Chapter 1

Preliminaries

This chapter introduces all abstract concepts needed for the rest of this book. Generic problem solving actually starts with a problem. In this book problems will appear in the form of examples. In order to solve a problem in a generic way, i.e., by generic algorithms, the first step is to formalize the problem using a generic language. A generic language has a mathematically precise syntax and semantics, because eventually it is analyzed by a program running on a computer. Such a language is called a *logic*. The problem becomes a *sentence*, i.e., a *formula* of the logic. In particular, semantics in this context always means a notion of truth. The notion of truth is a very expressive instrument to actually formalize what it means to eventually solve a particular problem. A solution to the formula should result in a solution to the problem. Detecting that the formula is true (false) corresponds to solving the problem.

Once the problem is described in a logic, the generic language, it needs rules that reason about the truth of formulas and hence eventually solve the problem. A logic plus its reasoning rules is called a *calculus*. The rules operate on a symbolic representation of a *problem state* that includes in particular the formula formalizing the problem. Typically, further information is added to the state representation in order to keep track of the solution process. The rules should enjoy a number of properties in order to be useful. They should be sound, i.e., whenever they compute a solution the result is actually a solution to the initial problem. And whenever they compute that there is no solution this should hold as well. The rules should be complete, i.e., whenever there is a solution to the problem they compute it. Finally, they should be terminating. If they are applied to a starting problem state, they always stop after a finite number of steps. Typically, because no more rule is applicable. Depending on the complexity of the problem and the involved logic, not all the desired properties soundness, completeness, termination, can be achieved, in general. But I will turn to this later.

The rules of a calculus are typically designed to operate independently and can therefore be executed in a non-deterministic way. The advantage of such a presentation is that properties of the rules, e.g., like soundness, can also be

shown independently for each rule. And if a property can be shown for the rule set, it applies to all potential execution orderings of the rules. The disadvantage of such a presentation is that a random application of the rules typically leads to an inefficient algorithm. Therefore, a strategy is added to the calculus (rules) and the strategy plus the rules build an *automated reasoning algorithm* or shortly an *algorithm*. Depending on the type of property and the actual calculus, sometimes we prove it for the calculus or the respective algorithm.

An automated reasoning algorithm is still an abstract, mathematical construct and there is typically a significant gap between such an algorithm and an actual computer program implementing the algorithm. An implementation often requires a dedicated control of the calculus plus the invention of specific data structures and algorithms. The implementation of an algorithm is called a *system*. Eventually the system is applied to real world problems, i.e., an *application*.

Application
System + Problem
System
Algorithm + Implementation
Algorithm
Calculus + Strategy
Calculus
Logic + States + Rules
Logic
Syntax + Semantics

Typically computer science algorithms are formulated in languages that are close to actual programming languages such as C, C++, or Java[1]. So, in particular, they rely on deterministic programming languages with an operational semantics. I overload the notion of a classical computer science algorithm and an automated reasoning algorithm. An automated reasoning algorithm is build on a rule set plus a strategy and typically the strategy does not turn the rules into a deterministic algorithm. There is still some room left that will eventually be decided for an application. The difference in design reflects the difference in scope. A classical computer science algorithm solves a very specific problem, e.g., it sorts a finite list of numbers. An algorithm is meant to solve a whole class of problems, e.g., later on I will show that ordered resolution can solve any polynomial time computable problem based on a fragment of first-order logic.

As a start, Section 1.1 studies the overall above approach including all mentioned properties in full detail on a concrete problem: 4×4-Sudokus. Although this is a rather trivial and actually finite problem and the suggested algorithm is

[1] copyright

very naive, it serves nicely as a throughout example demonstrating all aspects. Later on, I will develop far more complex logics that then can be used to solve more interesting problems. In particular, real world problems.

The subsequent sections abstract from solving Sudokus and develop the underlying concepts needed as a basic toolbox for the rest of this book. Basic mathematical notions on numbers, sets, relations, and words are defined in Section 1.2. In order to be able to talk about the complexity of algorithms Section 1.3 in particular explains Big O notation and NP-hardness. Section 1.4 is devoted to orderings, because they show up on the meta-level, e.g. as a means to prove termination. They also serve as a basis for proving properties of rule sets by induction, Section 1.5, and also on the logical reasoning level where they will be actually an effective means for defining more efficient rule sets. Finally, Section 1.6 introduces the most important concepts of rule based reasoning in general by an introduction to basic concepts of (abstract) rewrite systems.

1.1 Solving 4×4 Sudoku

Consider solving a 4×4 Sudoku as it is depicted on the left in Figure 1.1. The goal is to fill in natural numbers from 1 to 4 into the 4×4 square so that in each column, row and 2×2 box sharing an outer corner with the original square each number occurs exactly once. Conditions of this kind are called *constraints* as they restrict filling the Sudoku with numbers in an arbitrary way. The Sudoku (Solution) on the right (Figure 1.1) shows the, in this case, unique solution to the Sudoku (Start) on the left.

2	1		
		3	1
1		2	

Start

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

Solution

Figure 1.1: A 4×4 Sudoku and its Solution

Why is this solution unique? It is because the constraints of 4×4 Sudokus have already forced all other values. To start, the only square for the missing 1 is the square above the 3. All other squares would violate a constraint. But then the third column is almost filled so the top square of this column must be a 4, and so on.

In the following, I will build a specific logic for 4×4 Sudokus, including an algorithm in form of a set of rules and a strategy for solving the problem and actually prove that the algorithm is *sound*, *complete*, and *terminating*. As already said, an algorithm is sound if any solution the algorithm declares to have found is actually a solution. It is complete if it finds a solution in case

one exists. It is terminating if it does not run forever. Since Sudokus are finite combinatorial puzzles, such an algorithm exists. The most simple algorithm is to systematically guess all values for all undefined squares of the Sudoku and to check whether the guessed values actually constitute a solution. However, this amounts to checking 4^{16} different assignments of values to the squares. Such an approach is even worse than the one I will introduce in the sequel.

I consider a Sudoku to be a two dimensional array f indexed from 1 to 4 in each dimension, starting from the upper left corner. So $f(1,1)$ is the value of the square in the upper left corner and in case of our initial Sudoku. For the start Sudoku in Figure 1.1 the value of this square is given to be 2 which I denote by the equation $f(1,1) \approx 2$. So the logic for Sudokus are finite conjunctions (conjunction denoted by $\wedge$) of equations $f(x,y) \approx z$, where the variables x, y, z range over the domain 1, 2, 3, 4. The meaning of a conjunction is that all values given by the equations should be simultaneously true in the Sudoku. The overall left Sudoku (Start in Figure 1.1) is then given by the conjunction of equations

$$f(1,1) \approx 2 \wedge f(1,2) \approx 1 \wedge f(3,3) \approx 3 \wedge f(3,4) \approx 1 \wedge f(4,1) \approx 1 \wedge f(4,3) \approx 2$$

If you are already familiar with classical logic, you know that the formulas $f(1,1) \approx 2 \wedge f(1,2) \approx 1$ and $f(1,2) \approx 1 \wedge f(1,1) \approx 2$ cannot be distinguished semantically. They have always the same truth value, because conjunction ($\wedge$) is commutative, and, in addition, associative. However, here, the above conjunction will become part of a problem state. The sudoku logic rules syntactically manipulate problem states. A problem state containing $f(1,1) \approx 2 \wedge f(1,2) \approx 1$ will be different from one containing $f(1,2) \approx 1 \wedge f(1,1) \approx 2$, because the former implicitly means that there is no solution to the sudoku with $f(1,1) \approx 1$, whereas the latter means that there is no solution to the sudoku with $f(1,1) \approx 1$ in presence of $f(1,2) \approx 1$.

The goal of the algorithm is then to find the assignments for the empty squares with respect to the above mentioned constraints on the number occurrences in columns, rows and boxes. The algorithm consists of four rules that each take a state of the solution process and transform it into a different one, closer to a solution. A state is described by a triple $(N; D; r)$ where N contains the equations of the starting Sudoku, for example, the above conjunction of equations, D is a conjunction of additional equations computed by the algorithm, and $r \in \{\top, \bot\}$ describes whether the actual values for f in N and D potentially constitute a solution. If $r = \top$ then no constraint violation has been detected and if $r = \bot$ a constraint violation has been detected but not yet resolved. The initial problem state is represented by the triple $(N; \top; \top)$ where $\top$ also denotes an empty conjunction and hence truth. The problem state $(N; \top; \bot)$ denotes the fail state, i.e., there is no solution for a Sudoku starting with the assignments contained in N.

A square $f(x,y)$ where $x, y \in \{1,2,3,4\}$ is called *defined* by $N \wedge D$ if there is an equation $f(x,y) \approx z$, $z \in \{1,2,3,4\}$ in N or D. Otherwise, $f(x,y)$ is called *undefined*. For an initial state $(N; \top; \top)$ I assume that the same square is not

defined several times in N. We say that $N \wedge D'$ is a *solution* to a Sudoku N, if all squares are defined in $N \wedge D'$, no square is defined more than once in $N \wedge D'$ and the assignments in $N \wedge D'$ do not violate any constraint. It is a solution to a problem state $(N; D; \top)$ if all equations from D occur in D'. In the sequel we always assume that for any start state $(N; \top; \top)$ each square is defined at most once in N and all variables x, y, z (possibly indexed, primed) range over values 1 to 4. Then the four rules of a first (naive) algorithm are

Deduce $(N; D; \top) \quad \Rightarrow \quad (N; D \wedge f(x,y) \approx 1; \top)$
provided $f(x,y)$ is undefined in $N \wedge D$, for any $x, y \in \{1,2,3,4\}$.

Conflict $(N; D; \top) \quad \Rightarrow \quad (N; D; \bot)$
provided for (i) $f(x,y) = f(x,z)$ for $f(x,y)$, $f(x,z)$ defined in $N \wedge D$ for some x, y, z and $y \neq z$, or,
(ii) $f(y,x) = f(z,x)$ for $f(y,x)$, $f(z,x)$ defined in $N \wedge D$ for some x, y, z and $y \neq z$, or,
(iii) $f(x,y) = f(x',y')$ for $f(x,y)$, $f(x',y')$ defined in $N \wedge D$ and $[x, x' \in \{1,2\}$ or $x, x' \in \{3,4\}]$ and $[y, y' \in \{1,2\}$ or $y, y' \in \{3,4\}]$ and $(x,y) \neq (x',y')$.

Backtrack $(N; D' \wedge f(x,y) \approx z \wedge D''; \bot) \quad \Rightarrow \quad (N; D' \wedge f(x,y) \approx z+1; \top)$
provided $z < 4$ and $D'' = \top$ or D'' contains only equations of the form $f(x',y') \approx 4$.

Fail $(N; D; \bot) \quad \Rightarrow \quad (N; \top; \bot)$
provided $D \neq \top$ and D contains only equations of the form $f(x,y) \approx 4$.

Rules are applied to a state by first matching the left hand side of the rule (left side of $\Rightarrow$) to the state, checking the side conditions described below the rule and if they are fulfilled then replacing the state by the right hand side of the rule. There is no order among the rules, so they are applied "don't care nondeterministically". A strategy will fix the ordering and turn into an algorithm. Furthermore, even a single rule may not be deterministic. For example rule Deduce does not specify concrete values for x, y so it can be applied to any undefined square $f(x,y)$.

Starting with the state corresponding to the initial Sudoku shown on the left in Figure 1.1, a one step derivation by rule Deduce is $(N; \top; \top) \rightarrow (N; f(1,3) \approx 1; \top)$. Actually the rule Deduce is the only applicable rule to $(N; \top; \top)$. Concerning the new state $(N; f(1,3) \approx 1; \top)$ two rules are applicable: Deduce and Conflict. An application of Conflict, where side condition (i) is satisfied, yields $(N; f(1,3) \approx 1; \bot)$ and after an application of Backtrack to this state the rule computes $(N; f(1,3) \approx 2; \top)$. Applying Deduce to $(N; f(1,3) \approx 1; \top)$ results, e.g., in $(N; f(1,3) \approx 1. f(1,4) \approx 1; \top)$. Figure 1.2 shows this sequence of rule applications together with the corresponding Sudokus.

This is one reason why the rule set is inefficient. Deduce still fires in case of an already existing constraint violation and Deduce does not consider already

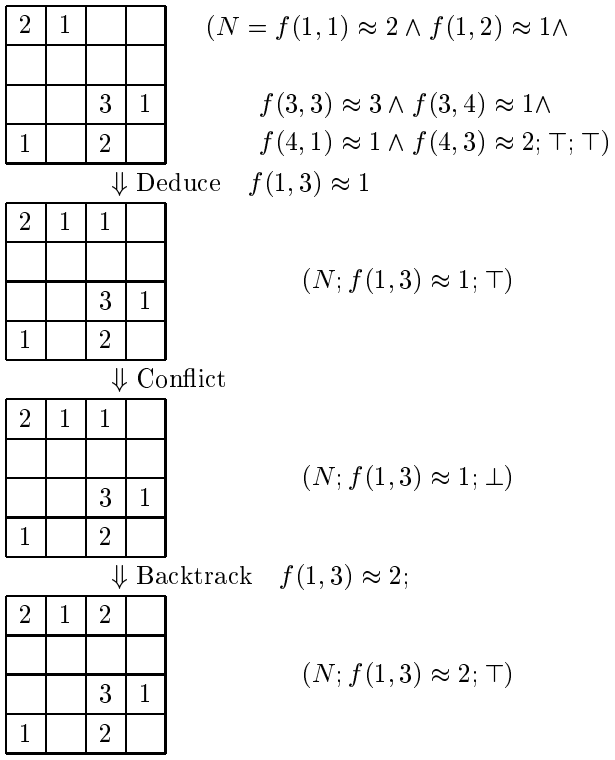

Figure 1.2: Effect of Applying the Inference Rules

existing equations when assigning a new value. It simply always assigns "1". Improving the algorithm along the second line is subject to Exercises **??**, **??**. Furthermore, note that if in a start state $(N; \top; \top)$ the initial assignments in N already contain a constraint violation, then the rule conflict directly produces the final fail state. An appropriate, very simple strategy turns the rule set into an algorithm and prefers Conflict over Deduce.

The Algorithm 1, SimpleSudoku(S), consists of the four rules together with a rule application strategy. The scope of loops and if-then-else statements is indicated by indentation. A statement ***Rule***(S) for some *Rule* means that the application of the rule is tested and if applicable it is applied to the problem state S. If such a statement occurs in a **ifrule** condition, it is applied as before and returns true iff (if and only if) the rule was applicable. For example, the statement at line 1

ifrule (**Conflict**(S)) **then**
    return S;

is a shorthand for

if (the rule **Conflict** is applicable to state S) **then**

Algorithm 1: SimpleSudoku(S)

Input : An initial state $S = (N; \top; \top)$.
Output: A final state $S = (N; D; \top)$ or $S = (N; \top; \bot)$

```
1  ifrule (Conflict(S)) then
2  |  return S;
3  while (any rule applicable) do
4  |  ifrule (Conflict(S)) then
5  |  |  Backtrack(S);
6  |  |  Fail(S);
7  |  else
8  |  |  Deduce(S);
9  |
10 end
11 return S;
```

apply rule **Conflict** to S;
return S;

where the application condition is separated from the rule application.

At line 1 the rule Conflict is tested and if applicable it will produce the final state $S = (N; \top; \bot)$, so the algorithm returns S. The while-loop starting at line 3 terminates if no rule is applicable anymore. For otherwise, the rule Conflict is tested before Deduce in order to prevent useless Deduce steps. The rules Backtrack and Fail are only applicable after an application of Conflict, so they are guarded by an application of Conflict. Therefore, SimpleSudoku is a fair algorithm in the sense that no rule application needed to compute a final state will be prohibited.

If the rules are considered in the context of the SimpleSudoku algorithm, then they can be simplified. For example, the condition for rule Fail that all equations are of the form $f(x, y) \approx 4$ can be dropped, because in SimpleSudoku the rule Fail is only tested and potentially applied after having tested Backtracking.

It is a design issue how much rule application control is actually put into the side conditions of the rules and how much control into the algorithm. It depends, of course, on the problem to be solved but also on which level properties can be shown. For SimpleSudoku all properties can be shown on the calculus, i.e., rule level. In general, showing termination of a rule set often requires a particular strategy, i.e., algorithm.

In the sequel, I will prove that the four rules are *sound*, *complete* and *terminating*. Sound means that whenever the rules compute some state $(N; D; \top)$ and it has a solution, then this solution is also a solution for N. Complete means that whenever there is a solution to the Sudoku, exhaustive application of the four rules will compute a solution. Note that for completeness the computation of any solution, not an a priori selected one, is sufficient. In case of the Sudoku

rules even strong completeness holds: for any solution $N \wedge D$ of the Sudoku, there is a sequence of rule applications so that $(N; D; \top)$ is a terminating state. So any a priori selected solution can be generated. Termination at the rule level means that independently of the actual sequence of rule applications to a start state, there is no infinite sequence of rule applications possible. In the sequel, I will consider a fourth property important for rule based systems: *confluence*. A set of rules is confluent if whenever there are several rules applicable to a given state, then the different generated states can be rejoined by further rule applications. So confluence guarantees unique results on termination. Because of the above informal fairness argument for the SimpleSudoku algorithm, all these properties also hold not only for the rule set but also for the algorithm.

Proposition 1.1.1 (Soundness)**.** The rules Deduce, Conflict, Backtrack and Fail are sound. Starting from an initial state $(N; \top; \top)$: (i) for any final state $(N; D; \top)$, the equations in $N \wedge D$ are a solution, and, (ii) for any final state $(N; \top; \bot)$ there is no solution to the initial problem.

Proof. First of all note that no rule manipulates N, the first component of a state $(N; D; r)$. This justifies the way this proposition is stated. (i) So assume a final state $(N; D; \top)$ so that no rule is applicable. In particular, this means that for all $x, y \in \{1, 2, 3, 4\}$ the square $f(x, y)$ is defined in $N \wedge D$ as for otherwise Deduce would be applicable, contradicting that $(N; D; \top)$ is a final state. So all squares are defined by $N \wedge D$. No square is defined more than once. What remains to be shown is that those assignments actually constitute a solution to the Sudoku. However, if some assignment in $N \wedge D$ results in a repetition of a number in some column, row or 2×2 box of the Sudoku, then rule Conflict is applicable, contradicting that $(N; D; \top)$ is a final state. In sum, $(N; D; \top)$ is a solution to the Sudoku and hence the rules Deduce, Conflict, Backtrack and Fail are sound.
(ii) So assume that the initial problem $(N; \top; \top)$ has a solution. I prove by contradiction based on an inductive argument that in this case the rules cannot generate a state $(N; \top; \bot)$. So let $(N; D; \top)$ be an arbitrary state with D of maximal length still having a solution, but $(N; \top; \bot)$ is reachable from $(N; D; \top)$. This includes the initial state if $D = \top$. An appropriate selection of rule applications correctly decides the next square. Since $(N; D; \top)$ still has a solution the only applicable rule is Deduce. It generates $(N; D \wedge f(x, y) \approx 1; \top)$ for some $x, y \in \{1, 2, 3, 4\}$. If $(N; D \wedge f(x, y) \approx 1; \top)$ still has a solution the proof is done since this violates D to be of maximal length. So $(N; D \wedge f(x, y) \approx 1; \top)$ does not have a solution anymore. But then eventually Conflict and Backtrack are applicable to a state $(N; D \wedge f(x, y) \approx 1 \wedge D'; \bot)$ where D' only contains equations of the form $f(x', y') \approx 4$ resulting in $(N; D \wedge f(x, y) \approx 2; \top)$. Now repeating the argument we will eventually reach a state $(N; D \wedge f(x, y) \approx k; \top)$ that has a solution, finally contradicting D to be of maximal length. □

For the first part of the soundness proof, Proposition 1.1.1, neither the rule Backtrack nor Fail shows up. This is because an empty rule system is trivially

sound. The rules Backtrack or Fail are indispensable for the second part of the proof and for showing completeness.

The above proof contains a "handwaving argument", the sentence "But then eventually Conflict and Backtrack are applicable to a state $(N; D \wedge f(x,y) \approx 1 \wedge D'; \bot)$ where D' only contains equations of the form $f(x',y') \approx 4$ resulting in $(N; D \wedge f(x,y) \approx 2; \top)$." needs a proof on its own. I will not do the proof here, but for some of the rule sets for deciding satisfiability of propositional logic, Chapter 2, I will do analogous proofs in full detail.

Proposition 1.1.2 (Strong Completeness)**.** The rules Deduce, Conflict, Backtrack and Fail are strongly complete. For any solution $N \wedge D$ of the Sudoku there is a sequence of rule applications so that $(N; D; \top)$ is a final state.

Proof. A particular strategy for the rule applications is needed to indeed generate $(N; D; \top)$ out of $(N; \top; \top)$ for some specific solution $N \wedge D$. Without loss of generality I assume the assignments in D to be sorted so that assignments to a number $k \in \{1,2,3,4\}$ precede any assignment to some number $l > k$. So if, for example, N does not assign all four values 1, then the first assignment in D is of the form $f(x,y) \approx 1$ for some x, y. Now I apply the following strategy, subsequently adding all assignments from D to $(N; \top; \top)$. The strategy has achieved state $(N; D'; \top)$ and the next assignment from D to be established is $f(x,y) \approx k$, meaning $f(x,y)$ is not defined in $N \wedge D'$. Then until $l = k$ the strategy does the following, starting from $l = 1$. It applies Deduce adding the assignment $f(x,y) \approx l$. If Conflict is applicable to this assignment, it is applied and then Backtrack, generating the new assignment $f(x,y) \approx l + 1$ and so on.

I need to show that this strategy in fact eventually adds $f(x,y) \approx k$ to D'. As long as $l < k$ any added assignment $f(x,y) \approx l$ results in rule Conflict applicable, because D is ordered and all four values for all $l < k$ are already established. The eventual assignment $f(x,y) \approx k$ does not generate a conflict because D is a solution. For the same reason, the rule Fail is never applicable. Therefore, the strategy generates $(N; D; \top)$ out of $(N; \top; \top)$. □

Note the subtle difference between the second part of proving Proposition 1.1.1 and the above strong completeness proof. The former shows that any solution can be produced by the rules whereas the latter shows that a specific, a priori selected solution can be generated.

Proposition 1.1.3 (Termination)**.** The rules Deduce, Conflict, Backtrack and Fail terminate on any input state $(N; \top; \top)$.

Proof. Once the rule Fail is applicable, no other rule is applicable on the result anymore. So there is no need to consider rule Fail for termination. The idea of the proof is to assign a *measure* over the natural numbers to every state so that each rule strictly decreases this measure and that the measure cannot get below 0. The measure is as follows.

For any given state $S = (N; D; r)$ with $r \in \{\top, \bot\}$ with $D = f(x_1, y_1) \approx k_1 \wedge \ldots \wedge f(x_n, y_n) \approx k_n$ I assign the measure $\mu(S)$ by

$$\mu(S) = 2^{49} - p - \sum_{i=1}^{n} k_i \cdot 2^{49-3i}$$

where $p = 0$ if $r = \top$ and $p = 1$ otherwise.

The measure $\mu(S)$ is well-defined and cannot become negative as $n \leq 16$, $p \leq 1$, and $1 \leq k_i \leq 4$ for any D. In particular, the former holds because the rule Deduce only adds values for undefined squares and the overall number of squares is bound to 16. What remains to be shown is that each rule application decreases μ. I do this by a case analysis over the rules.
Deduce:

$$\begin{aligned}\mu((N; D; \top)) &= 2^{49} - \textstyle\sum_{i=1}^{n} k_i \cdot 2^{49-3i} \\ &> 2^{49} - \textstyle\sum_{i=1}^{n} k_i \cdot 2^{49-3i} - 1 \cdot 2^{49-3(n+1)} \\ &= \mu((N; D \wedge f(x, y) \approx 1; \top))\end{aligned}$$

Conflict:

$$\begin{aligned}\mu((N; D; \top)) &= 2^{49} - \textstyle\sum_{i=1}^{n} k_i \cdot 2^{49-3i} \\ &> 2^{49} - 1 - \textstyle\sum_{i=1}^{n} k_i \cdot 2^{49-3i} \\ &= \mu((N; D; \bot))\end{aligned}$$

Backtrack:

$$\begin{aligned}&\mu((N; D' \wedge f(x_l, y_l) \approx k_l \wedge D''; \bot)) \\ &\quad= 2^{49} - 1 - (\textstyle\sum_{i=1}^{l-1} k_i \cdot 2^{49-3i}) - k_l \cdot 2^{49-3l} - \textstyle\sum_{i=l+1}^{n} k_i \cdot 2^{49-3i} \\ &\quad> 2^{49} - (\textstyle\sum_{i=1}^{l-1} k_i \cdot 2^{49-3i}) - (k_l + 1) \cdot 2^{49-3l} \\ &\quad= \mu(N; D' \wedge f(x_l, y_l) \approx k_l + 1; \top)\end{aligned}$$

where the strict inequation holds because $2^{49-3l} > \sum_{i=l+1}^{n} k_i \cdot 2^{49-3i} + 1$. □

As already mentioned, there is another important property for don't care non-deterministic rule sets: *confluence*. It means that whenever several sequences of rules are applicable to a given state, the respective results can be rejoined by further rule applications to a common problem state. A weaker condition is *local confluence* where only one step of different rule applications needs to be rejoined. In Section 1.6, Lemma 1.6.6, the equivalence of confluence and local confluence in case of a terminating rule system is shown. Assuming this result, for the Sudoku rule system only one step of so called *overlaps* needs to be considered. There are two potential kinds of overlaps for the Sudoku rule system. First, an application of Deduce and Conflict to some state. Second, two different applications of Deduce to a state. The below Proposition 1.1.4 shows that the former case can in fact be rejoined and Example 1.1.5 shows that the latter cannot. So in sum, the system is not locally confluent and hence not confluent. This fact has already shown up in the soundness and completeness proofs.

Proposition 1.1.4 (Deduce and Conflict are confluent)**.** Given a state $(N; D; \top)$ out of which two different states $(N; D_1; \top)$ and $(N; D_2; \bot)$ can be generated by Deduce and Conflict, respectively, then the two states can be rejoined to a state $(N; D'; *)$ via further rule applications.

Proof. Consider an application of Deduce and Conflict to a state $(N; D; \top)$ resulting in $(N; D \wedge f(x, y) \approx 1; \top)$ and $(N; D; \bot)$, respectively. We will now show that in fact we can rejoin the two states. Notice that since Conflict is applicable to $(N; D; \top)$ it is also applicable to $(N; D \wedge f(x, y) \approx 1; \top)$. So the first sequence of rejoin steps is

$$\begin{aligned} (N; D \wedge f(x, y) \approx 1; \top) &\Rightarrow (N; D \wedge f(x, y) \approx 1; \bot) \\ &\Rightarrow (N; D \wedge f(x, y) \approx 2; \top) \\ &\Rightarrow^* (N; D \wedge f(x, y) \approx 4; \bot) \end{aligned}$$

where we subsequently applied Conflict and Backtrack to reach the state $(N; D \wedge f(x, y) \approx 4; \bot)$ and $\Rightarrow^*$ abbreviates those finite number of rule applications. Finally applying Backtrack (or Fail) to $(N; D; \bot)$ and $(N; D \wedge f(x, y) \approx 4; \bot)$ results in the same state. □

Example 1.1.5 (Deduce is not confluent)**.** Consider the Sudoku state $(f(1, 1) \approx 1 \wedge f(2, 2) \approx 1; \top; \top)$ and two applications of Deduce generating the respective successor states $(f(1, 1) \approx 1 \wedge f(2, 2) \approx 1; f(3, 3) \approx 1; \top)$ and $(f(1, 1) \approx 1 \wedge f(2, 2) \approx 1; f(3, 4) \approx 1; \top)$. Obviously, both states can be completed to a solution, but don not have a common solution. Therefore, it will not be possible to rejoin the two states, see Figure 1.3.

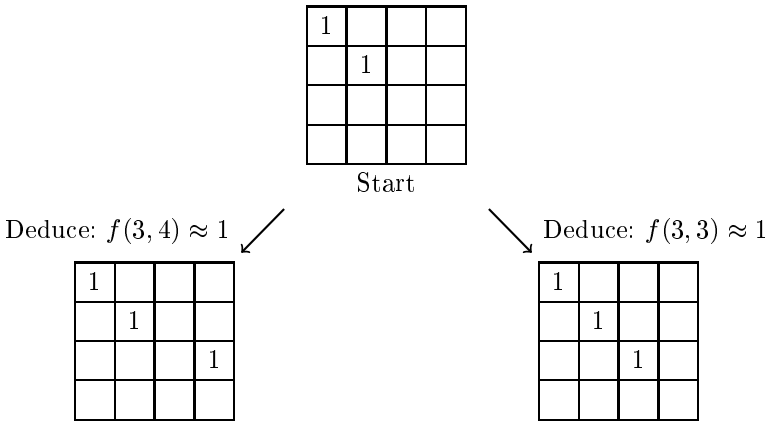

Figure 1.3: Divergence of Rule Deduce

Is it desirable that a rule set for Sudoku is confluent? It depends on the purpose of the algorithm. In case of the above rules set for Sudoku, strong completeness and confluence cannot both be achieved, because any solution of the Sudoku results in its own, unique, final state.

C

1.2 Basic Mathematical Prerequisites

The set of the natural numbers including 0 is denoted by $\mathbb{N}$, $\mathbb{N} = \{0, 1, 2, \ldots\}$, the set of positive natural numbers without 0 by $\mathbb{N}^+$, $\mathbb{N}^+ = \{1, 2, \ldots\}$, and the set of integers by $\mathbb{Z}$. Accordingly $\mathbb{Q}$ denotes the rational numbers and $\mathbb{R}$ the real numbers, respectively.

Given a set M, a *multi-set* S over M is a mapping $S\colon M \to \mathbb{N}$, where S specifies the number of occurrences of elements m of the base set M within the multiset S. I use the standard set notations $\in$, $\subset$, $\subseteq$, $\cup$, $\cap$ with the analogous meaning for multisets, for example $(S_1 \cup S_2)(m) = S_1(m) + S_2(m)$. I also write multi-sets in a set like notation, e.g., the multi-set $S = \{1, 2, 2, 4\}$ denotes a multi-set over the set $\{1, 2, 3, 4\}$ where $S(1) = 1$, $S(2) = 2$, $S(3) = 0$, and $S(4) = 1$. A multi-set S over a set M is *finite* if $\{m \in M \mid S(m) > 0\}$ is finite. For the purpose of this book I only consider finite multi-sets.

An n-ary *relation* R over some set M is a subset of M^n: $R \subseteq M^n$. For two n-ary relations R, Q over some set M, their union ($\cup$) or intersection ($\cap$) is again an n-ary relation, where $R \cup Q := \{(m_1, \ldots, m_n) \in M \mid (m_1, \ldots, m_n) \in R \text{ or } (m_1, \ldots, m_n) \in Q\}$ and $R \cap Q := \{(m_1, \ldots, m_n) \in M \mid (m_1, \ldots, m_n) \in R \text{ and } (m_1, \ldots, m_n) \in Q\}$. A relation Q is a *subrelation* of a relation R if $Q \subseteq R$. The *characteristic function* of a relation R or sometimes called *predicate* indicates membership. In addition of writing $(m_1, \ldots, m_n) \in R$ I also write $R(m_1, \ldots, m_n)$. So the predicate $R(m_1, \ldots, m_n)$ holds or is true if in fact $(m_1, \ldots, m_n)$ belongs to the relation R.

Given a nonempty alphabet Σ the set Σ^* of finite words over Σ is defined by the (i) empty word $\epsilon \in \Sigma^*$, (ii) for each letter $a \in \Sigma$ also $a \in \Sigma^*$ and, finally, (iii) if $u, v \in \Sigma^*$ so $uv \in \Sigma^*$ where uv denotes the concatenation of u and v. The length $|u|$ of a word $u \in \Sigma^*$ is defined by (i) $|\epsilon| := 0$, (ii) $|a| := 1$ for any $a \in \Sigma$ and (iii) $|uv| := |u| + |v|$ for any $u, v \in \Sigma^*$.

1.3 Basic Computer Science Prerequisites

1.3.1 Data Structures

1.3.2 While Languages over Rules

When presenting pseudocode for algorithms in textbooks typically so called **while** languages are used (e.g., see [15]). I assume familiarity with such languages and specialize it here to rules. So let **Rule** be a rule defined on some state S. Then

Rule(S);

is a shorthand for

if Rule is applicable to S **then** apply it once to S;

where in particular nothing happens if **Rule** is not applicable to S. There may be several potential applications of **Rule** to S. In this case any of these is chosen. The statement

whilerule(**Rule**(S)) **do** *Body*;

is a shorthand for

while (**Rule** is applicable to S) **do**
  apply **Rule** once to S;
  execute *Body*;

where the scope of the **while** loop is shown by indentation. The condition of the **whilerule** statement may also be a disjunction of rule statements. In this case the disjunction is executed in a non-deterministic, lazy way. We use || to indicate the disjunction. Furthermore, a single rule statement may be followed by a negation, indicated by !. In this case the rule is tested for application, if it is applicable it is applied and the condition becomes false. If the rule is not applicable the condition becomes true. Except for these extensions, boolean combinations over rule statements are not part of the language. Finally, the statement

ifrule(**Rule**(S)) **then** *Body*;

is a shorthand for

if (**Rule** is applicable to S) **then**
  apply **Rule** once to S;
  execute once *Body*;

In Section 1.1 I have already used the language for describing an algorithm solving sudokus, Algorithm 1, SimpleSudoku(S).

1.3.3 Complexity

This book is about algorithms solving problems presented in logic. Such an algorithm is typically represented by a finite set of rules, manipulating a problem state that contains the logical representation plus bookkeeping information. For example, for solving 4×4-Sudokus, see Section 1.1, we represented the board by a finite conjunction of equations. The problem state was given by the representation of the board plus assignments for remaining empty squares, plus an indication whether two conflicting assignments have been detected. The rules then take a start problem state and eventually transform it into a solved form. In order to compare the performance of this rule set with a different one or to give an overall performance guarantee of the rule set, the classical way in computer science is to consider the (worst case) running time until termination. A consequence of the Sudoku termination proof, Lemma 1.1.3, is that *at most* 2^{49} rule applications are needed. Generalizing this result, for a given $n \times n$-Sudoku, the running time would by of "order" n^{n^2}, because in the worst case we need to

guess n different numbers for each square and there are n^2 squares of the board. The so called *big O* notation covers the term "order" formally.

Definition 1.3.1 (Big O)**.** Let $f(n)$ and $g(n)$ be functions from the naturals into the nonnegative reals. Then

$$O(f(n)) = \{g(n) \mid \exists\, c > 0\, \exists\, n_0 \in \mathbb{N}^+\, \forall\, n \geq n_0\; g(n) \leq c \cdot f(n)\}$$

Thus, the running time of the Sudoku algorithm for an $n \times n$-Sudoku is $O(n^{n^2})$, if the number of rule applications are taken to be the constant time units. This sounds somewhat surprising because it means that the algorithm will already fail for reasonably small n, if implemented in practice. For example, for the well-established 9×9-Sudoku puzzles the algorithm will in the worst case need about $9^{81} \approx 2 \cdot 10^{77}$ rule applications to figure out whether a given Sudoku has a solution. This way, assuming a fast computer that can perform 1 Million rule applications per second it will take longer to solve a single Sudoku than the currently estimated age of the universe. Nevertheless, human beings typically solve a 9×9-Sudoku in some minutes. So what is wrong here? First of all, as I already said, the algorithm presented in Section 1.1 is completely naive. This algorithm will definitely not solve 9×9-Sudokus in reasonable time. It can be turned into an algorithm that will work nicely in practice, see Exercise (**??**). Nevertheless, problems such as Sudokus are difficult to solve, in general. Testing whether a particular assignment is a solution can be done efficiently, in case of Sudokus in time $O(n^2)$. For the purpose of this book, I say a problem can be *efficiently solved* if there is an algorithm solving the problem and a polynomial $p(n)$ so that the execution time on inputs of size n is $O(p(n))$. Although it is efficient for Sudokus to validate whether an assignment is a solution, there are exponentially many possible assignments to check in order to figure out whether there exists a solution. So if we are allowed to make guesses, then Sudokus can be solved efficiently. This property describes the class of NP (Nondeterministic Polynomial) problems in general that I will introduce now.

A *decision problem* is a subset $L \subseteq \Sigma^*$ for some fixed finite alphabet Σ. The function $\mathrm{chr}(L, x)$ denotes the *characteristic function* for some decision problem L and is defined by $\mathrm{chr}(L, u) = 1$ if $u \in L$ and $\mathrm{chr}(L, u) = 0$ otherwise. A decision problem is solvable in polynomial-time iff its characteristic function can be computed in polynomial-time. The class P denotes all polynomial-time decision problems.

Definition 1.3.2 (NP)**.** A decision problem L is in NP iff there is a predicate $Q(x, y)$ and a polynomial $p(n)$ so that for all $u \in \Sigma^*$ we have (i) $u \in L$ iff there is an $v \in \Sigma^*$ with $|v| \leq p(|u|)$ and $Q(u, v)$ holds, and (ii) the predicate Q is in P.

A decision problem L is *polynomial time reducible* to a decision problem L' if there is a function $g \in \mathrm{P}$ so that for all $u \in \Sigma^*$ we have $u \in L$ iff $g(u) \in L'$. For example, if L is reducible to L' and $L' \in \mathrm{P}$ then $L \in \mathrm{P}$. A decision problem is NP-*hard* if every problem in NP is polynomial time reducible to it. A decision

problem is NP-*complete* if it is NP-hard and in NP. Actually, the first NP-complete problem [7] has been propositional satisfiability (SAT). Chapter 2 is completely devoted to solving SAT.

1.3.4 Word Grammars

When Gödel presented his undecidability proof on the basis of arithmetic, many people still believed that the construction is so artificial that such problems will never arise in practice. This didn't change with Turing's invention of the Turing machine and the undecidable halting problem of such a machine. However, then Post presented his correspondence problem in 1946 [18] it became obvious that undecidability is not an artificial concept.

Definition 1.3.3 (Finite Word)**.** Given a nonempty alphabet Σ the set Σ^* of *finite words* over Σ is defined by

1. the empty word $\epsilon \in \Sigma^*$
2. for each letter $a \in \Sigma$ also $a \in \Sigma^*$
3. if $u, v \in \Sigma^*$ so $uv \in \Sigma^*$ where uv denotes the concatenation of u and v.

Definition 1.3.4 (Length of a Finite Word)**.** The length $|u|$ of a word $u \in \Sigma^*$ is defined by

1. $|\epsilon| := 0$,
2. $|a| := 1$ for any $a \in \Sigma$ and
3. $|uv| := |u| + |v|$ for any $u, v \in \Sigma^*$.

Definition 1.3.5 (Word Embedding)**.** Given two words u, v, then u is *embedded* in v written $u \sqsubseteq v$ if for $u = a_1 \dots a_n$ there are words $v_0, \dots, v_n$ such that $v = v_0 a_1 v_1 a_2 \dots a_n v_n$.

Reformulating the above definition, a word u is embedded in v if u can be obtained from v by erasing letters. For example, *higman* is embedded in *highmountain*.

Definition 1.3.6 (PCP)**.** Given two finite lists of words $(u_1, \dots, u_n)$ and $(v_1, \dots, v_n)$ the *Post Correspondence Problem* (PCP) is to find a finite index list $(i_1, \dots, i_k)$, $1 \leq i_j \leq n$, so that $u_{i_1} u_{i_2} \dots u_{i_k} = v_{i_1} v_{i_2} \dots v_{i_k}$.

Take for example the two lists (a, b, bb) and (ab, ab, b) over alphabet $\Sigma = \{a, b\}$. Then the index list $(1, 3)$ is a solution to the PCP with common word abb.

Theorem 1.3.7 (Post 1942)**.** PCP is undecidable.

Lemma 1.3.8 (Higman's Lemma 1952)**.** For any infinite sequence of words $u_1, u_2, \dots$ over a finite alphabet there are two words u_k, u_{k+l} such that $u_k \sqsubseteq u_{k+l}$.

Proof. By contradiction. Assume an infinite sequence $u_1, u_2, \ldots$ such that for any two words u_k, u_{k+l} they are not embedded, i.e., $u_k \not\sqsubseteq u_{k+l}$. Furthermore, I assume that the sequence is minimal at any word with respect to length, i.e., considering any u_k, there is no infinite sequence with the above property that shares the words up to u_{k-1} and then continues with a word of smaller length than u_k. Next, the alphabet is finite, so there must be a letter, say a that occurs infinitely often as the first letter of the words of the sequence. The words starting with a form an infinite subsequence $au'_{k_1}, au'_{k_2}, \ldots$ where $u_{k_i} = au'_{k_i}$. This infinite subsequence itself has the non-embedding property, because it is a subsequence of the originial sequence. Now consider the infinite sequence $u_1, u_2, \ldots, u_{k_1-1}, u'_{k_1}, u'_{k_2}, \ldots$. Also this sequence has the non-embedding property: if some $u_i \sqsubseteq u'_{k_j}$ then $u_i \sqsubseteq au'_{k_j}$ contradicting that the starting sequence is non-embedding. But then the constructed sequence contradicts the minimality assumption with respect to length, finishing the proof. □

Definition 1.3.9 (Context-Free Grammar)**.** A context-free grammar $G = (N, T, P, S)$ consists of:

1. a set of non-terminal symbols N
2. a set of terminal symbols T
3. a set P of rules $A \Rightarrow w$ where $A \in N$ and $w \in (N \cup T)^*$
4. a start symbol S where $S \in N$

For rules $A \Rightarrow w_1$, $A \Rightarrow w_2$ we write $A \Rightarrow w_1 \mid w_2$.

Given a context free grammar G and two words $u, v \in (N \cup T)^*$ I write $u \Rightarrow v$ if $u = u_1\, A\, u_2$ and $v = u_1\, w\, u_2$ and there is a rule $A \Rightarrow w$ in G. The *language* generated by G is $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$, where $\Rightarrow^*$ is the reflexive and transitive closure of $\Rightarrow$.

A context free grammar G is in *Chomsky Normal Form* [6] if all rules are if the form $A \Rightarrow B_1\, B_2$ with $B_i \in N$ or $A \Rightarrow w$ with $w \in T^*$. It is said to be in *Greibach Normal Form* [12] if all rules are of the form $A \Rightarrow a\, w$ with $a \in T$ and $w \in N^*$.

1.4 Orderings

An ordering R is a binary relation on some set M. Depending on particular properties such as

(reflexivity)	$\forall\, x \in M\ R(x, x)$
(irreflexivity)	$\forall\, x \in M\ \neg R(x, x)$
(antisymmetry)	$\forall\, x, y \in M\ (R(x, y) \wedge R(y, x) \rightarrow x = y)$
(transitivity)	$\forall\, x, y, z \in M\ (R(x, y) \wedge R(y, z) \rightarrow R(x, z))$
(totality)	$\forall\, x, y \in M\ (R(x, y) \vee R(y, x))$

there are different types of orderings. The relation $=$ is the identity relation on M. The quantifier $\forall$ reads "for all", and the boolean connectives $\wedge$, $\vee$, and $\rightarrow$ read "and", "or", and "implies", respectively. For example, the above formula stating reflexivity $\forall\, x \in M\; R(x,x)$ is a shorthand for "for all $x \in M$ the relation $R(x,x)$ holds".

Actually, the definition of the above properties is informal in the sense that I rely on the meaning of certain symbols such as $\in$ or $\rightarrow$. While the former is assumed to be known from school math, the latter is "explained" above. So, strictly speaking this book is neither self contained, nor overall formal. For the concrete logics developed in subsequent chapters, I will formally define $\rightarrow$ but here, where it is used to state properties needed to eventually define the notion of an ordering, it remains informal. Although it is possible to develop the overall content of this book in a completely formal style, such an approach is typically impossible to read and comprehend. Since this book is about teaching a general framework to eventually generate automated reasoning procedures this would not be the right way to go. In particular, being informal starts already with the use of natural language. In order to support this "mixed" style, examples and exercises deepen the understanding and rule out potential misconceptions.

Now, based on the above defined properties of a relation, the usual notions with respect to orderings are stated below.

Definition 1.4.1 (Orderings)**.** A *partial ordering* $\succeq$ (or simply ordering) on a set M, denoted $(M, \succeq)$, is a reflexive, antisymmetric, and transitive binary relation on M. It is a *total ordering* if it also satisfies the totality property. A *strict ordering* $\succ$ is a transitive and irreflexive binary relation on M. A strict ordering is *well-founded*, if there is no infinite descending chain $m_0 \succ m_1 \succ m_2 \succ \ldots$ where $m_i \in M$.

Given a strict partial order $\succ$ on some set M, its respective partial order $\succeq$ is constructed by taking the transitive closure of $(\succ \cup =)$. If the partial order $\succeq$ extension of some strict partial order $\succ$ is total, then we call also $\succ$ total. As an alternative, a strict partial order $\succ$ is total of it satisfies the strict totality axiom $\forall\, x, y \in M\; (x \neq y \rightarrow (R(x,y) \vee R(y,x)))$. Given some ordering $\succ$ the respective ordering $\prec$ is defined by $a \prec b$ iff $b \succ a$.

Example 1.4.2. The well-known relation $\leq$ on $\mathbb{N}$, where $k \leq l$ if there is a j so that $k + j = l$ for $k, l, j \in \mathbb{N}$, is a total ordering on the naturals. Its strict subrelation $<$ is well-founded on the naturals. However, $<$ is not well-founded on $\mathbb{Z}$.

Definition 1.4.3 (Minimal and Smallest Elements)**.** Given a strict ordering $(M, \succ)$, an element $m \in M$ is called *minimal*, if there is no element $m' \in M$ so that $m \succ m'$. An element $m \in M$ is called *smallest*, if $m' \succ m$ for all $m' \in M$ different from m.

Note the subtle difference between minimal and smallest. There may be several minimal elements in a set M but only one smallest element. Furthermore, in order for an element being smallest in M it needs to be comparable to all other elements from M.

Example 1.4.4. In $\mathbb{N}$ the number 0 is smallest and minimal with respect to $<$. For the set $M = \{q \in \mathbb{Q} \mid q \geq 5\}$ the ordering $<$ on M is total, has the minimal element 5 but is not well-founded.

If $<$ is the ancestor relation on the members of a human family, then $<$ typically will have several minimal elements, the currently youngest children of the family, but no smallest element, as long as there is a couple with more than one child. Furthermore, $<$ is not total, but well-founded.

Well-founded orderings can be combined to more complex well-founded orderings by lexicographic or multiset extensions.

Definition 1.4.5 (Lexicographic and Multi-Set Ordering Extensions)**.** Let $(M_1, \succ_1)$ and $(M_2, \succ_2)$ be two strict orderings. Their *lexicographic combination* $\succ_{\text{lex}} = (\succ_1, \succ_2)$ on $M_1 \times M_2$ is defined as $(m_1, m_2) \succ (m_1', m_2')$ iff $m_1 \succ_1 m_1'$ or $m_1 = m_1'$ and $m_2 \succ_2 m_2'$.

Let $(M, \succ)$ be a strict ordering. The multi-set extension $\succ_{\text{mul}}$ to multi-sets over M is defined by $S_1 \succ_{\text{mul}} S_2$ iff $S_1 \neq S_2$ and $\forall\, m \in M\, [S_2(m) > S_1(m) \rightarrow \exists\, m' \in M\, (m' \succ m \wedge S_1(m') > S_2(m'))]$.

The definition of the lexicographic ordering extensions can be exapanded to n-tuples in the obvious way. So it is also the basis for the standard lexicographic ordering on words as used, e.g., in dictionaries. In this case the M_i are alphabets, say a-z, where $a \prec b \prec \ldots \prec z$. Then according to the above definition *tiger* $\prec$ *tree*.

Example 1.4.6 (Multi Set Ordering)**.** Consider the multiset extension of $(\mathbb{N}, >)$. Then $\{2\} >_{\text{mul}} \{1, 1, 1\}$ because there is no element in $\{1, 1, 1\}$ that is larger than 2. As a border case, $\{2, 1\} >_{\text{mul}} \{2\}$ because there is no element that has more occurrences in $\{2\}$ compared to $\{2, 1\}$. The other way round, 1 has more occurrences in $\{2, 1\}$ than in $\{2\}$ and there is no larger element to compensate for it, so $\{2\} \not\succ_{\text{mul}} \{2, 1\}$.

Proposition 1.4.7 (Properties of Lexicographic and Multi-Set Ordering Extensions)**.** Let $(M, \succ)$, $(M_1, \succ_1)$, and $(M_2, \succ_2)$ be orderings. Then

1. $\succ_{\text{lex}}$ is an ordering on $M_1 \times M_2$.
2. if $(M_1, \succ_1)$ and $(M_2, \succ_2)$ are well-founded so is $\succ_{\text{lex}}$.
3. if $(M_1, \succ_1)$ and $(M_2, \succ_2)$ are total so is $\succ_{\text{lex}}$.
4. $\succ_{\text{mul}}$ is an ordering on multi-sets over M.
5. if $(M, \succ)$ is well-founded so is $\succ_{\text{mul}}$.
6. if $(M, \succ)$ is total so is $\succ_{\text{mul}}$.

The lexicographic ordering on words is not well-founded if words of arbitrary length are considered. Starting from the standard ordering on the alphabet, e.g., the following infinite descending sequence can be constructed: $b \succ ab \succ aab \succ \ldots$. It becomes well-founded if it is lexicographically combined with the length oordering, see Exercise **??**.

Lemma 1.4.8 (König's Lemma)**.** Every finitely branching tree with infinitely many nodes contains an infinite path.

1.5 Induction

More or less all sets of objects in computer science or logic are defined *inductively*. Typically, this is done in a bottom-up way, where starting with some definite set, it is closed under a given set of operations.

Example 1.5.1 (Inductive Sets)**.** In the following, some examples for inductively defined sets are presented:

1. The set of all Sudoku problem states, see Section 1.1, consists of the set of start states $(N; \top; \top)$ for consistent assignments N plus all states that can be derived from the start states by the rules Deduce, Conflict, Backtrack, and Fail. This is a finite set.

2. The set $\mathbb{N}$ of the natural numbers, consists of 0 plus all numbers that can be computed from 0 by adding 1. This is an infinite set.

3. The set of all strings Σ^* over a finite alphabet Σ. All letters of Σ are contained in Σ^* and if u and v are words out of Σ^* so is the word uv, see Section 1.2. This is an infinite set.

All the previous examples have in common that there is an underlying well-founded ordering on the sets induced by the construction. The minimal elements for the Sudoku are the problem states $(N; \top; \top)$, for the natural numbers it is 0 and for the set of strings it is the empty word. Now if we want to prove a property of an inductive set it is sufficient to prove it (i) for the minimal element(s) and (ii) assuming the property for an arbitrary set of elements, to prove that it holds for all elements that can be constructed "in one step" out those elements. This is the principle of *Noetherian Induction*.

Theorem 1.5.2 (Noetherian Induction)**.** Let $(M, \succ)$ be a well-founded ordering, and let Q be a predicate over elements of M. If for all $m \in M$ the implication

if $Q(m')$, for all $m' \in M$ so that $m \succ m'$,	(induction hypothesis)
then $Q(m)$.	(induction step)

is satisfied, then the property $Q(m)$ holds for all $m \in M$.

Proof. Let $X = \{m \in M \mid Q(m) \text{ does not hold}\}$. Suppose, $X \neq \emptyset$. Since $(M, \succ)$ is well-founded, X has a minimal element m_1. Hence for all $m' \in M$ with $m' \prec m_1$ the property $Q(m')$ holds. On the other hand, the implication which is presupposed for this theorem holds in particular also for m_1, hence $Q(m_1)$ must be true so that m_1 cannot be in X - a contradiction. □

Note that although the above implication sounds like a one step proof technique it is actually not. There are two cases. The first case concerns all elements that are minimal with respect to $\prec$ in M and for those the predicate Q needs to hold without any further assumption. The second case is then the induction step showing that by assuming Q for all elements strictly smaller than some m, we can prove it for m.

Now for context free grammars. *** Motivate Further *** Let $G = (N, T, P, S)$ be a context-free grammar (possibly infinite) and let q be a property of T^* (the words over the alphabet T of terminal symbols of G).

q holds for *all* words $w \in L(G)$, whenever one can prove the following two properties:

1. (*base cases*)
   $q(w')$ holds for each $w' \in T^*$ so that $X ::= w'$ is a rule in P.

2. (*step cases*)
   If $X ::= w_0 X_0 w_1 \ldots w_n X_n w_{n+1}$ is in P with $X_i \in N$, $w_i \in T^*$, $n \geq 0$, then for all $w_i' \in L(G, X_i)$, whenever $q(w_i')$ holds for $0 \leq i \leq n$, then also $q(w_0 w_0' w_1 \ldots w_n w_n' w_{n+1})$ holds.

Here $L(G, X_i) \subseteq T^*$ denotes the language generated by the grammar G from the nonterminal X_i.

Let $G = (N, T, P, S)$ be an *unambiguous* (why?) context-free grammar. A function f is well-defined on $L(G)$ (that is, unambiguously defined) whenever these 2 properties are satisfied:

1. (base cases)
   f is well-defined on the words $w' \in T^*$ for each rule $X ::= w'$ in P.

2. (step cases)
   If $X ::= w_0 X_0 w_1 \ldots w_n X_n w_{n+1}$ is a rule in P then $f(w_0 w_0' w_1 \ldots w_n w_n' w_{n+1})$ is well-defined, assuming that each of the $f(w_i')$ is well-defined.

1.6 Rewrite Systems

The final ingredient to actually start the journey through different logical systems is rewrite systems. Here I define the needed computer science background for defining algorithms in the form of rule sets. In Section 1.1 the rewrite rules Deduce, Conflict, Backtrack, and Fail defined an algorithm for solving 4×4 Sudokus. The rules operate on the set of Sudoku problem states, starting with a set of initial states $(N; \top; \top)$ and finishing either in a solution state $(N; D; \top)$

or a fail state $(N; \top; \bot)$. The latter are called *normal forms* (see below) with respect to the above rules, because no more rule is applicable to solution state $(N; D; \top)$ or a fail state $(N; \top; \bot)$.

Definition 1.6.1 (Rewrite System)**.** A *rewrite system* is a pair $(M, \to)$, where M is a non-empty set and $\to \subseteq M \times M$ is a binary relation on M. Figure 1.4 defines the needed notions for $\to$.

$\to^0$	$= \{ (a, a) \mid a \in M \}$	*identity*
$\to^{i+1}$	$= \to^i \circ \to$	$i+1$*-fold composition*
$\to^+$	$= \bigcup_{i>0} \to^i$	*transitive closure*
$\to^*$	$= \bigcup_{i\geq 0} \to^i = \to^+ \cup \to^0$	*reflexive transitive closure*
$\to^=$	$= \to \cup \to^0$	*reflexive closure*
$\to^{-1}$	$= \leftarrow = \{ (b, c) \mid c \to b \}$	*inverse*
$\leftrightarrow$	$= \to \cup \leftarrow$	*symmetric closure*
$\leftrightarrow^+$	$= (\leftrightarrow)^+$	*transitive symmetric closure*
$\leftrightarrow^*$	$= (\leftrightarrow)^*$	*refl. trans. symmetric closure*

Figure 1.4: Notation on $\to$

For a rewrite system $(M, \to)$ consider a sequence of elements a_i that are pairwise connected by the symmetric closure, i.e., $a_1 \leftrightarrow a_2 \leftrightarrow a_3 \ldots \leftrightarrow a_n$. We say that a_i is a *peak* in such a sequence, if actually $a_{i-1} \leftarrow a_i \to a_{i+1}$.

Actually, in Definition 1.6.1 I overload the symbol $\to$ that has already denoted logical implication, see Section 1.4, with a rewrite relation. This overloading will remain throughout this book. The rule symbol $\Rightarrow$ is only used on the meta level in this book, e.g., to define the Sudoku algorithm on problem states, Section 1.1. Nevertheless, this meta rule systems are also rewrite systems in the above sense. The rewrite symbol $\to$ is used on the formula level inside a problem state. This will become clear when I turn to more complex logics starting from Chapter 2.

Definition 1.6.2 (Reducible)**.** Let $(M, \to)$ be a rewrite system. An element $a \in M$ is *reducible*, if there is a $b \in M$ so that $a \to b$. An element $a \in M$ is *in normal form (irreducible)*, if it is not reducible. An element $c \in M$ is a *normal form* of b, if $b \to^* c$ and c is in normal form, notated $c = b{\downarrow}$ (if the normal form of b is unique). Two elements b and c are *joinable*, if there is an a so that $b \to^* a \ {}^*\!\leftarrow c$, notated $b \downarrow c$.

Definition 1.6.3 (Properties of $\to$)**.** A relation $\to$ is called

Church-Rosser	if $b \leftrightarrow^* c$ implies $b \downarrow c$
confluent	if $b \ {}^*\!\leftarrow a \to^* c$ implies $b \downarrow c$
locally confluent	if $b \leftarrow a \to c$ implies $b \downarrow c$
terminating	if there is no infinite descending chain $b_0 \to b_1 \ldots$
normalizing	if every $b \in A$ has a normal form
convergent	if it is confluent and terminating

Lemma 1.6.4. If $\rightarrow$ is terminating, then it is normalizing.

The reverse implication of Lemma 1.6.4 does not hold. Assuming this is a frequent mistake. Consider $M = \{a, b, c\}$ and the relation $a \rightarrow b$, $b \rightarrow a$, and $b \rightarrow c$. Then $(M, \rightarrow)$ is obviously not terminating, because we can cycle between a and b. However, $(M, \rightarrow)$ is normalizing. The normal form is c for all elements of M. Similarly, there are rewrite systems that are locally confluent, but not confluent, see Figure **??**. *** to be done *** In the context of termination the property holds, see Lemma 1.6.6.

Theorem 1.6.5. The following properties are equivalent for any rewrite system $(S, \rightarrow)$:

(i) $\rightarrow$ has the Church-Rosser property.

(ii) $\rightarrow$ is confluent.

Proof. (i) $\Rightarrow$ (ii): trivial.

(ii) $\Rightarrow$ (i): by induction on the number of peaks in the derivation $b \leftrightarrow^* c$. □

Lemma 1.6.6 (Newman's Lemma [**?**]: Confluence versus Local Confluence)**.** Let $(M, \rightarrow)$ be a terminating rewrite system. Then the following properties are equivalent:

(i) $\rightarrow$ is confluent

(ii) $\rightarrow$ is locally confluent

Proof. (i) $\Rightarrow$ (ii): trivial.

(ii) $\Rightarrow$ (i): Since $\rightarrow$ is terminating, it is a well-founded ordering (see Exercise **??**). This justifies a proof by Noetherian induction where the property $Q(a)$ is "a is confluent". Applying Noetherian induction, confluence holds for all $a' \in M$ with $a \rightarrow^+ a'$ and needs to be shown for a. Consider the confluence property for a: $b \,{}^*\!\leftarrow a \rightarrow^* c$. If $b = a$ or $c = a$ the proof is done. For otherwise, the situation can be expanded to $b \,{}^*\!\leftarrow b' \leftarrow a \rightarrow c' \rightarrow^* c$. By local confluence there is an a' with $b' \rightarrow^* a' \,{}^*\!\leftarrow c'$. Now a', b, c are strictly smaller than a, they are confluent and hence can be rewritten so a single a'', finishing the proof. □

Lemma 1.6.7. If $\rightarrow$ is confluent, then every element has at most one normal form.

Proof. Suppose that some element $a \in A$ has normal forms b and c, then $b \,{}^*\!\leftarrow a \rightarrow^* c$. If $\rightarrow$ is confluent, then $b \rightarrow^* d \,{}^*\!\leftarrow c$ for some $d \in A$. Since b and c are normal forms, both derivations must be empty, hence $b \rightarrow^0 d \,{}^0\!\leftarrow c$, so b, c, and d must be identical. □

Corollary 1.6.8. If $\rightarrow$ is normalizing and confluent, then every element b has a unique normal form.

Proposition 1.6.9. If $\rightarrow$ is normalizing and confluent, then $b \leftrightarrow^* c$ if and only if $b{\downarrow} = c{\downarrow}$.

Proof. Either using Theorem 1.6.5 or directly by induction on the length of the derivation of $b \leftrightarrow^* c$. □

Historic and Bibliographic Remarks

For context free languages see [2].

Chapter 2

Propositional Logic

2.1 Syntax

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

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

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

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

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

As a notational convention we assume that $\neg$ binds strongest and we omit outermost parenthesis. So $\neg P \vee Q$ is actually a shorthand for $((\neg P) \vee Q)$. For all other logical connectives we will explicitly put parenthesis when needed.

From the semantics we will see that $\wedge$ and $\vee$ are associative and commutative. Therefore instead of $((P \wedge Q) \wedge R)$ we simply write $P \wedge Q \wedge R$.

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

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

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

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

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

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

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

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

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

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

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

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

2.2 Semantics

In *classical logic* there are two truth values "true" and "false" which we shall denote, respectively, by 1 and 0. There are *many-valued logics* [21] having more than two truth values and in fact, as we will see later on, for the definition of some propositional logic calculi, we will need an implicit third truth value called "undefined".

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

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

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

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

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

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

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

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

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

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

Proposition 2.2.4. $\phi \models \psi$ iff $\models \phi \rightarrow \psi$

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

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

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

Note that the formulas $\phi \wedge \psi$ and $\psi \wedge \phi$ are equivalent. Nevertheless, recalling the problem state definition for Sudokus in Section 1.1 the two states $(N; f(2,3) = 1 \wedge f(2,4) = 4; \top)$ and $(N; f(2,4) = 4 \wedge f(2,3) = 1; \top)$ are significantly different. For example, it can be that the first state can lead to a solution by the rules of the algorithm where the latter cannot, because the latter implicitly means that the square $(2,4)$ has already

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

Figure 2.1: Valid Propositional Equivalences

been checked for all values smaller than 4. This reveals the important point that arguing by logical equivalence in the context of a rule set manipulating formulas can lead to wrong results.

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

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

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

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

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

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

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

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

□

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

Proof. By induction on the length of p. □

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

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

2.3 Abstract Properties of Calculi

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

	Validity	Satisfiability
Sound	Whenever the calculus outputs a proof the formula is valid.	Whenever the calculus outputs a model the formula has a model.
Complete	If the formula is valid the calculus outputs a proof.	If the formula is satisfiable, the calculus outputs a model.
Strongly Complete	For any proof of the formula, there is a sequence of rule applications that generates this proof.	For any model of the formula, there is a sequence of rule applications that generates this model.

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

Note that soundness and completeness are not closely related to termination. A sound and complete (strongly) complete calculus needs not to be terminating. For example, while investigating validity of an invalid formula, a sound and complete calculus for validity may not terminate.

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

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

A sound and terminating procedure needs not to be complete. It can simply terminate, "giving up", without producing a proof (model).

2.4 Truth Tables

The first calculus I consider are truth tables. For example, consider proving validity of the formula $\phi = (A \wedge B) \rightarrow A$. According to Definition 2.2.2 this is the case if actually for all valuations $\mathcal{A}$ over $\Sigma = \{A, B\}$ we have $\mathcal{A}(\phi) = 1$. The extension of $\mathcal{A}$ to formulas is defined inductively over the connectives, so if the result of $\mathcal{A}$ on the arguments of a connective is known, it can be straightforwardly computed for the overall formula. That's the idea behind truth tables. We simply make all valuations $\mathcal{A}$ on Σ explicit and then extend it connective by connective bottom-up to the overall formula. Stated otherwise, in order to establish the truth value for a formula ϕ we establish it subformula by subformula of ϕ according to $\leq$. If $p, q \in \mathrm{pos}(\phi)$ and $p \leq q$ then we first compute the truth value for $\phi|_q$. The truth table for $(P \wedge Q) \rightarrow P$ is then depicted in Figure 2.2

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

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

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

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

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

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

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

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

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

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

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

Figure 2.4: α- and β-Formulas

ous. All procedures I will introduce in the sequel, automatically detect this (and further) specific structures of a formula and use it to speed up the reasoning process.

2.5 Semantic Tableaux

Like resolution, semantic tableaux were developed in the sixties, independently by Lis [14] and Smullyan [19] on the basis of work by Gentzen in the 30s [11] and of Beth [3] in the 50s. For an at that time state of the art overview consider Fitting's book [10].

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

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

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

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

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

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

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

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

α-Expansion $N \uplus \{(\phi_1, \ldots, \psi, \ldots, \phi_n)\} \Rightarrow_{\mathrm{T}} N \uplus \{(\phi_1, \ldots, \psi, \ldots, \phi_n, \psi_1, \psi_2)\}$

provided ψ is an open α-formula, ψ_1, ψ_2 its direct descendants and the sequence is not closed.

β-Expansion $N \uplus \{(\phi_1, \ldots, \psi, \ldots, \phi_n)\} \Rightarrow_{\mathrm{T}} N \uplus \{(\phi_1, \ldots, \psi, \ldots, \phi_n, \psi_1)\} \uplus \{(\phi_1, \ldots, \psi, \ldots, \phi_n, \psi_2)\}$

provided ψ is an open β-formula, ψ_1, ψ_2 its direct descendants and the sequence is not closed.

Consider the question of validity of the formula $(P \wedge \neg(Q \vee \neg R)) \rightarrow (Q \wedge R)$. Applying the tableau rules generates the following derivation:

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

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

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

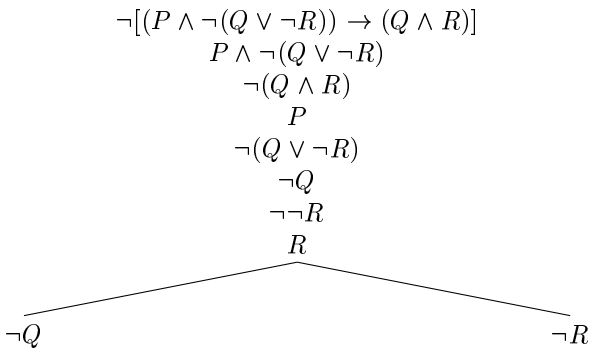

Figure 2.5: A Tableau for $(P \wedge \neg(Q \vee \neg R)) \rightarrow (Q \wedge R)$

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

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

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

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

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

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

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

$(n,k,l) >_{\text{lex}} \mu(\psi_2)$ and $(n,k,l) >_{\text{lex}} (0,0,0)$, the former because the ψ_i are subformulas and the latter because $l \neq 0$. This proves the case.

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

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

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

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

Proof. A consequence of the proof of Theorem 2.5.6 □

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

2.6 Normal Forms

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

2.6.1 Conjunctive and Disjunctive Normal Forms

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

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

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

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

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

On the other hand, checking the unsatisfiability of CNF formulas or the validity of DNF formulas is coNP-complete. For any propositional formula ϕ there is an equivalent formula in CNF and DNF and I will prove this below by actually providing an effective procedure for the transformation. However, also because of the above comment on validity and satisfiability checking for CNF and DNF formulas, respectively, the transformation is costly. In general, a CNF or DNF of a formula ϕ is exponentially larger than ϕ as long as the normal forms need to be logically equivalent. If this is not needed, then by the introduction of fresh propositional variables, CNF or DNF normal forms for ϕ can be computed in linear time in the size of ϕ. More concretely, given a formula ϕ instead of checking validity the unsatisfiability of $\neg\phi$ can be considered. Then the linear time CNF normal form algorithm (see Section **??**) computes a satisfiability preserving formula, i.e., the linear time CNF of $\neg\phi$ is unsatisfiable iff $\neg\phi$ is.

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

Figure 2.6: Basic CNF/DNF Transformation Rules

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

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

2.6.2 Basic CNF/DNF Transformation

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

Algorithm 2: bcnf(ϕ)

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

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

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

$$
\begin{aligned}
&\neg((P \vee Q) \leftrightarrow (P \to (Q \wedge \top)))\\
&\Rightarrow^{\text{Step 1}}_{\text{BCNF}} \neg([(P \vee Q) \to (P \to (Q \wedge \top))] \wedge [(P \to (Q \wedge \top)) \to (P \vee Q)])\\
&\Rightarrow^{\text{Step 2}}_{\text{BCNF}} \neg([\neg(P \vee Q) \vee (P \to (Q \wedge \top))] \wedge [(P \to (Q \wedge \top)) \to (P \vee Q)])\\
&\Rightarrow^{\text{Step 2}}_{\text{BCNF}} \neg([\neg(P \vee Q) \vee (P \to (Q \wedge \top))] \wedge [\neg(P \to (Q \wedge \top)) \vee (P \vee Q)])\\
&\Rightarrow^{\text{Step 2}}_{\text{BCNF}} \neg([\neg(P \vee Q) \vee (P \to (Q \wedge \top))] \wedge [\neg(\neg P \vee (Q \wedge \top)) \vee (P \vee Q)])\\
&\Rightarrow^{\text{Step 2}}_{\text{BCNF}} \neg([\neg(P \vee Q) \vee (\neg P \vee (Q \wedge \top))] \wedge [\neg(\neg P \vee (Q \wedge \top)) \vee (P \vee Q)])\\
&\Rightarrow^{\text{Step 3}}_{\text{BCNF}} \neg([\neg(P \vee Q) \vee (\neg P \vee Q)] \wedge [\neg(\neg P \vee Q) \vee (P \vee Q)])\\
&\Rightarrow^{\text{Step 4}}_{\text{BCNF}} \neg([(\neg P \wedge \neg Q) \vee (\neg P \vee Q)] \wedge [\neg(\neg P \vee Q) \vee (P \vee Q)])\\
&\Rightarrow^{\text{Step 4}}_{\text{BCNF}} \neg([(\neg P \wedge \neg Q) \vee (\neg P \vee Q)] \wedge [(\neg\neg P \wedge \neg Q) \vee (P \vee Q)])\\
&\Rightarrow^{\text{Step 4}}_{\text{BCNF}} \neg([(\neg P \wedge \neg Q) \vee (\neg P \vee Q)] \wedge [(\neg\neg P \wedge \neg Q) \vee (P \vee Q)])\\
&\Rightarrow^{*,\text{Step 4}}_{\text{BCNF}} [(\neg\neg P \vee \neg\neg Q) \wedge (\neg\neg P \wedge \neg Q)] \vee [(\neg\neg\neg P \vee \neg\neg Q) \wedge (\neg P \wedge \neg Q)]\\
&\Rightarrow^{*,\text{Step 4}}_{\text{BCNF}} [(P \vee Q) \wedge (P \wedge \neg Q)] \vee [(\neg P \vee Q) \wedge (\neg P \wedge \neg Q)]\\
&\Rightarrow^{*,\text{Step 5}}_{\text{BCNF}} (P \vee Q \vee \neg P \vee Q) \wedge (P \vee Q \vee \neg P) \wedge (P \vee Q \vee \neg Q) \wedge (P \vee \neg P \vee\\
&Q) \wedge (P \vee \neg P) \wedge (P \vee \neg Q) \wedge (\neg Q \vee \neg P \vee Q) \wedge (\neg Q \vee \neg P) \wedge (\neg Q \vee \neg Q)
\end{aligned}
$$

Figure 2.7: Example Basic CNF Transformation

Example 2.6.3. Consider the formula $\neg((P \vee Q) \leftrightarrow (P \to (Q \wedge \top)))$ and the application of $\Rightarrow_{\text{BCNF}}$ depicted in Figure 2.7. Already for this simple formula the CNF transformation via $\Rightarrow_{\text{BCNF}}$ becomes quite messy. Note that the CNF result in Figure 2.7 is still highly redundant. If I remove all disjunctions that are trivially true, because they contain a propositional literal and its negation, the result becomes

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

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

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

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 $\neq Q$ *subsumes* the formulas $P \vee \neg Q$ and $\neg Q \vee \neg P$ in this sense. The notion of subsumption will be discussed in detail for clauses in Section 2.7.

So it is eventually equivalent to

$$\neg Q.$$

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

After proving $\Rightarrow_{\text{BCNF}}$ correct and terminating, in the succeeding section I will present an algorithm $\Rightarrow_{\text{ACNF}}$ that actually generates $\neg Q$ out of $\neg((P \vee Q) \leftrightarrow (P \to (Q \wedge \top)))$ and does this without generating the mess of formulas $\Rightarrow_{\text{BCNF}}$

does. Please recall that the above rules apply modulo commutativity of $\vee$, $\wedge$, e.g., the rule ElimTB1 is both applicable to the formulas $\phi \wedge \top$ and $\top \wedge \phi$.

Figure 2.1 contains more potential for simplification. For example, the idempotency equivalences $(\phi \wedge \phi) \leftrightarrow \phi$, $(\phi \vee \phi) \leftrightarrow \phi$ can be turned into simplification rules by applying them left to right. However, the way they are stated they can only be applied in case of identical subformulas. The formula $(P \vee Q) \wedge (Q \vee P)$ does this way not reduce to $(Q \vee P)$. A solution is to consider identity modulo commutativity. But then identity modulo commutativity and associativity (AC) as in $((P \vee Q) \vee R) \wedge (Q \vee (R \vee P)$ is still not detected. On the other hand, in practice, checking identity modulo AC is often too expensive. An elegant way out of this situation is to implement AC connectives like $\vee$ or $\wedge$ with flexible arity, to normalize nested occurrences of the connectives, and finally to sort the arguments using some total ordering. Applying this to $((P \vee Q) \vee R) \wedge (Q \vee (R \vee P)$ with ordering $R > P > Q$ the result is $(Q \vee P \vee R) \wedge (Q \vee P \vee R)$. Now complete AC simplification is back at the cost of checking for identical subformulas. Note that in an appropriate implementation, the normalization and ordering process is only done once at the start and then normalization and argument ordering is kept as an invariant.

2.6.3 Advanced CNF Transformation

The simple algorithm for CNF transformation Algorithm 2 can be improved in various ways: (i) more aggressive formula simplification, (ii) renaming, (iii) polarity dependant transformations. The before studied Example 2.6.3 serves already as a nice motivation for (i) and (iii). Firstly, removing $\top$ from the formula $\neg((P \vee Q) \leftrightarrow (P \rightarrow (Q \wedge \top)))$ 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 $(\phi \leftrightarrow \psi) \leftrightarrow (\phi \wedge \psi) \vee (\neg\phi \wedge \neg\psi)$ 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 do $\psi[(\phi_1 \wedge \phi_2) \vee (\neg\phi_1 \wedge \neg\phi_2)]_p$ and if $\mathrm{pol}(\psi, p) = 1$ do $\psi[(\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)]_p$

Item (ii) can be motivated by a formula

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

where Algorithm 2 generates a CNF with 2^n clauses out of this formula. The way out of this problem is the introduction of additional fresh propositional variables that *rename* subformulas. The price to pay is that a renamed formula is not equivalent to the original formula due to the extra propositional variables, but satisfiability preserving. A renamed formula for the above formula is

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

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

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

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

Proof. Exercise. □

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

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

For renaming there are several choices which subformula to choose. Obviously, since a formula has only linearly many subformulas, renaming every subformula works [20, 17]. Basically this is what I show below. In the following section a renaming variant is introduced that produces smallest CNFs.

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

Actually, the rule SimpleRenaming does not provide an effective way to compute the set $\{p_1, \ldots, p_n\}$ of positions in ϕ to be renamed. Where are several choices. Following Plaisted and Greenbaum [17], 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 $P \wedge (P \to \phi)$ 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, let's call it the set of obvious positions, is still preventing the explosion and given by the rules: (i) if $\phi|_p$ is an equivalence and there is a position $q < p$ such that $\phi|_q$ is either an equivalence or disjunctive in ϕ then p is an obvious position (ii) if $\phi|_{pq}$ is a conjunctive formula in ϕ, $\phi|_p$ is a disjunctive formula in ϕ and for all positions r with $p < r < pq$ the formula $\phi|_r$ is not a conjunctive formula then pq is an obvious position. A formula $\phi|_p$ is conjunctive in ϕ if $\phi|_p$ is a conjunction and $\mathrm{pol}(\phi, p) \in \{0, 1\}$ or $\phi|_p$ is a disjunction or implication and $\mathrm{pol}(\phi, p) \in \{0, -1\}$. Analogously, a formula $\phi|_p$ is disjunctive in ϕ if $\phi|_p$ is a disjunction or implication and $\mathrm{pol}(\phi, p) \in \{0, 1\}$ or $\phi|_p$ is a conjunction and $\mathrm{pol}(\phi, p) \in \{0, -1\}$.

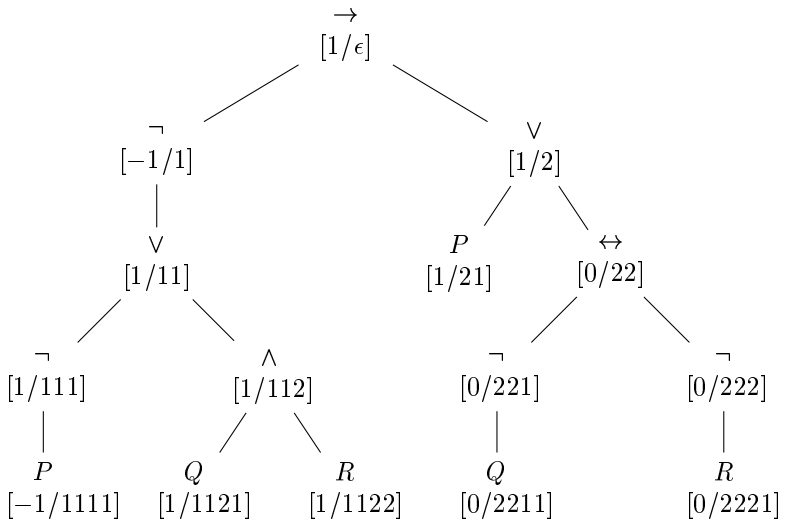

Figure 2.8: Tree representation of $[\neg(\neg P \vee (Q \wedge R))] \rightarrow [P \vee (\neg Q \leftrightarrow \neg R)]$ where each node is annotated with its [polarity/position].

Consider as an example the formula

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

. Its tree representation as well as the polarity and position of each node is shown in Figure 2.8.

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

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

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

Proposition 2.6.5 (Renaming Preservers Models)**.** 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., $\phi' = \phi[P]_p \wedge \text{def}(\phi, p, P)$. There are three cases depending on the polarity. (i) if $\text{pol}(\phi, p) = 1$ then $\phi' = \phi[P]_p \wedge P \rightarrow \phi|_p$. 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 $\text{pol}(\phi, p) = 1$ also $\mathcal{A}(\phi) = 1$ by Lemma 2.2.7. (ii) if $\text{pol}(\phi, p) = -1$ the case is symmetric to the previous one. Finally, (iii) if $\text{pol}(\phi, p) = 0$ for any $\mathcal{A}$ satisfying ϕ' it holds $\mathcal{A}(\phi|_p) = \mathcal{A}(P)$ and hence $\mathcal{A}(\phi) = 1$. □

Algorithm 3: acnf(ϕ)

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

```
1 whilerule (ElimTB1(φ),...,ElimTB6(φ)) do ;
2 SimpleRenaming(φ) on obvious positions;
3 whilerule (ElimEquiv1(φ),ElimEquiv2(φ)) do ;
4 whilerule (ElimImp(φ)) do ;
5 whilerule (PushNeg1(φ),...,PushNeg3(φ)) do ;
6 whilerule (PushDisj(φ)) do ;
7 return φ;
```

$$
\begin{aligned}
&\neg((P \vee Q) \leftrightarrow (P \rightarrow (Q \wedge \top))) \\
&\Rightarrow^{\text{Step 1}}_{\text{ACNF}} \neg((P \vee Q) \leftrightarrow (P \rightarrow Q)) \\
&\Rightarrow^{\text{Step 3}}_{\text{ACNF}} \neg(((P \vee Q) \wedge (P \rightarrow Q)) \vee (\neg(P \vee Q) \wedge \neg(P \rightarrow Q))) \\
&\Rightarrow^{*,\text{Step 4}}_{\text{ACNF}} \neg(((P \vee Q) \wedge (\neg P \vee Q)) \vee (\neg(P \vee Q) \wedge \neg(\neg P \vee Q))) \\
&\Rightarrow^{*,\text{Step 5}}_{\text{ACNF}} ((\neg P \wedge \neg Q) \vee (P \wedge \neg Q)) \wedge ((P \vee Q) \vee (\neg P \vee Q)) \\
&\Rightarrow^{*,\text{Step 6}}_{\text{ACNF}} (\neg P \vee P) \wedge (\neg P \vee \neg Q) \wedge (\neg Q \vee P) \wedge (\neg Q \vee \neg Q) \wedge (P \vee Q \vee \neg P \vee Q)
\end{aligned}
$$

Figure 2.9: Example Advanced CNF Transformation

2.6.4 Computing Small CNFs

In the previous chapter obvious positions are a suggestion for smaller CNFs with respect to the renaming positions suggested by Plaisted and Greenbaum. In this section I develop a set of renaming posisions 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 [5, 16]. 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 [5]. The below function ac computes the number of clauses generated by the algorithm bcnf, as long as the formula does not contain $\top$ or $\bot$.

C

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

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

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

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

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

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

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

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

The obvious problem with this condition is that the function ac cannot be efficiently computed in general, for it grows exponentially in the size of the input formula. Moreover, a straightforward, naive top-down implementation of ac following the above table results in an algorithm with exponential time complexity, due to the duplication of recursive calls. The exponential complexity can be avoided using a dynamic programming idea: simply store intermediate results for subformulas. Nevertheless, because ac grows exponentially, computing ac requires arbitrary precision integer arithmetic. It turns out that this can

hardly be afforded in practice. The rest of this section is therefore concerned with a solution to this problem, i.e., I show that it is not necessary to compute ac at all for deciding the above inequation.

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

p	$\psi\|_q$	a_p^ψ	b_p^ψ
$q.i$	$\phi_1 \wedge \phi_2$	a_q^ψ	$b_q^\psi \prod_{j\neq i} \mathrm{bc}(\phi_j)$
$q.i$	$\phi_1 \vee \phi_2$	$a_q^\psi \prod_{j\neq i} \mathrm{ac}(\phi_j)$	b_q^ψ
$q.1$	$\phi_1 \rightarrow \phi_2$	b_q^ψ	$a_q^\psi\, \mathrm{ac}(\phi_2)$
$q.2$	$\phi_1 \rightarrow \phi_2$	$a_q^\psi\, \mathrm{bc}(\phi_1)$	b_q^ψ
$q.1$	$\phi_1 \leftrightarrow \phi_2$	$a_q^\psi\, \mathrm{bc}(\phi_2) + b_q^\psi\, \mathrm{ac}(\phi_2)$	$a_q^\psi\, \mathrm{ac}(\phi_2) + b_q^\psi\, \mathrm{bc}(\phi_2)$
$q.2$	$\phi_1 \leftrightarrow \phi_2$	$a_q^\psi\, \mathrm{bc}(\phi_1) + b_q^\psi\, \mathrm{ac}(\phi_1)$	$a_q^\psi\, \mathrm{ac}(\phi_1) + b_q^\psi\, \mathrm{bc}(\phi_1)$
$q.1$	$\neg\phi_1$	b_q^ψ	a_q^ψ
ϵ	ψ	1	0

Table 2.1: Calculating the Coefficients

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

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

$$a_p^\psi\, \mathrm{ac}(\phi) + b_p^\psi\, \mathrm{bc}(\phi) \geq a_p^\psi + b_p^\psi + \mathrm{ac}(\mathrm{def}(\psi, p, P))$$

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

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

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

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

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

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

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

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

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

$$[P_1 \vee (Q_1 \wedge Q_2)] \wedge [P_2 \vee (Q_1 \wedge Q_2)] \wedge \ldots [P_n \vee (Q_1 \wedge Q_2)]$$

where our renaming criterion does not apply. But now a simultaneous renaming of all occurrences $(Q_1 \wedge Q_2)$ may pay off. It results in $n + 2$ clauses whereas the standard CNF translation yields $2n$ clauses. Hence, it is useful to search for multiple occurrences of the same subformula. The problem here is to find an appropriate "equality" or "instance" relation between subformulae. In our example syntactic equality was sufficient to detect all such occurrences. In general, a matching process – probably with respect to the commutativity, associativity of some logical operators or even logical implication – may be needed to obtain a suitable renaming result. So we run here into a tradeoff between compact CNFs and computational complexity to achieve these CNFs.

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

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

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

Table 2.2: The Boolean Conditions for ac

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

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

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

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

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

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

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

Table 2.3: The Boolean Conditions for bc

a linear time processing phase.

Applying Algorithm 4 to the formula $\neg((P \vee Q) \leftrightarrow (P \rightarrow (Q \wedge \top)))$ of Example 2.6.3 results in the transformation depicted in Figure 2.10. Looking at the result it is already very close to $\neg Q$, as it contains the clause $(\neg Q \vee \neg Q)$. Removing duplicate literals in clauses and removing clauses containing complementary literals from the result yields

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

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

2.7 Propositional Resolution

A *calculus* is a set of *inference* and *reduction* rules for a given logic (here $\mathrm{PROP}(\Sigma)$). We only consider calculi operating on a set of clauses N. Inference rules *add* new clauses to N whereas reduction rules *remove* clauses from

Algorithm 4: ocnf(ϕ)

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

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

$$
\begin{aligned}
&\neg((P \vee Q) \leftrightarrow (P \rightarrow (Q \wedge \top))) \\
&\Rightarrow^{\text{Step 1}}_{\text{OCNF}} \ \neg([(P \vee Q) \leftrightarrow (P \rightarrow Q)]) \\
&\Rightarrow^{\text{Step 3}}_{\text{OCNF}} \ \neg([(P \vee Q) \wedge (P \rightarrow Q)] \vee [\neg(P \vee Q) \wedge \neg(P \rightarrow Q)]) \\
&\Rightarrow^{\text{Step 2}}_{\text{OCNF}} \ \neg([(P \vee Q) \wedge (\neg P \vee Q)] \vee [\neg(P \vee Q) \wedge \neg(\neg P \vee Q)]) \\
&\Rightarrow^{*,\text{Step 3}}_{\text{OCNF}} \ (\neg[(P \vee Q) \wedge (\neg P \vee Q)] \wedge \neg[\neg(P \vee Q) \wedge \neg(\neg P \vee Q)]) \\
&\Rightarrow^{*,\text{Step 3}}_{\text{OCNF}} \ [\neg(P \vee Q) \vee \neg(\neg P \vee Q)] \wedge [(P \vee Q) \vee (\neg P \vee Q)] \\
&\Rightarrow^{*,\text{Step 3}}_{\text{OCNF}} \ [(\neg P \wedge \neg Q) \vee (P \wedge \neg Q)] \wedge [(P \vee Q) \vee (\neg P \vee Q)] \\
&\Rightarrow^{*,\text{Step 4}}_{\text{OCNF}} \ [(\neg P \vee P) \wedge (\neg P \vee \neg Q) \wedge (\neg Q \vee P) \wedge (\neg Q \vee \neg Q)] \wedge [P \vee Q \vee \neg P \vee Q]
\end{aligned}
$$

Figure 2.10: Example Optimized CNF Transformation

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

Table 2.4: The Boolean Conditions for a

N or *replace* clauses by "simpler" ones.

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

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

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

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

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

Theorem 2.7.1. The resolution calculus is sound and complete:
N is unsatisfiable iff $N \Rightarrow_{\mathrm{RES}}^* \{\bot\}$

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

($\Rightarrow$) The traditional method of proving resolution completeness are *semantic trees*. A *semantic tree* is a binary tree where the edges a labeled with literals such that: (i) edges of children of the same parent are labeled with L and $\neg L$, and (ii) any node has either no or two children, and (iii) for any path from the root to a leave, each propositional variable occurs at most once. Therefore, each path corresponds to a partial valuation. Now for an unsatisfiable clause

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

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

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

and the corresponding semantic tree ...

The reduction rules are

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

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

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

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

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

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

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

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

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

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

2.8 Propositional Superposition

Superposition was originally developed for first-order logic [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 for inference guidance, and (iv) a *saturation* concept.

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

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

1. If $Q \prec_L P$ then $D \prec_C C$.

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

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

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

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

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

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

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

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

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

Proposition 2.8.6. Some properties of the partial model construction.

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

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

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

Superposition Left $(N \uplus \{C_1 \vee P, C_2 \vee \neg P\}) \Rightarrow_{\text{SUP}} (N \cup \{C_1 \vee P, C_2 \vee \neg P\} \cup \{C_1 \vee C_2\})$

where (i) P is strictly maximal in $C_1 \vee P$ (ii) no literal in $C_1 \vee P$ is selected (iii) $\neg P$ is maximal or selected in $C_2 \vee \neg P$

Factoring $(N \uplus \{C \vee P \vee P\}) \Rightarrow_{\text{SUP}} (N \cup \{C \vee P \vee P\} \cup \{C \vee P\})$

where (i) P is maximal in $C \vee P \vee P$ (ii) no literal is selected in $C \vee P \vee P$

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

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

Examples for specific redundancy rules that can be efficiently decided are

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

provided $C_1 \subset C_2$

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

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

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

where $C_1 \subseteq C_2$

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

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

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

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

I distinguish the case L is a positive and no literal selected in $C \vee L$ or L is a negative literal. Firstly, assume L is positive, i.e., $L = P$ for some propositional variable P. Now if P is strictly maximal in $C \vee P$ then actually $\delta_{C \vee P} = \{P\}$ and hence $N_{\mathcal{I}} \models C \vee P$, a contradiction. So P is not strictly maximal. But then actually $C \vee P$ has the form $C'_1 \vee P \vee P$ and Factoring derives $C'_1 \vee P$ where $(C'_1 \vee P) \prec (C'_1 \vee P \vee P)$. Now $C'_1 \vee P$ is not redundant, strictly smaller than $C \vee L$, we have $C'_1 \vee P \in N$ and $N_{\mathcal{I}} \not\models C'_1 \vee P$, a contradiction against the choice that $C \vee L$ is minimal.

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

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

2.9 Davis Putnam Logemann Loveland Procedure (DPLL)

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

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

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

Here are the rules

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

Figure 2.11: The DPLL Calculus

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

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

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

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

Induction step: $n \to n+1$. Assume $(M'; N)$ is reached via n rule applications. Then by the use of the induction hypothesis it holds for all $1 \leq i < m$ that $N, M_1, \dots, L_i^{\top} \models M_{i+1}$ so that it remains to be shown that $N, M_1, \dots, L_m^{\top} \models M_{m+1}$

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

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

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

□

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Propagate:

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

Decide:

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

Backtrack:

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

□

2.10 Conflict Driven Clause Learning (CDCL)

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

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

A literal L is of *level* k with respect to a problem state $(M; N; U; j; C)$ if L or $\neg L$ occurs in M and the first decision literal left from L ($\neg L$)in M is annotated with k or if there is no such literal 0. A clause D is of *level* k with respect to a

problem state $(M; N; U; j; C)$ if k is the maximal level of a literal in D. Recall C is a non-empty clause or $\top$ or $\bot$. The rules are

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

Decide $(M; N; U; k; \top) \Rightarrow_{\text{CDCL}} (M\, L^{k+1}; N; U; k+1; \top)$
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 $(M\, L^{C \vee L}; N; U; k; D) \Rightarrow_{\text{CDCL}} (M; N; U; k; D)$
provided $D \notin \{\top, \bot\}$ and $\neg L$ does not occur in D

Resolve $(M\, L^{C \vee L}; N; U; k; D \vee \neg L) \Rightarrow_{\text{CDCL}} (M; N; U; k; D \vee C)$
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 2.12. In the special case of a unit clause L, the rule Propagate actually annotates the literal L with itself.

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