Chapter 2

Propositional Logic

2.1 Syntax

Consider a finite, non-empty signature Σ of propositional variables, the "alphabet" of propositional logic. In addition to the alphabet "propositional connectives" are further building blocks composing the sentences (formulas) of the language and auxiliary symbols such as parentheses enable disambiguation.

Definition 2.1.1 (Propositional Formula)**.** The set $\mathrm{PROP}(\Sigma)$ of *propositional formulas* over a signature Σ is inductively defined by:

$\mathrm{PROP}(\Sigma)$	Comment
$\bot$	connective $\bot$ denotes "false"
$\top$	connective $\top$ denotes "true"
P	for any propositional variable $P \in \Sigma$
$(\neg\phi)$	connective $\neg$ denotes "negation"
$(\phi \land \psi)$	connective $\land$ denotes "conjunction"
$(\phi \lor \psi)$	connective $\lor$ denotes "disjunction"
$(\phi \rightarrow \psi)$	connective $\rightarrow$ denotes "implication"
$(\phi \leftrightarrow \psi)$	connective $\leftrightarrow$ denotes "equivalence"

where $\phi, \psi \in \mathrm{PROP}(\Sigma)$.

The above definition is an abbreviation for setting $\mathrm{PROP}(\Sigma)$ to be the language of a context free grammar $\mathrm{PROP}(\Sigma) = L((N, T, P, S))$ (see Definition 1.3.9) where $N = \{\phi, \psi\}$, $T = \Sigma \cup \{(,)\} \cup \{\bot, \top, \neg, \land, \lor, \rightarrow, \leftrightarrow\}$ with start symbol rules $S \Rightarrow \phi \mid \psi$, $\phi \Rightarrow \bot \mid \top \mid (\neg\phi) \mid (\phi\land\psi) \mid (\phi\lor\psi) \mid (\phi \rightarrow \psi) \mid (\phi \leftrightarrow \psi)$, $\psi \Rightarrow \bot \mid \top \mid (\neg\phi) \mid (\phi \land \psi) \mid (\phi \lor \psi) \mid (\phi \rightarrow \psi) \mid (\phi \leftrightarrow \psi)$, and $\phi \Rightarrow P$, $\psi \Rightarrow P$ for every $P \in \Sigma$.

As a notational convention we assume that $\neg$ binds strongest and we omit outermost parenthesis. So $\neg P \lor Q$ is actually a shorthand for $((\neg P) \lor Q)$. For all other logical connectives we will explicitly put parenthesis when needed. From the semantics we will see that $\land$ and $\lor$ are associative and commutative. Therefore instead of $((P \land Q) \land R)$ we simply write $P \land Q \land R$.

Definition 2.1.2 (Atom, Literal,Clause)**.** A propositional formula P is called an *atom*. It is also called a *(positive) literal* and its negation $\neg P$ is called a *(negative) literal*. If L is a literal, then $\neg L = P$ if $L = \neg P$ and $\neg L = \neg P$ if $L = P$, $|\neg P| = P$ and $|P| = P$. Literals are denoted by letters L, K. The literals P and $\neg P$ are called *complementary*. A disjunction of literals $L_1 \vee \ldots \vee L_n$ is called a *clause*.

Automated reasoning is very much formula manipulation. In order to precisely represent the manipulation of a formula, we introduce positions.

Definition 2.1.3 (Position)**.** A *position* is a word over $\mathbb{N}$. The set of positions of a formula ϕ is inductively defined by

$$\begin{array}{rcl} \mathrm{pos}(\phi) & := & \{\epsilon\} \text{ if } \phi \in \{\top, \bot\} \text{ or } \phi \in \Sigma \\ \mathrm{pos}(\neg\phi) & := & \{\epsilon\} \cup \{1p \mid p \in \mathrm{pos}(\phi)\} \\ \mathrm{pos}(\phi \circ \psi) & := & \{\epsilon\} \cup \{1p \mid p \in pos(\phi)\} \cup \{2p \mid p \in \mathrm{pos}(\psi)\} \end{array}$$

where $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$.

The prefix order $\leq$ on positions is defined by $p \leq q$ if there is some p' such that $pp' = q$. Note that the prefix order is partial, e.g., the positions 12 and 21 are not comparable, they are "parallel", see below. By $<$ we denote the strict part of $\leq$, i.e., $p < q$ if $p \leq q$ but not $q \leq p$. By $\parallel$ we denote incomparable positions, i.e., $p \parallel q$ if neither $p \leq q$, nor $q \leq p$. A position p is *above* q if $p \leq q$, p is *strictly above* q if $p < q$, and p and q are *parallel* if $p \parallel q$.

The *size* of a formula ϕ is given by the cardinality of $\mathrm{pos}(\phi)$: $|\phi| := |\,\mathrm{pos}(\phi)|$. The *subformula* of ϕ at position $p \in \mathrm{pos}(\phi)$ is recursively defined by $\phi|_\epsilon := \phi$, $\neg\phi|_{1p} := \phi|_p$, and $(\phi_1 \circ \phi_2)|_{ip} := \phi_i|_p$ where $i \in \{1, 2\}$, $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$. Finally, the *replacement* of a subformula at position $p \in \mathrm{pos}(\phi)$ by a formula ψ is recursively defined by $\phi[\psi]_\epsilon := \psi$ and $(\phi_1 \circ \phi_2)[\psi]_{1p} := (\phi_1[\psi]_p \circ \phi_2)$, $(\phi_1 \circ \phi_2)[\psi]_{2p} := (\phi_1 \circ \phi_2[\psi]_p)$, where $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$.

Example 2.1.4. The set of positions for the formula $\phi = (P \wedge Q) \rightarrow (P \vee Q)$ is $\mathrm{pos}(\phi) = \{\epsilon, 1, 11, 12, 2, 21, 22\}$. The subformula at position 22 is Q, $\phi|_{22} = Q$ and replacing this formula by $P \leftrightarrow Q$ results in $\phi[P \leftrightarrow Q]_{22} = (P \wedge Q) \rightarrow (P \vee (P \leftrightarrow Q))$.

A further prerequisite for efficient formula manipulation is the notion of the *polarity* of the subformula $\phi|_p$ of ϕ at position p. The polarity considers the number of "negations" starting from ϕ at ϵ down to p. It is 1 for an even number along the path, -1 for an odd number and 0 if there is at least one equivalence connective along the path.

Definition 2.1.5 (Polarity)**.** The *polarity* of the subformula $\phi|_p$ of ϕ at position $p \in \mathrm{pos}(\phi)$ is inductively defined by

$$\begin{array}{rcl} \mathrm{pol}(\phi, \epsilon) & := & 1 \\ \mathrm{pol}(\neg\phi, 1p) & := & -\mathrm{pol}(\phi, p) \\ \mathrm{pol}(\phi_1 \circ \phi_2, ip) & := & \mathrm{pol}(\phi_i, p) \text{ if } \circ \in \{\wedge, \vee\} \\ \mathrm{pol}(\phi_1 \rightarrow \phi_2, 1p) & := & -\mathrm{pol}(\phi_1, p) \\ \mathrm{pol}(\phi_1 \rightarrow \phi_2, 2p) & := & \mathrm{pol}(\phi_2, p) \\ \mathrm{pol}(\phi_1 \leftrightarrow \phi_2, ip) & := & 0 \end{array}$$

Example 2.1.6. We reuse the formula $\phi = (A \wedge B) \rightarrow (A \vee B)$ of Example 2.1.4. Then $\mathrm{pol}(\phi, 1) = \mathrm{pol}(\phi, 11) = -1$ and $\mathrm{pol}(\phi, 2) = \mathrm{pol}(\phi, 22) = 1$. For the formula $\phi' = (A \wedge B) \leftrightarrow (A \vee B)$ we get $\mathrm{pol}(\phi', \epsilon) = 1$ and $\mathrm{pol}(\phi', p) = 0$ for all other $p \in \mathrm{pos}(\phi')$, $p \neq \epsilon$.

2.2 Semantics

In *classical logic* there are two truth values "true" and "false" which we shall denote, respectively, by 1 and 0. There are *many-valued logics* [36] having more than two truth values and in fact, as we will see later on, for the definition of some propositional logic calculi, we will need an implicit third truth value called "undefined".

Definition 2.2.1 ((Partial) Valuation)**.** A Σ-*valuation* is a map

$$\mathcal{A} : \Sigma \rightarrow \{0, 1\}.$$

where $\{0, 1\}$ is the set of *truth values*. A *partial* Σ-*valuation* is a map $\mathcal{A}' : \Sigma' \rightarrow \{0, 1\}$ where $\Sigma' \subseteq \Sigma$.

Definition 2.2.2 (Semantics)**.** A Σ-valuation $\mathcal{A}$ is inductively extended from propositional variables to propositional formulas $\phi, \psi \in \mathrm{PROP}(\Sigma)$ by

$$\begin{array}{rcl}
\mathcal{A}(\bot) & := & 0 \\
\mathcal{A}(\top) & := & 1 \\
\mathcal{A}(\neg\phi) & := & 1 - \mathcal{A}(\phi) \\
\mathcal{A}(\phi \wedge \psi) & := & \min(\{\mathcal{A}(\phi), \mathcal{A}(\psi)\}) \\
\mathcal{A}(\phi \vee \psi) & := & \max(\{\mathcal{A}(\phi), \mathcal{A}(\psi)\}) \\
\mathcal{A}(\phi \rightarrow \psi) & := & \max(\{(1 - \mathcal{A}(\phi)), \mathcal{A}(\psi)\}) \\
\mathcal{A}(\phi \leftrightarrow \psi) & := & \text{if } \mathcal{A}(\phi) = \mathcal{A}(\psi) \text{ then } 1 \text{ else } 0
\end{array}$$

If $\mathcal{A}(\phi) = 1$ for some Σ-valuation $\mathcal{A}$ of a formula ϕ then ϕ is *satisfiable* and we write $\mathcal{A} \models \phi$. In this case $\mathcal{A}$ is a *model* of ϕ. If $\mathcal{A}(\phi) = 1$ for all Σ-valuations $\mathcal{A}$ of a formula ϕ then ϕ is *valid* and we write $\models \phi$. If there is no Σ-valuation $\mathcal{A}$ for a formula ϕ where $\mathcal{A}(\phi) = 1$ we say ϕ is *unsatisfiable*. A formula ϕ *entails* ψ, written $\phi \models \psi$, if for all Σ-valuations $\mathcal{A}$ whenever $\mathcal{A} \models \phi$ then $\mathcal{A} \models \psi$.

Accordingly, a formula ϕ is satisfiable, valid, unsatisfiable, respectively, with respect to a partial valuation $\mathcal{A}'$ with domain Σ', if for any valuation $\mathcal{A}$ with $\mathcal{A}(P) = \mathcal{A}'(P)$ for all $P \in \Sigma'$ the formula ϕ is satisfiable, valid, unsatisfiable, respectively, with respect to a $\mathcal{A}$.

I call the fact that some formula ϕ is satisfiable, unsatisfiable, or valid, the *status* of ϕ. Note that if ϕ is valid it is also satisfiable, but not the other way round.

Valuations can be nicely represented by sets or sequences of literals that do not contain complementary literals nor duplicates. If $\mathcal{A}$ is a (partial) valuation of domain Σ then it can be represented by the set $\{P \mid P \in \Sigma \text{ and } \mathcal{A}(P) = 1\} \cup \{\neg P \mid P \in \Sigma \text{ and } \mathcal{A}(P) = 0\}$. For example, for the valuation $\mathcal{A} = \{P, \neg Q\}$

the truth value of $P \lor Q$ is $\mathcal{A}(P \lor Q) = 1$, for $P \lor R$ it is $\mathcal{A}(P \lor R) = 1$, for $\neg P \land R$ it is $\mathcal{A}(\neg P \land R) = 0$, and the status of $\neg P \lor R$ cannot be established by $\mathcal{A}$. In particular, $\mathcal{A}$ is a partial valuation for $\Sigma = \{P, Q, R\}$.

Example 2.2.3. The formula $\phi \lor \neg\phi$ is valid, independently of ϕ. According to Definition 2.2.2 we need to prove that for all Σ-valuations $\mathcal{A}$ of ϕ we have $\mathcal{A}(\phi \lor \neg\phi) = 1$. So let $\mathcal{A}$ be an arbitrary valuation. There are two cases to consider. If $\mathcal{A}(\phi) = 1$ then $\mathcal{A}(\phi \lor \neg\phi) = 1$ because the valuation function takes the maximum if distributed over $\lor$. If $\mathcal{A}(\phi) = 0$ then $\mathcal{A}(\neg\phi) = 1$ and again by the before argument $\mathcal{A}(\phi \lor \neg\phi) = 1$. This finishes the proof that $\models \phi \lor \neg\phi$.

Proposition 2.2.4 (Deduction Theorem). $\phi \models \psi$ iff $\models \phi \rightarrow \psi$

Proof. ($\Rightarrow$) Suppose that ϕ entails ψ and let $\mathcal{A}$ be an arbitrary Σ-valuation. We need to show $\mathcal{A} \models \phi \rightarrow \psi$. If $\mathcal{A}(\phi) = 1$, then $\mathcal{A}(\psi) = 1$, because ϕ entails ψ, and therefore $\mathcal{A} \models \phi \rightarrow \psi$. For otherwise, if $\mathcal{A}(\phi) = 0$, then $\mathcal{A}(\phi \rightarrow \psi) = \max(\{(1 - \mathcal{A}(\phi)), \mathcal{A}(\psi)\}) = \max(\{(1, \mathcal{A}(\psi)\}) = 1$, independently of the value of $\mathcal{A}(\psi)$. In both cases $\mathcal{A} \models \phi \rightarrow \psi$.
($\Leftarrow$) By contraposition. Suppose that ϕ does not entail ψ. Then there exists a Σ-valuation $\mathcal{A}$ such that $\mathcal{A} \models \phi$, $\mathcal{A}(\phi) = 1$ but $\mathcal{A} \not\models \psi$, $\mathcal{A}(\psi) = 0$. By definition, $\mathcal{A}(\phi \rightarrow \psi) = \max(\{(1 - \mathcal{A}(\phi)), \mathcal{A}(\psi)\}) = \max(\{(1 - 1), 0\}) = 0$, hence $\phi \rightarrow \psi$ does not hold in $\mathcal{A}$. ☐

Proposition 2.2.5. The equivalences of Figure 2.1 are valid for all formulas ϕ, ψ, χ.

From Figure 2.1 we conclude that the propositional language introduced in Definition 2.1.1 is redundant in the sense that certain connectives can be expressed by others. For example, the equivalence Eliminate $\rightarrow$ expresses implication by means of disjunction and negation. So for any propositional formula ϕ there exists an equivalent formula ϕ' such that ϕ' does not contain the implication connective. In order to prove this proposition we need the below replacement lemma.

T

Note that the formulas $\phi \land \psi$ and $\psi \land \phi$ are equivalent. Nevertheless, recalling the problem state definition for Sudokus in Section 1.1 the two states $(N; f(2,3) = 1 \land f(2,4) = 4; \top)$ and $(N; f(2,4) = 4 \land f(2,3) = 1; \top)$ are significantly different. For example, it can be that the first state can lead to a solution by the rules of the algorithm where the latter cannot, because the latter implicitly means that the square $(2,4)$ has already been checked for all values smaller than 4. This reveals the important point that arguing by logical equivalence in the context of a rule set manipulating formulas can lead to wrong results.

Lemma 2.2.6 (Formula Replacement). Let ϕ be a propositional formula containing a subformula ψ at position p, i.e., $\phi|_p = \psi$. Furthermore, assume $\models \psi \leftrightarrow \chi$. Then $\models \phi \leftrightarrow \phi[\chi]_p$.

(I)	$(\phi \wedge \phi) \leftrightarrow \phi$	Idempotency $\wedge$
	$(\phi \vee \phi) \leftrightarrow \phi$	Idempotency $\vee$
(II)	$(\phi \wedge \psi) \leftrightarrow (\psi \wedge \phi)$	Commutativity $\wedge$
	$(\phi \vee \psi) \leftrightarrow (\psi \vee \phi)$	Commutativity $\vee$
(III)	$(\phi \wedge (\psi \wedge \chi)) \leftrightarrow ((\phi \wedge \psi) \wedge \chi)$	Associativity $\wedge$
	$(\phi \vee (\psi \vee \chi)) \leftrightarrow ((\phi \vee \psi) \vee \chi)$	Associativity $\vee$
(IV)	$(\phi \wedge (\psi \vee \chi)) \leftrightarrow (\phi \wedge \psi) \vee (\phi \wedge \chi)$	Distributivity $\wedge\vee$
	$(\phi \vee (\psi \wedge \chi)) \leftrightarrow (\phi \vee \psi) \wedge (\phi \vee \chi)$	Distributivity $\vee\wedge$
(V)	$(\phi \wedge (\phi \vee \psi)) \leftrightarrow \phi$	Absorption $\wedge\vee$
	$(\phi \vee (\phi \wedge \psi)) \leftrightarrow \phi$	Absorption $\vee\wedge$
(VI)	$\neg(\phi \vee \psi) \leftrightarrow (\neg\phi \wedge \neg\psi)$	De Morgan $\neg\vee$
	$\neg(\phi \wedge \psi) \leftrightarrow (\neg\phi \vee \neg\psi)$	De Morgan $\neg\wedge$
(VII)	$(\phi \wedge \neg\phi) \leftrightarrow \bot$	Introduction $\bot$
	$(\phi \vee \neg\phi) \leftrightarrow \top$	Introduction $\top$
	$\neg\top \leftrightarrow \bot$	Propagate $\neg\top$
	$\neg\bot \leftrightarrow \top$	Propagate $\neg\bot$
	$(\phi \wedge \top) \leftrightarrow \phi$	Absorption $\top\wedge$
	$(\phi \vee \bot) \leftrightarrow \phi$	Absorption $\bot\vee$
	$(\neg\neg\phi) \leftrightarrow \phi$	Absorption $\neg\neg$
	$(\phi \rightarrow \bot) \leftrightarrow \neg\phi$	Eliminate $\rightarrow \bot$
	$(\bot \rightarrow \phi) \leftrightarrow \top$	Eliminate $\bot \rightarrow$
	$(\phi \rightarrow \top) \leftrightarrow \top$	Eliminate $\rightarrow \top$
	$(\top \rightarrow \phi) \leftrightarrow \phi$	Eliminate $\top \rightarrow$
	$(\phi \leftrightarrow \bot) \leftrightarrow \neg\phi$	Eliminate $\bot \leftrightarrow$
	$(\phi \leftrightarrow \top) \leftrightarrow \phi$	Eliminate $\top \leftrightarrow$
	$(\phi \vee \top) \leftrightarrow \top$	Propagate $\top$
	$(\phi \wedge \bot) \leftrightarrow \bot$	Propagate $\bot$
(VIII)	$(\phi \rightarrow \psi) \leftrightarrow (\neg\phi \vee \psi)$	Eliminate $\rightarrow$
(IX)	$(\phi \leftrightarrow \psi) \leftrightarrow (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$	Eliminate1 $\leftrightarrow$
	$(\phi \leftrightarrow \psi) \leftrightarrow (\phi \wedge \psi) \vee (\neg\phi \wedge \neg\psi)$	Eliminate2 $\leftrightarrow$

Figure 2.1: Valid Propositional Equivalences

Proof. By induction on $|p|$ and structural induction on ϕ. For the base step let $p = \epsilon$ and $\mathcal{A}$ be an arbitrary valuation.

$$\begin{aligned}
\mathcal{A}(\phi) &= \mathcal{A}(\psi) && \text{(by definition of replacement)}\\
&= \mathcal{A}(\chi) && \text{(because } \mathcal{A} \models \psi \leftrightarrow \chi\text{)}\\
&= \mathcal{A}(\phi[\chi]_\epsilon) && \text{(by definition of replacement)}
\end{aligned}$$

For the induction step the lemma holds for all positions p and has to be shown for all positions ip. By structural induction on ϕ, I show the cases where $\phi = \neg\phi_1$ and $\phi = \phi_1 \rightarrow \phi_2$ in detail. All other cases are analogous.

If $\phi = \neg\phi_1$ then showing the lemma amounts to proving $\models \neg\phi_1 \leftrightarrow \neg\phi_1[\chi]_{1p}$. Let $\mathcal{A}$ be an arbitrary valuation.

$$\begin{aligned}
\mathcal{A}(\neg\phi_1) &= 1 - \mathcal{A}(\phi_1) && \text{(expanding semantics)}\\
&= 1 - \mathcal{A}(\phi_1[\chi]_p) && \text{(by induction hypothesis)}\\
&= \mathcal{A}(\neg\phi[\chi]_{1p}) && \text{(applying semantics)}
\end{aligned}$$

If $\phi = \phi_1 \rightarrow \phi_2$ then showing the lemma amounts to proving the two cases $\models (\phi_1 \rightarrow \phi_2) \leftrightarrow (\phi_1 \rightarrow \phi_2)[\chi]_{1p}$ and $\models (\phi_1 \rightarrow \phi_2) \leftrightarrow (\phi_1 \rightarrow \phi_2)[\chi]_{2p}$. Both cases are similar so I show only the first case. Let $\mathcal{A}$ be an arbitrary valuation.

$$\begin{aligned}
\mathcal{A}(\phi_1 \rightarrow \phi_2) &= \max(\{(1 - \mathcal{A}(\phi_1)), \mathcal{A}(\phi_2)\}) && \text{(expanding semantics)}\\
&= \max(\{(1 - \mathcal{A}(\phi_1[\chi]_p)), \mathcal{A}(\phi_2)\}) && \text{(by induction hypothesis)}\\
&= \mathcal{A}((\phi_1 \rightarrow \phi_2)[\chi]_{1p}) && \text{(applying semantics)}
\end{aligned}$$

□

Lemma 2.2.7 (Polarity Dependent Replacement)**.** Consider a formula ϕ, position $p \in \text{pos}(\phi)$, $\text{pol}(\phi, p) = 1$ and (partial) valuation $\mathcal{A}$ with $\mathcal{A}(\phi) = 1$. If for some formula ψ, $\mathcal{A}(\psi) = 1$ then $\mathcal{A}(\phi[\psi]_p) = 1$. Symmetrically, if $\text{pol}(\phi, p) = -1$ and $\mathcal{A}(\psi) = 0$ then $\mathcal{A}(\phi[\psi]_p) = 1$.

Proof. By induction on the length of p. □

Note that the case for the above lemma where $\text{pol}(\phi, p) = 0$ is actually Lemma 2.2.6.

C

The equivalences of Figure 2.1 show that the propositional language introduced in Definition 2.1.1 is redundant in the sense that certain connectives can be expressed by others. For example, the equivalence Eliminate $\rightarrow$ expresses implication by means of disjunction and negation. So for any propositional formula ϕ there exists an equivalent formula ϕ' such that ϕ' does not contain the implication connective. In order to prove this proposition the above replacement lemma is key.

2.3 Abstract Properties of Calculi

A proof procedure can be *sound*, *complete*, *strongly complete*, *refutationally complete* or *terminating*. Terminating means that it terminates on any input formula. Now depending on whether the calculus investigates validity (unsatisfiability) or satisfiability the aforementioned notions have (slightly) different meanings.

	Validity	Satisfiability
Sound	If the calculus derives a proof of validity for the formula, it is valid.	If the calculus derives satisfiability of the formula, it has a model.
Complete	If the formula is valid, a proof of validity is derivable by the calculus.	If the formula has a model, the calculus derives satisfiability.
Strongly Complete	For any proof of the formula, there is a derivation in the calculus producing this proof.	For any model of the formula, there is a derivation in the calculus producing this model.

There are some assumptions underlying these informal definitions. First, the calculus actually produces a proof in case of investigating validity, and in case of investigating satisfiability it produces a model. This in fact requires the notion of a proof and a model. Then soundness means in both cases that the calculus has no bugs. The results it produces are correct. Completeness means that if there is a proof (model) for a formula, the calculus could eventually find it. Strong completeness requires in addition that any proof (model) can be found by the calculus. A variant of complete calculus is a *refutationally complete* calculus: a calculus is refutationally complete, if for any unsatisfiable formula it derives a proof of contradiction. Many automated theorem procedures like resolution (see Section 2.7), or tableaux (see Section 2.5) are actually only refutationally complete.

2.4 Truth Tables

The first calculus I consider are truth tables. For example, consider proving validity of the formula $\phi = (A \wedge B) \rightarrow A$. According to Definition 2.2.2 this is the case when actually for all valuations $\mathcal{A}$ over $\Sigma = \{A, B\}$ we have $\mathcal{A}(\phi) = 1$. The extension of $\mathcal{A}$ to formulas is defined inductively over the connectives, so if the result of $\mathcal{A}$ on the arguments of a connective is known, it can be straightforwardly computed for the overall formula. That's the idea behind truth tables. We simply make all valuations $\mathcal{A}$ on Σ explicit and then extend it connective by connective bottom-up to the overall formula. Stated differently, in order to establish the truth value for a formula ϕ we establish it subformula by subformula

of ϕ according to $\leq$. If $p, q \in \text{pos}(\phi)$ and $p \leq q$ then we first compute the truth value for $\phi|_q$. The truth table for $(P \wedge Q) \rightarrow P$ is then depicted in Figure 2.2

P	Q	$P \wedge Q$	$(P \wedge Q) \rightarrow P$
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	1

Figure 2.2: Truth Table for $(P \wedge Q) \rightarrow P$

Definition 2.4.1 (Truth Table)**.** Let ϕ be a propositional formula over variables $P_1, \ldots, P_n$, $p_i \in \text{pos}(\phi)$, $1 \leq i \leq k$ and $p_k = \epsilon$. Then a *truth table* for ϕ is a table with $n + k$ columns and $2^n + 1$ rows of the form

P_1	$\ldots$	P_n	$\phi\|_{p_1}$	$\ldots$	$\phi\|_{p_k}$
0	$\ldots$	0	$\mathcal{A}_1(\phi\|_{p_1})$	$\ldots$	$\mathcal{A}_1(\phi\|_{p_k})$
			$\vdots$		
1	$\ldots$	1	$\mathcal{A}_{2^n}(\phi\|_{p_1})$	$\ldots$	$\mathcal{A}_{2^n}(\phi\|_{p_k})$

such that the $\mathcal{A}_i$ are exactly the 2^n different valuations for $P_1, \ldots, P_n$ and either $p_i \parallel p_{i+j}$ or $p_i \geq p_{i+j}$, for all $i, j \geq 0$, $i + j \leq k$ and whenever $\phi|_{p_i}$ has a proper subformula ψ that is not an atom, there is exactly one $j < i$ with $\phi|_{p_j} = \psi$.

Now given a truth table for some formula ϕ, ϕ is satisfiable, if there is at least one 1 in the ϕ column. It is valid, if there is no 0 in the ϕ column. It is unsatisfiable, if there is no 1 in the ϕ column. So truth tables are a simple and "easy" way to establish the status of a formula. They need not to be completely computed in order to establish the status of a formula. For example, as soon as the column of ϕ in a truth table contains a 1 and a 0, then ϕ is satisfiable but neither valid nor unsatisfiable.

The formula $(P \vee Q) \leftrightarrow (P \vee R)$ is satisfiable but not valid. Figure 2.3 contains a truth table for the formula.

P	Q	R	$P \vee Q$	$P \vee R$	$(P \vee Q) \leftrightarrow (P \vee R)$
0	0	0	0	0	1
0	1	0	1	0	0
1	0	0	1	1	1
1	1	0	1	1	1
0	0	1	0	1	0
0	1	1	1	1	1
1	0	1	1	1	1
1	1	1	1	1	1

Figure 2.3: Truth Table for $(P \vee Q) \leftrightarrow (P \vee R)$

Of course, there are cases where a truth table for some formula ϕ can have less columns than the number of variables occurring in ϕ plus the number of subformulas in ϕ. For example, for the formula $\phi = (P \vee Q) \wedge (R \rightarrow (P \vee Q))$ only one column with formula $(P \vee Q)$ is needed for both subformulas $\phi|_1$ and $\phi|_{22}$. In general, a single column is needed for each *different* subformula. Detecting subformula equivalence is beneficial. For the above example, this was simply syntactic, i.e., the two subformulas $\phi|_1$ and $\phi|_{22}$. But what about a slight variation of the formula $\phi' = (P \vee Q) \wedge (R \rightarrow (Q \vee P))$? Strictly speaking, now the two subformulas $\phi'|_1$ and $\phi'|_{22}$ are different, but since disjunction is commutative, they are equivalent. One or two columns in the truth table for the two subformulas? Again, saving a column is beneficial but in general, detecting equivalence of two subformulas may become as difficult as checking whether the overall formula is valid. A compromise, often performed in practice, are normal forms that guarantee that certain occurrences of equivalent subformulas can be found in polynomial time. For the running example, we can simply assume some ordering on the propositional variables and assume that for a disjunction of two propositional variables, the smaller variable always comes first. So if $P < Q$ then the normal form of $P \vee Q$ and $Q \vee P$ is in fact $P \vee Q$.

C

In practice, nobody uses truth tables as a reasoning procedure. Worst case, computing a truth table for checking the status of a formula ϕ requires $O(2^n)$ steps, where n is the number of different propositional variables in ϕ. But this is actually not the reason why the procedure is impractical, because the worst case behavior of all other procedures for propositional logic known today is also of exponential complexity. So why are truth tables not a good procedure? The answer is: because they do not adapt to the inherent structure of a formula. The reasoning mechanism of a truth table for two formulas ϕ and ψ sharing the same propositional variables is exactly the same: we enumerate all valuations. However, if ϕ is, e.g., of the form $\phi = P \wedge \phi'$ and we are interested in the satisfiability of ϕ, then ϕ can only become true for a valuation $\mathcal{A}$ with $\mathcal{A}(P) = 1$. Hence, 2^{n-1} rows of ϕ's truth table are superfluous. All procedures I will introduce in the sequel, automatically detect this (and further) specific structures of a formula and use it to speed up the reasoning process.

2.5 Propositional Tableaux

Like resolution, semantic tableaux were developed in the sixties, independently by Lis [25] and Smullyan [34] on the basis of work by Gentzen in the 30s [18] and of Beth [8] in the 50s. For an at that time state of the art overview consider Fitting's book [16].

In contrast to the calculi introduced in subsequent sections, semantic tableau does not rely on a normal form of input formulas but actually applies to any propositional formula. The formulas are divided into α- and β-formulas, where intuitively an α formula represents a (hidden) conjunction and a β formula a

α	Left Descendant	Right Descendant
$\neg\neg\phi$	ϕ	ϕ
$\phi_1 \wedge \phi_2$	ϕ_1	ϕ_2
$\phi_1 \leftrightarrow \phi_2$	$\phi_1 \rightarrow \phi_2$	$\phi_2 \rightarrow \phi_1$
$\neg(\phi_1 \vee \phi_2)$	$\neg\phi_1$	$\neg\phi_2$
$\neg(\phi_1 \rightarrow \phi_2)$	ϕ_1	$\neg\phi_2$

β	Left Descendant	Right Descendant
$\phi_1 \vee \phi_2$	ϕ_1	ϕ_2
$\phi_1 \rightarrow \phi_2$	$\neg\phi_1$	ϕ_2
$\neg(\phi_1 \wedge \phi_2)$	$\neg\phi_1$	$\neg\phi_2$
$\neg(\phi_1 \leftrightarrow \phi_2)$	$\neg(\phi_1 \rightarrow \phi_2)$	$\neg(\phi_2 \rightarrow \phi_1)$

Figure 2.4: α- and β-Formulas

(hidden) disjunction.

Definition 2.5.1 (α-, β-Formulas)**.** A formula ϕ is called an α-formula if ϕ is a formula $\neg\neg\phi_1$, $\phi_1 \wedge \phi_2$, $\phi_1 \leftrightarrow \phi_2$, $\neg(\phi_1 \vee \phi_2)$, or $\neg(\phi_1 \rightarrow \phi_2)$. A formula ϕ is called an β-formula if ϕ is a formula $\phi_1 \vee \phi_2$, $\phi_1 \rightarrow \phi_2$, $\neg(\phi_1 \wedge \phi_2)$, or $\neg(\phi_1 \leftrightarrow \phi_2)$.

A common property of α-, β-formulas is that they can be decomposed into direct descendants representing (modulo negation) subformulas of the respective formulas. Then an α-formula is valid iff all its descendants are valid and a β-formula is valid iff one of its descendants is valid. Therefore, the literature uses both the notions semantic tableaux and analytic tableaux.

Definition 2.5.2 (Direct Descendant)**.** Given an α- or β-formula ϕ, Figure 2.4 shows its direct descendants.

Duplicating ϕ for the α-descendants of $\neg\neg\phi$ is a trick for conformity. Any propositional formula is either an α-formula or a β-formula or a literal.

Proposition 2.5.3. For any valuation $\mathcal{A}$: (i) if ϕ is an α-formula then $\mathcal{A}(\phi) = 1$ iff $\mathcal{A}(\phi_1) = 1$ and $\mathcal{A}(\phi_2) = 1$ for its descendants ϕ_1, ϕ_2. (ii) if ϕ is a β-formula then $\mathcal{A}(\phi) = 1$ iff $\mathcal{A}(\phi_1) = 1$ or $\mathcal{A}(\phi_2) = 1$ for its descendants ϕ_1, ϕ_2.

The tableaux calculus operates on states that are sets of sequences of formulas. Semantically, the set represents a disjunction of sequences that are interpreted as conjunctions of the respective formulas. A sequence of formulas $(\phi_1, \ldots, \phi_n)$ is called *closed* if there are two formulas ϕ_i and ϕ_j in the sequence where $\phi_i = \neg\phi_j$ or $\neg\phi_i = \phi_j$. A state is *closed* if all its formula sequences are closed. A state actually represents a tree and this tree is called a tableau in the literature. So if a state is closed, the respective tree, the tableau is closed too. The tableaux calculus is a calculus showing unsatisfiability. Such calculi are called *refutational* calculi. Later on soundness and completeness of the calculus

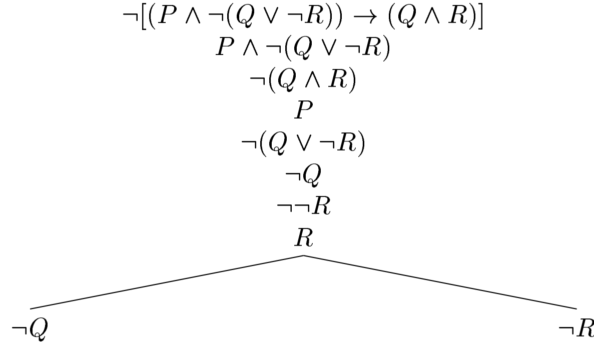


Figure 2.5: A Tableau for $(P \land \neg(Q \lor \neg R)) \to (Q \land R)$

imply that a formula ϕ is valid iff the rules of tableaux produce a closed state starting with $N = \{(\neg\phi)\}$.

A formula ϕ occurring in some sequence is called *open* if in case ϕ is an α-formula not both direct descendants are already part of the sequence and if it is a β-formula none of its descendants is part of the sequence.

α-Expansion $\quad N \uplus \{(\phi_1, \ldots, \psi, \ldots, \phi_n)\} \Rightarrow_{\mathrm{T}} N \uplus \{(\phi_1, \ldots, \psi, \ldots, \phi_n, \psi_1, \psi_2)\}$
provided ψ is an open α-formula, ψ_1, ψ_2 its direct descendants and the sequence is not closed.

β-Expansion $\quad N \uplus \{(\phi_1, \ldots, \psi, \ldots, \phi_n)\} \Rightarrow_{\mathrm{T}} N \uplus \{(\phi_1, \ldots, \psi, \ldots, \phi_n, \psi_1)\} \uplus \{(\phi_1, \ldots, \psi, \ldots, \phi_n, \psi_2)\}$
provided ψ is an open β-formula, ψ_1, ψ_2 its direct descendants and the sequence is not closed.

Consider the question of validity of the formula $(P \land \neg(Q \lor \neg R)) \to (Q \land R)$. Applying the tableau rules generates the following derivation:

$$\begin{array}{l}
\{(\neg[(P \land \neg(Q \lor \neg R)) \to (Q \land R)])\} \\
\quad \alpha\text{-Expansion} \Rightarrow_{\mathrm{T}}^{*} \{(\neg[(P \land \neg(Q \lor \neg R)) \to (Q \land R)], \\
\qquad P \land \neg(Q \lor \neg R), \neg(Q \land R), P, \neg(Q \lor \neg R), \neg Q, \neg\neg R, R)\} \\
\quad \beta\text{-Expansion} \Rightarrow_{\mathrm{T}} \{(\neg[(P \land \neg(Q \lor \neg R)) \to (Q \land R)], \\
\qquad P \land \neg(Q \lor \neg R), \neg(Q \land R), P, \neg(Q \lor \neg R), \neg Q, \neg\neg R, R, \neg Q), \\
\qquad (\neg[(P \land \neg(Q \lor \neg R)) \to (Q \land R)], \\
\qquad P \land \neg(Q \lor \neg R), \neg(Q \land R), P, \neg(Q \lor \neg R), \neg Q, \neg\neg R, R, \neg R)\}
\end{array}$$

The state after β-expansion is final, i.e., no more rule can be applied. The first sequence is not closed, whereas the second sequence is because it contains R and $\neg R$. A tree representation, where common formulas of sequences are shared, can be seen in Figure 2.5.

Theorem 2.5.4 (Propositional Tableaux is Sound)**.** If for a formula ϕ the tableaux calculus computes $\{(\neg\phi)\} \Rightarrow^*_{\mathrm{T}} N$ and N is closed, then ϕ is valid.

Proof. It is sufficient to show the following: (i) if N is closed then the disjunction of the conjunction of all sequence formulas is unsatisfiable (ii) the two tableaux rules preserve satisfiability.

Part (i) is obvious: if N is closed all its sequences are closed. A sequence is closed if it contains a formula and its negation. The conjunction of two such formulas is unsatisfiable.

Part (ii) is shown by induction on the length of the derivation and then by a case analysis for the two rules. α-Expansion: for any valuation $\mathcal{A}$ if $\mathcal{A}(\psi) = 1$ then $\mathcal{A}(\psi_1) = \mathcal{A}(\psi_2) = 1$. β-Expansion: for any valuation $\mathcal{A}$ if $\mathcal{A}(\psi) = 1$ then $\mathcal{A}(\psi_1) = 1$ or $\mathcal{A}(\psi_2) = 1$ (see Proposition 2.5.3). ◻

Theorem 2.5.5 (Propositional Tableaux Terminates)**.** Starting from a start state $\{(\phi)\}$ for some formula ϕ, $\Rightarrow^+_{\mathrm{T}}$ is well-founded.

Proof. Take the two-folded multi-set extension of the lexicographic extension of $>$ on the naturals to triples (n, k, l). The measure μ is first defined on formulas by $\mu(\phi) := (n, k, l)$ where n is the number of equivalence symbols in ϕ, k is the sum of all disjunction, conjunction, implication symbols in ϕ and l is $|\phi|$. On sequences $(\phi_1, \ldots, \phi_n)$ the measure is defined to deliver a multiset by $\mu((\phi_1, \ldots, \phi_n)) := \{t_1, \ldots, t_n\}$ where $t_i = \mu(\phi_i)$ if ϕ is open in the sequence and $t_i = (0, 0, 0)$ otherwise. Finally, μ is extended to states by computing the multiset $\mu(N) := \{\mu(s) \mid s \in N\}$.

Note, that α-, as well as β-expansion strictly extend sequences. Once a formula is closed in a sequence by applying an expansion rule, it remains closed forever in the sequence.

An α-expansion on a formula $\psi_1 \wedge \psi_2$ on the sequence $(\phi_1, \ldots, \psi_1 \wedge \psi_2, \ldots, \phi_n)$ results in $(\phi_1, \ldots, \psi_1 \wedge \psi_2, \ldots, \phi_n, \psi_1, \psi_2)$. It needs to be shown $\mu((\phi_1, \ldots, \psi_1 \wedge \psi_2, \ldots, \phi_n)) >_{\mathrm{mul}} \mu((\phi_1, \ldots, \psi_1 \wedge \psi_2, \ldots, \phi_n, \psi_1, \psi_2))$. In the second sequence $\mu(\psi_1 \wedge \psi_2) = (0, 0, 0)$ because the formula is closed. For the triple (n, k, l) assigned by μ to $\psi_1 \wedge \psi_2$ in the first sequence, it holds $(n, k, l) >_{\mathrm{lex}} \mu(\psi_1)$, $(n, k, l) >_{\mathrm{lex}} \mu(\psi_2)$ and $(n, k, l) >_{\mathrm{lex}} (0, 0, 0)$, the former because the ψ_i are subformulas and the latter because $l \neq 0$. This proves the case.

A β-expansion on a formula $\psi_1 \vee \psi_2$ on the sequence $(\phi_1, \ldots, \psi_1 \vee \psi_2, \ldots, \phi_n)$ results in $(\phi_1, \ldots, \psi_1 \vee \psi_2, \ldots, \phi_n, \psi_1)$, $(\phi_1, \ldots, \psi_1 \vee \psi_2, \ldots, \phi_n, \psi_2)$. It needs to be shown $\mu((\phi_1, \ldots, \psi_1 \vee \psi_2, \ldots, \phi_n)) >_{\mathrm{mul}} \mu((\phi_1, \ldots, \psi_1 \vee \psi_2, \ldots, \phi_n, \psi_1))$ and $\mu((\phi_1, \ldots, \psi_1 \vee \psi_2, \ldots, \phi_n)) >_{\mathrm{mul}} \mu((\phi_1, \ldots, \psi_1 \vee \psi_2, \ldots, \phi_n, \psi_2))$. In the derived sequences $\mu(\psi_1 \vee \psi_2) = (0, 0, 0)$ because the formula is closed. For the triple (n, k, l) assigned by μ to $\psi_1 \vee \psi_2$ in the starting sequence, it holds $(n, k, l) >_{\mathrm{lex}} \mu(\psi_1)$, $(n, k, l) >_{\mathrm{lex}} \mu(\psi_2)$ and $(n, k, l) >_{\mathrm{lex}} (0, 0, 0)$, the former because the ψ_i are subformulas and the latter because $l \neq 0$. This proves the case. ◻

Theorem 2.5.6 (Propositional Tableaux is Complete)**.** If ϕ is valid, tableaux computes a closed state out of $\{(\neg\phi)\}$.

Proof. If ϕ is valid then $\neg\phi$ is unsatisfiable. Now assume after termination the resulting state and hence at least one sequence is not closed. For this sequence consider a valuation $\mathcal{A}$ consisting of the literals in the sequence. By assumption there are no opposite literals, so $\mathcal{A}$ is well-defined. I prove by contradiction that $\mathcal{A}$ is a model for the sequence. Assume it is not. Then there is a minimal formula in the sequence, with respect to the ordering on triples considered in the proof of Theorem 2.5.5, that is not satisfied by $\mathcal{A}$. By definition of $\mathcal{A}$ the formula cannot be a literal. So it is an α-formula or a β-formula. In all cases at least one descendant formula is contained in the sequence, is smaller than the original formula, false in $\mathcal{A}$ (Proposition 2.5.3) and hence contradicts the assumption. Therefore, $\mathcal{A}$ satisfies the sequence contradicting that $\neg\phi$ is unsatisfiable. □

Corollary 2.5.7 (Propositional Tableaux generates Models)**.** Let ϕ be a formula, $\{(\phi)\} \Rightarrow^*_{\mathrm{T}} N$ and $s \in N$ be a sequence that is not closed and neither α-expansion nor β-expansion are applicable to s. Then the literals in s form a (partial) valuation that is a model for ϕ.

Proof. A consequence of the proof of Theorem 2.5.6 □

Consider the example tableau shown in Figure 2.5. The open first branch corresponds to the valuation $\mathcal{A} = \{P, R, \neg Q\}$ which is a model of the formula $\neg[(P \wedge \neg(Q \vee \neg R)) \rightarrow (Q \wedge R)]$.

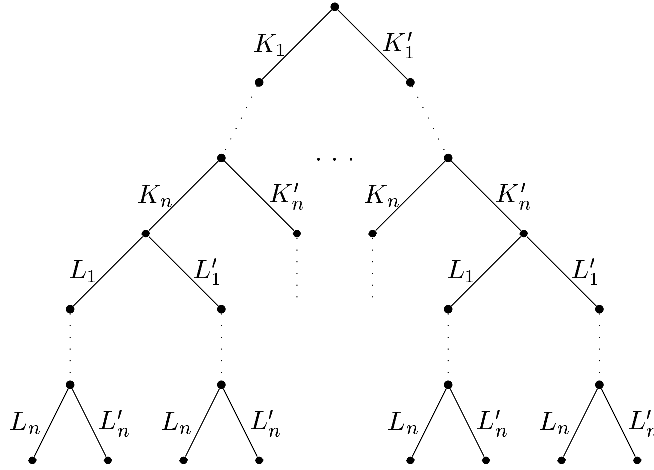


Figure 2.6: Semantic tableau.

2.6 Normal Forms

In order to check the status of a formula ϕ via truth tables, the truth table contains a column for the subformulas of ϕ and all valuations for its variables.

Any shape of ϕ is fine in order to generate the respective truth table. The superposition calculus (Section 2.8) and the CDCL (Conflict Driven Clause Learning) calculus (Section 2.10) both operate on a normal form, i.e., the shape of ϕ is restricted. Both calculi accept only conjunctions of disjunctions of literals, a particular *normal form*. It is called *Clause Normal Form* or simply *CNF*. The purpose of this section is to show that an arbitrary formula ϕ can be effectively transformed into an equivalent formula in CNF.

2.6.1 Conjunctive and Disjunctive Normal Forms

Definition 2.6.1 (CNF, DNF). A formula is in *conjunctive normal form (CNF)* or *clause normal form* if it is a conjunction of disjunctions of literals, or in other words, a conjunction of clauses.

A formula is in *disjunctive normal form (DNF)*, if it is a disjunction of conjunctions of literals.

So a CNF has the form $\bigwedge_i \bigvee_j L_j$ and a DNF the form $\bigvee_i \bigwedge_j L_j$ where L_j are literals. In the sequel the logical notation with $\lor$ is overloaded with a multiset notation. Both the disjunction $L_1 \lor \ldots \lor L_n$ and the multiset $\{L_1, \ldots, L_n\}$ are clauses. For clauses the letters C, D, possibly indexed are used. Furthermore, a conjunction of clauses is considered as a set of clauses. Then, for a set of clauses, the empty set denotes $\top$. For a clause, the empty multiset denotes $\emptyset$ and at the same time $\bot$.

T Although CNF and DNF are defined in almost any text book on automated reasoning, the definitions in the literature differ with respect to the "border" cases: (i) are complementary literals permitted in a clause? (ii) are duplicated literals permitted in a clause? (iii) are empty disjunctions/conjunctions permitted? The above Definition 2.6.1 answers all three questions with "yes". A clause containing complementary literals is valid, as in $P \lor Q \lor \neg P$. Duplicate literals may occur, as in $P \lor Q \lor P$. The empty disjunction is $\bot$ and the empty conjunction $\top$, i.e., the empty disjunction is always false while the empty conjunction is always true.

Checking the validity of CNF formulas or the unsatisfiability of DNF formulas is easy: (i) a formula in CNF is valid, if and only if each of its disjunctions contains a pair of complementary literals P and $\neg P$, (ii) conversely, a formula in DNF is unsatisfiable, if and only if each of its conjunctions contains a pair of complementary literals P and $\neg P$ (see Exercise **??**).

C On the other hand, checking the unsatisfiability of CNF formulas or the validity of DNF formulas is coNP-complete. For any propositional formula ϕ there is an equivalent formula in CNF and DNF and I will prove this below by actually providing an effective procedure for the transformation. However, also because of the above comment on validity and satisfiability checking for CNF and DNF formulas, respectively, the transformation is costly. In general, a CNF or DNF of a formula ϕ is exponentially larger than ϕ as

long as the normal forms need to be logically equivalent. If this is not needed, then by the introduction of fresh propositional variables, CNF or DNF normal forms for ϕ can be computed in linear time in the size of ϕ. More concretely, given a formula ϕ instead of checking validity the unsatisfiability of $\neg\phi$ can be considered. Then the linear time CNF normal form algorithm (see Section **??**) is satisfiability preserving, i.e., the linear time CNF of $\neg\phi$ is unsatisfiable iff $\neg\phi$ is.

Proposition 2.6.2. For every formula there is an equivalent formula in CNF and also an equivalent formula in DNF.

Proof. See the rewrite systems $\Rightarrow_{\mathrm{BCNF}}$, and $\Rightarrow_{\mathrm{ACNF}}$ below and the lemmata on their properties. □

2.6.2 Basic CNF/DNF Transformation

The below algorithm bcnf is a basic algorithm for transforming any propositional formula into CNF, or DNF if the rule **PushDisj** is replaced by **PushConj**.

Algorithm 2: bcnf(ϕ)

Input : A propositional formula ϕ.
Output: A propositional formula ψ equivalent to ϕ in CNF.

1 **whilerule** *(***ElimEquiv**(ϕ)*)* **do** ;
2 **whilerule** *(***ElimImp**(ϕ)*)* **do** ;
3 **whilerule** *(***ElimTB1**(ϕ),. . .,**ElimTB6**(ϕ)*)* **do** ;
4 **whilerule** *(***PushNeg1**(ϕ),. . .,**PushNeg3**(ϕ)*)* **do** ;
5 **whilerule** *(***PushDisj**(ϕ)*)* **do** ;
6 **return** ϕ;

In the sequel I study only the CNF version of the algorithm. All properties hold in an analogous way for the DNF version. To start an informal analysis of the algorithm, consider the following example CNF transformation.

Example 2.6.3. Consider the formula $\neg((P \lor Q) \leftrightarrow (P \rightarrow (Q \land \top)))$ and the application of $\Rightarrow_{\mathrm{BCNF}}$ depicted in Figure 2.8. Already for this simple formula the CNF transformation via $\Rightarrow_{\mathrm{BCNF}}$ becomes quite messy. Note that the CNF result in Figure 2.8 is still highly redundant. If I remove all disjunctions that are trivially true, because they contain a propositional literal and its negation, the result becomes

$$(P \lor \neg Q) \land (\neg Q \lor \neg P) \land (\neg Q \lor \neg Q)$$

now elimination of duplicate literals beautifies the third clause and the overall formula into

$$(P \lor \neg Q) \land (\neg Q \lor \neg P) \land \neg Q.$$

Now let's inspect this formula a little closer. Any valuation satisfying the formula must set $\mathcal{A}(Q) = 0$, because of the third clause. But then the first two clauses are already satisfied. The formula $\neg Q$ *subsumes* the formulas $P \lor \neg Q$ and $\neg Q \lor \neg P$

ElimEquiv	$\chi[(\phi \leftrightarrow \psi)]_p$	$\Rightarrow_{\mathrm{BCNF}}$	$\chi[(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)]_p$
ElimImp	$\chi[(\phi \rightarrow \psi)]_p$	$\Rightarrow_{\mathrm{BCNF}}$	$\chi[(\neg\phi \vee \psi)]_p$
PushNeg1	$\chi[\neg(\phi \vee \psi)]_p$	$\Rightarrow_{\mathrm{BCNF}}$	$\chi[(\neg\phi \wedge \neg\psi)]_p$
PushNeg2	$\chi[\neg(\phi \wedge \psi)]_p$	$\Rightarrow_{\mathrm{BCNF}}$	$\chi[(\neg\phi \vee \neg\psi)]_p$
PushNeg3	$\chi[\neg\neg\phi]_p$	$\Rightarrow_{\mathrm{BCNF}}$	$\chi[\phi]_p$
PushDisj	$\chi[(\phi_1 \wedge \phi_2) \vee \psi]_p$	$\Rightarrow_{\mathrm{BCNF}}$	$\chi[(\phi_1 \vee \psi) \wedge (\phi_2 \vee \psi)]_p$
PushConj	$\chi[(\phi_1 \vee \phi_2) \wedge \psi]_p$	$\Rightarrow_{\mathrm{BDNF}}$	$\chi[(\phi_1 \wedge \psi) \vee (\phi_2 \wedge \psi)]_p$
ElimTB1	$\chi[(\phi \wedge \top)]_p$	$\Rightarrow_{\mathrm{BCNF}}$	$\chi[\phi]_p$
ElimTB2	$\chi[(\phi \wedge \bot)]_p$	$\Rightarrow_{\mathrm{BCNF}}$	$\chi[\bot]_p$
ElimTB3	$\chi[(\phi \vee \top)]_p$	$\Rightarrow_{\mathrm{BCNF}}$	$\chi[\top]_p$
ElimTB4	$\chi[(\phi \vee \bot)]_p$	$\Rightarrow_{\mathrm{BCNF}}$	$\chi[\phi]_p$
ElimTB5	$\chi[\neg\bot]_p$	$\Rightarrow_{\mathrm{BCNF}}$	$\chi[\top]_p$
ElimTB6	$\chi[\neg\top]_p$	$\Rightarrow_{\mathrm{BCNF}}$	$\chi[\bot]_p$

Figure 2.7: Basic CNF/DNF Transformation Rules

in this sense. The notion of subsumption will be discussed in detail for clauses in Section 2.7.

So it is eventually equivalent to

$$\neg Q.$$

The correctness of the result is obvious by looking at the original formula and doing a case analysis. For any valuation $\mathcal{A}$ with $\mathcal{A}(Q) = 1$ the two parts of the equivalence become true, independently of P, so the overall formula is false. For $\mathcal{A}(Q) = 0$, for any value of P, the truth values of the two sides of the equivalence are different, so the equivalence becomes false and hence the overall formula true.

After proving $\Rightarrow_{\mathrm{BCNF}}$ correct and terminating, in the succeeding section I will present an algorithm $\Rightarrow_{\mathrm{ACNF}}$ that actually generates $\neg Q$ out of $\neg((P \vee Q) \leftrightarrow (P \rightarrow (Q \wedge \top)))$ and does this without generating the mess of formulas $\Rightarrow_{\mathrm{BCNF}}$ does. Please recall that the above rules apply modulo commutativity of $\vee$, $\wedge$, e.g., the rule ElimTB1 is both applicable to the formulas $\phi \wedge \top$ and $\top \wedge \phi$.

I

Figure 2.1 contains more potential for simplification. For example, the idempotency equivalences $(\phi \wedge \phi) \leftrightarrow \phi$, $(\phi \vee \phi) \leftrightarrow \phi$ can be turned into simplification rules by applying them left to right. However, the way they are stated they can only be applied in case of identical subformulas. The formula $(P \vee Q) \wedge (Q \vee P)$ does this way not reduce to $(Q \vee P)$. A solution is to consider identity modulo commutativity. But then identity modulo commutativity and associativity (AC) as in $((P \vee Q) \vee R) \wedge (Q \vee (R \vee P)$ is still not detected. On the other hand, in practice, checking identity modulo AC is often too expensive. An elegant way out of this situation is to implement AC

$\neg((P \lor Q) \leftrightarrow (P \to (Q \land \top)))$
$\Rightarrow^{\text{Step 1}}_{\text{BCNF}} \neg([(P \lor Q) \to (P \to (Q \land \top))] \land [(P \to (Q \land \top)) \to (P \lor Q)])$
$\Rightarrow^{\text{Step 2}}_{\text{BCNF}} \neg([\neg(P \lor Q) \lor (P \to (Q \land \top))] \land [(P \to (Q \land \top)) \to (P \lor Q)])$
$\Rightarrow^{\text{Step 2}}_{\text{BCNF}} \neg([\neg(P \lor Q) \lor (P \to (Q \land \top))] \land [\neg(P \to (Q \land \top)) \lor (P \lor Q)])$
$\Rightarrow^{\text{Step 2}}_{\text{BCNF}} \neg([\neg(P \lor Q) \lor (P \to (Q \land \top))] \land [\neg(\neg P \lor (Q \land \top)) \lor (P \lor Q)])$
$\Rightarrow^{\text{Step 2}}_{\text{BCNF}} \neg([\neg(P \lor Q) \lor (\neg P \lor (Q \land \top))] \land [\neg(\neg P \lor (Q \land \top)) \lor (P \lor Q)])$
$\Rightarrow^{\text{Step 3}}_{\text{BCNF}} \neg([\neg(P \lor Q) \lor (\neg P \lor Q)] \land [\neg(\neg P \lor Q) \lor (P \lor Q)])$
$\Rightarrow^{\text{Step 4}}_{\text{BCNF}} \neg([(\neg P \land \neg Q) \lor (\neg P \lor Q)] \land [\neg(\neg P \lor Q) \lor (P \lor Q)])$
$\Rightarrow^{\text{Step 4}}_{\text{BCNF}} \neg([(\neg P \land \neg Q) \lor (\neg P \lor Q)] \land [(\neg\neg P \land \neg Q) \lor (P \lor Q)])$
$\Rightarrow^{\text{Step 4}}_{\text{BCNF}} \neg([(\neg P \land \neg Q) \lor (\neg P \lor Q)] \land [(\neg\neg P \land \neg Q) \lor (P \lor Q)])$
$\Rightarrow^{*,\text{Step 4}}_{\text{BCNF}} [(\neg\neg P \lor \neg\neg Q) \land (\neg\neg P \land \neg Q)] \lor [(\neg\neg\neg P \lor \neg\neg Q) \land (\neg P \land \neg Q)]$
$\Rightarrow^{*,\text{Step 4}}_{\text{BCNF}} [(P \lor Q) \land (P \land \neg Q)] \lor [(\neg P \lor Q) \land (\neg P \land \neg Q)]$
$\Rightarrow^{*,\text{Step 5}}_{\text{BCNF}} (P \lor Q \lor \neg P \lor Q) \land (P \lor Q \lor \neg P) \land (P \lor Q \lor \neg Q) \land (P \lor \neg P \lor Q) \land (P \lor \neg P) \land (P \lor \neg Q) \land (\neg Q \lor \neg P \lor Q) \land (\neg Q \lor \neg P) \land (\neg Q \lor \neg Q)$

Figure 2.8: Example Basic CNF Transformation

connectives like $\lor$ or $\land$ with flexible arity, to normalize nested occurrences of the connectives, and finally to sort the arguments using some total ordering. Applying this to $((P \lor Q) \lor R) \land (Q \lor (R \lor P)$ with ordering $R > P > Q$ the result is $(Q \lor P \lor R) \land (Q \lor P \lor R)$. Now complete AC simplification is back at the cost of checking for identical subformulas. Note that in an appropriate implementation, the normalization and ordering process is only done once at the start and then normalization and argument ordering is kept as an invariant.

2.6.3 Advanced CNF Transformation

The simple algorithm for CNF transformation Algorithm 2 can be improved in various ways: (i) more aggressive formula simplification, (ii) renaming, (iii) polarity dependant transformations. The before studied Example 2.6.3 serves already as a nice motivation for (i) and (iii). Firstly, removing $\top$ from the formula $\neg((P \lor Q) \leftrightarrow (P \to (Q \land \top)))$ first and not in the middle of the algorithm obviously shortens the overall process. Secondly, if the equivalence is replaced polarity dependant, i.e., using the equivalence $(\phi \leftrightarrow \psi) \leftrightarrow (\phi \land \psi) \lor (\neg\phi \land \neg\psi)$ and not the one used in rule ElimEquiv applied before, a lot of redundancy generated by $\Rightarrow_{\text{BCNF}}$ is prevented. In general, if $\psi[\phi_1 \leftrightarrow \phi_2]_p$ and $\text{pol}(\psi, p) = -1$ then for CNF transformation do $\psi[(\phi_1 \land \phi_2) \lor (\neg\phi_1 \land \neg\phi_2)]_p$ and if $\text{pol}(\psi, p) = 1$ do $\psi[(\phi_1 \to \phi_2) \land (\phi_2 \to \phi_1)]_p$

Item (ii) can be motivated by a formula

$$P_1 \leftrightarrow (P_2 \leftrightarrow (P_3 \leftrightarrow (\ldots (P_{n-1} \leftrightarrow P_n) \ldots)))$$

where Algorithm 2 generates a CNF with 2^n clauses out of this formula. The way out of this problem is the introduction of additional fresh propositional

variables that *rename* subformulas. The price to pay is that a renamed formula is not equivalent to the original formula due to the extra propositional variables, but satisfiability preserving. A renamed formula for the above formula is

$$(P_1 \leftrightarrow (P_2 \leftrightarrow Q_1)) \wedge (Q_1 \leftrightarrow (P_3 \leftrightarrow Q_2)) \wedge \ldots$$

where the Q_i are additional, fresh propositional variables. The number of clauses of the CNF of this formula is $4(n-1)$ where each conjunct $(Q_i \leftrightarrow (P_j \leftrightarrow Q_{i+1}))$ contributes four clauses.

Proposition 2.6.4. Let P be a propositional variable not occurring in $\psi[\phi]_p$.

1. If $\operatorname{pol}(\psi, p) = 1$, then $\psi[\phi]_p$ is satisfiable if and only if $\psi[P]_p \wedge (P \rightarrow \phi)$ is satisfiable.
2. If $\operatorname{pol}(\psi, p) = -1$, then $\psi[\phi]_p$ is satisfiable if and only if $\psi[P]_p \wedge (\phi \rightarrow P)$ is satisfiable.
3. If $\operatorname{pol}(\psi, p) = 0$, then $\psi[\phi]_p$ is satisfiable if and only if $\psi[P]_p \wedge (P \leftrightarrow \phi)$ is satisfiable.

Proof. Exercise. □

So depending on the formula ψ, the position p where the variable P is introduced definition of P is given by

$$\operatorname{def}(\psi, p, P) := \begin{cases} (P \rightarrow \psi|_p) & \text{if } \operatorname{pol}(\psi, p) = 1 \\ (\psi|_p \rightarrow P) & \text{if } \operatorname{pol}(\psi, p) = -1 \\ (P \leftrightarrow \psi|_p) & \text{if } \operatorname{pol}(\psi, p) = 0 \end{cases}$$

For renaming there are several choices which subformula to choose. Obviously, since a formula has only linearly many subformulas, renaming every subformula works [35, 29]. Basically this is what I show below. In the following section a renaming variant is introduced that produces smallest CNFs.

SimpleRenaming $\phi \Rightarrow_{\text{SimpRen}} \phi[P_1]_{p_1}[P_2]_{p_2} \ldots [P_n]_{p_n} \wedge \operatorname{def}(\phi, p_1, P_1) \wedge \ldots \wedge \operatorname{def}(\phi[P_1]_{p_1}[P_2]_{p_2} \ldots [P_{n-1}]_{p_{n-1}}, p_n, P_n)$
provided $\{p_1, \ldots, p_n\} \subset \operatorname{pos}(\phi)$ and for all $i, i+j$ either $p_i \parallel p_{i+j}$ or $p_i > p_{i+j}$ and the P_i are different and new to ϕ

Actually, the rule SimpleRenaming does not provide an effective way to compute the set $\{p_1, \ldots, p_n\}$ of positions in ϕ to be renamed. Where are several choices. Following Plaisted and Greenbaum [29], the set contains all positions from ϕ that do not point to a propositional variable or a negation symbol. In addition, renaming position ϵ does not make sense because it would generate the formula $P \wedge (P \rightarrow \phi)$ which results in more clauses than just ϕ. Choosing the set of Plaisted and Greenbaum prevents the explosion in the number of clauses during CNF transformation. But not all renamings are needed to this end.

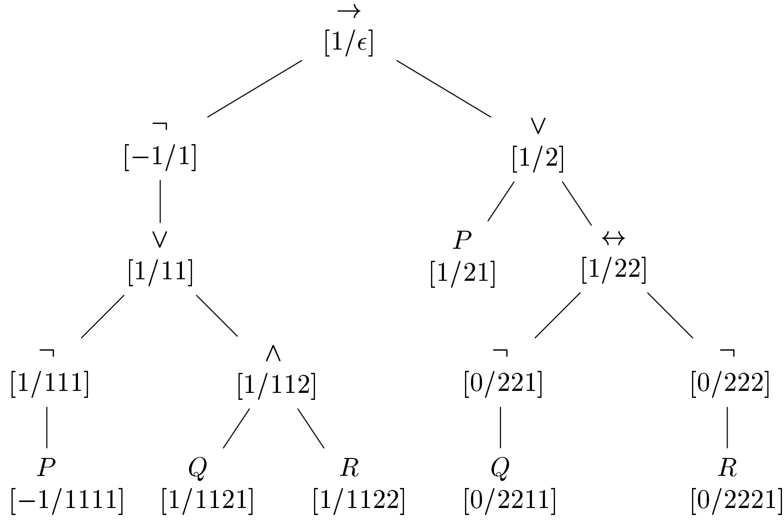


Figure 2.9: Tree representation of $[\neg(\neg P \vee (Q \wedge R))] \rightarrow [P \vee (\neg Q \leftrightarrow \neg R)]$ where each node is annotated with its [polarity/position].

A smaller set of positions from ϕ, let's call it the set of obvious positions, is still preventing the explosion and given by the rules: (i) if $\phi|_p$ is an equivalence and there is a position $q < p$ such that $\phi|_q$ is either an equivalence or disjunctive in ϕ then p is an obvious position (ii) if $\phi|_{pq}$ is a conjunctive formula in ϕ, $\phi|_p$ is a disjunctive formula in ϕ and for all positions r with $p < r < pq$ the formula $\phi|_r$ is not a conjunctive formula then pq is an obvious position. A formula $\phi|_p$ is conjunctive in ϕ if $\phi|_p$ is a conjunction and $\text{pol}(\phi, p) \in \{0, 1\}$ or $\phi|_p$ is a disjunction or implication and $\text{pol}(\phi, p) \in \{0, -1\}$. Analogously, a formula $\phi|_p$ is disjunctive in ϕ if $\phi|_p$ is a disjunction or implication and $\text{pol}(\phi, p) \in \{0, 1\}$ or $\phi|_p$ is a conjunction and $\text{pol}(\phi, p) \in \{0, -1\}$.

Consider as an example the formula

$$[\neg(\neg P \vee (Q \wedge R))] \rightarrow [P \vee (\neg Q \leftrightarrow \neg R)]$$

. Its tree representation as well as the polarity and position of each node is shown in Figure 2.9.

The before mentioned polarity dependent transformations for equivalences are realized by the following two rules:

ElimEquiv1 $\chi[(\phi \leftrightarrow \psi)]_p \;\Rightarrow_{\text{ACNF}}\; \chi[(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)]_p$
provided $\text{pol}(\chi, p) \in \{0, 1\}$

ElimEquiv2 $\chi[(\phi \leftrightarrow \psi)]_p \;\Rightarrow_{\text{ACNF}}\; \chi[(\phi \wedge \psi) \vee (\neg\phi \wedge \neg\psi)]_p$
provided $\text{pol}(\chi, p) = -1$

Algorithm 3: acnf(ϕ)

Input : A formula ϕ.
Output: A formula ψ in CNF satisfiability preserving to ϕ.
1 **whilerule** *(***ElimTB1**(ϕ),...,**ElimTB6**(ϕ)*)* **do** ;
2 **SimpleRenaming**(ϕ) on obvious positions;
3 **whilerule** *(***ElimEquiv1**(ϕ),**ElimEquiv2**(ϕ)*)* **do** ;
4 **whilerule** *(***ElimImp**(ϕ)*)* **do** ;
5 **whilerule** *(***PushNeg1**(ϕ),...,**PushNeg3**(ϕ)*)* **do** ;
6 **whilerule** *(***PushDisj**(ϕ)*)* **do** ;
7 **return** ϕ;

$$\neg((P \lor Q) \leftrightarrow (P \rightarrow (Q \land \top)))$$
$$\Rightarrow^{\text{Step 1}}_{\text{ACNF}} \neg((P \lor Q) \leftrightarrow (P \rightarrow Q))$$
$$\Rightarrow^{\text{Step 3}}_{\text{ACNF}} \neg(((P \lor Q) \land (P \rightarrow Q)) \lor (\neg(P \lor Q) \land \neg(P \rightarrow Q)))$$
$$\Rightarrow^{*,\text{Step 4}}_{\text{ACNF}} \neg(((P \lor Q) \land (\neg P \lor Q)) \lor (\neg(P \lor Q) \land \neg(\neg P \lor Q)))$$
$$\Rightarrow^{*,\text{Step 5}}_{\text{ACNF}} ((\neg P \land \neg Q) \lor (P \land \neg Q)) \land ((P \lor Q) \lor (\neg P \lor Q))$$
$$\Rightarrow^{*,\text{Step 6}}_{\text{ACNF}} (\neg P \lor P) \land (\neg P \lor \neg Q) \land (\neg Q \lor P) \land (\neg Q \lor \neg Q) \land (P \lor Q \lor \neg P \lor Q)$$

Figure 2.10: Example Advanced CNF Transformation

Proposition 2.6.5 (Models of Renamed Formulas)**.** Let ϕ be a formula and ϕ' a renamed CNF of ϕ computed by acnf. Then any (partial) model $\mathcal{A}$ of ϕ' is also a model for ϕ.

Proof. By an inductive argument it is sufficient to consider one renaming application, i.e., $\phi' = \phi[P]_p \land \text{def}(\phi, p, P)$. There are three cases depending on the polarity. (i) if $\text{pol}(\phi, p) = 1$ then $\phi' = \phi[P]_p \land P \rightarrow \phi|_p$. If $\mathcal{A}(P) = 1$ then $\mathcal{A}(\phi|_p) = 1$ and hence $\mathcal{A}(\phi) = 1$. The interesting case is $\mathcal{A}(P) = 0$ and $\mathcal{A}(\phi|_p) = 1$. But then because $\text{pol}(\phi, p) = 1$ also $\mathcal{A}(\phi) = 1$ by Lemma 2.2.7. (ii) if $\text{pol}(\phi, p) = -1$ the case is symmetric to the previous one. Finally, (iii) if $\text{pol}(\phi, p) = 0$ for any $\mathcal{A}$ satisfying ϕ' it holds $\mathcal{A}(\phi|_p) = \mathcal{A}(P)$ and hence $\mathcal{A}(\phi) = 1$. ☐

2.6.4 Computing Small CNFs

In the previous chapter obvious positions are a suggestion for smaller CNFs with respect to the renaming positions suggested by Plaisted and Greenbaum. In this section I develop a set of renaming positions that is in fact minimal with respect to the resulting CNF. A subformula is renamed if the eventual number of generated clauses by bcnf decreases after renaming [10, 28]. If formulas are checked top-down for this condition, and profitable formulas in the above sense are renamed, the resulting CNF is optimal in the number of clauses [10]. The below function ac computes the number of clauses generated by the algorithm bcnf, as long as the formula does not contain $\top$ or $\bot$.

C

A state of the art CNF algorithm first tries to simplify a formula before doing the actual CNF transformation. Eliminating $\top$ or $\bot$ using the ElimTB is a standard part of any such simplification procedure. Further simplifications are discussed in Section 2.13.

ψ	$\mathrm{ac}(\psi)$	$\mathrm{bc}(\psi)$
$\phi_1 \land \phi_2$	$\mathrm{ac}(\phi_1) + \mathrm{ac}(\phi_2)$	$\mathrm{bc}(\phi_1)\,\mathrm{bc}(\phi_2)$
$\phi_1 \lor \phi_2$	$\mathrm{ac}(\phi_1)\,\mathrm{ac}(\phi_2)$	$\mathrm{bc}(\phi_1) + \mathrm{bc}(\phi_2)$
$\phi_1 \rightarrow \phi_2$	$\mathrm{bc}(\phi_1)\,\mathrm{ac}(\phi_2)$	$\mathrm{ac}(\phi_1) + \mathrm{bc}(\phi_2)$
$\phi_1 \leftrightarrow \phi_2$	$\mathrm{ac}(\phi_1)\,\mathrm{bc}(\phi_2) + \mathrm{bc}(\phi_1)\,\mathrm{ac}(\phi_2)$	$\mathrm{ac}(\phi_1)\,\mathrm{ac}(\phi_2) + \mathrm{bc}(\phi_1)\,\mathrm{bc}(\phi_2)$
$\neg\phi_1$	$\mathrm{bc}(\phi_1)$	$\mathrm{ac}(\phi_1)$
P	1	1

Let ϕ be a formula that does not contain $\bot$, or $\top$, then $\mathrm{ac}(\phi)$ computes exactly the number of clauses generated by $\mathrm{bcnf}(\phi)$. The proof is left as an exercise, but as an example consider the case where $\phi = L_1 \ldots L_n$ is a disjunction of literals. In this case bcnf does not change ϕ at all ad produces exactly the clause ϕ. Expanding the definition of $\mathrm{ac}(\phi)$ produces $\mathrm{ac}(\phi) = \mathrm{ac}(L_1)\,\mathrm{ac}(L_2)\ldots\mathrm{ac}(L_n) = 1$ because if some L_i is a propositional variable, then $\mathrm{ac}(L_i) = 1$. If some L_j is negative, i.e., $L_j = \neg P$ then $\mathrm{ac}(L_j) = \mathrm{ac}(\neg P) = \mathrm{bc}(P) = 1$.

A renaming yields fewer clauses, if the difference between the number of clauses generated without and with a renaming is positive. Consider the renaming of a subformula at position p within a formula ψ with fresh variable P. The condition to be checked is

$$\mathrm{ac}(\psi) \geq \mathrm{ac}(\psi[P]_p) + \mathrm{ac}(\mathrm{def}(\psi, p, P)).$$

The inequality above is not strict. If some formula $\phi = \psi|_p$ is replaced inside ψ where $\mathrm{ac}(\psi) = \mathrm{ac}(\psi[P]_p) + \mathrm{ac}(\mathrm{def}(\psi, p, P))$ then this equation turns into a strict inequality as soon as we do another replacement inside ϕ. In this case $\mathrm{ac}(\mathrm{def}(\psi, p, P))$ will strictly decrease. Therefore, when searching for a minimal CNF it is mandatory to consider the above inequality non-strict.

Example 2.6.6. For a formula $P_1 \leftrightarrow P_2$ renaming does not pay off. If P_2 is replaced by some fresh variable Q the result is $P_1 \leftrightarrow Q \land Q \leftrightarrow P_2$ where the original formula generates 2 clauses and the formula after replacement generates 4 clauses.

The break even point for nested equivalences is the formula $P_1 \leftrightarrow (P_2 \leftrightarrow (P_3 \leftrightarrow P_4))$ where replacement at position 22 using the fresh variable Q results in $P_1 \leftrightarrow (P_2 \leftrightarrow Q) \land Q \leftrightarrow (P_3 \leftrightarrow P_4)$. Both formulas eventually generate 8 clauses. So this is an example for the above inequality to be non-strict.

The obvious problem with this condition is that the function ac cannot be efficiently computed in general, for it grows exponentially in the size of the input formula. Moreover, a straightforward, naive top-down implementation of ac following the above table results in an algorithm with exponential time complexity, due to the duplication of recursive calls. The exponential complexity

can be avoided using a dynamic programming idea: simply store intermediate results for subformulas. Nevertheless, because ac grows exponentially, computing ac requires arbitrary precision integer arithmetic. It turns out that this can hardly be afforded in practice. The rest of this section is therefore concerned with a solution to this problem, i.e., I show that it is not necessary to compute ac at all for deciding the above inequation.

Obviously, the formulas ψ and $\psi[P]_p$ differ only at position p, the other parts of the formulas remain identical. We make use of this fact by an abstraction of those parts of ψ that do not influence the changed position. To this end we introduce the notion of a coefficient as shown in Table 2.1.

p	$\psi\|_q$	a^ψ_p	b^ψ_p
$q.i$	$\phi_1 \wedge \phi_2$	a^ψ_q	$b^\psi_q \prod_{j \neq i} \mathrm{bc}(\phi_j)$
$q.i$	$\phi_1 \vee \phi_2$	$a^\psi_q \prod_{j \neq i} \mathrm{ac}(\phi_j)$	b^ψ_q
$q.1$	$\phi_1 \rightarrow \phi_2$	b^ψ_q	$a^\psi_q \, \mathrm{ac}(\phi_2)$
$q.2$	$\phi_1 \rightarrow \phi_2$	$a^\psi_q \, \mathrm{bc}(\phi_1)$	b^ψ_q
$q.1$	$\phi_1 \leftrightarrow \phi_2$	$a^\psi_q \, \mathrm{bc}(\phi_2) + b^\psi_q \, \mathrm{ac}(\phi_2)$	$a^\psi_q \, \mathrm{ac}(\phi_2) + b^\psi_q \, \mathrm{bc}(\phi_2)$
$q.2$	$\phi_1 \leftrightarrow \phi_2$	$a^\psi_q \, \mathrm{bc}(\phi_1) + b^\psi_q \, \mathrm{ac}(\phi_1)$	$a^\psi_q \, \mathrm{ac}(\phi_1) + b^\psi_q \, \mathrm{bc}(\phi_1)$
$q.1$	$\neg\phi_1$	b^ψ_q	a^ψ_q
ϵ	ψ	1	0

Table 2.1: Calculating the Coefficients

The coefficients determine how often a particular subformula and its negation are duplicated in the course of a basic CNF translation. The coefficient a^ψ_p is the factor of $\mathrm{ac}(\psi|_p)$ in the recursive computation whereas the factor b^ψ_p is the factor of $\mathrm{bc}(\psi|_p)$. The first column of Table 2.1 shows the form of p, the second column the form of ψ directly above position p (ψ itself if $p = \epsilon$). The next two columns demonstrate the corresponding recursive bottom-up calculations for a^ψ_p and b^ψ_p, respectively. Applied to our starting example formula $\psi = \phi_1 \vee \forall x\, \phi_2$ where we renamed position 2.1, i.e., the subformula ϕ_2, the coefficients are $a^\psi_{2.1} = \mathrm{ac}(\phi_1)$ (Table 2.1, eighth, second and last row, first column) and $b^\psi_{2.1} = 0$ (eighth, second and last row, second column). Note that a^ψ_p (b^ψ_p) is always 0 if $\mathrm{pol}(\psi, p) = -1$ ($\mathrm{pol}(\psi, p) = 1$).

Using the notion of a coefficient, the previously stated condition can be reformulated as

$$a^\psi_p \, \mathrm{ac}(\phi) + b^\psi_p \, \mathrm{bc}(\phi) \geq a^\psi_p + b^\psi_p + \mathrm{ac}(\mathrm{def}(\psi, p, P))$$

where we still assume that $\phi = \psi|_p$ and P is a fresh propositional variable. Note that, since ϕ is replaced by P in ψ at position p, the coefficients a^ψ_p, b^ψ_p are multiplied by 1 in the renamed version, because $\mathrm{ac}(P) = \mathrm{bc}(P) = 1$. Depending on the polarity of $\psi|_p$ the inequality is equivalent to one of the three inequalities:

$$
\begin{array}{rl}
a_p^\psi \,\mathrm{ac}(\phi) \geq a_p^\psi + \mathrm{ac}(\phi) & \text{if } \mathrm{pol}(\psi,p) = 1 \\
b_p^\psi \,\mathrm{bc}(\phi) \geq b_p^\psi + \mathrm{bc}(\phi) & \text{if } \mathrm{pol}(\psi,p) = -1 \\
a_p^\psi \,\mathrm{ac}(\phi) + b_p^\psi \,\mathrm{bc}(\phi) \geq a_p^\psi + b_p^\psi + \mathrm{ac}(\phi) + \mathrm{bc}(\phi) & \text{if } \mathrm{pol}(\psi,p) = 0
\end{array}
$$

By simple arithmetical transformations, we can group all occurrences of factors a_p^ψ, b_p^ψ and all occurrences of $\mathrm{ac}(\phi)$ and $\mathrm{bc}(\phi)$, respectively:

$$
\begin{array}{rl}
(a_p^\psi - 1)(\mathrm{ac}(\phi) - 1) \geq 1 & \text{if } \mathrm{pol}(\psi,p) = 1 \\
(b_p^\psi - 1)(\mathrm{bc}(\phi) - 1) \geq 1 & \text{if } \mathrm{pol}(\psi,p) = -1 \\
(a_p^\psi - 1)(\mathrm{ac}(\phi) - 1) + (b_p^\psi - 1)(\mathrm{bc}(\phi) - 1) \geq 2 & \text{if } \mathrm{pol}(\psi,p) = 0
\end{array}
$$

Let us abbreviate the product $(a_p^\psi - 1)(\mathrm{ac}(\phi) - 1)$ with p_a and $(b_p^\psi - 1)(\mathrm{bc}(\phi) - 1)$ with p_b. Since neither p_a nor p_b can become negative, in any of the cases where they appear, the first inequality holds if $p_a \geq 1$, the second inequality holds if $p_b \geq 1$ and the third inequality holds if (i) $p_a \geq 2$ or (ii) $p_b \geq 2$ or (iii) $p_a \geq 1$ and $p_b \geq 1$. In order to check these conditions, it suffices to test whether the coefficients a_p^ψ, b_p^ψ and the number of clauses $\mathrm{ac}(\phi)$, $\mathrm{bc}(\phi)$ are strictly greater than 1, 2 or 3, respectively. This can always be checked in linear time with respect to the size of ψ. The condition $\mathrm{ac}(\phi) > 1$ holds iff there exists a position p such that $\phi[\phi_1 \leftrightarrow \phi_2]_p$ or $\phi[\phi_1 \land \phi_2]_p$ and $\mathrm{pol}(\phi, p) = 1$ or $\phi[\phi_1 \circ \phi_2]_p$ with $\mathrm{pol}(\phi, p) = -1$ and $\circ \in \{\lor, \rightarrow\}$. The computations for the boolean conditions $\mathrm{ac}(\phi) > 2$ and $\mathrm{ac}(\phi) > 3$ are depicted in Table 2.2. The computation of the conditions for bc works accordingly, see Table 2.3.

As for the factors, Table 2.4 shows how to compute $a_p^\psi > 1$ and, following Table 2.1, this can be extended to the other cases for the a factor and the corresponding conditions for the b factor.

Hence we turned a test that required the computation of exponentially growing functions into a boolean condition that does not require any arithmetic calculation at all.

Theorem 2.6.7 (Formula Renaming)**.** Formula Renaming preserves satisfiability and can be computed in polynomial time.

In order to further reduce the number of eventually generated clauses it may still be useful to rename a formula, even if the above considerations do not apply. For example, renaming the formula $P_1 \lor (Q_1 \land Q_2)$ at position 2 results in three clauses, whereas a standard CNF translation of the original formula yields two clauses. This calculation also applies if this situation is repeated, as in

$$[P_1 \lor (Q_1 \land Q_2)] \land [P_2 \lor (Q_1 \land Q_2)] \land \ldots [P_n \lor (Q_1 \land Q_2)]$$

where our renaming criterion does not apply. But now a simultaneous renaming of all occurrences $(Q_1 \land Q_2)$ may pay off. It results in $n + 2$ clauses whereas the standard CNF translation yields $2n$ clauses. Hence, it is useful to search for multiple occurrences of the same subformula. The problem here is to find an appropriate "equality" or "instance" relation between subformulas. In our example syntactic equality was sufficient to detect all such occurrences. In general, a matching process – probably with respect to the commutativity, associativity

ψ	$\text{ac}(\psi) > 1$
$\phi_1 \wedge \phi_2$	*true*
$\phi_1 \vee \phi_2$	$\text{ac}(\phi_1) > 1$ or $\text{ac}(\phi_2) > 1$
$\phi_1 \rightarrow \phi_2$	$\text{bc}(\phi_1) > 1$ or $\text{ac}(\phi_2) > 1$
$\phi_1 \leftrightarrow \phi_2$	*true*
$\neg\phi$	$\text{bc}(\phi) > 1$

ψ	$\text{ac}(\psi) > 2$
$\phi_1 \wedge \phi_2$	$\text{ac}(\phi_1) > 1$ or $\text{ac}(\phi_2) > 1$
$\phi_1 \vee \phi_2$	$\text{ac}(\phi_i) > 2$ or $[\text{ac}(\phi_1) > 1$ and $\text{ac}(\phi_2) > 1]$
$\phi_1 \rightarrow \phi_2$	$\text{bc}(\phi_1) > 2$ or $\text{ac}(\phi_2) > 2$ or $[\text{bc}(\phi_1) > 1$ and $\text{ac}(\phi_2) > 1]$
$\phi_1 \leftrightarrow \phi_2$	at least one out of ϕ_1, ϕ_2 is not a literal
$\neg\phi$	$\text{bc}(\phi) > 2$

ψ	$\text{ac}(\psi) > 3$
$\phi_1 \wedge \phi_2$	$\text{ac}(\phi_i) > 2$
$\phi_1 \vee \phi_2$	$\text{ac}(\phi_i) > 3$ or $[\text{ac}(\phi_i) > 2$ and $\text{ac}(\phi_j) > 1, i \neq j]$
$\phi_1 \rightarrow \phi_2$	$\text{bc}(\phi_1) > 2$ or $\text{ac}(\phi_2) > 2$ or $[\text{bc}(\phi_1) > 1$ and $\text{ac}(\phi_2) > 1]$
$\phi_1 \leftrightarrow \phi_2$	$\text{ac}(\phi_i) > 3$ or $\text{bc}(\phi_i) > 3$ or ϕ_2 is not a literal
$\neg\phi$	$\text{bc}(\phi) > 3$

Table 2.2: The Boolean Conditions for ac

of some logical operators or even logical implication – may be needed to obtain a suitable renaming result. So we run here into a tradeoff between compact CNFs and computational complexity to achieve these CNFs.

For the formulation of the optimized CNF algorithm I rely on the equivalences from categories (I), (V) and (VII) from Figure 2.1. They are used to transform the formula. The equivalences are always applied from left to right. So "applying" such an equivalence means turning it into a rule. For example, the equivalence $(\phi \vee (\phi \wedge \psi)) \leftrightarrow \phi$ from category (V) generates the rule

$$\chi[\phi \vee (\phi \wedge \psi)]_p \quad \Rightarrow_{\text{OCNF}} \quad \chi[\phi]_p$$

Applying this rule with respect to commutativity of $\vee$ means, for example, that both the formulas $(\phi \vee (\phi \wedge \psi))$ and $((\phi \wedge \psi) \vee \phi)$ can be transformed by the rule to ϕ where in both cases $p = \epsilon$. Rules are always applied modulo associativity and commutativity of $\wedge$, $\vee$.

ψ	$\text{bc}(\psi) > 1$
$\phi_1 \wedge \phi_2$	$\text{bc}(\phi_1) > 1$ or $\text{bc}(\phi_2) > 1$
$\phi_1 \vee \phi_2$	*true*
$\phi_1 \rightarrow \phi_2$	*true*
$\phi_1 \leftrightarrow \phi_2$	*true*
$\neg\phi$	$\text{ac}(\phi) > 1$

ψ	$\text{bc}(\psi) > 2$
$\phi_1 \vee \phi_2$	$\text{bc}(\phi_1) > 1$ or $\text{bc}(\phi_2) > 1$
$\phi_1 \wedge \phi_2$	$\text{bc}(\phi_i) > 2$ or $\text{bc}(\phi_1) > 1$ and $\text{bc}(\phi_2) > 1$
$\neg\phi$	$\text{ac}(\phi) > 2$

ψ	$\text{bc}(\psi) > 3$
$\phi_1 \vee \phi_2$	$\text{bc}(\phi_i) > 2$
$\phi_1 \wedge \phi_2$	$\text{bc}(\phi_i) > 3$ or $[\text{bc}(\phi_i) > 2$ and $\text{bc}(\phi_j) > 1, i \neq j]$
$\neg\phi$	$\text{ac}(\phi) > 3$

Table 2.3: The Boolean Conditions for bc

The procedure is depicted in Algorithm 4. Although computing ac for Step 2 is not practical in general, because the function is exponentially growing, the test $\text{ac}(\psi[\phi]_p) > \text{ac}(\psi[P]_p \wedge \text{def}(\psi, p, P))$ can be computed in constant time after a linear time processing phase.

Applying Algorithm 4 to the formula $\neg((P \vee Q) \leftrightarrow (P \rightarrow (Q \wedge \top)))$ of Example 2.6.3 results in the transformation depicted in Figure 2.11. Looking at the result it is already very close to $\neg Q$, as it contains the clause $(\neg Q \vee \neg Q)$. Removing duplicate literals in clauses and removing clauses containing complementary literals from the result yields

$$(\neg P \vee \neg Q) \wedge (\neg Q \vee P) \wedge \neg Q$$

which is even closer to just $\neg Q$. The first two clauses can actually be removed because they are *subsumed* by $\neg Q$, i.e., considered as multisets, $\neg Q$ is a subset of these clauses. Subsumption will be introduced in the next section. Logically, they can be removed because $\neg Q$ has to be true for any satisfying assignment of the formula and then the first two clauses are satisfied anyway.

Algorithm 4: ocnf(ϕ)

Input : A formula ϕ.
Output: A formula ψ in CNF satisfiability preserving to ϕ.

1 **whilerule** *(* **ElimRedI**(ϕ)*,* **ElimRedV**(ϕ)*,* **ElimRedVII**(ϕ)*)* **do** ;
2 **SimpleRenaming**(ϕ) on beneficial positions;
3 **whilerule** *(* **ElimEquiv1**(ϕ)*,* **ElimEquiv2**(ϕ)*)* **do** ;
4 **whilerule** *(* **ElimImp**(ϕ)*)* **do** ;
5 **whilerule** *(* **PushNeg1**(ϕ),...,**PushNeg3**(ϕ)*)* **do** ;
6 **whilerule** *(* **PushDisj**(ϕ)*)* **do** ;
7 **return** ϕ;

$$
\begin{array}{l}
\neg((P \lor Q) \leftrightarrow (P \rightarrow (Q \land \top))) \\
\Rightarrow^{\text{Step 1}}_{\text{OCNF}} \ \neg([(P \lor Q) \leftrightarrow (P \rightarrow Q)]) \\
\Rightarrow^{\text{Step 3}}_{\text{OCNF}} \ \neg([(P \lor Q) \land (P \rightarrow Q)] \lor [\neg(P \lor Q) \land \neg(P \rightarrow Q)]) \\
\Rightarrow^{\text{Step 2}}_{\text{OCNF}} \ \neg([(P \lor Q) \land (\neg P \lor Q)] \lor [\neg(P \lor Q) \land \neg(\neg P \lor Q)]) \\
\Rightarrow^{*,\text{Step 3}}_{\text{OCNF}} \ (\neg[(P \lor Q) \land (\neg P \lor Q)] \land \neg[\neg(P \lor Q) \land \neg(\neg P \lor Q)]) \\
\Rightarrow^{*,\text{Step 3}}_{\text{OCNF}} \ [\neg(P \lor Q) \lor \neg(\neg P \lor Q)] \land [(P \lor Q) \lor (\neg P \lor Q)] \\
\Rightarrow^{*,\text{Step 3}}_{\text{OCNF}} \ [(\neg P \land \neg Q) \lor (P \land \neg Q)] \land [(P \lor Q) \lor (\neg P \lor Q)] \\
\Rightarrow^{*,\text{Step 4}}_{\text{OCNF}} \ [(\neg P \lor P) \land (\neg P \lor \neg Q) \land (\neg Q \lor P) \land (\neg Q \lor \neg Q)] \land [P \lor Q \lor \neg P \lor Q]
\end{array}
$$

Figure 2.11: Example Optimized CNF Transformation

p	$\psi\|_p$	$a_p^\psi > 1$
$q.i$	$\phi_1 \wedge \phi_2$	$a_p^\psi > 1$
$q.i$	$\phi_1 \vee \phi_2$	$a_p^\psi > 1$ or $\text{ac}(\phi_i) > 1$ for some i

Table 2.4: The Boolean Conditions for a

2.7 Propositional Resolution

A *calculus* is a set of *inference* and *reduction* rules for a given logic (here PROP(Σ)). We only consider calculi operating on a set of clauses N. Inference rules *add* new clauses to N whereas reduction rules *remove* clauses from N or *replace* clauses by "simpler" ones.

We are only interested in unsatisfiability, i.e., the considered calculi test whether a clause set N is unsatisfiable. This is in particular motivated by the renaming step of CNF transformation, see Section 2.6.3. So, in order to check validity of a formula ϕ we check unsatisfiability of the clauses generated from $\neg\phi$.

For clauses we switch between the notation as a disjunction, e.g., $P \vee Q \vee P \vee \neg R$, and the notation as a multiset, e.g., $\{P, Q, P, \neg R\}$. This makes no difference as we consider $\vee$ in the context of clauses always modulo AC. Note that $\bot$, the empty disjunction, corresponds to $\emptyset$, the empty multiset. Clauses are typically denoted by letters C, D, possibly with subscript.

The *resolution calculus* consists of the inference rules *Resolution* and *Factoring*. So, if we consider clause sets N as states, $\uplus$ is disjoint union, we get the inference rules

Resolution $\quad (N \uplus \{C_1 \vee P, C_2 \vee \neg P\}) \Rightarrow_{\text{RES}} (N \cup \{C_1 \vee P, C_2 \vee \neg P\} \cup \{C_1 \vee C_2\})$

Factoring $\quad (N \uplus \{C \vee L \vee L\}) \Rightarrow_{\text{RES}} (N \cup \{C \vee L \vee L\} \cup \{C \vee L\})$

Theorem 2.7.1. The resolution calculus is sound and complete:

$$N \text{ is unsatisfiable iff } N \Rightarrow^*_{\text{RES}} \{\bot\}$$

Proof. ($\Leftarrow$) Soundness means for all rules that $N \models N'$ where N' is the clause set obtained from N after applying Resolution or Factoring. For Resolution it is sufficient to show that $C_1 \vee P, C_2 \vee \neg P \models C_1 \vee C_2$. This is obvious by a case analysis of valuations satisfying $C_1 \vee P, C_2 \vee \neg P$: of P is true in such a valuation so must be C_2, hence $C_1 \vee C_2$. If P is false in some valuation then C_1 must be true and so $C_1 \vee C_2$. Soundness for Factoring is obvious this way because it simply removes a duplicate literal in the respective clause.

($\Rightarrow$) The traditional method of proving resolution completeness are *semantic trees*. A *semantic tree* is a binary tree where the edges a labeled with literals

such that: (i) edges of children of the same parent are labeled with L and $\neg L$, and (ii) any node has either no or two children, and (iii) for any path from the root to a leave, each propositional variable occurs at most once. Therefore, each path corresponds to a partial valuation. Now for an unsatisfiable clause set N there is a semantic tree such that for each leaf of the tree there is a clause in N that is false with respect to the partial valuation at that leaf. Let this tree be minimal in the sense that there is no smaller tree with less nodes having this property. Now consider two sister leaves of the same parent of this tree, where the edges are labeled with L and $\neg L$, respectively. Let C_1 and C_2 be the two false clauses at the respective leaves. Obviously, $C_1 = C_1' \vee L$ and $C_2 = C_2' \vee \neg L$ as for otherwise the tree would not be minimal. If C_1 (or C_2) contains further occurrences of L (or C_2 of $\neg L$), then the rule Factoring is applied to eventually remove all additional occurrences. Therefore, I can assume $L \notin C_1'$ and $\neg L \notin C_2'$. A resolution step between these two clauses on L yields $C_1' \vee C_2'$ which is false at the parent of the two leaves, because the resolvent neither contains L nor $\neg L$. Furthermore, the resulting tree from cutting the two leaves is minimal for $N \cup \{C_1' \vee C_2'\}$ and strictly smaller. By an inductive argument this proves completeness. ☐

Example 2.7.2 (Resolution Completeness)**.** Consider the clause set

$$P \vee Q,\ \neg P \vee Q,\ P \vee \neg Q,\ \neg P \vee \neg Q \vee S,\ \neg P \vee \neg Q \vee \neg S$$

and the corresponding semantic tree as shown in Figure 2.12.

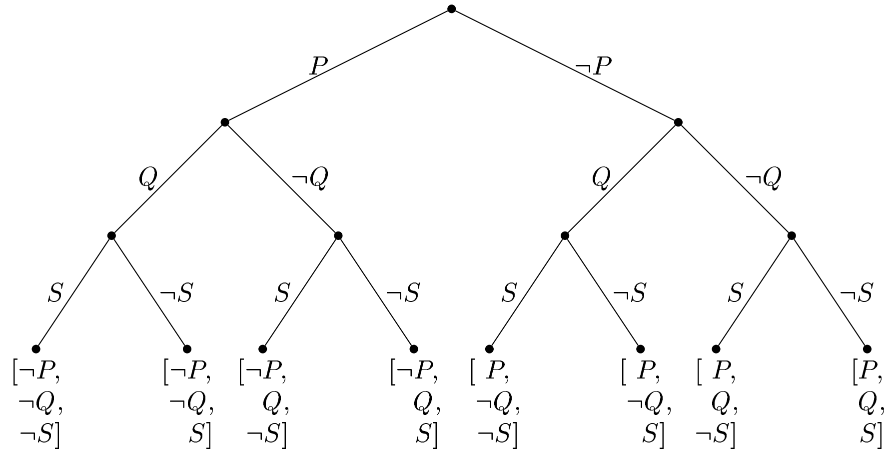


Figure 2.12: Semantic tree representation of $\{P \vee Q,\ \neg P \vee Q,\ P \vee \neg Q,\ \neg P \vee \neg Q \vee S,\ \neg P \vee \neg Q \vee \neg S\}$ where each leaf is labeled with the literals that falsify the partial valuation at that leaf.

The reduction rules are

Subsumption $(N \uplus \{C_1, C_2\}) \Rightarrow_{\text{RES}} (N \cup \{C_1\})$
provided $C_1 \subset C_2$

Tautology Deletion $(N \uplus \{C \lor P \lor \neg P\}) \Rightarrow_{\text{RES}} (N)$

Condensation $(N \uplus \{C_1 \lor L \lor L\}) \Rightarrow_{\text{RES}} (N \cup \{C_1 \lor L\})$

Note the different nature of inference rules and reduction rules. Resolution and Factorization only add clauses to the set whereas Subsumption, Tautology Deletion and Condensation delete clauses or replace clauses by "simpler" ones. In the next section, Section 2.8, I will show that "simpler" means.

At first, it looks strange to have the same rule both as a reduction rules and as an inference rule, i.e., Factorization and Condensation. On the propositional level there is obviously no difference and it is possible to get rid of one of the two. In Section 3.13 the resolution calculus is extended to first-order logic. In first-order logic Factorization and Condensation are actually different. They are separated here to eventually obtain the same set of resolution calculus rules for propositional and first-order logic.

Proposition 2.7.3. The reduction rules Subsumption, Tautology Deletion and Condensation are sound.

Proof. This is obvious for Tautology Deletion and Condensation. For Subsumption we have to show that $C_1 \models C_2$, because this guarantees that if $N \cup \{C_1\}$ has a model, $N \uplus \{C_1, C_2\}$ has a model too. So assume $\mathcal{A}(C_1) = 1$ for an arbitrary $\mathcal{A}$. Then there is some literal $L \in C_1$ with $\mathcal{A}(L) = 1$. Since $C_1 \subseteq C_2$, also $L \in C_2$ and therefore $\mathcal{A}(C_2) = 1$. ☐

Theorem 2.7.4 (Resolution Termination)**.** If redundancy rules are preferred over inference rules and no inference rule is applied twice to the same clause(s), then $\Rightarrow^{+}_{\text{RES}}$ is well-founded.

Proof. For some given clause set N the redundancy rules Subsumption, Tautology Deletion and Condensation always terminate because they all reduce the number of literals occurring in N. Furthermore, a clause set N where the redundancy rules have been exhaustively applied does not contain any tautology, no clause with duplicate literals and, in particular, no duplicate clauses. The number of such clauses can be overestimated by 3^n where n is the number of propositional variables in N. Hence, there are at most 2^{3^n} different, finite clause sets with respect to clause sets where the redundancy rules have been applied. Obviously, for each of such clause sets there are only finitely many different Resolution and Factoring steps. ☐

C Of course, what needs to be shown is that the strategy employed in Theorem 2.7.4 is still complete. This is not completely trivial and gets very nasty using semantic trees as the proof method of choice. So let's wait until superposition is established where this result becomes a particular case of superposition completeness.

2.8 Propositional Superposition

Superposition was originally developed for first-order logic [5]. Here I introduce its projection to propositional logic. Compared to the resolution calculus superposition adds (i) ordering and selection restrictions on inferences, (ii) an abstract redundancy notion, (iii) the notion of a partial model for inference guidance, and (iv) a *saturation* concept.

Definition 2.8.1 (Clause Ordering)**.** Let $\prec$ be a total strict ordering on Σ. Then $\prec$ can be lifted to a total ordering on literals by $\prec \subseteq \prec_L$ and $P \prec_L \neg P$ and $\neg P \prec_L Q$, $\neg P \prec_L \neg Q$ for all $P \prec Q$. The ordering $\prec_L$ can be lifted to a total ordering on clauses $\prec_C$ by considering the multiset extension of $\prec_L$ for clauses.

Proposition 2.8.2 (Properties of the Clause Ordering)**.** (i) The orderings on literals and clauses are total and well-founded.
(ii) Let C and D be clauses with $P = |\max(C)|$, $Q = |\max(D)|$, where $\max(C)$ denotes the maximal literal in C.

1. If $Q \prec_L P$ then $D \prec_C C$.
2. If $P = Q$, P occurs negatively in C but only positively in D, then $D \prec_C C$.

Eventually, I overload $\prec$ with $\prec_L$ and $\prec_C$. So if $\prec$ is applied to literals it denotes $\prec_L$, if it is applied to clauses, it denotes $\prec_C$. Note that $\prec$ is a total ordering on literals and clauses as well. Eventually we will restrict inferences to maximal literals with respect to $\prec$. For a clause set N, I define $N^{\prec C} = \{D \in N \mid D \prec C\}$.

Definition 2.8.3 (Abstract Redundancy)**.** A clause C is *redundant* with respect to a clause set N if $N^{\prec C} \models C$.

Tautologies are redundant. Subsumed clauses are redundant if $\subseteq$ is strict. Duplicate clauses are anyway eliminated quietly because the calculus operates on sets of clauses.

C Note that for finite N, and any $C \in N$ redundancy $N^{\prec C} \models C$ can be decided but is as hard as testing unsatisfiability for a clause set N. So the goal is to invent redundancy notions that can be efficiently decided and that are useful.

Definition 2.8.4 (Selection Function)**.** The selection function sel maps clauses to one of its negative literals or $\bot$. If $\mathrm{sel}(C) = \neg P$ then $\neg P$ is called *selected* in C. If $\mathrm{sel}(C) = \bot$ then no literal in C is *selected*.

The selection function is, in addition to the ordering, a further means to restrict superposition inferences. If a negative literal is selected on a clause, any superposition inference must be on the selected literal.

Definition 2.8.5 (Partial Model Construction)**.** Given a clause set N and an ordering $\prec$ we can construct a (partial) model $N_\mathcal{I}$ for N inductively as follows:

$$\begin{array}{lcl} N_C & := & \bigcup_{D \prec C} \delta_D \\ \delta_D & := & \begin{cases} \{P\} & \text{if } D = D' \vee P, P \text{ strictly maximal, no literal} \\ & \text{selected in } D \text{ and } N_D \not\models D \\ \emptyset & \text{otherwise} \end{cases} \\ N_\mathcal{I} & := & \bigcup_{C \in N} \delta_C \end{array}$$

Clauses C with $\delta_C \neq \emptyset$ are called *productive*.

Proposition 2.8.6. Some properties of the partial model construction.

1. For every D with $(C \vee \neg P) \prec D$ we have $\delta_D \neq \{P\}$.
2. If $\delta_C = \{P\}$ then $N_C \cup \delta_C \models C$.
3. If $N_C \models D$ and $D \prec C$ then for all C' with $C \prec C'$ we have $N_{C'} \models D$ and in particular $N_\mathcal{I} \models D$.
4. There is no clause C with $P \vee P \prec C$ such that $\delta_C = \{P\}$.

T

Please properly distinguish: N is a set of clauses interpreted as the conjunction of all clauses. $N^{\prec C}$ is of set of clauses from N strictly smaller than C with respect to $\prec$. $N_\mathcal{I}$, N_C are sets of atoms, often called *Herbrand Interpretations*. $N_\mathcal{I}$ is the overall (partial) model for N, whereas N_C is generated from all clauses from N strictly smaller than C. Validity is defined by $N_\mathcal{I} \models P$ if $P \in N_\mathcal{I}$ and $N_\mathcal{I} \models \neg P$ if $P \notin N_\mathcal{I}$, accordingly for N_C.

Given some clause set N the partial model $N_\mathcal{I}$ can be extended to a valuation $\mathcal{A}$ by defining $\mathcal{A}(N_\mathcal{I}) := N_\mathcal{I} \cup \{\neg P \mid P \notin N_\mathcal{I}\}$. So we can also define for some Herbrand interpretation $N_\mathcal{I}$ (N_C) that $N_\mathcal{I} \models \phi$ iff $\mathcal{A}(N_\mathcal{I})(\phi) = 1$.

Superposition Left $\quad (N \uplus \{C_1 \vee P, C_2 \vee \neg P\}) \Rightarrow_{\text{SUP}} (N \cup \{C_1 \vee P, C_2 \vee \neg P\} \cup \{C_1 \vee C_2\})$
where (i) P is strictly maximal in $C_1 \vee P$ (ii) no literal in $C_1 \vee P$ is selected (iii) $\neg P$ is maximal and no literal selected in $C_2 \vee \neg P$, or $\neg P$ is selected in $C_2 \vee \neg P$

Factoring $\quad (N \uplus \{C \vee P \vee P\}) \Rightarrow_{\text{SUP}} (N \cup \{C \vee P \vee P\} \cup \{C \vee P\})$
where (i) P is maximal in $C \vee P \vee P$ (ii) no literal is selected in $C \vee P \vee P$

Note that the superposition factoring rule differs from the resolution factoring rule in that it only applies to positive literals.

Definition 2.8.7 (Saturation)**.** A set N of clauses is called *saturated up to redundancy*, if any inference from non-redundant clauses in N yields a redundant clause with respect to N.

Examples for specific redundancy rules that can be efficiently decided are

Subsumption provided $C_1 \subset C_2$	$(N \uplus \{C_1, C_2\}) \ \Rightarrow_{\text{SUP}} \ (N \cup \{C_1\})$
Tautology Deletion	$(N \uplus \{C \vee P \vee \neg P\}) \ \Rightarrow_{\text{SUP}} \ (N)$
Condensation	$(N \uplus \{C_1 \vee L \vee L\}) \ \Rightarrow_{\text{SUP}} \ (N \cup \{C_1 \vee L\})$
Subsumption Resolution where $C_1 \subseteq C_2$	$(N \uplus \{C_1 \vee L, C_2 \vee \neg L\}) \ \Rightarrow_{\text{SUP}} \ (N \cup \{C_1 \vee L, C_2\})$

Proposition 2.8.8. All clauses removed by Subsumption, Tautology Deletion, Condensation and Subsumption Resolution are redundant with respect to the kept or added clauses.

Theorem 2.8.9. If N is saturated up to redundancy and $\bot \notin N$ then N is satisfiable and $N_\mathcal{I} \models N$.

Proof. The proof is by contradiction. So I assume: (i) for any clause D derived by Superposition Left or Factoring from N that D is redundant, i.e., $N^{\prec D} \models D$, (ii) $\bot \notin N$ and (iii) $N_\mathcal{I} \not\models N$. Then there is a minimal, with respect to $\prec$, clause $C \vee L \in N$ such that $N_\mathcal{I} \not\models C \vee L$ and L is a selected literal in $C \vee L$ or no literal in $C \vee L$ is selected and L is maximal. This clause must exist because $\bot \notin N$.

The clause $C \vee L$ is not redundant. For otherwise, $N^{\prec C \vee L} \models C \vee L$ and hence $N_\mathcal{I} \models C \vee L$, because $N_\mathcal{I} \models N^{\prec C \vee L}$, a contradiction.

I distinguish the case L is a positive and no literal selected in $C \vee L$ or L is a negative literal. Firstly, assume L is positive, i.e., $L = P$ for some propositional variable P. Now if P is strictly maximal in $C \vee P$ then actually $\delta_{C \vee P} = \{P\}$ and hence $N_\mathcal{I} \models C \vee P$, a contradiction. So P is not strictly maximal. But then actually $C \vee P$ has the form $C_1' \vee P \vee P$ and Factoring derives $C_1' \vee P$ where $(C_1' \vee P) \prec (C_1' \vee P \vee P)$. Now $C_1' \vee P$ is not redundant, strictly smaller than $C \vee L$, we have $C_1' \vee P \in N$ and $N_\mathcal{I} \not\models C_1' \vee P$, a contradiction against the choice that $C \vee L$ is minimal.

Secondly, let us assume L is negative, i.e., $L = \neg P$ for some propositional variable P. Then, since $N_\mathcal{I} \not\models C \vee \neg P$ we know $P \in N_\mathcal{I}$. So there is a clause $D \vee P \in N$ where $\delta_{D \vee P} = \{P\}$ and P is strictly maximal in $D \vee P$ and $(D \vee P) \prec (C \vee \neg P)$. So Superposition Left derives $C \vee D$ where $(C \vee D) \prec (C \vee \neg P)$. The derived clause $C \vee D$ cannot be redundant, because for otherwise either $N^{\prec D \vee P} \models D \vee P$ or $N^{\prec C \vee \neg P} \models C \vee \neg P$. So $C \vee D \in N$ and $N_\mathcal{I} \not\models C \vee D$, a contradiction against the choice that $C \vee L$ is the minimal false clause. □

Propagate $(M;N) \Rightarrow_{\text{DPLL}} (ML;N)$
provided $C \lor L \in N$, $M \models \neg C$, and L is undefined in M
Decide $(M;N) \Rightarrow_{\text{DPLL}} (ML^\top;N)$
provided L is undefined in M
Backtrack $(M_1 L^\top M_2;N) \Rightarrow_{\text{DPLL}} (M_1 \neg L;N)$
provided there is a $D \in N$ and $M \models \neg D$ and no $K^\top$ in M_2

Figure 2.13: The DPLL Calculus

So the proof actually tells us that at any point in time we need only to consider either a superposition left inference between a minimal false clause and a productive clause or a factoring inference on a minimal false clause.

2.9 Davis Putnam Logemann Loveland Procedure (DPLL)

A DPLL problem state is a pair $(M;N)$ where M a sequence of partly annotated literals, and N is a set of clauses. In particular, the following states can be distinguished:

$(\epsilon;N)$	is the start state for some clause set N
$(M;N)$	is a final state, if $M \models N$
$(M;N)$	is a final state, if $M \models \neg N$ and there is no literal $L^\top$ in M
$(M;N)$	is an intermediate state if M neither is a model for N nor does it falsify a clause in N

The sequence M will, by construction, neither contain duplicate nor complementary literals. So it will always serve as a partial valuation for the clause set N.

Here are the rules

Lemma 2.9.1. Let $(M;N)$ be a state reached by the DPLL algorithm from the initial state $(\epsilon;N)$. If $M = M_1 L_1^\top M_2 L_2^\top \dots L_m^\top M_{m+1}$ and all M_i have no decision literals then for all $0 \leq i \leq m$ it holds: $N, M_1, \dots, L_i^\top \models M_{i+1}$

Proof. Proof by complete induction on the number n of rule applications.

Induction basis: $n = 0$. No rule has been applied so that $M = \epsilon$ and M does not contain any decision literal. Therefore the statement holds.

Induction hypothesis: If $(M;N)$ is reached via n or less rule applications where $M = M_1 L_1^\top M_2 L_2^\top \dots L_m^\top M_{m+1}$ and all M_i have no decision literals then for all $1 \leq i \leq m$ it holds: $N, M_1, \dots, L_i^\top \models M_{i+1}$.

Induction step: $n \to n+1$. Assume $(M';N)$ is reached via n rule applications. Then by the use of the induction hypothesis it holds for all $1 \leq i < m$ that

$N, M_1, \ldots, L_i^\top \models M_{i+1}$ so that it remains to be shown that $N, M_1, \ldots, L_m^\top \models M_{m+1}$

1. Rule Propagate $(M'; N) \Rightarrow_{\text{DPLL}} (M'L; N)$: If $M' = M_1 L_1^\top M_2 L_2^\top \ldots L_m^\top M_{m+1}$ and all M_i have no decision literals then by definition there is a clause $C \lor L \in N$ with $M' \models \neg C$, i.e. $C \lor L, M' \models L$ and $N, M_1 L_1^\top M_2 L_2^\top \ldots L_m^\top M_{m+1} \models L$. Using the induction hypothesis it follows $N, M_1 L_1^\top M_2 L_2^\top \ldots L_m^\top \models M_{m+1}, L$.

2. Rule Decide $(M'; N) \Rightarrow_{\text{DPLL}} (M' L^\top; N)$: The statement holds because of $M', L^\top \models \top$ and the induction hypothesis.

3. Rule Backtrack $(M_1' L^\top M_2'; N) \Rightarrow_{\text{DPLL}} (M_1' \neg L; N)$: By definition M_2' has no decision literals and there is a clause $D \in N$ with $M_1' L^\top M_2' \models \neg D$. With the induction hypothesis $M_1' L^\top \models M_2'$ holds. It follows that $M_1' L^\top \models \neg D$ which is equivalent to $M_1' L^\top, D \models \bot$ and $M_1', D \models \neg L^\top$. Since $D \in N$ it holds that $N, M_1' \models \neg L$. Let $M_1' = M_1 L_1^\top M_2 L_2^\top \ldots L_m^\top M_{m+1}$ where all M_i have no decision literals then by induction hypothesis $N, M_1 L_1^\top M_2 L_2^\top \ldots L_m^\top \models M_{m+1}, \neg L$.

□

Proposition 2.9.2. For a state $(M; N)$ that is reached from the initial state $(\epsilon; N)$ where M contains k decision literals $L_1 \ldots L_k$ with $k \geq 0$ and for each valuation $\mathcal{A}$ with $\mathcal{A} \models N, L_1, \ldots, L_k$ it holds that $\mathcal{A}(K) = 1$ for all $K \in M$.

Proof. Let $M = M_1 L_1^\top \ldots L_k^\top M_{k+1}$ where all M_i have no decision literals. With Lemma 2.9.1 for all i it holds that $N, M_1 L_1^\top \ldots L_{i-1}^\top \models M_i$, i.e., for all i, literals $K \in M_i$ and each valuation $\mathcal{A}$ with $\mathcal{A} \models N, L_1, \ldots, L_k$ it holds that $\mathcal{A}(K) = 1$. □

Lemma 2.9.3. If M contains only propagated literals and $M = L_1 \ldots L_n$ and there is a $D \in N$ with $M \models \neg D$ where $D = K_1 \ldots K_m$ then N is unsatisfiable.

Proof. Since $M \models \neg D$ it holds that $\neg K_i \in M$ for all $1 \leq i \leq m$. With Proposition 2.9.2 for each valuation $\mathcal{A}$ with $\mathcal{A} \models N$ it holds that $\mathcal{A}(L_j) = 1$ for all $1 \leq j \leq n$. Thus in particular it holds that $\mathcal{A}(\neg K_i) = 1$ for all $1 \leq i \leq m$. Therefore D is always false under any valuation $\mathcal{A}$ and N is always unsatisfiable. □

Proposition 2.9.4 (DPLL Soundness)**.** The rules Propagate, Decide, and Backtrack are sound, i.e. whenever the algorithm terminates in state $(M; N)$ starting from the initial state $(\epsilon; N)$ then it holds: $M \models N$ iff N is satisfiable

Proof. $(\Rightarrow)$ if $M \models N$ then obviously N is satisfiable.

$(\Leftarrow)$ Proof by contradiction. Assume N is satisfiable and the algorithm terminates in state $(M; N)$ starting from the initial state $(\epsilon; N)$. Furthermore, assume $M \models N$ does not hold, i.e. either there is at least one literal that is not defined in M or there is a clause $D \in N$ with $M \models \neg D$.

For the first case the rule Decide is applicable. This contradicts that the algorithm terminated.

For the second case either M only contains propagated literals then N is unsatisfiable with Lemma 2.9.3. This is a contradiction to the assumption that N is satisfiable. If M does not only contain propagated literals there must be at least one decision literal in M. Then the rule Backtrack is applicable but this contradicts that the algorithm terminated.
Therefore $M \models N$ and the rules Propagate, Decide, and Backtrack are sound. □

Proposition 2.9.5 (DPLL Strong Completeness)**.** The rules Propagate, Decide, and Backtrack are strongly complete: for any valuation M with $M \models N$, there is a sequence of rule application generating (M, N) as a final state.

Proof. Let $M = L_1 L_2 \dots L_k$. Since it is a valuation there are no duplicates in M and k applications of rule Decide yield $(L_1^\top L_2^\top \dots L_k^\top, N)$ out of $(\epsilon; N)$. This is a final state because backtrack is not applicable since $M \models N$ and Propagate and Decide are no further applicable since M is a valuation. □

Proposition 2.9.6 (DPLL Termination)**.** The rules Propagate, Decide, and Backtrack terminate on any input state (ϵ, N).

Proof. Let n be the number of propositional variables in N. As usual, termination is shown by assigning a well-founded measure and proving that it decreases with each rule application. The domain for the measure μ are n-tuples over $\{1, 2, 3\}$.

$$\mu((L_1 \dots L_k; N)) = (m_1, \dots, m_k, 3, \dots, 3)$$

where $m_i = 2$ if L_i is annotated with $\top$ and $m_i = 1$ otherwise. So $\mu((\epsilon, N)) = (3, \dots, 3)$. The well-founded ordering is the lexicographic extension of $<$ to n-tuples. What remains to be shown is that each rule application decreases μ. I do this by a case analysis over the rules.

Propagate:

$$\begin{aligned}\mu((L_1 \dots L_k; N)) &= (m_1, \dots, m_k, 3, 3, \dots, 3)\\ &> (m_1, \dots, m_k, 1, 3, \dots, 3)\\ &= \mu((L_1 \dots L_k L; N))\end{aligned}$$

Decide:

$$\begin{aligned}\mu((L_1 \dots L_k; N)) &= (m_1, \dots, m_k, 3, 3, \dots, 3)\\ &> (m_1, \dots, m_k, 2, 3, \dots, 3)\\ &= \mu((L_1 \dots L_k L^\top; N))\end{aligned}$$

Backtrack:

$$\begin{aligned}\mu((L_1 \dots L_j L^\top L_{j+1} \dots L_k; N)) &= (m_1, \dots, m_j, 2, m_{j+1}, \dots, m_k, 3, \dots, 3)\\ &> (m_1, \dots, m_j, 1, 3, \dots, 3)\\ &= \mu((L_1 \dots L_j \neg L; N))\end{aligned}$$

□

2.10 Conflict Driven Clause Learning (CDCL)

A CDCL problem state is a five-tuple $(M; N; U; k; C)$ where M a sequence of annotated literals, N and U are sets of clauses, $k \in \mathbb{N}$, and C is a non-empty clause or $\top$ or $\bot$. In particular, the following states can be distinguished:

$(\epsilon; N; \emptyset; 0; \top)$	is the start state for some clause set N
$(M; N; U; k; \top)$	is a final state, if $M \models N$ and all literals from N are defined in M
$(M; N; U; k; \bot)$	is a final state, where N has no model
$(M; N; U; k; \top)$	is an intermediate model search state if $M \not\models N$
$(M; N; U; k; D)$	is a backtracking state if $D \notin \{\top, \bot\}$

A literal L is of *level* k with respect to a problem state $(M; N; U; j; C)$ if L or $\neg L$ occurs in M and the first decision literal left from L $(\neg L)$ in M is annotated with k or if there is no decision literal $k = 0$. A clause D is of *level* k with respect to a problem state $(M; N; U; j; C)$ if k is the maximal level of a literal in D. Recall C is a non-empty clause or $\top$ or $\bot$. The rules are

Propagate $(M; N; U; k; \top) \Rightarrow_{\text{CDCL}} (ML^{C\lor L}; N; U; k; \top)$
provided $C \lor L \in (N \cup U)$, $M \models \neg C$, and L is undefined in M

Decide $(M; N; U; k; \top) \Rightarrow_{\text{CDCL}} (ML^{k+1}; N; U; k+1; \top)$
provided L is undefined in M

Conflict $(M; N; U; k; \top) \Rightarrow_{\text{CDCL}} (M; N; U; k; D)$
provided $D \in (N \cup U)$ and $M \models \neg D$

Skip $(ML^{C\lor L}; N; U; k; D) \Rightarrow_{\text{CDCL}} (M; N; U; k; D)$
provided $D \notin \{\top, \bot\}$ and $\neg L$ does not occur in D

Resolve $(ML^{C\lor L}; N; U; k; D \lor \neg L) \Rightarrow_{\text{CDCL}} (M; N; U; k; D \lor C)$
provided D contains a literal of level k or $k = 0$

For rule Resolve we assume that duplicate literals in $D \lor C$ are always removed.

Backtrack $(M_1 K^{i+1} M_2; N; U; k; D \lor L) \Rightarrow_{\text{CDCL}} (M_1 L^{D\lor L}; N; U \cup \{D \lor L\}; i; \top)$
provided L is of maximal level k in $D \lor L$ and D is of level i, where $i < k$.

Restart $(M; N; U; k; \top) \Rightarrow_{\text{CDCL}} (\epsilon; N; U; 0; \top)$
provided $M \not\models N$

Forget $(M; N; U \cup \{C\}; k; \top) \Rightarrow_{\text{CDCL}} (M; N; U; k; \top)$

provided $M \not\models N$

Recall that $\bot$ denotes the empty clause, hence failure of searching for a model. The level of the empty clause $\bot$ is 0. The clause $D \lor L$ added in rule Backtrack to U is called a *learned* clause. The CDCL algorithm stops with a model M if neither Propagate nor Decide nor Conflict are applicable to a state $(M; N; U; k; \top)$, hence $M \models N$ and all literals of N are defined in M. The only possibility to generate a state $(M; N; U; k; \bot)$ is by the rule Resolve. So in case of detecting unsatisfiability the CDCL algorithm actually generates a resolution proof as a certificate. I will discuss this aspect in more detail in Section 2.12. In the special case of a unit clause L, the rule Propagate actually annotates the literal L with itself.

Obviously, the CDCL rule set does not terminate in general for a number of reasons. For example, starting with $(\epsilon; N; \emptyset; 0; \top)$ a simple combination Propagate, Decide and eventually Restart yields the start state again. Even after a successful application of Backtrack, exhaustive application of Forget followed by Restart again produces the start state. So why these rules? Actually, any modern SAT solver is based on this rule set and the underlying mechanisms. I will motivate the rules later on and how they are actually used in an efficient way.

Example 2.10.1 (CDCL Strategy I)**.** Consider the clause set $N = \{P \lor Q, \neg P \lor Q, \neg Q\}$ which is unsatisfiable. The below is a CDCL derivation proving this fact. The chosen strategy for CDCL rule selection produces a lengthy proof.

$(\epsilon; N; \emptyset; 0; \top)$
$\Rightarrow^{\text{Decide}}_{\text{CDCL}} (P^1; N; \emptyset; 1; \top)$
$\Rightarrow^{\text{Decide}}_{\text{CDCL}} (P^1 \neg Q^2; N; \emptyset; 2; \top)$
$\Rightarrow^{\text{Conflict}}_{\text{CDCL}} (P^1 \neg Q^2; N; \emptyset; 2; \neg P \lor Q)$
$\Rightarrow^{\text{Backtrack}}_{\text{CDCL}} (P^1 Q^{\neg P \lor Q}; N; \{\neg P \lor Q\}; 1; \top)$
$\Rightarrow^{\text{Conflict}}_{\text{CDCL}} (P^1 Q^{\neg P \lor Q}; N; \{\neg P \lor Q\}; 1; \neg Q)$
$\Rightarrow^{\text{Backtrack}}_{\text{CDCL}} (\neg Q^{\neg Q}; N; \{\neg P \lor Q, \neg Q\}; 0; \top)$
$\Rightarrow^{\text{Decide}}_{\text{CDCL}} (\neg Q^{\neg Q} P^1; N; \{\neg P \lor Q, \neg Q\}; 1; \top)$
$\Rightarrow^{\text{Conflict}}_{\text{CDCL}} (\neg Q^{\neg Q} P^1; N; \{\neg P \lor Q, \neg Q\}; 1; \neg P \lor Q)$
$\Rightarrow^{\text{Backtrack}}_{\text{CDCL}} (\neg Q^{\neg Q} \neg P^{\neg P \lor Q}; N; \{\neg P \lor Q, \neg Q\}; 0; \top)$
$\Rightarrow^{\text{Conflict}}_{\text{CDCL}} (\neg Q^{\neg Q} \neg P^{\neg P \lor Q}; N; \{\neg P \lor Q, \neg Q\}; 0; P \lor Q)$
$\Rightarrow^{\text{Resolve}}_{\text{CDCL}} (\neg Q^{\neg Q}; N; \{\neg P \lor Q, \neg Q\}; 0; Q)$
$\Rightarrow^{\text{Resolve}}_{\text{CDCL}} (\epsilon; N; \{\neg P \lor Q, \neg Q\}; 0; \bot)$

Example 2.10.2 (CDCL Strategy II)**.** Consider again the clause set $N = \{P \lor Q, \neg P \lor Q, \neg Q\}$ from Example 2.10.1. For the following CDCL derivation the rules Propagate and Conflict are preferred over the other rules.

$$
\begin{array}{ll}
(\epsilon; N; \emptyset; 0; \top) & \\
\Rightarrow^{\text{Propagate}}_{\text{CDCL}} & (\neg Q^{\neg Q}; N; \emptyset; 0; \top) \\
\Rightarrow^{\text{Propagate}}_{\text{CDCL}} & (\neg Q^{\neg Q} P^{Q \lor P}; N; \emptyset; 0; \top) \\
\Rightarrow^{\text{Conflict}}_{\text{CDCL}} & (\neg Q^{\neg Q} P^{Q \lor P}; N; \emptyset; 0; \neg P \lor Q) \\
\Rightarrow^{\text{Resolve}}_{\text{CDCL}} & (\neg Q^{\neg Q}; N; \emptyset; 0; Q) \\
\Rightarrow^{\text{Resolve}}_{\text{CDCL}} & (\epsilon; N; \emptyset; 0; \bot)
\end{array}
$$

I

In an implementation the rule Conflict is preferred over the rule Propagate and both over all other rules. Exactly this strategy has been used in Example 2.10.2 and is called *reasonable* below. A further ingredient is a dynamic heuristic which literal is actually used by the rule Decide. This heuristic typically depends on the usage of literals by the rule Resolve, i.e., literals used in Resolve "get a bonus".

Definition 2.10.3 (Reasonable CDCL Strategy)**.** A CDCL strategy is *reasonable* if Conflict is always preferred over rule Propagate is always preferred over all other rules.

Proposition 2.10.4 (CDCL Basic Properties)**.** Consider a CDCL state $(M; N; U; k; C)$ derived by a reasonable strategy from start state $(\epsilon, N, \emptyset, 0, \top)$ without using the rules Restart and Forget. Then the following properties hold:

1. M is consistent.
2. All learned clauses are entailed by N.
3. If $C \notin \{\top, \bot\}$ then $M \models \neg C$.
4. If $C = \top$ and M contains only propagated literals then for each valuation $\mathcal{A}$ with $\mathcal{A} \models N$ it holds that $\mathcal{A} \models M$.
5. If $C = \top$, M contains only propagated literals and $M \models \neg D$ for some $D \in (N \cup U)$ then N is unsatisfiable.
6. If $C = \bot$ then CDCL terminates and N is unsatisfiable.
7. Each infinite derivation

$$(\epsilon; N; \emptyset; 0; \top) \Rightarrow_{\text{CDCL}} (M_1; N; U_1; k_1; D_1) \Rightarrow_{\text{CDCL}} \dots$$

   contains an infinite number of Backtrack applications.
8. CDCL never learns the same clause twice if Conflict selects the smallest clause out of $N \cup U$.

Proof. 1. M is consistent if it does does not contain L and $\neg L$ at the same time. The rules Propagate, Decide only add undefined literals to M. By an inductive

argument this holds also for Backtrack as it just removes literals from M and flips one literal already contained in M.

2. A learned clause is a always a resolvent of clauses from $N \cup U$ and eventually added to U where U is initially empty. By soundness of resolution (Theorem 2.7.1) and an inductive argument it is entailed by N.

3. A clause $C \notin \{\top, \bot\}$ can only occur after Conflict where $M \models \neg C$. The rule Skip does not change C and only deletes propagated literals from M that are not contained in C. By an inductive argument, if the rule Resolve is applied to a state $(M'L^{D'\vee L}; N; U; k; D \vee \neg L)$ where $C = D \vee \neg L$ resulting in $(M'; N; U; k; D \vee D')$ then $M' \models \neg D$ because $M' \models \neg C$ and $M' \models \neg D'$ because L was propagated with respect to M' and $D' \vee L$.

4. Proof by induction on the number n of propagated literals in M. Let $M = L_1, \ldots, L_n, L_{n+1}$. There are two rules that could have added L_{n+1}. (i) rule Propagate: in this case there is a clause $C = D \vee L_{n+1}$ where L_{n+1} was undefined in M and $M \models \neg D$. By induction hypothesis for each valuation $\mathcal{A}$ with $\mathcal{A} \models N$ it holds that $\mathcal{A}(L_i) = 1$ for all $i \in \{1, \ldots, n\}$. Since all literals in D appear negated in M with the induction hypothesis it holds that all those literals must have the truth value 1 in any valuation $\mathcal{A}$. Therefore, for the clause C to be true L_{n+1} must be true as well in any valuation. It follows that for each valuation $\mathcal{A}$ it holds that $\mathcal{A}(L_i) = 1$ for all $i \in \{1, \ldots, n+1\}$. (ii) rule Backtrack: the state $(M_1 K^{i+1} M_2; N; U; k; D \vee L^k_{n+1})$ where $M \models \neg(D \vee L^k_{n+1})$ (with Proposition 2.10.4-3) and $M_1 = L_1 \ldots L_n$ with only propagated literals becomes $(M_1 L_{n+1}^{D\vee L_{n+1}}; N; U; i; \top)$. With the induction hypothesis for each valuation $\mathcal{A}$ with $\mathcal{A} \models N$ it holds that $\mathcal{A}(L_i) = 1$ for all $1 \leq i \leq n$ i.e. in particular it holds that for each literal L in D $\mathcal{A}(L) = 0$ since each literal in D appears negated in M_1. Thus, for each each valuation $\mathcal{A}$ with $\mathcal{A} \models N$ $\mathcal{A}(L_{n+1}) = 1$ holds.

5. Since $M \models \neg D$ it holds that $\neg K_i \in M$ for all $1 \leq i \leq m$. With Proposition 2.10.4-4 for each valuation $\mathcal{A}$ with $\mathcal{A} \models N$ it holds that $\mathcal{A}(L_j) = 1$ for all $1 \leq j \leq n$. Thus in particular it holds that $\mathcal{A}(\neg K_i) = 1$ for all $1 \leq i \leq m$. Therefore D is always false under any valuation $\mathcal{A}$ and N is always unsatisfiable.

6. By the definition of the rules the state $(M; N; U; k; \bot)$ can only be reached if the rule Conflict has been applied to set some conflict clause C of a state $(M'; N; U; k; \top)$ as the last component and Resolve is used in the last rule application to derive $\bot$. Before the last call of Resolve the state had the following form $(ML^{\bot\vee L}; N; U; k; \neg L)$ otherwise $\bot$ could not be derived. M cannot contain any decision literal because L is a propagated literal and due to the strategy the rule Propagate is applied before the rule Decide. With Proposition 2.10.4-5 it follows that N is unsatisfiable.

7. Proof by contradiction. Assume Backtrack is applied only finitely often in the infinite trace. Then there exists an $i \in \mathbb{N}^+$ with $R_j \neq$ Backtrack for all $j > i$. Propagate and Decide can only be applied as long as there are undefined literals in M. Since there is only a finite number of propositional variables they can only be applied finitely often.

By definition the application of the rules Skip, Resolve and Backtrack is preceded by an application of the rule Conflict since the initial state has a $\top$ as the last component and Conflict is the only rule that replaces the last component by a clause. For the rule Conflict to be applied infinitely often the last component has to change to $\top$. By definition that can only be performed by the rules Resolve and Backtrack (a contradiction to the assumption). For Resolve assume the following rule application $(ML^{C\lor L};N;U;k;D\lor\neg L) \Rightarrow_{\text{CDCL}}$ $(M;N;U;k;D \lor C)$. For $D \lor C = \top$ there must be a literal K with $K, \neg K \in (D \lor C)$. With Proposition 2.10.4-3 $M \models \neg(D \lor C)$ holds which is equivalent to $M \models \bot$,a contradiction because of Proposition 2.10.4-1. Therefore Conflict is applied finitely often.

Skip and Resolve are also applied finitely often since Conflict is applied finitely often and they cannot be applied infinitely often interchangeably. Otherwise the first component M has to be of infinite length, a contradiction.

8. By Proposition 2.12.4. □

Lemma 2.10.5. Assume the algorithm CDCL with all rules is applied using the strategy *eager application of Conflict and Propagate where Conflict is applied before Propagate.* The CDCL algorithm has only 2 termination states: $(M;N;U;k;\top)$ where $M \models N$ and $(M;N;U;k;\bot)$ where N is unsatisfiable.

Proof. Let the CDCL algorithm terminate in a state $(M;N;U;k;\phi)$ starting from the initial state $(\epsilon;N;\emptyset;0;\top)$.

1. Let $\phi = \bot$. No rule can be applied and $(M;N;U;k;\bot)$ is indeed a termination state. With Proposition 2.10.4-6 it also holds that N is unsatisfiable.

2. Let $\phi = \top$ and $M \models N$. Then the algorithm found a total valuation M for N and no literal in N is undefined in M (otherwise we could apply Decide, contradicting that the algorithm terminated). Since $M \models N$ there can also be no conflict clause D. Hence, no further rule can be applied and the state $(M;N;U;k;\top)$ where $M \models N$ is a termination state.

3. Let $\phi = \top$ and $M \models N$ does not hold. Since $M \models N$ does not hold there is either a clause $D \in N$ with $M \models \neg D$ or there is no such clause D but there is a literal in N that is undefined in M. For the first case the rule Conflict is applicable and for the second case the rule Decide is applicable. Thus, for both cases it holds that $(M;N;U;k;\top)$ is not a termination state, a contradiction.

4. Let ϕ be a clause $C = D \lor L$. With Proposition 2.10.4-3 the clause C must be a conflicting clause where $M \models \neg C$.

   If the rightmost literal in M is a propagated literal then the rules Skip or Resolve are applicable if their conditions are satisfied. This would contradict that the algorithm terminated. The case that the conditions are not satisfied is handled in a similar way as the decided literal case.

If the rightmost literal is a decision literal L then L is contained in C. This is due to the fact that with the assumed strategy before deciding literal L (via the rule Decide) neither Propagate nor Conflict were applicable. Thus, L is of maximal level k and the remaining part of C can only be of a level i with $i < k$. The same holds for the case that the rightmost literal is a propagated literal but D does not contain a literal of level k and Skip is also not applicable. Then D must again be of a level i with $i < k$ and L must be the literal of level k in C (otherwise, due to the strategy, the rule Conflict would have been called before the rule Propagate and the rightmost literal in M could not be the propagated literal L). Therefore, in both cases the rule Backtrack is applicable, contradicting that the algorithm terminated.

□

Proposition 2.10.6 (CDCL Soundness)**.** Assume the algorithm CDCL with all rules is applied using the strategy *eager application of Conflict and Propagate where Conflict is applied before Propagate*. The rules of the CDCL algorithm are sound, i.e. whenever the algorithm terminates in state $(M; N; U; k; \phi)$ starting from the initial state $(\epsilon; N; \emptyset; 0; \top)$ then it holds that $M \models N$ iff N is satisfiable.

Proof. ($\Rightarrow$) if $M \models N$ and M is consistent with Proposition 2.10.4-1 then N is satisfiable.

($\Leftarrow$) Proof by contradiction. Assume N is satisfiable and the algorithm terminates in state $(M; N; U; k; \phi)$ starting from the initial state $(\epsilon; N; \emptyset; 0; \top)$. Furthermore, assume $M \models N$ does not hold. With Lemma 2.10.5 there are only 2 termination states, i.e. ϕ can only be $\top$ or $\bot$.

Case $\phi = \top$ then by Lemma 2.10.5 $M \models N$. This is a contradiction to the assumption that $M \models N$ does not hold.

Case $\phi = \bot$ then by Lemma 2.10.5 N is unsatisfiable. This is a contradiction to N being satisfiable. □

Therefore all rules of the CDCL algorithm are sound.

Proposition 2.10.7 (CDCL Completeness)**.** The CDCL rule set is complete: for any valuation M with $M \models N$ there is a sequence of rule application generating $(M; N; U; k; \top)$ as a final state.

Proof. Let $M = L_1 L_2 \dots L_k$. Since M is a valuation there are no duplicates in M and k applications of rule Decide yield $(L_1^1 L_2^2 \dots L_k^k; N; \emptyset; k; \top)$ out of $(\epsilon; N; \emptyset; 0; \top)$. Since $M \models N$ this is a final state and all literals from N are defined in M. The rules Propagate and Decide cannot be applied anymore and there is no conflict because $M \models N$. Therefore Conflict, Skip, Resolve and Backtrack are not applicable. The rule Forget is not applicable since $U = \emptyset$ and there is no need to restart. □

C As an alternative proof of Proposition 2.10.7 the strategy of an alternation of an exhaustive application of Propagate and one application of Decide produces $(M; N; \emptyset; i; \top)$ as a final state where $M \models N$. As in the proof of Proposition 2.10.7 let $M = L_1 L_2 \dots L_k$. First apply Propagate m-times exhaustively resulting in $(L_1 \dots L_m; N; \emptyset; 0; \top)$ where $m \leq k$. With Proposition 2.10.4-4 the literals $L_1 \dots L_m$ must be true in any valuation $\mathcal{A}$ with $\mathcal{A} \models N$. Thus, if $m = k$ then $(L_1 \dots L_m; N; \emptyset; 0; \top)$ is a final state and $M \models N$. If $m < k$ then apply Decide once on a literal from M resulting in $(L_1 \dots L_m L^1; N; \emptyset; 1; \top)$. Since L^1 is contained in M it must be true. This strategy can be applied equivalently to all further literals in M resulting in the desired state.

Proposition 2.10.8 (CDCL Termination)**.** Assume the algorithm CDCL with all rules except Restart and Forget is applied using the strategy *eager application of Conflict and Propagate where Conflict is applied before Propagate.* Then it terminates in a state $(M; N; U; k; D)$ with $D \in \{\top, \bot\}$.

Proof. Proof by contradiction. Assume there is an infinite trace that starts in a state $(M'; N; U'; k'; D')$. With Proposition 2.10.4-**??** and 2.10.4-8 there can only be a finite number of clauses that are learned during the infinite run. By definition of the rules only the rule Backtrack causes that a clause is learned so that the rule Backtrack can only be applied finitely often. But with Proposition 2.10.4-7 the rule Backtrack must be applied infinitely often, a contradiction. Therefore there does not exist an infinite trace, i.e. the algorithm always terminates under the given assumptions. ◻

The CDCL rule set does not in general terminate. This is due to the rules Restart and Forget. If they are applied only finitely often then the algorithm terminates. At some point the last application of Restart and Forget was reached since they are only applied finitely often. From this point onwards Proposition 2.10.8 can be applied and the algorithm eventually terminates.

Example 2.10.9 (CDCL Termination I)**.** Consider the clause set $N = \{P \lor Q, \neg P \lor Q, \neg Q\}$. The CDCL algorithm does not terminate due to the rule Restart.

$$
\begin{array}{ll}
(\epsilon; N; \emptyset; 0; \top) & \\
\Rightarrow^{\text{Propagate}}_{\text{CDCL}} & (\neg Q^{\neg Q}; N; \emptyset; 0; \top) \\
\Rightarrow^{\text{Propagate}}_{\text{CDCL}} & (\neg Q^{\neg Q} P^{Q \lor P}; N; \emptyset; 0; \top) \\
\Rightarrow^{\text{Restart}}_{\text{CDCL}} & (\epsilon; N; \emptyset; 0; \top) \\
\Rightarrow^{\text{Propagate}}_{\text{CDCL}} & (\neg Q^{\neg Q}; N; \emptyset; 0; \top) \\
\Rightarrow^{\text{Propagate}}_{\text{CDCL}} & (\neg Q^{\neg Q} P^{Q \lor P}; N; \emptyset; 0; \top) \\
\Rightarrow^{\text{Restart}}_{\text{CDCL}} & (\epsilon; N; \emptyset; 0; \top) \\
\Rightarrow_{\text{CDCL}} & \dots
\end{array}
$$

Example 2.10.10 (CDCL Termination II)**.** Consider the clause set $N = \{\neg P \lor Q \lor \neg R, \neg P \lor Q \lor R\}$. The CDCL algorithm does not terminate due to the rule Forget.

$$
\begin{array}{ll}
(\epsilon; N; \emptyset; 0; \top) & \\
\Rightarrow^{\text{Decide}}_{\text{CDCL}} & (P^1; N; \emptyset; 1; \top) \\
\Rightarrow^{\text{Decide}}_{\text{CDCL}} & (P^1 \neg Q^2; N; \emptyset; 2; \top) \\
\Rightarrow^{\text{Propagate}}_{\text{CDCL}} & (P^1 \neg Q^2 \neg R^{\neg P \vee Q \vee \neg R}; N; \emptyset; 2; \top) \\
\Rightarrow^{\text{Conflict}}_{\text{CDCL}} & (P^1 \neg Q^2 \neg R^{\neg P \vee Q \vee \neg R}; N; \emptyset; 2; \neg P \vee Q \vee R) \\
\Rightarrow^{\text{Resolve}}_{\text{CDCL}} & (P^1 \neg Q^2; N; \emptyset; 2; \neg P \vee Q) \\
\Rightarrow^{\text{Backtrack}}_{\text{CDCL}} & (P^1; N; \{\neg P \vee Q\}; 1; \top) \\
\Rightarrow^{\text{Forget}}_{\text{CDCL}} & (P^1; N; \emptyset; 1; \top) \\
\Rightarrow^{\text{Decide}}_{\text{CDCL}} & (P^1 \neg Q^2; N; \emptyset; 2; \top) \\
\Rightarrow^{\text{Propagate}}_{\text{CDCL}} & (P^1 \neg Q^2 \neg R^{\neg P \vee Q \vee \neg R}; N; \emptyset; 2; \top) \\
\Rightarrow^{\text{Conflict}}_{\text{CDCL}} & (P^1 \neg Q^2 \neg R^{\neg P \vee Q \vee \neg R}; N; \emptyset; 2; \neg P \vee Q \vee R) \\
\Rightarrow_{\text{CDCL}} & \ldots
\end{array}
$$

As an alternative for the proof of Proposition 2.10.8 the termination can be shown by assigning a well-founded measure μ and proving that it decreases with each rule application except for the rules Restart and Forget. Let n be the number of propositional variables in N. The domain for the measure μ is $\mathbb{N} \times \{0, 1\} \times \mathbb{N}$.

$$
\mu((M; N; U; k; D)) = \begin{cases} (3^n - 1 - |U|, 1, n - |M|) & , D = \top \\ (3^n - 1 - |U|, 0, |M|) & , else \end{cases}
$$

The well-founded ordering is the lexicographic extension of $<$ to triples. What remains to be shown is that each rule application except Restart and Forget decreases μ. This is done via a case analysis over the rules:

Propagate:

$$
\begin{aligned}
\mu((M; N; U; k; \top)) &= (3^n - 1 - |U|, 1, n - |M|) \\
&> (3^n - 1 - |U|, 1, n - |ML^{C \vee L}|) \\
&= \mu((ML^{C \vee L}; N; U; k; \top))
\end{aligned}
$$

Decide:

$$
\begin{aligned}
\mu((M; N; U; k; \top)) &= (3^n - 1 - |U|, 1, n - |M|) \\
&> (3^n - 1 - |U|, 1, n - |ML^{k+1}|) \\
&= \mu((ML^{k+1}; N; U; k; \top))
\end{aligned}
$$

Conflict:

$$
\begin{aligned}
\mu((M; N; U; k; \top)) &= (3^n - 1 - |U|, 1, n - |M|) \\
&> (3^n - 1 - |U|, 0, |M|) \\
&= \mu((M; N; U; k; D))
\end{aligned}
$$

Skip:

$$
\begin{aligned}
\mu((ML^{C \vee L}; N; U; k; D)) &= (3^n - 1 - |U|, 0, |ML^{C \vee L}|) \\
&> (3^n - 1 - |U|, 0, |M|) \\
&= \mu((M; N; U; k; D))
\end{aligned}
$$

Resolve:

$$\begin{aligned}\mu((ML^{C\vee L};N;U;k;D\vee\neg L)) &= (3^n-1-|U|,0,|ML^{C\vee L}|)\\ &> (3^n-1-|U|,0,|M|)\\ &= \mu((M;N;U;k;D\vee C))\end{aligned}$$

Backtrack: with Proposition 2.10.4-8 it holds that $D \vee L \notin U$ so that the first component decreases.

$$\begin{aligned}\mu((M_1K^{i+1}M_2;N;U;k;D\vee L)) &= (3^n-1-|U|,0,|M_1K^{i+1}M_2|)\\ &> (3^n-1-|U\cup\{D\vee L\}|,1,n-|M_1L^{D\vee L}|\\ &= \mu((M_1L^{D\vee L};N;U\cup\{D\vee L\};i;\top))\end{aligned}$$

2.11 Implementing CDCL

For an effective CDCL implementation the underlying data structure of the implementation plays a crucial part. The technique that proved to be very successful in modern SAT solvers and that is also used in a CDCL implementation is the *2-watched literals* data structure. For choosing the decision variables a special heuristic plays an important role in the implementation as well. This heuristic is called *VSIDS* (Variable State Independent Decaying Sum) that works on natural numbers. Furthermore, the decision for choosing the most reasonable clause to be learned after a discovered conflict is handled by the notion of *UIPs* (Unique Implication Points). In the following these main concepts (2-watched literals, VSIDS and 1UIP scheme) will be introduced in accordance with the CDCL rule set.

2.11.1 Lazy Data Structure: 2-Watched Literals (2WL)

For applying the rule Propagate, the number of literals in each clause that are not false need to be known. Maintaining this number is expensive, however, since it has to be updated whenever Backtrack is applied. Therefore, the better approach is to use a more efficient representation called *2-watched literals*. A list as represented in Figure 2.14 has references for each variable P to clauses where P occurs positive and references to clauses where P occurs negative. A variable is either unassigned, true or false. For each clause within the clause list 2 watched (unassigned) variables are maintained. The way of working with the watched literals is as follows:

1. Let an unassigned variable P be set to false (or true).
2. Visit all clauses in which P (or $\neg P$) is watched.
3. In every clause where P (or $\neg P$) is watched find an unwatched and non-falsified variable to be watched. If there is no other unassigned or true variable then this clause is either a unit clause and the rule Propagate can be applied or there is a conflict and the rule Backtrack is applied or the clause set is already satisfied.

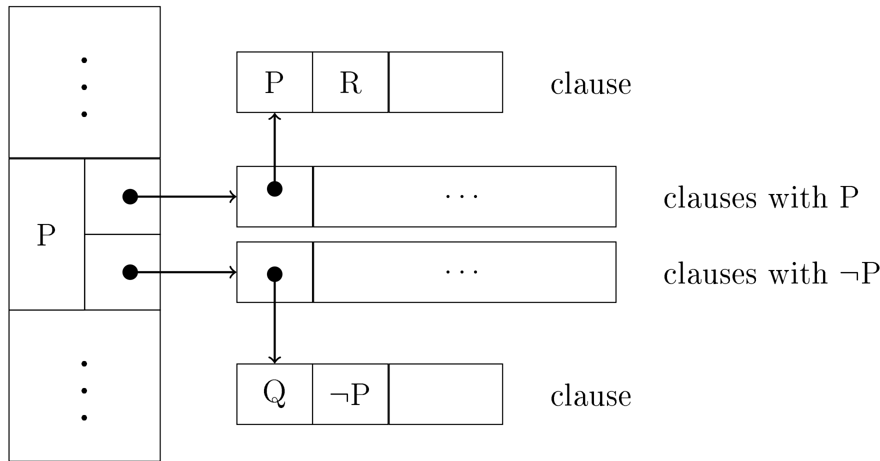


Figure 2.14: The watched literals list with the variables P, Q, R and the watched literals P, R and $\neg P$, Q.

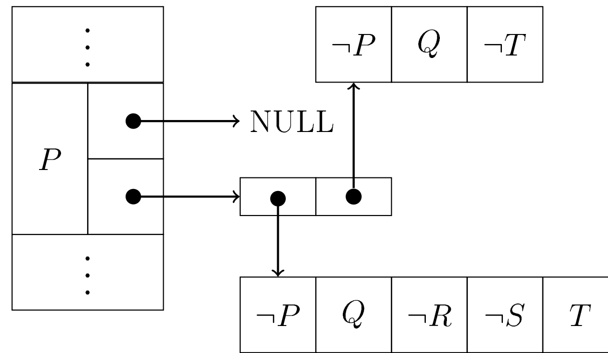
An advantage of the data structure as shown in the example below is no extra cost for variables that are not watched (but assigned false).

As an example consider the formula $\phi = \{\neg P \vee Q \vee \neg R \vee \neg S \vee T, \neg P \vee Q \vee \neg T, R \vee T, S \vee T\}$. Figure 2.17 shows how to derive unit clauses and finally satisfy the formula within the watched literals data structure. The watched literals are the first two entries in a clause. The trail (see next section on Backtracking) represents the assigned literals for the current state.

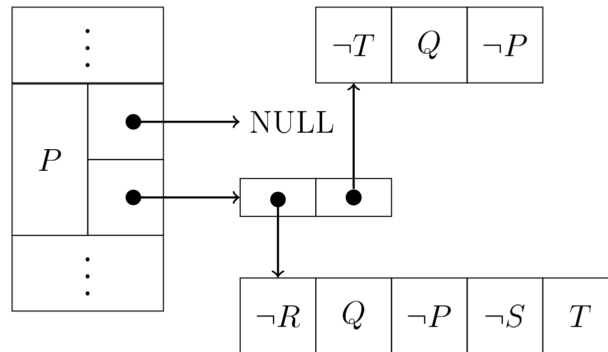
2.11.2 Backtracking

Another main advantage of the 2-watched literals data structure is discovered when considering backtracking. For this purpose a *trail*, a *decision level* and a *control stack* are maintained together with the watched literals data structure. The *trail* is a stack of variables that stores the order in which the variables are assigned. The *decision level* counts the number of calls of the rule Decide. The *control stack* stores the trail height for each decision level, i.e. once Decide is applied the control stack increases by one entry and saves the height of the previous trail stack.

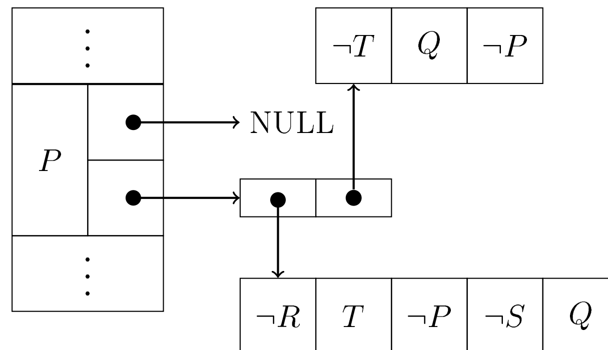
If the rule Backtrack is applied the trail height entry from the control stack is taken and every variable from that trail height on will be unassigned, i.e. every assignment value that was made since the last application of the rule Decide is deleted. A detailed example is shown in Figure 2.18. Again, the advantage with the watched literals data structure is that the watched variables stay unchanged and will not be considered by this backtracking step.



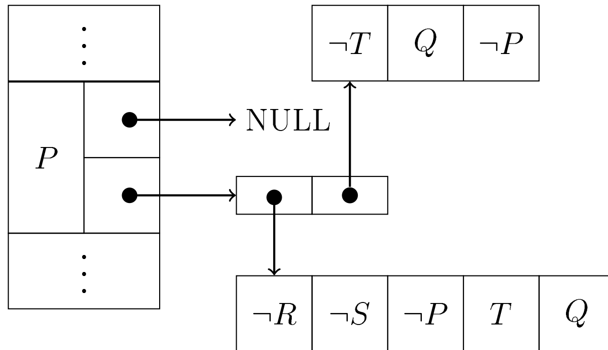
(a) Initialized 2WL data structure for the literal P and the current trail is empty.



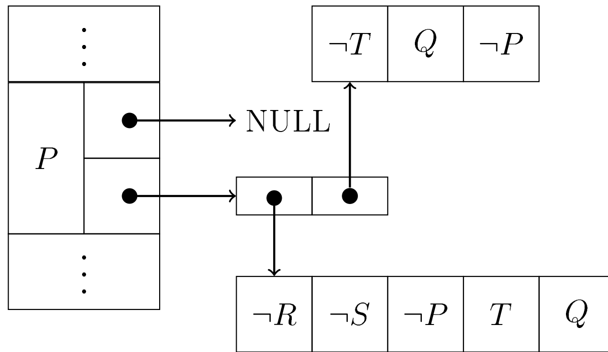
(b) After deciding P the watched literals have changed and the current trail is: P.



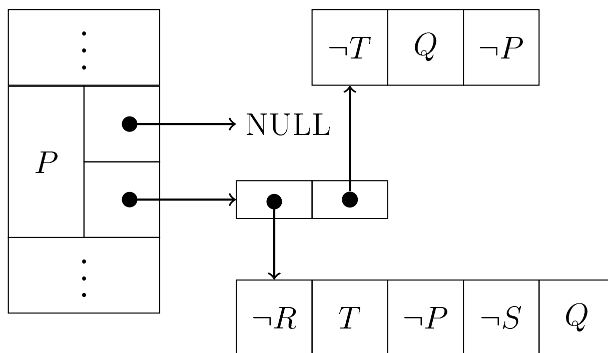
(c) After deciding $\neg Q$ the unit clause $\{\neg P \vee Q \vee \neg T\}$ is achieved and the current trail is: $P, \neg Q$.



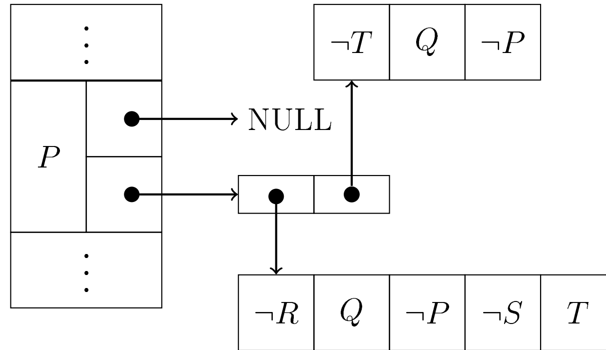
(d) After propagating $\neg T, R$ and S the current trail is: $P, \neg Q, \neg T, R, S$ and the clause $\{\neg P \vee Q \vee \neg R \vee \neg S \vee T\}$ evaluates to false, a conflict.



(e) After backtracking S, R, T, Q the current trail is: P.



(f) After propagating Q and deciding S the trail is: P, Q, S.



(g) After deciding $\neg T$ and propagating R the trail is: $P, Q, S, \neg T, R$.

Figure 2.17: The watched literals list for the formula $\phi = \{\neg P \vee Q \vee \neg R \vee \neg S \vee T, \neg P \vee Q \vee \neg T, R \vee T, S \vee T\}$ before and after deciding / propagating variables with a focus on the literal P.

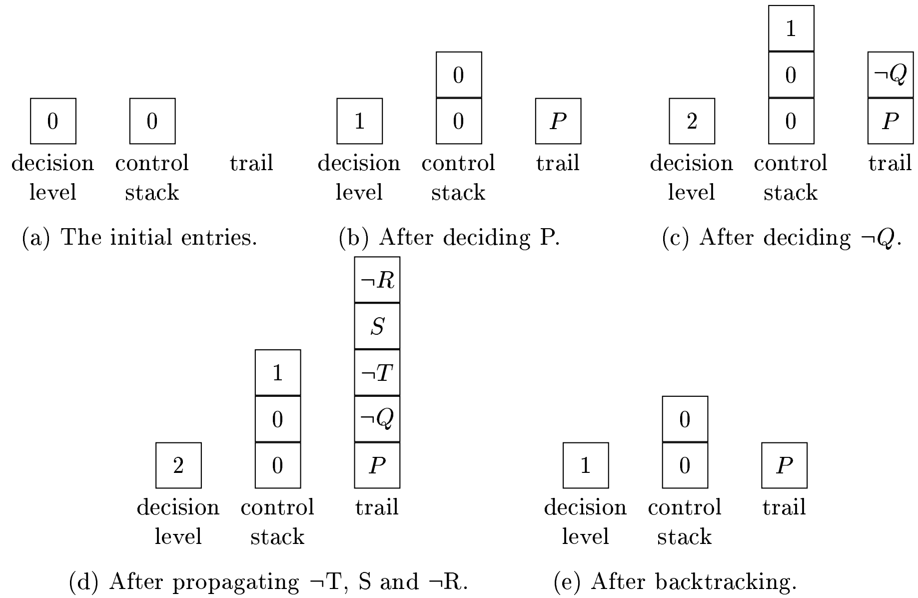


Figure 2.18: The entries for decision level, control stack and trail for the formula $\phi = \{S \vee Q, P \vee Q, \neg P \vee R \vee \neg S, \neg P \vee \neg R \vee T, \neg P \vee Q \vee \neg T\}$.

2.11.3 Dynamic Decision Heuristic: VSIDS

Choosing the right unassigned variable to decide is important for efficiency, but the heuristic may be expensive itself. Therefore, the aim is to use a heuristic that needs not to be recomputed too often, that for example chooses variables which occur frequently and prefers variables from recent conflicts.

The *VSIDS* (Variable State Independent Decaying Sum) is such a heuristic. The strategy is as follows:

1. Initially assign each variable a score e.g. its number of occurrences in the formula.

2. Adjust the scores during a CDCL run: whenever a conflict clause is resolved with another clause the resolved variable gets its score increased by a bonus d, initially $d = 1$ and d increases with every conflict: $d = \lceil \frac{6}{5} d \rceil$.

3. Furthermore, whenever a clause is learned the score of the variables of this clause is additionally increased by adding d to its score.

4. As soon as a variable score s or d reaches a certain limit k, e.g. $k = 2^{60}$, all variables get their score rescaled by a constant, e.g. $s = \lceil s \cdot 2^{60} \rceil$. At this point d is also rescaled: $d = \lceil d \cdot 2^{-50} \rceil$.

5. At a decision point with probability $\frac{1}{50}$ choose a variable at random. In the other cases choose an unassigned variable with the highest score.

The heuristic has very low overhead since it is independent of variable assignments which makes it a fast strategy. Furthermore, it favors variables that satisfy the most possible number of clauses and prefers variables that are more involved in conflicts.

2.11.4 Conflict Analysis and Learning: 1UIP scheme

If a conflicting clause is found, the algorithm needs to derive a new clause from the conflict and add it to the current set of clauses. But the problem is that this may produce a large number of new clauses, therefore it becomes necessary to choose a clause that is most reasonable.

This section examines how to derive such a conflict clause once a conflict is detected. The key idea is to find an *asserting clause* that includes the *first UIP* (Unique Implication Point). For this purpose the concept of implication graphs is required and hence defined first. An *implication graph* $G = (V, E)$ is a directed graph with a node set V and an edge set E. Each node has the form l/L, which means that the variable L was set to a value (either true or false) at the decision level l either via the rule Propagate or Decide. If a variable L of a node n was set via the rule Propagate with clause $C = D \lor L$ then there must be an edge from every node of the variables in D to n. This means that the variables from D imply L. In particular, decision variable nodes have no incoming edges. A *cut* of an implication graph is a partition of the graph into

two nonempty sets such that the decision variable nodes will be in a different set than the conflict node. Every edge that crosses a specific cut will be part of a conflict set, i.e. the number of cuts denotes the number of conflict sets. There is a total of 2^{n-k} possible cuts, where $n = \#$ variables and $k =$ level of conflict clause ($= \#$ decision variables). A *UIP* in the graph is a variable of the conflict level l that lies on every path from the decision variable of level l and the conflict. The first UIP (1UIP) is a UIP that lies closest to the conflict in the implication graph. The strategy for deriving the most useful conflict clause is as follows:

1. Construct the implication graph according to a given set of clauses, a formula ϕ. As an example consider Figure 2.19 that depicts an implication graph of the formula $\phi = \{S \vee Q, P \vee Q, \neg P \vee R \vee \neg S, \neg P \vee \neg R \vee T, \neg P \vee Q \vee \neg T\}$ where the node $1/\emptyset$ denotes a conflict. The corresponding trail, control stack and decision level are shown in Figure 2.18. The corresponding watched literals list is shown in Figure 2.23.

2. Identify the conflict sets by means of the implication graph, i.e. the cuts of the graph need to be considered. In Figure 2.19 there are three cuts depicted representing the following conflict sets: $\{P, \neg Q\}, \{P, \neg T, S\}$ and $\{P, \neg R, S\}$.

3. Choose the most useful clause from the set of all conflicts. It proved to be most effective to choose a clause that has exactly one variable that was assigned at the same decision level in which the conflict arose. This is why the clause is also called asserting clause. If there is more than one asserting clause for a conflict as in Figure 2.19, then take the asserting clause that contains the 1UIP. In Figure 2.20 there is only one UIP which is also the 1UIP that is $\neg Q$. Therefore, the most useful clause from the conflict set is $\{P, \neg Q\}$.

4. Learn the clause: After determining the asserting clause C with the 1UIP the actual conflict clause is obtained by negating all assignments of the variables within clause C. This conflict clause will eventually be learned by adding it to the set of clauses of the original formula ϕ. In the example from Figure 2.19 the clause $\neg P \vee Q$ will be learned.

The combination of conflict analysis and non-chronological backtracking ensures that the learned clause becomes a unit clause and thereby preventing the solver from making the same mistakes over again.

2.11.5 Restart and Forget

As mentioned in the section on VSIDS (see 2.11.3) the runtime of the CDCL implementation depends on the choice of the decision variable. In case no suitable variable is found within a certain time limit it might be useful to apply a restart, another important technique applied in the CDCL implementation. With the rule Restart all currently assigned variables will become unassigned

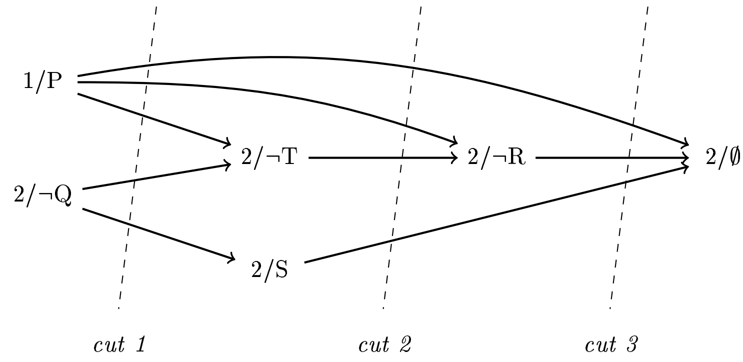


Figure 2.19: An implication graph for the formula ϕ with cuts.

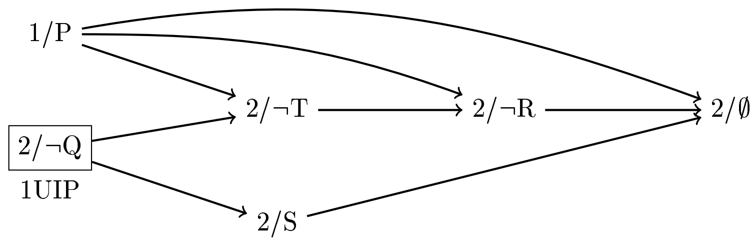
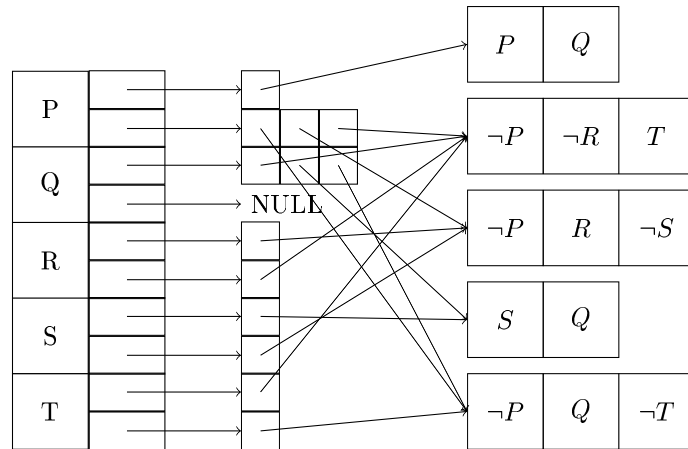
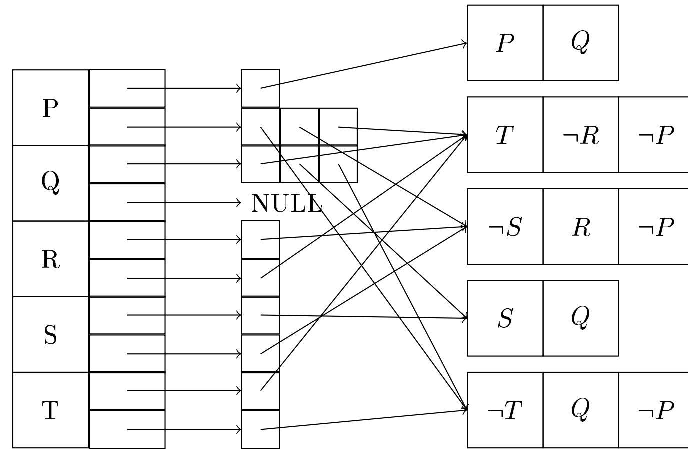


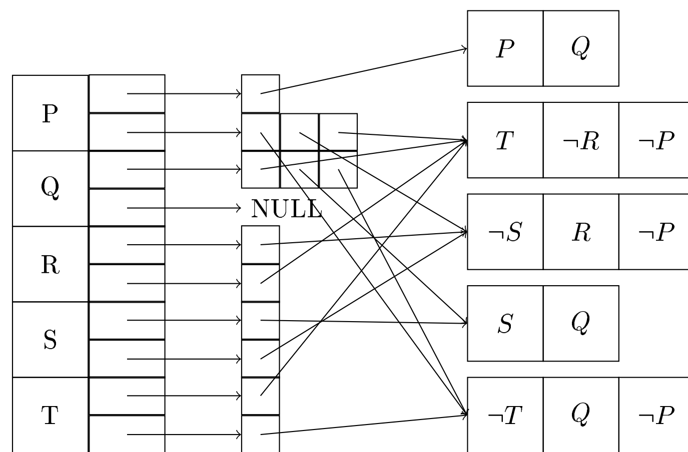
Figure 2.20: The implication graph denoted with the 1UIP.



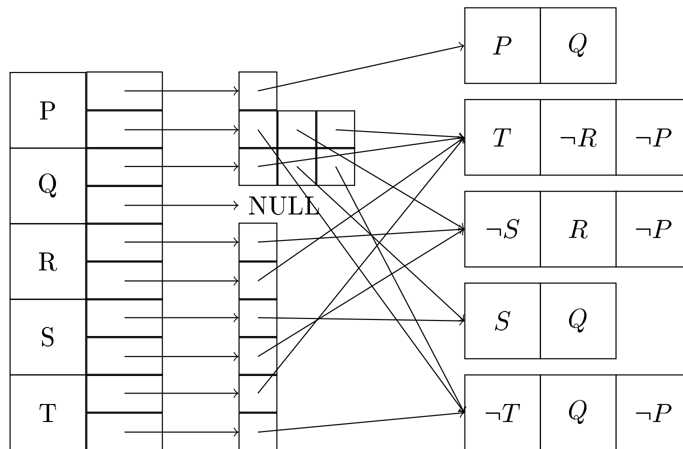
(a) The initial state and the current trail is empty.



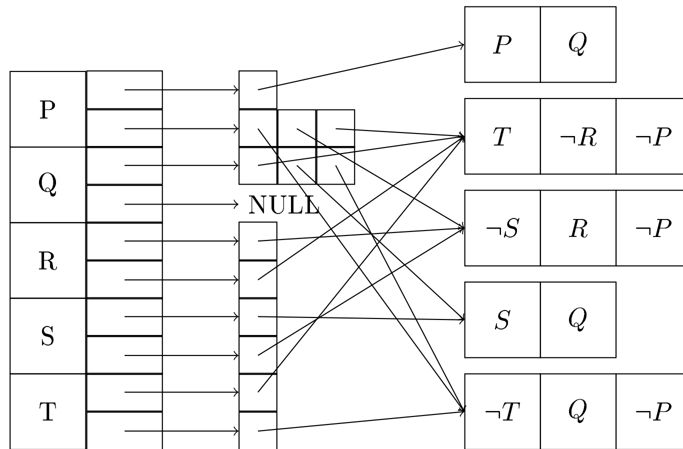
(b) After deciding P watched literals are swapped, the trail is: P.



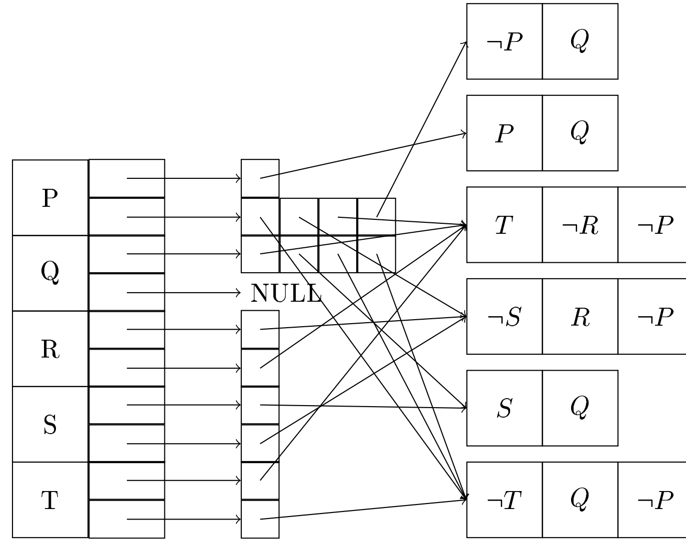
(c) After deciding $\neg Q$, no change in the watched literals, the trail is: $P, \neg Q$.



(d) After propagating $\neg T, S$ and $\neg R$, no change of watched literals but a conflict occurs in $\neg P \vee R \vee S$, the trail is: $P, \neg Q, \neg T, S, \neg R$.



(e) After backtracking the literals $\neg Q, \neg T, S, \neg R$, the trail is: P.



(f) After learning the clause $\neg P \vee Q$, the trail is still P.

Figure 2.23: The watched literals list according to the implication graph from Figures 2.19 and 2.20 as well as the control stack, trail and decision level of Figure 2.18.

while learned clauses will be maintained. The motivation for this technique has to do with the fact that the solver can reach a point where incorrect variable assignments were made and the solver is not able to resolve within a reasonable amount of time the literals that are needed to find a conflict. In that case a restart is performed intending to make better variable assignments earlier on with the previous learned information.

A further technique that contributes to the performance of the CDCL solver is the rule Forget. With every conflict clause the number of learned clauses increases. Recording all learned clauses can be very expensive especially if some clauses are repeatedly stored or if some clauses are subsumed by others. As a result, this can lead to an exhaustion of available memory and to an additional overhead. Therefore deleting suitable clauses from the learned clause set can be useful. The criteria by which the rule Forget is applied are the following: either if the number of learned clauses is 4 times the number of original clauses or if a specific maximum number of learned clauses is reached that is previously given. In both cases the minimum of the following 2 cases is executed: either half of the learned clauses are deleted or all learned clauses are deleted until a clause is reached that implies or has implied a current assignment. Furthermore, an implementation could also check the subsumption of learned clauses over existing clauses but this check is often omitted due to performance reasons.

2.11.6 Algorithm and Strategy

As shown in the examples 2.10.1 and 2.10.2 a certain CDCL rule application order can improve the performance of the rule-based CDCL algorithm. The algorithm 5 depicts the strategy where Conflict is preferred over Propagate and Propagate over any other rule. In general the rules Decide and Propagate should not be applied when a conflict already exists. For otherwise, the additional literals that are added via Decide or Propagate become useless and will be deleted again when backtracking. Therefore the application of the rule Conflict is checked before any other rule. The statements from line 1 onwards describe the actual strategy, i.e. Conflict is always preferred over any other rule and Propagate is preferred over Decide. The reason why the rules Skip and Resolve are always applied excessively once a conflict was found is due to finding the clause with the 1UIP of the conflict level. The rule Skip is applied to those literals that are not involved in the conflict. Via the rule Resolve the conflict clause is resolved with clauses that implied the conflict and thereby yielding a new potentially learned clause. Once both rules cannot be applied anymore the state is either a fail state, Backtrack cannot be applied and the algorithm returns the fail state $(M; N; U; k; \bot)$ or the state is not a fail state and the conflict clause with the 1UIP was found. In the latter case the current conflict clause will be learned via the rule Backtrack. At this point it is checked whether the total number of approached conflicts reached a certain limit, i.e. a restart is necessary, indicating that the solver needs too much time detecting an incorrect value assignment that was previously made. Since the number of learned clauses increases with every conflict it is also checked whether previously learned clauses can be deleted, i.e. forget is necessary. In case the current state has no conflict, the rule Propagate is preferred over the rule Decide in line 15 since the chances of taking wrong decisions when deciding a literal's truth value decreases. The rule Decide takes the value of the VSIDS heuristic for the current state into account.

2.12 Superposition and CDCL

At the time of this writing it is often believed that the superposition (resolution) calculus is not successful in practice whereas most of the successful SAT solvers implemented in 2012 are based on CDCL. In this section I will develop some relationships between superposition and CDCL.

The start is a modification of the superposition model operator, Definition 2.8.5. The goal of the original model operator is to create minimal models with respect to positive literals, i.e., if $N_\mathcal{I} \models N$ for some N, then there is no $M' \subset N_\mathcal{I}$ such that $M' \models N$. However, if the goal generating minimal models is dropped, then there is more freedom to construct the model while preserving the general properties of the superposition calculus. So, let's assume a heuristic $\mathcal{H}$ that selects whether a literal should be productive or not.

Algorithm 5: CDCL(S)

```
Input  : An initial state (ϵ; N; ∅; 0; ⊤).
Output: A final state S = (M; N; U; k; ⊤) or S = (M; N; U; k; ⊥)
1  while (any rule applicable) do
2      ifrule (Conflict(S)) then
3          while (Skip(S) || Resolve(S)) do
4              update VSIDS scores on resolved literals;
5          end
6          update VSIDS scores on learned clause;
7          Backtrack(S);
8          scale VSIDS scores;
9          if (forget heuristic) then
10             Forget(S) redundant clauses ;
11         Restart(S);
12     else
13         ifrule (!Propagate(S)) then
14             Decide(S);
15
16
17 end
18 return(S);
```

Definition 2.12.1 (Heuristic-Based Partial Model Construction)**.** Given a clause set N, an ordering $\prec$ and a variable heuristic $\mathcal{H} : \Sigma \to \{0, 1\}$, the (partial) model $N_\Sigma^\mathcal{H}$ for N and signature Σ, with $P, Q \in \Sigma$ is inductively constructed as follows:

$$N_P^\mathcal{H} := \bigcup_{Q \prec P} \delta_Q^\mathcal{H}$$

$$\delta_P^\mathcal{H} := \begin{cases} \{P\} & \text{if } (D \vee P) \in N, P \text{ strictly maximal and } N_P^\mathcal{H} \not\models D \quad \text{or} \\ & \mathcal{H}(P) = 1 \text{ and for all clauses } (C \vee \neg P) \in N, C \prec \neg P \\ & \text{it holds } N_P^\mathcal{H} \models C \\ \emptyset & \text{otherwise} \end{cases}$$

$$N_\Sigma^\mathcal{H} := \bigcup_{P \in \Sigma} \delta_P^\mathcal{H}$$

T

Please note that $N_\mathcal{I}$ is defined inductively over the clause ordering $\prec$ whereas $N_\Sigma^\mathcal{H}$ is defined inductively over the atom ordering $\prec$.

Proposition 2.12.2. If $\mathcal{H}(P) = 0$ for all $P \in \Sigma$ then $N_\mathcal{I} = N_\Sigma^\mathcal{H}$ for any N.

Proof. The proof is by contradiction. Assume $N_\mathcal{I} \neq N_\Sigma^\mathcal{H}$, i.e., there is a minimal $P \in \Sigma$ such that P occurs only in one set out of $N_\mathcal{I}$ and $N_\Sigma^\mathcal{H}$.

Case 1: $P \in N_{\mathcal{I}}$ but $P \notin N_{\Sigma}^{\mathcal{H}}$.
Then there is a productive clause $D = D' \vee P \in N$ such that P is strictly maximal in this clause and $N_D \not\models D'$. Since P is strictly maximal in D the clause D' only contains literals strictly smaller than P. Since both interpretations agree on all literals smaller than P from $N_D \not\models D'$ it follows $N_P^{\mathcal{H}} \not\models D'$ and therefore $\delta_P^{\mathcal{H}} = \{P\}$ contradicting $P \notin N_{\Sigma}^{\mathcal{H}}$.

Case 2: $P \notin N_{\mathcal{I}}$ but $P \in N_{\Sigma}^{\mathcal{H}}$.
Then there is a productive clause $D = D' \vee P \in N$ such that P is strictly maximal in this clause and $N_P^{\mathcal{H}} \not\models D'$ because $\mathcal{H}(P) = 0$. Since P is strictly maximal in D the clause D' only contains literals strictly smaller than P. Since both interpretations agree on all literals smaller than P from $N_P^{\mathcal{H}} \not\models D'$ it follows $N_D \not\models D'$ and therefore $\delta_D = \{P\}$ contradicting $P \notin N_{\mathcal{I}}$. □

So the new model operator $N_{\Sigma}^{\mathcal{H}}$ is a generalization of $N_{\mathcal{I}}$. Next, I will show that with the help of $N_{\Sigma}^{\mathcal{H}}$ a close relationship between the model operator run by the CDCL calculus and the superposition model operator can be established. This result can then further be used to relate the abstract superposition redundancy criteria to CDCL. But before going into the relationship I first show that the generalized superposition partial model operator $N_{\Sigma}^{\mathcal{H}}$ supports the standard superposition completeness result, analogous to Theorem 2.8.9. Recall that the same notion of redundancy, Definition 2.8.3, is used.

Theorem 2.12.3. If N is saturated up to redundancy and $\bot \notin N$ then N is satisfiable and $N_{\Sigma}^{\mathcal{H}} \models N$.

Proof. The proof is by contradiction. So I assume (i) any clause C derived by Superposition Left or Factoring from N that C is redundant, i.e., $N^{\prec C} \models C$, (ii) $\bot \notin N$ and (iii) $N_{\Sigma}^{\mathcal{H}} \not\models N$. Then there is a minimal, with respect to $\prec$, clause $C_1 \vee L \in N$ such that $N_{\mathcal{I}} \not\models C_1 \vee L$ and L is a maximal literal in $C_1 \vee L$. This clause must exist because $\bot \notin N$.

The clause $C_1 \vee L$ is not redundant. For otherwise, $N^{\prec C_1 \vee L} \models C_1 \vee L$ and hence $N_{\Sigma}^{\mathcal{H}} \models C_1 \vee L$, because $N_{\Sigma}^{\mathcal{H}} \models N^{\prec C_1 \vee L}$, a contradiction.

I distinguish the case whether L is a positive or a negative literal. Firstly, assume L is positive, i.e., $L = P$ for some propositional variable P. Now if P is strictly maximal in $C_1 \vee P$ then actually $\delta_P^{\mathcal{H}} = \{P\}$ and hence $N_P^{\mathcal{H}} \models C_1 \vee P$, a contradiction. So P is not strictly maximal. But then actually $C_1 \vee P$ has the form $C_1' \vee P \vee P$ and Factoring derives $C_1' \vee P$ where $(C_1' \vee P) \prec (C_1' \vee P \vee P)$. Now $C_1' \vee P$ is not redundant, strictly smaller than $C_1 \vee L$, we have $C_1' \vee P \in N$ and $N_{\Sigma}^{\mathcal{H}} \not\models C_1' \vee P$, a contradiction against the choice that $C_1 \vee L$ is minimal.

Secondly, assume L is negative, i.e., $L = \neg P$ for some propositional variable P. Then, since $N_{\Sigma}^{\mathcal{H}} \not\models C_1 \vee \neg P$ we know $P \in N_{\mathcal{I}}$, i.e., $\delta_P^{\mathcal{H}} = \{P\}$. There are two cases to distinguish. Firstly, there is a clause $C_2 \vee P \in N$ where P is strictly maximal and by definition $(C_2 \vee P) \prec (C_1 \vee \neg P)$. So a Superposition Left inference derives $C_1 \vee C_2$ where $(C_1 \vee C_2) \prec (C_1 \vee \neg P)$. The derived clause $C_1 \vee C_2$ cannot be redundant, because for otherwise either $N^{\prec C_2 \vee P} \models C_2 \vee P$ or $N^{\prec C_1 \vee \neg P} \models C_1 \vee \neg P$. So $C_1 \vee C_2 \in N$ and $N_{\Sigma}^{\mathcal{H}} \not\models C_1 \vee C_2$, a contradiction

against the choice that $C_1 \vee L$ is minimal. Secondly, there is no clause $C_2 \vee P \in N$ where P is strictly maximal but $\mathcal{H}(P) = 1$. But a further condition for this case is that there is no clause $(C_1 \vee \neg P) \in N$ such that $N_P^{\mathcal{H}} \not\models C_1$ contradicting the above choice of $C_1 \vee \neg P$. ☐

Recalling Section 2.8 Superposition is based on an ordering $\prec$. It relies on a model assumption $N_{\mathcal{I}}$, Definition 2.8.5 or its generalization $N_{\Sigma}^{\mathcal{H}}$, Definition 2.12.1. Given a set N of clauses, either $N_{\mathcal{I}}$ ($N_{\Sigma}^{\mathcal{H}}$) is a model for N, N contains the empty clause, or there is an inference on the minimal false clause with respect to $\prec$, see the proof of Theorem 2.8.9 or Theorem 2.12.3, respectively.

CDCL is based on a variable selection heuristic. It computes a model assumption via decision variables and propagation. Either this assumption is a model of N, N contains the empty clause, or there is a backjump clause that is learned.

For a CDCL state (M, N, U, k, D) generated by an application of the rule Conflict, where $M = L_1, \ldots, L_n$ any following Resolve step actually corresponds to a superposition step between a minimal false clause and its productive counterpart, where $\text{atom}(L_1) \prec \text{atom}(L_2) \prec \ldots \prec \text{atom}(L_n)$. Furthermore, for a positive decision literal $L_m^{\top}$ occurring in M the heuristic $\mathcal{H}(\text{atom}(L_m)) = 1$ and $\mathcal{H}(\text{atom}(L_m)) = 0$ otherwise. Then the learned clause is in fact generated by superposition with respect to the model operator $N_{\Sigma}^{\mathcal{H}}$. The following propositions present this relationship between Superposition and CDCL in full detail.

Proposition 2.12.4. Let (M, N, U, k, D) be a CDCL state generated by a strategy with eager application of Conflict and Propagate, in this order. Let $M = L_1, \ldots, L_n$, $\mathcal{H}(\text{atom}(L_m)) = 1$ for any positive decision literal $L_m^{\top}$ occurring in M and $\mathcal{H}(\text{atom}(L_m)) = 0$ otherwise. The superposition ordering is $\text{atom}(L_1) \prec \text{atom}(L_2) \prec \ldots \prec \text{atom}(L_n)$. Then

1. L_n is a propagated literal.
2. The resolvent between $C \vee \neg L_k$ and the clause $C' \vee L_k$ propagating L_k is a superposition inference and the conclusion is not redundant.

Proof. 1. Assume L_n is a decision literal. Then, since Conflict and Propagation are applied eagerly, D has the form $D = D' \vee \neg L_n$. But then at trail $L_1, \ldots, L_{n-1}$ the clause $D' \vee \neg L_n$ propagates $\neg L_n$ with respect to $L_1 \ldots L_{n-1}$, so with eager propagation, the literal L_n cannot be decision literal but its negation was propagated by a clause $D' \vee \neg L_n \in N$.

2. Both C and C' only contain literals with variables from $\text{atom}(L_1)$, $\ldots, \text{atom}(L_{k-1})$. Since we assume duplicate literals to be removed and tautologies to be deleted, the literal $\neg L_k$ is strictly maximal in $C \vee \neg L_k$ and L_k is strictly maximal in $C' \vee L_k$. So resolving on L_k is a superposition inference with respect to the variable ordering $\text{atom}(L_1) \prec \text{atom}(L_2) \ldots \prec \text{atom}(L_k)$. Now assume $C \vee C'$ is redundant, i.e., there are clauses $D_1, \ldots, D_n$ from N with $D_i \prec C \vee C'$ and $D_1, \ldots, D_n \models C \vee C'$. Since $C \vee C'$ is false in $L_1 \ldots L_{k-1}$ there is at least one D_i that is also false in $L_1 \ldots L_{k-1}$. A contradiction against the

assumption that $L_1 \ldots L_{k-1}$ does not falsify any clause in N, i.e., rule Conflict was applied eagerly. $\square$

Proposition 2.12.4 is actually a nice explanation for the efficiency of the CDCL procedure: a learned clause is never redundant. Recall that redundancy here means that the learned clause C is not entailed by smaller clauses in $N \cup U$. Furthermore, the ordering underlying Proposition 2.12.4 is based on the trail, i.e., it changes during a CDCL run. For superposition it is well known that changing the ordering is not compatible with the notion of redundancy, i.e., superposition is incomplete when the ordering may be changed infinitely often and the superposition redundancy notion is applied.

Example 2.12.5. Consider the superposition left inference between the clauses $P \lor Q$ and $R \lor \neg Q$ with ordering $P < R < Q$ resulting in $P \lor R$. Changing the ordering to $Q < P < R$ the inference $P \lor R$ becomes redundant. So flipping infinitely often between $P < R < Q$ and $Q < P < R$ is already sufficient to prevent any saturation progress.

Although Example 2.12.5 shows that changing the ordering is not compatible with redundancy and superposition completeness, Proposition 2.12.4 proves that any CDCL learned clause is not redundant in the superposition sense and the CDCL procedure changes the ordering and is complete. This relationship shows the power of reasoning with respect to a model assumption. The model assumption actually prevents the generation of redundant clauses. Nevertheless, also in the CDCL framework completeness would be lost if redundant clauses are eagerly removed in general. So either the ordering is not changed and the superposition redundancy notion can be eagerly applied or only a weaker notion of redundancy is possible while keeping completeness.

The crucial point is that for the superposition calculus the ordering is also the bases for termination and completeness. If the completeness proof can be decoupled from the ordering, then the ordering might be changed infinitely often and other notions of redundancy become available. However, these new notions of redundancy need to be compatible with the completeness, termination proof.

Definition 2.12.6 (Abstract Length Redundancy)**.** A clause C is *length redundant* with respect to a clause set N if $N^{\leq |C|} \models C$, where $N^{\leq |C|} = \{D \mid |D| \leq |C|\}$.

Theorem 2.12.7 (Length Redundancy and Superposition)**.** Arbitrary Ordering Changes plus fairness plus length redundancy preserves completeness.

Theorem 2.12.8 (Length Redundancy and CDCL)**.** At any time length redundant clauses may be removed.

2.13 Redundancy

One of the most successful and robust heuristics is to keep the formula, clause set "small". This heuristic is already the motivation for the specific renaming

algorithm presented in Section 2.6.3. So getting rid of superfluous, i.e., redundant formulas or clauses is typically beneficial to any efficient reasoning. The section on normal form transformation (Section 2.6) and the sections on CDCL and superposition already introduced some redundancy criteria. In this section they are extended for the case of clause sets.

There is an important difference between clause redundancy *before* a CDCL or superposition calculus starts reasoning and clause redundancy *while* the calculus (superposition, CDCL) is operating on a set of clauses. For the former it is sufficient that the redundancy procedure is sound and terminating. For the latter the procedure has in addition to respect the redundancy notion of the respective calculus in order to preserve completeness, see Definition 2.8.3, Example 2.12.5, and Theorem 2.12.8, Theorem 2.12.7.

2.13.1 Redundancy before Superposition and CDCL

Here are some standard rules for removing redundant clauses before superposition or CDCL starts. Subsumption, Tautology Deletion and Subsumption Resolution have already been introduced in Section 2.8. Purity and Blocked Clause Deletion are new.

Subsumption Deletion
$(N \uplus \{C_1, C_2\}) \;\Rightarrow_{\text{RBSC}}\; (N \cup \{C_1\})$
provided $C_1 \subseteq C_2$

Tautology Deletion
$(N \uplus \{C \lor P \lor \neg P\}) \;\Rightarrow_{\text{RBSC}}\; (N)$

Subsumption Resolution
$(N \uplus \{C_1 \lor L, C_2 \lor \overline{L}\}) \;\Rightarrow_{\text{RBSC}}\; (N \cup \{C_1 \lor L, C_2\})$
where $C_1 \subseteq C_2$

Purity
$(N \uplus \{C_1 \lor L, \ldots, C_k \lor L\}) \;\Rightarrow_{\text{RBSC}}\; (N)$
where L, $\overline{L}$ do not occur in N

Blocked Clause Elimination
$(N \uplus \{C_1 \lor L, \ldots, C_k \lor L, C'_1 \lor \overline{L}, \ldots, C'_l \lor \overline{L}\}) \;\Rightarrow_{\text{RBSC}}\; (N)$
where L, $\overline{L}$ do not occur in N and all resolvents on L between any $C_i \lor L$ and $C'_j \lor \overline{L}$ result in tautologies

Example 2.13.1. Consider a clause set consisting of the five clauses
(1) $P \lor Q$
(2) $P \lor Q \lor R \lor S$
(3) $\neg R \lor S$
(4) $R \lor \neg S$
(5) $\neg Q \lor S$
Clause (1) subsumes clause (2). Subsumption resolution is applicable to

clause (2) and clause (5) resulting in $P \lor R \lor S$. Purity is applicable to P. Blocked clause elimination is not applicable.

Applying first subsumption deletion results in the clauses
(1) $P \lor Q$
(3) $\neg R \lor S$
(4) $R \lor \neg S$
(5) $\neg Q \lor S$
Now subsumption resolution is no longer applicable, but blocked clause elimination is to R and clauses (3), (4). After application of blocked clause elimination the resulting clauses are
(1) $P \lor Q$
(5) $\neg Q \lor S$
Now P and S are pure and after applying purity the result is the empty set of clauses indicating satisfiability.

For the above Example 2.13.1 other rule application orderings are possible, e.g., starting with purity on P. Nevertheless, any application ordering results in an empty set of clauses. However, $\Rightarrow_{\mathrm{RBSC}}$ is not confluent.

Lemma 2.13.2 ($\Rightarrow_{\mathrm{RBSC}}$ terminates)**.**

Proof. Exercise □

Lemma 2.13.3 ($\Rightarrow_{\mathrm{RBSC}}$ is sound)**.** If $(N) \Rightarrow_{\mathrm{RBSC}} (N')$ then N is satisfiable iff N' is.

Proof. $\Rightarrow$: All rules remove clauses except subsumption resolution. Removing clauses obviously preservers satisfiability. For subsumption resolution any model satisfying $C_1 \lor L$ and $C_2 \lor \overline{L}$ has to satisfy C_1 or C_2. Since $C_1 \subseteq C_2$ it satisfies C_2.

$\Leftarrow$: The direction is obvious for Subsumption Deletion, Tautology Deletion, and Subsumption Resolution. Since, actually, Purity is a special case of Blocked Clause Elimination, it suffices to show the case of Blocked Clause Elimination. In this case $N = N' \uplus \{C_1 \lor L, \ldots, C_k \lor L, C'_1 \lor \overline{L}, \ldots, C'_l \lor \overline{L}\}$ and L, $\overline{L}$ do not occur in N' and all resolvents on L between any $C_i \lor L$ and $C'_j \lor \overline{L}$ result in tautologies. Let $\mathcal{A}$ be a model for N'. Obviously, being $\mathcal{A}$ a model for N does not depend on the truth value of L, because neither L nor $\overline{L}$ occurs in N. If $\mathcal{A}$ does not satisfy some clause $C_i \lor L$ (analogously $C'_j \lor \overline{L}$), then $\mathcal{A}(L) = 0$ and $\mathcal{A}(C_i) = 0$. Since all combinations $C_i \lor C'_j$, for any j are tautologies, $\mathcal{A}(C'_j) = 1$ for all j. Hence $\mathcal{A}'$ which is like $\mathcal{A}$ except that $\mathcal{A}'(L) = 1$ is a model for N. □

2.13.2 Redundancy while Superposition and CDCL

2.14 Complexity

This book does not focus on complexity but on how to build systems that are useful for selected applications. Nevertheless, any system, calculus presented in

this chapter on SAT has a worst case exponential running time. So it cannot run efficiently on any SAT instance. So some background knowledge about relevant complexity results is useful. Here I concentrate on a personal selection of "classics", complexity results everybody interested in propositional logic reasoning should know.

The pigeon hole formulas are such a classic, because they were among the first detected formulas that don't have polynomial length resolution proofs. In addition, they explain why the renaming techniques introduced in Section 2.6.3 are not only useful to prevent an explosion in the number of generated clauses out of a formula, but also for the afterwards reasoning process.

Definition 2.14.1 (Pigeon Hole Formulas $\mathrm{ph}(n)$)**.** For some given n and propositional variables $P_{i,j}$, where $1 \leq j \leq n$, $1 \leq i \leq n+1$, the corresponding pigeon hole formula (clause set) $\mathrm{ph}(n)$ is

$$\mathrm{ph}(n) = \bigwedge_{1 \leq i \leq n+1} P_{i,1} \vee \ldots \vee P_{i,n} \quad \wedge \quad \bigwedge_{1 \leq j \leq n} \quad \bigwedge_{\substack{1 \leq i,k \leq n+1 \\ i<k}} \neg P_{i,j} \vee \neg P_{k,j}$$

The intuition behind a variable $P_{i,j}$ is that it is true iff pigeon i sits in hole j. Then the formulas $P_{i,1} \vee \ldots \vee P_{i,n}$ express that every pigeon has to sit in some hole and the formulas $\neg P_{i,j} \vee \neg P_{k,j}$ that a hole can host at most one pigeon. Since there is one more pigeon than holes, the formula is unsatisfiable.

Note that the number of clauses of a pigeon hole formula $\mathrm{ph}(n)$ grows cubic in n. The famous theorem on the pigeon whole formulas says that any resolution proof showing unsatisfiability of $\mathrm{ph}(n)$ has a length at least exponential in n, i.e., no resolution-based system can efficiently show unsatisfiability of a pigeon hole formula.

Theorem 2.14.2 (Haken [22])**.** The length of any resolution refutation of $\mathrm{ph}(n)$ is exponential in n.

Recall that any refutation of a CDCL procedure corresponds to a resolution refutation, where each conflict generates some new resolvents. Now, a CDCL procedure solves the pigeon hole problem by an enumeration of all possible combinations how to put the $n+1$ pigeons into the n holes. It guesses some pigeon in some whole, potentially propagates the consequences of the decision, guesses the next one and so on until a conflict for the particular guess shows that there is one hole missing for the final pigeon. Then it backtracks by remembering that for the particular guess, i.e., combination pigeons, holes, there is no solution. The CDCL procedure never "recognizes" the fact that the problem is completely symmetric in pigeons and holes, e.g., once it has shown that there is no solution with pigeon 1 in hole 1 ($P_{1,1}$ true) then the problem cannot be solved at all. It is not necessary anymore to test the holes 2 to n for pigeon 1, because these cases are symmetric. This is an informal explanation for the above theorem.

The pigeon hole problem can be easily solved by an inductive argument. For $\mathrm{ph}(n)$ we put pigeon $n+1$ in hole n. Then the problem is solvable iff $\mathrm{ph}(n-1)$ has a solution. Repeating this argument $n-1$ times it remains to show that

there is no solution for ph(1), i.e., the clause set $P_{1,1}$, $P_{2,1}$, $\neg P_{1,1} \vee \neg P_{2,1}$ is unsatisfiable.

This reasoning can be perfectly simulated by resolution if additional clauses over extra variables are added to ph(n). Let $B^k_{i,j}$ be fresh propositional variables where $2 \leq k \leq n$, $1 \leq j < k$, $1 \leq i \leq k$, where we add the clauses resulting from

$$\begin{array}{ll} B^n_{i,j} \leftrightarrow (P_{i,j} \vee (P_{i,n} \wedge P_{n+1,j})) & \text{for the first step} \\ B^k_{i,j} \leftrightarrow (B^{k+1}_{i,j} \vee (B^{k+1}_{i,k} \wedge B^{k+1}_{k+1,j})) & \text{for all subsequent steps} \end{array}$$

to ph(n), where $2 \leq k \leq n-1$ and the i, j run in the limits corresponding to $B^k_{i.j}$ or $B^n_{i.j}$, respectively. Since the $B^k_{i,j}$ are fresh and there is only one defining equivalence for each $B^k_{i,j}$, the resulting problem is unsatisfiable iff the original is. Each equivalence results in four clauses, e.g., the first equivalence generates the clauses $B^n_{i,j} \vee \neg P_{i,j}$, $B^n_{i,j} \vee \neg P_{i,n} \vee \neg P_{n+1,j}$, $\neg B^n_{i,j} \vee P_{i,j} \vee P_{i,n}$, $\neg B^n_{i,j} \vee P_{i,j} \vee P_{n+1,j}$. Thus there are only polynomially many clauses added to ph(n). Now the additional clauses enable to reproduce via resolution the inductive argument, where for each "induction step" only polynomially many resolution steps are needed. Thus the extended pigeon hole problem can be refuted by resolution in polynomially many steps [13].

For example, for the case $n = 2$ the pigeon hole clauses are

$$\begin{array}{ll} (1) & P_{1,1} \vee P_{1,2} \\ (2) & P_{2,1} \vee P_{2,2} \\ (3) & P_{3,1} \vee P_{3,2} \\ (4) & \neg P_{1,1} \vee \neg P_{2,1} \\ (5) & \neg P_{1,1} \vee \neg P_{3,1} \\ (6) & \neg P_{2,1} \vee \neg P_{3,1} \\ (7) & \neg P_{1,2} \vee \neg P_{2,2} \\ (8) & \neg P_{1,2} \vee \neg P_{3,2} \\ (9) & \neg P_{2,2} \vee \neg P_{3,2} \end{array}$$

and the additional equivalences defining the $B^2_{i,j}$ are

$$\begin{array}{l} B^2_{1,1} \leftrightarrow (P_{1,1} \vee (P_{1,2} \wedge P_{3,1})) \\ B^2_{2,1} \leftrightarrow (P_{2,1} \vee (P_{2,2} \wedge P_{3,1})) \end{array}$$

Now from $\neg B^2_{1,1} \vee P_{1,1} \vee P_{3,1}$, $\neg B^2_{2,1} \vee P_{2,1} \vee P_{3,1}$ with (1), (2), (4), (5), (6), (7) via resolution the clause

$$(10) \quad \neg B^2_{1,1} \vee \neg B^2_{2,1}$$

can be derived. From $B^2_{1,1} \vee \neg P_{1,1}$, $B^2_{1,1} \vee \neg P_{1,2} \vee \neg P_{3,1}$ with (1), (3), (8) via resolution the clause

$$(11) \quad B^2_{1,1}$$

can be derived. Analogously, from $B^2_{2,1} \vee \neg P_{2,1}$, $B^2_{2,1} \vee \neg P_{2,2} \vee \neg P_{3,1}$ with (2), (3), (9) via resolution the clause

$$(12) \quad B^2_{2,1}$$

can be derived. Now, (10), (11), (12) constitute ph(1), i.e., the above resolution steps successfully perform the reduction from ph(2) to ph(1).

C

There are two reasons why I discuss the pigeon hole problem in such detail. First, it shows that the invention of new names (propositional variables) for subformulas, can lead to an exponential reduction in proof size. So it constitutes a further justification for renaming during CNF transformation (see Section 2.6.3). However, in general, there is no easy answer when additional names help in proof length reduction or in proof search. Second, and in my opinion even more important, the pigeon hole problem example nicely shows that "inductive reasoning" can be done in propositional logic and that it can pay off. For many real world problems, e.g., hardware verification, inductive reasoning is key to solve the problems. At the time of this writing, research in how to automatically detect and make use of inductive properties has just started for propositional logic. This holds as well and gets even more difficult for more expressive logics, such as first-order logic.

For the rest of this section I will study some well-known classes for which SAT can be solved in polynomial time, namely, Horn-SAT and 2-SAT. Horn SAT is the class of clauses where each clause has at most one positive literal, 2-SAT the class of clauses where each clause has at most two literals. For both clauses SAT is decidable in polynomial time. Actually, the 2-SAT class constitutes a sharp border between polynomially solvable and NP-complete, because the 3-SAT class is already NP-complete.

Definition 2.14.3 (Horn-SAT)**.** A propositional clause set N belongs to the class of *Horn-SAT* problems if every clause contains at most one positive literal.

Definition 2.14.4 (k-SAT)**.** A propositional clause set N belongs to the class of *k-SAT* problems if every clause contains at most k literals.

Proposition 2.14.5. Any Horn-SAT clause set N can be decided in time linear in the size of N.

Proof. Superposition with selection is complete for SAT (Theorem 2.12.3). So consider a superposition saturation for N where in every clause containing a negative literal it is selected. Then the saturation process has two nice properties. First, any superposition inference is an inference between a positive unit clause and a clause containing at least one negative literal. Second, there is always a clause where all negative literals can be resolved away by positive unit clauses or the clause set N is satisfiable. Combining the two properties results in a linear-time algorithm for Horn-SAT. ☐

Actually, the proof of the above proposition implies that the CDCL rules Propagate and Conflict (see Section 2.10) are complete for Horn-SAT. Another consequence is that unit superposition, a restriction to superposition where for all inferences one parent clause must be a unit clause, is also complete for Horn-SAT. For unit superposition the result can even be reversed. If for some clause set N there is a unit superposition refutation, then the subset of clauses involved

in the unit refutation can be transformed into a Horn clause set by flipping signs of literals.

The clause set $P \vee Q$, $\neg P \vee R$, $\neg R \vee Q$, $\neg Q$ is unsatisfiable and refutable by unit superposition. It is not Horn because of the clause $P \vee Q$. Now by flipping the sign of Q in all clauses results in the clause set $P \vee \neg Q$, $\neg P \vee R$, $\neg R \vee \neg Q$, Q which is Horn, equisatisfiable, and still unit refutable.

Proposition 2.14.6. Any 2-SAT clause set N can be decided in time polynomial in the size of N.

Proof. (Idea) Firstly, all unit clauses can be eliminated by recursively resolving away the respective literals, following the algorithm of Proposition 2.14.5. For a clause set N containing only clauses of length two a directed graph is constructed. The nodes are the propositional literals from N. For each clause $L \vee K \in N$, the graph contains the two directed edges $(\overline{L}, K)$ and $(\overline{K}, L)$. Then N is unsatisfiable iff there is a cycle in the graph containing two nodes L, $\overline{L}$. This can be decided in time at most quadratic in N. ☐

Interestingly, 2-SAT constitutes the border to NP-completeness, because 3-SAT is already NP-complete. This can be seen by reducing any clause set to a satisfiability equivalent 3-SAT clause set via the following transformation. For any clause

$$L_1 \vee \ldots \vee L_n$$

consisting of more than three literals ($n > 3$) replace the clause by the clauses

$$L_1 \vee \ldots \vee L_{\lfloor n/2 \rfloor} \vee P$$
$$L_{\lfloor n/2 \rfloor + 1} \vee \ldots \vee L_n \vee \neg P$$

where P is a fresh propositional variable. Obviously, $L_1 \vee \ldots \vee L_n$ is satisfiable iff $L_1 \vee \ldots \vee L_{\lfloor n/2 \rfloor} \vee P$, $L_{\lfloor n/2 \rfloor + 1} \vee \ldots \vee L_n \vee \neg P$ are.

Proposition 2.14.7. 3-SAT is NP-complete.

2.15 Applications

For the application of propositional logic on an arbitrary problem it needs to be encoded into a propositional formula ϕ. The satisfiability of ϕ can then be checked via one of the calculi developed in this chapter, e.g. Resolution or DPLL. In case ϕ is satisfiable the corresponding calculus derives a model which has to be interpreted as a solution to the original problem. The unsatisfiability of ϕ must be interpreted correspondingly.

2.15.1 Sudoku

As a suitable application of propositional logic serves the Sudoku puzzle. In chapter 1.1 a specific 4×4 Sudoku puzzle was solved using a specific calculus. In this section a general $n^2 \times n^2$ Sudoku puzzle is encoded into propositional

logic and exemplarily the Resolution calculus from this chapter is applied to a 4×4 Sudoku puzzle.

For the encoding propositional variables $P^d_{i,j}$ are defined where $P^d_{i,j}$ is *true* iff the value of square (i,j) is d. Square boxes are denoted by $Q_{i,j}$ where $Q_{i,j}$ includes the squares $(i,j),\ldots,(i+n-1,j+n-1)$. The corresponding propositional clauses are constructed as follows:

1. For every initially assigned square (i,j) with value d generate $P^d_{i,j}$
2. For every square (i,j) generate $P^1_{i,j} \vee \ldots \vee P^{n^2}_{i,j}$
3. For every square (i,j) and pair of values $d < d'$ generate $\neg P^d_{i,j} \vee \neg P^{d'}_{i,j}$
4. For every value d and column i generate $P^d_{i,1} \vee \ldots \vee P^d_{i,n^2}$ (analogously for rows)
5. For every value d and square box $Q_{i,j}$ generate $P^d_{i,j} \vee \ldots \vee P^d_{i+n-1,j+n-1}$
6. For every value d, column i and pair of rows $j < j'$ generate $\neg P^d_{i,j} \vee \neg P^d_{i,j'}$ (analogously for rows)
7. For every value d, square box $Q_{i,j}$ and pair of squares $(k,l) <_{\text{lex}} (k',l')$ where $i \le k, k' < i+n$ and $j \le l, l' < j+n$ generate $\neg P^d_{k,l} \vee \neg P^d_{k',l'}$

The corresponding formula ϕ is the conjunction of each subformula generated by the steps 1 to 7. This makes a total of $m + n^4 + \frac{1}{2}n^6(n^2-1) + 2n^4 + n^4 + \frac{1}{2}n^6(n^2-1) + \frac{1}{2}n^6(n^2-1) = m + 4n^4 + \frac{3}{2}n^6(n^2-1)$ clauses where m is the number of initially assigned squares.

After the application of a propositional logic calculus the remaining unit clauses $P^d_{i,j}$, i.e. the missing numbers to the initial Sudoku puzzle, are derived if the encoded formula is satisfiable. Otherwise there is no solution to the Sudoku puzzle.

	1	2	3	4
1	1			
2			1	
3		2		
4				4

Figure 2.24: A 4×4 Sudoku

The application of this encoding on the puzzle from Figure 2.24 yields for example the clauses $P^1_{3,4} \vee P^2_{3,4} \vee P^3_{3,4} \vee P^4_{3,4}$, $\neg P^2_{2,3} \vee \neg P^2_{3,3}$, $\neg P^2_{2,3} \vee \neg P^2_{4,3}$ and $P^2_{2,3}$. Applying the rule Resolution from the Resolution calculus from chapter 2.7 results in:

$(N \uplus \{\neg P^2_{2,3} \vee \neg P^2_{3,3}, P^2_{2,3}\} \Rightarrow_{\text{RES}} (N \cup \{\neg P^2_{2,3} \vee \neg P^2_{3,3}, P^2_{2,3}\} \cup \{\neg P^2_{3,3}\})$ and $(N' \uplus \{P^1_{3,4} \vee P^2_{3,4} \vee P^3_{3,4} \vee P^4_{3,4}, \neg P^2_{3,3}\}) \Rightarrow_{\text{RES}} (N' \cup \{P^1_{3,4} \vee P^2_{3,4} \vee P^3_{3,4} \vee P^4_{3,4}, \neg P^2_{3,3}\} \cup$

$\{P^1_{3,4} \vee P^3_{3,4} \vee P^4_{3,4}\}) \Rightarrow^*_{\text{RES}} (N'' \cup \{P^2_{3,4}\})$ see Figure 2.25. After exhaustive application of the Resolution calculus the remaining unit constraints are derived and the solution is found.

	1	2	3	4
1	1			
2			1	
3		2		
4			2	4

Figure 2.25: A 4×4 Sudoku after generating the unit constraint $P^2_{3,4}$

2.15.2 Hardware Verification

Another example for the application of propositional logic is the verification of logic hardware circuits. Since specific logic hardware circuits can be transformed into CNF the satisfiability of small logic circuits as well as certain properties of logic circuits can be checked with a propositional calculus from this chapter. This chapter shows how to encode specific logic circuits into propositional logic and how to apply the encoding on an exemplary logic circuit as shown in Figure 2.26.

This chapter considers logic circuits with three different types of gates G_i: AND-, OR- and NOT-gates. Each gate has one output, AND- and OR-gates have two inputs whereas the NOT-gate has only one input. For the encoding of the logic circuits a propositional variable Q_i is defined for each gate G_i where Q_i is true iff the gate G_i has output value 1. The propositional clauses are constructed as follows:

1. For every AND-gate G_i with inputs Q_j and Q_k we have $Q_i \leftrightarrow (Q_j \wedge Q_k)$ which is equivalent to $(\neg Q_i \vee Q_j) \wedge (\neg Q_i \vee Q_k) \wedge (\neg Q_j \vee \neg Q_k \vee Q_i)$

2. For every OR-gate G_i with inputs Q_j and Q_k we have $Q_i \leftrightarrow (Q_j \vee Q_k)$ which is equivalent to $(\neg Q_i \vee Q_j \vee Q_k) \wedge (\neg Q_j \vee Q_i) \wedge (\neg Q_k \vee Q_i)$

3. For every NOT-gate G_i with input Q_j we have $Q_i \leftrightarrow \neg Q_j$ which is equivalent to $(\neg Q_i \vee \neg Q_j) \wedge (Q_j \vee Q_i)$.

The corresponding formula ϕ is the conjunction of all clauses generated by the steps 1 to 3. After generating this encoding a propositional calculus from chapter 2 can be applied in order to check certain properties of logic circuits (note that the calculi presented in chapter 2 are inefficient on larger logic circuit constructions). Some of the properties that can be checked are for example the satisfiability of logic circuits given a partial truth assignment β (which assigns boolean values to outputs), the satisfiability of logic circuits in case of a recursive construction, the equivalence of two logic circuits or to check if certain properties for example $Q_0 \rightarrow Q_5$ for the logic circuit in Figure 2.26 hold.

As an example the satisfiability of the logic circuit in Figure 2.26 under a given partial truth assignment $\beta(Q_0) = 1$ and $\beta(Q_5) = 1$ can be checked using the DPLL calculus:

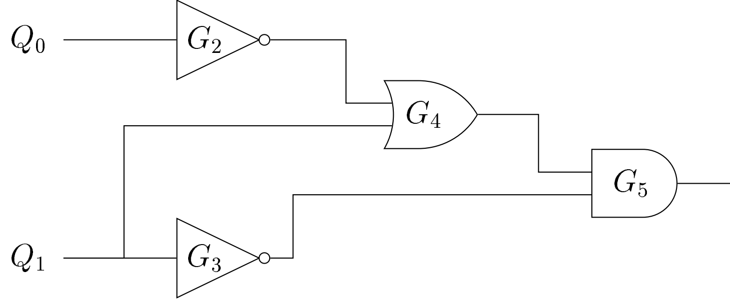


Figure 2.26: A logic circuit with two NOT-gates (G_2 and G_3), an OR-gate G_4 and an AND-gate G_5

The application of the encoding to the logic circuit of Figure 2.26 together with the partial truth assignment β yields a total of 12 clauses: $N = \{Q_0, Q_5, \neg Q_4 \lor Q_2 \lor Q_1, \neg Q_2 \lor Q_4, \neg Q_1 \lor Q_4, \neg Q_2 \lor \neg Q_0, Q_2 \lor Q_0, \neg Q_3 \lor \neg Q_1, Q_3 \lor Q_1, \neg Q_5 \lor Q_4, \neg Q_5 \lor Q_3, \neg Q_4 \lor \neg Q_3 \lor Q_5\}$. Applying the DPLL calculus we achieve: $(\epsilon; N) \Rightarrow^{\text{Propagate}}_{\text{DPLL}} (Q_0; N) \Rightarrow^{\text{Propagate}}_{\text{DPLL}} (Q_0Q_5; N) \Rightarrow^{\text{Propagate}}_{\text{DPLL}} (Q_0Q_5Q_4; N) \Rightarrow^{\text{Propagate}}_{\text{DPLL}} (Q_0Q_5Q_4Q_3; N) \Rightarrow^{\text{Propagate}}_{\text{DPLL}} (Q_0Q_5Q_4Q_3\neg Q_1; N) \Rightarrow^{\text{Propagate}}_{\text{DPLL}} (Q_0Q_5Q_4Q_3\neg Q_1Q_2; N)$. Let $M = (Q_0Q_5Q_4Q_3\neg Q_1Q_2)$ then the logic circuit is unsatisfiable under the given truth assignment since $M \models \neg N$ and there is no decision literal in M.

If the logic circuit of Figure 2.26 is considered without a partial truth assignment then the construction is satisfiable for example with $M = (\neg Q_0 \neg Q_1)$. If the gate G_4 of Figure 2.26 is replaced by an AND-gate instead of an OR-gate then the construction will always be unsatisfiable independent of any truth assignment.

Historic and Bibliographic Remarks

Although already Greek philosophers like Aristotle (384 BC – 322 BC) were interested in "truth of propositions" the syntax and semantics of propositional logic goes back to the modern logicians, mathematicians and philosophers Augustus de Morgan (1806 – 1871), George Boole (1815 – 1864), Charles Sanders Peirce (1839 – 1914), and Gottlob Frege (1848 – 1925). In particular, today Boole's calculus [9] is known as "propositional logic". For a nice historic perspective see Martin Davis's book [15].

Chapter 3

First-Order Logic

First-Order logic is a generalization of propositional logic. Propositional logic can represent propositions, whereas first-order logic can represent individuals and propositions about individuals. For example, in propositional logic from "Socrates is a man" and "If Socrates is a man then Socrates is mortal" the conclusion "Socrates is mortal" can be drawn. In first-order logic this can be represented much more fine-grained. From "Socrates is a man" and "All man are mortal" the conclusion "Socrates is mortal" can be drawn.

This chapter introduces first-order logic with equality. However, all calculi presented here, namely Tableaux (Section 3.6) and Superposition (Section **??**) are presented only for its restriction without equality. Purely equational logic and first-order logic with equality are presented separately in Chapter **??** and Chapter **??**, respectively.

3.1 Syntax

Definition 3.1.1 (Many-Sorted Signature)**.** A *many-sorted signature* $\Sigma = (\mathcal{S}, \Omega, \Pi)$ is a triple consisting of a finite non-empty set $\mathcal{S}$ of *sort symbols*, a non-empty set Ω of *operator symbols* (also called *function symbols*) over $\mathcal{S}$ and a set Π of *predicate symbols*. Every operator symbol $f \in \Omega$ has a unique sort declaration $f : S_1 \times \ldots \times S_n \to S$, indicating the sorts of arguments (also called *domain sorts*) and the *range sort* of f, respectively, for some $S_1, \ldots, S_n, S \in \mathcal{S}$ where $n \geq 0$ is called the *arity* of f, also denoted with $\operatorname{arity}(f)$. An operator symbol $f \in \Omega$ with arity 0 is called a *constant*. Every predicate symbol $P \in \Pi$ has a unique sort declaration $P \subseteq S_1 \times \ldots \times S_n$. A predicate symbol $P \in \Pi$ with arity 0 is called a *propositional variable*. For every sort $S \in \mathcal{S}$ there must be at least one constant $a \in \Omega$ with range sort S.

In addition to the signature Σ, a variable set $\mathcal{X}$, disjoint from Ω is assumed, so that for every sort $S \in \mathcal{S}$ there exists a countably infinite subset of $\mathcal{X}$ consisting of variables of the sort S. A variable x of sort S is denoted by x_S.

Definition 3.1.2 (Term)**.** Given a signature $\Sigma = (\mathcal{S}, \Omega, \Pi)$, a sort $S \in \mathcal{S}$ and

a variable set $\mathcal{X}$, the set $T_S(\Sigma, \mathcal{X})$ of all *terms* of sort S is recursively defined by (i) $x_S \in T_S(\Sigma, \mathcal{X})$ if $x_S \in \mathcal{X}$, (ii) $f(t_1, \ldots, t_n) \in T_S(\Sigma, \mathcal{X})$ if $f \in \Omega$ and $f : S_1 \times \ldots \times S_n \to S$ and $t_i \in T_{S_i}(\Sigma, \mathcal{X})$ for every $i \in \{1, \ldots, n\}$.

The sort of a term t is denoted by $\text{sort}(t)$, i.e., if $t \in T_S(\Sigma, \mathcal{X})$ then $\text{sort}(t) = S$. A term not containing a variable is called *ground*.

For the sake of simplicity it is often written: $T(\Sigma, \mathcal{X})$ for $\bigcup_{S \in \mathcal{S}} T_S(\Sigma, \mathcal{X})$, the set of all terms, $T_S(\Sigma)$ for the set of all ground terms of sort $S \in \mathcal{S}$, and $T(\Sigma)$ for $\bigcup_{S \in \mathcal{S}} T_S(\Sigma)$, the set of all ground terms over Σ.

Definition 3.1.3 (Equation, Atom, Literal)**.** If $s, t \in T_S(\Sigma, \mathcal{X})$ then $s \approx t$ is an *equation* over the signature Σ. Any equation is an *atom* (also called *atomic formula*) as well as every $P(t_1, \ldots, t_n)$ where $t_i \in T_{S_i}(\Sigma, \mathcal{X})$ for every $i \in \{1, \ldots, n\}$ and $P \in \Pi$, $\text{arity}(P) = n$, $P \subseteq S_1 \times \ldots \times S_n$. An atom or its negation of an atom is called a *literal*.

The literal $s \dot{\approx} t$ denotes either $s \approx t$ or $t \approx s$. A literal is *positive* if it is an atom and *negative* otherwise. A negative equational literal $\neg(s \approx t)$ is written as $s \not\approx t$.

C

Non equational atoms can be transformed into equations: For this a given signature is extended for every predicate symbol P as follows: (i) add a distinct sort B to $\mathcal{S}$, (ii) introduce a fresh constant true of the sort B to Ω, (iii) for every predicate P, $P \subseteq S_1 \times \ldots \times S_n$ add a fresh function $f_P : S_1, \ldots, S_n \to B$ to Ω, and (iv) encode every atom $P(t_1, \ldots, t_n)$ as a function $f_P : S_1, \ldots, S_n \to B$. Thus, predicate atoms are turned into equations $f_P(t_1, \ldots, t_n) \approx \text{true}$. are overloaded here.

Definition 3.1.4 (Formulas)**.** The set $\text{FOL}(\Sigma, \mathcal{X})$ of *many-sorted first-order formulas with equality* over the signature Σ is defined as follows for formulas $\phi, \psi \in F_\Sigma(\mathcal{X})$ and a variable $x \in \mathcal{X}$:

$\text{FOL}(\Sigma, \mathcal{X})$	Comment
$\bot$	falsum
$\top$	verum
$P(t_1, \ldots, t_n), s \approx t$	atom
$(\neg\phi)$	negation
$(\phi \wedge \psi)$	conjunction
$(\phi \vee \psi)$	disjunction
$(\phi \rightarrow \psi)$	implication
$(\phi \leftrightarrow \psi)$	equivalence
$\forall x.\phi$	universal quantification
$\exists x.\phi$	existential quantification

A consequence of the above definition is that $\text{PROP}(\Sigma) \subseteq \text{FOL}(\Sigma', \mathcal{X})$ if the propositional variables of Σ are contained in Σ' as predicates of arity 0. A formula not containing a quantifier is called *quantifier-free*.

Definition 3.1.5 (Positions)**.** It follows from the definitions of terms and formulas that they have tree-like structure. For referring to a certain subtree, called subterm or subformula, respectively, sequences of natural numbers are used, called *positions* (as introduced in Chapter 2.1.3). The set of positions of a term, formula is inductively defined by:

$$
\begin{array}{rl}
\mathrm{pos}(x) & := \{\epsilon\} \text{ if } x \in \mathcal{X} \\
\mathrm{pos}(\phi) & := \{\epsilon\} \text{ if } \phi \in \{\top, \bot\} \\
\mathrm{pos}(\neg\phi) & := \{\epsilon\} \cup \{1p \mid p \in \mathrm{pos}(\phi)\} \\
\mathrm{pos}(\phi \circ \psi) & := \{\epsilon\} \cup \{1p \mid p \in \mathrm{pos}(\phi)\} \cup \{2p \mid p \in \mathrm{pos}(\psi)\} \\
\mathrm{pos}(s \approx t) & := \{\epsilon\} \cup \{1p \mid p \in \mathrm{pos}(s)\} \cup \{2p \mid p \in \mathrm{pos}(t)\} \\
\mathrm{pos}(f(t_1, \ldots, t_n)) & := \{\epsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in pos(t_i)\} \\
\mathrm{pos}(P(t_1, \ldots, t_n)) & := \{\epsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in pos(t_i)\} \\
\mathrm{pos}(\forall x.\phi) & := \{\epsilon\} \cup \{1p \mid p \in \mathrm{pos}(\phi)\} \\
\mathrm{pos}(\exists x.\phi) & := \{\epsilon\} \cup \{1p \mid p \in \mathrm{pos}(\phi)\}
\end{array}
$$

where $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ and $t_i \in T(\Sigma, \mathcal{X})$ for all $i \in \{1, \ldots, n\}$.

The *prefix orders* (above, strictly above and parallel), the selection and replacement with respect to positions are defined exactly as in Chapter 2.1.3.

An term t (formula ϕ) is said to *contain* another term s (formula ψ) if $t_p = s$ ($\phi_p = \psi$). It is called a *strict subexpression* if $p \neq \epsilon$. The term t (formula ϕ) is called an *immediate subexpression* of s (formula ψ) if $|p| = 1$. For terms a subexpression is called a *subterm* and for formulas a *subformula*, respectively.

The *size* of a term t (formula ϕ), written $|t|$ ($|\phi|$), is the cardinality of $\mathrm{pos}(t)$, i.e., $|t| := |\mathrm{pos}(t)|$ ($|\phi| := |\mathrm{pos}(\phi)|$). The *depth* of a term, formula is the maximal length of a position in the term, formula: $\mathrm{depth}(t) := max\{|p| \mid p \in \mathrm{pos}(t)\}$ ($\mathrm{depth}(\phi) := max\{|p| \mid p \in \mathrm{pos}(\phi)\}$). The set of *all* variables occurring in a term t (formula ϕ) is denoted by $\mathrm{vars}(t)$ ($\mathrm{vars}(phi)$) and formally defined as $\mathrm{vars}(t) := \{x \in \mathcal{X} \mid x = t|_p, p \in \mathrm{pos}(t)\}$ ($\mathrm{vars}(\phi) := \{x \in \mathcal{X} \mid x = \phi|_p, p \in \mathrm{pos}(\phi)\}$). A term t (formula ϕ) is *ground* if $vars(t) = \emptyset$ ($vars(\phi) = \emptyset$).

Note that $\mathrm{vars}(\forall x.a \approx b) = \emptyset$ where a, b are constants. This is justified by the fact that the formula does not depend on the quantifier, see semantics below. The set of *free* variables of a formula ϕ (term t) is given by $\mathrm{fvars}(\phi, \emptyset)$ ($\mathrm{fvars}(t, \emptyset)$) and recursively defined by $\mathrm{fvars}(\psi_1 \circ \psi_2, B) := \mathrm{fvars}(\psi_1, B) \cup \mathrm{fvars}(\psi_2, B)$ where $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$, $\mathrm{fvars}(\forall x.\psi, B) := \mathrm{fvars}(\psi, B \cup \{x\})$, $\mathrm{fvars}(\exists x.\psi, B) := \mathrm{fvars}(\psi, B \cup \{x\})$, $\mathrm{fvars}(\neg\psi, B) := \mathrm{fvars}(\psi, B)$, $\mathrm{fvars}(L, B) := \mathrm{vars}(L) \setminus B$ ($\mathrm{fvars}(t, B) := \mathrm{vars}(t) \setminus B$. For $\mathrm{fvars}(\phi, \emptyset)$ I also write $\mathrm{fvars}(\phi)$

In $\forall x.\phi$ ($\exists x.\phi$) the formula ϕ is called the *scope* of the quantifier. An occurrence q of a variable x in a formula ϕ ($\phi|_q = x$) is called *bound* if there is some $p < q$ with $\phi|_p = \forall x.\phi'$ or $\phi|_p = \exists x.\phi'$. Any other occurrence of a variable is called *free*. A formula not containing a free occurrence of a variable is called *closed*. If $\{x_1, \ldots, x_n\}$ are the variables freely occurring in a formula ϕ then $\forall x_1, \ldots, x_n.\phi$ and $\exists x_1, \ldots, x_n.\phi$ (abbreviations for $\forall x_1.\forall x_2 \ldots \forall x_n.\phi$, $\exists x_1.\forall x_2 \ldots \forall x_n.\phi$, respectively) are the *universal* and the *existential closure* of ϕ.

Example 3.1.6. For the literal $\neg P(f(x, g(a)))$ the atom $P(f(x, g(a)))$ is an immediate subformula of occurring at position 1. The terms x and $g(a)$ are strict subterms occurring at positions 111 and 112, respectively. The formula $\neg P(f(x, g(a)))[b]_{111} = \neg P(f(b, g(a)))$ is obtained by replacing x with b. $\text{pos}(\neg P(f(x, g(a)))) = \{\epsilon, 1, 11, 111, 112, 1121\}$ meaning its size is 6, its depth 4 and $vars(\neg P(f(x, g(a)))) = \{x\}$.

The *polarity* of a subformula $\psi = \phi|_p$ at position p is $pol(\phi, p)$ where pol is recursively defined by

$$
\begin{aligned}
\text{pol}(\phi, \epsilon) &:= 1 \\
\text{pol}(\neg\phi, 1p) &:= -\text{pol}(\phi, p) \\
\text{pol}(\phi_1 \circ \phi_2, ip) &:= \text{pol}(\phi_i, p) \text{ if } \circ \in \{\wedge, \vee\} \\
\text{pol}(\phi_1 \rightarrow \phi_2, 1p) &:= -\text{pol}(\phi_1, p) \\
\text{pol}(\phi_1 \rightarrow \phi_2, 2p) &:= \text{pol}(\phi_2, p) \\
\text{pol}(\phi_1 \leftrightarrow \phi_2, ip) &:= 0 \\
\text{pol}(P(t_1, \ldots, t_n), p) &:= 1 \\
\text{pol}(t \approx s, p) &:= 1 \\
\text{pol}(\forall x.\phi, 1p) &:= \text{pol}(\phi, p) \\
\text{pol}(\exists x.\phi, 1p) &:= \text{pol}(\phi, p)
\end{aligned}
$$

3.2 Semantics

Definition 3.2.1 (Σ-algebra). Let $\Sigma = (\mathcal{S}, \Omega, \Pi)$ be a signature with set of sorts $\mathcal{S}$, operator set Ω and predicate set Π. A Σ*-algebra* $\mathcal{A}$, also called Σ-*interpretation*, is a mapping that assigns (i) a non-empty carrier set $S^{\mathcal{A}}$ to every sort $S \in \mathcal{S}$, so that $(S_1)^{\mathcal{A}} \cap (S_2)^{\mathcal{A}} = \emptyset$ for any distinct sorts $S_1, S_2 \in \mathcal{S}$, (ii) a total function $f^{\mathcal{A}} : (S_1)^{\mathcal{A}} \times \ldots \times (S_n)^{\mathcal{A}} \rightarrow (S)^{\mathcal{A}}$ to every operator $f \in \Omega$, $\text{arity}(f) = n$ where $f : S_1 \times \ldots \times S_n \rightarrow S$, (iii) a relation $P^{\mathcal{A}} \subseteq ((S_1)^{\mathcal{A}} \times \ldots \times (S_m)^{\mathcal{A}})$ to every predicate symbol $P \in \Pi$, $\text{arity}(P) = m$. (iv) the equality relation becomes $\approx^{\mathcal{A}} = \{(e, e) \mid e \in \mathcal{U}^{\mathcal{A}}\}$ where the set $\mathcal{U}^{\mathcal{A}} := \bigcup_{S \in \mathcal{S}} (S)^{\mathcal{A}}$ is called the *universe* of $\mathcal{A}$.

A (variable) *assignment*, also called a *valuation* for an algebra $\mathcal{A}$ is a function $\beta : \mathcal{X} \rightarrow \mathcal{U}_{\mathcal{A}}$ so that $\beta(x) \in S_{\mathcal{A}}$ for every variable $x \in \mathcal{X}$, where $S = \text{sort}(x)$. A *modification* $\beta[x \mapsto e]$ of an assignment β at a variable $x \in \mathcal{X}$, where $e \in S_{\mathcal{A}}$ and $S = \text{sort}(x)$, is the assignment defined as follows:

$$
\beta[x \mapsto e](y) = \begin{cases} e & \text{if } x = y \\ \beta(y) & \text{otherwise.} \end{cases}
$$

Informally speaking, the assignment $\beta[x \mapsto e]$ is identical to β for every variable except x, which is mapped by $\beta[x \mapsto e]$ to e.

The homomorphic extension $\mathcal{A}(\beta)$ of β onto terms is a mapping $T(\Sigma, \mathcal{X}) \rightarrow \mathcal{U}_{\mathcal{A}}$ defined as (i) $\mathcal{A}(\beta)(x) = \beta(x)$, where $x \in \mathcal{X}$ and (ii) $\mathcal{A}(\beta)(f(t_1, \ldots, t_n)) = f_{\mathcal{A}}(\mathcal{A}(\beta)(t_1), \ldots, \mathcal{A}(\beta)(t_n))$, where $f \in \Omega$, $\text{arity}(f) = n$.

Given a term $t \in T(\Sigma, \mathcal{X})$, the value $\mathcal{A}(\beta)(t)$ is called the *interpretation* of t under $\mathcal{A}$ and β. If the term t is ground, the value $\mathcal{A}(\beta)(t)$ does not depend on a particular choice of β, for which reason the interpretation of t under $\mathcal{A}$ is denoted by $\mathcal{A}(t)$.

An algebra $\mathcal{A}$ is called *term-generated*, if every element e of the universe $\mathcal{U}_\mathcal{A}$ of $\mathcal{A}$ is the image of some ground term t, i.e., $\mathcal{A}(t) = e$.

Definition 3.2.2 (Semantics)**.** An algebra $\mathcal{A}$ and an assignment β are extended to formulas $\phi \in \mathrm{FOL}(\Sigma, \mathcal{X})$ by

$$\begin{array}{rcl}
\mathcal{A}(\beta)(\bot) & := & 0 \\
\mathcal{A}(\beta)(\top) & := & 1 \\
\mathcal{A}(\beta)(s \approx t) & := & 1 \text{ if } \mathcal{A}(\beta)(s) = \mathcal{A}(\beta)(t) \text{ and } 0 \text{ otherwise} \\
\mathcal{A}(\beta)(P(t_1, \ldots, t_n)) & := & 1 \text{ if } (\mathcal{A}(\beta)(t_1), \ldots, \mathcal{A}(\beta)(t_n)) \in P_\mathcal{A} \text{ and } 0 \text{ otherwise} \\
\mathcal{A}(\beta)(\neg\phi) & := & 1 - \mathcal{A}(\beta)(\phi) \\
\mathcal{A}(\beta)(\phi \land \psi) & := & \min(\{\mathcal{A}(\beta)(\phi), \mathcal{A}(\beta)(\psi)\}) \\
\mathcal{A}(\beta)(\phi \lor \psi) & := & \max(\{\mathcal{A}(\beta)(\phi), \mathcal{A}(\beta)(\psi)\}) \\
\mathcal{A}(\beta)(\phi \rightarrow \psi) & := & \max(\{(1 - \mathcal{A}(\beta)(\phi)), \mathcal{A}(\beta)(\psi)\}) \\
\mathcal{A}(\beta)(\phi \leftrightarrow \psi) & := & \text{if } \mathcal{A}(\beta)(\phi) = \mathcal{A}(\beta)(\psi) \text{ then } 1 \text{ else } 0 \\
\mathcal{A}(\beta)(\exists x_S.\phi) & := & 1 \text{ if } \mathcal{A}(\beta[x \mapsto e])(\phi) = 1 \text{ for some } e \in S_\mathcal{A} \text{ and } 0 \text{ otherwise} \\
\mathcal{A}(\beta)(\forall x_S.\phi) & := & 1 \text{ if } \mathcal{A}(\beta[x \mapsto e])(\phi) = 1 \text{ for all } e \in S_\mathcal{A} \text{ and } 0 \text{ otherwise}
\end{array}$$

A formula ϕ is called *satisfiable by* $\mathcal{A}$ *under* β (or *valid in* $\mathcal{A}$ *under* β) if $\mathcal{A}, \beta \models \phi$; in this case, ϕ is also called *consistent*; *satisfiable by* $\mathcal{A}$ if $\mathcal{A}, \beta \models \phi$ for some assignment β; *satisfiable* if $\mathcal{A}, \beta \models \phi$ for some algebra $\mathcal{A}$ and some assignment β; *valid in* $\mathcal{A}$, written $\mathcal{A} \models \phi$, if $\mathcal{A}, \beta \models \phi$ for any assignment β; in this case, $\mathcal{A}$ is called a *model* of ϕ; *valid*, written $\models \phi$, if $\mathcal{A}, \beta \models \phi$ for any algebra $\mathcal{A}$ and any assignment β; in this case, ϕ is also called a *tautology*; *unsatisfiable* if $\mathcal{A}, \beta \not\models \phi$ for any algebra $\mathcal{A}$ and any assignment β; in this case ϕ is also called *inconsistent*.

Note that $\bot$ is *inconsistent* whereas $\top$ is valid. If ϕ is a sentence that is a formula not containing a free variable, it is valid in $\mathcal{A}$ if and only if it is satisfiable by $\mathcal{A}$. This means the truth of a sentence does not depend on the choice of an assignment.

Given two formulas ϕ and ψ, ϕ *entails* ψ, or ψ is a *consequence* of ϕ, written $\phi \models \psi$, if for any algebra $\mathcal{A}$ and assignment β, if $\mathcal{A}, \beta \models \phi$ then $\mathcal{A}, \beta \models \psi$. The formulas ϕ and ψ are called *equivalent*, written $\phi \models\!\!\mid \psi$, if $\phi \models \psi$ and $\psi \models \phi$. Two formulas ϕ and ψ are called *equisatisfiable*, if ϕ is satisfiable iff ψ is satisfiable (not necessarily in the same models). Note that if ϕ and ψ are equivalent then they are equisatisfiable, but not the other way around. The notions of “entailment”, “equivalence” and “equisatisfiability” are naturally extended to sets of formulas, that are treated as conjunctions of single formulas. Thus, given formula sets M_1 and M_2, the set M_1 entails M_2, written $M_1 \models M_2$, if for any algebra $\mathcal{A}$ and assignment β, if $\mathcal{A}, \beta \models \phi$ for every $\phi \in M_1$ then $\mathcal{A}, \beta \models \psi$ for every $\psi \in M_2$. The sets M_1 and M_2 are equivalent, written $M_1 \models\!\!\mid M_2$, if $M_1 \models M_2$ and $M_2 \models M_1$. Given an arbitrary formula ϕ and formula set M, $M \models \phi$ is written to denote $M \models \{\phi\}$; analogously, $\phi \models M$ stands for $\{\phi\} \models M$.

Since clauses are implicitly universally quantified disjunctions of literals, a clause C is satisfiable by an algebra $\mathcal{A}$ if for every assignment β there is a literal $L \in C$ with $\mathcal{A}, \beta \models L$. Note that if $C = \{L_1, \ldots, L_k\}$ is a ground clause, i.e., every L_i is a ground literal, then $\mathcal{A} \models C$ if and only if there is a literal L_j in C so that $\mathcal{A} \models L_j$. A clause set N is satisfiable iff all clauses $C \in N$ are satisfiable by the same algebra $\mathcal{A}$. Accordingly, if N and M are two clause sets, $N \models M$ iff every model $\mathcal{A}$ of N is also a model of M.

3.3 Equality

The equality predicate is build into the first-order language in Section 3.1 and not part of the signature. It is a first class citizen. This is the case although it can be actually axiomatized in the language. The motivation is that firstly, many real world problems naturally contain equations. They are a means to define functions. Then predicates over terms model properties of the functions. Secondly, without special treatment in a calculus, it is almost impossible to automatically prove non-trivial properties of a formula containing equations.

In this section I firstly show that any formula can be transformed into a formula where all atoms are equations. Secondly, that any formula containing equations can be transformed into a formula where the equality predicate is replaced by a fresh predicate together with some axioms. In the first case the respective clause sets are equivalent, in the second case the transformation is satisfiability preserving. For the replacement of any predicate R by equations over a fresh function f_R we assume an additional fresh sort Bool with two fresh constants true and false.

InjEq $\chi[R(t_{1,1}, \ldots, t_{1,n})]_{p_1} \ldots [R(t_{m,1}, \ldots, t_{m,n})]_{p_m} \Rightarrow_{\mathrm{IE}} \chi[f_R(t_{1,1}, \ldots, t_{1,n}) \approx \mathrm{true}]_{p_1} \ldots [f_R(t_{m,1}, \ldots, t_{m,n}) \approx \mathrm{true}]_{p_m}$
provided R is a predicate occurring in χ, $\{p_1, \ldots, p_m\}$ are all positions of atoms with predicate R in χ and f_R is new with appropriate sorting

Proposition 3.3.1. Let $\chi \Rightarrow^*_{\mathrm{IE}} \chi'$ then χ is satisfiable (valid) iff χ' is satisfiable (valid).

Proof. (Sketch) The basic proof idea is to establish the relation $(t_1^{\mathcal{A}}, \ldots, t_n^{\mathcal{A}}) \in R^{\mathcal{A}}$ iff $f_R^{\mathcal{A}}(t_1^{\mathcal{A}}, \ldots, t_n^{\mathcal{A}}) = \mathrm{true}^{\mathcal{A}}$. Furthermore, the sort of true is fresh to χ and the equations $f_R(t_1, \ldots, t_n) \approx \mathrm{true}$ do not interfere with any term t_i because the f_R are all fresh and only occur on top level of the equations. □

When removing equality from a formula it needs to be axiomatized. For simplicity, I assume here that the considered formula χ is one-sorted, i.e., there is only one sort occurring for functions, relations in χ. The extension to formulas with many sorts is straightforward and discussed below.

RemEq $\chi[l_1 \approx r_1]_{p_1} \ldots [l_m \approx r_m]_{p_m} \Rightarrow_{\mathrm{RE}} \chi[E(l_1, r_1)]_{p_1} \ldots [E(l_m, r_m)]_{p_m} \land \mathrm{def}(\chi, E)$

provided $\{p_1, \ldots, p_m\}$ are all positions of equations $l_i = r_i$ in χ and E is a new binary predicate

The formula $\text{def}(\chi, E)$ is the axiomatization of equality for χ and it consists of a conjunction of the equivalence relation axioms for E

$\forall x.E(x,x)$
$\forall x,y.(E(x,y) \rightarrow E(y,x))$
$\forall x,y,z.((E(x,y) \wedge E(x,z)) \rightarrow E(x,z))$

plus the congruence axioms for E for every n-ary function symbol f

$\forall x_1,y_1,\ldots,x_n,y_n.((E(x_1,y_1) \wedge \ldots \wedge E(x_n,y_n)) \rightarrow E(f(x_1,\ldots,x_n), f(y_1,\ldots,y_n)))$

plus the congruence axioms for E for every m-ary predicate symbol P

$\forall x_1,y_1,\ldots,x_m,y_m.((E(x_1,y_1) \wedge \ldots \wedge E(x_m,y_m) \wedge P(x_1,\ldots,x_m)) \rightarrow P(y_1,\ldots,y_m)$

Proposition 3.3.2. Let $\chi \Rightarrow_{\text{RE}} \chi'$ then χ is satisfiable iff χ' is satisfiable.

Proof. (Sketch) The identity on an algebra (see Definition 3.2.2) is a congruence relation proving the direction from left to right. The direction from right to left is more involved. □

Note that $\Rightarrow_{\text{RE}}$ is not validity preserving. Consider the simple example formula $a \approx a$ which is valid for any constant a. Its translation $E(a,a) \wedge \text{def}(a \approx a, E)$ is not valid, e.g., consider an algebra with $E^{\mathcal{A}} = \emptyset$.

Now in case χ has many different sorts then for each sort S one new fresh predicate E_S is needed for the translation. For each of these predicates equivalence relation and congruence axioms need to be generated where for every function f only one axiom using E_S is needed, where S is the range sort of S. Similar for the domain sorts of f and accordingly for predicates.

3.4 Substitution and Unifier

Definition 3.4.1 (Substitution)**.** A *substitution* is a mapping $\sigma : \mathcal{X} \rightarrow T(\Sigma, \mathcal{X})$ so that

1. $\sigma(x) \neq x$ for only finitely many variables x and
2. $sort(x) = sort(t)$ for every variable $x \in \mathcal{X}$ that is mapped to a term $t \in T_S(\Sigma, \mathcal{X})$.

The application $\sigma(x)$ of a substitution σ to a variable x is often written in postfix notation as $x\sigma$. The variable set $\text{dom}(\sigma) := \{x \in \mathcal{X} \mid x\sigma \neq x\}$ is called the *domain* of σ. The term set $\text{codom}(\sigma) := \{x\sigma \mid x \in \text{dom}(\sigma)\}$ is called the *codomain* of σ. From the above definition of substitution it follows that $\text{dom}(\sigma)$ is finite for any substitution σ. The composition of two substitutions σ and τ is written as a juxtaposition $\sigma\tau$, i.e., $t\sigma\tau = (t\sigma)\tau$. A substitution σ is called *idempotent* if $\sigma\sigma = \sigma$. σ is idempotent iff $\text{dom}(\sigma) \cap \text{vars}(\text{codom}(\sigma)) = \emptyset$.

Substitutions are often written as $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ if $dom(\sigma) = \{x_1, \ldots, x_n\}$ and $x_i\sigma = t_i$ for every $i \in \{1, \ldots, n\}$. The *modification*of a substitution σ at a variable x is defined as follows:

$$\sigma[x \mapsto t](y) = \begin{cases} t & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

A substitution σ is identified with its extension to expression and defined as following:

1. $\bot\sigma = \bot$,
2. $\top\sigma = \top$,
3. $(f(t_1, \ldots, t_n))\sigma = f(t_1\sigma, \ldots, t_n\sigma)$,
4. $(P(t_1, \ldots, t_n))\sigma = P(t_1\sigma, \ldots, t_n\sigma)$,
5. $(s \approx t)\sigma = (s\sigma \approx t\sigma)$,
6. $(\neg\phi)\sigma = \neg(\phi\sigma)$,
7. $(\phi \circ \psi)\sigma = \phi\sigma \circ \psi\sigma$ where $\circ \in \{\vee, \wedge\}$,
8. $(Qx\phi)\sigma = Qz(\phi\sigma[x \mapsto z])$ where $Q \in \{\forall, \exists\}$, z and x are of the same sort and z is a fresh variable.

The result $e\sigma$ of applying a substitution σ to an expression e is called an *instance* of e. The substitution σ is called *ground* if it maps every domain variable to a ground term. If the application of a substitution σ to an expression e produces a ground expression $e\sigma$ then $e\sigma$ is called *ground instance* of e. A ground substitution σ is called *grounding for an expression* e if $e\sigma$ is ground. A substitution σ is called *variable renaming* if $im(\sigma) \subseteq \mathcal{X}$ and for any $x, y \in \mathcal{X}$, if $x \neq y$ then $x\sigma \neq y\sigma$.

Definition 3.4.2 (Unifier)**.** Two terms s and t are said to be *unifiable* if there exists a substitution σ so that $s\sigma = t\sigma$, the substitution σ is then called a *unifier* of s and t. The unifier σ is called *most general unifier*, written $\sigma = mgu(s,t)$, if any other unifier τ of s and t can be represented as $\tau = \sigma\tau'$, for some substitution τ'.

3.5 Unification Calculi

The first calculus is the naive standard unification calculus that is typically found in the (old) literature on automated reasoning. A state of the naive standard unification calculus is a set of equations E or $\bot$, where $\bot$ denotes that no unifier exists. The set E is also called a *unification problem.* The start state for checking whether two terms s, t with $\text{sort}(s) = \text{sort}(t)$ (or atoms A, B) are unifiable is the set $E = \{s = t\}$. A variable x is solved in E if $E = \{x = t\} \uplus E'$, $x \notin \text{vars}(t)$ and $x \notin \text{vars}(E)$.

Tautology $\quad E \uplus \{t = t\} \;\Rightarrow_{\text{SU}}\; E$

Decomposition $E \uplus \{f(s_1,\ldots,s_n) = f(t_1,\ldots,t_n)\} \;\Rightarrow_{\mathrm{SU}}\; E \cup \{s_1 = t_1,\ldots,s_n = t_n\}$

Clash $E \uplus \{f(s_1,\ldots,s_n) = g(s_1,\ldots,s_m)\} \;\Rightarrow_{\mathrm{SU}}\; \bot$
if $f \neq g$

Substitution $E \uplus \{x = t\} \;\Rightarrow_{\mathrm{SU}}\; E\{x \mapsto t\} \cup \{x = t\}$
if $x \in vars(E)$ and $x \notin vars(t)$

Occurs Check $E \uplus \{x = t\} \;\Rightarrow_{\mathrm{SU}}\; \bot$
if $x \neq t$ and $x \in vars(t)$

Orient $E \uplus \{t = x\} \;\Rightarrow_{\mathrm{SU}}\; E \cup \{x = t\}$
if $t \notin \mathcal{X}$

Theorem 3.5.1 (Soundness, Completeness and Termination of $\Rightarrow_{\mathrm{SU}}$)**.** If s, t are two terms with $\mathrm{sort}(s) = \mathrm{sort}(t)$ then

1. if $\{s = t\} \Rightarrow^*_{\mathrm{SU}} E$ then any equation $(s' = t') \in E$ is well-sorted, i.e., $\mathrm{sort}(s') = \mathrm{sort}(t')$.
2. $\Rightarrow_{\mathrm{SU}}$ terminates on $\{s = t\}$.
3. if $\{s = t\} \Rightarrow^*_{\mathrm{SU}} E$ then σ is a unifier (mgu) of E iff σ is a unifier (mgu) of $\{s = t\}$.
4. if $\{s = t\} \Rightarrow^*_{\mathrm{SU}} \bot$ then s and t are not unifiable.
5. if $\{s = t\} \Rightarrow^*_{\mathrm{SU}} \{x_1 = t_1,\ldots,x_n = t_n\}$ and this is a normal form, then $\{x_1 \mapsto t_1,\ldots,x_n \mapsto t_n\}$ is an mgu of s, t.

Proof. 1. by induction on the length of the derivation and a case analysis for the different rules.
2. for a state $E = \{s_1 = t_1,\ldots,s_n = t_n\}$ take the measure $\mu(E) := (n, M, k)$ where n is the number of unsolved variables, M the multiset of all term depths of the s_i, t_i and k the number of equations $t = x$ in E where t is not a variable. The state $\bot$ is mapped to $(0, \emptyset, 0)$. Then the lexicographic combination of $>$ on the naturals and its multiset extension shows that any rule application decrements the measure.
3. by induction on the length of the derivation and a case analysis for the different rules. Clearly, for any state where Clash, or Occurs Check generate $\bot$ the respective equation is not unifiable.
4. a direct consequence of 3.
5. if $E = \{x_1 = t_1,\ldots,x_n = t_n\}$ is a normal form, then for all $x_i = t_i$ we have $x_i \notin \mathrm{vars}(t_i)$ and $x_i \notin \mathrm{vars}(E \setminus \{x_i = t_i\})$, so $\{x_1 = t_1,\ldots,x_n = t_n\}\{x_1 \mapsto t_1,\ldots,x_n \mapsto t_n\} = \{t_1 = t_1,\ldots,t_n = t_n\}$ and hence $\{x_1 \mapsto t_1,\ldots,x_n \mapsto t_n\}$ is an mgu of $\{x_1 = t_1,\ldots,x_n = t_n\}$. By 3. it is also an mgu of s, t. □

Example 3.5.2 (Size of Standard Unification Problems)**.** Any normal form of the unification problem E given by
$\{f(x_1, g(x_1, x_1), x_3, \ldots, g(x_n, x_n)) = f(g(x_0, x_0), x_2, g(x_2, x_2), \ldots, x_{n+1})\}$
with respect to $\Rightarrow_{\mathrm{SU}}$ is exponentially larger than E.

The second calculus, polynomial unification, prevents the problem of exponential growth by introducing an implicit representation for the mgu. For this calculus the size of a normal form is always polynomial in the size of the input unification problem.

Tautology $E \uplus \{t = t\} \;\Rightarrow_{\mathrm{PU}}\; E$

Decomposition $E \uplus \{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)\} \;\Rightarrow_{\mathrm{PU}}\; E \uplus \{s_1 = t_1, \ldots, s_n = t_n\}$

Clash $E \uplus \{f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)\} \;\Rightarrow_{\mathrm{PU}}\; \bot$
if $f \neq g$

Occurs Check $E \uplus \{x = t\} \;\Rightarrow_{\mathrm{PU}}\; \bot$
if $x \neq t$ and $x \in vars(t)$

Orient $E \uplus \{t = x\} \;\Rightarrow_{\mathrm{PU}}\; E \uplus \{x = t\}$
if $t \notin \mathcal{X}$

Substitution $E \uplus \{x = y\} \;\Rightarrow_{\mathrm{PU}}\; E\{x \mapsto y\} \uplus \{x = y\}$
if $x \in vars(E)$ and $x \neq y$

Cycle $E \uplus \{x_1 = t_1, \ldots, x_n = t_n\} \;\Rightarrow_{\mathrm{PU}}\; \bot$
if there are positions p_i with $t_i|_{p_i} = x_{i+1}, t_n|_{p_n} = x_1$ and some $p_i \neq \epsilon$

Merge $E \uplus \{x = t, x = s\} \;\Rightarrow_{\mathrm{PU}}\; E \uplus \{x = t, t = s\}$
if $t, s \notin \mathcal{X}$ and $|t| \leq |s|$

Theorem 3.5.3 (Soundness, Completeness and Termination of $\Rightarrow_{\mathrm{PU}}$)**.** If s, t are two terms with $\mathrm{sort}(s) = \mathrm{sort}(t)$ then

1. if $\{s = t\} \Rightarrow^*_{\mathrm{PU}} E$ then any equation $(s' = t') \in E$ is well-sorted, i.e., $\mathrm{sort}(s') = \mathrm{sort}(t')$.
2. $\Rightarrow_{\mathrm{PU}}$ terminates on $\{s = t\}$.
3. if $\{s = t\} \Rightarrow^*_{\mathrm{PU}} E$ then σ is a unifier (mgu) of E iff σ is a unifier (mgu) of $\{s = t\}$.
4. if $\{s = t\} \Rightarrow^*_{\mathrm{PU}} \bot$ then s and t are not unifiable.

Theorem 3.5.4 (Unifier generated by $\Rightarrow_{\mathrm{PU}}$)**.** Let $\{s = t\} \Rightarrow^*_{\mathrm{PU}} \{x_1 = t_1, \ldots, x_n = t_n\}$. Then

γ	Descendant $\gamma(t)$
$\forall x_S.\psi$	$\psi\{x_S \mapsto t\}$
$\neg\exists x_S.\psi$	$\neg\psi\{x_S \mapsto t\}$
	for any ground term $t \in T_S(\Sigma)$

δ	Descendant $\delta(c)$
$\exists x_S.\psi$	$\psi\{x_S \mapsto c\}$
$\neg\forall x_S.\psi$	$\neg\psi\{x_S \mapsto c\}$
	for some fresh Skolem constant $c \in T_S(\Sigma)$

Figure 3.1: γ- and δ-Formulas

1. $x_i \neq x_j$ for all $i \neq j$ and without loss of generality $x_i \notin \text{vars}(t_{i+k})$ for all i, k, $1 \leq i < n$, $i + k \leq n$.
2. the substitution $\{x_1 \mapsto t_1\}\{x_2 \mapsto t_2\} \ldots \{x_n \mapsto t_n\}$ is an mgu of $s = t$.

Proof. 1. If $x_i = x_j$ for some $i \neq j$ then Merge is applicable. If $x_i \in \text{vars}(t_i)$ for some i then Occurs Check is applicable. If the x_i cannot be ordered in the described way, then either Substitution or Cycle is applicable.
2. Since $x_i \notin \text{vars}(t_{i+k}$ the composition yields the mgu. ☐

3.6 First-Order Tableaux

The different versions of tableaux for first-order logic differ in particular in the treatment of variables by the tableaux rules. The first variant is standard first-order tableaux where variables are instantiated by ground terms.

Definition 3.6.1 (γ-,δ-Formulas)**.** A formula ϕ is called a γ-formula if ϕ is a formula $\forall x_S.\psi$ or $\neg\exists x_S.\psi$. A formula ϕ is called a δ-formula if ϕ is a formula $\exists x_S.\psi$ or $\neg\forall x_S.\psi$.

Definition 3.6.2 (Direct Standard Tableaux Descendant)**.** Given a γ- or δ-formula ϕ, Figure 3.1 shows its direct descendants.

For the standard first-order tableaux rules to make sense "enough" Skolem constants are needed in the signature, e.g., countably infinitely many for each sort. A δ formula ϕ occurring in some sequence is called *open* if no direct descendant of it is part of the sequence. In general, the number of γ descendants cannot be limited for a successful tableaux proof.

γ-Expansion $\quad N \uplus \{(\phi_1, \ldots, \psi, \ldots, \phi_n)\} \Rightarrow_{\text{FT}} N \uplus \{(\phi_1, \ldots, \psi, \ldots, \phi_n, \psi')\}$
provided ψ is a γ-formula, ψ' a $\gamma(t)$ descendant where t is an arbitrary ground term in the signature of the sequence (branch) and the sequence is not closed.

δ-Expansion $\quad N \uplus \{(\phi_1, \ldots, \psi, \ldots, \phi_n)\} \Rightarrow_{\text{FT}} N \uplus \{(\phi_1, \ldots, \psi, \ldots, \phi_n, \psi')\}$

provided ψ is an open δ-formula, ψ' a $\delta(c)$ descendant where c is fresh to the sequence and the sequence is not closed.

The standard first-order tableaux calculus consists of the rules α-, and β-expansion (see Section 2.5) and the above two rules γ-Expansion and δ-Expansion.

Theorem 3.6.3 (Standard First-Order Tableaux is Sound and Complete)**.** A formula ϕ (without equality) is valid iff standard tableaux computes a closed state out of $\{(\neg\phi)\}$.

Skolem *constants* are sufficient: In a δ-formula $\exists x\,\phi$, $\exists$ is the outermost quantifier and x is the only free variable in ϕ. The γ rule has to be applied several times to the same formula for tableaux to be complete. For instance, constructing a closed tableau for

$$\{\forall x\,(P(x) \rightarrow P(f(x))),\ P(b),\ \neg P(f(f(b)))\}$$

is impossible without applying γ-expansion twice on one path.

The main disadvantage of standard first-order tableau is that the γ ground term instances need to be guessed. The whole complexity of the problem lies in this guessing as for otherwise tableaux terminates. A natural idea is to guess ground terms that can eventually be used to close a branch. This is the idea of free-variable first-order tableaux. Instead of guessing a ground term for a γ formula the variable remains, the instantiation is delayed until a branch is closed for two literals via unification. As a consequence, for δ formulas no longer constants are introduced but Skolem terms in the formerly universally quantified variables that had the δ formula in their scope.

The new calculus suggests to keep track of scopes of variables, so I move from a state as a set of sequences of formulas to a set of sequences of pairs $l_i = (\phi_i, X_i)$ where X_i is a set of variables.

Definition 3.6.4 (Direct Free-Variable Tableaux Descendant)**.** Given a γ- or δ-formula ϕ, Figure 3.2 shows its direct descendants.

γ-Expansion $\quad N \uplus \{(l_1, \ldots, (\psi, X), \ldots, l_n)\} \Rightarrow_{\mathrm{FT}} N \uplus \{(l_1, \ldots, (\psi, X), \ldots, l_n, (\psi', X \cup \{y\}))\}$

provided ψ is a γ-formula, ψ' a $\gamma(y)$ descendant where y is fresh to the sequence (branch) and the sequence is not closed.

δ-Expansion $\quad N \uplus \{(l_1, \ldots, (\psi, X), \ldots, l_n)\} \Rightarrow_{\mathrm{FT}} N \uplus \{(l_1, \ldots, (\psi, X), \ldots, l_n, (\psi', X))\}$

provided ψ is an open δ-formula, ψ' a $\delta(f(y_1, \ldots, y_n))$ descendant where f is fresh to the sequence, $X = \{y_1, \ldots, y_n\}$ and the sequence is not closed.

Branch-Closing $\quad N \uplus \{(l_1, \ldots, (L, X), \ldots, (K, X'), \ldots, l_n)\} \Rightarrow_{\mathrm{FT}} N\sigma \uplus \{(l_1, \ldots, (L, X), \ldots, (K, X'), \ldots, l_n, \}\sigma$

γ	Descendant $\gamma(y)$
$\forall x_S.\psi$	$\psi\{x_S \mapsto y\}$
$\neg\exists x_S.\psi$	$\neg\psi\{x_S \mapsto y\}$
	for a fresh variable $y, \mathrm{sort}(y) = S$

δ	Descendant $\delta(f(y_1, \ldots, y_n))$
$\exists x_S.\psi$	$\psi\{x_S \mapsto f(y_1, \ldots, y_n)\}$
$\neg\forall x_S.\psi$	$\neg\psi\{x_S \mapsto f(y_1, \ldots, y_n)\}$
	for some fresh Skolem function f
	where $f(y_1, \ldots, y_n) \in T_S(\Sigma, \mathcal{X})$

Figure 3.2: γ- and δ-Formulas

provided K and L are literals and there is an mgu σ such that $K\sigma = \neg L\sigma$ and the sequence is not closed.

The standard first-order tableaux calculus consists of the rules α-, and β-expansion (see Section 2.5) which are adapted to pairs and the above three rules γ-Expansion, δ-Expansion and Branch-Closing.

Theorem 3.6.5 (Free-variable First-Order Tableaux is Sound and Complete)**.** A formula ϕ (without equality) is valid iff free-variable tableaux computes a closed state out of $\{(\neg\phi)\}$.

Example 3.6.6.

1.	$\neg[\exists w \forall x R(x, w, f(x, w)) \rightarrow \exists w \forall x \exists y R(x, w, y)]$	
2.	$\exists w \forall x\ R(x, w, f(x, w))$	1_1 $[\alpha]$
3.	$\neg\exists w \forall x \exists y\ R(x, w, y)$	1_2 $[\alpha]$
4.	$\forall x\ R(x, c, f(x, c))$	$2(c)$ $[\delta]$
5.	$\neg\forall x \exists y\ R(x, v_1, y)$	$3(v_1)$ $[\gamma]$
6.	$\neg\exists y\ R(g(v_1), v_1, y)$	$5(g(v_1))$ $[\delta]$
7.	$R(v_2, c, f(v_2, c))$	$4(v_2)$ $[\gamma]$
8.	$\neg R(g(v_1), v_1, v_3)$	$6(v_3)$ $[\gamma]$

7. and 8. are complementary (modulo unification):

$$v_2 = g(v_1),\ c = v_1,\ f(v_2, c) = v_3$$

is solvable with an mgu $\sigma = \{v_1 \mapsto c, v_2 \mapsto g(c), v_3 \mapsto f(g(c), c)\}$, and hence, $T\sigma$ is a closed (linear) tableau for the formula in 1.

Problem: Strictness for γ is still incomplete. For instance, constructing a closed tableau for

$$\{\forall x\,(P(x) \rightarrow P(f(x))),\ P(b),\ \neg P(f(f(b)))\}$$

is impossible without applying γ-expansion twice on one path.

Semantic Tableau vs. Resolution

1. Tableau: global, goal-oriented, "backward".
2. Resolution: local, "forward".
3. Goal-orientation is a clear advantage if only a small subset of a large set of formulas is necessary for a proof. (Note that resolution provers saturate also those parts of the clause set that are irrelevant for proving the goal.)
4. Resolution can be combined with more powerful redundancy elimination methods; because of its global nature this is more difficult for the tableau method.
5. Resolution can be refined to work well with equality; for tableau this seems to be impossible.
6. On the other hand tableau calculi can be easily extended to other logics; in particular tableau provers are very successful in modal and description logics.

3.7 First-Order CNF Transformation

Similar to the propositional case, first-order superposition operates on clauses. In this section I show how any first-order sentence can be efficiently transformed into a CNF, preserving satisfiability. To this end all existentially quantified variables are replaced with so called Skolem functions. Similar to renaming this replacement only preserves satisfiability. Eventually, all variables in clauses are implicitly universally quantified.

As usual, the CNF transformation is done by a set of rules. All rules known from the propositional case apply. Further rules deal with the quantifies $\forall$, $\exists$ and some of the propositional rules need an extension in order to cope with first-order variables.

The first set of rules eliminates $\top$ and $\bot$ from a first-order formula.

ElimTB1 $\chi[(\phi \land \top)]_p \Rightarrow_{\text{CNF}} \chi[\phi]_p$

ElimTB2 $\chi[(\phi \land \bot)]_p \Rightarrow_{\text{CNF}} \chi[\bot]_p$

ElimTB3 $\chi[(\phi \lor \top)]_p \Rightarrow_{\text{CNF}} \chi[\top]_p$

ElimTB4 $\chi[(\phi \lor \bot)]_p \Rightarrow_{\text{CNF}} \chi[\phi]_p$

ElimTB5 $\chi[\neg\bot]_p \Rightarrow_{\text{CNF}} \chi[\top]_p$

ElimTB6 $\chi[\neg\top]_p \Rightarrow_{\text{CNF}} \chi[\bot]_p$

ElimTB7 $\chi[\phi \leftrightarrow \bot]_p \ \Rightarrow_{\text{CNF}} \ \chi[\neg\phi]_p$

ElimTB8 $\chi[\phi \leftrightarrow \top]_p \ \Rightarrow_{\text{CNF}} \ \chi[\phi]_p$

ElimTB9 $\chi[\phi \rightarrow \bot]_p \ \Rightarrow_{\text{CNF}} \ \chi[\neg\phi]_p$

ElimTB10 $\chi[\phi \rightarrow \top]_p \ \Rightarrow_{\text{CNF}} \ \chi[\top]_p$

ElimTB11 $\chi[\bot \rightarrow \phi]_p \ \Rightarrow_{\text{CNF}} \ \chi[\top]_p$

ElimTB12 $\chi[\top \rightarrow \phi]_p \ \Rightarrow_{\text{CNF}} \ \chi[\phi]_p$

ElimTB13 $\chi[\{\forall, \exists\}x.\top]_p \ \Rightarrow_{\text{CNF}} \ \chi[\top]_p$

ElimTB14 $\chi[\{\forall, \exists\}x.\bot]_p \ \Rightarrow_{\text{CNF}} \ \chi[\bot]_p$

where the expression $\{\forall, \exists\}x.\phi$ covers both cases $\forall x.\phi$ and $\exists x.\phi$. The next step is to rename all variable such that different quantifiers bind different variables. This step is necessary to prevent a later on confusion of variables.

RenVar $\phi \ \Rightarrow_{\text{CNF}} \ \phi\sigma$
for $\sigma = \{\}$

Once the variable renaming is done, renaming of beneficial subformulas is the next step. The mechanism of renaming and the concept of a beneficial subformula is exactly the same as in propositional logic. The only difference is that renaming does introduce an atom in the free variables of the respective subformula. When some formula ψ is renamed at position p an atom $P(\vec{x_n})$, $\vec{x_n} = x_1, \ldots, x_n$ replaces $\psi|_p$ where $\text{fvars}(\psi|_p) = \{x_1 \ldots, x_n\}$. The respective definition of $P(\vec{x_n})$ becomes

$$\text{def}(\psi, p, P(\vec{x_n})) := \begin{cases} \forall \vec{x_n}.(P(\vec{x_n}) \rightarrow \psi|_p) & \text{if } \text{pol}(\psi, p) = 1 \\ \forall \vec{x_n}.(\psi|_p \rightarrow P(\vec{x_n})) & \text{if } \text{pol}(\psi, p) = -1 \\ \forall \vec{x_n}.(P(\vec{x_n}) \leftrightarrow \psi|_p) & \text{if } \text{pol}(\psi, p) = 0 \end{cases}$$

and the rule SimpleRenaming is changed accordingly.

SimpleRenaming $\phi \ \Rightarrow_{\text{CNF}} \ \phi[A_1]_{p_1}[A_2]_{p_2} \ldots [A_n]_{p_n} \ \wedge \ \text{def}(\phi, p_1, A_1) \ \wedge \ \ldots \wedge \ \text{def}(\phi[A_1]_{p_1}[A_2]_{p_2} \ldots [A_{n-1}]_{p_{n-1}}, p_n, A_n)$
provided $\{p_1, \ldots, p_n\} \subset \text{pos}(\phi)$ and for all $i, i+j$ either $p_i \parallel p_{i+j}$ or $p_i > p_{i+j}$ and the $A_i = P_i(x_{i,1}, \ldots, x_{i,k_i})$ where $\text{fvars}(\phi|_{p_i}) = \{x_{i,1}, \ldots, x_{i,k_i}\}$ and all P_i are different and new to ϕ

Negation normal form is again done as in the propositional case with additional rules for the quantifiers.

ElimEquiv1 $\chi[(\phi \leftrightarrow \psi)]_p \;\Rightarrow_{\text{CNF}}\; \chi[(\phi \rightarrow \psi) \land (\psi \rightarrow \phi)]_p$
provided $\text{pol}(\chi, p) \in \{0, 1\}$

ElimEquiv2 $\chi[(\phi \leftrightarrow \psi)]_p \;\Rightarrow_{\text{CNF}}\; \chi[(\phi \land \psi) \lor (\neg\phi \land \neg\psi)]_p$
provided $\text{pol}(\chi, p) = -1$

ElimImp $\chi[(\phi \rightarrow \psi)]_p \;\Rightarrow_{\text{CNF}}\; \chi[(\neg\phi \lor \psi)]_p$

PushNeg1 $\chi[\neg(\phi \lor \psi)]_p \;\Rightarrow_{\text{CNF}}\; \chi[(\neg\phi \land \neg\psi)]_p$

PushNeg2 $\chi[\neg(\phi \land \psi)]_p \;\Rightarrow_{\text{CNF}}\; \chi[(\neg\phi \lor \neg\psi)]_p$

PushNeg3 $\chi[\neg\neg\phi]_p \;\Rightarrow_{\text{CNF}}\; \chi[\phi]_p$

PushNeg4 $\chi[\neg\forall x.\phi]_p \;\Rightarrow_{\text{CNF}}\; \chi[\exists x.\neg\phi]_p$

PushNeg5 $\chi[\neg\exists x.\phi]_p \;\Rightarrow_{\text{CNF}}\; \chi[\forall x.\neg\phi]_p$

In propositional logic after NNF, the CNF can be generated using distributivity. In first-order logic the existential quantifiers are eliminated first by the introduction of Skolem functions. In order to receive Skolem functions with few arguments, the quantifiers are first moved inwards as far as passible. This step is called *mini-scoping*.

MiniScope1 $\chi[\forall x.(\psi_1 \circ \psi_2)]_p \;\Rightarrow_{\text{CNF}}\; \chi[(\forall x.\psi_1) \circ \psi_2]_p$
provided $\circ \in \{\land, \lor\}$, $x \notin \text{fvars}(\psi_2)$

MiniScope2 $\chi[\exists x.(\psi_1 \circ \psi_2)]_p \;\Rightarrow_{\text{CNF}}\; \chi[(\exists x.\psi_1) \circ \psi_2]_p$
provided $\circ \in \{\land, \lor\}$, $x \notin \text{fvars}(\psi_2)$

MiniScope3 $\chi[\forall x.(\psi_1 \land \psi_2)]_p \;\Rightarrow_{\text{CNF}}\; \chi[(\forall x.\psi_1) \land (\forall x.\psi_2)\sigma]_p$
where $\sigma = \{\}$, $x \in (\text{fvars}(\psi_1) \cap \text{fvars}(\psi_2))$

MiniScope4 $\chi[\exists x.(\psi_1 \lor \psi_2)]_p \;\Rightarrow_{\text{CNF}}\; \chi[(\exists x.\psi_1) \lor (\exists x.\psi_2)\sigma]_p$
where $\sigma = \{\}, x \in (\text{fvars}(\psi_1) \cap \text{fvars}(\psi_2))$

The rules MiniScope1, MiniScope2 are applied modulo the commutativity of $\wedge$, $\vee$. Once the quantifiers are moved inwards Skolemization can take place.

Skolemization $\chi[\exists x, \psi]_p \Rightarrow_{\text{CNF}} \chi[\psi\{x \mapsto f(y_1, \ldots, y_n)\}]_p$
provided there is no q, $q < p$ with $\phi|_q = \exists x'.\psi'$, $\text{fvars}(\exists x.\psi) = \{y_1, \ldots, y_n\}$, $\text{arity}(f) = n$ is a new function symbol to ϕ matching the respective sorts of the y_i with range sort $\text{sort}(x)$

Example 3.7.1 (Mini-Scoping and Skolemization)**.** Consider the simple formula $\forall x.\exists y.(R(x,x) \wedge P(y)$. Applying Skolemization directly to this formula, without mini-scoping results in

$$\forall x.\exists y.(R(x,x) \wedge P(y)) \Rightarrow_{\text{CNF,Skolemization}} \forall x.(R(x,x) \wedge P(g(x))$$

for a unary Skolem function g because $\text{fvars}(\exists y.(R(x,x) \wedge P(y))) = \{x\}$. Applying mini-scoping and then Skolemization generates

$$\begin{array}{lll} \forall x.\exists y.(R(x,x) \wedge P(y)) & \Rightarrow^*_{\text{CNF,MiniScope2,1}} & \forall x.R(x,x) \wedge \exists y.P(y) \\ & \Rightarrow_{\text{CNF,Skolemization}} & \forall x.R(x,x) \wedge P(a) \end{array}$$

for some Skolem constant a because $\text{fvars}(\exists y.P(y)) = \emptyset$. Now the former formula after Skolemization is seriously more complex than the latter. The former belongs to an undecidable fragment of first-order logic while the latter belongs to a decidable one (see Section 3.14).

Finally, the universal quantifiers are removed. In a first-order logic CNF any variable is universally quantified by default. Furthermore, the variables of two different clauses are always assumed to be different.

RemForall $\chi[\forall x.\psi]_p \Rightarrow_{\text{CNF}} \chi[\psi]_p$

The actual CNF is then done by distributivity.

PushDisj $\chi[(\phi_1 \wedge \phi_2) \vee \psi]_p \Rightarrow_{\text{CNF}} \chi[(\phi_1 \vee \psi) \wedge (\phi_2 \vee \psi)]_p$

Theorem 3.7.2 (Properties of the CNF Transformation)**.** Let ϕ be a first-order sentence, then

1. $\text{cnf}(\phi)$ terminates
2. ϕ is satisfiable iff $\text{cnf}(\phi)$ is satisfiable

Proof. (Idea) 1. is a straightforward extension of the propositional case. It is easy to define a measure for any line of Algorithm 6.
2. can also be established separately for all rule applications. The rules SimpleRenaming and Skolemization need separate proofs, the rest is straightforward or copied from the propositional case. □

Algorithm 6: cnf(ϕ)

Input : A first-order formula ϕ.
Output: A formula ψ in CNF satisfiability preserving to ϕ.
1 **whilerule** *(***ElimTB1**(ϕ),...,**ElimTB14**(ϕ)*)* **do** ;
2 **RenVar**(ϕ);
3 **SimpleRenaming**(ϕ) on obvious positions;
4 **whilerule** *(***ElimEquiv1**(ϕ),**ElimEquiv2**(ϕ)*)* **do** ;
5 **whilerule** *(***ElimImp**(ϕ)*)* **do** ;
6 **whilerule** *(***PushNeg1**(ϕ),...,**PushNeg5**(ϕ)*)* **do** ;
7 **whilerule** *(***MiniScope1**(ϕ),...,**MiniScope4**(ϕ)*)* **do** ;
8 **whilerule** *(***Skolemization**(ϕ)*)* **do** ;
9 **whilerule** *(***RemForall**(ϕ)*)* **do** ;
10 **whilerule** *(***PushDisj**(ϕ)*)* **do** ;
11 **return** ϕ;

C

In addition to the consideration of repeated subformulas, discussed in Section 2.6, for first-order renaming another technique can pay off: generalization. Consider the formula $[\phi_1 \lor (Q_1(a_1) \land Q_2(a_1))] \land [\phi_2 \lor (Q_1(a_2) \land Q_2(a_2))] \land \ldots \land [\phi_n \lor (Q_1(a_n) \land Q_2(a_n)]$. SimpleRenaming on obvious renamings applied to this formula will independently rename any occurrences of a formula $(Q_1(a_i) \land Q_2(a_i))$. However generalization pays off here. By adding the definition $\forall x, y\, (R(x, y) \rightarrow (Q_1(x) \land Q_2(y)))$ and replacing the ith occurrence of the conjunct by $R(x, y)\{x \mapsto a_i, y \mapsto a_i\}$ one definition for all subformula occurrences suffices.

3.8 Herbrand Interpretations

For propositional logic the existence of a canonical model is straightforward because the definition of the semantics leads to an effective representation. A propositional variable can be either true or false. For first-order logic this is no longer straightforward because an interpretation can assign any non-empty set to a sort, any function to a function symbol and any relation to a predicate symbol. A giant step forward towards the mechanization of first-order logic was the discovery of a canonical model construction by Herbrand. A first-order formula has a model iff it has such a canonical model which is build out of the syntax.

For this and the following section I restrict the focus to first-order logic without equality. Equality is then considered and added to the concepts of this chapter in Chapters **??**, **??**.

Definition 3.8.1 (Herbrand Interpretation)**.** A *Herbrand Interpretation* (over Σ) is a Σ-algebra $\mathcal{A}$ so that

1. $S^{\mathcal{A}} = T_S(\Sigma)$ for every sort $S \in \mathcal{S}$

2. $f^{\mathcal{A}} : (s_1, \ldots, s_n) \mapsto f(s_1, \ldots, s_n)$ where $f \in \Omega$, $\text{arity}(f) = n$, $s_i \in T_{S_i}(\Sigma)$ and $f : S_1 \times \ldots \times S_n \to S$ is the sort declaration for f

3. $P^{\mathcal{A}} \subseteq (T_{S_1}(\Sigma) \times \ldots \times T_{S_m}(\Sigma))$ where $P \in \Pi$, $\text{arity}(P) = m$ and $P \subseteq S_1 \times \ldots \times S_m$ is the sort declaration for P

In other words, values are fixed to be ground terms and functions are fixed to be the term constructors. Only predicate symbols may be freely interpreted as relations over ground terms.

Proposition 3.8.2. Every set of ground atoms I uniquely determines a Herbrand interpretation $\mathcal{A}$ via

$$(s_1, \ldots, s_n) \in P_{\mathcal{A}} \text{ iff } P(s_1, \ldots, s_n) \in I$$

Thus Herbrand interpretations (over Σ) can be identified with sets of Σ-ground atoms. A Herbrand interpretation I is called a *Herbrand model* of ϕ, if $I \models \phi$.

Example 3.8.3. Consider the signature $\Sigma = (\{S\}, \{a, b\}, \{P, Q\})$, where a, b are constants, $\text{arity}(P) = 1$, $\text{arity}(Q) = 2$, and all constants, predicates are defined over the sort S. Then the following are examples of Herbrand interpretations over Σ, where for all interpretations $S_{\mathcal{A}} = \{a, b\}$.

$I_1 := \emptyset$
$I_2 := \{P(a), Q(a, a), Q(b, b)\}$
$I_3 := \{P(a), P(b), Q(a, a), Q(b, b), Q(a, b), Q(b, a)\}$

Now consider the extension Σ' of Σ by one unary function symbol $g : S \to S$. Then the following are examples of Herbrand interpretations over Σ', where for all interpretations $S_{\mathcal{A}} = \{a, b, g(a), g(b), g(g(a)), \ldots\}$.

$I'_1 := \emptyset$
$I'_2 := \{P(a), Q(a, g(a)), Q(b, b)\}$
$I'_3 := \{P(a), P(g(a)), P(g(g(a))), \ldots, Q(a, a), Q(b, b), Q(b, g(b)), Q(b, g(g(b))), \ldots\}$

Theorem 3.8.4 (Herbrand)**.** Let N be a set of Σ-clauses. Then N is satisfiable iff N has a Herbrand model over Σ iff $\text{ground}(\Sigma, N)$ has a Herbrand model over Σ, where $\text{ground}(\Sigma, N) = \{C\sigma \mid C \in N, \text{dom}(\sigma) = \text{vars}(C), \text{ and } x\sigma \in T_{\text{sort}(x)}(\Sigma) \text{ for all } x \in \text{dom}(\sigma)\}$ is the set of *ground instances* of N.

Example 3.8.5 (Example of a $\text{ground}(\Sigma, N)$)**.** Consider Σ' from Example 3.8.3 and the clause set $N = \{Q(x, x) \vee \neg P(x), \neg P(x) \vee P(g(x))\}$. Then the set of ground instances $\text{ground}(\Sigma', N) = \{$

$Q(a, a) \vee \neg P(a)$
$Q(b, b) \vee \neg P(b)$
$Q(g(a), g(a)) \vee \neg P(g(a))$
$\ldots$
$\neg P(a) \vee P(g(a))$
$\neg P(b) \vee P(g(b))$
$\neg P(g(a)) \vee P(g(g(a)))$
$\ldots\}$

is satisfiable. For example by the Herbrand models
$I_1 := \emptyset$
$I_2 := \{P(b), Q(b,b), P(g(b)), Q(g(b), g(b)), \ldots\}$

3.9 Orderings

Definition 3.9.1 (Σ-Operation Compatible Relation)**.** A binary relation $\sqsupset$ over $T(\Sigma, \mathcal{X})$ is called *compatible with* Σ*-operations,* if $s \sqsupset s'$ implies $f(t_1, \ldots, s, \ldots, t_n) \sqsupset f(t_1, \ldots, s', \ldots, t_n)$ for all $f \in \Omega$ and $s, s', t_i \in T(\Sigma, \mathcal{X})$.

Lemma 3.9.2. A relation $\sqsupset$ is compatible with Σ-operations iff $s \sqsupset s'$ implies $t[s]_p \sqsupset t[s']_p$ for all $s, s', t \in T(\Sigma, \mathcal{X})$ and $p \in pos(t)$.

In the literature *compatible with* Σ*-operations* is sometimes also called *compatible with contexts.*

Definition 3.9.3 (Substitution Stable Relation, Rewrite Relation)**.** A binary relation $\sqsupset$ over $T(\Sigma, \mathcal{X})$ is called *stable under substitutions,* if $s \sqsupset s'$ implies $s\sigma \sqsupset s'\sigma$ for all $s, s' \in T(\Sigma, \mathcal{X})$ and substitutions σ. A binary relation $\sqsupset$ is called a *rewrite relation,* if it is compatible with Σ-operations and stable under substitutions.

Definition 3.9.4 (Lexicographical Path Ordering (LPO))**.** Let $\Sigma = (\mathcal{S}, \Omega, \Pi)$ be a signature and let $\succ$ be a strict partial ordering on operator symbols in Ω, called *precedence.* The *lexicographical path ordering* $\succ_{lpo}$ on $T(\Sigma, \mathcal{X})$ is defined as follows: if s, t are terms in $T_S(\Sigma, \mathcal{X})$ then $s \succ_{lpo} t$ iff

1. $t = x \in \mathcal{X}$, $x \in vars(s)$ and $s \neq t$ or
2. $s = f(s_1, \ldots, s_n)$, $t = g(t_1, \ldots, t_m)$ and
   (a) $s_i \succeq_{lpo} t$ for some $i \in \{1, \ldots, n\}$ or
   (b) $f \succ g$ and $s \succ_{lpo} t_j$ for every $j \in \{1, \ldots, m\}$ or
   (c) $f = g$, $s \succ_{lpo} t_j$ for every $j \in \{1, \ldots, m\}$ and $(s_1, \ldots, s_n)(\succ_{lpo})_{lex}(t_1, \ldots, t_m)$.

Theorem 3.9.5. 1. The LPO is a rewrite ordering.

2. If the precedence $\succ$ is total on Ω then $\succ_{lpo}$ is total on the set of ground terms $T(\Sigma)$.
3. If Ω is finite then $\succ_{lpo}$ is well-founded.

Example 3.9.6. Consider the terms $g(x)$, $g(y)$, $g(g(a))$, $g(b)$, $g(a)$, b, a. With respect to the precedence $g \succ b \succ a$ the ordering on the ground terms is $g(g(a)) \succ_{lpo} g(b) \succ_{lpo} g(a) \succ_{lpo} b \succ_{lpo} a$. The terms $g(x)$ and $g(y)$ are not comparable. Note that the terms $g(g(a))$, $g(b)$, $g(a)$ are all instances of both $g(x)$ and $g(y)$.

With respect to the precedence $b \succ a \succ g$ the ordering on the ground terms is $g(b) \succ_{lpo} b \succ_{lpo} g(g(a)) \succ_{lpo} g(a) \succ_{lpo} a$.

Definition 3.9.7 (The Knuth-Bendix Ordering)**.** Let $\Sigma = (\mathcal{S}, \Omega, \Pi)$ be a finite signature, let $\succ$ be a strict partial ordering *("precedence")* on Ω, let $w : \Omega \cup \mathcal{X} \to \mathbb{R}_0^+$ be a *weight function*, so that the following admissibility conditions are satisfied:

1. $w(x) = w_0 \in \mathbb{R}^+$ for all variables $x \in \mathcal{X}$; $w(c) \geq w_0$ for all constants $c \in \Omega$.

2. If $w(f) = 0$ for some $f \in \Omega$ with $\text{arity}(f) = 1$, then $f \succeq g$ for all $g \in \Omega$.

Then, the weight function w can be extended to terms recursively:

$$w(f(t_1, \ldots, t_n)) = w(f) + \sum_{1 \leq i \leq n} w(t_i)$$

or alternatively

$$\sum w(t) = \sum_{x \in vars(t)} w(x) \cdot \#(x, t) + \sum_{f \in \Omega} w(f) \cdot \#(f, t)$$

where $\#(a, t)$ is the number of occurrences of a in t.

The *Knuth-Bendix ordering* $\succ_{kbo}$ on $T(\Sigma, \mathcal{X})$ induced by $\succ$ and admissible w is defined by: $s \succ_{kbo} t$ iff

1. $\#(x, s) \geq \#(x, t)$ for all variables x and $w(s) > w(t)$, or

2. $\#(x, s) \geq \#(x, t)$ for all variables x, $w(s) = w(t)$, and

   (a) $t = x$, $s = f^n(x)$ for some $n \geq 1$, or

   (b) $s = f(s_1, \ldots, s_m)$, $t = g(t_1, \ldots, t_n)$, and $f \succ g$, or

   (c) $s = f(s_1, \ldots, s_m)$, $t = f(t_1, \ldots, t_m)$, and $(s_1, \ldots, s_m)(\succ_{kbo})_{lex}(t_1, \ldots, t_m)$.

Theorem 3.9.8.

1. The KBO is a rewrite ordering.

2. If the precedence $\succ$ is total on Ω then $\succ_{kbo}$ is total on the set of ground terms $T(\Sigma)$.

3. If Ω is finite then $\succ_{kbo}$ is well-founded.

The LPO ordering as well as the KBO ordering can be extended to atoms in a straightforward way. The precedence $\succ$ is extended to Π. For LPO atoms are then compared according to Definition 3.9.4-2. For KBO the weight function w is also extended to atoms by giving predicates a non-zero positive weight and then atoms are compared according to terms.

Actually, since atoms are never substituted for variables in first-order logic, an alternative to the above would be to first compare the predicate symbols and let $\succ$ decide the ordering. Only if the atoms share the same predicate symbol, the argument terms are considered, e.g., in a lexicographic way and are then compared with respect to KBO or LPO, respectively.

3.10 Ground Superposition

Propositional clauses and ground clauses are essentially the same, as long as equational atoms are not considered. This section deals only with ground clauses and recalls mostly the material from Section 2.7 for first-order ground clauses. Let N be a set of ground clauses.

Definition 3.10.1 (Clause Ordering)**.** Let $\prec$ be a total strict rewrite ordering on terms and atoms. Then $\prec$ can be lifted to a total ordering $\prec_L$ on literals by its multiset extension $\prec_{\text{mul}}$ where a positive literal $P(t_1,\ldots,t_n)$ is mapped to the multiset $\{P(t_1,\ldots,t_n)\}$ and a negative literal $\neg P(t_1,\ldots,t_n)$ to the multiset $\{P(t_1,\ldots,t_n), P(t_1,\ldots,t_n)\}$. The ordering $\prec_L$ is further lifted to a total ordering on clauses $\prec_C$ by considering the multiset extension of $\prec_L$ for clauses.

Proposition 3.10.2 (Properties of the Clause Ordering)**.** (i) The orderings on literals and clauses are total and well-founded.
(ii) Let C and D be clauses with $P(t_1,\ldots,t_n) = |\max(C)|$, $Q(s_1,\ldots,s_m) = |\max(D)|$, where $\max(C)$ denotes the maximal literal in C.

1. If $Q(s_1,\ldots,s_m) \prec_L P(t_1,\ldots,t_n)$ then $D \prec_C C$.
2. If $P(t_1,\ldots,t_n) = Q(s_1,\ldots,s_m)$, $P(t_1,\ldots,t_n)$ occurs negatively in C but only positively in D, then $D \prec_C C$.

Eventually, as I did for propositional logic, I overload $\prec$ with $\prec_L$ and $\prec_C$. So if $\prec$ is applied to literals it denotes $\prec_L$, if it is applied to clauses, it denotes $\prec_C$. Note that $\prec$ is a total ordering on literals and clauses as well. For superposition, inferences are restricted to maximal literals with respect to $\prec$. For a clause set N, I define $N^{\prec C} = \{D \in N \mid D \prec C\}$.

Definition 3.10.3 (Abstract Redundancy)**.** A ground clause C is *redundant* with respect to a ground clause set N if $N^{\prec C} \models C$.

Tautologies are redundant. Subsumed clauses are redundant if $\subseteq$ is strict. Duplicate clauses are anyway eliminated quietly because the calculus operates on sets of clauses.

C

Note that for finite N, and any $C \in N$ redundancy $N^{\prec C} \models C$ can be decided but is as hard as testing unsatisfiability for a clause set N. So the goal is to invent redundancy notions that can be efficiently decided and that are useful.

Definition 3.10.4 (Selection Function)**.** The selection function sel maps clauses to one of its negative literals or $\bot$. If $\text{sel}(C) = \neg P(t_1,\ldots,t_n)$ then $\neg P(t_1,\ldots,t_n)$ is called *selected* in C. If $\text{sel}(C) = \bot$ then no literal in C is *selected*.

The selection function is, in addition to the ordering, a further means to restrict superposition inferences. If a negative literal is selected on a clause, any superposition inference must be on the selected literal.

Definition 3.10.5 (Partial Model Construction)**.** Given a clause set N and an ordering $\prec$ we can construct a (partial) model $N_{\mathcal{I}}$ for N inductively as follows:

$$N_C := \bigcup_{D \prec C} \delta_D$$

$$\delta_D := \begin{cases} \{P(t_1,\dots,t_n)\} & \text{if } D = D' \lor P(t_1,\dots,t_n), P(t_1,\dots,t_n) \text{ strictly maximal, no literal} \\ & \text{selected in } D \text{ and } N_D \not\models D \\ \emptyset \quad \text{otherwise} & \end{cases}$$

$$N_{\mathcal{I}} := \bigcup_{C \in N} \delta_C$$

Clauses C with $\delta_C \neq \emptyset$ are called *productive*.

Proposition 3.10.6. Some properties of the partial model construction.

1. For every D with $(C \lor \neg P(t_1,\dots,t_n)) \prec D$ we have $\delta_D \neq \{P(t_1,\dots,t_n)\}$.
2. If $\delta_C = \{P(t_1,\dots,t_n)\}$ then $N_C \cup \delta_C \models C$.
3. If $N_C \models D$ and $D \prec C$ then for all C' with $C \prec C'$ we have $N_{C'} \models D$ and in particular $N_{\mathcal{I}} \models D$.
4. There is no clause C with $P(t_1,\dots,t_n) \lor P(t_1,\dots,t_n) \prec C$ such that $\delta_C = \{P\}$.

Please properly distinguish: N is a set of clauses interpreted as the conjunction of all clauses. $N^{\prec C}$ is of set of clauses from N strictly smaller than C with respect to $\prec$. $N_{\mathcal{I}}$, N_C are Herbrand interpretations (see Proposition 3.8.2). $N_{\mathcal{I}}$ is the overall (partial) model for N, whereas N_C is generated from all clauses from N strictly smaller than C.

T

Superposition Left $(N \uplus \{C_1 \lor P(t_1,\dots,t_n), C_2 \lor \neg P(t_1,\dots,t_n)\}) \Rightarrow_{\text{SUP}} (N \cup \{C_1 \lor P(t_1,\dots,t_n), C_2 \lor \neg P(t_1,\dots,t_n)\} \cup \{C_1 \lor C_2\})$
where (i) $P(t_1,\dots,t_n)$ is strictly maximal in $C_1 \lor P(t_1,\dots,t_n)$ (ii) no literal in $C_1 \lor P(t_1,\dots,t_n)$ is selected (iii) $\neg P(t_1,\dots,t_n)$ is maximal and no literal selected in $C_2 \lor \neg P(t_1,\dots,t_n)$, or $\neg P(t_1,\dots,t_n)$ is selected in $C_2 \lor \neg P(t_1,\dots,t_n)$

Factoring $(N \uplus \{C \lor P(t_1,\dots,t_n) \lor P(t_1,\dots,t_n)\}) \Rightarrow_{\text{SUP}} (N \cup \{C \lor P(t_1,\dots,t_n) \lor P(t_1,\dots,t_n)\} \cup \{C \lor P(t_1,\dots,t_n)\})$
where (i) $P(t_1,\dots,t_n)$ is maximal in $C \lor P(t_1,\dots,t_n) \lor P(t_1,\dots,t_n)$ (ii) no literal is selected in $C \lor P(t_1,\dots,t_n) \lor P(t_1,\dots,t_n)$

Note that the superposition factoring rule differs from the resolution factoring rule in that it only applies to positive literals.

Definition 3.10.7 (Saturation)**.** A set N of clauses is called *saturated up to redundancy*, if any inference from non-redundant clauses in N yields a redundant clause with respect to N.

Examples for specific redundancy rules that can be efficiently decided are

Subsumption $(N \uplus \{C_1, C_2\}) \Rightarrow_{\text{SUP}} (N \cup \{C_1\})$
provided $C_1 \subset C_2$

Tautology Deletion $(N \uplus \{C \lor P(t_1,\ldots,t_n) \lor \neg P(t_1,\ldots,t_n)\}) \Rightarrow_{\text{SUP}} (N)$

Condensation $(N \uplus \{C_1 \lor L \lor L\}) \Rightarrow_{\text{SUP}} (N \cup \{C_1 \lor L\})$

Subsumption Resolution $(N \uplus \{C_1 \lor L, C_2 \lor \neg L\}) \Rightarrow_{\text{SUP}} (N \cup \{C_1 \lor L, C_2\})$
where $C_1 \subseteq C_2$

Proposition 3.10.8. All clauses removed by Subsumption, Tautology Deletion, Condensation and Subsumption Resolution are redundant with respect to the kept or added clauses.

Theorem 3.10.9. Let N be a, possibly countably infinite, set of ground clauses. If N is saturated up to redundancy and $\bot \notin N$ then N is satisfiable and $N_{\mathcal{I}} \models N$.

Proof. The proof is by contradiction. So I assume: (i) for any clause D derived by Superposition Left or Factoring from N that D is redundant, i.e., $N^{\prec D} \models D$, (ii) $\bot \notin N$ and (iii) $N_{\mathcal{I}} \not\models N$. Then there is a minimal, with respect to $\prec$, clause $C \lor L \in N$ such that $N_{\mathcal{I}} \not\models C \lor L$ and L is a selected literal in $C \lor L$ or no literal in $C \lor L$ is selected and L is maximal. This clause must exist because $\bot \notin N$.

The clause $C \lor L$ is not redundant. For otherwise, $N^{\prec C\lor L} \models C \lor L$ and hence $N_{\mathcal{I}} \models C \lor L$, because $N_{\mathcal{I}} \models N^{\prec C\lor L}$, a contradiction.

I distinguish the case L is a positive and no literal selected in $C \lor L$ or L is a negative literal. Firstly, assume L is positive, i.e., $L = P(t_1,\ldots,t_n)$ for some ground atom $P(t_1,\ldots,t_n)$. Now if $P(t_1,\ldots,t_n)$ is strictly maximal in $C \lor P(t_1,\ldots,t_n)$ then actually $\delta_{C\lor P} = \{P(t_1,\ldots,t_n)\}$ and hence $N_{\mathcal{I}} \models C \lor P$, a contradiction. So $P(t_1,\ldots,t_n)$ is not strictly maximal. But then actually $C \lor P(t_1,\ldots,t_n)$ has the form $C_1' \lor P(t_1,\ldots,t_n) \lor P(t_1,\ldots,t_n)$ and Factoring derives $C_1' \lor P(t_1,\ldots,t_n)$ where $(C_1' \lor P(t_1,\ldots,t_n)) \prec (C_1' \lor P(t_1,\ldots,t_n) \lor P(t_1,\ldots,t_n))$. Now $C_1' \lor P(t_1,\ldots,t_n)$ is not redundant, strictly smaller than $C \lor L$, we have $C_1' \lor P(t_1,\ldots,t_n) \in N$ and $N_{\mathcal{I}} \not\models C_1' \lor P(t_1,\ldots,t_n)$, a contradiction against the choice that $C \lor L$ is minimal.

Secondly, let us assume L is negative, i.e., $L = \neg P(t_1,\ldots,t_n)$ for some ground atom $P(t_1,\ldots,t_n)$. Then, since $N_{\mathcal{I}} \not\models C \lor \neg P(t_1,\ldots,t_n)$ we know $P(t_1,\ldots,t_n) \in N_{\mathcal{I}}$. So there is a clause $D \lor P(t_1,\ldots,t_n) \in N$ where $\delta_{D\lor P(t_1,\ldots,t_n)} = \{P(t_1,\ldots,t_n)\}$ and $P(t_1,\ldots,t_n)$ is strictly maximal in $D \lor P(t_1,\ldots,t_n)$ and $(D \lor P(t_1,\ldots,t_n)) \prec (C \lor \neg P(t_1,\ldots,t_n))$. So Superposition Left derives $C \lor D$ where $(C \lor D) \prec (C \lor \neg P(t_1,\ldots,t_n))$. The derived clause $C \lor D$ cannot be redundant, because for otherwise either $N^{\prec D\lor P(t_1,\ldots,t_n)} \models$

$D \vee P(t_1, \ldots, t_n)$ or $N^{\prec C \vee \neg P(t_1,\ldots,t_n)} \models C \vee \neg P(t_1, \ldots, t_n)$. So $C \vee D \in N$ and $N_{\mathcal{I}} \not\models C \vee D$, a contradiction against the choice that $C \vee L$ is the minimal false clause. □

So the proof actually tells us that at any point in time we need only to consider either a superposition left inference between a minimal false clause and a productive clause or a factoring inference on a minimal false clause.

Theorem 3.10.10 (Compactness of First-Order Logic)**.** Let N be a, possibly infinite, set of first-order logic ground clauses. Then N is unsatisfiable iff there is a finite subset $N' \subseteq N$ such that N' is unsatisfiable.

Proof. If N is unsatisfiable, saturation via superposition generates $\bot$. So there is an i such that $N \Rightarrow^i_{\text{SUP}} N'$ and $\bot \in N'$. The clause $\bot$ is the result of at most i many superposition inferences, reductions on clauses $\{C_1, \ldots, C_n\} \subseteq N$. Superposition is sound, so $\{C_1, \ldots, C_n\}$ is a finite, unsatisfiable subset of N. □

Corollary 3.10.11 (Compactness of First-Order Logic: Classical)**.** A set N of clauses is satisfiable iff all finite subsets of N are satisfiable

Theorem 3.10.12 (Soundness and Completeness of Ground Superposition)**.** A first-order Σ-sentence ϕ is valid iff there exists a ground superposition refutation for $\text{ground}(\Sigma, \text{cnf}(\neg\phi))$.

Proof. A first-order sentence ϕ is valid iff $\neg\phi$ is unsatisfiable iff $\text{cnf}(\neg\phi)$ is unsatisfiable iff $\text{ground}(\Sigma, \text{cnf}(\neg\phi))$ is unsatisfiable iff superposition provides a refutation of $\text{ground}(\Sigma, \text{cnf}(\neg\phi))$. □

Theorem 3.10.13 (Semi-Decidability of First-Order Logic by Ground Superposition)**.** If a first-order Σ-sentence ϕ is valid then a ground superposition refutation can be computed.

Proof. In a fair way enumerate $\text{ground}(\Sigma, \text{cnf}(\neg\phi))$ and perform superposition inference steps. The enumeration can, e.g., be done by considering Herbrand terms of increasing size. □

Example 3.10.14 (Ground Superposition)**.** Consider the below clauses 1-4 and superposition refutation with respect a KBO with precedence $P \succ Q \succ g \succ f \succ c \succ b \succ a$ where the weight function w returns 1 for all signature symbols. Maximal literals are marked with a *.

1.	$\neg P(f(c))^* \vee \neg P(f(c))^* \vee Q(b)$	(Input)
2.	$P(f(c))^* \vee Q(b)$	(Input)
3.	$\neg P(g(b,c))^* \vee \neg Q(b)$	(Input)
4.	$P(g(b,c))^*$	(Input)
5.	$\neg P(f(c))^* \vee Q(b)$	(Cond(1))
6.	$Q(b)^* \vee Q(b)^*$	(Sup(5, 2)))
7.	$Q(b)^*$	(Fact(6))
8.	$\neg Q(b)^*$	(Sup(3, 4))
10.	$\bot$	(Sup(8, 7))

Note that clause 5 cannot be derived by Factoring whereas clause 7 can also be derived by Condensation. Clause 8 is also the result of a Subsumption Resolution application to clauses 3, 4.

Theorem 3.10.15 (Craig Theorem [14])**.** Let ϕ and ψ be two propositional formulas so that $\phi \models \psi$. Then there exists a formula χ (called the *interpolant* for $\phi \models \psi$), so that χ contains only propositional variables occurring both in ϕ and in ψ so that $\phi \models \chi$ and $\chi \models \psi$.

Proof. Translate ϕ and $\neg\psi$ into CNF. let N and M, respectively, denote the resulting clause set. Choose an atom ordering $\succ$ for which the propositional variables that occur in ϕ but not in ψ are maximal. Saturate N into N^* w.r.t. $Sup^{\succ}_{sel}$ with an empty selection function sel. Then saturate $N^* \cup M$ w.r.t. $Sup^{\succ}_{sel}$ to derive $\bot$. As N^* is already saturated, due to the ordering restrictions only inferences need to be considered where premises, if they are from N^*, only contain symbols that also occur in ψ. The conjunction of these premises is an interpolant χ. The theorem also holds for first-order formulas. For universal formulas the above proof can be easily extended. In the general case, a proof based on superposition technology is more complicated because of Skolemization. □

3.11 First-Order Superposition with Selection

The completeness proof of ground superposition (Section 3.10) talks about (strictly) maximal literals of *ground* clauses. The non-ground calculus considers those literals that correspond to (strictly) maximal literals of ground instances.

The used ordering is exactly the ordering of Definition 3.10.1 where clauses with variables are projected to their ground instances for ordering computations.

Definition 3.11.1 (Maximal Literal)**.** A literal L is called *[strictly] maximal* in a clause C if and only if there exists a grounding substitution σ so that $L\sigma$ is [strictly] maximal in $C\sigma$ (i.e., if for no other L' in C: $L\sigma \prec L'\sigma$ [$L\sigma \preceq L'\sigma$]).

Superposition Left $(N \uplus \{C_1 \vee P(t_1,\ldots,t_n), C_2 \vee \neg P(s_1,\ldots,s_n)\}) \Rightarrow_{\text{SUP}}$ $(N \cup \{C_1 \vee P(t_1,\ldots,t_n), C_2 \vee \neg P(s_1,\ldots,s_n)\} \cup \{(C_1 \vee C_2)\sigma\})$

where (i) $P(t_1,\ldots,t_n)\sigma$ is strictly maximal in $(C_1 \vee P(t_1,\ldots,t_n))\sigma$ (ii) no literal in $C_1 \vee P(t_1,\ldots,t_n)$ is selected (iii) $\neg P(s_1,\ldots,s_n)\sigma$ is maximal and no literal selected in $(C_2 \vee \neg P(s_1,\ldots,s_n))\sigma$, or $\neg P(s_1,\ldots,s_n)$ is selected in $(C_2 \vee \neg P(s_1,\ldots,s_n))\sigma$ (iv) σ is the mgu of $P(t_1,\ldots,t_n)$ and $P(s_1,\ldots,s_n)$

Factoring $(N \uplus \{C \vee P(t_1,\ldots,t_n) \vee P(s_1,\ldots,s_n)\}) \Rightarrow_{\text{SUP}}$ $(N \cup \{C \vee P(t_1,\ldots,t_n) \vee P(s_1,\ldots,s_n)\} \cup \{(C \vee P(t_1,\ldots,t_n))\sigma\})$

where (i) $P(t_1,\ldots,t_n)\sigma$ is maximal in $(C \vee P(t_1,\ldots,t_n) \vee P(s_1,\ldots,s_n))\sigma$ (ii) no literal is selected in $C \vee P(t_1,\ldots,t_n) \vee P(s_1,\ldots,s_n)$ (iii) σ is the mgu of $P(t_1,\ldots,t_n)$ and $P(s_1,\ldots,s_n)$

Note that the above inference rules Superpositions Left and Factoring are generalizations of their respective counterparts from Section 3.10. On ground clauses they coincide. Therefore, we can safely overload them in the sequel.

Definition 3.11.2 (Abstract Redundancy)**.** A clause C is *redundant* with respect to a clause set N if for all ground instances $C\sigma$ where are clauses $\{C_1, \ldots, C_n\} \subseteq N$ with ground instances $C_1\tau_1, \ldots, C_n\tau_n$ such that $C_i\tau_i \prec C\sigma$ for all i and $C_1\tau_1, \ldots, C_n\tau_n \models C\sigma$.

Definition 3.11.3 (Saturation)**.** A set N of clauses is called *saturated up to redundancy*, if any inference from non-redundant clauses in N yields a redundant clause with respect to N.

In contrast to the ground case, the above abstract notion of redundancy is not effective, i.e., it is undecidable for some clause C whether it is redundant, in general. Nevertheless, the concrete redundancy notions from Section 3.10 carry over to the non-ground case. Let dup be a function from clauses to clauses that removes duplicate literals, i.e., $\text{dup}(C) = C'$ where $C' \subseteq C$, C' does not contain any duplicate literals, and for each $L \in C$ also $L \in C'$.

Subsumption $(N \uplus \{C_1, C_2\}) \Rightarrow_{\text{SUP}} (N \cup \{C_1\})$
provided $C_1\sigma \subset C_2$ for some σ

Tautology Deletion $(N \uplus \{C \vee P(t_1, \ldots, t_n) \vee \neg P(t_1, \ldots, t_n)\}) \Rightarrow_{\text{SUP}} (N)$

Condensation $(N \uplus \{C_1 \vee L \vee L'\}) \Rightarrow_{\text{SUP}} (N \cup \{\text{dup}((C_1 \vee L \vee L')\sigma)\})$
provided $L\sigma = L'$ and $\text{dup}((C_1 \vee L \vee L')\sigma)$ subsumes $C_1 \vee L \vee L'$ for some σ

Subsumption Resolution $(N \uplus \{C_1 \vee L, C_2 \vee L'\}) \Rightarrow_{\text{SUP}} (N \cup \{C_1 \vee L, C_2\})$
where $L\sigma = \neg L'$ and $C_1\sigma \subseteq C_2$ for some σ

Lemma 3.11.4. All reduction rules are instances of the abstract redundancy criterion

Lemma 3.11.5 (Subsumption is NP-complete)**.** Subsumption is NP-complete.

Proof. Let C_1 subsume C_2 with substitution σ Subsumption is in NP because the size of σ is bound by the size of C_2 and the subset relation can be checked in time at most quadratic in the size of C_1 and C_2.

Propositional SAT can be reduced as follows. Assume a 3-SAT clause set N. Consider a 3-place predicate R and a unary function g and a mapping from propositional variables P to first order variables x_P. ... □

Lemma 3.11.6 (Lifting)**.** Let $D \vee L$ and $C \vee L'$ be variable-disjoint clauses and σ a grounding substitution for $C \vee L$ and $D \vee L'$. If there is a superposition left inference

$$(N \uplus \{(D \vee L)\sigma, (C \vee L')\sigma\}) \Rightarrow_{\text{SUP}} (N \cup \{(D \vee L)\sigma, (C \vee L')\sigma\} \cup \{D\sigma \vee C\sigma\})$$

and if $\text{sel}((D \lor L)\sigma) = \text{sel}((D \lor L))\sigma$, $\text{sel}((C \lor L')\sigma) = \text{sel}((C \lor L'))\sigma$, then there exists a mgu τ such that

$$(N \uplus \{D \lor L, C \lor L'\}) \Rightarrow_{\text{SUP}} (N \cup \{D \lor L, C \lor L'\} \cup \{(D \lor C)\tau\}).$$

Let $C \lor L \lor L'$ be a clause and σ a grounding substitution for $C \lor L \lor L'$. If there is a factoring inference

$$(N \uplus \{(C \lor L \lor L')\sigma\}) \Rightarrow_{\text{SUP}} (N \cup \{(C \lor L \lor L')\sigma\} \cup \{(C \lor L)\sigma\})$$

and if $\text{sel}((C \lor L \lor L')\sigma) = \text{sel}((C \lor L \lor L'))\sigma$, then there exists a mgu τ such that

$$(N \uplus \{C \lor L \lor L'\}) \Rightarrow_{\text{SUP}} (N \cup \{C \lor L \lor L'\} \cup \{(C \lor L)\tau\})$$

Note that in the above lemma the clause $D\sigma \lor C\sigma$ is an instance of the clause $(D \lor C)\tau$ The reduction rules cannot be lifted in the same way as the following example shows.

Example 3.11.7 (First-Order Reductions are not Liftable)**.** Consider the two clauses $P(x) \lor Q(x)$, $P(g(y))$ and grounding substitution $\{x \mapsto g(a), y \mapsto a\}$. Then $P(g(y))\sigma$ subsumes $(P(x) \lor Q(x))\sigma$ but $P(g(y))$ does not subsume $P(x) \lor Q(x)$. For all other reduction rules similar examples can be constructed.

Lemma 3.11.8 (Soundness and Completeness)**.** Superposition is sound and complete.

Proof. Soundness is obvious. For completeness, Theorem 3.10.12 proves the ground case. Now by applying Lemma 3.11.6 to this proof it can be lifted to the first-order level. □

There are questions left open by Lemma 3.11.8. It just says that a ground refutation can be lifted to a first-order refutation. But what about abstract redundancy, Definition 3.11.2? Can first-order redundant clauses be deleted without harming completeness? And what about the ground model operator with respect to clause sets N saturated on the first order level. Is in this case $\text{ground}(\Sigma, N)_{\mathcal{I}}$ a model? The next two lemmas answer these questions positively.

Lemma 3.11.9 (Redundant Clauses are Obsolete)**.** If a clause set N is unsatisfiable, then there is a derivation $N \Rightarrow^*_{\text{SUP}} N'$ such that $\bot \in N'$ and no clause in the derivation of $\bot$ is redundant.

Proof. If N is unsatisfiable then there is a ground superposition refutation of $\text{ground}(\Sigma, N)$ such that no ground clause in the refutation is redundant. Now according to Lemma 3.11.8 this proof can be lifted to the first-order level. Now assume some clause C in the first-order proof is redundant that is the lifting of some clause $C\sigma$ from the ground proof with respect to a grounding substitution σ. The clause C is redundant by Definition 3.11.2 if all its ground instances are, in particular, $C\sigma$. But this contradicts the fact that the lifted ground proof does not contain redundant clauses. □

Lemma 3.11.10 (Model Property)**.** If N is a saturated clause set and $\bot \notin N$ then $\text{ground}(\Sigma, N)_{\mathcal{I}} \models N$.

Proof. As usual we assume that selection on the ground and respective non-ground clauses is identical. Assume $\text{ground}(\Sigma, N)_{\mathcal{I}} \not\models N$. Then there is a minimal ground clause $C\sigma$, $C \neq \bot$, $C \in N$ such that $\text{ground}(\Sigma, N)_{\mathcal{I}} \not\models C\sigma$. Note that $C\sigma$ is not redundant as for otherwise $\text{ground}(\Sigma, N)_{\mathcal{I}} \models C\sigma$. So $\text{ground}(\Sigma, N)$ is not saturated. If $C\sigma$ is productive, i.e., $C\sigma = (C' \lor L)\sigma$ such that L is positive, $L\sigma$ strictly maximal in $(C' \lor L)\sigma$ then $L\sigma \in \text{ground}(\Sigma, N)_{\mathcal{I}}$ and hence $\text{ground}(\Sigma, N)_{\mathcal{I}} \models C\sigma$ contradicting $\text{ground}(\Sigma, N)_{\mathcal{I}} \not\models C\sigma$.

If $C\sigma = (C' \lor L \lor L')\sigma$ such that L is positive, $L\sigma$ maximal in $(C' \lor L \lor L')\sigma$ then, because N is saturated, there is a clause $(C' \lor L)\tau \in N$ such that $(C' \lor L)\tau\sigma = (C' \lor L)\sigma$. Now $(C' \lor L)\tau$ is not redundant, $\text{ground}(\Sigma, N)_{\mathcal{I}} \not\models (C' \lor L)\tau$, contradicting the minimal choice of $C\sigma$.

If $C\sigma = (C' \lor L)\sigma$ such that L is selected, or negative and maximal then there is a clause $(D' \lor L') \in N$ and grounding substitution ρ, such that $L'\rho$ is a strictly maximal positive literal in $(D' \lor L')\rho$, $L'\rho \in \text{ground}(\Sigma, N)_{\mathcal{I}}$ and $L'\rho = \neg L\sigma$. Again, since N is saturated, there is variable disjoint clause $(C' \lor D')\tau \in N$ for some unifier τ, $(C' \lor D')\tau\sigma\rho \prec C\sigma$, and $\text{ground}(\Sigma, N)_{\mathcal{I}} \not\models (C' \lor D')\tau\sigma\rho$ contradicting the minimal choice of $C\sigma$. ☐

Definition 3.11.11 (Persistent Clause)**.** Let $N_0 \Rightarrow_{\text{SUP}} N_1 \Rightarrow_{\text{SUP}} \ldots$ be a, possibly infinite, superposition derivation. A clause C is called *persistent* in this derivation if $C \in N_i$ for some i and for all $j > i$ also $C \in N_j$.

Definition 3.11.12 (Fair Derivation)**.** A derivation $N_0 \Rightarrow_{\text{SUP}} N_1 \Rightarrow_{\text{SUP}} \ldots$ is called *fair* if for any persistent clause $C \in N_i$ where factoring is applicable to C, there is a j such that the factor of $C' \in N_j$ or $\bot \in N_j$. If $\{C, D\} \subseteq N_i$ are persistent clauses such that superposition left is applicable to C, D then the superposition left result is also in N_j for some j or $\bot \in N_j$.

Theorem 3.11.13 (Dynamic Superposition Completeness)**.** If N is unsatisfiable and $N = N_0 \Rightarrow_{\text{SUP}} N_1 \Rightarrow_{\text{SUP}} \ldots$ is a fair derivation, then there is $\bot \in N_j$ for some j.

Proof. If N is unsatisfiable, then by Lemma 3.11.8 there is a derivation of $\bot$ by superposition. Furthermore, no clause contributing to the derivation of $\bot$ is redundant (Lemma 3.11.9). So all clauses in the derivation of $\bot$ are persistent. The derivation $N_0 \Rightarrow_{\text{SUP}} N_1 \Rightarrow_{\text{SUP}} \ldots$ is fair, hence $\bot \in N_j$ for some j. ☐

Lemma 3.11.14. Let $\text{red}(N)$ be all clauses that are redundant with respect to the clauses in N and N, M be clause sets. Then

1. if $N \subseteq M$ then $\text{red}(N) \subseteq \text{red}(M)$
2. if $M \subseteq \text{red}(N)$ then $\text{red}(N) \subseteq \text{red}(N \setminus M)$

It follows that redundancy is preserved when, during a theorem proving process, new clauses are added (or derived) or redundant clauses are deleted. Furthermore, $\text{red}(N)$ may include clauses that are not in N.

Algorithm 7: SupProver(N)

```
   Input   : A set of clauses N.
   Output: A saturated set of clauses N', equivalent to N.
 1 WO := ∅;
 2 US := N;
 3 while (US ≠ ∅ and ⊥ ∉ US) do
 4 |  Given:= pick a clause from US;
 5 |  WO := WO ∪ {Given};
 6 |  New := SupLeft(WO,Given) ∪ Fact(Given);
 7 |  while (New ≠ ∅) do
 8 |  |  Given:= pick a clause from New;
 9 |  |  if (!TautDel(Given)) then
10 |  |  |  if (!SubDel(Given,WO ∪US)) then
11 |  |  |  |  Given:= Cond(Given);
12 |  |  |  |  Given:= SubRes(Given,WO);
13 |  |  |  |  WO:= SubDel(WO,Given);
14 |  |  |  |  US:= SubDel(US,Given);
15 |  |  |  |  New:= New ∪ SubRes(WO ∪US,Given);
16 |  |  |  |  US:= US ∪ {Given };
17 |  |  |
18 |  |
19 |  end
20 end
21 return WO;
```