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

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

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

Backtrack $(M_1 K^{i+1} M_2; N; U; k; D \vee L) \Rightarrow_{\text{CDCL}} (M_1 L^{D\vee L}; N; U \cup \{D \vee L\}; i; \top)$
provided L is of maximal level k in $D \vee L$ 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 $M \not\models N$

Here $\bot$ denotes the empty clause, hence fail. The level of the empty clause $\bot$ is 0. The clause $D \vee L$ 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 2.11. 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 2.9.1 (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.

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

Example 2.9.2 (CDCL Strategy II)**.** Consider again the clause set $N = \{P \lor Q, \neg P \lor Q, \neg Q\}$ from Example 2.9.1. For the following CDCL derivation the rules Propagate and Conflict are preferred over the other rules.

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

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 2.9.2 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 2.9.3 (Reasonable CDCL Strategy)**.** A CDCL strategy is *reasonable* if Conflict is always preferred over rule Propagate is always preferred over all other rules.

Proposition 2.9.4 (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

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

contains an infinite number of Backtrack applications.

8. CDCL never learns the same clause twice if Conflict selects the smalles 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 2.6.1) and an inductive argument it is enatailed 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 $(M_1 K^{i+1} M_2; N; U; k; D \vee L_{n+1}^k)$ where $M \models \neg(D \vee L_{n+1}^k)$ (with Proposition 2.9.4-3) 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 2.9.4-4 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 2.9.4-5 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 $(ML^{C \vee L}; N; U; k; D \vee \neg L) \Rightarrow_{\text{CDCL}} (M; N; U; k; D \vee C)$. For $D \vee C = \top$ there must be a literal K with $K, \neg K \in (D \vee C)$. With Proposition 2.9.4-3 $M \models \neg(D \vee C)$ holds which is equivalent to $M \models \bot$,a contradiction because of Proposition 2.9.4-1. 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 2.11.4. □

Lemma 2.9.5. 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 2.9.4-6 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 $C = D \vee L$. With Proposition 2.9.4-3 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 2.9.6 (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 2.9.4-1 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 2.9.5 there are only 2 termination states, i.e. ϕ can only be $\top$ or $\bot$.

Case $\phi = \top$ then by Lemma 2.9.5 $M \models N$. This is a contradiction to the assumption that $M \models N$ does not hold.

Case $\phi = \bot$ then by Lemma 2.9.5 N is unsatisfiable. This is a contradiction to N being satisfiable. □

Therefore all rules of the CDCL algorithm are sound.

Proposition 2.9.7 (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 $M = L_1 L_2 \ldots L_k$. Since M is a valuation there are no duplicates in M and k applications of rule Decide yield $(L_1^1 L_2^2 \ldots L_k^k; N; \emptyset; k; \top)$ 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 2.9.7 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 2.9.7 let $M = L_1 L_2 \ldots L_k$. First apply Propagate m-times exhaustively resulting in $(L_1 \ldots L_m; N; \emptyset; 0; \top)$ where $m \leq k$. With Proposition 2.9.4-4 the literals $L_1 \ldots L_m$ must be true in any valuation $\mathcal{A}$ with $\mathcal{A} \models N$. Thus, if $m = k$ then $(L_1 \ldots L_m; N; \emptyset; 0; \top)$ is a final state and $M \models N$. If $m < k$ then apply Decide once on a literal from M resulting in $(L_1 \ldots L_m L^1; N; \emptyset; 1; \top)$. 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 2.9.8 (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 2.9.4-**??** and 2.9.4-8 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 2.9.4-7 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 2.9.8 can be applied and the algorithm eventually terminates.

Example 2.9.9 (CDCL Termination I)**.** Consider the clause set $N = \{P \vee Q, \neg P \vee Q, \neg Q\}$. The CDCL algorithm does not terminate due to the rule Restart.

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

Example 2.9.10 (CDCL Termination II)**.** Consider the clause set $N = \{\neg P \vee Q \vee \neg R, \neg P \vee Q \vee R\}$. The CDCL algorithm does not terminate due to the rule Forget.

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

C

As an alternative for the proof of Proposition 2.9.8 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 2.9.4-8 it holds that $D \vee L \notin U$ so that the first component decreases.

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

2.10 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.

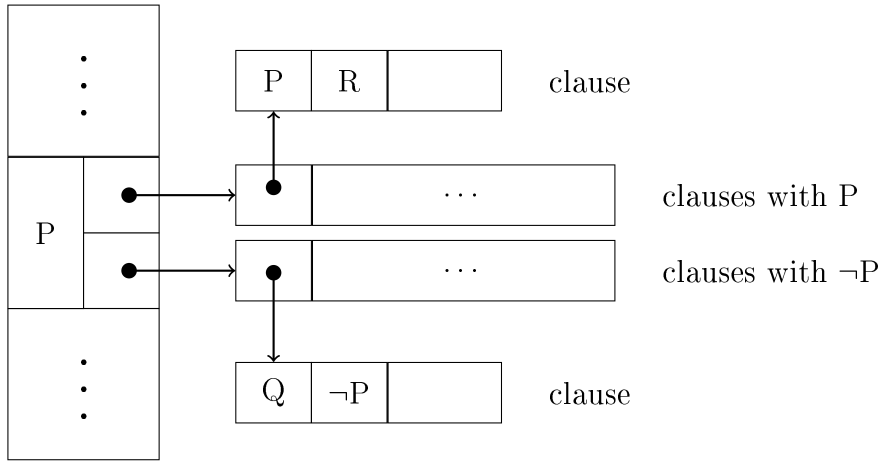


Figure 2.10: The watched literals list with the variables P, Q, R and the watched literals P, R and $\neg P$, Q.

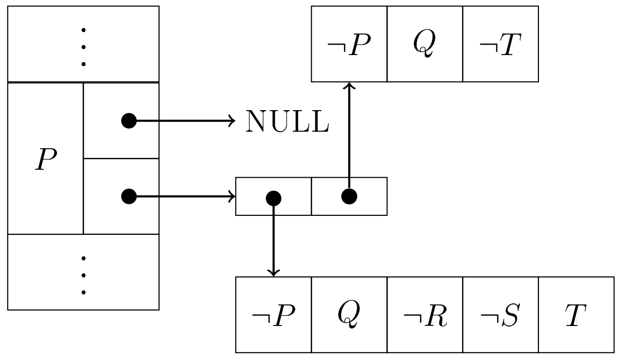
2.10.1 Lazy Data Structure: 2-Watched Literals (2WL)

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 2.10 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:

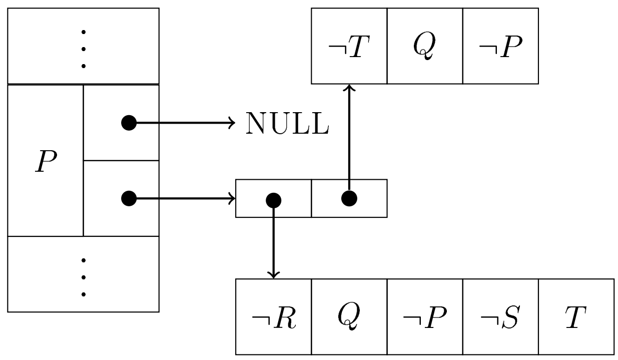
1. Let an unassigned variable P be set to false (or true).
2. Visit all clauses in which P (or $\neg P$) is watched.
3. In every clause where P (or $\neg P$) is watched find an unwatched and non-falsified variable to be watched. If there is no other unassigned or true variable then this clause is either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

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).

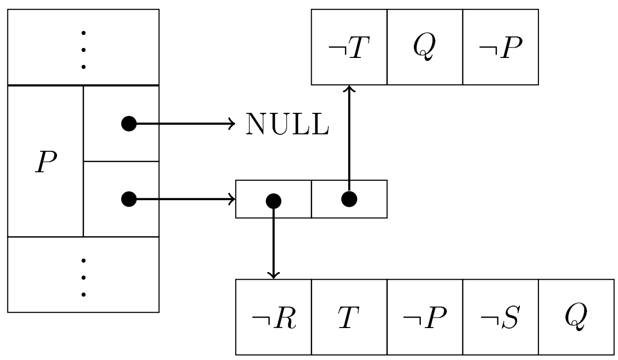
As an example consider the formula $\phi = \{\neg P \vee Q \vee \neg R \vee \neg S \vee T, \neg P \vee Q \vee \neg T, R \vee T, S \vee T\}$. Figure 2.13 shows how to derive unit clauses and finally satisfy the formula within the watched literals data structure. The watched literals are the first two entries in a clause. The trail (see next section on Backtracking) represents the assigned literals for the current state.



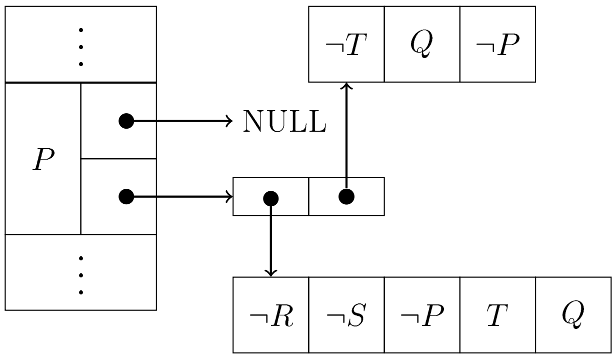
(a) Initialized 2WL data structure for the literal P and the current trail is empty.



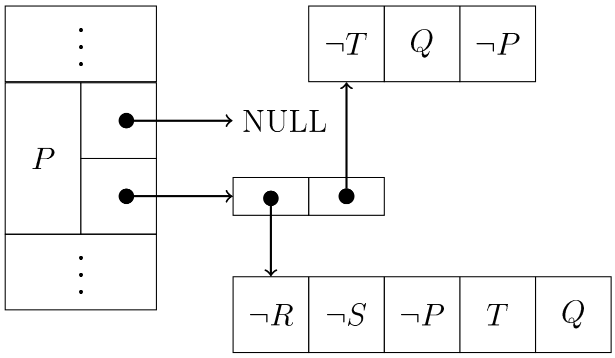
(b) After deciding P the watched literals have changed and the current trail is: P.



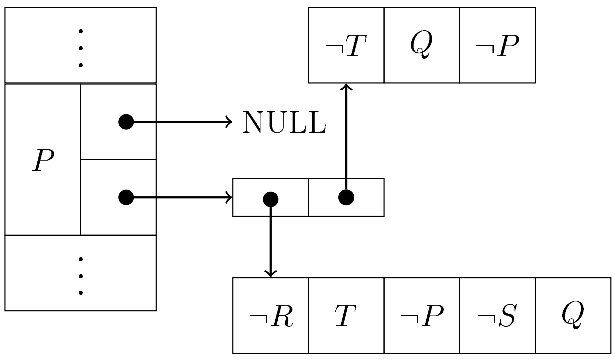
(c) After deciding $\neg Q$ the unit clause $\{\neg P \vee Q \vee \neg T\}$ is achieved and the current trail is: $P, \neg Q$.



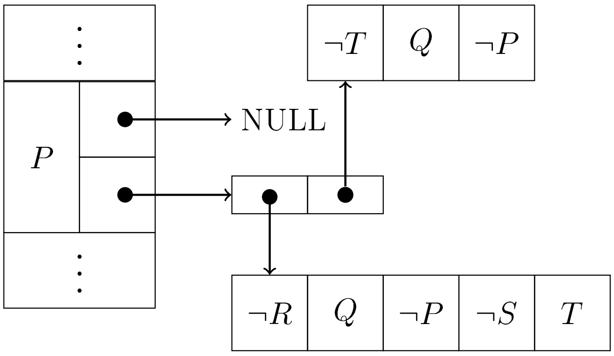
(d) After propagating $\neg T, R$ and S the current trail is: $P, \neg Q, \neg T, R, S$ and the clause $\{\neg P \vee Q \vee \neg R \vee \neg S \vee T\}$ evaluates to false, a conflict.



(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 2.13: 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.

2.10.2 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.

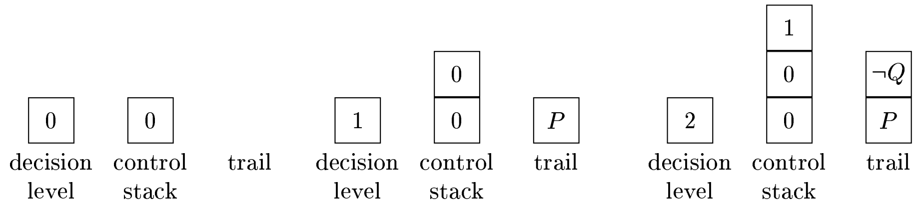
If the rule Backtrack is applied the trail height entry from the control stack is taken and every variable from that trail height on will be unassigned, i.e. every assignment value that was made since the last application of the rule Decide is deleted. A detailed example is shown in Figure 2.14. Again, the advantage with the watched literals data structure is that the watched variables stay unchanged and will not be considered by this backtracking step.

2.10.3 Dynamic Decision Heuristic: VSIDS

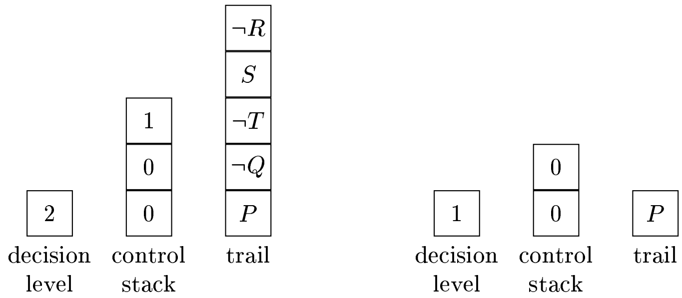
Choosing the right unassigned variable to decide is important for efficiency, but the heuristic may be expensive itself. Therefore, the aim is to use a heuristic that needs not to be recomputed too often, that for example chooses variables which occur frequently and prefers variables from recent conflicts.

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.



(a) The initial entries. (b) After deciding P. (c) After deciding $\neg Q$.



(d) After propagating ¬T, S and ¬R. (e) After backtracking.

Figure 2.14: 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. 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.

2.10.4 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 $C = D \vee L$ 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 2.15 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 2.14. The corresponding watched literals list is shown in Figure 2.19.

2. Identify the conflict sets by means of the implication graph, i.e. the cuts of the graph need to be considered. In Figure 2.15 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 2.15, then take the asserting clause that contains the 1UIP. In Figure 2.16 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 2.15 the clause $\neg P \vee Q$ will be learned.

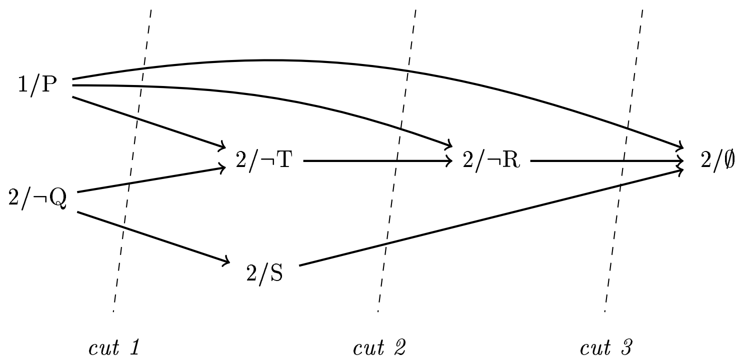


Figure 2.15: An implication graph for the formula ϕ with cuts.

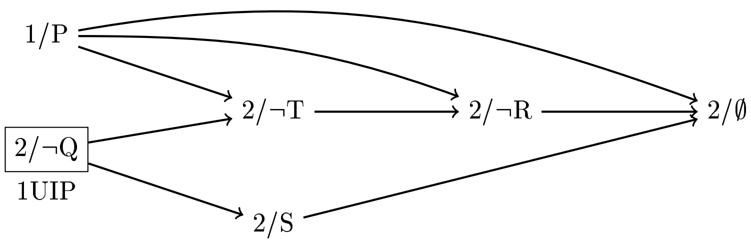


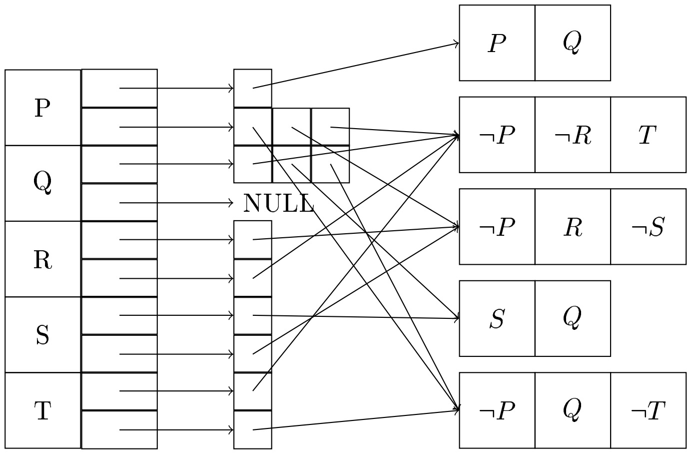
Figure 2.16: The implication graph denoted with the 1UIP.

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.

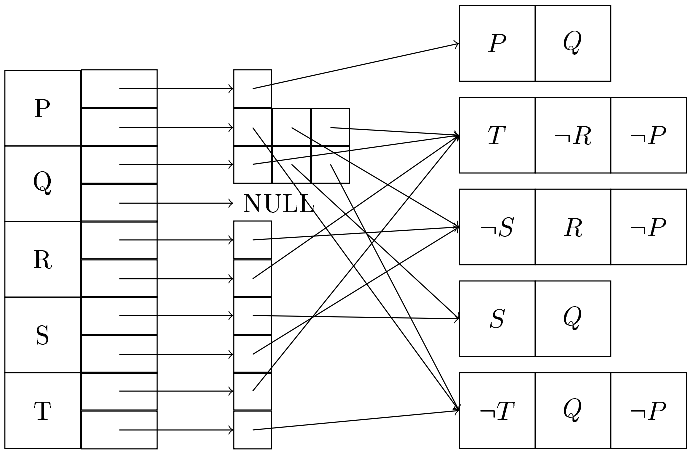
2.10.5 Restart and Forget

As mentioned in the section on VSIDS (see 2.10.3) 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 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.

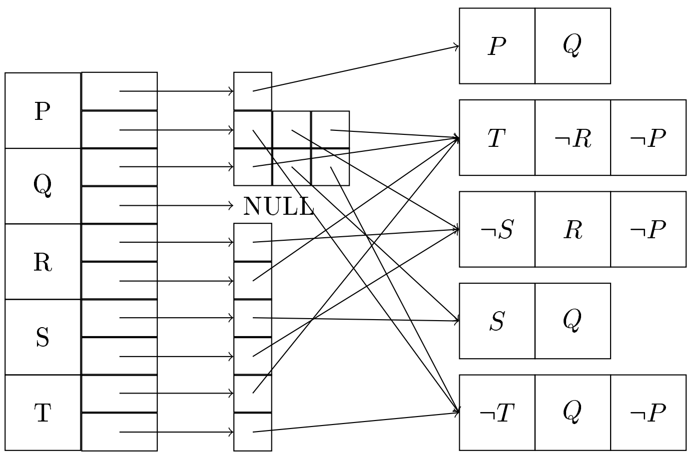
A further technique that contributes to the performance of the CDCL solver



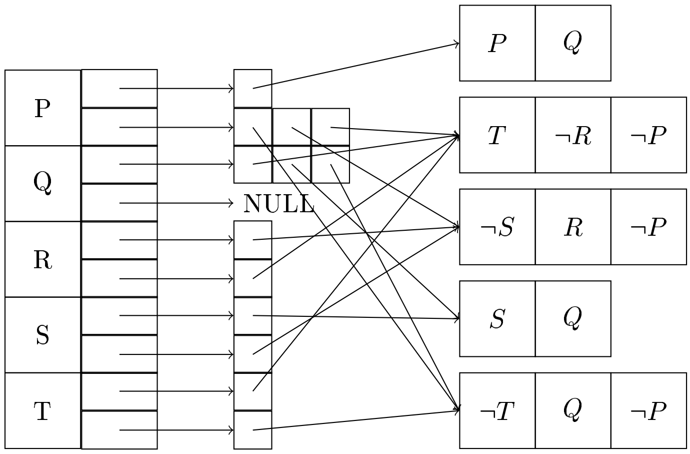
(a) The initial state and the current trail is empty.



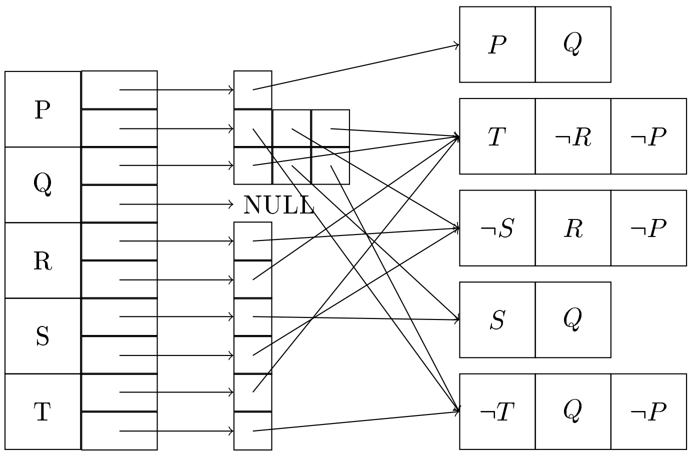
(b) After deciding P watched literals are swapped, the trail is: P.



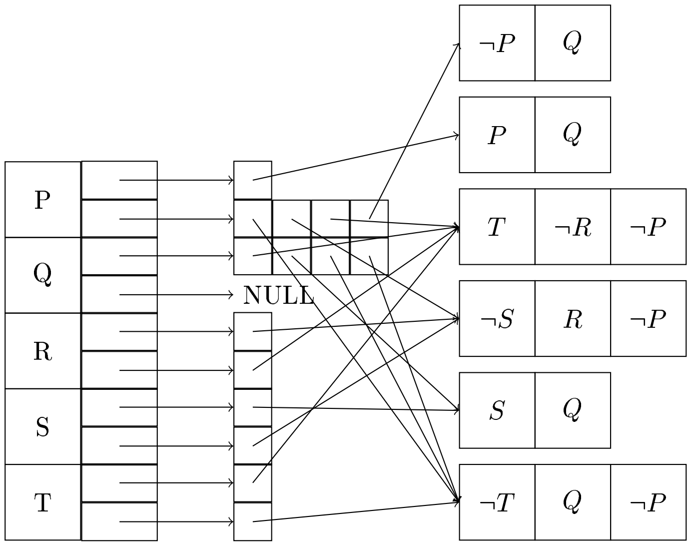
(c) After deciding $\neg Q$, no change in the watched literals, the trail is: $P, \neg Q$.



(d) After propagating $\neg T$, S and $\neg R$, no change of watched literals but a conflict occurs in $\neg P \lor R \lor S$, 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 2.19: The watched literals list according to the implication graph from Figures 2.15 and 2.16 as well as the control stack, trail and decision level of Figure 2.14.

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.

2.10.6 Algorithm and Strategy

As shown in the examples 2.9.1 and 2.9.2 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.

Algorithm 5: CDCL(S)

Input : An initial state $(\epsilon; N; \emptyset; 0; \top)$.
Output: A final state $S = (M; N; U; k; \top)$ or $S = (M; N; U; k; \bot)$

```
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);
```

2.11 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 2.7.5. 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.

Definition 2.11.1 (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 $N^{\mathcal{H}}_{\Sigma}$ for N and signature Σ, with $P, Q \in \Sigma$ is inductively constructed as follows:

$$
\begin{aligned}
N^{\mathcal{H}}_P &:= \textstyle\bigcup_{Q \prec P} \delta^{\mathcal{H}}_Q \\
\delta^{\mathcal{H}}_P &:= \begin{cases} \{P\} & \text{if } (D \vee P) \in N, P \text{ strictly maximal and } N^{\mathcal{H}}_P \not\models D \quad \text{or} \\ & \mathcal{H}(P) = 1 \text{ and for all clauses } (C \vee \neg P) \in N, C \prec \neg P \\ & \text{ it holds } N^{\mathcal{H}}_P \models C \\ \emptyset & \text{otherwise} \end{cases} \\
N^{\mathcal{H}}_{\Sigma} &:= \textstyle\bigcup_{P \in \Sigma} \delta^{\mathcal{H}}_P
\end{aligned}
$$

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

T

Proposition 2.11.2. If $\mathcal{H}(P) = 0$ for all $P \in \Sigma$ then $N_{\mathcal{I}} = N^{\mathcal{H}}_{\Sigma}$ for any N.

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

Case 1: $P \in N_{\mathcal{I}}$ but $P \notin N^{\mathcal{H}}_{\Sigma}$.
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 $N^{\mathcal{H}}_P \not\models D'$ and therefore $\delta^{\mathcal{H}}_P = \{P\}$ contradicting $P \notin N^{\mathcal{H}}_{\Sigma}$.

Case 2: $P \notin N_{\mathcal{I}}$ but $P \in N^{\mathcal{H}}_{\Sigma}$.
Then there is a productive clause $D = D' \vee P \in N$ such that P is strictly maximal in this clause and $N^{\mathcal{H}}_P \not\models D'$ 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 $P \notin N_{\mathcal{I}}$. $\square$

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 2.7.9. Recall that the same notion of redundancy, Definition 2.7.3, is used.

Theorem 2.11.3. 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 $C_1 \lor L \in N$ such that $N_{\mathcal{I}} \not\models C_1 \lor L$ and L is a maximal literal in $C_1 \lor L$. This clause must exist because $\bot \notin N$.

The clause $C_1 \lor L$ is not redundant. For otherwise, $N^{\prec C_1 \lor L} \models C_1 \lor L$ and hence $N_\Sigma^{\mathcal{H}} \models C_1 \lor L$, because $N_\Sigma^{\mathcal{H}} \models N^{\prec C_1 \lor L}$, 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 $C_1 \lor P$ then actually $\delta_P^{\mathcal{H}} = \{P\}$ and hence $N_P^{\mathcal{H}} \models C_1 \lor P$, a contradiction. So P is not strictly maximal. But then actually $C_1 \lor P$ has the form $C_1' \lor P \lor P$ and Factoring derives $C_1' \lor P$ where $(C_1' \lor P) \prec (C_1' \lor P \lor P)$. Now $C_1' \lor P$ is not redundant, strictly smaller than $C_1 \lor L$, we have $C_1' \lor P \in N$ and $N_\Sigma^{\mathcal{H}} \not\models C_1' \lor P$, a contradiction against the choice that $C_1 \lor L$ is minimal.

Secondly, assume L is negative, i.e., $L = \neg P$ for some propositional variable P. Then, since $N_\Sigma^{\mathcal{H}} \not\models C_1 \lor \neg P$ 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 $C_2 \lor P \in N$ where P is strictly maximal and by definition $(C_2 \lor P) \prec (C_1 \lor \neg P)$. So a Superposition Left inference derives $C_1 \lor C_2$ where $(C_1 \lor C_2) \prec (C_1 \lor \neg P)$. The derived clause $C_1 \lor C_2$ cannot be redundant, because for otherwise either $N^{\prec C_2 \lor P} \models C_2 \lor P$ or $N^{\prec C_1 \lor \neg P} \models C_1 \lor \neg P$. So $C_1 \lor C_2 \in N$ and $N_\Sigma^{\mathcal{H}} \not\models C_1 \lor C_2$, a contradiction against the choice that $C_1 \lor L$ is minimal. Secondly, there is no clause $C_2 \lor P \in N$ where P is strictly maximal but $\mathcal{H}(P) = 1$. But a further condition for this case is that there is no clause $(C_1 \lor \neg P) \in N$ such that $N_P^{\mathcal{H}} \not\models C_1$ contradicting the above choice of $C_1 \lor \neg P$. $\square$

Recalling Section 2.7 Superposition is based on an ordering $\prec$. It relies on a model assumption $N_{\mathcal{I}}$, Definition 2.7.5 or its generalization $N_\Sigma^{\mathcal{H}}$, Definition 2.11.1. 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 2.7.9 or Theorem 2.11.3, 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 2.11.4. 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. ☐

Proposition 2.11.4 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 2.11.4 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 2.11.5. Consider the superposition left inference between the clauses $P \vee Q$ and $R \vee \neg Q$ with ordering $P < R < Q$ resulting in $P \vee R$. Changing the ordering to $Q < P < R$ the inference $P \vee R$ becomes redundant. So flipping infinitely often between $P < R < Q$ and $Q < P < R$ is already sufficient to prevent any saturation progress.

Although Example 2.11.5 shows that changing the ordering is not compatible with redundancy and superposition completeness, Proposition 2.11.4 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 2.11.6 (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 2.11.7 (Length Redundancy and Superposition)**.** Arbitrary Ordering Changes plus fairness plus length redundancy preserves completeness.

Theorem 2.11.8 (Length Redundancy and CDCL)**.** At any time length redundant clauses may be removed.

2.12 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 2.5.3. 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 2.5) 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 2.7.3, Example 2.11.5, and Theorem 2.11.8, Theorem 2.11.7.

2.12.1 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 2.7. Purity and Blocked Clause Deletion are new.

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

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

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

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

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

Example 2.12.1. Consider a clause set consisting of the five clauses
(1) $P \vee Q$
(2) $P \vee Q \vee R \vee S$
(3) $\neg R \vee S$
(4) $R \vee \neg S$
(5) $\neg Q \vee S$
Clause (1) subsumes clause (2). Subsumption resolution is applicable to clause (2) and clause (5) resulting in $P \vee R \vee S$. Purity is applicable to P. Blocked clause elimination is not applicable.

Applying first subsumption deletion results in the clauses
(1) $P \vee Q$
(3) $\neg R \vee S$
(4) $R \vee \neg S$
(5) $\neg Q \vee S$
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) $P \vee Q$
(5) $\neg Q \vee S$
Now P and S are pure and after applying purity the result is the empty set of clauses indicating satisfiability.

For the above Example 2.12.1 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, $\Rightarrow_{\text{RBSC}}$ is not confluent.

Lemma 2.12.2 ($\Rightarrow_{\text{RBSC}}$ terminates)**.**

Proof. Exercise □

Lemma 2.12.3 ($\Rightarrow_{\text{RBSC}}$ is sound)**.** If $(N) \Rightarrow_{\text{RBSC}} (N')$ 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 ot 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. □

2.12.2 Redundancy while Superposition and CDCL

2.13 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 2.5.3 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 2.13.1 (Pigeon Hole Formulas $\text{ph}(n)$)**.** 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) 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 ph(n) grows cubic in n. The famous theorem on the pigeon whole formulas says that any resolution proof showing unsatisfiability of 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 2.13.2 (Haken [23])**.** The length of any resolution refutation of 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 ph(n) we put pigeon $n+1$ in hole n. Then the problem is solvable iff 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 $\mathrm{ph}(n)$. 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 [14].

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 $\mathrm{ph}(1)$, i.e., the above resolution steps successfully perform the reduction from $\mathrm{ph}(2)$ to $\mathrm{ph}(1)$.

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 2.5.3). 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 2.13.3 (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 2.13.4 (k-SAT)**.** A propositional clause set N belongs to the class of *k-SAT* problems if every clause contains at most k literals.

Proposition 2.13.5. 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 2.11.3). 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. $\square$

Actually, the proof of the above proposition implies that the CDCL rules Propagate and Conflict (see Section 2.9) 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 $P \lor Q$, $\neg P \lor R$, $\neg R \lor Q$, $\neg Q$ is unsatisfiable and refutable by unit superposition. It is not Horn because of the clause $P \lor Q$. Now by flipping the sign of Q in all clauses results in the clause set $P \lor \neg Q$, $\neg P \lor R$, $\neg R \lor \neg Q$, Q which is Horn, equisatisfiable, and still unit refutable.

Proposition 2.13.6. 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 2.13.5. 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

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

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 2.13.7. 3-SAT is NP-complete.

2.14 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.

2.14.1 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 $i \leq k, k' < i+n$ and $j \leq l, l' < j+n$ 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 2.20: A 4×4 Sudoku

The application of this encoding on the puzzle from Figure 2.20 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 2.6 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 2.21. After exhaustive application of the Resolution calculus the remaining unit constraints are derived and the solution is found.

2.14.2 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

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

Figure 2.21: A 4×4 Sudoku after generating the unit constraint $P^2_{3,4}$

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 2.22.

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 2.22 hold.

As an example the satisfiability of the logic circuit in Figure 2.22 under a given partial truth assignment $\beta(Q_0) = 1$ and $\beta(Q_5) = 1$ can be checked using the DPLL calculus:

The application of the encoding to the logic circuit of Figure 2.22 together with the partial truth assignment β yields a total of 12 clauses: $N = \{Q_0, Q_5, \neg Q_4 \vee Q_2 \vee Q_1, \neg Q_2 \vee Q_4, \neg Q_1 \vee Q_4, \neg Q_2 \vee \neg Q_0, Q_2 \vee Q_0, \neg Q_3 \vee \neg Q_1, Q_3 \vee Q_1, \neg Q_5 \vee Q_4, \neg Q_5 \vee Q_3, \neg Q_4 \vee \neg Q_3 \vee Q_5\}$. 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 $M = (Q_0Q_5Q_4Q_3\neg Q_1Q_2)$

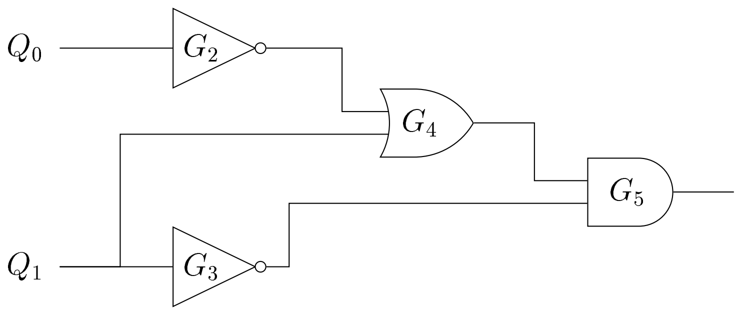


Figure 2.22: A logic circuit with two NOT-gates (G_2 and G_3), an OR-gate G_4 and an AND-gate G_5

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 2.22 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 2.22 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 [10] is known as "propositional logic". For a nice historic perspective see Martin Davis's book [16].

Chapter 3

First-Order Logic

3.1 Syntax

Definition 3.1.1 (Many-Sorted Signature)**.** A *many-sorted signature* $\Sigma = (\mathcal{S}, \Omega, \Pi)$ is a pair 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 $\operatorname{sort}(t)$, i.e., if $t \in T_S(\Sigma, \mathcal{X})$ then $\operatorname{sort}(t) = S$. 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 $P \in \Pi$, $\operatorname{arity}(P) = n$, $P \subseteq S_1 \times \ldots \times S_n$. An atom or its negation of an atom is called a *literal*.

The literal $s \mathrel{\dot{\approx}} t$ 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 $s \not\approx t$.

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 $\mathrm{FOL}(\Sigma, \mathcal{X})$ 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 $x \in \mathcal{X}$:

$\mathrm{FOL}(\Sigma, \mathcal{X})$	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 $\mathrm{PROP}(\Sigma) \subseteq \mathrm{FOL}(\Sigma', \mathcal{X})$ 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:

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

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.

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. $\mathrm{pos}(\neg P(f(x, g(a)))) = \{\epsilon, 1, 11, 111, 112, 1121\}$ 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

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