ”stdin”, but only by reading a file containing the formula. Output is also printed to a file, with the exception of the proof structure itself, which can also be printed to ”stdout”.

Due to speed considerations, CGOL can be run in two different modes: First in ”classical” GOL mode with polarity restriction in forward search, which is the fastest mode, but which has the drawback that such a run does not derive enough information for a counter lattice. That is, the ”classical” mode can only output the proof structure together with the result if the searched sequent was found or not, that is if the formula was valid or not. If the user wants a counter lattice, he has to run CGOL in the ”no polarity” mode. Here, the inferences are made using GOL without polarity restriction in forward search. For valid formulas, this mode also generates a proof. For invalid formulas, it derives enough information for a counter lattice and automatically constructs one, printing the information needed to draw the lattice out to a file designated by the user. In Chapter 4, we will also introduce an extension mode for CGOL, which allows to reduce the size of the counter lattice generated in the ”no polarity” run. The suggested procedure for users is: first

run CGOL in "classical" mode on the input formula to quickly test whether the formula is valid or not. If it is invalid rerun CGOL in "no polarity" mode. Although it is slower than "classical" mode, the "no polarity" mode is also polynomial bounded in the worst case. If the counter lattice is to large to convince the user, it is recommended to use the size reduction extension which will be introduced in Chapter 4.

The program logic of CGOL is split into three major parts (modules): input, inference and counter lattice. As the name suggests, input is designated for parsing the input sequent. The parser also checks for the syntactical correctness of the formula and if it is not syntactically correct, it rejects it terminating the program. We will introduce the allowed syntax later on. Input splits the input sequent up in its subformulas such that inference can then build a lookup data structure. The data structure contains the information which subformulas can be generated by a specific subformula using the operations $\neg, \wedge, \vee$.

Based on the lookup data structure that is derived from the subformulas of the input sequent, inference implements the application of GOL. It first generates the lookup data structure. Then it builds the list of axioms, by looking up which subformulas occur in both polarities, or in "no polarity" mode by just taking all subformulas. The axioms are inserted into a proof data structure. Working on this proof data structure it goes through the already generated sequents and applies the rules of GOL with the help of the lookup data structure from input. If the last sequent has been tried for generating new sequents and no more new sequent was derivable, the run is finished and the proof structure is complete. That is, the searched sequent was not derived, thus the input sequent is invalid. If the searched sequent is derived during the run, the run finishes and the result is that the input sequent is valid. In "classical" mode, we always finish the program now and output the proof structure, but in "no polarity" mode, if the input sequent was invalid, the counter lattice part of CGOL is invoked.

Counter lattice takes the proof structure from inference and builds a data

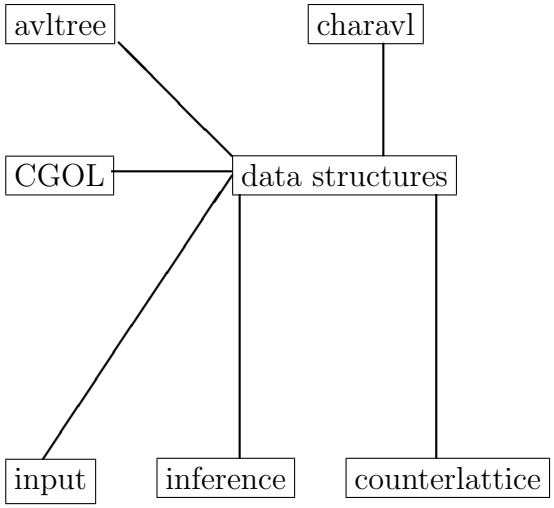


Figure 3.1: The splitting in the different modules of CGOL

structure which stores the $\leq$-relation. This data structure is realized as an array of AVL-trees. Each entry in the array is for one element of the lattice and the elements relations to other elements are stored in the corresponding AVL-tree. This data structure allows us to quickly search for equivalences. The relation is given by the derived sequents, but as CGOL is an implementation of GOL, sequents of the form $\vdash x$, $\vdash x, y$, $x \vdash$, $x, y \vdash$ can be generated. Those sequents must be translated to GOL-1-1 sequents first and then be filled into the data structure.
In the data structure, it is easy to find relational equivalences, i.e., if $x \leq y$ and $y \leq x$ holds. Those equivalent subformulas are merged to one equivalence class. It is also easy to find out in the data structure if a smallest respectively largest element for the lattice already exists. If a smallest respectively largest element is needed, one is artificially inserted and the counter lattice is printed to a file in form of a list of its $\leq$-relations, which is sufficient to reconstruct the lattice and draw it. In Chapter 4, counter lattice will be extended to reduce the size of the generated lattice.

In Figure 3.1, the splitting of the program into its modules can easily be

seen. Note that additionally to input, inference and counterlattice, there exist the modules CGOL, avltree, charavl and data structures. The module data structures defines the data structures used in the program and the declarations of the functions which are used as interfaces between the modules. CGOL contains the main function which is the programs entry point in C. It also implements the functionality for reading the command line arguments and controls the flow of the program, according to the command line arguments. Avltree and charavl are the implementations of the well-known AVL-tree (balanced search tree) data structure, which is used quite often within the program for fast lookups. AVL-trees have the property that inserting and searching an element is done in $O(\lceil ld(n) \rceil)$, with $n$ the number of elements in the AVL-tree, which is very fast. What the figure does not show is that, for compiling CGOL, the open source library popt.lib and the ANSI-C standard libraries are needed.

## 3.2 Overview on the Implementation Details

In this section, we will take a close look on the implementation and describe the used algorithms in a detailed way. Algorithms are presented in an intuitive pseudo-code with a touch of C-style. In the implementation, the logical symbols of $\wedge, \vee, \neg$ are substituted by `&,|,~`, because unlike $\wedge, \vee, \neg$ they can be found on a standard keyboard, which makes programming far more easier.

### 3.2.1 Module Input

The module input is just a simple parser, that constructs a formula tree out of the input sequent. From this formula tree it generates a list of subformulas, which the module inference requires. We will only discuss the data structure for the list of subformulas, because it is important for the next section. We

will also show the allowed syntax for the input sequent.
The subformula list is a single-chained list. Each element has a pointer to its successor element. It also has a numeric value as a unique key, which will be used as unique identifier for this subformula throughout the whole run of CGOL. To understand how a subformula is represented by the element, we have to recall how a subformula looks like. A subformula is either an atom, or a topmost binary connective such as $\wedge$, $\vee$ and $\neg$. If the topmost connective is binary it has two subformulas which are the two *operands* of the connective. We also have the special case of $\neg$ which is unary, thus it has one subformula as its operand. Thus, the element of the subformula list stores the topmost connective of the subformula and two pointers which point at the subformulas which are the operands of the connective. If the connective is $\neg$ one of the pointers is NULL. In case of an atom, both pointers are NULL, but the string representation of the atom is stored.
The syntax for the input sequent is firmly restricted:

$$
\begin{aligned}
\textit{Input Sequent} &\rightarrow \textit{Formula}\ \text{“<”}\ \textit{Formula} \\
\textit{Formula} &\rightarrow \text{“(”}\,\textit{Atom}\,\text{“)”}\ \|\ \text{“}\sim\text{(”}\,\textit{Formula}\,\text{“)”}\ \| \\
&\quad\ \text{“}\&\text{(”}\,\textit{Formula}\,\text{“,”}\,\textit{Formula}\,\text{“)”}\ \| \\
&\quad\ \text{“}|\text{(”}\,\textit{Formula}\,\text{“,”}\,\textit{Formula}\,\text{“)”} \\
\textit{Atom} &\rightarrow ([\mathbf{a}\text{-}\mathbf{z}])+
\end{aligned}
$$

The formula in front of the ”<” is the antecedent and the other formula the succedent of the input sequent. Note that the parenthesises are mandatory.

### 3.2.2 Module Inference

The module inference requires a list of subformulas as constructed by the module input and builds a lookup data structure that contains all information that is needed for generating a GOL saturation as defined in Chapter 2. For the module inference we need two data structures, one for representing the

saturation and one for fast lookups. We first describe the data structure for the saturation. The saturation is internally represented as a proof data structure, which consists of a single chained list of levels of inference. We define the *level of an inference* as follows:

1. Let the sequent $S$ be an axiom. Then $\text{level}(S) = 0$.
2. If $S$ is a sequent directly derived from a sequent $T$ then $\text{level}(S) = \text{level}(T) + 1$.
3. If $S$ is a sequent derived from the sequents $T$ and $U$ with a binary rule, then $\text{level}(S) = \max(\text{level}(T), \text{level}(U)) + 1$.

Every level of inference contains a single chained list of sequents which were generated from sequents of the previous level and a pointer on the next level of inference. Note that we keep track of generated sequents in a lookup AVL-tree where we can quickly check if a sequent already exists. If a sequent already exists, it is not generated again. This means that every sequent appears uniquely in the proof structure. A sequent itself consists of two subformula and their polarity i.e., on which side of the $\vdash$ they appear and a pointer on the next sequent in the list. We will call this next sequent the *successor-sequent*. Recall that one subformula might be empty. The sequent also contains a unique identifier and information about the rule that was applied to generate it and from which other sequent it was derived. This information is necessary for printing a detailed proof for the user, such that he can check if the proof really was correct. Building the saturation is done using the following simple forward search algorithm:

```
saturate(lookups)
{
  build axioms(lookups);
  loop
  {
    weakening(actual-sequent, lookups);
    alpha(actual-sequent, lookups);
    negation(actual-sequent, lookups);
    beta(actual-sequent, lookups);
    actual-sequent = successor-sequent;
  } while(actual-sequent has a successor-sequent)
}
```

Although the sequents are not in one list, but a number of lists, i.e., for each level of inference there is one, it is possible to iterate through the sequents, as if they are in a single list. This is possible because the levels are ordered and every newly generated sequent is inserted in the next higher level as the current one and in the level always on the last free position. So when the proof is traversed level-wise in the proper order, it holds that a sequent generated later always appears after an earlier generated one. The termination of the algorithm is obvious. Every sequent is generated only once, because we keep track of the generated sequents. Because of this, we are able to prevent further generations. Every sequent is only used once to infer all directly derivable new sequents from it, because we run through the list of sequents only once. Note that the sequent can be reused for $\beta$-rules, but this does not mean that we directly derive new sequents from it. It means that we might need the existence of the sequent as justification to directly derive new sequents from another sequent by using a $\beta$-rule. For 2-restricted sequents the number of sequents that are possible to be generated is $O(n^2)$, see [2]. This means that once all possible sequents are generated, no new sequents are generated that is inserted in the list and thus the end of the list will be reached sometime, thus terminating the loop. Later on we will see

that the required time is polynomially bounded in the worst case.
The second important data structure are the lookup AVL-trees. To implement the lookups efficiently, we require a data structure where the lookup and insert operations are very fast, because they are needed very often. The data structure is built only once and there is no need for the deletion of single elements or merging two data structures together. The well-known AVL-trees, also known as balanced search trees, have these properties. Lookup and insertion of a new element are done in logarithmic time. The great drawback of AVL-trees is the deletion of an element, but this operation is not required for our purposes. A node in the AVL-tree has a pointer to its parent node and one pointer for each of its two child nodes. It also has a numerical value as its unique key. The node also contains a "void " pointer to point on some data that is to be stored under its key, we will call this pointer the "data pointer". The data structure this pointer points at depends on the purpose we use AVL-tree for. For the lookups the inference rules of the module inference requires, the data structure is a list of numeric values.
The implementation of the inference rules is done as follows. We build one AVL-tree for every connective i.e., one for $\wedge$, one for $\vee$ and one for $\neg$. In a node of an AVL-tree, we store the key of a subformula as the key of the node. Then we let the data pointer point at a list of the keys of all subformulas that are derivable from this one subformula by using the connective for which the AVL-tree stands for, i.e., if a node in the $\wedge$-AVL says the key is $x$ and the list is $\{y, z\}$, then read it as: "Subformulas $y$ and $z$ can be derived from subformula $x$ using connective $\wedge$". But for $\beta$-rules, we need one more information, namely if the second sequent exists. This information is contained in the AVL-tree in which we keep track of the generated sequents.
The application of inference rules is the core of CGOL and happens very often. Because of this, it is of greatest importance to implement them very efficiently by using an important property of GOL, the subformula property. Recall that we showed in Chapter 2 that we can keep the subformula property for the counter lattice, but we have to add certain sequents. Which sequents

have to be added was already explained in detail in Chapter 2. Recall that CGOL has a polarity restricted mode and a no polarity restricted mode. In polarity restricted mode as well as in no polarity restricted mode, every formula appearing in any sequent at any time, is restricted to be a subformula of one of the possible two formulas of the input sequent. Furthermore every subformula is either an atom, or the result of an application of $\wedge$ respectively $\vee$ on two other subformulas or the application of $\neg$ on one subformula. Now the inference rules, except for the structural rule of weakening, are just the application of one of the above connectives on one subformula, for binary connectives introducing another subformula, to generate a new subformula. This leads us to the following idea illustrated on a little example:
Given a sequent $a \vdash N$ and an input sequent $s \vdash t$ we want to prove. We want to apply the inference rule $\wedge l$ on $a$. This is done by looking into the set of all allowed subformulas and find the set $S$ with $S = \{(a \wedge x) | (a \wedge x), x \in \mathrm{sf}(s) \cup \mathrm{sf}(t)\}$. For every element of $S$, we build a new sequent with $x$ replaced by the element from $S$.
The other $\alpha$-rule $\vee r$ is implemented in exactly the same way but on a sequent $N \vdash a$ and of course using the $\vee$ operator. It is also easy to extend this idea to the negation- and the $\beta$-rules, but $\beta$-rules are binary rules and require a second sequent to be applied. This means for example that, if we try to apply $\vee l$ on a sequent $a \vdash N$ and we found in our subformulas that it would be possible to build the sequent $a \vee b \vdash N$ out of it, we first have to check that the sequent $b \vdash N$ already exists, otherwise the $\beta$-rule is not applicable.

Recall that in Chapter 2 we stated that for GOL the subformula property holds in a strict sense. We called this the polarity restriction. For a "classical" run we want to use this restriction for speeding up our inferences. We do this by using more AVL-trees. Like the information of the connective, which was stored as meta-information in the AVL-tree, we also store information about the polarity as meta-information in the AVL-trees. This is done by splitting the AVL-tree for every connective into two AVL-trees, one for the

connective in negative polarity and one for the same connective in positive polarity. Now if we want to apply $\wedge l$, we only look in the AVL-tree labeled as connective $\wedge$ with negative polarity.
There is one question left open, how do we fill this AVL-trees with data? The parsing of the input formula returns a list of all subformulas, where each subformula $A$ has pointers on all subformulas $B$, where $A$ appears as one operand of the upper most connective of $B$, i.e., $B = A \wedge C$ or $B = A \vee C$ or $B = \neg A$. Filling the AVL-trees is quite easy with this data structure:

```
fillLookups(subformulas)
{
  go through list of subformulas
  {
    go through list of pointers
    switch(operator of the pointed at subformula)
    {
      case: not
        insertInAvlTree(Key of actual subformula,
        key of pointed at subformula,
        not-AVL-tree);
      case: and
        analogous but and-AVL-tree
      case: or
        analogous but or-AVL-tree
    }
  }
}
```

In the list of subformulas, the polarity is also known. Thus, it is easy to take care of the polarity by just inserting a conditional statement for the polarity in the above algorithm.
After the logical rules, we discuss the implementation of the structural rules

of GOL, i.e., weakening. It is quite easy to implement it. Weakening in forward search only appears if one of the formulas in a sequent gets empty. This means we consider sequents of the form $x \vdash$ and $\vdash x$. The set $S$ of sequents that is derivable from a sequent $x \vdash$ and input sequent $s \vdash t$ using weakening is $S = \{x \vdash y \mid y \in \text{sf}(s) \cup \text{sf}(t)\}$. For $\vdash x$, $S = \{y \vdash x \mid y \in \text{sf}(s) \cup \text{sf}(t)\}$. Of course, in "classical" mode, we restrict the set to the corresponding polarity. For the implementation, this means that whenever we encounter a sequent where one formula is empty (i.e., the corresponding pointer points to NULL), we go through the list of subformulas and build a new sequent for every element of the list of subformulas.
Although the contraction is a rule that is not required in GOL, we need to implement it. We need it, because GOL operates on sets of formulas. We use contraction to simulate this set property. It is handled whenever a new sequent was generated by one of the above rules. Contraction checks if both formulas of the new sequent are identical. If they are, one of the formulas can be omitted i.e., its pointer is set to NULL.

### 3.2.3 Module Counterlattice

For the module counterlattice we only require AVL-trees and for each a numerical counter of the number of its elements. The AVL-trees are used because insertion and lookup are needed very often, whilst we do not need to delete single elements. The AVL-trees in the module counterlattice are identical to the lookup AVL-trees of module inference but they do not need the data pointer. The key of each node is the data they store. How we use this AVL-trees and what we store in them will be explained in the following.
The module counterlattice is the extension of CGOL for the generation of the counter lattice for a non-provable input formula $f$. This part is only used if the input sequent is non-provable and CGOL was used in no polarity restricted mode. The latter is necessary for acquiring enough information

about the relations that hold for the set $S$ of all subformulas of the input sequent $s \vdash t$, i.e., $S = \mathrm{sf}(s) \cup \mathrm{sf}(t)$. Recall from Chapter 2 that GOL in forward search without polarity restriction and the application of Theorem 2.3 finds those $\leq$-relations that are needed for generating a counter lattice. The information about the $\leq$-relations is contained in the sequents, which are derived during the proof-search. Recall the application of $\delta_{1-1}, \gamma$ and $\tau$ as defined in Chapter 2 on the set of sequents yielded these $\leq$-relations. We will further on call this *translation*. The details about the translation were discussed in Chapter 2.

The module counterlattice requires a saturation as generated by the module inference and translates it into a relational structure of $\leq$-expressions with the above mentioned translation. This is done by going through all generated sequents and applying $\delta_{1-1}, \gamma$ and $\tau$ on them. Afterward relational equivalences in the relational structure are searched and handled. Two elements are equivalent if $x \leq y$ and $y \leq x$ holds. This equivalence relation automatically defines equivalence classes on the elements of the lattice. For such an equivalence class we do not need to store all elements, but simply represent it with one element as its prototype. The elements of an equivalence class always have the same $\leq$-relations regarding all other elements of the lattice. We keep a list of all equivalences we found in form of an AVL-tree. For big formulas the number of equivalences can be quite considerable, that is why the AVL-tree is used.

When all equivalences are handled, we have a poset where $s \leq t$ does not hold, i.e., it holds that $t \leq s$ or $s$ and $t$ are unrelated. In Chapter 2, we already discussed that this poset is not necessarily a lattice. If the poset is not a lattice, we extend it to a lattice by using the $\top, \bot$-introduction as described in Chapter 2. This requires that we find all largest and smallest elements in the poset and introduce for every largest element that it is smaller $\top$ and for every smallest element that it is larger $\bot$. After this step, we have the lattice we were looking for. At the end the lattice is printed in some user friendly way. Lattices are often represented in human readable form by drawing a

Hasse diagram of them. Such a Hasse-Diagram can be constructed out of the $\leq$-relations. CGOL prints them to a file which some lattice drawing program could use to print the lattice in a pretty way.

The module counterlattice is built around a data structure that stores the information about the relational structure. This data structure is an array with $N = (2 * |S|)$ entries. The indizes we will use for the array range from 1 to $N$. Every subformula gets a unique number between 1 and $|S|$ such that the subformula is identifiable by the number. Every orthocomplement of a subformula $I$, which is not in $\mathrm{sf}(f)$ gets the identifier $|S| + i$ where $i$ is the identifier of $I$. We will call such orthocomplements which are not a subformula of $f$ *virtual complement* whereas we will refer to orthocomplements which are a subformula of $f$ as *real complements*. The index of the array is equivalent to the identifier of the subformulas (and their orthocomplements), this means, for example, that the relations for subformula 5 can be found in the array under index 5. We illustrate this by giving the identifiers for the subformulas of Example 2.2. Recall that the input sequent in Example 2.2 is $\vdash s$ with $s = (a \wedge b) \wedge \neg(a \vee b)$.

ID 1 is: $a$

ID 2 is: $b$

ID 3 is: $a \vee b$

ID 4 is: $a \wedge b$

ID 5 is: $\neg(a \vee b)$

ID 6 is: $(a \wedge b) \wedge (\neg(a \vee b))$

Every entry of the array contains two pointers where each of them points to the root of an AVL-tree and a counter of elements for each AVL-tree. Those AVL-trees store the relations of the subformula which corresponds to the index of the entry in the array. One AVL-tree stores the $\leq$-relation and the other the $\geq$-relation. Note the $\geq$-relations are given by the $\leq$-relations, and thus would be redundant, but storing them explicitly helps increasing the speed of some operations. The drawback of this redundancy is the double need for memory.

A problem we have is the question how to find out which subformulas might need a virtual complement and which already have a real complement. This information is already inherent in the AVL-trees which were utilized for the lookup-operations during the generation of the saturation in the inference module, to be more precise in the AVL-tree for the ¬-connective. It stores which subformula is the negation of another subformula. So it only requires a simple lookup to find out if a given subformula has a real complement. The above discussion leads us to a function getNeg:

```
getNeg(formula)
{
  if(formula in negation AVL-tree)
  {
    return corresponding identifier;
  }
  else
  {
    return identifier(formula)+|S|;
  }
}
```

The algorithm for building the counter lattice looks as follows:

```
buildCounterLattice(saturation)
{
  go through sequents in saturation
  {
    switch(sequent scheme)
    {
      case a |- b:
        insert b in '<='-avl of a;
        insert a in '>='-avl of b;
      case |- a,b:
        insert b in '<='-avl of getNeg(a);
        insert getNeg(a) in '>='-avl of b;
        insert a in '<='-avl of getNeg(b);
        insert getNeg(b) in '>='-avl of a;
      case a,b|-:
        insert getNeg(a) in '<='-avl of b;
        insert b in '>='-avl of getNeg(a);
        insert getNeg(b) in '<='-avl of a;
        insert a in '>='-avl of getNeg(b);
      case |-a:
        insert a in '<='-avl of getNeg(a);
        insert getNeg(a) in '>='-avl a;
      case a|-:
        insert getNeg(a) in '>='-avl a;
        insert a in '<='-avl of getNeg(a);
    }
  }
  go through all relations a <= b;
  {
    insert getNeg(a) in '<='-avl of getNeg(b);
```

```
    insert getNeg(b) in '>='-avl of getNeg(a);
  }
  /* handle equivalence */
  /* a is the id of the current subformula
  t the id of the subformula in the avl */
  s = go through relations array
  {
    t = go through '<='-avl of s
    {
      if(s in '<='-avl of t)
      {
        /* t equivalent to s */
        mark t deleted;
        insert ''s = t'' in avl tree
        which stores the equivalences;
      }
    }
  }
}
```

Note that, whenever we insert a new relation, we perform an implicit check whether the relation already exists. Also reflexivity is checked and ignored, i.e., that $a \leq a$ holds is trivial and does not need to be saved or output for the user.

With this data structure filled, the only topic left to deal with is to check if the $\top, \bot$-introduction is needed. If this is the case, perform $\top, \bot$-introduction and then print out the list of $\leq$-relations for the user.

For the $\top, \bot$-introduction, it is necessary to find the set of smallest, respectively largest elements. A subformula $y$ is element of the smallest elements set $Y$, if the counter for the $\geq$-AVL-tree is 0 and the counter for the $\leq$-AVL-tree is greater 0. This means that there is no subformula that is smaller (with respect to the $\leq$-relation) than $y$ but there are subformulas that are

smaller than $y$ and $y$ is not set equivalent to another subformula. If it would be set equivalent to another subformula both counters would have been set to 0. An element $x$ of the set of largest elements $X$ is defined similar but the counters exchanged. If $|X| = 1$, then no artificial $\top$-element is needed and if $|Y| = 1$ no artificial $\bot$-element is needed. In Figure 2.3 for Example 2.2, we see that the subformula $(a \wedge b) \wedge (\neg(a \vee b))$ which has the identifier 6 in the list on page 44, is such a smallest element. The element $\neg(a \wedge b) \wedge (\neg(a \vee b))$ which has the identifier c6 in the list on page 44, is such a largest element. Note that the "c" in front of an identifier like "c6" stands for the c-indicator as in a 1-1-c-sequent. This c-indicator results from the translation to 1-1-c-sequents, which was necessary due to the reasons explained in Chapter 2. On a semantical level we can interpret this "c" as a negation sign in front of the formula. We have now built a lattice out of the poset we had before and this lattice is our searched counter lattice.
To output the lattice in a user-friendly way, we build the list of $\leq$-relations of the lattice and print it to a file. This is done quite easily with the following algorithm:

```
printToFile(M[][])
{
  for(i = 1; i < N; ++i)
  {
    print(subformulaID(i), "(");
    j = go through <-avl
    {
        print(subformulaID(j), ",");
    }
    print(")"\newline);
  }
}
```

For our Example 2.2, this yields the following output:

```
(
(1 (3 c6))
(2 (3 c6))
(3 (c6))
(4 (1 2 3 c6))
(5 (c1 c2 c4 c6))
(6 (1 2 3 4 5 c1 c2 c4 c6))
(c1 (c4 c6))
(c2 (c4 c6))
(c4 (c6))
(c6 ())
)
```

It is quite easy to understand this syntax. The list of relations is a list, enclosed with parenthesis, where each entry is an element together with a list, also in parenthesis, of those elements which are greater than it. The numbers are the identifiers of the subformulas which are given in the list on page 44. The Figure 2.3 shows the relations between the subformulas. For example, in the above, 1 is smaller as 3 and c6, or 4 is smaller as 1, 2, 3 and c6. Recall that the "c" stands for the c-indicator of 1-1-c-sequents. The largest element is the one with an empty list, in our case c6. With this list, it is possible to construct a Hasse-diagram that shows the lattice we constructed, but the drawing of such a Hasse-diagram is not subject of this work. For details on drawing lattices with this representation as Hasse-diagrams see [8].

## 3.3 Runtime Considerations

In this section, we will focus on an approximation of the worst case runtime the important parts of CGOL have. Important for the approximations are the number of sequents that can be generated PSN and the number of sub-

formulas SFN. An approximation for PSN is given in [2] with $O(\text{SFN})^7$, the important fact about it is that it is polynomial. PSN is approximated with a great value, because it also reflects, that a sequent is considered more than once to be derived, because a sequent can be derived from more than one sequent. Note that most possible sequents are not derivable and thus most sequents are not even considered to be derived. Because the above estimations are based on SFN, we want to show that SFN is polynomial in the number of connectives of the input formula. For SFN the worst case is that every connective is a binary one ($\wedge$ or $\vee$).

**Lemma 3.1** *Let* CON*(f) be the number of connectives in a formula f. The number of subformulas* SFN($f$) *is less than* $(2{*}\text{CON}(f)) + 1$.

*Proof.* We show inductively that $\text{SFN}(f) \leq (2{*}\text{CON}(f)) + 1$ in the worst case:
Throughout the whole proof $\wedge$ stands also for $\vee$. It is regardless for the proof which operator is used as long it is a binary one, because a binary connective is the worst case.
Base case:
Consider an atomic formula where $\text{CON}(f) = 0$. Obviously an atomic formula has exactly 1 subformula. This is the same as $2 * (0) + 1 = 1$.
Induction hypothesis: Suppose $\text{SFN}(f) \leq (2 * m) + 1$ holds for all $m \leq n$ where $m$ is the number of connectives of the formula $f$.
Induction step:
We now can assume that for every formula with $n$ or less connectives the induction hypothesis $\text{SFN}(f) = (2{*}n){+}1$ holds. We now show that this also holds for the case of formula $F$ with $n + 1$ connectives ,i.e., that $\text{SFN}(F) = 2 * (n + 1) + 1 = 2 * n + 3$. Consider some formula $X$ with $c$ connectives and another formula $Y$ with $d$ connectives, such that $c{+}d = n$ holds. We connect $X$ and $Y$ using the connective $\wedge$ such that a new formula $F$ is generated with $c{+}d{+}1 = n{+}1$ connectives. We now have to show that $\text{SFN}(F) \leq 2{*}n{+}3$. Applying the induction hypothesis on $X$ yields that $\text{SFN}(X) \leq 2 * c + 1$ and

applying the induction hypothesis on $Y$ yields that $\mathrm{SFN}(Y) \leq 2{*}d{+}1$. $F$ has as subformulas all subformulas of $X$ and $Y$ and additionally $\{X \wedge Y\}$. Thus, $\mathrm{SFN}(F) = \mathrm{SFN}(X){+}\ \mathrm{SFN}(Y){+}1 \leq (2{*}c{+}1){+}(2{*}d{+}1){+}1 = c{+}d{+}c{+}d{+}3$. Recall that $c + d = n$, thus $c + d + c + d + 3 = n + n + 3 = 2 * n + 3$. Hence, $\mathrm{SFN}(F) \leq 2 * n + 3$. □

The above shows that SFN is polynomial with respect to CON. Thus, if we show that some measure is polynomial with respect to SFN, it is polynomial with respect to CON too. We consider the runtime of the two most important parts of CGOL, the generation of the saturation and the construction of the counter lattice.

**Theorem 3.1** *A saturation has a worst case runtime of* $O(\mathrm{PSN}{*}(6{*}ld(\mathrm{SFN}))$.

*Proof.* Every sequent appears exactly once in the saturation we are generating. It might be derived from many sequents, but we make sure it is derived only once. Every sequent is used only once to check which new sequents can be derived from it. Binary rules require "the help" of a second sequent to be applicable. This means when we consider the sequent, the binary rule might not be applicable yet. Let us consider the second required sequent is derived later on. In this case we do not reconsider the first sequent, but when we consider what is derivable from the second sequent, the binary rule must be applicable from the second sequent because it requires the first one as "helping" sequent, which already exists. Thus a sequent never needs to be reconsidered to check whether some new sequents are derivable from it. For each sequent we look at most once into every AVL-tree for applying the inference rules. Thus the inference rules are tried to be applied PSN-times. There are six such AVL-trees and a lookup in one is done in the ld of its size. The size of one AVL-tree is maximally all subformulas, that means a lookup needs at most $\lceil ld(\mathrm{SFN}) \rceil$. The resulting runtime is $O(\mathrm{PSN}{*}(6 * ld(\mathrm{SFN}))$. □

**Theorem 3.2** *The worst case runtime of the counter lattice generation is* $O((ld(\text{SFN}) * (\text{SFN})^2) + ((\text{PSN}) * ld(\text{SFN})))$.

*Proof.* Filling the data structure for the relations takes two *insert operations* per generated sequent ($x \leq y$ and $\neg y \leq \neg x$), means totally 2∗PSN *insert operations*. Every *insert operation* consists of inserting the relation into two AVL-trees. An insert into an AVL-tree is bounded by $O(\lceil ld(\text{SFN}) \rceil)$. Thus, we have a total bound for filling the data structure of $O(2 * (2 * (\text{PSN}) * ld(\text{SFN})))$. For handling the equivalences, the relations have to be completely searched. Together with their complements, there are 2∗SFN entries in the array with possibly 2∗SFN entries in the AVL-tree, thus $(2*\text{SFN})^2$ times a check for equivalence has to be done. Every check requires a lookup in an AVL-tree with is done in $O(\lceil ld(2*\text{SFN}) \rceil)$. The search for the $\top, \bot$-list needs only one check of the counter for every entry of the array, thus 2∗SFN. Printing the relations requires one full run through the entire relations, thus adding another $(2*\text{SFN})^2$. □

The important property about all these approximations is that every expression is polynomial, thus CGOL is polynomially bounded in the worst case!

## 3.4 Experimental Results

The results for this section were gathered from a test on formulas from the literature by McCune in [7] and on 15 randomly generated formulas. The formulas of McCune are three equations E1, E2 and E3 that arose from work on quantum logic by Norman Megill. Megill asked McCune if these equations are provable in orthologic, because he knew that they hold in orthomodular lattices, but he did not know if they hold in ortholattices. In [7] McCune shows his results he achieved on this equations using MACE and EQP. The

formula E1 is invalid whilst E2 and E3 are valid. For formula E2 he found a proof within 4 seconds and for E3 within 22 hours. This shows the need for a fast automatic deduction system for orthologic. The comparison with CGOL shows the efficiency of CGOL and how it is able to fill this gap. CGOL required less than a millisecond for both formulas! Although I have to remark that the hardware I ran CGOL on is much more faster than the hardware McCune used. He only used a 180MHz i686 processor with 128 MB RAM. But even if one considers my hardware 1000 times faster than McCune's, the runtime of CGOL would have stayed below one second on his hardware!
Interesting is the comparison of E1. McCune required about 15 minutes using 84MB to find a counter example. CGOL found a counter example in less than a millisecond requiring only 25KB of RAM. The only difference is that McCune's counter example as Hasse diagram consists only of 10 nodes, whereas the counter example of CGOL consists of 20 nodes as Hasse diagram. But applying the size reduction of CGOL, which will be introduced in the next chapter, CGOL finds in 300 milliseconds a similar counter example with only two nodes more than McCune. Even if you consider the hardware difference this is still considerably faster and the memory usage is more than 1000 times lower.
The test results on the 15 randomly generated formulas is also very interesting. The random generator we used is a simple program that builds up a formula tree using a recursive top-down method. At every node in the tree it randomly decides, whether a connective should be inserted, or the branch of the tree should end here with an atom. The probability of choosing an atom is increasing with increasing depth of the tree, such that the growth of the tree will come to an end. The atoms are chosen randomly from a set of possible atoms. The set was restricted to only 15 distinctive atoms. This should ensure that the atoms appear frequently in the formula tree in many different subformulas and prevent the atoms from appearing in one polarity only. The choice of the connective at every node is also made randomly. With a probability of 0.4 a conjunction is chosen, with same prob-

| name | atoms | $\wedge$ | $\vee$ | $\neg$ | subformulas |
|---|---|---|---|---|---|
| rf1 | 214 | 107 | 106 | 60 | 247 |
| rf2 | 115 | 54 | 60 | 41 | 149 |
| rf3 | 213 | 98 | 114 | 57 | 257 |
| rf4 | 494 | 256 | 237 | 92 | 515 |
| rf5 | 202 | 109 | 92 | 47 | 233 |
| rf6 | 348 | 168 | 179 | 84 | 388 |
| rf7 | 1401 | 713 | 687 | 364 | 1387 |
| rf8 | 411 | 186 | 224 | 97 | 446 |
| rf9 | 970 | 504 | 465 | 239 | 979 |
| rf10 | 2315 | 1177 | 1137 | 612 | 2204 |
| rf11 | 3928 | 1932 | 1995 | 933 | 3564 |
| rf12 | 2132 | 1044 | 1087 | 536 | 2043 |
| rf13 | 3636 | 1851 | 1784 | 883 | 3306 |
| rf14 | 5173 | 2492 | 2680 | 1289 | 4683 |
| rf15 | 6813 | 3344 | 3468 | 1630 | 6051 |

Table 3.1: The characteristics of the test formulas.

ability 0.4 a disjunction is chosen. The negation has only a probability of 0.2. The reduced probability for negation should ensure that the formula tree does not only consist of multiple negations. Table 3.1 shows the characteristics of the test formulas such as the number of atoms, connectives and subformulas. All tests where performed on a PC with an AMD ATHLON XP 2200+ cpu, 512MB RAM using MS Windows XP professional version 2002 as operating system. In this section, we will discuss the observations for the runs of CGOL in "classical" mode and its performance in searching counter lattices. "Classical" mode means the optimized mode where CGOL is only a decision method for the validity of a formula and does not try to generate a counter lattice. In this mode CGOL uses the strict subformula property. The runtime results in classical mode are given in Table 3.2. All

| name | proof | sequents | sflist-t | lookup-t | inf-t | overall-t |
|---|---|---|---|---|---|---|
| rf1 | NO | 3090 | 0.016 | 0.000 | 0.000 | 0.016 |
| rf2 | NO | 1908 | 0.000 | 0.016 | 0.000 | 0.016 |
| rf3 | NO | 1457 | 0.015 | 0.000 | 0.000 | 0.015 |
| rf4 | NO | 15840 | 0.016 | 0.015 | 0.078 | 0.109 |
| rf5 | NO | 4853 | 0.000 | 0.000 | 0.016 | 0.016 |
| rf6 | NO | 6363 | 0.016 | 0.000 | 0.031 | 0.047 |
| rf7 | NO | 140558 | 0.156 | 0.032 | 1.062 | 1.250 |
| rf8 | NO | 7731 | 0.016 | 0.000 | 0.031 | 0.470 |
| rf9 | NO | 61668 | 0.078 | 0.016 | 0.406 | 0.500 |
| rf10 | NO | 213905 | 0.438 | 0.093 | 1.672 | 2.203 |
| rf11 | NO | 595820 | 1.797 | 0.250 | 5.828 | 7.891 |
| rf12 | NO | 228847 | 0.375 | 0.078 | 1.828 | 2.281 |
| rf13 | NO | 607667 | 1.437 | 0.219 | 6.047 | 7.703 |
| rf14 | YES | 1228121 | 3.985 | 0.531 | 14.312 | 18.843 |
| rf15 | YES | 2075696 | 8.516 | 1.438 | 30.156 | 40.110 |

Table 3.2: Results for the randomly generated formulas. The prover uses the classical mode. All times are measured in seconds.

runtimes are measured in seconds. The column "proof" tells whether a proof was found or not. "Sequents" is the number of derived sequents during the proof attempt, "sflist-t" is the time used to build the subformula list. The column "lookup-t" shows the time used for filling the lookup AVL-trees and the column "inf-t" shows the time required for the inference itself. "Overall-t" is the overall runtime of the program. The runtime performance is quite good, even for the largest formula with 6051 subformulas, the run takes only 40 seconds. For formulas with less than 500 subformulas, the run takes less than 0,1 seconds. The non-linear growth of runtime is obvious and depends on the non-linear growth of the number of derived sequents. From formula "rf15" (6051 subformulas) over 2 million sequents are derived, whereas from

| name | ftree-m | sflist-m | lookup-m | existing-m | sat-m | overall-m |
|---|---|---|---|---|---|---|
| rf1 | 25811 | 27900 | 37917 | 86520 | 124000 | 302148 |
| rf2 | 14310 | 16340 | 22990 | 53424 | 76656 | 183720 |
| rf3 | 25546 | 28548 | 40378 | 40796 | 58608 | 193876 |
| rf4 | 57187 | 59580 | 87886 | 443520 | 634176 | 1282349 |
| rf5 | 23850 | 26276 | 38343 | 135884 | 194488 | 418841 |
| rf6 | 41287 | 44176 | 64131 | 178164 | 254944 | 582702 |
| rf7 | 167745 | 163948 | 256334 | 3935624 | 5623288 | 10146939 |
| rf8 | 48654 | 51144 | 77873 | 216468 | 309816 | 703955 |
| rf9 | 115434 | 114636 | 178854 | 1726704 | 2467560 | 4603188 |
| rf10 | 277773 | 263904 | 438060 | 5989340 | 8557440 | 15526517 |
| rf11 | 465764 | 432608 | 718861 | 16682960 | 23834288 | 42134481 |
| rf12 | 254347 | 243772 | 400511 | 6407716 | 9155016 | 16461362 |
| rf13 | 432162 | 401160 | 696339 | 17014676 | 24300048 | 42852385 |
| rf14 | 616602 | 569452 | 1040947 | 34387388 | 49126336 | 85740725 |
| rf15 | 808515 | 739772 | 1365522 | 58119488 | 83029560 | 144062857 |

Table 3.3: Results for the randomly generated formulas. The prover uses the classical mode. The memory consumption is measured in bytes.

"rf4" (515 subformulas) only 15840 are derived. Although "rf15" has only 12 times more subformulas, 131 times more sequents are derived. This lets us conclude that *the number of derived sequents grows quadratically with the number of subformulas of the input formula*.

The memory usage for "classical" mode is shown in Table 3.3. Memory usage is measured in bytes. The column "ftree-m" shows the memory consumption for the formula tree, "sflist-m" for the subformula list and "lookup-m" for the lookup AVL-trees. The column "existing-m" shows the memory consumption for the lookup AVL-tree which contains the already derived sequents, "sat-m" is the memory consumption for storing the saturation and overall-m is the overall memory consumption of the program. It is easy to see that the

proof structure and the AVL-tree for fast lookup which sequents already exist use most memory. The growth of memory usage is similar to the growth of runtime. On the test system, memory was the main restriction for not using larger test formulas, because with approximately 150MB RAM usage, "rf15" is just small enough to fit in the RAM, but having a quadratic growth of memory usage a formula with twice the number of formulas would need approximately four times more memory. Therefore 600MB RAM would be needed which is more than the systems memory. This would mean that the hard disk is used as virtual memory which leads to exorbitant runtime.
We will discuss now the results for the counter lattice generation run. The results for runtimes are given in Table 3.4 with the measures and columns as in Table 3.2. The column "counter-t" shows the time needed to construct the counter lattice. "Nodes" is the number of nodes a resulting Hasse Diagram would have to depict the lattice and "relations" is the number of $\leq$-relations that hold between the elements of the lattice. As expected, the counter lattice generation mode highly increases the number of derivable sequents and thus increases memory usage and runtime. Depending on the formula, between three and five times more sequents were derived. We tested only non-provable formulas, i.e., test formulas "rf1" to "rf13". We did this because only for non-provable formulas, counter lattices can be generated. The time used for inference and the memory for the existing sequents AVL-tree and proof structure is also three to five times larger, which is no surprise due to the larger number of derived sequents. The generation of the counter lattice needs most time, but only approximately twice as long as the inference itself. The motivation for the next chapter, the chapter about size reduction of counter lattices, can be seen easily from the complexity of the resulting counter ortholattices. Even for the smallest formula "rf2", the counter ortholattice has 184 nodes with 1997 relations between them. Although the number of edges is far smaller because transitivity could be omitted when drawing a graph that shows the ortholattice, it is easy to figure out that even this smallest of our test counter ortholattices is too large to be comprehended

| name | sequents | nodes | relations | inf-t | counter-t | overall-t |
|---|---|---|---|---|---|---|
| rf1 | 14479 | 328 | 5865 | 0.078 | 0.110 | 0.203 |
| rf2 | 6890 | 184 | 1997 | 0.031 | 0.047 | 0.078 |
| rf3 | 8196 | 352 | 6047 | 0.047 | 0.046 | 0.093 |
| rf4 | 63674 | 676 | 23769 | 0.453 | 0.703 | 1.203 |
| rf5 | 14250 | 326 | 6095 | 0.062 | 0.125 | 0.203 |
| rf6 | 23465 | 572 | 17512 | 0.156 | 0.188 | 0.359 |
| rf7 | 448190 | 1714 | 140549 | 4.516 | 7.890 | 12.782 |
| rf8 | 42333 | 602 | 22401 | 0.281 | 0.422 | 0.734 |
| rf9 | 201343 | 1294 | 86939 | 1.750 | 2.859 | 4.797 |
| rf10 | 933622 | 2830 | 371503 | 10.812 | 21.016 | 32.843 |
| rf11 | 2241683 | 4538 | 971315 | 35.782 | 79.265 | 118.781 |
| rf12 | 931140 | 2043 | 308973 | 10.782 | 21.312 | 32.953 |
| rf13 | 2275452 | 4192 | 876621 | 36.500 | 83.734 | 123.484 |

Table 3.4: Results for the randomly generated formulas. The prover uses the counter lattice generation mode. All times are measured in seconds.

| name | relations-m | lookup-m | existing-m | sat-m | overall-m |
|---|---|---|---|---|---|
| rf1 | 1269952 | 42381 | 405412 | 581336 | 2352792 |
| rf2 | 595152 | 26206 | 192920 | 277000 | 1121928 |
| rf3 | 701424 | 44746 | 229488 | 330096 | 1359848 |
| rf4 | 5589744 | 97854 | 1782872 | 2551272 | 10138509 |
| rf5 | 1290288 | 42487 | 399000 | 572096 | 2353997 |
| rf6 | 2068768 | 69619 | 657020 | 941928 | 3822798 |
| rf7 | 39028744 | 286398 | 12549320 | 17938744 | 70134683 |
| rf8 | 3722880 | 86113 | 1185324 | 1697096 | 6791211 |
| rf9 | 17388496 | 196822 | 5637604 | 8061672 | 31514664 |
| rf10 | 82257264 | 482332 | 26141416 | 37362512 | 146785201 |
| rf11 | 200703296 | 793629 | 62767124 | 89695616 | 354858037 |
| rf12 | 82856336 | 445903 | 26071920 | 37261872 | 147134150 |
| rf13 | 202181888 | 765795 | 63712656 | 91044264 | 358537925 |

Table 3.5: Results for the randomly generated formulas. The prover uses the counter lattice generation mode. The memory consumption measured in bytes.

by a human user.

The memory usage for the counter lattice generation is given in Table 3.5. Memory usage is again measured in bytes. The columns are the same as in Table 3.3, but "relations-m" shows the memory consumption for storing the $\leq$ and $\geq$ relations. As in "classical" mode, the growth of memory usage is in counter lattice generation mode similar to the growth of runtime. The size of the data structure for the relations is quiet considerable, using more memory as the saturation and the AVL-tree for the already existing sequents lookup.

# Chapter 4

# Reducing the Size of the Counter Lattice

In the previous chapters, we discussed how a counter lattice can be generated. We always stated that this counter lattice should be generated to convince some human user that the inputed formula does not hold in all (ortho)lattices. The solution we proposed is able to build such a lattice, but the problem is that it is only usable on small formulas. For a large formula, the generated counter lattice is too large for a human user to be readable. Thus, it is necessary to somehow reduce the size of the generated counter lattice. This chapter is devoted to this topic. We will first discuss some theoretic background for the reduction. Afterward a section follows about the approach to handle this problem, which is implemented in CGOL. Another section about the results CGOL achieved with this approach will conclude the chapter.

## 4.1 Theoretical Background for Size Reduction

In Chapter 2 we discussed the theoretical background that was needed for generating counter lattices based on GOL, but we have not discussed how we can extend this to reduce the size of the counter lattice. But first we need to introduce the use of non-logical axioms for GOL in the following section.

### 4.1.1 Non-Logical Axioms in GOL

The idea of non-logical axioms in GOL is to assume some sequents as valid although they are not provable. This means we search a proof under the assumption that the relations given in the non-logical axioms hold. It is sufficient for our purposes that we restrict the formulas in the sequents of the non-logical axioms to be subformulas of the input formula. A simple example should illustrate this. We could ask: Does the formula $a \vee \neg b$ hold for all ortholattices? Without non-logical axioms, this obviously does not hold. Now we add the non-logical axiom $b \vdash a$. The resulting question is now: Does $a \vee \neg b$ hold if we assume that $b \leq a$ holds? This is obviously valid and we would find a proof in GOL with these non-logical axioms for $\vdash a \vee \neg b$. For building a saturation in forward search in GOL as we do in CGOL, it is important to note that using non-logical axioms always leads to saturations with a greater number of sequents than the saturation without non-logical axioms. This property is trivial. The use of non-logical axioms does not effect the generation of the logical axioms, which are generated in the same as before. Thus, from the same set of logical axioms the proof-search with non-logical axioms derives the sequents of the proof-search without non-logical axioms and additionally those sequents derivable from the non-logical axioms. Non-logical axioms have one drawback: they require the cut rule. Only for

GOL with logical axioms there exists a cut-elimination theorem which states that the cut rule is not needed. But when we use non-logical axioms this cut-elimination theorem does not hold any longer. The problem can be illustrated by a very simple example. Let us assume the non-logical axioms $a \vdash b$ and $b \vdash c$ with $a, b, c$ being atomic formulas. Because of the transitivity of the $\leq$-relation, $a \leq c$ must hold, i.e., $a \vdash c$ must be derivable. The problem is that GOL without the cut has no rule with which we could derive $a \vdash c$ from these non-logical axioms and any given set of logical axioms. With the cut rule the derivation of $a \vdash c$ is trivial. Thus if we want to use non-logical axioms we have to use GOL+cut further on. But recall that we are only interested in relations that hold on the subformulas of the input formula. Also recall that we restricted the non-logical axioms to consist only of subformulas of the input formula. Thus, the non-logical axioms never introduce new formulas, this means the cut rule is only applied on subformulas of the input formula. That is why we can restrict the cut to the analytic cut rule. Adding the acut rule is not a critical drawback for the efficiency. As we will see in the section on implementation issues, the acut rule can be handled quite easily and efficiently.

### 4.1.2 Basic Idea of Reducing the Size of the Counter Lattice

The intention of finding a counter example is to construct a lattice which shows that a given relation $s \leq t$ does not hold in this specific lattice. For an orthologic sequent $\vdash t$, after translating it into a GOL-1-1 sequent, we can interpret it as the question, if the relation $\neg t \leq t$ does not hold. We start with a given lattice where $s \leq t$ does not hold and which we want to reduce in size. The fact $s \leq t$ does not hold in this lattice means that either $t \leq s$ holds or $s$ and $t$ are unrelated. But for the counter lattice it is of no importance whether $t \leq s$ holds or $s$ and $t$ are unrelated. The only important property is that $s \leq t$ does not hold. Every lattice for which this

holds is equally good as counter example than every other lattice where this condition is met. Thus, it is sufficient to find a new lattice with a map, which is *not* order-preserving except for the circumstance that $s \leq t$ must not hold in the new lattice.
We will construct this new lattice by utilizing GOL with non-logical axioms. Because of the non-logical axioms, more relations hold on the set $S = \mathrm{sf}(s) \cup \mathrm{sf}(t)$. This means that more sequents are derivable in GOL due to the non-logical axioms. If more sequents are derivable, more relational equivalences of the form $a \leq b$ and $b \leq a$ will potentially arise and thus more subformulas may be set equivalent. This reduces the size of the lattice, because fewer subformulas have to be considered when drawing the lattice. But the drawback is that the non-logical axioms could cause that the order-preservation of $s \leq t$ gets violated. In this case however a proof-search in GOL with these non-logical axioms would have as result that the input formula is provable. This is a useful property for us, because the reduction method consists of the following steps:

1. Search a non-logical axiom.
2. Search a proof using this non-logical axiom.
3. If the formula becomes provable, discard the non-logical axiom. Otherwise keep it and repeat the procedure until the lattice is small enough or no new non-logical axiom can be found.

The problems for searching non-logical axioms are discussed in detail later on. We will illustrate the idea of the size reduction via non-logical axioms in Example 4.1.

**Example 4.1** *Given the counter lattice of Example 2.1 on page 18. We add the arbitrary non-logical axioms* $a \vdash b \vee c$ *and* $a \wedge b \vdash c$*. The saturation that results from these non-logical axioms has only four sequents more:*

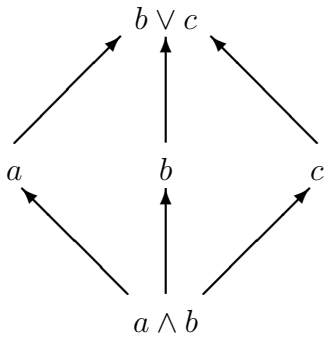


Figure 4.1: The poset generated in Example 4.1.

*19:* $a \vdash b \vee c$ *non-logical axiom*

*20:* $a \wedge b \vdash c$ *non-logical axiom*

*21:* $a \vdash a \wedge (b \vee c)$ *from 0 with 19 using* $\wedge r$

*22:* $(a \wedge b) \vee c \vdash c$ *from 2 with 20 using* $\vee l$

*Sequent 8 is translated to* $a \wedge (b \vee c) \leq a$ *and the new sequent 21 is translated to* $a \leq a \wedge (b \vee c)$*. We see that they are of the form* $x \leq y$ *and* $y \leq x$ *and are thus relational equivalent. This means that* $a$ *and* $a \wedge (b \vee c)$ *are in the same equivalence class. The same holds for sequent 11 and the new sequent 22. They are translated to* $(a \wedge b) \vee c \leq c$ *and* $c \leq (a \wedge b) \vee c$ *and thus* $c$ *and* $(a \wedge b) \vee c$ *are in the same equivalence class. We use* $a$ *respectively* $c$ *as the representative for their equivalence class. Figure 4.1 shows the resulting poset with reflexivity and transitivity omitted.*

In the above example, note that, because of the equivalence, now $a \leq c$ is not allowed to hold. It is easy to see that it indeed does not hold, because $a$ and $c$ are unrelated. Also note that, because of the non-logical axioms, the relational structure changed in a way that now a smallest and largest element already exist and thus the $\top, \bot$-introduction is no longer necessary.

### 4.1.3 Isomorphism

The goal of the counter lattice is to convince a user that $s \leq t$ does not hold for all lattices. How the concrete counter lattice really looks like is not so important as long as it is obvious for the user that $s \leq t$ does not hold. This means that simplifying parts of the lattice that have nothing to do with $s$ or $t$ will not harm the lattice in convincing the user of the property that $s \leq t$ does not hold.

In Figure 4.1 it is easy to see that in the graph elements may appear which are unrelated, but share the same relations with all other elements. In this example this are the elements $a, b$ and $c$. They are all unrelated but are related with the same elements in the same way. Thus, they are isomorph in the graph. The problem is that $a$ and $b$ are the elements $s$ and $t$ for which we want to show that they are unrelated, so any consideration setting them equal is senseless. But for the following assume that they have nothing to do with $s$ and $t$. Setting $a$ and $b$ equal on a logical level might cause that $s \leq t$ becomes provable, but on the representation level for the user, it is unnecessary to have one node in the graph for each element to convince the user that $s \not\leq t$ holds. We could instead represent such isomorph elements with a single node stating that this single node stands for the isomorphism class of all elements with this certain $\leq$-relations.

This idea could be extended to the isomorphism of whole subgraphs, but as stated in [3] subgraph isomorphism belongs to NP although it is not NP-complete. Isomorphism of subgraphs is only polynomial if the graphs are trees. The Hasse-diagram that represents our lattice is a directed acyclic graph though, but [3] show that this does not reduce the complexity of the problem such that it is polynomial.

To stay efficient, CGOL only implements a search for single elements which have the same $\leq$-relations and unites them to an isomorphism class without touching the elements of $s$ and $t$. Checking which elements have the same relations is done quite easily in the data structure of the $\leq$-relations. First

the counters for the number of elements in the AVL-trees are compared. Only if they are equal, the entries in the AVL-trees are checked if they are equivalent. If they are, then the elements are set equivalent. The unification of isomorphic elements is the last step done in the simplification of the lattice, which means it is done only after the size has been reduced using non-logical axioms. We will illustrate the above idea with a simple example.

**Example 4.2** *Let $F$ be the formula $a \vee b \leq a \wedge b$. It is obvious that this formulas is invalid. A possible counterlattice would be the lattice shown in Figure 4.2. The important property of this lattice is that $a \wedge b \leq a \vee b$ holds. The elements between $a$ and $b$ are important to get a correct lattice, but they are not important to increase the convincability of the fact that $a \vee b \leq a \wedge b$ does not hold for the user. Rather the opposite is the case, because the elements distract the attention of the user and make the graph unnecessarily complex. Now look at the graph in Figure 4.3. This is the graph that results from the graph in Figure 4.2 by applying the isomorphism simplification as described above. We see that the fact $a \wedge b \leq a \vee b$ still holds. It also shows the fact that there are the elements $a$ and $b$ between them, but instead of distracting the users attention with two separate nodes, they are represented by only a single node. The new graph is simpler and easier to comprehend for a human user and the important fact that $a \wedge b \leq a \vee b$ holds is easier recognisable. What this example illustrates is also the important difference to the simplification with non-logical axioms. Using non-logical axioms yields equivalence classes on the logical level. The isomorphism simplification is only a simplification of the representation of the lattice. The elements $a$ and $b$ which now share one node are* not logical equivalent*! If they were, $a \wedge b \leq a \vee b$ would no longer hold. So it is important to remember that this is a simplification on the representation of the lattice, not of the relation in the lattice.*

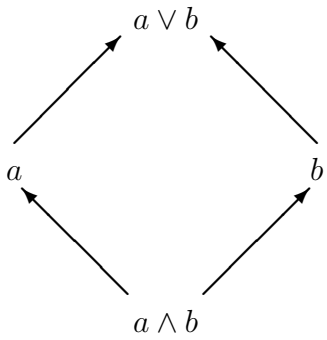


Figure 4.2: The original counter lattice of Example 4.2.

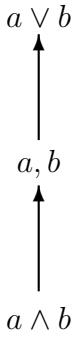


Figure 4.3: The isomorphism simplified counter lattice of Example 4.2.

## 4.2 Size Reduction in CGOL

In this section, we will focus on how the above idea for size reduction is realized in CGOL. The basic idea is always the use of non-logical axioms, but the important question is how non-logical axioms are selected.

### 4.2.1 Implementing Proof-search with Non-Logical Axioms

For being able to handle non-logical axioms, CGOL needs to be extended. First it needs a way to select non-logical axioms. The easiest way is to let the user input the non-logical axioms that shall be used. The user designates a file with the non-logical axioms as $\leq$-expressions. This file is parsed and translated into sequents. Another possibility would be to implement an automated search for non-logical axioms. In the next section we will discuss how such an automated search is realized in CGOL.
After inputting the non-logical axioms, the saturation generation is started as usual with the generation of logical axioms. But before the proof-search is started, the non-logical axioms are inserted into the saturation in addition to the logical axioms. The proof-search itself stays the same, with the exception, that the acut rule is added to the set of inference rules. Remember that all inferences where realized by lookups in AVL-trees to check if a subformula exists that can be generated by the given inference rule. Because of this, we have for every operation in every polarity an AVL-tree, which was filled based on the subformulas of the input sequent. For acut, we also require such an AVL-tree. The problem is that acut operates only on the sequents that where generated and no information about its applicability is in the input formula. Thus the AVL-tree for acut has to be filled during the saturation. We will use four different AVL-trees. One stores the key as being the subformula on the left side and the elements stored under the key being the subformulas on the right side. This AVL-tree will be called "keyleft". The second AVL-tree stores also as key the subformula on the left side, but the elements stored under the key are are subformulas of the left side of the sequent. We will call this AVL-tree "samesideleft". AVL-tree number three is equal to "keyleft" but with changed sides. Which means that the key is on the right side and the elements are on the left. Thus, we will call this tree "keyright". The last AVL-tree is equal to "samesideleft" but of course

on the right and therefore called "samesideright". These AVL-trees are filled as follows: Whenever a sequent of the form $a \vdash b$ is derived (also if it is a logical or non-logical axiom), $b$ is stored under key $a$ in "keyleft and $a$ is stored under key $b$ in "keyright". If a sequent $a, b \vdash$ is derived both are stored in "samesideleft" once as key with the other as element and once as element with the other as key. Case $\vdash a, b$ is handled analogously but with "samesideright". The application of acut with these AVL-trees is divided into the following cases:

1. $x \vdash y$. Looking up in "keyleft" yields all derivable sequents of the form $x \vdash z$, i.e., using $y$ as the cut formula, whereas looking up in "keyright" yields all sequents $z \vdash y$, i.e., using $x$ as the cut formula. Thus, acut is realized by these two lookups and inserting the generated sequents into the saturation.

2. $x, y \vdash$. In this case, we can only cut over sequents of the form $z \vdash y$ and $\vdash z, y$ respectively $z \vdash x$ and $\vdash z, x$. Thus we look up in "keyright" and "samesideright", once for $x$ and once for $y$.

3. $\vdash x, y$. Analogous to case 2, but using "keyleft" and "samesideleft" instead.

### 4.2.2 Automated Size Reduction

The manual input of non-logical axioms requires good guessing and a "feeling" which $\leq$-expressions could be useful and which would make the formula provable and thus the saturation useless for a counter example. This requires that the user spends some effort in investigating the original counter lattice and gets quite familiar with its structure. This might be no problem with a small counter lattice, but if it has hundreds of elements, it is too complicated to be handled in this way. Thus, an automated way to reduce the size of the counter lattice would be very helpful. But how should the non-logical axioms

be selected? A very simple way that surely yields a good result is exhaustive search, i.e., trying every combination of two elements as non-logical axiom. This approach would of course find all useful non-logical axioms, but is useless in praxis. We normally require not only a single non-logical axiom but a number of non-logical axioms. Thus, an exhaustive search would also need to search the whole space of possible combinations of all possible non-logical axioms. For every combination we would require one complete proof-search to test whether it makes the formula provable or not. Thus, it is obvious that this approach works only on small formulas in sensible runtime, because for large formulas the search space of non-logical axioms increases dramatically. Instead, an efficient heuristic might be more useful, i.e., guessing some non-logical axioms, such that a few such "guessing" runs (which should be rather fast) yield a useful set of non-logical axioms that reduces the size of the counter lattice considerably. This means we want to guess non-logical axioms that result in highly increasing the number of equivalences between elements in the lattice. The algorithm is quite simple:

```
autoReduceSize(relationmatrix, endsequent)
{
  while(endflag not set)
  {
    new-axiom = selectAxiom(relationmatrix);
    non-log = addNon-log-axioms(new-axiom);
    result = saturateWithNon-log-axioms(non-log, endsequent);
    if(result == provable)
      non-log = remove(new-axiom);
    new-axiom_invert = invert(new-axiom); //a|-b -> b|-a
    non-log = addToNon-log-axioms(new-axiom_invert);
    result = saturateWithNon-log-axioms(non-log, endsequent);
    if(result == provable)
      non-log = remove(new_axiom_invert);
    if(result == not_better)
```

```
    {
      non-log = remove(new-axiom);
      non-log = remove(new-axiom_invert);
    }
    endflag = checkIfAbortConditionMet();
  }
}
```

Important is the heuristic for selecting the non-logical axiom and the abort condition to guarantee termination. If we cannot guarantee termination our program might get stuck in an endless loop. The heuristic is very simple but useful for our purpose. Its basic idea is the assumption that a non-logical axiom might have most impact on the relational structure, if it affects the element which is in relation with most other elements. Thus, it searches for this element and depending if it is an element which mostly is smaller than other elements it is used as the larger element in the non-logical axiom. The same also works for searching an element which is mostly greater than other elements, but in this case it is used as the smaller element in the non-logical axiom. For the second element, two approaches are implemented. The first one is just searching the element with second most relations, the other is searching the element which has the most similar relations to the first element. We define the *similarity* $\text{sim}(A, B)$ between an element $A$ and $B$ as a numeric value as follows. We initially define $\text{sim}(A, B) = 0$. Let $C$ be an element different from $A$ and $B$. If $A \leq C$ and $B \leq C$ holds or $C \leq A$ and $C \leq B$ holds, then $\text{sim}(A, B) = \text{sim}(A, B)+1$. Using the most similar element is based on the assumption that relating two rather similar elements more easily leads to an equivalence of them. There might be more then one most similar element, i.e., two elements with the same similarity value. In this case it is not important which one we choose, but it is important to choose it in a deterministic way, such that the results of a non-logical axiom search are deterministic. If a non-logical axiom found with this heuristic proves useless, i.e., it improves nothing or, even worse, makes the formula

provable, we first try changing the second element until a certain threshold is reached. After that, we change the first element and then again the second element. This is done until a certain threshold of trials is reached or a useful non-logical axiom is found. If such a useful one is found, the counter for the trials are reset and the heuristic restarts. The search is aborted when the threshold of trials is exceeded. This signals that it is too hard to find a useful axiom. Another termination condition that is also checked is the time already used for the search for non-logical axioms. During the whole automated size-reduction, a timer runs in parallel and, if a certain time limit is exceeded, it stops the search and generates the relational structure for the counter lattice using the last useful set of non-logical axioms. This timer is quite useful for the user. Often, a user just wants to reduce the complexity only roughly but wait only few minutes instead of reducing the size nearly perfect but waiting hours for the result.
Although this approach is simple and naive, it proves useful. The most important aim of the automated size reduction is reducing the size not perfectly but reduce it enough to be easily comprehensible for the user. For this aim, this heuristic is quite useful.

## 4.3 Results for Automated Size Reduction

The test system and formulas for this section are the same as for the results in Section 3.4, but the results of Chapter 3 let us expect a high runtime for the automated search. Thus, we restricted our attention to the "smaller" test formulas, i.e., "rf1" to "rf5", where a timeout of 300 or 600 seconds should already yield useful results. As expected, the runtime increases considerably. One reason is the extra time, the search for the new non-logical axiom requires in each run. The other reason is that, after each run, the memory for the saturation and the data structure for the $\leq$-relations has to be deallocated. Both heuristics for the search for non-logical axioms were tested. Which of

the two approaches is the better is hard to say in general. Both reduced the node and relation complexity enormously. For some formulas, similarity search was better, for others the search for the next most related node was better. Thus, it is recommended to try both approaches for a better result. One property that is shown by the test results is that similarity search is slower. It always made less non-logical axiom search runs in the same time. In general, the size is greatly reduced. Within 300 seconds, most formulas were reduced to approximately one tenth of their original node complexity and a similarly good reduction of their relational complexity. For example, "rf1" is reduced using similarity search in 300 seconds to 20 nodes, which is a number that is useful for a human user. It originally had 328 nodes, which is useless for a human user. It is interesting to note that some formulas can be reduced to a lattice with only two nodes with one relation. Formula "rf2" is such an example. This means that all subformulas can be set equivalent to two equivalence classes using non-logical axioms and that the original $s$ and $t$ of the input sequent $s \vdash t$ are each in one of those with $t \leq s$. The automated search is surely not perfect but proves quite useful in reducing the size to a complexity where it is useful for the user. Table 4.1 shows the results of the similarity heuristic and Table 4.2 shows the results of the next most relations heuristic. All times are measured in seconds as in the results section of Chapter 3. The column "runs" tells the number of axiom search runs. "Startequis" is the number of logical equivalences before the size reduction and "endequis" is the number of logical equivalences after the size reduction. The column "nodes" hows the number of nodes a Hasse-Diagram would need to depict the lattice and "relations" shows the number of relations between the elements of the lattice.

| name | runs | startequis | endequis | nodes | relations | runtime | timeout |
|---|---|---|---|---|---|---|---|
| rf1 | 45 | 139 | 447 | 20 | 85 | 316.953 | 300 |
| rf1 | 55 | 139 | 449 | 18 | 73 | 608.281 | 600 |
| rf2 | 19 | 100 | 282 | 2 | 1 | 68.594 | 300 |
| rf3 | 18 | 136 | 436 | 52 | 367 | 302.813 | 300 |
| rf3 | 32 | 136 | 440 | 48 | 313 | 611.093 | 600 |
| rf4 | 3 | 311 | 933 | 54 | 417 | 592.500 | 300 |
| rf5 | 22 | 129 | 453 | 2 | 1 | 363.735 | 300 |

Table 4.1: Results of automated size reduction similarity heuristic. All times are measured in seconds.

| name | runs | startequis | endequis | nodes | relations | runtime | timeout |
|---|---|---|---|---|---|---|---|
| rf1 | 92 | 139 | 231 | 236 | 2745 | 314.234 | 300 |
| rf1 | 122 | 139 | 239 | 228 | 2597 | 609.454 | 600 |
| rf2 | 25 | 100 | 282 | 2 | 1 | 64.656 | 300 |
| rf3 | 29 | 136 | 462 | 26 | 119 | 318.703 | 300 |
| rf3 | 33 | 136 | 486 | 2 | 1 | 449.281 | 600 |
| rf4 | 9 | 311 | 885 | 102 | 1179 | 495.515 | 300 |
| rf5 | 17 | 129 | 453 | 2 | 1 | 206.703 | 300 |

Table 4.2: Results of automated size reduction next most relations heuristic. All times are measured in seconds.

# Chapter 5

# Conclusion

In this master thesis, we discussed the implementation of a C-program for the generation of counter ortholattices for formulas which are not provable in orthologic. Chapter 2 was dedicated to the theoretical background. First we introduced what a lattice, respectively ortholattice is and gave an equational characterisation for them. But we also introduced a characterisation using partial ordered sets and partial ordered sets with orthocomplementation. After defining what a lattice is, we considered the logical aspects, i.e., how a logic is defined over lattices and ortholattices. This leads us to the term of orthologic, which is the logic defined over ortholattices.
The next section discussed proof systems for orthologic, i.e., Gentzen systems. We introduced the calculus GL for pure lattices and GOL for orthologic. Using a saturation-based forward search approach, we showed how GOL is sufficient to generate our searched counter lattice and how this is performed.
Chapter 3 discussed the practical part, i.e., the concrete implementation. The implementation results in a C-program named "CGOL", which I programmed as part of this master thesis. It is written in ANSI-C, thus it is fast and compatible with the three major OS-platforms. It is on the one side an efficient implementation of the calculus GOL as decision procedure whether

some given orthologic formula is valid or not. On the other side it realizes a method for the construction of counter ortholattices if the orthologic formula is non-provable. The chapter describes the algorithms used in the program and how the theoretical results are used in the praxis. One section discussed the worst case runtime of CGOL which is polynomially bounded.

The problem of the resulting counter ortholattices is that they grow quite large and even formulas which are rather moderate or even small in size, might have a counter ortholattice which is no more readable for a human user. That is the reason why we tried to reduce the size of the counter ortholattice in Chapter 4. The approach was the use of GOL with non-logical axioms. The idea was that non-logical axioms lead to more $\leq$-expressions that are derivable and thus increasing the chance that two elements get relational equivalent, i.e., that $x \leq y$ and $y \leq x$ holds. Such two elements are members of the same equivalence class and only one member is needed as representative for each equivalence class. This reduces the complexity of the counter ortholattice. Two main problems arise from this approach. The first is that the use of non-logical axioms requires the extension of GOL by the acut rule. That is why the implementation had to be extended. The second problem is the selection of the non-logical axioms. For this problem, two alternatives have been implemented: The user can manually input the non-logical axioms, or they can be searched automatically. The automated approach proves quite useful for large lattices which a human user cannot handle.

For the theoretical aspects, Egly has a work in progress [1] which gives details about the theory how a counter lattice can be generated and also about reducing its size. One drawback of CGOL is that it is only a command line program which only prints the relations of the counter lattice to a file, but does not draw them. This gap is currently getting closed by the master thesis [8], which is dedicated to implement a program for drawing ortholattices.

Using Gentzen systems is not the only possibility for generating counter lattices. The usage of non-Gentzen-like calculi for the generation of counter

lattices could be an interesting field of research. Another field where much can be done is the question of reducing the size of the counter lattice. Maybe there exist better ways than using non-logical axioms and also better heuristics could be developed for the automated search for non-logical axioms.
Although, as can be seen from the results section, CGOL performs very well, there is some room left for improvements. The memory consumption, especially for the representation of the $\leq$-relations could be optimized. At the moment, CGOL is optimized for speed, which sometimes leads to increased memory usage. For example the explicit storage of the $\geq$-relations could be avoided. This would reduce the memory consumption for storing the relations by 50 percent. Another possibility could be the usage of a totally different data structure for representing the $\leq$-relations. One data structure already tried for the relations was to use a simple quadratic matrix of all subformulas and their complements to store the relations. The problem with this data structure was the explicit storage of the unrelated elements, which required a lot of memory. Maybe a good solution for a sparse matrix could improve this problem. These questions could be tasks of future works.

# List of Figures

# List of Tables

# Bibliography

[1] U. Egly. Counter (Ortho-)Lattice Construction. unpublished, 2006.

[2] U. Egly and H. Tompits. On Different Proof-Search Strategies for Orthologic. *Studia Logica*, 73:131–152, 2003.

[3] M.R. Garey and D.S. Johnson. *Computers and Intractability.* Bell Laboratories, Murray Hill New Jersey, twenty-second edition, 2000.

[4] G. Kalmbach. *Orthomodular Lattices.* Academic Press, London, 1983.

[5] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Bell Telephone Laboratories Inc., second edition, 1988.

[6] A. Leitsch and C.G. Fermüller. Hyperresolution and Automated Model Building. *Journal of Logic and Computation*, 6(2):173–203, 1996.

[7] W. McCune. Automatic Proofs and Counterexamples for some Ortholattice Examples. *Information Processing Letters*, 65:285–291, 1998.

[8] G. Ziegler. Lattice Drawing in 3D. Master's thesis, Vienna University of Technology, 2006.