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

C

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

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

T

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

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

Fail $\quad (N; D; \bot) \quad \Rightarrow \quad (N; \top; \bot)$
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 non-deterministically". 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

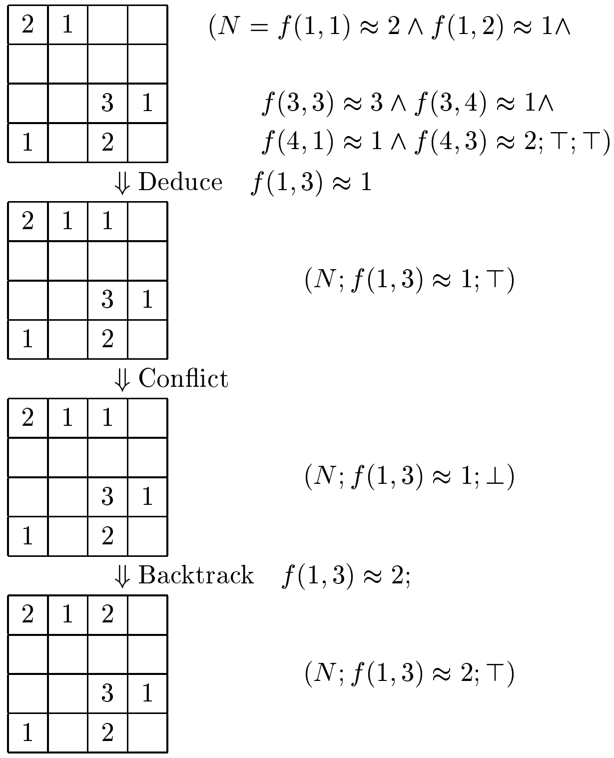

Figure 1.2: Effect of Applying the Inference Rules

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; ⊤; ⊤).
   Output : A final state S = (N; D; ⊤) or S = (N; ⊤; ⊥)
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  end
10 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

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

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.

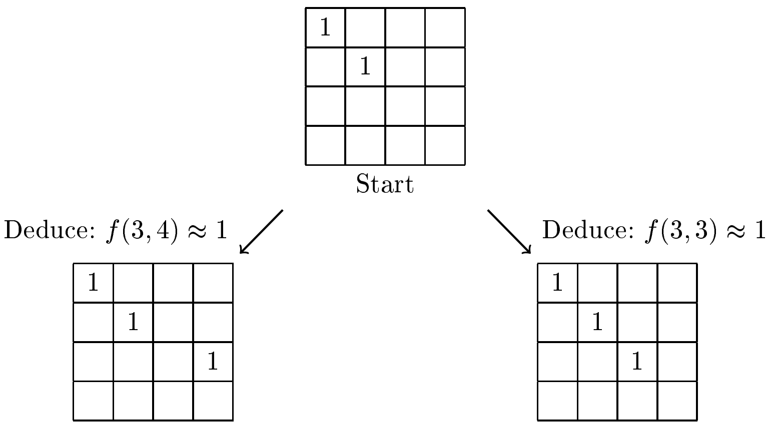

Figure 1.3: Divergence of Rule Deduce

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 $R \cup Q := \{(m_1, \ldots, m_n) \in M \mid (m_1, \ldots, m_n) \in R$ or $(m_1, \ldots, m_n) \in Q\}$ and $R \cap Q := \{(m_1, \ldots, m_n) \in M \mid (m_1, \ldots, m_n) \in R$ 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 $\operatorname{chr}(L, x)$ denotes the *characteristic function* for some decision problem L and is defined by $\operatorname{chr}(L, u) = 1$ if $u \in L$ and $\operatorname{chr}(L, u) = 0$ 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 $g \in$ 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 $L' \in$ P then $L \in$ 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 $u = a_1 \ldots a_n$ there are words $v_0, \ldots, v_n$ such that $v = v_0 a_1 v_1 a_2 \ldots a_n v_n$.

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

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 $u_1, u_2, \ldots$ 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 $u_1, u_2, \dots$ 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 $au'_{k_1}, au'_{k_2}, \dots$ 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 $u_1, u_2, \dots, u_{k_1-1}, u'_{k_1}, u'_{k_2}, \dots$. 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 $\{2\} \not>_{\text{mul}} \{2, 1\}$.

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 $\succ_{\text{mul}}$.

T

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 $a_{i-1} \leftarrow a_i \to a_{i+1}$.

C

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 PROP(Σ) of *propositional formulas* over a signature Σ is inductively defined by:

PROP(Σ)	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"
$(\phi \rightarrow \psi)$	connective $\rightarrow$ denotes "implication"
$(\phi \leftrightarrow \psi)$	connective $\leftrightarrow$ denotes "equivalence"

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

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

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