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 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 2.5.3) is satisfiability preserving, i.e., the linear time CNF of $\neg\phi$ is unsatisfiable iff $\neg\phi$ is.

Proposition 2.5.2. 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. □

2.5.2 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 $\phi$.
   Output: A propositional formula $\psi$ equivalent to $\phi$ in CNF.
 1 whilerule (ElimEquiv($\phi$)) do ;
 2 whilerule (ElimImp($\phi$)) do ;
 3 whilerule (ElimTB1($\phi$),...,ElimTB6($\phi$)) do ;
 4 whilerule (PushNeg1($\phi$),...,PushNeg3($\phi$)) do ;
 5 whilerule (PushDisj($\phi$)) do ;
 6 return $\phi$;
```

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 2.5.3. Consider the formula $\neg((P \vee Q) \leftrightarrow (P \rightarrow (Q \wedge \top)))$ 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 highly redundant. If I remove all disjunctions that are trivially true, because they contain a propositional literal and its negation, the result becomes

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

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 $\neg Q \lor \neg P$ in this sense. The notion of subsumption will be discussed in detail for clauses in Section 2.6. 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, Section 2.5.3, I will present an algorithm $\Rightarrow_{\mathrm{ACNF}}$ that actually generates a much more compact CNF out of $\neg((P \lor Q) \leftrightarrow (P \rightarrow (Q \land \top)))$ and does this without generating the mess of formulas $\Rightarrow_{\mathrm{BCNF}}$ does, see Figure 2.10. Applying standard redundancy elimination rules Tautology Deletion, Condensation, and Subsumption, see Section 2.6 and Section 2.7, then actually generates $\neg Q$ as the overall result. Please recall that the above rules apply modulo commutativity of $\lor$, $\land$, e.g., the rule ElimTB1 is both applicable to the formulas $\phi \land \top$ and $\top \land \phi$.

$\boxed{\mathtt{I}}$ The equivalences in Figure 2.1 suggest more potential for simplification. For example, the idempotency equivalences $(\phi \land \phi) \leftrightarrow \phi$, $(\phi \lor \phi) \leftrightarrow \phi$ 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 $(P \lor Q) \land (Q \lor P)$ does this way not reduce to $(Q \lor P)$. A solution is to consider identity modulo commutativity. But then identity modulo commutativity and associativity (AC) as in $((P \lor Q) \lor R) \land (Q \lor (R \lor P)$ 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 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.

2.5.3 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 2.5.3 serves already as a nice motivation for (i) and (iii). Firstly, removing $\top$ from the formula $\neg((P \lor Q) \leftrightarrow (P \rightarrow (Q \land \top)))$ 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 $\Rightarrow_{\mathrm{BCNF}}$ is prevented. In general, if $\psi[\phi_1 \leftrightarrow \phi_2]_p$ and $\mathrm{pol}(\psi, p) = -1$ then for CNF transformation the equivalence is replaced by $\psi[(\phi_1 \land \phi_2) \lor (\neg\phi_1 \land \neg\phi_2)]_p$ and if $\mathrm{pol}(\psi, p) = 1$ by $\psi[(\phi_1 \rightarrow \phi_2) \land (\phi_2 \rightarrow \phi_1)]_p$ in ψ.

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-1} 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

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

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

1. If $\mathrm{pol}(\psi, p) = 1$, then $\psi[\phi]_p$ is satisfiable if and only if $\psi[P]_p \land (P \rightarrow \phi)$ is satisfiable.

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

3. If $\mathrm{pol}(\psi, p) = 0$, then $\psi[\phi]_p$ is satisfiable if and only if $\psi[P]_p \land (P \leftrightarrow \phi)$ is satisfiable.

Proof. Exercise. □

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

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

$\boxed{\text{C}}$ The polarity dependent definition of some predicate P introduces fewer clauses in case $\operatorname{pol}(\psi, p)$ has polarity 1 or -1. Still, even if always an equivalence is used to define predicates, for a properly chosen renaming the number of eventually generated clauses remains polynomial. Depending on the afterwards used calculus the former or latter results in a typically smaller search space. If a calculus relies on an explicitly building a partial model, e.g., CDCL, Section 2.9 and Section 2.10, then always defining predicates via equivalences is to be preferred. It guarantees that once the valuation of all variables in $\psi|_p$ is determined, also the value P is determined by propagation. If a calculus relies on building inferences in a syntactic way, e.g., Resolution, Section 2.6 and Section 2.12, then using a polarity dependent definition of P results in fewer inference opportunities.

For renaming there are several choices which subformula to choose. Obviously, since a formula has only linearly many subformulas, renaming every subformula works [30, 25]. However, this produces a number of renamings that do even increase the size of an eventual CNF. For example renaming in $\psi[\neg\phi]_p$ the subformulas $\neg\phi$ and ϕ at positions p, $p1$, respectively, produces more clauses than just renaming one position out of the two. This will be captured below by the notion of an *obvious position*. Then, in the following section a renaming variant is introduced that actually produces smallest CNFs. For all variants, renaming relies on a set of positions $\{p_1, \ldots, p_n\}$ that are replaced by fresh propositional variables.

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

The term $\phi[P_1]_{p_1}[P_2]_{p_2}\ldots[P_n]_{p_n}$ is evaluated left to right, i.e., a shorthand for $(\ldots((\phi[P_1]_{p_1})[P_2]_{p_2})\ldots[P_n]_{p_n})$. 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 [25], 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.

A smaller set of positions from ϕ, called *obvious positions*, is still preventing the explosion and given by the rules: (i) p is an obvious position 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 or (ii) pq is an obvious position 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.

A formula $\phi|_p$ is conjunctive in ϕ if $\phi|_p$ is a conjunction and $\operatorname{pol}(\phi, p) \in \{0, 1\}$ or $\phi|_p$ is a disjunction or implication and $\operatorname{pol}(\phi, p) \in \{0, -1\}$. Analogously, a formula $\phi|_p$ is disjunctive in ϕ if $\phi|_p$ is a disjunction or implication and $\operatorname{pol}(\phi, p) \in \{0, 1\}$ or $\phi|_p$ is a conjunction and $\operatorname{pol}(\phi, p) \in \{0, -1\}$.

Example 2.5.5. Consider as an example the formula

$$\phi = [\neg(\neg P \lor (Q \land R))] \rightarrow [P \lor (\neg Q \leftrightarrow \neg R)]\,.$$

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

$$\{22, 112\}$$

where 22 is obvious, because $\phi|_{22}$ is an equivalence and $\phi|_2$ is disjunctive, case (i) of the above definition. The position 112 is obvious, because it is conjunctive and $\phi|_{11}$ is a disjunctive formula, case (ii) of the above definition. Both positions are also considered by the Plaisted and Greenbaum definition, but they also add the positions $\{11, 2\}$ to this set, resulting in the set

$$\{2, 22, 11, 112\}.$$

Then applying SimpleRenaming to ϕ with respect to obvious positions results in

$$[\neg(\neg P \lor P_1)] \rightarrow [P \lor P_2] \land (P_1 \rightarrow (Q \land R)) \land (P_2 \rightarrow (\neg Q \leftrightarrow \neg R))$$

and applying SimpleRenaming with respect to the Plaisted Greenbaum positions results in

$$\begin{array}{c}[\neg P_3] \rightarrow [P_4] \land (P_1 \rightarrow (Q \land R)) \land (P_2 \rightarrow (\neg Q \leftrightarrow \neg R)) \quad \land \\ (P_3 \rightarrow (\neg P \lor P_1)) \land (P_4 \rightarrow (P \lor P_2))\end{array}$$

where I applied in both cases a polarity dependent definition of the freshly introduced propoaitional variables. A CNF generated by bcnf out of the renamed formula using obvious positions results in 5 clauses, where the renamed formula using the Plaisted Greenbaum positions results in 7 clauses.

I

Formulas are naturally implemented by trees in the style of the tree in Figure 2.9. Every node contains the connective of the respective subtree and an array with pointers to its children. Optionally, there is also a back-pointer to the father of a node. Then a subformula at a particular position can be represented by a pointer to the respective subtree. The polarity or position of a subformula can either be a stored additionally in each node, or, if back-pointers are available, it can be efficiently computed by traversing all nodes up to the root.

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

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

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

Furthermore, the advanced algorithm eliminates $\top$ and $\bot$ before eliminating $\leftrightarrow$ and $\rightarrow$. Therefore the respective rules are added:

ElimTB7	$\chi[\phi \rightarrow \bot]_p$	$\Rightarrow_{\text{ACNF}}$	$\chi[\neg\phi]_p$
ElimTB8	$\chi[\bot \rightarrow \phi]_p$	$\Rightarrow_{\text{ACNF}}$	$\chi[\top]_p$
ElimTB9	$\chi[\phi \rightarrow \top]_p$	$\Rightarrow_{\text{ACNF}}$	$\chi[\top]_p$
ElimTB10	$\chi[\top \rightarrow \phi]_p$	$\Rightarrow_{\text{ACNF}}$	$\chi[\phi]_p$
ElimTB11	$\chi[\phi \leftrightarrow \bot]_p$	$\Rightarrow_{\text{ACNF}}$	$\chi[\neg\phi]_p$
ElimTB12	$\chi[\phi \leftrightarrow \top]_p$	$\Rightarrow_{\text{ACNF}}$	$\chi[\phi]_p$

where the two rules ElimTB11, ElimTB12 for equivalences are applied with respect to commutativity of $\leftrightarrow$.

Algorithm 3: acnf(ϕ)

```
Input  : A formula $\phi$.
Output: A formula $\psi$ in CNF satisfiability preserving to $\phi$.
1 whilerule (ElimTB1($\phi$),...,ElimTB12($\phi$)) do ;
2 SimpleRenaming($\phi$) on obvious positions;
3 whilerule (ElimEquiv1($\phi$),ElimEquiv2($\phi$)) do ;
4 whilerule (ElimImp($\phi$)) do ;
5 whilerule (PushNeg1($\phi$),...,PushNeg3($\phi$)) do ;
6 whilerule (PushDisj($\phi$)) do ;
7 return $\phi$;
```

For an implementation the Algorithm 3 can be further improved. For example, once equivalences are eliminated the polarity of each literal is exactly known. So eliminating implications and pushing negations inside is not needed. Instead the eventual CNF can be directly constructed from the formula.

Proposition 2.5.6 (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., $\phi' = \phi[P]_p \wedge \operatorname{def}(\phi, p, P)$. There are three cases depending on the polarity. (i) if $\operatorname{pol}(\phi, p) = 1$ then $\phi' = \phi[P]_p \wedge P \rightarrow \phi|_p$. 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 $\operatorname{pol}(\phi, p) = 1$ also $\mathcal{A}(\phi) = 1$ by Lemma 2.2.7. (ii) if $\operatorname{pol}(\phi, p) = -1$ the case is symmetric to the previous one. Finally, (iii) if $\operatorname{pol}(\phi, p) = 0$ for any $\mathcal{A}$ satisfying ϕ' it holds $\mathcal{A}(\phi|_p) = \mathcal{A}(P)$ and hence $\mathcal{A}(\phi) = 1$. □

Note that Proposition 2.5.6 does not hold the other way round. Whenever a formula is manipulated by introducing fresh symbols, the truth of the original formula does not depend on the truth of the fresh symbols. For example, consider the formula

$$\phi \lor \psi$$

which is renamed to

$$\phi \lor P \land P \to \psi$$

.

Then any interpretation $\mathcal{A}$ with $\mathcal{A}(\phi) = 1$ is a model for $\phi \lor \psi$. It is not necessarily a model for $\phi \lor P \land P \to \psi$. If $\mathcal{A}(P) = 1$ and $\mathcal{A}(\psi) = 0$ it does not satisfy $\phi \lor P \land P \to \psi$.

$\boxed{\texttt{C}}$

The introduction of fresh symbols typically does not preserve validity but only satisfiability of formulas. Hence, it is well-suited for refutational reasoning based on a CNF, but not for equivalence reasoning based on a DNF. On the other hand renaming is mandatory to prevent a potential explosion of the formula size by normal form transformation. This is one explanation while typical automated reasoning calculi rely on a CNF. An alternative would be to develop automated reasoning calculi like resolution or CDCL on the formula level. It is an open research question whether this can lead to more efficient calculi.

$\boxed{\texttt{I}}$

All techniques in this section ignore redundancy, i.e., it might actually happen that a renamed formula produces eventually more clauses than the original formula due to redundancy. Putting it to the extreme, consider a complicated formula composed of only the propositional variable P, e.g., a nested equivalence $P \leftrightarrow (P \leftrightarrow P)$. Such a formula is always equivalent to $\top$, $\bot$, P, or $\neg P$ by applying the rules (I) and (VII) of Figure 2.1 to the eventual CNF. However, once it is renamed the redundancy is no longer detected by the mentioned rules, because of the freshly introduced variables. Therefore, in practice, the renaming techniques introduced in this section are comnbined with redundancy elimination techniques.

2.5.4 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 [6, 24]. 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 [6]. The 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 Section 2.6. 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.

2.6 Propositional Resolution

The propositional resolution calculus operates on a set of clauses and tests unsatisfiability. This enables advanced CNF transformation and, in particular, renaming, see Section 2.5.3. In order to check validity of a formula ϕ we check unsatisfiability of the clauses generated from $\neg\phi$.

Recall, see Section 2.1, that for clauses I switch between the notation as a disjunction, e.g., $P \vee Q \vee P \vee \neg R$, and the multiset notation, 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 2.6.1. The resolution calculus is sound and complete:

$$N \text{ is unsatisfiable iff } N \Rightarrow^*_{\text{RES}} N' \text{ and } \bot \in N' \text{ for some } N'$$

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 $C_1 \vee P, C_2 \vee \neg P$: if 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 comp(L), (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 finite semantic tree such that for each leaf of the tree there is a clause from N that is false with respect to the partial valuation at that leaf. By structural induction on the size of the tree we prove completeness. If the tree is empty, then $\bot \in N$. Now consider two sister leaves of the same parent of this tree, where the edges are labeled with L and comp(L), respectively. Let C_1 and C_2 be the two false clauses at the respective leaves. If some C_i does neither contain L or comp(L) then C_i is also false at the parent and we are done. So assume both C_1 and C_2 contain L or comp(L): $C_1 = C_1' \lor L$ and $C_2 = C_2' \lor \neg L$. If C_1 (or C_2) contains further occurrences of L (or C_2 of comp(L)), then the rule Factoring is applied to eventually remove all additional occurrences. Therefore, eventually $L \notin C_1'$ and comp$(L) \notin C_2'$. Note that if some C_i contains both L and comp(L) it is true, contradicting the assumption that C_i is false at its leaf. A resolution step between these two clauses on L yields $C_1' \lor C_2'$ which is false at the parent of the two leaves, because the resolvent neither contains L nor comp(L). Furthermore, the resulting tree is smaller, proving completeness. □

Example 2.6.2 (Resolution Completeness)**.** Consider the clause set

$$P \lor Q,\ \neg P \lor Q,\ P \lor \neg Q,\ \neg P \lor \neg Q \lor S,\ \neg P \lor \neg Q \lor \neg S$$

and the corresponding semantic tree as shown in Figure 2.13.

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

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

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 2.7, I will show what "simpler" means.

Example 2.6.3 (Refutation by Simplification)**.** Consider the clause set

$$N = \{P \lor Q,\ P \lor \neg Q,\ \neg P \lor Q,\ \neg P \lor \neg Q\}$$

that can be deterministically refuted by Subsumption Resolution:

$$\begin{array}{ll}
\multicolumn{2}{l}{(\{P \vee Q,\, P \vee \neg Q,\, \neg P \vee Q,\, \neg P \vee \neg Q\})} \\
\Rightarrow_{\mathrm{RES}}^{\mathrm{SubRes}} & (\{P \vee Q,\, P,\, \neg P \vee Q,\, \neg P \vee \neg Q\}) \\
\Rightarrow_{\mathrm{RES}}^{\mathrm{Subumption}} & (\{P,\, \neg P \vee Q,\, \neg P \vee \neg Q\}) \\
\Rightarrow_{\mathrm{RES}}^{\mathrm{SubRes}} & (\{P,\, Q,\, \neg P \vee \neg Q\}) \\
\Rightarrow_{\mathrm{RES}}^{\mathrm{SubRes}} & (\{P,\, Q,\, \neg Q\}) \\
\Rightarrow_{\mathrm{RES}}^{\mathrm{SubRes}} & (\{P,\, Q,\, \bot\})
\end{array}$$

where I abbreviated the rule Subumption Resolution by SubRes.

While the above example can be refuted by the rule Subsumption Resolution, the Resolution rule itself may derive redundant clauses, e.g., a tautology.

$$\begin{array}{ll}
\multicolumn{2}{l}{(\{P \vee Q,\, P \vee \neg Q,\, \neg P \vee Q,\, \neg P \vee \neg Q\})} \\
\Rightarrow_{\mathrm{RES}}^{\mathrm{Resolution}} & (\{P \vee Q,\, P \vee \neg Q,\, \neg P \vee Q,\, \neg P \vee \neg Q,\, Q \vee \neg Q\})
\end{array}$$

For three variables, the respective clause set is

$$\begin{array}{l}
(\{P \vee Q \vee R,\, P \vee \neg Q \vee R,\, \neg P \vee Q \vee R,\, \neg P \vee \neg Q \vee R, \\
\; P \vee Q \vee \neg R,\, P \vee \neg Q \vee \neg R,\, \neg P \vee Q \vee \neg R,\, \neg P \vee \neg Q \vee \neg R\})
\end{array}$$

C

The above deterministic, linear resolution refutation, Example 2.6.3, cannot be simulated by the tableau calculus without generating an exponential overhead, see also the comment on page 49. At first, it looks strange to have the same rule, namely Factorization and Condensation, both as a reduction rules and as an inference rule. On the propositional level there is obviously no difference and it is possible to get rid of one of the two. In Section **??** the resolution calculus is lifted to first-order logic. In first-order logic Factorization and Condensation are actually different, i.e., a Factorization inference is no longer a Condensation simplification, in general. They are separated here to eventually obtain the same set of rules propositional and first-order logic. This is needed for a proper lifting proof of first-order completeness that us actually reduced to the ground fragment of first-order logic that can be considered as a variant of propositional logic.

Proposition 2.6.4. The reduction rules Subsumption, Tautology Deletion, Condensation and Subsumption Resolution 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 $\mathcal{A}(C_2) = 1$. Subsumption Resolution is the combination of a Resolution application followed by a Subsumption application. □

Theorem 2.6.5 (Resolution Termination)**.** If reduction rules are preferred over inference rules and no inference rule is applied twice to the same clause(s), then $\Rightarrow_{\mathrm{RES}}^{+}$ is well-founded.

Proof. If reduction rules are preferred over inference rules, then the overall length if a clause cannot exceed n, where n is the number of variables. Multiple occurrences of the same literal are removed by rule Condensation, multiple occurrences of the same variable with different sign result in an application of rule Tautology Deletion. The number of such clauses can be overestimated by 3^n because every variable occurs at most once positively, negatively or not at all in clause. Hence, there are at most $2n3^n$ different resolution applications. □

Of course, what needs to be shown is that the strategy employed in Theorem 2.6.5 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 instance of superposition completeness.

2.7 Propositional Superposition

Superposition was originally developed for first-order logic with equality [1]. 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, based on the ordering for inference guidance, and (iv) a *saturation* concept.

Definition 2.7.1 (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.

For example, if $P \prec Q$, then $P \prec_L \neg P \prec_L Q \prec_L \neg Q$ and $P \lor Q \prec_C P \lor Q \lor Q \prec_C \neg Q$ because $\{P, Q\} \prec_L^{\text{mul}} \{P, Q, Q\} \prec_L^{\text{mul}} \{\neg Q\}$.

Proposition 2.7.2 (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 2.7.3 (Abstract Redundancy)**.** A clause C is *redundant* with respect to a clause set N if $N^{\prec C} \models C$.