university of groningen

faculty of mathematics and natural sciences

Computation of the Size of the Fragment $[\wedge, \vee, \neg\neg]^n$ of Intuitionistic Logic

Bachelor thesis in Mathematics

June 2015

Student: T. van der Vlugt (s2231689)

Primary supervisor: prof. dr. G.R. Renardel de Lavalette

Secondary supervisor: prof. dr. J. Top

Abstract

This thesis will give a step-by-step solution to the calculation of the size of the fragments $F_n = [\wedge, \vee, \neg\neg]^n$ of intuitionistic propositional logic. These fragments are sets of formulae that contain at most n different propositional variables and only the connectives $\wedge$, $\vee$ and $\neg\neg$. The size of a fragment will be defined as the total number of equivalence classes of F_n. The diagram of a fragment is the partially ordered set of equivalence classes ordered by derivability. The calculation of the size of F_n will be done by finding the set of join-irreducible elements $\mathcal{I}(F_n)$ of the diagram of F_n, and generate all upward closed subsets of $\mathcal{I}(F_n)$. These will form a finite distributive lattice by Birkhoff's theorem. The set of upper sets is isomorphic to the diagram of F_n.

The algorithm successfully computed the sizes $\#F_1 = 4$, $\#F_2 = 21$ and $\#F_3 = 1891$ (with $\top$ and $\bot$ included in each fragment), but is not efficient enough to compute $\#F_4$, yet.

Keywords: Intuitionistic propositional logic, fragment, diagram, lattice, upper set

Contents

1 Introduction and Preliminaries **3**

1.1 Intuitionistic Propositional Logic 3

1.2 Partial Ordered Sets 5

1.3 Fragments 7

2 Exact Models & Irreducible formulae **9**

2.1 Models 9

2.2 Irreducibility 9

3 Exact Model of $[\wedge, \vee, \neg\neg]^n$ **12**

3.1 $[\wedge, \vee]^n$ 12

3.2 $[\wedge, \vee, \neg\neg]^n$ 14

4 Computing the Size of $[\wedge, \vee, \neg\neg]^n$ **18**

4.1 Data Types 18

4.2 Generating Upsets 20

4.3 Constructing Irreducible Formulae of $[\wedge, \vee]^n$ 21

4.4 Constructing Diagram of $[\wedge, \vee]^n$ 22

4.5 Constructing Irreducible Formulae of $[\wedge, \vee, \neg\neg]^n$ 23

4.6 Counting the Size of $[\wedge, \vee, \neg\neg]^n$ 24

5 Conclusion **25**

A List of Formulae in $\mathcal{I}([\wedge, \vee, \neg\neg]^4)$ **26**

B Program **30**

Bibliography **41**

Chapter 1

Introduction and Preliminaries

In the early years of the twentieth century Dutch logician L.E.J. Brouwer created the foundations of intuitionistic logic. It can be seen as the logical implementation of *mathematical constructivism*. Constructivism is based on the idea that mathematical objects can only be proven to exist if it is possible to construct them. In the same manner *intuitionism* considers mathematics as the product of the human mind and not as a representation of properties of objective reality. Intuitionistic logic therefore rejects the classical concepts of truth and falsehood and instead reads $\vdash \phi$ as *"ϕ is provable"* and $\vdash \neg\phi$ as *"ϕ is refutable"* (id est *"there exists a counterexample for ϕ"*).
In practice this entails that the negation operator loses its symmetry. If a formula ϕ is provable, then it is impossible to prove $\neg\phi$, however it is not enough to show that it is impossible to prove $\neg\phi$ to say that ϕ is provable. In other words, the classical law of *double negation elimination* $\neg\neg\phi \vdash \phi$ does not hold in intuitionistic logic, while the reverse $\phi \vdash \neg\neg\phi$ is still valid.
With these concepts in mind, intuitionistic logic can be seen as an extension of classical logic where the *law of the excluded middle* ($\vdash \phi \vee \neg\phi$) and the aforementioned *double negation elimination* are not valid.
In this thesis we will take a look at the $[\wedge, \vee, \neg\neg]^n$ *fragment* of intuitionistic logic. In short this is the set of formulae consisting of n propositional atoms $\{p, q, r, \dots\}$ and compositions thereof using only the operators $\wedge$, $\vee$ and $\neg\neg$. In particular we shall compute the number of equivalence classes in $[\wedge, \vee, \neg\neg]^n$.

1.1 Intuitionistic Propositional Logic

Intuitionistic propositional logic (IPL from now on) is a language consisting of the constants $\top$ (*tautology*) and $\bot$ (*contradiction*), infinitely many propositional variables $\{p_1, p_2, \dots\}$, and the logical connectives $\wedge$, $\vee$ and $\rightarrow$ (respectively *conjunction*, *disjunction* and *implication*). Propositional variables are also called *atomic formulae* or *atoms* for short. Some authors choose to classify $\bot$ and $\top$ as atoms as well, but in this text we will not regard them as such.

Definition 1.1. A **formula** is defined inductively:

- $\bot$ and $\top$ are formulae.
- ϕ is a formula if ϕ is an atom.
- If ϕ and ψ are formulae, then $\phi \wedge \psi$, $\phi \vee \psi$ and $\phi \to \psi$ are formulae.
- Nothing else is a formula.

Negation of a formula is defined by $\neg\phi \equiv \phi \to \bot$, and *bi-implication* is defined similarly to the classical case by $\phi \leftrightarrow \psi \equiv (\phi \to \psi) \wedge (\psi \to \phi)$.
Atoms will be written with lowercase letters from the Latin alphabet (p, q, r and s), while formulae will be represented by lowercase Greek letters (ϕ, ψ and χ). Sets of atoms will be written with uppercase Latin letters (P, Q and R) and sets of formulae will have uppercase Greek letters (Γ and Δ) In order to avoid excessive parentheses the binding of the operators $\vee$ and $\wedge$ is more powerful than the binding of $\to$ and $\leftrightarrow$, while $\neg$ binds stronger than any of the other operators, so with $\neg p \wedge q \to r \vee s$ we mean $[(\neg p) \wedge q] \to (r \vee s)$.
Furthermore there is the notion of entailment. If Γ is a set of formulae, then $\Gamma \vdash \phi$ (read as Γ *entails* ϕ) is interpreted as *"from the proofs of each formula* $\psi \in \Gamma$ *it is possible to construct a proof of the formula* ϕ*."* It is customary to omit Γ if it is the empty set (e.g. $\vdash \phi$ means *"*ϕ *is provable"*), also $\Gamma \cup \{\phi\} \vdash \psi$ is abbreviated as $\Gamma, \phi \vdash \psi$. Two formulae ϕ and ψ are called *equivalent* if $\phi \vdash \psi$ and $\psi \vdash \phi$. This is usually written as $\phi \equiv \psi$. The converse of $\vdash$ is $\dashv$, and $\phi \vdash \psi$ if and only if $\psi \dashv \phi$.
The rules of IPL can be described with a natural deduction system. This system consists of three introduction rules:

- A proof of ϕ and a proof of ψ imply a proof of $\phi \wedge \psi$, ($\wedge$-*introduction*)
- A proof of ϕ implies proofs of $\phi \vee \psi$ and $\psi \vee \phi$, ($\vee$-*introduction*)
- $\phi \vdash \psi$ implies a proof of $\phi \to \psi$, ($\to$-*introduction*)

three elimination rules:

- A proof of $\phi \wedge \psi$ implies proofs of ϕ and ψ, ($\wedge$-*elimination*)
- $\phi \vdash \chi$ and $\psi \vdash \chi$ implies $\phi \vee \psi \vdash \chi$, ($\vee$-*elimination*)
- A proof of ϕ and a proof of $\phi \to \psi$ imply a proof of ψ, ($\to$-*elimination*)

and the *principle of explosion*:

- $\bot \vdash \phi$

Note that this system can be converted to classical propositional logic if one adds the rule

- A proof of $\neg\neg\phi$ implies a proof of ϕ. ($\neg\neg$-*elimination*)

Although $\neg\neg$-elimination is not possible in IPL, the reverse $\phi \vdash \neg\neg\phi$ is still a valid inference. Some other classical laws hold as well, such as the associativity and commutativity of $\wedge$ and $\vee$, the distributivity of $\wedge$ and $\vee$ over each other, and $\neg\neg\neg\phi \equiv \neg\phi$. Of the de Morgan laws only $\neg(\phi \wedge \psi) \vdash \neg\phi \vee \neg\psi$ is invalid, the three other laws hold.
The principle of double-negation translation can be very useful, as it translates a classical formula to a intuitionistic equivalent. The easiest variant, which holds in propositional

logic, but not in predicate logic, was discovered by Glivenko, and therefore bears his name. Let $\vdash_C$ mean classical entailment and $\vdash_I$ intuitionistic entailment.

Theorem 1.2 (Glivenko's Theorem)**.** If ϕ is a formula, then $\vdash_C \phi$ if and only if $\vdash_I \neg\neg\phi$.

Two very useful laws which are confirmed by Glivenko's theorem are $\neg\neg(\phi\wedge\psi) \equiv \neg\neg\phi\wedge\neg\neg\psi$ and $\neg\neg(\phi \to \psi) \equiv \neg\neg\phi \to \neg\neg\psi$. We will call this *double negation shift*. These laws can be used to extend Glivenko's theorem:

Corollary 1.3. Let ϕ be a formula, $\Gamma = \{\psi_1, \ldots, \psi_k\}$ a set of formulae, and $\Gamma^* = \{\neg\neg\psi_1, \ldots, \neg\neg\psi_k\}$ the set with the double negation of the formulae of Γ. Then $\Gamma \vdash_C \phi$ if and only if $\Gamma^* \vdash_I \neg\neg\phi$.

Proof. Suppose $\Gamma \vdash_C \phi$, then $\vdash_C \bigwedge\Gamma \to \phi$ and therefore by Glivenko's theorem $\vdash_I \neg\neg(\bigwedge\Gamma \to \phi)$. Use double negation shift to rewrite this as $\vdash_I (\bigwedge\Gamma^*) \to \neg\neg\phi$ (note that $\neg\neg\bigwedge\Gamma$ is equivalent to $\bigwedge\Gamma^*$, since Γ^* is the double negation of Γ). Finally we conclude that $\Gamma^* \vdash_I \neg\neg\phi$. □

1.2 Partial Ordered Sets

Before we can define what a fragment is, we need a basic understanding and terminology of partial ordered sets. The following section will briefly summarise some basic concept and properties related to partial ordered sets. A more thorough explanation of this subject can be found in [1] or [2].

Definition 1.4. A **partial order** on a set A is a binary relation $\leq$ in which the following conditions hold for all $a, b, c \in A$:

- $a \leq a$, (*reflexive*)
- $a \leq b$ and $b \leq a$ imply $a = b$, (*antisymmetric*)
- $a \leq b$ and $b \leq c$ imply $a \leq c$. (*transitive*)

The pair $\langle A, \leq\rangle$ is called a *partially ordered set*, often abbreviated as **poset**.

The statement ($a \leq b$ and $a \neq b$) is to be written as $a < b$. The inverse $\geq$ of $\leq$ will be defined by $a \geq b \Leftrightarrow b \leq a$. Similarly $>$ is the inverse of $<$.

Definition 1.5. The **power set** of a set A is the set of all subsets of A, denoted $\mathcal{P}(A) = \{X \mid X \subseteq A\}$.

Definition 1.6. Let $\langle S, \leq\rangle$ be a poset. An **upset** (upward closed subset) is a subset $X \subseteq S$ such that $\forall x \in X, \forall y \in S$ we have $x \leq y \Rightarrow y \in X$. A **downset** (downward closed subset) is a subset $Y \subseteq S$ such that $\forall y \in Y, \forall x \in S$ we have $y \geq x \Rightarrow x \in Y$.

The set of all upsets will be $\mathcal{P}^{\uparrow}(A) = \{X \in \mathcal{P}(A) \mid X \text{ is an upset}\}$, whereas the set of all downsets will be $\mathcal{P}^{\downarrow}(A) = \{X \in \mathcal{P}(A) \mid X \text{ is a downset}\}$. The function $\uparrow$ maps a subset X to the upset $\uparrow(X) = \{x \in A \mid \exists y \in X : x \geq y\}$. Similarly $\downarrow(X) = \{x \in A \mid \exists y \in X : x \leq y\}$ gives us a downset.

Definition 1.7. Let $\langle S, \leq \rangle$ be a poset and let $x, y \in S$. The element $w \in S$ / $v \in S$ is defined as the **join** / **meet** of x and y if:

- $x \leq w$ and $y \leq w$ / $v \leq x$ and $v \leq y$
- $\forall z \in S$ we have $x \leq z$ and $y \leq z \Rightarrow w \leq z$ / $z \leq x$ and $z \leq y \Rightarrow z \leq v$.

Such an element w / v will be denoted as $w = x \varovee y$ / $v = x \varowedge y$. Note that the definition implies that the join / meet of two elements is either unique or non-existent.

Well known examples of join and meet operators are the $\cup$ and $\cap$ operators in set theory, the lcm and gcd in number theory, and, as we will see in lemma 1.18, $\vee$ and $\wedge$ in logic.

Definition 1.8. Let $\langle S, \leq \rangle$ be a poset. The **top** of S, if existing, is the element $w \in S$ such that $\forall x \in S$ we have $w \varovee x = w$. The **bottom** of S, if existing, is the element $v \in S$ such that $\forall x \in S$ we have $v \varowedge x = v$.

Definition 1.9. A poset S is called a **lattice** if for all elements $x, y \in S$ there exists a join $x \varovee y$ and meet $x \varowedge y$.

In a lattice, the join and meet operators are associative, commutative and idempotent binary relations. We can therefore define the join of a finite subset $X = \{x_1, x_2, \ldots, x_n\}$ of lattice S as $\bigcirc\!\!\!\!\vee X = x_1 \varovee x_2 \varovee \ldots \varovee x_n$. The meet of a subset is defined similarly. Note that the top of S is equal to $\bigcirc\!\!\!\!\vee S$, and the bottom of S is equal to $\bigcirc\!\!\!\!\wedge S$. We will define $\bigcirc\!\!\!\!\vee \varnothing$ as the bottom element and $\bigcirc\!\!\!\!\wedge \varnothing$ as the top element.

Definition 1.10. A lattice is **distributive** if one of the distributive laws hold:

- $x \varovee (y \varowedge z) = (x \varovee y) \varowedge (y \varovee z)$,
- $x \varowedge (y \varovee z) = (x \varowedge y) \varovee (y \varowedge z)$.

Note that each distributive law implies the other, so both laws are automatically valid in a distributive lattice.

Definition 1.11. Let S and P be lattices. An **isomorphism** is a bijective mapping $f : S \mapsto P$ such that for every $x, y \in S$ it holds that $f(x \varovee y) = f(x) \varovee f(y)$ and $f(x \varowedge y) = f(x) \varowedge f(y)$. If such a mapping exists, S and P are called **isomorphic**.

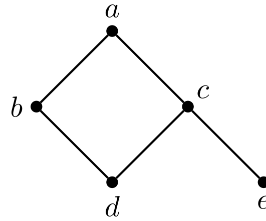
Definition 1.12. Let $\langle S, \leq \rangle$ and $\langle P, \leq' \rangle$ be posets and $f : S \mapsto P$ a function. f is called **order-preserving** if $a \leq b$ implies that $f(a) \leq' f(b)$.

Theorem 1.13. Let $\langle S, \leq \rangle$ and $\langle P, \leq' \rangle$ be lattices and $f : S \mapsto P$ a bijective function with

inverse f^{-1}, then f is an *isomorphism* if and only if both f and f^{-1} are *order preserving.*

| *Proof.* A proof can be found in [1]. □ |

Finally we will introduce a way to draw partial ordered sets: the **Hasse diagram**. In a Hasse diagram each element of the poset is drawn as a vertex of a graph. The order relation is drawn as an edge between two vertices. An edge will only be drawn if an element is directly larger than the other (e.g. if $x < y < z$, then there will never be an edge between x and z). The vertical placement of the elements determines the direction of the relation. If for example $a > b$, the vertex a will be drawn higher than the vertex b. The Hasse diagram of poset $\langle\{a, b, c, d, e\}, (a>b,\ a>c,\ b>d,\ c>d,\ c>e)\rangle$ is drawn below:



1.3 Fragments

Definition 1.14. A **fragment** of intuitionistic logic $F = [c_1, \ldots, c_k, p_1, \ldots, p_n]$ is a set of formulae (and a sublanguage of IPL) generated by restricting the set of atoms to the (possibly infinite) set $\{p_1, \ldots, p_n\}$ and the set of connectives to $\{c_1, \ldots, c_k\}$.

If $P = \{p_1, \ldots, p_n\}$ is a set of propositional variables, then we sometimes use the notation $[c_1, \ldots, c_k, P]$ where we mean $[c_1, \ldots, c_k, p_1, \ldots, p_n]$. When it is of lesser importance which atoms are used, the fragment will be written as $F = [c_1, \ldots, c_k]^n$, where n is the number of atoms. The set $Atom(F)$ denotes the propositional variables F is restricted to, and the set $Con(F)$ denotes the connectives F is restricted to.
Although $\top$ and $\bot$ need not necessarily be embedded in every fragment, they will be included in all the fragments containing $\wedge$ and $\vee$ for the remainder of this text. The reason for this will become apparent in lemma 1.19 and 1.20, using the facts that the join and meet of the empty set are defined as the bottom and top elements.

Definition 1.15. The **size** of fragment F will be defined if F has finitely many equivalence classes. In that case F will be called a *finite* fragment, and we will denote the number of equivalence classes by $\#F$. If F has infinitely many equivalence classes we will call F an *infinite* fragment.

Definition 1.16. The **diagram** of fragment F (denoted $Diag(F)$) is the partially ordered set $\langle F, \vdash \rangle$, which is created by ordering the equivalence classes of F by derivability.

Definition 1.17. Let F be a finite fragment, let $n = \#F$, and let $\mathcal{F} = \{\phi_1, \ldots, \phi_n\}$ be a set of formulae in F such that $\phi_i \equiv \phi_j$ implies $i = j$. $\bigwedge F$ will be defined as the conjunction of all elements of $\mathcal{F}$. Similarly $\bigvee F$ will be defined as the disjunction of all elements of $\mathcal{F}$.

Lemma 1.18. If we consider $Diag(F)$ where $\vee \in Con(F)$, then $\phi \varovee \psi$ is equivalent to $\phi \vee \psi$. If $\wedge \in Con(F)$, then $\phi \varowedge \psi$ is equivalent to $\phi \wedge \psi$.

Proof. Let F be a fragment containing $\vee$ and let $\phi, \psi \in F$. $\vee$-introduction gives us $\phi \vdash \phi \vee \psi$ and $\psi \vdash \phi \vee \psi$. Suppose $\phi \vdash \chi$ and $\psi \vdash \chi$ for some $\chi \in F$, then $\vee$-elimination says that $\phi \vee \psi \vdash \chi$. We can see that $\vee$ satisfies all conditions to be the join operator.

The equivalence between $\varowedge$ and $\wedge$ can be proven similarly with $\wedge$-elimination and $\wedge$-introduction. □

Lemma 1.19. Let F be a finite fragment with $\vee, \wedge \in Con(F)$, then $Diag(F)$ is a finite distributive lattice with top $\bigvee F$ and bottom $\bigwedge F$.

Proof. $\bigwedge F \vdash \phi$ for any $\phi \in F$ by $\wedge$-elimination, so $\bigwedge F \vdash \bigwedge F \wedge \phi$. Furthermore $\bigwedge F \wedge \phi \vdash \bigwedge F$ by $\wedge$-elimination, so we have shown that $\bigwedge F \equiv \bigwedge F \wedge \phi$ for any ϕ (thus $\bigwedge F$ is the bottom element).

On the other hand we have $\bigvee F \vdash \bigvee F$ and $\phi \vdash \bigvee F$, therefore $\bigvee F \vee \phi \vdash \bigvee F$ by $\vee$-elimination, and conversely $\bigvee F \vdash \bigvee F \vee \phi$ by $\vee$-introduction. This means that $\bigvee F \equiv \bigvee F \vee \phi$ for any ϕ (thus $\bigvee F$ is the top element).

We can conclude that $Diag(F)$ with join and meet operators $\vee$ and $\wedge$ is a lattice, since $\wedge$ and $\vee$ are defined for each pair $\phi, \psi \in F$.

The classical distributive laws for $\wedge$ and $\vee$ are valid in IPL, so the lattice $Diag(F)$ is distributive. □

Theorem 1.20. If $\top, \bot \in F$, then $\top$ is the top element and $\bot$ the bottom element.

Proof. In the proof of lemma 1.19 we saw that $\bigvee F$ is the top element. Since $\top \vdash \bigvee F$ (by $\vee$-introduction) and $\bigvee F \vdash \top$ (anything entails $\top$), we can assure that $\top \equiv \bigvee F$. Similarly we saw that $\bigwedge F$ is the bottom element, and both $\bigwedge F \vdash \bot$ (by $\wedge$-elimination) and $\bot \vdash \bigwedge F$ (by the principle of explosion) are true, so $\bigwedge F \equiv \bot$. □

Chapter 2

Exact Models & Irreducible formulae

Since the diagram of a fragment consists of all equivalence classes of the fragment, the computation of $Diag(F)$ automatically yields the size of F. While the computation of $Diag(F)$ is a hard task in general, it can be simplified by constructing a *model* of $Diag(F)$.

2.1 Models

Definition 2.1. A **model** for fragment F is a triple $\langle W, \leq, \omega \rangle$, where $\langle W, \leq \rangle$ is a poset, and ω is a function $F \mapsto \mathcal{P}^{\uparrow}(W)$ which maps formulae $\phi, \psi \in F$ to upsets of W in an order-preserving fashion, so $\phi \vdash \psi$ implies $\omega(\phi) \subseteq \omega(\psi)$. The set W is called a set of *worlds*. We will define a **forcing relation** for a world $w \in W$ and a formula $\phi \in F$ as $w \Vdash \phi$ (*w forces ϕ*) if and only if $w \in \omega(\phi)$, and the following rules for the forcing relation:

- If $p \in Atom(F)$ and $w \Vdash p$, then $w \leq u$ implies $u \Vdash p$,
- $w \Vdash \phi \wedge \psi$ if and only if $w \Vdash \phi$ and $w \Vdash \psi$,
- $w \Vdash \phi \vee \psi$ if and only if $w \Vdash \phi$ or $w \Vdash \psi$,
- $w \Vdash \phi \to \psi$ if and only if for each $u \geq w$ we have $u \Vdash \phi$ implies $u \Vdash \psi$,
- $w \nVdash \bot$.

Definition 2.2. A model $M = \langle W, \leq, \omega \rangle$ is called **complete for** F if $\forall \phi, \psi \in F$ it holds that $\phi \vdash \psi \ \Leftrightarrow \ \omega(\phi) \subseteq \omega(\psi)$.

Definition 2.3. A model $M = \langle W, \leq, \omega \rangle$ is called **exact for** F if it is complete and ω is a surjection.

This means that ω forms a bijection between a fragment F and the downsets of a (most likely considerably smaller) set of worlds W.

2.2 Irreducibility

Definition 2.4. Let $\langle A, \leq \rangle$ be a poset. An element $a \in A$ is called (join-)**irreducible** if it is not the bottom element of A and if $a \leq x \varovee y$ then $a \leq x$ or $a \leq y$. The set of irreducible elements of a poset A is written as $\mathcal{I}(A)$.

The set of irreducible formulae in $Diag(F)$ is denoted as $\mathcal{I}(F)$. Note that $\mathcal{I}(F)$ is a poset as well, ordered by entailment.

Theorem 2.5 (Birkhoff's theorem)**.** If $\langle A, \leq \rangle$ is a finite distributive lattice, then it is isomorphic to the lattice $\langle \mathcal{P}^{\downarrow}(\mathcal{I}(A)), \subseteq \rangle$.

Proof. Our proof will be constructive: define the mapping:

$$\omega(x) = \{a \in \mathcal{I}(A) \mid a \leq x\}$$

Since $\leq$ is transitive, ω produces an downset of $\mathcal{I}(A)$ for each element $x \in A$. Now, to prove ω is an isomorphism, we use $\omega(x \curlyvee y) = \{a \in \mathcal{I}(A) \mid a \leq x \curlyvee y\}$. Since each a is join-irreducible, $a \leq x$ or $a \leq y$, so we can derive:

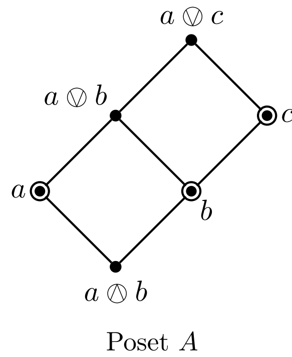
$$\begin{aligned} \omega(x \curlyvee y) &= \{a \in \mathcal{I}(A) \mid a \leq x \curlyvee y\} \\ &= \{a \in \mathcal{I}(A) \mid a \leq x \vee a \leq y\} \\ &= \{a \in \mathcal{I}(A) \mid a \leq x\} \cup \{a \in \mathcal{I}(A) \mid a \leq y\} \end{aligned}$$

Since $\cup$ is the join operator of $\subseteq$, we conclude $\omega(x \curlyvee y) = \omega(x) \curlyvee \omega(y)$.
Now for $\omega(x \curlywedge y)$ we have the following derivation:

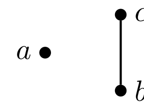
$$\begin{aligned} \omega(x \curlywedge y) &= \{a \in \mathcal{I}(A) \mid a \leq x \curlywedge y\} \\ &= \{a \in \mathcal{I}(A) \mid a \leq x \wedge a \leq y\} \\ &= \{a \in \mathcal{I}(A) \mid a \leq x\} \cap \{a \in \mathcal{I}(A) \mid a \leq y\} \end{aligned}$$

Once again $\cap$ is the meet operator of $\subseteq$, so we conclude $\omega(x \curlywedge y) = \omega(x) \curlywedge \omega(y)$.
The mapping $\omega(x) = X$ is a bijection, since it has the inverse $\omega^{-1}(X) = \bigcurlyvee X$, the join of all elements in X. This fulfils the conditions given in definition 1.11, so A and $\mathcal{P}^{\downarrow}(\mathcal{I}(A))$ are isomorphic. $\square$

Example 2.6. Below we see a *Hasse diagram* of a poset A with six elements. The irreducible elements are encircled and form the exact model of A. Each downset of the exact model corresponds with an element of A; the empty set corresponds with the bottom element, $a \curlywedge b$, while every other downset X corresponds with the element $\bigcurlyvee X$, the join of the irreducible elements that make up X.



Poset A



Exact model of A

Theorem 2.7. Let F be a fragment and let $\wedge, \vee \in Con(F)$. The set of irreducible elements $\mathcal{I}(F)$ of a fragment F forms an exact model for F.

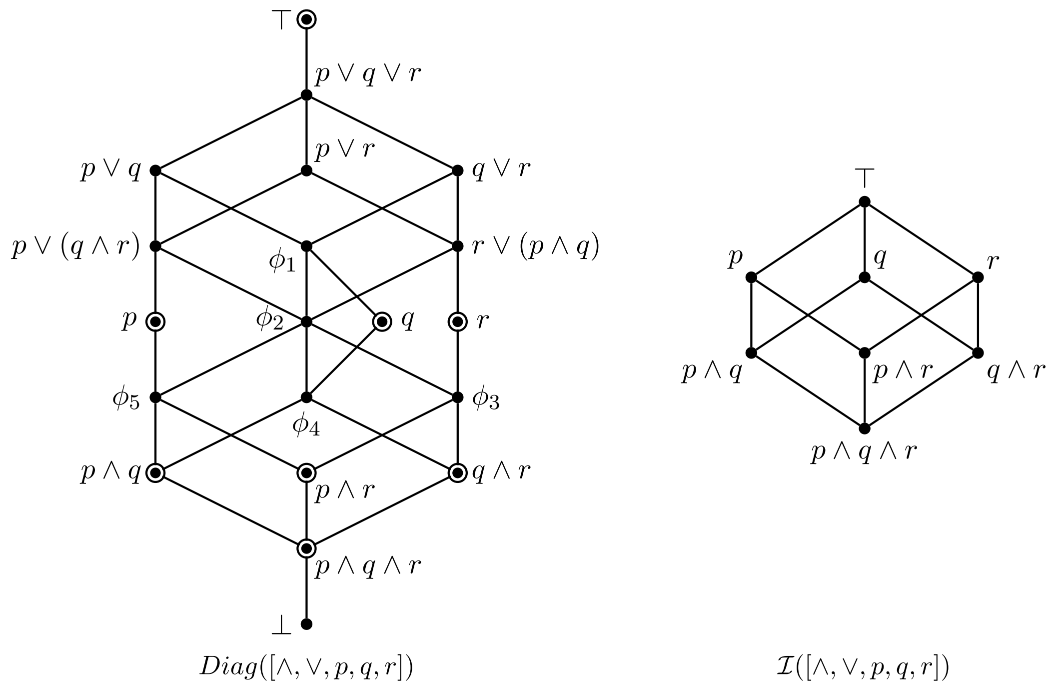
Proof. Lemma 1.19 showed that $Diag(F)$ is a distributive lattice if $\wedge, \vee \in Con(F)$, so we can use the ω function given in the proof of Birkhoff's theorem to formulate the model $\langle \mathcal{I}(F), \vdash, \omega \rangle$. Birkhoff's theorem states that this ω is an isomorphism, so ω is a surjection. Furthermore the model is complete by theorem 1.13. We conclude that $\langle \mathcal{I}(F), \vdash, \omega \rangle$ is an exact model for $Diag(F)$. $\square$

Exact Model of $[\wedge, \vee, \neg\neg]^n$

Theorem 2.7 states that it is possible to construct the exact model of $[\wedge, \vee, \neg\neg]^n$ from its irreducible elements. To find each irreducible formula, we need to be able to generate all formulae of $[\wedge, \vee]^n$, so we start by finding $\mathcal{I}([\wedge, \vee]^n)$.

3.1 $[\wedge, \vee]^n$

As an example, the diagrams of fragment $[\wedge, \vee]^3$ and its irreducible elements are drawn below. The formulae that were too long to be drawn in the fragment are $\phi_1 \equiv q \vee (p \wedge r)$, $\phi_2 \equiv (p\wedge r)\vee(p\wedge q)\vee(q\wedge r)$, $\phi_3 \equiv (p\wedge r)\vee(q\wedge r)$, $\phi_4 \equiv (p\wedge q)\vee(q\wedge r)$ and $\phi_5 \equiv (p\wedge q)\vee(p\wedge r)$. The encircled elements are irreducible.



$Diag([\wedge, \vee, p, q, r])$ $\qquad$ $\mathcal{I}([\wedge, \vee, p, q, r])$

Lemma 3.1. Let $F = [\wedge, \vee]^n$. Every formula $\phi \in F$ can be written as a disjunction of formulae $\bigwedge Q$ where each Q is a subset of $Atom(F)$.

Proof. Any nested disjunction $\phi \wedge (\psi \vee \chi)$ can be unnested by application of the distributive law $\phi \wedge (\psi \vee \chi) \equiv (\phi \wedge \psi) \vee (\phi \wedge \chi)$. The lemma follows by induction. Nota bene: the top and bottom can be written as $\top \equiv \bigwedge \varnothing$ and $\bot \equiv \bigvee \varnothing$. □

Theorem 3.2. Let $F = [\wedge, \vee]^n$, then ϕ is an irreducible element of F if and only if $\phi \equiv \bigwedge Q$, where $Q \subseteq Atom(F)$ is some set of atoms.

Proof. Let ϕ be an irreducible formula. Lemma 3.1 says that it has to be of the form $\phi \equiv \bigvee[\bigwedge Q_I]$, where each Q_i $(i \in I)$ is a set of atoms. Since $\bigvee[\bigwedge Q_I] \vdash \bigvee[\bigwedge Q_I]$ is true, if ϕ is irreducible, we need $\bigvee[\bigwedge Q_I] \vdash \bigwedge Q_i$ for some $i \in I$. But then $\vee$-introduction gives us $\bigwedge Q_i \vdash \bigvee[\bigwedge Q_I]$ and therefore $\bigvee[\bigwedge Q_i] \equiv \bigwedge Q_i$.
Bottom element $\bot$ cannot be written as the conjunction of atoms, so the only candidates which can be irreducible are the formulae $\bigwedge Q$ where $Q \subseteq Atom(F)$.
Let P be a non-emty set of atoms which is disjoint from Q (so $P \cap Q = \varnothing$), then $\bigwedge Q \nvdash \bigwedge P$. Now let $P_1, \ldots, P_k$ be nonempty sets of atoms which are all disjoint from Q, then we have $\bigwedge Q \nvdash \bigvee[\bigwedge P_I]$, where $I = \{1, \ldots, k\}$. Suppose that $\bigwedge Q \vdash \bigvee[\bigwedge Q_I]$. Then we have the following:

$$\begin{aligned} \bigwedge Q &\vdash \bigwedge Q \wedge \bigvee[\bigwedge Q_I] \\ &\equiv \bigwedge Q \wedge \bigvee[\bigwedge (Q_I \setminus Q)] \\ &\vdash \bigvee[\bigwedge (Q_I \setminus Q)] \end{aligned}$$

Since each $Q_i \setminus Q$ is disjoint from Q, the only way this entailment holds is if one of the $Q_i \setminus Q$ is empty. In that case we have $\bigwedge(Q_i \setminus Q) \equiv \bigwedge \varnothing \equiv \top$, and therefore $\bigvee[\bigwedge(Q_I \setminus Q)] \equiv \top$. However, if $Q_i \setminus Q = \varnothing$, then $Q_i \subseteq Q$, so then $\bigwedge Q \vdash \bigwedge Q_i$. We can conclude that $\bigwedge Q \vdash \bigvee[\bigwedge Q_I]$ implies that $\bigwedge Q \vdash \bigwedge Q_i$ for some $i \in I$, and therefore $\bigwedge Q$ is irreducible. □

If we want to construct the diagram of $\mathcal{I}([\wedge, \vee])$, we also need to be able to say when an irreducible formula ϕ directly entails another irreducible formula ψ.

Definition 3.3. Let F be a set of formulae and $\phi, \psi \in F$, then ϕ **directly entails** ψ in F if $\phi \not\equiv \psi$ and for every $\chi \in F$ such that $\phi \vdash \chi$ and $\chi \vdash \psi$ that either $\phi \equiv \chi$ or $\psi \equiv \chi$. This will be denoted as $\phi \vdash_1 \psi$ in F.

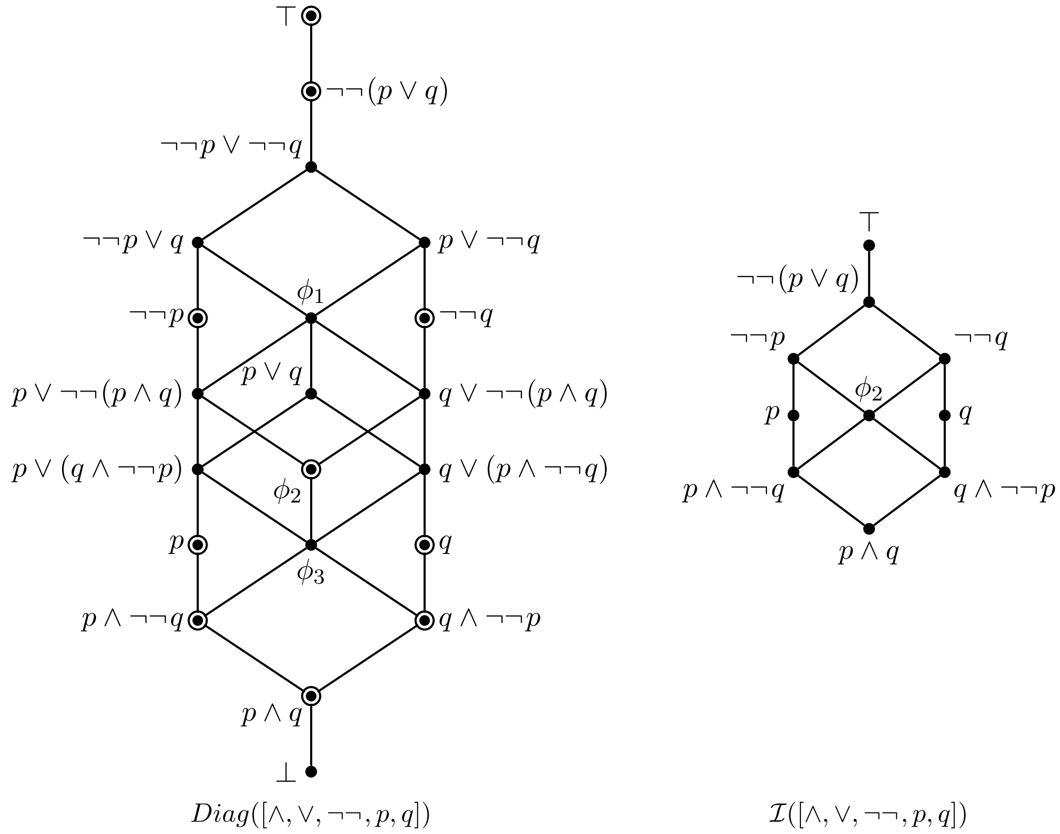
Lemma 3.4. Let $\phi \equiv \bigwedge Q_1$ and $\psi \equiv \bigwedge Q_2$ $(Q_1, Q_2 \subseteq Atom(F))$ be irreducible formulae of fragment $F = [\wedge, \vee]^n$, then $\phi \vdash_1 \psi$ in $\mathcal{I}(F)$ if and only if $Q_1 = Q_2 \cup \{p\}$ for some atom $p \notin Q_2$.

Proof. Naturally $\bigwedge Q_1 \vdash \bigwedge Q_2$ if and only if $Q_2 \subseteq Q_1$. Suppose $Q_2 \cup \{p\} \equiv Q_1$ for some $p \notin Q_2$ and $\bigwedge Q_1 \vdash \bigwedge P$ and $\bigwedge P \vdash \bigwedge Q_2$. P has to be a subset of Q_1, but Q_2 has to be a subset of P. If $p \notin P$, we have that $P \subseteq Q_1 \setminus \{p\} = Q_2$, so $P = Q_2$. On

the other hand if $p \in P$, then $Q_2 \cup \{p\} = Q_1 \subseteq P$, so $P = Q_1$. We conclude that $\bigwedge Q_1 \vdash_1 \bigwedge Q_2$.
Of course $\bigwedge Q_1 \nvdash_1 \bigwedge Q_2$ if $Q_1 = Q_2$ or if $Q_2 \not\subset Q_1$, so let us focus on the remaining case where $Q_2 \subset Q_1$ and $Q_1 \backslash Q_2$ has more than one element. Let's say $p, q \in Q_1 \backslash Q_2$, where $p \neq q$. Then of course $\bigwedge Q_1 \vdash \bigwedge (Q_1 \setminus \{p\}) \vdash \bigwedge Q_2$. Since $\bigwedge Q_1 \not\equiv \bigwedge (Q_1 \setminus \{p\}) \not\equiv \bigwedge Q_2$ (because p is only in Q_1, while q is in each set except for Q_2) we have $\bigwedge Q_1 \nvdash_1 \bigwedge Q_2$ $\square$

3.2 $[\wedge, \vee, \neg\neg]^n$

We are now able to construct the $[\wedge, \vee]^n$ fragments, so we can move on to the $[\wedge, \vee, \neg\neg]^n$ fragments. We have again drawn an example of the diagram and irreducible elements of $[\wedge, \vee, \neg\neg]^2$ below. In the diagrams the formulae $\phi_1 \equiv p \vee q \vee \neg\neg(p \wedge q)$, $\phi_2 \equiv \neg\neg(p \wedge q)$ and $\phi_3 \equiv (p \wedge \neg\neg q) \vee (q \wedge \neg\neg p)$.



$Diag([\wedge, \vee, \neg\neg, p, q])$ $\mathcal{I}([\wedge, \vee, \neg\neg, p, q])$

Lemma 3.5. All formulae $\phi \in F = [\wedge, \vee, \neg\neg]^n$ are equivalent to a disjunction of formulae $(\bigwedge Q) \wedge \neg\neg\psi$, where each $Q \subseteq Atom(F)$ and each $\psi \in [\wedge, \vee, Atom(F) \setminus Q]$.

Proof. We will show with Glivenko's theorem (or actually corollary 1.3) that any formula $\neg\neg\psi^*$ in which $\psi^* \in [\wedge, \vee, \neg\neg]^n$ can be reduced to a formula $\neg\neg\psi$, where $\psi \in [\wedge, \vee]^n$. That is, double negations inside double negations are unnecessary. Suppose $\psi^* \in [\wedge, \vee, \neg\neg]^n$ contains double negations, then classically ψ^* is equivalent to a formula $\psi \in [\wedge, \vee]^n$, because double negation elimination can remove any double negation. Since $\psi \equiv_C \psi^*$, Glivenko's theorem tells us that $\neg\neg\psi \equiv_I \neg\neg\psi^*$.
Now suppose we have a formula that consists of conjunctions and disjunctions of double negated formulae of $[\wedge, \vee]^n$, then we can arrange that formula in the form of $\bigvee(\bigwedge\Gamma_I^*)$ by the distributive rules over $\wedge$ and $\vee$, where Γ_i^* is a set of double negated formulae for each $i \in I$. We can convert each Γ_i^* to Γ_i, where each Γ_i is equal to Γ_i^* with all negations removed. We know because of double negation shift, $\bigwedge\neg\neg\phi \equiv \neg\neg\bigwedge\phi$, that therefore $\bigvee(\bigwedge\Gamma_I^*) \equiv \bigvee\neg\neg(\bigwedge\Gamma_I)$. Note that $\bigwedge\Gamma_I$ is itself a formula in $[\wedge, \vee]^n$, therefore we know that there are some $\psi_i \in [\wedge, \vee]^n$ for each $i \in I$ such that $\bigvee\neg\neg(\bigwedge\Gamma_I) \equiv \bigvee\neg\neg\psi_I$.
Finally we look at a formula that consists of the conjunctions and disjunctions of both double negated formulae from $[\wedge, \vee]^n$ and atomic formulae (this is in fact any formula in $[\wedge, \vee, \neg\neg]^n$). By the same arguments as before this formula is equivalent to $\bigvee(\bigwedge\Delta_I)$, where each Δ_i $(i \in I)$ is a set with double negated formulae and atoms. Define a set Q_i and a set Γ_i^* for each Δ_i, such that Q_i is the atomic part of Δ_i and Γ_i^* the double negated part, then we will see that $\bigvee(\bigwedge\Delta_I) \equiv \bigvee(\bigwedge Q_I \wedge \bigwedge\Gamma_I^*) \equiv \bigvee(\bigwedge Q_I \wedge \neg\neg\psi_I)$ with each $\psi_i \in [\wedge, \vee]^n$. We have now shown that any formula of $[\wedge, \vee, \neg\neg]^n$ can be written as the disjunction of formulae $(\bigwedge Q) \wedge \neg\neg\psi$. □

Theorem 3.6. Let $F = [\wedge, \vee, \neg\neg]^n$. A formula $\phi \in F$ is irreducible or the bottom element if and only if there exist a subset $Q \subseteq \mathit{Atom}(F)$ and a formula $\psi \in [\wedge, \vee, \mathit{Atom}(F) \setminus Q]$, such that $\phi \equiv (\bigwedge Q) \wedge \neg\neg\psi$.

Proof. Let an element $\phi \in \mathcal{I}([\wedge, \vee, \neg\neg]^n)$ be irreducible, then lemma 3.5 says it can be written as a disjunction of elements $(\bigwedge Q_i) \wedge \neg\neg\psi_i$ for $i \in I$. Since we assumed ϕ to be irreducible, there has to be an $i \in I$ such that $\phi \vdash (\bigwedge Q_i) \wedge \neg\neg\psi_i$, and therefore we have $\phi \equiv (\bigwedge Q) \wedge \neg\neg\psi$.
If we assume that $(\bigwedge Q) \wedge \neg\neg\psi$ is not irreducible, it must be the disjunction of some other irreducible formulae. So there must be a set of formulae $\phi_i \equiv Q_i \wedge \neg\neg\psi_i$ $(i \in I)$ not equivalent to $\phi \equiv (\bigwedge Q) \wedge \neg\neg\psi$ such that for all $i \in I$ we have $\phi_i \vdash \phi$ and $\phi \equiv \bigvee[\phi_I]$. This means that for all i we need $Q_i \supseteq Q$ and $\psi_i \wedge \bigwedge Q_i \setminus Q \vdash \psi$.
We can see that $\bigwedge \mathit{Atom}(F)$ has to be irreducible: the only possible superset Q_i of $\mathit{Atom}(F)$ is $\mathit{Atom}(F)$ itself, which gives only two options for $\psi_i \in [\wedge, \vee]^0$, namely $\top$ and $\bot$. Note that in the case of $\top$ we have $\mathit{Atom}(F) \wedge \neg\neg\top \equiv \mathit{Atom}(F)$ and in the case of $\bot$ we have $\mathit{Atom}(F) \wedge \neg\neg\bot \equiv \bot$. The first case is obviously equivalent

to $Atom(F)$, while the second is the bottom element, therefore $Atom(F)$ must be irreducible.
If we take a look at formula $\phi \equiv (\bigwedge Q) \wedge \neg\neg\bigwedge(Atom(F) \setminus Q)$, with $Q \neq Atom(F)$, then $Q_i = Q$ implies that $\psi_i \equiv \bot$ (this is because $\bigwedge Atom(F \setminus Q)$ is implied by only $\bigwedge Atom(F \setminus Q)$ and $\bot$, and only in the latter case we have $\phi_i \not\equiv \phi$), which makes $\phi_i \equiv \bot$, therefore for every $i \in I$ we have $Q_i \supset Q$. This means that $\phi \nvdash \bigvee \phi_I$, since for all $i \in I$ we have $\phi_i \vdash p$ for some $p \notin Q$. Therefore we can conclude that $(\bigwedge Q) \wedge \neg\neg\bigwedge(Atom(F) \setminus Q)$ is irreducible.
Lastly we have formulae of the form $\phi \equiv (\bigwedge Q) \wedge \neg\neg\psi$ where $\psi \not\equiv \bigwedge Atom(F) \setminus Q$. In this case for any ϕ_i with $Q_i \supset Q$ we can find the irreducible formula $\phi_i^* \equiv (\bigwedge Q) \wedge \neg\neg(\psi_i \wedge \bigwedge[Q_i \setminus Q])$, for which we have $\bigwedge Q_i \wedge \neg\neg\psi_i \vdash \phi_i^* \vdash (\bigwedge Q) \wedge \neg\neg\psi$, so the set of irreducible formulae for which $\phi \equiv \bigvee \phi_I$ with the least elements has for each $i \in I$ that $Q_i = Q$. Because the atoms in ψ and each ψ_i do not overlap with the atoms in Q, we can see that $\neg\neg\psi \equiv \bigvee \neg\neg\psi_I$ has to be true. Since double negation shift for disjunction is not generally valid in IPL (that is $\neg\neg(\chi \vee \omega) \nvdash \neg\neg\chi \vee \neg\neg\omega$), we can see that the double negations in $\bigvee \neg\neg\psi_I$ can only be brought outside of the disjunction when there is a certain ψ_j such that $\psi_j \vdash \psi_i$ for every $i \in I$ (in that case $\bigvee \neg\neg\psi_I \equiv \neg\neg\psi_j$). But then we have $\psi_j \equiv \psi$, which contradicts our assumption that $\phi_i \not\equiv \phi$ for any $i \in I$. Therefore ϕ has to be irreducible.
In conclusion, any formula $(\bigwedge Q) \wedge \neg\neg\psi$ either is equivalent to $\bot$ or is irreducible. □

And finally we will determine the order of the irreducible elements of $[\wedge, \vee, \neg\neg]^n$.

Lemma 3.7. Let $F = [\wedge, \vee, \neg\neg]^n$ and let $\phi_1 \equiv (\bigwedge Q_1) \wedge \neg\neg\psi_1$ and $\phi_2 \equiv (\bigwedge Q_2) \wedge \neg\neg\psi_2$ be formulae of $\mathcal{I}([F)$, with $Q_1, Q_2 \subseteq Atom(F)$ and $\psi_1 \in G_1 = [\wedge, \vee, Atom(F) \setminus Q_1]$, $\psi_2 \in G_2 = [\wedge, \vee, Atom(F) \setminus Q_2]$. Then $\phi_1 \vdash_1 \phi_2$ if and only if one of the following holds:

- $Q_1 = Q_2$ and $\psi_1 \vdash_1 \psi_2$,
- $Q_1 = Q_2 \cup \{p\}$ for $p \notin Q_2$ and $\psi_2 \equiv \psi_1 \wedge p$.

Proof. First we will mention that either $Q_1 = Q_2$ or $Q_1 = Q_2 \cup \{p\}$ for some $p \notin Q_2$ has to hold. Of course if $Q_1 \supset Q_2$ we have that $(\bigwedge Q_1) \wedge \neg\neg\psi_1 \nvdash (\bigwedge Q_2) \wedge \neg\neg\psi_2$. If $Q_1 = Q_2 \cup \{p, q\}$ for $p \neq q$ and $p, q \notin Q_2$, then we can find the formula $(\bigwedge Q_2 \cup \{p\}) \wedge \neg\neg\psi_2$ which is not equivalent to $(\bigwedge Q_1) \wedge \neg\neg\psi_1$ and not equivalent to $(\bigwedge Q_2) \wedge \neg\neg\psi_2$, but for which $(\bigwedge Q_1) \wedge \neg\neg\psi_1 \vdash (\bigwedge Q_2 \cup \{p\}) \wedge \neg\neg\psi_2 \vdash (\bigwedge Q_2) \wedge \neg\neg\psi_2$.
If we consider the first case, wherein $Q_1 = Q_2$, then also $G_1 = G_2$ and $\psi_1, \psi_2 \in [\wedge, \vee, Atom(F) \setminus Q_1]$. So if $(\bigwedge Q_1) \wedge \neg\neg\psi_1 \vdash (\bigwedge Q^*) \wedge \neg\neg\psi^* \vdash (\bigwedge Q_2) \wedge \neg\neg\psi_2$ we know that $Q^* = Q_1 = Q_2$ and $\psi_1 \vdash \psi^* \vdash \psi_2$. We need $\psi^* \equiv \psi_1$ or $\psi^* \equiv \psi_2$, so therefore $\psi_1 \vdash_1 \psi_2$ has to be true.
In the second case we have $(\bigwedge Q_1) \wedge \neg\neg\psi_1 \vdash (\bigwedge Q^*) \wedge \neg\neg\psi^* \vdash (\bigwedge Q_2) \wedge \neg\neg(\psi_1 \wedge p)$. Since $(\bigwedge Q_1) \wedge \neg\neg\psi_1 \equiv (\bigwedge Q_1) \wedge \neg\neg(\psi_1 \wedge p)$ (because $p \in Q_1$), we know that $\psi_1 \wedge p \vdash$

$\psi^* \vdash \psi_1 \wedge p$, so therefore that $\psi^* \equiv \psi_1 \wedge p \equiv \psi_2$. Together with the fact that $Q_1 \supseteq Q^* \supseteq Q_2$ we can deduct that Q^* is equal to either Q_1 or Q_2, which makes $(Q^*) \wedge \neg\neg\psi^*$ equivalent to either $(\bigwedge Q_1) \wedge \neg\neg\psi_1$ or $(\bigwedge Q_2) \wedge \neg\neg\psi_2$ respectively, and thus $(\bigwedge Q_2 \cup \{p\}) \wedge \neg\neg\psi_1 \vdash_1 (\bigwedge Q_2) \wedge \neg\neg(\psi_1 \wedge p)$. $\square$

Chapter 4

Computing the Size of $[\wedge, \vee, \neg\neg]^n$

The computation of the size of a fragment with an exact model can be split in two main parts: generating the irreducible formulae and generating each upset of the exact model. We will first discuss the data types that are being used. Although the code itself is written in `C++`, this section will mostly use pseudocode to explain the algorithms, mainly because pseudocode allows us to use the mathematical notations from the other parts of this thesis. In the appendix the complete program can be found in `C++`.

4.1 Data Types

If we look back to how an irreducible formula of $[\wedge, \vee, \neg\neg]^n$ is described in the last chapter, we see that each formula is uniquely defined by the conjunction of a set of atoms Q and the double negation of a formula ψ. This ψ is a formula of $[\wedge, \vee]^n$, and therefore it is described as the disjunction of a set of conjunctions of sets of atoms P_i. So in the end our most basic building blocks can be conjunctions of atoms. We will use the least significant bits of an `int` in the range $[0, 15]$ to define the presence of a certain atom in the conjunction; that is, the ones-bit will stand for the atom p, the twos-bit for the atom q, the fours-bit for the atom r and the eights-bit for the atom s. Since the computation of $[\wedge, \vee, \neg\neg]^5$ will most likely be too expensive, the algorithm will be designed specifically to calculate $[\wedge, \vee, \neg\neg]^n$ for $n \in \{1, 2, 3, 4\}$. It can be easily modified to work with 5 atoms, however, by defining the 16s-bit as the fifth atom. In our case the 16s-bit will be used to represent $\bot$, which cannot be expressed as the conjunction of atoms alone.
We will define the data structures `forc`, `forcd` and `forcddn` to be formulae from $[\wedge]$, $[\wedge, \vee]$ and $\mathcal{I}([\wedge, \vee, \neg\neg])$ respectively. The `forc` will be only a conjunction, and therefore only consists of the integer we just described. The `forcd` will be a list of `forc`s, where the formula is represented as the disjunction of all elements of the list. Finally the `forcddn` will consist of a `forc` Q and a `forcd` ψ, representing $\bigwedge Q \wedge \neg\neg\psi$.
Apart from having a data representation of the formulae, we need the ability to relate formulae to one another. Such relation gives us the opportunity to put the structures in sorted lists and heaps, so we can construct a diagram. To do this the operators `==` (equivalence), `<` (entailment) and `-` (direct entailment) will be overloaded for our data structures.

In the case of `forc` it is pretty easy. We saw in the last chapter that if $\bigwedge Q_1 \vdash \bigwedge Q_2$, then all the atoms of Q_2 must be present in Q_1, therefore Q_1 `OR` Q_2 should equal Q_1, where `OR` stands for the bitwise or operator. Direct entailment follows if Q_1 has all the 1-bits of Q_2 plus one extra (that is in the four least significant places), and equality is only true if both integers are equal. The ideal part of this system is that the standard ordering of integers by value also implies that the `forc`s are ordered by our overloaded operators.
The ordering of `forcd`s in general is a bit more complicated, but it can be made easier if we only allow those `forcd` such that their list of `forc`s is maximal. This means that for any formula $\phi \in [\wedge]$ if $\phi \vdash \bigvee\Gamma$, where Γ is our set of conjunctions, then $\phi \in \Gamma$ (so Γ contains all irreducible elements that entail $\bigvee\Gamma$). In the case where the set of conjunctions is maximal, entailment of $\bigvee\Gamma_1 \vdash \bigvee\Gamma_2$ follows once again if Γ_2 is a subset of Γ_1. Since we can sort Γ_1 and Γ_2 (they contain elements of the `forc` type), this is not too difficult to check. Direct entailment follows from direct subsets and equality follows if the lists are equal.
Finally there is `forcddn`. We will start with direct entailment. There are two possibilities if $\bigwedge Q_1 \wedge \neg\neg\psi_1 \vdash \bigwedge Q_2 \wedge \neg\neg\psi_2$. In the first case $Q_1 = Q_2$ and we need the direct entailment of $\psi_1 \vdash_1 \psi_2$ both of which can be checked with the operators for `forc` and `forcd`. The second case is when Q_1 is a direct superset of Q_2. Now we need ψ_2 to be equivalent to $\psi_1 \wedge p$, where p is the difference between Q_1 and Q_2. However, in our program the formula $\psi_1 \wedge p$ needs to be in disjunctive normal form, which can be done with one application of the distributive rules. We therefore have to add p to each `forc` in the list of `forc`s contained in the `forcd` ψ_1. This is done by the `forcd` member-function `augment()`, which replaces the original integers of the `forc`s with integers that have the new atom included. Normal entailment now follows by checking that $Q_1 \supset Q_2$ and that $\psi_1 \wedge \bigwedge(Q_1 \setminus Q_2) \vdash \psi_2$ (that is, ψ_1 augmented with $Q_1 \setminus Q_2$). Equality is once again the simplest, two `forcddn`s are equal if the contained `forc`s and `forcd`s are equal.
The two remaining structures are the `poset` and the `uptrie`. The `poset` is a structure that contains a list of elements, of the type `poElem`. These `poElem`s are structures that contain an element `e` (of an undefined type, in this application they will be of the type `forc`, `forcd`, or `forcddn`), a list of children and a list of parents (both are lists of `poElem`s of the same type). Let p be a `poElem`, then the functions *element*(p), *children*(p) and *parents*(p) respectively return its element, its list of children and its list of parents. The list of `poElem`s in the `poset` will be topologically ordered, so that if p is the parent of q, then p will be earlier on the list than q. An `uptrie` is a trie structure that also contains an element `e` (of an undefined type), a list of children and its parental node. If u is an uptrie, then *element*(u), *children*(u) and *parent*(u) respectively return its element, its list of children and its parent. The function *upset*(u) will return a list containing *element*(u) and all elements of *upset*$($*parent*$(u))$, effectively returning the elements of the corresponding upset. We will speak of the *reach* of an uptrie u as the set containing u and the reach of its children (in terms of upsets this will be the set of upsets that are supersets of the original).

4.2 Generating Upsets

The algorithm used to transform a `poset` into a set of upsets is described in detail in [8], in Dutch. In this section we will discuss briefly how it works.
An uptrie is a standard trie structure that describes all the upsets of a given poset P. The root of the tree will be the empty set. Every other node will contain an element of P. Furthermore the tree is topologically sorted, therefore each child is topologically smaller than its parent and we can ascertain there will not be two different nodes corresponding to the same upset.
Finding all upsets will thus be as easy as counting all the nodes of an uptrie. In our case, if each node u contains an irreducible formula of fragment F, then the disjunction of $upset(u)$ will result in a unique formula in F. The hard part will be converting a poset to an uptrie. This is done by two algorithms, `findUpsetsWithElements()` and `generateUptrie()`. Below we have given the first algorithm in pseudo-code. The function takes two arguments; an uptrie u and a list (in our case of formulae) S, and it returns a list of uptries H:

```
function FINDUPSETSWITHELEMENTS(uptrie u, list S)
  list H ← ∅
  if u is topologically the largest element of S then
    S ← S \ {element(u)}
  end if
  if S = ∅ then
    H ← H ∪ {u}
  else if ∃v ∈ S : element(u) < v then
    return ∅
  end if
  for all uptries v ∈ children(u) do
    H ← H ∪ FINDUPSETSWITHELEMENTS(v, S)
  end for
  return H
end function
```

It will recursively determine the set of uptries that can be reached from u such that the corresponding upset contains all the elements in S. This is done with a few steps:

- If $element(u)$ is topologically the largest element of S, then all uptries that can be reached from u will correspond with an upset containing $element(u)$, so we can remove $element(u)$ from S (thereby continuing the search for the remainder of S).
- If S is empty, we know for sure that all uptries that can be reached from u correspond with upsets that contain S, so we can add u to our list of uptries H.

- If there is an element v in S such that $element(u)$ is smaller, we know that v will not be in the upsets that correspond with u. Since uptries are topologically sorted, we know that this will also be the case for all the uptries that can be reached from u. Therefore stop the search and return an empty list.

We can now search through all the children of u to find more uptries corresponding with upsets that contain S.
The second algorithm utilises `findUpsetsWithElements()` to construct an uptrie from a poset P. It will also depend on the fact that P is topologically ordered, and the resulting uptrie will have the same ordering. The function has one argument, the `poset` P, and returns the root of the uptrie:

```
function GENERATEUPTRIE(poset P)
   uptrie r
   for all poElems p ∈ P do
      list H = FINDUPSETSWITHELEMENTS(r, parents(p))
      for all uptries u ∈ H do
         let u_p be an uptrie with element p
         children(u) ← children(u) ∪ {u_p}
      end for
   end for
end function
```

Since P is topologically ordered, we know that at each iteration $element(p)$ is smaller than all elements that can be reached from r, therefore, since H contains all the uptrie nodes that have an upset containing all of $parents(p)$, we know that the union of such an upset and $element(p)$ is an upset as well, therefore p can be added as a child to each uptrie node.

4.3 Constructing Irreducible Formulae of $[\wedge, \vee]^n$

Of the two fragments we will compute, the generation of $\mathcal{I}([\wedge, \vee]^n)$ is the easiest, since this is equal to $[\wedge]^n$ without $\bot$. Our structure for a formula of $[\wedge]^n$ gives us a fairly easy solution: all irreducible formulae of $[\wedge, \vee]^n$ (with $n < 5$) are uniquely representable by an integer between 0 and 15. We want the possibility of choosing which atoms to include, therefore the function `irrCD()` has a list of `chars` (atoms) as argument. It returns a `poset` with `forcs`:

```
function IRRCD(list L)
   list F
   let f be a forc equivalent to ⋀L.
   for i from 0 to 15 do
      let g be a forc with integer i
```

```
        let p_g be a poElem with element g
        if f ⊢ g then
            F ← F ∪ {p_g}
        end if
    end for
    for all poElems p_i ∈ F[0,1,..] do
        for all poElems p_j ∈ F[i+1, i+2,..] do
            if element(p_i) ⊢_1 element(p_j) then
                add p_j as child of p_i.
            end if
        end for
    end for
    let P be a poset with list F
    return P
end function
```

In this function the list F is a list of poElems. To this list we will add all the conjunctions of atoms that are entailed by the formula $\bigwedge L$, with L the list of atoms (therefore making $[\wedge, L]$ without $\bot$). After that we will compare each poElem p_i with every other p_j in the list F, and if they entail directly we will add p_j as child of p_i. In the end we create and return a poset with list F.

4.4 Constructing Diagram of $[\wedge, \vee]^n$

The generation of the diagram of $[\wedge, \vee]^n$ will be generating all the formulae of $[\wedge, \vee]^n$ and putting them in the right order. Luckily we can already do this with all the tools we have. The function diagCD() also has a list of chars (atoms) as its only argument, like we saw in the last section, and the returning value will be a poset with forcds:

```
function DIAGCD(list L)
    list F
    poset I ← IRRCD(L)
    uptrie r ← GENERATEUPTRIE(I)
    list U ← FINDUPSETSWITHELEMENTS(r, ∅)
    for all uptries u ∈ U do
        let f be a forcd with list upset(u)
        let p_f be a poElem with element f
        F ← F ∪ {p_f}
    end for
    F ← SORT(F)
```

```
    for all poElems p_i ∈ F[0, 1, ..] do
        for all poElems p_j ∈ F[i + 1, i + 2, ..] do
            if element(p_i) ⊢_1 element(p_j) then
                add p_j as child of p_i.
            end if
        end for
    end for
    let P be a poset with list F
    return P
end function
```

First we generate $\mathcal{I}([\wedge, \vee])$ using the already familiar function `irrCD()`, convert this poset to an `uptrie` using `generateUptrie()`, then make a list with all the nodes of the uptrie by calling `findUpsetsWithElements()` with an empty list as second argument and convert each uptrie node to a `poElem` with type `forcd`. All the `poElem`s will be put inside F and sorted by the number of elements of the upset that resulted in each formula. This also implies sorting by entailment. Now each `poElem` p_i will be compared with every other p_j in the list F, and if they entail directly we will add p_j as child of p_i. Finally we return a `poset` with list F.

4.5 Constructing Irreducible Formulae of $[\wedge, \vee, \neg\neg]^n$

The function that constructs $\mathcal{I}([\wedge, \vee, \neg\neg]^n)$ is called `irrCDDN()` and like the last two functions, it has a list of `char`s L as argument and returns a `poset` with `forcddn`s. One additional function that is used is the `complementaryAtoms()` function, which takes two arguments: a `forc` f and a list of `char`s L. If L_f is the list of atoms that are used in f, then `complementaryAtoms()` returns the list $L \setminus L_f$.

```
function IRRCDDN(list L)
    list F
    poset P_Q ← IRRCD(L)
    for all poElems q ∈ P_Q do
        list C ← COMPLEMENTARYATOMS(element(q), L)
        poset P_Γ ← DIAGCD(C) \ {⊥}
        for all poElems ψ ∈ P_Γ do
            let f be a forcddn with forc element(q) and forcd element(ψ)
            let p_f be a poElem with element f
            F ← F ∪ {p_f}
        end for
    end for
```

```
    for all poElems p_i ∈ F[0, 1, ..] do
      for all poElems p_j ∈ F[i + 1, i + 2, ..] do
        if element(p_i) ⊢_1 element(p_j) then
          add p_j as child of p_i.
        end if
      end for
    end for
    let P be a poset with list F
    return P
  end function
```

The algorithm first creates a poset P_Q, containing all the possible values for Q in $\bigwedge Q \wedge \neg\neg\psi$. They will be the `forc` parts of the `forcddns` that will be generated. It then creates for every `forc` in P_Q a poset of `forcds`, that will be the ψ parts of the formulae, or the `forcd` parts of the `forcddns`. In the inner `for`-loop each `forcddn` is constructed, gets put in a `poElem` and added to the list F. Then finally the same procedure as with the last two function happens: the child-parent relations of the poset are determined and the `poset` with list F is returned.

4.6 Counting the Size of $[\wedge, \vee, \neg\neg]^n$

The very last step is to convert a poset of `forcddn` to an uptrie with `generateUptrie()` and counting the number of nodes of this uptrie. This all happens in the `main` function and is quite straightforward. It is possible to make the diagram of $[\wedge, \vee, \neg\neg]^n$ with a method similar to `diagCD()`. Since the main goal of this text was only to find the size of $[\wedge, \vee, \neg\neg]^n$, such a function has not been implemented.

Chapter 5

Conclusion

This text has explained a way to calculate the size of the fragment $[\wedge, \vee, \neg\neg]^n$ for arbitrary n and describes an algorithm that can compute this number for $n < 5$. The number $\#[\wedge, \vee]^n$ is equal to the Dedekind number D_n, and these numbers grow super-exponentially, see [7]. Since $[\wedge, \vee]^n$ is embedded in $[\wedge, \vee, \neg\neg]^n$, we know that $\#[\wedge, \vee, \neg\neg]^n$ also grows at least super-exponentially. The complexity of the given algorithm is therefore quite poor. It produces results for $n < 4$ without a problem and in a short timespan. The sizes for $n = 1$, $n = 2$ and $n = 3$ are 4, 21 and 1891 respectively. This is in line with the results showed in [6] (Nota bene, in that text the sizes are computed without $\top$ and $\bot$, therefore the sizes result in the numbers 2, 19 and 1889).
However, the size for $n = 4$ is problematic. The diagram of $\mathcal{I}([\wedge, \vee, \neg\neg]^4)$ is computed pretty quickly and learns us that there are 282 irreducible formulae. The list of computed elements of $\mathcal{I}([\wedge, \vee, \neg\neg]^4)$ is given in the appendix. The program wil then proceed to generate the uptrie for this poset, by looping through the `poElems`. Timing the duration of each iteration shows that the duration will have a more or less exponential growth, and extrapolation of these results indicates that the estimated time needed to complete the algorithm will be in the order of hundreds of millions of years.
It might be possible to improve the current algorithm by using some properties of the diagrams. The lattices $\mathcal{I}([\wedge, \vee, \neg\neg]^n)$ are *stratified*, which means it can be divided in layers A_0 to A_k, such that for all elements in A_i the children of those elements are in A_{i+1}. Naturally these layers are only a fraction of the size of the original diagram, so we can compute the upsets that have at least one element in a layer A_i but none in layer A_{i+1}. However this still might be difficult, since the number of elements per layer grows rapidly. For the fragment with four atoms, as a matter of fact, the number of elements per layer is given by:

$$\{1, 4, 10, 16, 19, 22, 22, 22, 25, 22, 23, 24, 19, 18, 13, 10, 6, 4, 1, 1\}$$

The sum of the total number of subsets per layer is approximately 76 million, therefore the computation might still be quite expensive.

List of Formulae in $\mathcal{I}([\wedge, \vee, \neg\neg]^4)$

1. $p \wedge q \wedge r \wedge s$
2. $q \wedge r \wedge s \wedge \neg\neg p$
3. $q \wedge r \wedge s$
4. $p \wedge r \wedge s \wedge \neg\neg q$
5. $p \wedge r \wedge s$
6. $r \wedge s \wedge \neg\neg(p \wedge q)$
7. $r \wedge s \wedge \neg\neg p$
8. $r \wedge s \wedge \neg\neg q$
9. $r \wedge s \wedge \neg\neg[q \vee p]$
10. $r \wedge s$
11. $p \wedge q \wedge s \wedge \neg\neg r$
12. $p \wedge q \wedge s$
13. $q \wedge s \wedge \neg\neg(p \wedge r)$
14. $q \wedge s \wedge \neg\neg p$
15. $q \wedge s \wedge \neg\neg r$
16. $q \wedge s \wedge \neg\neg(r \vee p)$
17. $q \wedge s$
18. $p \wedge s \wedge \neg\neg(q \wedge r)$
19. $p \wedge s \wedge \neg\neg q$
20. $p \wedge s \wedge \neg\neg r$
21. $p \wedge s \wedge \neg\neg(r \vee q)$
22. $p \wedge s$
23. $s \wedge \neg\neg(p \wedge q \wedge r)$
24. $s \wedge \neg\neg(p \wedge q)$
25. $s \wedge \neg\neg(p \wedge r)$
26. $s \wedge \neg\neg(q \wedge r)$
27. $s \wedge \neg\neg[(p \wedge r) \vee (p \wedge q)]$
28. $s \wedge \neg\neg[(q \wedge r) \vee (p \wedge q)]$
29. $s \wedge \neg\neg[(q \wedge r) \vee (p \wedge r)]$
30. $s \wedge \neg\neg p$
31. $s \wedge \neg\neg q$
32. $s \wedge \neg\neg[(q \wedge r) \vee (p \wedge r) \vee (p \wedge q)]$
33. $s \wedge \neg\neg r$
34. $s \wedge \neg\neg[(q \wedge r) \vee p]$
35. $s \wedge \neg\neg[(p \wedge r) \vee q]$
36. $s \wedge \neg\neg[r \vee (p \wedge q)]$
37. $s \wedge \neg\neg(q \vee p)$
38. $s \wedge \neg\neg(r \vee p)$
39. $s \wedge \neg\neg(r \vee q)$
40. $s \wedge \neg\neg(r \vee q \vee p)$
41. s
42. $p \wedge q \wedge r \wedge \neg\neg s$
43. $p \wedge q \wedge r$
44. $q \wedge r \wedge \neg\neg(p \wedge s)$
45. $q \wedge r \wedge \neg\neg p$
46. $q \wedge r \wedge \neg\neg s$
47. $q \wedge r \wedge \neg\neg(s \vee p)$
48. $q \wedge r$
49. $p \wedge r \wedge \neg\neg(q \wedge s)$
50. $p \wedge r \wedge \neg\neg q$
51. $p \wedge r \wedge \neg\neg s$
52. $p \wedge r \wedge \neg\neg(s \vee q)$
53. $p \wedge r$
54. $r \wedge \neg\neg(p \wedge q \wedge s)$
55. $r \wedge \neg\neg(p \wedge q)$
56. $r \wedge \neg\neg(p \wedge s)$
57. $r \wedge \neg\neg(q \wedge s)$
58. $r \wedge \neg\neg[(p \wedge s) \vee (p \wedge q)]$
59. $r \wedge \neg\neg[(q \wedge s) \vee (p \wedge q)]$
60. $r \wedge \neg\neg[(q \wedge s) \vee (p \wedge s)]$
61. $r \wedge \neg\neg p$
62. $r \wedge \neg\neg q$
63. $r \wedge \neg\neg[(q \wedge s) \vee (p \wedge s) \vee (p \wedge q)]$
64. $r \wedge \neg\neg s$
65. $r \wedge \neg\neg[(q \wedge s) \vee p]$
66. $r \wedge \neg\neg[(p \wedge s) \vee q]$
67. $r \wedge \neg\neg[s \vee (p \wedge q)]$
68. $r \wedge \neg\neg(q \vee p)$
69. $r \wedge \neg\neg(s \vee p)$
70. $r \wedge \neg\neg(s \vee q)$
71. $r \wedge \neg\neg(s \vee q \vee p)$
72. r
73. $p \wedge q \wedge \neg\neg(r \wedge s)$
74. $p \wedge q \wedge \neg\neg r$
75. $p \wedge q \wedge \neg\neg s$
76. $p \wedge q \wedge \neg\neg(s \vee r)$

77. $p \wedge q$
78. $q \wedge \neg\neg(p \wedge r \wedge s)$
79. $q \wedge \neg\neg(p \wedge r)$
80. $q \wedge \neg\neg(p \wedge s)$
81. $q \wedge \neg\neg(r \wedge s)$
82. $q \wedge \neg\neg[(p \wedge s) \vee (p \wedge r)]$
83. $q \wedge \neg\neg[(r \wedge s) \vee (p \wedge r)]$
84. $q \wedge \neg\neg[(r \wedge s) \vee (p \wedge s)]$
85. $q \wedge \neg\neg p$
86. $q \wedge \neg\neg r$
87. $q \wedge \neg\neg[(r \wedge s) \vee (p \wedge s) \vee (p \wedge r)]$
88. $q \wedge \neg\neg s$
89. $q \wedge \neg\neg[(r \wedge s) \vee p]$
90. $q \wedge \neg\neg[(p \wedge s) \vee r]$
91. $q \wedge \neg\neg[s \vee (p \wedge r)]$
92. $q \wedge \neg\neg(r \vee p)$
93. $q \wedge \neg\neg(s \vee p)$
94. $q \wedge \neg\neg(s \vee r)$
95. $q \wedge \neg\neg(s \vee r \vee p)$
96. q
97. $p \wedge \neg\neg(q \wedge r \wedge s)$
98. $p \wedge \neg\neg(q \wedge r)$
99. $p \wedge \neg\neg(q \wedge s)$
100. $p \wedge \neg\neg(r \wedge s)$
101. $p \wedge \neg\neg[(q \wedge s) \vee (q \wedge r)]$
102. $p \wedge \neg\neg[(r \wedge s) \vee (q \wedge r)]$
103. $p \wedge \neg\neg[(r \wedge s) \vee (q \wedge s)]$
104. $p \wedge \neg\neg q$
105. $p \wedge \neg\neg r$
106. $p \wedge \neg\neg[(r \wedge s) \vee (q \wedge s) \vee (q \wedge r)]$
107. $p \wedge \neg\neg s$
108. $p \wedge \neg\neg[(r \wedge s) \vee q]$
109. $p \wedge \neg\neg[(q \wedge s) \vee r]$
110. $p \wedge \neg\neg[s \vee (q \wedge r)]$
111. $p \wedge \neg\neg(r \vee q)$
112. $p \wedge \neg\neg(s \vee q)$
113. $p \wedge \neg\neg(s \vee r)$
114. $p \wedge \neg\neg(s \vee r \vee q)$
115. p
116. $\neg\neg(p \wedge q \wedge r \wedge s)$
117. $\neg\neg(p \wedge q \wedge r)$
118. $\neg\neg(p \wedge q \wedge s)$
119. $\neg\neg(p \wedge r \wedge s)$
120. $\neg\neg(q \wedge r \wedge s)$
121. $\neg\neg[(p \wedge q \wedge s) \vee (p \wedge q \wedge r)]$
122. $\neg\neg[(p \wedge r \wedge s) \vee (p \wedge q \wedge r)]$
123. $\neg\neg[(p \wedge r \wedge s) \vee (p \wedge q \wedge s)]$
124. $\neg\neg[(q \wedge r \wedge s) \vee (p \wedge q \wedge r)]$
125. $\neg\neg[(q \wedge r \wedge s) \vee (p \wedge q \wedge s)]$
126. $\neg\neg[(q \wedge r \wedge s) \vee (p \wedge r \wedge s)]$
127. $\neg\neg(p \wedge q)$
128. $\neg\neg(p \wedge r)$
129. $\neg\neg[(p \wedge r \wedge s) \vee (p \wedge q \wedge s) \vee (p \wedge q \wedge r)]$
130. $\neg\neg(p \wedge s)$
131. $\neg\neg(q \wedge r)$
132. $\neg\neg[(q \wedge r \wedge s) \vee (p \wedge q \wedge s) \vee (p \wedge q \wedge r)]$
133. $\neg\neg(q \wedge s)$
134. $\neg\neg[(q \wedge r \wedge s) \vee (p \wedge r \wedge s) \vee (p \wedge q \wedge r)]$
135. $\neg\neg[(q \wedge r \wedge s) \vee (p \wedge r \wedge s) \vee (p \wedge q \wedge s)]$
136. $\neg\neg(r \wedge s)$
137. $\neg\neg[(p \wedge r \wedge s) \vee (p \wedge q)]$
138. $\neg\neg[(p \wedge q \wedge s) \vee (p \wedge r)]$
139. $\neg\neg[(p \wedge s) \vee (p \wedge q \wedge r)]$
140. $\neg\neg[(q \wedge r \wedge s) \vee (p \wedge q)]$
141. $\neg\neg[(p \wedge q \wedge s) \vee (q \wedge r)]$
142. $\neg\neg[(q \wedge s) \vee (p \wedge q \wedge r)]$
143. $\neg\neg[(q \wedge r \wedge s) \vee (p \wedge r)]$
144. $\neg\neg[(p \wedge r \wedge s) \vee (q \wedge r)]$
145. $\neg\neg[(q \wedge r \wedge s) \vee (p \wedge r \wedge s) \vee (p \wedge q \wedge s) \vee (p \wedge q \wedge r)]$
146. $\neg\neg[(q \wedge r \wedge s) \vee (p \wedge s)]$
147. $\neg\neg[(p \wedge r \wedge s) \vee (q \wedge s)]$
148. $\neg\neg[(r \wedge s) \vee (p \wedge q \wedge r)]$
149. $\neg\neg[(r \wedge s) \vee (p \wedge q \wedge s)]$
150. $\neg\neg[(p \wedge r) \vee (p \wedge q)]$
151. $\neg\neg[(p \wedge s) \vee (p \wedge q)]$
152. $\neg\neg[(p \wedge s) \vee (p \wedge r)]$
153. $\neg\neg[(q \wedge r) \vee (p \wedge q)]$
154. $\neg\neg[(q \wedge s) \vee (p \wedge q)]$
155. $\neg\neg[(q \wedge s) \vee (q \wedge r)]$
156. $\neg\neg[(q \wedge r) \vee (p \wedge r)]$
157. $\neg\neg[(q \wedge r \wedge s) \vee (p \wedge r \wedge s) \vee (p \wedge q)]$
158. $\neg\neg[(q \wedge r \wedge s) \vee (p \wedge q \wedge s) \vee (p \wedge r)]$
159. $\neg\neg[(p \wedge r \wedge s) \vee (p \wedge q \wedge s) \vee (q \wedge r)]$
160. $\neg\neg[(q \wedge r \wedge s) \vee (p \wedge s) \vee (p \wedge q \wedge r)]$
161. $\neg\neg[(p \wedge r \wedge s) \vee (q \wedge s) \vee (p \wedge q \wedge r)]$
162. $\neg\neg[(q \wedge s) \vee (p \wedge s)]$
163. $\neg\neg[(r \wedge s) \vee (p \wedge r)]$
164. $\neg\neg[(r \wedge s) \vee (q \wedge r)]$
165. $\neg\neg[(r \wedge s) \vee (p \wedge q \wedge s) \vee (p \wedge q \wedge r)]$
166. $\neg\neg[(r \wedge s) \vee (p \wedge s)]$
167. $\neg\neg[(r \wedge s) \vee (q \wedge s)]$
168. $\neg\neg[(p \wedge s) \vee (p \wedge r) \vee (p \wedge q)]$
169. $\neg\neg[(q \wedge s) \vee (q \wedge r) \vee (p \wedge q)]$
170. $\neg\neg[(q \wedge r \wedge s) \vee (p \wedge r) \vee (p \wedge q)]$
171. $\neg\neg[(p \wedge r \wedge s) \vee (q \wedge r) \vee (p \wedge q)]$
172. $\neg\neg[(p \wedge q \wedge s) \vee (q \wedge r) \vee (p \wedge r)]$

173. $\neg\neg[(q \wedge r \wedge s) \vee (p \wedge s) \vee (p \wedge q)]$
174. $\neg\neg[(q \wedge r \wedge s) \vee (p \wedge s) \vee (p \wedge r)]$
175. $\neg\neg[(p \wedge s) \vee (q \wedge r)]$
176. $\neg\neg[(p \wedge r \wedge s) \vee (q \wedge s) \vee (p \wedge q)]$
177. $\neg\neg[(q \wedge s) \vee (p \wedge r)]$
178. $\neg\neg[(p \wedge r \wedge s) \vee (q \wedge s) \vee (q \wedge r)]$
179. $\neg\neg[(q \wedge s) \vee (p \wedge s) \vee (p \wedge q \wedge r)]$
180. $\neg\neg[(r \wedge s) \vee (q \wedge r) \vee (p \wedge r)]$
181. $\neg\neg[(r \wedge s) \vee (p \wedge q)]$
182. $\neg\neg[(r \wedge s) \vee (p \wedge q \wedge s) \vee (p \wedge r)]$
183. $\neg\neg[(r \wedge s) \vee (p \wedge q \wedge s) \vee (q \wedge r)]$
184. $\neg\neg[(r \wedge s) \vee (p \wedge s) \vee (p \wedge q \wedge r)]$
185. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee (p \wedge q \wedge r)]$
186. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee (p \wedge s)]$
187. $\neg\neg p$
188. $\neg\neg q$
189. $\neg\neg[(q \wedge r) \vee (p \wedge r) \vee (p \wedge q)]$
190. $\neg\neg[(q \wedge r \wedge s) \vee (p \wedge s) \vee (p \wedge r) \vee (p \wedge q)]$
191. $\neg\neg[(p \wedge s) \vee (q \wedge r) \vee (p \wedge q)]$
192. $\neg\neg[(p \wedge s) \vee (q \wedge r) \vee (p \wedge r)]$
193. $\neg\neg[(q \wedge s) \vee (p \wedge r) \vee (p \wedge q)]$
194. $\neg\neg[(p \wedge r \wedge s) \vee (q \wedge s) \vee (q \wedge r) \vee (p \wedge q)]$
195. $\neg\neg[(q \wedge s) \vee (q \wedge r) \vee (p \wedge r)]$
196. $\neg\neg[(q \wedge s) \vee (p \wedge s) \vee (p \wedge q)]$
197. $\neg\neg[(q \wedge s) \vee (p \wedge s) \vee (p \wedge r)]$
198. $\neg\neg[(q \wedge s) \vee (p \wedge s) \vee (q \wedge r)]$
199. $\neg\neg r$
200. $\neg\neg[(r \wedge s) \vee (p \wedge r) \vee (p \wedge q)]$
201. $\neg\neg[(r \wedge s) \vee (q \wedge r) \vee (p \wedge q)]$
202. $\neg\neg[(r \wedge s) \vee (p \wedge q \wedge s) \vee (q \wedge r) \vee (p \wedge r)]$
203. $\neg\neg[(r \wedge s) \vee (p \wedge s) \vee (p \wedge q)]$
204. $\neg\neg[(r \wedge s) \vee (p \wedge s) \vee (p \wedge r)]$
205. $\neg\neg[(r \wedge s) \vee (p \wedge s) \vee (q \wedge r)]$
206. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee (p \wedge q)]$
207. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee (p \wedge r)]$
208. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee (q \wedge r)]$
209. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee (p \wedge s) \vee (p \wedge q \wedge r)]$
210. $\neg\neg s$
211. $\neg\neg[(q \wedge r \wedge s) \vee p]$
212. $\neg\neg[(p \wedge s) \vee (q \wedge r) \vee (p \wedge r) \vee (p \wedge q)]$
213. $\neg\neg[(p \wedge r \wedge s) \vee q]$
214. $\neg\neg[(q \wedge s) \vee (q \wedge r) \vee (p \wedge r) \vee (p \wedge q)]$
215. $\neg\neg[(q \wedge s) \vee (p \wedge s) \vee (p \wedge r) \vee (p \wedge q)]$
216. $\neg\neg[(q \wedge s) \vee (p \wedge s) \vee (q \wedge r) \vee (p \wedge q)]$
217. $\neg\neg[(q \wedge s) \vee (p \wedge s) \vee (q \wedge r) \vee (p \wedge r)]$
218. $\neg\neg[(r \wedge s) \vee (q \wedge r) \vee (p \wedge r) \vee (p \wedge q)]$
219. $\neg\neg[(p \wedge q \wedge s) \vee r]$
220. $\neg\neg[(r \wedge s) \vee (p \wedge s) \vee (p \wedge r) \vee (p \wedge q)]$
221. $\neg\neg[(r \wedge s) \vee (p \wedge s) \vee (q \wedge r) \vee (p \wedge q)]$
222. $\neg\neg[(r \wedge s) \vee (p \wedge s) \vee (q \wedge r) \vee (p \wedge r)]$
223. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee (p \wedge r) \vee (p \wedge q)]$
224. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee (q \wedge r) \vee (p \wedge q)]$
225. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee (q \wedge r) \vee (p \wedge r)]$
226. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee (p \wedge s) \vee (p \wedge q)]$
227. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee (p \wedge s) \vee (p \wedge r)]$
228. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee (p \wedge s) \vee (q \wedge r)]$
229. $\neg\neg[s \vee (p \wedge q \wedge r)]$
230. $\neg\neg[(q \wedge r) \vee p]$
231. $\neg\neg[(p \wedge r) \vee q]$
232. $\neg\neg[(q \wedge s) \vee p]$
233. $\neg\neg[(p \wedge s) \vee q]$
234. $\neg\neg[(q \wedge s) \vee (p \wedge s) \vee (q \wedge r) \vee (p \wedge r) \vee (p \wedge q)]$
235. $\neg\neg[r \vee (p \wedge q)]$
236. $\neg\neg[(r \wedge s) \vee p]$
237. $\neg\neg[(r \wedge s) \vee (p \wedge s) \vee (q \wedge r) \vee (p \wedge r) \vee (p \wedge q)]$
238. $\neg\neg[(p \wedge s) \vee r]$
239. $\neg\neg[(r \wedge s) \vee q]$
240. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee (q \wedge r) \vee (p \wedge r) \vee (p \wedge q)]$
241. $\neg\neg[(q \wedge s) \vee r]$
242. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee (p \wedge s) \vee (p \wedge r) \vee (p \wedge q)]$
243. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee (p \wedge s) \vee (q \wedge r) \vee (p \wedge q)]$
244. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee (p \wedge s) \vee (q \wedge r) \vee (p \wedge r)]$
245. $\neg\neg[s \vee (p \wedge q)]$
246. $\neg\neg[s \vee (p \wedge r)]$
247. $\neg\neg[s \vee (q \wedge r)]$
248. $\neg\neg[(q \wedge s) \vee (q \wedge r) \vee p]$
249. $\neg\neg[(p \wedge s) \vee (p \wedge r) \vee q]$
250. $\neg\neg[(r \wedge s) \vee (q \wedge r) \vee p]$
251. $\neg\neg[(p \wedge s) \vee r \vee (p \wedge q)]$
252. $\neg\neg[(r \wedge s) \vee (p \wedge r) \vee q]$
253. $\neg\neg[(q \wedge s) \vee r \vee (p \wedge q)]$
254. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee p]$
255. $\neg\neg[(r \wedge s) \vee (p \wedge s) \vee q]$
256. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee (p \wedge s) \vee (q \wedge r) \vee (p \wedge r) \vee (p \wedge q)]$
257. $\neg\neg[(q \wedge s) \vee (p \wedge s) \vee r]$
258. $\neg\neg[s \vee (p \wedge r) \vee (p \wedge q)]$
259. $\neg\neg[s \vee (q \wedge r) \vee (p \wedge q)]$
260. $\neg\neg[s \vee (q \wedge r) \vee (p \wedge r)]$
261. $\neg\neg(q \vee p)$
262. $\neg\neg(r \vee p)$
263. $\neg\neg(r \vee q)$
264. $\neg\neg[(r \wedge s) \vee (q \wedge s) \vee (q \wedge r) \vee p]$
265. $\neg\neg[(r \wedge s) \vee (p \wedge s) \vee (p \wedge r) \vee q]$
266. $\neg\neg[(q \wedge s) \vee (p \wedge s) \vee r \vee (p \wedge q)]$
267. $\neg\neg(s \vee p)$

268. $\neg\neg(s \vee q)$
269. $\neg\neg[s \vee (q \wedge r) \vee (p \wedge r) \vee (p \wedge q)]$
270. $\neg\neg(s \vee r)$
271. $\neg\neg[(r \wedge s) \vee q \vee p]$
272. $\neg\neg[(q \wedge s) \vee r \vee p]$
273. $\neg\neg[(p \wedge s) \vee r \vee q]$
274. $\neg\neg[s \vee (q \wedge r) \vee p]$
275. $\neg\neg[s \vee (p \wedge r) \vee q]$
276. $\neg\neg[s \vee r \vee (p \wedge q)]$
277. $\neg\neg(r \vee q \vee p)$
278. $\neg\neg(s \vee q \vee p)$
279. $\neg\neg(s \vee r \vee p)$
280. $\neg\neg(s \vee r \vee q)$
281. $\neg\neg(s \vee r \vee q \vee p)$
282. $\top$

Appendix B

Program

```cpp
#include <iostream>
#include <list>
#include <string>
#include <ctime>
#include <cstdio>

using namespace std;

// Function that separates elements in a list l with the element
// t, used for printing
template<typename T> list<T> intersperse(list<T> l, T t) {
    list<T> k = {*l.begin()};
    for (typename list<T>::iterator it = l.begin(); ++it != l.end();) {
        k.push_back(t);
        k.push_back(*it);
    }
    return k;
}

// Node of a partial ordered set, contains the element e, and two
// lists of pointers to the children and parents of the node.
template<typename T> struct poElem {
    T e;
    list<poElem<T>*> children;
    list<poElem<T>*> parents;

    // constructor
    poElem(T t) { e = t; }

    // adds a child to this node
    void addChild(poElem<T> *c) { children.push_back(c);
    c->parents.push_back(this); }

    // returns the list of elements that are in the parental
    // nodes
    list<T> listParents() {
        list<T> l;
        for (poElem<T> *p : parents)
```

```
            l.push_back(p->e);
        return l;
40  }

    string print() { return e.print(); }
};

// overload equality operator between a node and an element
template<typename T> bool operator == (poElem<T> *p, T t) { return p->e == t; }

// A partial ordered set, contains a list of nodes (poElem) l.
template<typename T> struct poset {
50  list<poElem<T>*> l;

    // constructor
    poset(list<poElem<T>*> x) { l = x; }

    // returns the number of nodes
    unsigned long size() { return l.size(); }

    string print() {
        list<string> ss;
60      string s = "";
        for (poElem<T> *p : l)
            ss.push_back(p->print());
        ss = intersperse<string>(ss, "\n");
        for (string t : ss)
            s += t;
        return s;
    }
};

70 // Uptrie node, contains an element e, a list of pointers to the
// children of this node and a pointer to its parent.
template<typename T> struct uptrie {
    T e;
    list<uptrie<T>*> children;
    uptrie<T> *parent;

    // constructors
    uptrie() { e = T::bottom(); parent = NULL; }
    uptrie(T t) { e = t; parent = NULL; }
80
    // adds a child node
    void addChild(uptrie<T> &u) { u.parent = this; children.push_back(&u); }

    // returns the total number of nodes that can be reached from
    // this node
```

```
    int size() {
        int i = 1;
        for (uptrie<T> *child : children) i += child->size();
        return i;
    }

    // returns the path from the root of the uptrie to this node
    list<T> Upset() {
        list<T> l = {e};
        if (parent != nullptr)
            l.splice(l.begin(),parent->Upset());
        return l;
    }

    // prints an uptrie for debugging purposes
    ostream& deepPrint(ostream& os, int i) {
        for (int j=0; j < i; j++) os << "\t";
        os << e << " {\n";
        for (uptrie<T> *u : children)
            u->deepPrint(os, i+1);
        for (int j=0; j < i; j++) os << "\t";
        os << "}\n";
        return os;
    }
};

// overloading the output stream operator << for poElems, posets
// and uptries
template<typename T> ostream& operator << (ostream& os, poElem<T> p) { return
    os << p.print(); }
template<typename T> ostream& operator << (ostream& os, poset<T> e) { return
    os << e.print(); }
template<typename T> ostream& operator << (ostream& os, uptrie<T> u) { return
    os << u.e; }

// Finds the collections of Upsets that contain the elements
// in the sorted list s, where u is the root of the uptrie it
// searches in
template<typename T> void findUpsetsWithElements(uptrie<T> &u, list<T> s,
    list<uptrie<T>*> &h) {


    // if u.e is in s, it is the first element (s is sorted)
    if (s.size() > 0 && u.e == *s.begin())
        s.pop_front(); // u.e has been found, remove from the list

    // if the list is empty, then the current u corresponds to an
    // upset that contains s
```

```
130     if (s.size() == 0)
            h.push_back(&u);

        // if there exists an element is s that is smaller than y,
        // then this u and all its children will not contain s,
        // therefore return an empty list.
         for (T x : s)
         if (x < u.e)
         return *new list<uptrie<T>*>;

140     // continue the search in each child of u
        for (uptrie<T> *child : u.children)
            // merge the found list with the existing lists
            findUpsetsWithElements(*child, s, h);
    }

    // converts a sorted poset p into the root of an uptrie that
    // contains all Upsets of p
    template<typename T> uptrie<T> generateUptrie(poset<T> p) {
        uptrie<T> root = *new uptrie<T>();
150     for (poElem<T> *e : p.l) {
            // Since p is sorted, if all parents of e have been added
            // to the uptrie, so a node containing e can be added to
            // all uptrie nodes containing all of the parents of e
            list<uptrie<T>*> h;
            findUpsetsWithElements(root, e->listParents(), h);
            for (uptrie<T> *v : h)
                v->addChild(*new uptrie<T>(e->e));
        }
        return root;
160 }

    // A data structure to represent a formula containing only
    // conjunctions; the fragment [and]^4.
    struct forc {
        // the last 4 bits represent a specific atom:
        // 0 stands for T (which is the conjunction of the empty set)
        // the ones digit represents atom p
        // the twos digit represents atom q
        // the fours digit represents atom r
170     // the eights digit represents atom s
        // 16 (10000) stands for contradiction
        int Q;

        // constructors
        forc() {}
        forc(list<char> l) {
            Q = 0;
```

```
        if (*find(l.begin(),l.end(),'p') == 'p') Q = Q | 1;
        if (*find(l.begin(),l.end(),'q') == 'q') Q = Q | 2;
180     if (*find(l.begin(),l.end(),'r') == 'r') Q = Q | 4;
        if (*find(l.begin(),l.end(),'s') == 's') Q = Q | 8;
    }
    static forc bottom() { // returns contradiction
        forc f;
        f.Q = 16;
        return f;
    };

    string print() {
190
        if (Q == 0) return "T";
        else if (Q == 16) return "B";
        list<char> l; // list of atoms
        if ((Q & 1) == 1) l.push_back('p');
        if ((Q & 2) == 2) l.push_back('q');
        if ((Q & 4) == 4) l.push_back('r');
        if ((Q & 8) == 8) l.push_back('s');
        // intersperse the ampersand in between the atoms
        l = intersperse<char>(l, '&');
200     string s = "";
        for (char c : l) // concatenate the list
            s += c;
        return s;
    };
};

struct indexSort {
    // sorts forc by their integer value, also implies sorting by
    // entailment
210 inline bool operator() (const forc& f1, const forc& f2) {
        return (f2.Q < f1.Q);
    }
};

// overloading operators, the minus operator will function as
// direct entailment, since < will be used as general entailment.
ostream& operator << (ostream& os, forc f) { return os << f.print(); }
// equivalence
bool operator == (forc f, forc g) { return f.Q == g.Q; }
220 bool operator < (forc f, forc g) { return (f.Q | g.Q) == f.Q; } // entailment
bool operator - (forc f, forc g) { return (f < g) && ((f.Q ^ g.Q) == 1 || (f.Q
    ^ g.Q) == 2 || (f.Q ^ g.Q) == 4 || (f.Q ^ g.Q) == 8); } // direct
    entailment

// A data structure that represents the disjunction of
```

```
// conjunctions; the fragment [and,or]^4
struct forcd {
    // list of conjunctions. This list will be maximal
    list<forc> P;

    // constructors
    forcd() {}
    forcd(list<forc> fs) { fs.sort(indexSort()); P = fs; }

    // converts a formula p to the formula (p and a), where a is
    // an atom. used to find direct entailment for the
    // irreducible formulae of [and,or,double negation]
    forcd augment(int i) { // i is in {1,2,4,8}
        list<forc> l;
        // the atom gets added to each conjunction in P
        for (forc f : P){
            if (!(f == forc::bottom())) {
                forc g;
                g.Q = (f.Q | i);
                l.push_back(g);
            } else
                // if f is the bottom element, then (f and a) = f
                l.push_back(forc::bottom());
        }
        return *new forcd(l);
    }

    // Returns the number of conjunctions
    unsigned long size() { return P.size(); }

    string print() {
        string s = "(";
        list<string> ss;
        for (forc f : P)
            ss.push_back(f.print());
        ss = intersperse<string>(ss, ")|(");
        for (string t : ss)
            s += t;
        return s + ")";
    }
};

ostream& operator << (ostream& os, forcd f) { return os << f.print(); }
// equivalence
bool operator == (forcd f, forcd g) { return f.P == g.P; }
bool operator < (forcd f, forcd g) { // entailment
    // Since f.P and g.P are assumed to be maximal, we have f < g
    // iff f.P is a subset of g.P
```

```
    if (f.P.size() > g.P.size())
        // f.P is a subset, therefore smaller than g.P
        return false;

    // check if all conjunctions in f are present in g by
    // iterating through the list
    list<forc>::iterator gi = g.P.begin();
    for (list<forc>::iterator fi = f.P.begin(); fi != f.P.end(); fi++)
280     while (!(*fi == *gi)) {
            gi++;
            if (gi == g.P.end())
                return false;
        }
    return true;
}
bool operator - (forcd f, forcd g) { // directly entails
    if (f.P.size() != g.P.size()-1)
        // f.P must be a direct subset of g.P
290     return false;
    bool b = true; // flag to check if there has been an element
                   // of g.P not present in f.P
    list<forc>::iterator gi = g.P.begin();
    for (list<forc>::iterator fi = f.P.begin(); fi != f.P.end(); fi++) {
        if (*fi == *gi)
            gi++;
        else if (b && *fi == *(++gi)) {
            // if flag b is true and the elements differ, change
            // flag to false
300         b = false;
            gi++;
        } else
            // in this case the flag b is false and more than one
            // element of g.P is not in f.P
            return false;
    }
    return true;
}

310 // A data structure that represents irreducible formulae of the
// fragment [and,or,double negation]^4
struct forcddn {
    forc Q; // The conjunction part
    forcd psi; // The double negated part

    // constructors
    forcddn() {}
    forcddn (forc f, forcd g) { Q = f; psi = g; }
    static forcddn bottom() {
```

```
320         // not strictly irreducible, but needed to build the
            // uptrie
            forcddn f;
            f.Q = *new forc();
            f.Q.Q = 0;
            f.psi = *new forcd( { forc::bottom() } );
            return f;
        };

        string print() {
330         string s = Q.print() + "&!![" + psi.print() + "]";
            return s;
        }
    };

    ostream& operator << (ostream& os, forcddn f) { return os << f.print(); }
    bool operator == (forcddn f, forcddn g) { return f.Q == g.Q && f.psi == g.psi;
        } // equivalence
    bool operator < (forcddn f, forcddn g) { return (f.Q < g.Q &&
        f.psi.augment(f.Q.Q ^ g.Q.Q) < g.psi); } // entailment
    bool operator - (forcddn f, forcddn g) { // direct entailment
        if (f.Q == g.Q && f.psi - g.psi) return true;
340     return (f.Q - g.Q && f.psi.augment(f.Q.Q ^ g.Q.Q) == g.psi);
    }

    // Creates a poset with the irreducible elements of the [and,or]
    // fragment
    poset<forc> irrCD(list<char> l) {
        list<poElem<forc>*> fs; // list of poElems containing the
                                // formulae
        forc f (l);
        // create each formula in [and]^4 with exception of
350     // contradiction
        for (int i=15; i>=0; i--)
            if ((f.Q | i) == f.Q) {
                forc g;
                g.Q = i;
                fs.push_back(new poElem<forc>(g));
            }
        for (list<poElem<forc>*>::iterator i=fs.begin(); i!=fs.end(); i++)
            for (list<poElem<forc>*>::iterator j=i; ++j!=fs.end();)
                if ((*i)->e - (*j)->e)
360                 // if i directly entails j, add j as child of i
                    (*i)->addChild(*j);
        poset<forc> p (fs); // make poset of the list of poElems
        return p;
    }
```

```
    // Needed for fragCD(), sorts a formula on the number of
    // disjunctions also implies that the list is sorted by
    // entailment
    struct sizeSort {
370     inline bool operator() (poElem<forcd>*& p1, poElem<forcd>*& p2) {
            return (p1->e.size() < p2->e.size());
        }
    };

    // Creates all formulae in the [and,or] fragments with atoms
    // specified by l
    poset<forcd> fragCD(list<char> l) {
        list<poElem<forcd>*> fs;

380     poset<forc> i = irrCD(l);
        uptrie<forc> u = generateUptrie(i); // generates the uptrie
        list<uptrie<forc>*> ul;

        // list all the uptries
        findUpsetsWithElements(u, *new list<forc>(), ul);
        for (uptrie<forc> *x : ul) {
            // convert each uptrie to a formula (forcd)
            static forcd f;
            f.P = x->Upset();
390         fs.push_front(new poElem<forcd>(f));
        }
        fs.sort(sizeSort()); // sorts the list of formulae

        // Creates the direct connections in the poset
        for (list<poElem<forcd>*>::iterator ip = fs.begin(); ip != fs.end(); ip++)
            for (list<poElem<forcd>*>::iterator jp = ip; ++jp != fs.end();)
                if ((**ip).e - (**jp).e)
                    (**jp).addChild(*ip);

400     poset<forcd> p (fs); // make poset of the list of poElems
        return p;
    }

    // Needed for irrCDDN(), if Q represents a group of atoms and l
    // represents all atoms in use, then this function returns a list
    // of atoms that are not in Q but are in l.
    list<char> complementaryAtoms(int Q, list<char> l) {
        list<char> c;
        if ((*find(l.begin(),l.end(),'p') == 'p') && (Q & 1) != 1)
        c.push_front('p');
410     if ((*find(l.begin(),l.end(),'q') == 'q') && (Q & 2) != 2)
        c.push_front('q');
```

```
    if ((*find(l.begin(),l.end(),'r') == 'r') && (Q & 4) != 4)
    c.push_front('r');
    if ((*find(l.begin(),l.end(),'s') == 's') && (Q & 8) != 8)
    c.push_front('s');
    return c;
}

// generates all the irreducible formulae of fragment
// [and,or,double negation] with atoms in l.
poset<forcddn> irrCDDN(list<char> l) {
    list<poElem<forcddn>*> fs;
    // An irreducible formula of this fragment consists of a           420
    // conjunction of atoms Q and a double negated formula of
    // [and,or] which we will call psi.

    // Generate all possibilities for Q
    poset<forc> Qs = irrCD(l);
    for (poElem<forc> *p : Qs.l) {
        // find the atoms that are not in Q
        list<char> c = complementaryAtoms(p->e.Q, l);

        // generate all possibilities for psi with this Q              430
        poset<forcd> psis = fragCD(c);

        // Remove first element (B) (id est contradiction) from
        // the poset
        psis.l.pop_front();

        // create a forcddn object with each Q and psi
        for (poElem<forcd> *q : psis.l) {
            forcddn f (p->e, q->e);
            fs.push_back(new poElem<forcddn>(f));                      440
        }
    }

    // create the direct connections in the poset
    for (poElem<forcddn> *p : fs)
        for (poElem<forcddn> *q : fs)
            if (p->e - q->e)
                p->addChild(q);

    poset<forcddn> p (fs); // make poset of the list of poElems       450
    return p;
}

int main(int argc, const char * argv[]) {
    clock_t start = clock();
```

```
    // 3 atoms:
    poset<forcddn> p = irrCDDN({'p','q','r'});

460 // 4 atoms:
    //poset<forcddn> p = irrCDDN({'p','q','r','s'});

    cout << "Irreducible formulae:\n" << p << "\nNumber of irreducible
    formulae: " << p.size() << "\n\n";
    uptrie<forcddn> u = generateUptrie(p);
    cout << "Total number of formulae in the fragment: " <<  u.size() <<
    "\n\n";

    cout << "Computation time: " << (clock() - start) / (double)
    CLOCKS_PER_SEC << " sec\n";
    return 0;
}
```

Bibliography

[1] Burris, S., and Sankappanavar, H. P. *A Course in Universal Algebra*. Springer-Verlag, 1981.

[2] Davey, B. A., and Priestley, H. A. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[3] De Bruijn, N. *Exact finite models for minimal propositional calculus over a finite alphabet*. Technological University Eindhoven, Report, 75-WSK-02, 1975.

[4] De Jongh, D., and Bezhanishvili, N. *Intuitionistic Logic*. University of Amsterdam: Institute for Logic, Language and Computation, 2010.

[5] De Jongh, D., Hendriks, L., and Renardel de Lavalette, G. R. Computations in fragments of intuitionistic propositional logic. *Journal of Automated Reasoning 7*, 4 (1991), 537–561.

[6] Hendriks, L. *Computations in Propositional Logic*. PhD thesis, University of Amsterdam: Institute for Logic, Langage and Computation, 1996.

[7] Kleitman, D., and Markowsky, G. On Dedekind's problem: The number of isotone boolean functions. II. *Trans. Amer. Math. Soc. 213* (1975), 373–390.

[8] Triesscheijn, R. A. Opwaarts gesloten deelverzamelingen in partieel geordende verzamelingen. Bachelor thesis, University of Groningen: Faculty of Mathematics & Natural Sciences, 2012.