
# Theoretical Computer Science: An Introduction to Logic via *Python*

## — Summer 2023 —

## Baden-Wuerttemberg Cooperative State University (DHBW)

Prof. Dr. Karl Stroetmann

March 30, 2023

These lecture notes, the corresponding LaTeX sources and the programs discussed in these lecture notes are available at

`https://github.com/karlstroetmann/Logic`.

The lecture notes can be found in the directory `Lecture-Notes` in the file `logic.pdf`. The Jupyter Notebooks discussed in this lecture are found in the directory `Python`. These lecture notes are revised occasionally. To automatically update the lecture notes, you can install the program `git`. Then, using the command line of your favourite operating system, you can clone my repository using the command

```
git clone https://github.com/karlstroetmann/Logic.git.
```

Once the repository has been cloned, it can be updated using the command

```
git pull.
```

# Contents

# Chapter 1

# Introduction

For the uninitiated, mathematical logic is both quite abstract and pretty arcane. In this short chapter, I would like to motivate why you have to learn logic in order to become a computer scientist. After that, I will give a short overview of the topics covered in this lecture.

## 1.1 Motivation

When we discussed algorithms in the previous lecture, we identified three important properties of an algorithm: An algorithm should be

- correct,
- efficient, and
- simple.

We have already discussed efficiency in the lecture on algorithms. This lecture will therefore focus on the correctness. In the rest of this section I want to further motivate the importance of correctness of algorithms.

Modern software systems are among the most complex systems developed by mankind. You can get a sense of the complexity of these systems if you look at the amount of work that is necessary to build and maintain complex software systems. Today it is quite common that complex software projects require more than a thousand developers. Of course, the failure of a project of this size is very costly and can have catastrophic consequences. Nevertheless, the recent history shows that these failures happen. The article

5 of the Biggest Information Technology Failures and Scares

presents several examples showing big software projects that have failed and have subsequently caused huge financial losses. These and numerous other examples show that the development of complex software systems requires a high level of precision and diligence. Hence, the development of software needs a solid scientific foundation. mathematical logic is an important parts of this foundation that has immediate applications in computer science.

(a) Logic can be used to specify the interfaces of complex systems.

(b) The correctness of digital circuits can be verified using automatic theorem provers that are based on propositional logic.

It is easy to extend this enumeration. However, besides their immediate applications, there is another reason you have to study both logic and set theory: Without the proper use of abstractions, complex software systems cannot be managed. After all, nobody is able to keep millions of lines of program code in her head. The only way to construct and manage a software system of this size is to introduce the right abstractions and to develop the system in layers. Hence, the ability to work with abstract concepts is one of the main virtues of a modern computer scientist. Exposing students to mathematics in general and logic in particular trains their abilities to grasp abstract concepts.

From my past teaching experience I know that many students think that a good programmer already is a good computer scientist. In reality, we have

$$\text{good programmer} \neq \text{good computer scientist}.$$

This should not be too surprising. After all, there is no reason to believe that a good bricklayer is a good architect and neither is a good architect necessarily a good bricklayer. In computer science, a good programmer need not be a scientist at all, while a computer scientist, by its very name, is a scientist. There is no denying that mathematics in general and logic in particular is an important part of science. Furthermore, these topics form the foundation of computer science. Therefore, you should master them. In addition, this part of your scientific education is much more permanent than the knowledge of a particular programming language. Nobody knows which programming language will be *en vogue* in 10 years from now. In three years, when you start your professional career, quite a lot of you will have to learn a new programming language. Then your ability to quickly grasp new concepts will be much more important than your skills in any particular programming language.

## 1.2 Overview

The first lecture in theoretical computer science creates the foundation that is needed for future lectures. This lecture deals mostly with mathematical logic and is structured as follows.

(a) We begin our lecture by investigating the limits of computability.

For certain problems there is no algorithm that can solve the problem algorithmically. For example, the question whether a given program will terminate for a given input is not decidable. This is known as the halting problem. We will prove the undecidability of the halting problem in the second chapter.

(b) The third chapter presents two techniques that we can use to establish the correctness of algorithms.

- Computational induction is the method of choice for proving the correctness of recursive algorithms.
- Hoare logic is used to verify iterative algorithms.

(c) The fourth chapter discusses propositional logic.

In logic, we distinguish between propositional logic, first order logic, and higher order logic. Propositional logic is only concerned with the logical connectives

$$\text{“}\neg\text{”}, \text{“}\wedge\text{”}, \text{“}\vee\text{”}, \text{“}\rightarrow\text{” und “}\leftrightarrow\text{”},$$

while first-order logic also investigates the quantifiers

"$\forall$" and "$\exists$",

where these quantifiers range over the objects of the domain of discourse. Finally, in higher order logic these quantifiers also range over sets, functions, and predicates.

As propositional logic is easier to grasp than first-order logic, we start our investigation of logic with propositional logic. Furthermore, propositional logic has the advantage of being decidable: We will present an algorithm that can check whether a propositional formula is satisfiable. In contrast to propositional logic, first-order logic is not decidable.

Next, we discuss applications of propositional logic: We will show how the 8 queens problem can be reduced to the question whether a formula from propositional logic is satisfiable. We present the algorithm of Davis and Putnam that can decide the satisfiability of a propositional formula. and, for example, is able to solve the 8 queens problem.

(d) Finally, we discuss first-order logic.

The most important concept of the last chapter will be the notion of a formal proof in first order logic. To this end, we introduce a formal proof system that is complete for first order logic. Completeness means that we will develop an algorithm that can prove the correctness of every first-order formula that is universally valid. This algorithm is the foundation of automated theorem proving.

As an application of theorem proving we discuss the systems Prover9 and Mace4. Prover9 is an automated theorem prover, while Mace4 can be used to refute a mathematical conjecture.

# Chapter 2

# Limits of Computability

Every discipline of the sciences has its limits: Students of the medical sciences soon realize that it is difficult to raise the dead and even religious zealots have trouble to walk on water. Similarly, computer science has its limits. We will discuss these limits next. First, we show that we cannot decide whether a computer program will eventually terminate or whether it will run forever. Second, we prove that it is impossible to automatically check whether two functions are equivalent.

## 2.1 The Halting Problem

In this subsection we prove that it is not possible for a computer program to decide whether another computer program does terminate. This problem is known as the *halting problem*. Before we give a formal proof that the halting problem is undecidable, let us discuss one example that shows why it is indeed difficult to decide whether a program does always terminate. Consider the program shown in Figure 2.1 on page 7. This program contains a `while`-loop in line 18. If there is a natural number $n \geq m$ such that the expression,

$$\texttt{legendre}(n)$$

in line 19 evaluates to `false`, then the program prints a message and terminates. However, if $\texttt{legendre}(n)$ is true for all $n \geq m$, then the `while`-loop does not terminate.

Given a natural number `n`, the expression $\texttt{legendre}(n)$ tests whether there is a prime number between $\texttt{n}^2$ and $(\texttt{n}+1)^2$. If, however, the set

$$\{k \in \mathbb{N} \mid n^2 \leq k \wedge k \leq (n+1)^2\}$$

does not contain a prime number, then $\texttt{legendre}(n)$ evaluates to `False` for this value of $n$. The function `legendre` is defined in line 7. Given a natural number $n$, it returns `True` if and only if the formula

$$\exists k \in \mathbb{N} : \big(n^2 < k \wedge k < (n+1)^2 \wedge \mathit{isPrime}(k)\big)$$

holds true. The French mathematican Adrien-Marie Legendre (1752 – 1833) conjectured that for any natural number $n \in \mathbb{N}$ there is prime number $p$ such that

$$n^2 < p \wedge p < (n+1)^2$$

holds. Although there are a number of arguments in support of Legendre's conjecture, to this day nobody has been able to prove it. The answer to the question, whether the invocation of the function $f$ will terminate for every user input is, therefore, unknown as it depends on the truth

of Legendre's conjecture: If we had some procedure that could check whether the function call `find_counter_example(1)` does terminate, then this procedure would be able to decide whether Legendre's theorem is true. Therefore, it should come as no surprise that such a procedure does not exist.

```
def divisors(k):
    return { t for t in range(1, k+1) if k % t == 0 }

def is_prime(k):
    return divisors(k) == {1, k}

def legendre(n):
    k = n * n + 1;
    while k < (n + 1) ** 2:
        if is_prime(k):
            print(f'{n}**2 < {k} < {n+1}**2')
            return True
        k += 1
    return False

def find_counter_example(m):
    n = m
    while True:
       if legendre(n):
           n = n + 1
       else:
           print(f'Counter example found: No prime between {n}**2 and {n+1}**2!')
           return
```

Figure 2.1: A program checking Legendre's conjecture.

Let us proceed to prove formally that the halting problem is not solvable. To this end, we need the following definition.

**Definition 1 (Test Function)** *A string $t$ is a test function with name $f$ iff $t$ has the form*

```
"""
def f(x):
   body
"""
```

*and, furthermore, the string $t$ can be parsed as a Python function, that is the evaluation of the expression*

$$\texttt{exec}(t)$$

*does not yield an error. The set of all test functions is denoted as $TF$. If $t \in TF$ and $t$ has the name $f$, then this is written as*

$$\texttt{name}(t) = f.$$ □

**Examples**:

1. We define the string $s_1$ as follows:

```
"""
def simple(x):
    return 0
"""
```

   Then $s_1$ is a test function with the name `simple`.

2. We define the string $s_2$ as

```
"""
def loop(x):
    while True:
        x = x + 1
"""
```

   Then $s_2$ is a test function with the name `loop`.

3. We define the string $s_3$ as

```
"""
def hugo(x):
    return ++x
"""
```

   Then $s_3$ is not a test function. The reason is that *Python* does not support the operator "++". Therefore,

```
exec(s3)
```

   yields an error message complaining about the two "+" characters.

In order to be able to formalize the halting problem succinctly, we introduce three additional notations.

**Notation 2 ($\rightsquigarrow$, $\downarrow$, $\uparrow$)** *If $n$ is the name of a Python function that takes $k$ arguments $a_1, \cdots, a_k$, then we write*

$$n(a_1, \cdots, a_k) \rightsquigarrow r$$

*iff the evaluation of the expression $n(a_1, \cdots, a_k)$ yields the result $r$. If we are not concerned with the result $r$ but only want to state that the evaluation terminates eventually, then we will write*

$$n(a_1, \cdots, a_k) \downarrow$$

*and read this notation as "*evaluation of $n(a_1, \cdots, a_k)$ terminates*". If the evaluation of the expression $n(a_1, \cdots, a_k)$ does not terminate, this is written as*

$$n(a_1, \cdots, a_k) \uparrow.$$

*This notation is read as "*evaluation of $n(a_1, \cdots, a_k)$ diverges*".* □

**Examples**: Using the test functions defined earlier, we have:

1. `simple("emil")` $\rightsquigarrow 0$,
2. `simple("emil")` $\downarrow$,
3. `loop(2)` $\uparrow$.

The halting problem for *Python* functions is the question whether there is a *Python* function

```
def stops(t, a):
    ⋮
```

that takes as input a test function $t$ and a string $a$ and that satisfies the following specification:

1. $t \notin TF \quad \Leftrightarrow \quad \texttt{stops}(t, a) \rightsquigarrow 2$.
   If the first argument of `stops` is not a test function, then `stops(`$t$`, `$a$`)` returns the number 2.
2. $t \in TF \wedge \texttt{name}(t) = n \wedge n(a) \downarrow \quad \Leftrightarrow \quad \texttt{stops}(t, a) \rightsquigarrow 1$.
   If the first argument of `stops` is a test function with name $n$ and, furthermore, the evaluation of $n(a)$ terminates, then `stops(`$t$`, `$a$`)` returns the number 1.
3. $t \in TF \wedge \texttt{name}(t) = n \wedge n(a) \uparrow \quad \Leftrightarrow \quad \texttt{stops}(t, a) \rightsquigarrow 0$.
   If the first argument of `stops` is a test function with name $n$ but the evaluation of $n(a)$ diverges, then `stops(`$t$`, `$a$`)` returns the number 0.

If there was a *Python* function `stops` that did satisfy the specification given above, then the halting problem for *Python* would be decidable.

**Theorem 3 (Alan Turing, 1936)** *The halting problem is undecidable.*

**Proof**: In order to prove the undecidabilty of the halting problem we have to show that there can be no function `stops` satisfying the specification given above. This calls for an indirect proof also known as a *proof by contradiction*. We will therefore assume that a function `stops` solving the halting problem does exist and we will then show that this assumption leads to a contradiction. This contradiction will leave us with the conclusion that there can be no function `stops` that satisfies the specification given above and that, therefore, the halting problem is undecidable.

In order to proceed, let us assume that a *Python* function `stops` satisfying the specification given above exists and let us define the string *turing* as shown in Figure 2.2 below.

Given this definition it is easy to check that *turing* is, indeed, a test function with the name "`alan`", that is we have

$$\textit{turing} \in TF \ \wedge \ \texttt{name}(\textit{turing}) = \texttt{alan}.$$

Therefore, we can use the string *turing* as the first argument of the function `stops`. Let us determine the value of the following expression:

```
stops(turing, turing)
```

Since we have already noted that *turing* is test function, according to the specification of the function

```
1  turing = """
2           def alan(x):
3               result = stops(x, x)
4               if result == 1:
5                   while True:
6                       print("... looping ...")
7               return result
8           """
```

Figure 2.2: Definition of the string *turing*.

`stops` there are only two cases left:

$$\texttt{stops}(\textit{turing}, \textit{turing}) \rightsquigarrow 0 \quad \vee \quad \texttt{stops}(\textit{turing}, \textit{turing}) \rightsquigarrow 1.$$

Let us consider these cases in turn.

1. $\texttt{stops}(\textit{turing}, \textit{turing}) \rightsquigarrow 0$.

   According to the specification of `stops` we should then have

   $$\texttt{alan}(\textit{turing}) \uparrow.$$

   Let us check whether this is true. In order to do this, we have to check what happens when the expression

   $$\texttt{alan}(\textit{turing})$$

   is evaluated:

   (a) Since we have assumed for this case that the expression $\texttt{stops}(\textit{turing}, \textit{turing})$ yields 0, in line 2, the variable `result` is assigned the value 0.

   (b) Line 3 now tests whether `result` is 1. Of course, this test fails. Therefore, the block of the `if`-statement is not executed.

   (c) Finally, in line 8 the value of the variable `result` is returned.

   All in all we see that the call of the function `alan` does terminate when given the argument *turing*. However, this is the opposite of what the function `stops` has claimed.

   Therefore, this case has lead us to a contradiction.

2. $\texttt{stops}(\textit{turing}, \textit{turing}) \rightsquigarrow 1$.

   According to the specification of `stops` we should then have

   $$\texttt{alan}(\textit{turing}) \downarrow,$$

   i.e. the evaluation of $\texttt{alan}(\textit{turing})$ should terminate.

   Again, let us check in detail whether this is true.

   (a) Since we have assumed for this case that the expression $\texttt{stops}(\textit{turing}, \textit{turing})$ yields 1, in line 2, the variable `result` is assigned the value 1.

(b) Line 3 now tests whether `result` is 1. Of course, this time the test succeeds. Therefore, the block of the `if`-statement is executed.

(c) However, this block contains an infinite loop. Therefore, the evaluation of `alan`(*turing*) diverges. But this contradicts the specification of `stops`!

Therefore, the second case also leads to a contradiction.

As we have obtained contradictions in both cases, the assumption that there is a function `stops` that solves the halting problem is refuted. □

**Remark**: The proof of the fact that the halting problem is undecidable was given 1936 by Alan Turing (1912 – 1954) [Tur36]. Of course, Turing did not solve the problem for *Python* but rather for the so called *Turing machines*. A Turing maschine can be interpreted as a formal description of an algorithm. Therefore, Turing has shown that there is no algorithm that is able to decide whether some given algorithm will always terminate.

**Remark**: At this point you might wonder whether there might be another programming language that is more powerful so that programming in this more powerful language it would be possible to solve the halting problem. However, if you check the proof given for *Python* you will easily see that this proof can be adapted to any other programming language that is as least as powerful as *Python*. ◇

Of course, if a programming language is very restricted, then it might be possible to check the halting problem for this weak programming language. But for any programming language that supports at least `while`-loops, `if`-statements, and the definition of procedures the argument given above shows that the halting problem is not solvable.

**Exercise 1**: Show that if the halting problem would be solvable, then it would be possible to write a program that checks whether there are infinitely many twin primes. A twin prime is pair of natural numbers $\langle p, p+2 \rangle$ such that both $p$ and $p+2$ are prime numbers. The *twin prime conjecture* is one of the oldest unsolved mathematical problems. ◇

**Exercise 2**: A set $X$ is countably infinite iff $X$ is infinite and there is a function

$$f : \mathbb{N} \to X$$

such that for all $x \in X$ there is a $n \in \mathbb{N}$ such that $x$ is the image of $n$ under $f$:

$$\forall x \in X : \exists n \in \mathbb{N} : x = f(n).$$

(A function of this kind is called surjective. Some authors define a set to be countably infinite iff there is an injective function $f : \mathbb{N} \to X$. It can be shown that if there is a surjective function $f : \mathbb{N} \to X$ and $X$ is infinite, then there also is an injective function $f : \mathbb{N} \to X$. Therefore, these definitions are equivalent.) If a set is infinite, but not countably infinite, we call it uncountable. Prove that the set $2^{\mathbb{N}}$, which is the set of all subsets of $\mathbb{N}$ is not countably infinite.

**Hint**: Your proof should be similar to the proof that the halting problem is undecidable. Proceed as follows: Assume that there is a function $f$ enumerating the subsets of $\mathbb{N}$, that is assume that

$$\forall x \in 2^{\mathbb{N}} : \exists n \in \mathbb{N} : x = f(n)$$

holds. Next, and this is the crucial step, define a set `Cantor` as follows:

$$\texttt{Cantor} := \{n \in \mathbb{N} \mid n \notin f(n)\}.$$

Now try to derive a contradiction. ◇

## 2.2 Undecidability of the Equivalence Problem

Unfortunately, the halting problem is not the only undecidable problem in computer science. Another important problem that is undecidable is the question whether two given functions always compute the same result. To state this more formally, we need the following definition.

**Definition 4 ($\simeq$)** *Assume $n_1$ and $n_2$ are the names of two Python functions that take arguments $a_1$, $\cdots$, $a_k$. Let us define*

$$n_1(a_1, \cdots, a_k) \simeq n_2(a_1, \cdots, a_k)$$

*if and only if either of the following cases is true:*

1. $n_1(a_1, \cdots, a_k) \uparrow \quad \wedge \quad n_2(a_1, \cdots, a_k) \uparrow$,
   *that is both function calls diverge.*
2. $\exists r : \Big(n_1(a_1, \cdots, a_k) \rightsquigarrow r \quad \wedge \quad n_2(a_1, \cdots, a_k) \rightsquigarrow r\Big)$
   *that is both function calls terminate and compute the same result.*

*If $n_1(a_1, \cdots, a_k) \simeq n_2(a_1, \cdots, a_k)$ holds, then the expressions $n_1(a_1, \cdots, a_k)$ and $n_2(a_1, \cdots, a_k)$ are partially equivalent.* □

We are now ready to state the equivalence problem. A *Python* function `equal` solves the *equivalence problem* if it is defined as

```
def equal(p1, p2, a):
    body
```

and, furthermore, it satisfies the following specification:

1. $p_1 \notin TF \ \vee \ p_2 \notin TF \quad \Leftrightarrow \quad \texttt{equal}(p_1, p_2, a) \rightsquigarrow 2.$
2. If
   (a) $p_1 \in TF \ \wedge \ \texttt{name}(p_1) = n_1$,
   (b) $p_2 \in TF \ \wedge \ \texttt{name}(p_2) = n_2$ and
   (c) $n_1(a) \simeq n_2(a)$

   holds, then we must have:

   $$\texttt{equal}(p_1, p_2, a) \rightsquigarrow 1.$$
3. Otherwise we must have

   $$\texttt{equal}(p_1, p_2, a) \rightsquigarrow 0.$$

**Theorem 5** *The equivalence problem is undecidable.*

**Proof**: The proof is by contradiction. Therefore, assume that there is a function `equal` such that `equal` solves the equivalence problem. Assuming `equal` exists, we will then proceed to define a function `stops` that solves the halting problem. Figure 2.3 shows this construction of the function `stops`.

Notice that in line 6 the function `equal` is called with a string that is test function with name `loop`. This test function has the following form:

```
def stops(t, a):
    l = """def loop(x):
                while True:
                    x = 1
        """
    e = equal(l, t, a);
    if e == 2:
        return 2
    else:
        return 1 - e
```

Figure 2.3: An implementation of the function `stops`.

```
def loop(x):
    while True:
        x = 1
```

Independent from the argument $x$, the function `loop` does not terminate. Therefore, if the first argument $t$ of `stops` is a test function with name $n$, the function `equal` will return 1 if $n(a)$ diverges, and will return 0 otherwise. But this implementation of `stops` would then solve the halting problem as for a given test function $t$ with name $n$ and argument $a$ the function `stops` would return 1 if and only the evaluation of $n(a)$ terminates. As we have already proven that the halting problem is undecidable, there can be no function `equal` that solves the equivalence problem either. □

**Remark**: The unsolvability of the equivalence problem has been proven by Henry Gordon Rice [Ric53] in 1953. ◇

## 2.3 Concluding Remarks

Although, in general, we cannot decide whether a program terminates for a given input, this does not mean that we should not attempt to do so. After all, we only have proven that there is no procedure that can always check whether a given program will terminate. There might well exist a procedure for termination checking that works most of the time. Indeed, there are a number of systems that try to check whether a program will terminate for every input. For example, for Prolog programs, the paper "*Automated Modular Termination Proofs for Real Prolog Programs*" [MGS96] describes a successful approach. The recent years have seen a lot of progress in this area. The article "Proving Program Termination" [CPR11] reviews these developments. However, as the recently developed systems rely on both *automatic theorem proving* and *Ramsey theory* they are quite out of the scope of this lecture.

## 2.4 Chapter Review

You should be able to solve the following exercises.

(a) Define the halting problem.

(b) Prove that the halting problem is not decidable.

(c) Define the equivalence problem.

(d) Prove that the equivalence problem is not decidable.

(e) Define the notion of a countable set.

(f) Prove that the set $2^{\mathbb{N}}$ is not countable.

## 2.5 Further Reading

The book "*Introduction to the Theory of Computation*" by Michael Sipser [Sip96] discusses the undecidability of the halting problem in section 4.2. It also covers many related undecidable problems.

Another good book discussing undecidability is the book "*Introduction to Automata Theory, Languages, and Computation*" written by John E. Hopcroft, Rajeev Motwani and Jeffrey D. Ullman [HMU06]. This book is the third edition of a classic text. In this book, the topic of undecidability is discussed in chapter 9.

The exposition in these books is based on Turing machines and is therefore more formal than the exposition given here. This increased formality is necessary to prove that, for example, it is undecidable whether two context free grammars are equivalent.

A word of warning: The two books mentioned above are not intended to be read by undergraduates in their first year. If you want to dive deeper into the concept of undecidability, you should do so only after you have finished your second year.

# Chapter 3

# Proving the Correctness of an Algorithm*

In this chapter we will show two different methods that can be used to prove the correctness of *Python* function.

(a) The method of computational induction can be used to verify the correctness of a *Python* function that is defined recursively.

(b) In order to establish the correctness of a *Python* function that is defined iteratively we use symbolic execution.

## 3.1 Computational Induction

Figure 3.1 shows the definition of the function `power`$(m, n)$ that computes the value $m^n$. We will verify the correctness of this function.

```
1  def power(m, n):
2      if n == 0:
3          return 1
4      p = power(m, n // 2)
5      if n % 2 == 0:
6          return p * p
7      else:
8          return p * p * m
```

Figure 3.1: Computation of $m^n$ for $m, n \in \mathbb{N}$.

It is by no means obvious that the program shown in 3.1 does compute $m^n$. We prove this claim by computational induction. Computational induction is an induction on the number of recursive invocations. This method is the method of choice to prove the correctness of a function if this function is defined recursively. A proof by computational induction consists of three parts:

1. The base case.

   In the base case we have to show that the function definition is correct in all those cases where the function does not invoke itself recursively.

2. The induction step.

   In the induction step we have to prove that the function definition works in all those cases where the function does invoke itself recursively. In order to carry out this proof we may assume that the results computed by the recursively invocations are correct. This assumption is called the induction hypotheses.

3. The termination proof.

   In this final step we have to show that the recursive definition of the function is well founded, i.e. we have to prove that the recursive invocations terminate.

Let us prove the claim

$$\texttt{power}(m, n) = m^n$$

by computational induction.

1. **Base case**:

   The only case where `power` does not invoke itself recursively is the case $n = 0$. In this case, we have

   $$\texttt{power}(m, 0) = 1 = m^0. \quad \surd$$

2. **Induction step**:

   The recursive invocation of `power` has the form $\texttt{power}(m, n \mathbin{//} 2)$. By the induction hypotheses we may assume that

   $$\texttt{power}(m, n \mathbin{//} 2) = m^{n \mathbin{//} 2}$$

   holds. After the recursive invocation there are two cases that have to be dealt with separately.

   (a) $n \mathbin{\%} 2 = 0$, therefore $n$ is even.
   Then there exists a number $k \in \mathbb{N}$ such that $n = 2 \cdot k$ and therefore $n \mathbin{//} 2 = k$. Hence we have:

   $$\begin{aligned} \texttt{power}(m, n) &= \texttt{power}(m, k) \cdot \texttt{power}(m, k) \\ &\overset{\text{IV}}{=} m^k \cdot m^k \\ &= m^{2 \cdot k} \\ &= m^n. \end{aligned}$$

   (b) $n \mathbin{\%} 2 = 1$, therefore $n$ is odd.
   Then there exists a number $k \in \mathbb{N}$ such that $n = 2 \cdot k + 1$ and we have $n \mathbin{//} 2 = k$. In this case we have:

   $$\begin{aligned} \texttt{power}(m, n) &= \texttt{power}(m, k) \cdot \texttt{power}(m, k) \cdot m \\ &\overset{\text{IV}}{=} m^k \cdot m^k \cdot m \\ &= m^{2 \cdot k + 1} \\ &= m^n. \end{aligned}$$

As we have shown that $\texttt{power}(m,n) = m^n$ in both cases, the induction step is finished. √

3. **Termination proof**: Every time the function `power` is invoked as $\texttt{power}(m,n)$ and $n > 0$, the recursive invocation has the form $\texttt{power}(m, n \mathbin{//} 2)$ and, since $n \mathbin{//} 2 < n$ for all $n > 0$, the second argument is decreased. As this argument is a natural number, it must eventually reach 0. But if the second argument of the function `power` is 0, the function terminates immediately. √ □

```
def div_mod(m, n):
    if m < n:
        return 0, m
    q, r = div_mod(m // 2, n)
    if 2 * r + m % 2 < n:
        return 2 * q, 2 * r + m % 2
    else:
        return 2 * q + 1, 2 * r + m % 2 - n
```

Figure 3.2: The function `div_mod`.

**Exercise 3**: Prove that the function `div_mod` that is shown in Figure 3.3 satisfies the specification

$$\texttt{div\_mod}(m,n) = (q,r)\rangle \rightarrow m = q \cdot n + r \;\wedge\; r < n.$$ ◇

```
def gcd(x, y):
    while y != 0:
        x, y = y, x % y
    return x
```

Figure 3.3: The function `gcd`.

**Exercise 4**: Prove that the function `gcd` that is shown in Figure 3.3 computes the greatest common divisor of its arguments. ◇

**Exercise 5**: The integer square root of a natural number $n$ is defined as

$$\texttt{isqrt}(n) := \max\bigl(\{r \in \mathbb{N} \mid r^2 \leq n\}\bigr).$$

Prove that the function `isqrt` that is shown in Figure 3.4 on page 18 computes the integer square root of its argument. ◇

## 3.2 Symbolic Execution

In the last chapter we have seen how to prove the correctness of a recursive function via computational induction. If a function is implemented via loops instead of recursion, then the method of computational induction is not applicable. Therefore, this section introduces the method of symbolic

```
1  def isqrt(n):
2      if n == 0:
3          return 0
4      r = isqrt(n // 4)
5      if (2 * r + 1) ** 2 <= n:
6          return 2 * r + 1
7      else:
8          return 2 * r
```

Figure 3.4: The function `isqrt`.

execution. Using this method it is possible to verify the correctness of programs that are implemented in an iterative fashion using loops. We will introduce this method via a simple example. Consider the program shown in Figure 3.10.

```
1  def power(x_1, y_1):
2      r_1 = 1
3      while y_n > 0:
4          if y_n % 2 == 1:
5              r_{n+1} = r_n * x_n
6          x_{n+1} = x_n * x_n
7          y_{n+1} = y_n // 2
8      return r_N
```

Figure 3.5: An annotated programm to compute powers.

The main difference between a mathematical formula and a program is that in a formula all occurrences of a variable refer to the same value. This is different in a program because the variables change their values dynamically. In order to deal with this property of program variables we have to be able to distinguish the different occurrences of a variable. To this end, we index the program variables. When doing this we have to be aware of the fact that the same occurrence of a program variable can still denote different values if the variable occurs inside a loop. In this case we have to index the variables in a way that the index includes a counter that counts the number of loop iterations. For concreteness, consider the program shown in Figure 3.10. Here, in line 5 the variable `r` has the index $n$ on the right side of the assignment, while it has the index $r_{n+1}$ on the left side of the assignment in line 5. The index $n$ denotes the number of times that the test $y_n$ `> 0` of the `while` loop has been executed. After the `while`-loop finishes, the variable `r` is indexed as $r_N$ in line 8, where $N$ denotes the total number of times that the test `y > 0` has been executed. We show the correctness of the given program next. Let us define

$$a := x_1, \quad b := y_1.$$

We will show that the `while` loop satisfies the invariant

$$r_n \cdot x_n^{y_n} = a^b. \tag{3.1}$$

This claim is proven by induction on the number of loop iterations.

B.C.: $n = 1$.

Since we have $r_1 = 1$, $x_1 = a$, and $y_1 = b$ we have

$$r_n \cdot x_n^{y_n} = r_1 \cdot x_1^{y_1} = 1 \cdot a^b = a^b.$$

I.S.: $n \mapsto n + 1$.

We proof proceeds by a case distinction with respect to the expression $y \,\%\, 2$:

(a) $y_n \,\%\, 2 = 1$.

Then we have $y_n = 2 \cdot (y_n \,//\, 2) + 1$ and $r_{n+1} = r_n \cdot x_n$. Hence

$$\begin{aligned} & r_{n+1} \cdot x_{n+1}^{y_{n+1}} \\ = \;& (r_n \cdot x_n) \cdot (x_n \cdot x_n)^{y_n \,//\, 2} \\ = \;& r_n \cdot x_n^{2 \cdot (y_n \,//\, 2) + 1} \\ = \;& r_n \cdot x_n^{y_n} \\ \overset{i.h.}{=} \;& a^b \end{aligned}$$

(b) $y_n \,\%\, 2 = 0$.

Then we have $y_n = 2 \cdot (y_n \,//\, 2)$ and $r_{n+1} = r_n$. Therefore

$$\begin{aligned} & r_{n+1} \cdot x_{n+1}^{y_{n+1}} \\ = \;& r_n \cdot (x_n \cdot x_n)^{y_n \,//\, 2} \\ = \;& r_n \cdot x_n^{2 \cdot (y_n \,//\, 2)} \\ = \;& r_n \cdot x_n^{y_n} \\ \overset{i.h.}{=} \;& a^b \end{aligned}$$

This shows the validity of equation (3.10). If the `while` loop terminates, we must have $y_N = 0$. If $n = N$, then equation (3.10) yields:

$$\begin{aligned} & r_N \cdot x_N^{y_N} = a^b \\ \Leftrightarrow \;& r_N \cdot x_N^0 = a^b \\ \Leftrightarrow \;& r_N \cdot 1 = a^b \\ \Leftrightarrow \;& r_N = a^b \end{aligned}$$

This shows $r_N = a^b$ and since it is obvious that the `while` loop terminates, we have proven that $\texttt{power}(a, b) = a^b$ holds. □

**Aufgabe 6**: Use the method of symbolic program execution to prove the correctness of the implementation of the Euclidean algorithm that is shown in Figure 3.11 on page 33. During the proof you should make use of the fact that for all positive natural numbers $x$ and $y$ the function `ggt` that computes the greatest common divisor of $m$ and $n$ satisfies the equation

$$\texttt{ggt}(x, y) = \texttt{ggt}(x \,\%\, y, y).$$

Furthermore, the invariant of the `while` loop is

$$\texttt{ggt}(x_n, y_n) = \texttt{ggt}(a, b) \quad \text{where } a := x_1 \text{ and } b := y_1.$$

Using this invariant you should be able to prove that $\texttt{gcd}(a, b) = \texttt{ggt}(a, b)$ for all $a, b \in \mathbb{N}$ such that $a > 0$. ◇

```
def gcd(x, y):
    while y != 0:
        x, y = y, x % y
    return x
```

Figure 3.6: The Euclidean algorithm.

## 3.3 Check Your Understanding

(a) Explain the method of computational induction.

(b) Use the method of computational induction to prove the correctness of the function `div_mod`.

(c) Explain the method of symbolic execution.

(d) Use the method of symbolic execution to prove the correctness of Euklid's algorithm.

(e) When would you use computational induction and when would you choose symbolic execution instead?

## 3.4 Hoare Logic

In this chapter we introduce *Hoare logic*. This is a formal system that is used to prove the correctness of imperative computer programs. Hoare logic has been introduced 1969 by Sir Charles Antony Richard Hoare, who is the inventor of the quicksort algorithm.

### 3.4.1 Preconditions and Postconditions

Hoare logic is based on preconditions and postconditions. If `P` is a program fragment and if $F$ and $G$ are logical formulæ, then we call $F$ a precondition and $G$ a postcondition for the program fragment `P` if the following holds: If `P` is executed in a state $s$ such that the formula $F$ holds in $s$, then the execution of `P` will change the state $s$ into a new state $s'$ such that $G$ holds in $s'$. This is written as

$$\{F\} \quad \texttt{P} \quad \{G\}.$$

We will read this notation as "*executing P changes F into G*". The formula

$$\{F\} \quad \texttt{P} \quad \{G\}$$

is called a *Hoare triple*.

**Examples**:

1. The assignment "`x := 1;`" satisfies the specification

   $$\{\texttt{true}\} \quad \texttt{x := 1;} \quad \{x = 1\}.$$

   Here, the precondition is the trivial condition "`true`", since the postcondition "$x = 1$" will always be satisfied after this assignment.

2. The assignment "`x = x + 1;`" satisfies the specification

   $$\{x = 1\} \quad \texttt{x := x + 1;} \quad \{x = 2\}.$$

   If the precondition is "$x = 1$", then it is obvious that the postcondition has to be "$x = 2$".

3. Let us consider the assignment "`x = x + 1;`" again. However, this time the precondition is given as "$prime(x)$", which is only true if $x$ is a prime number. This time, the Hoare triple is given as

   $$\{prime(x)\} \quad \texttt{x := x + 1;} \quad \{prime(x-1)\}.$$

This might look strange at first. Many students think that this Hoare triple should rather be written as

$$\{prime(x)\} \quad \texttt{x := x + 1;} \quad \{prime(x+1)\}.$$

However, this can easily be refuted by taking $x$ to have the value 2. Then, the precondition $prime(x)$ is satisfied since 2 is a prime number. After the assignment, $x$ has the value 3 and

$$x - 1 = 3 - 1 = 2$$

still is a prime number. However, we also have

$$x + 1 = 3 + 1 = 4$$

and as $4 = 2 \cdot 2$ we see that $x + 1$ is not a prime number!

Let us proceed to show how the different parts of a program can be specified using Hoare triples. We start with the analysis of assignments.

### 3.4.2 Assignments

Let us generalize the previous example. Let us therefore assume that we have an assignment of the form

```
x := h(x);
```

and we want to investigate how the postcondition $G$ of this assignment is related to the precondition $F$. To simplify matters, let us assume that the function $h$ is invertible, i. e. we assume that there is a function $h^{-1}$ such that we have

$$h^{-1}\big(h(x)\big) = x \quad \texttt{and} \quad h\big(h^{-1}(x)\big) = x$$

for all $x$. Then, the function $h^{-1}$ is the inverse of the function $h$. In order to understand the problem of computing the postcondition for the assignment statement given above, let us first consider an example. The assignment

```
x := x + 1;
```

can be written as

```
x := h(x);
```

where the function $h$ is given as

$$h(x) = x + 1$$

and the inverse function $h^{-1}$ is

$$h^{-1}(x) = x - 1.$$

Now we are able to compute the postcondition of the assignment "`x := h(x);`" from the precondition. We have

$$\{F\} \quad \texttt{x := h(x);} \quad \{F\sigma\} \quad \text{where} \quad \sigma = \big[x \mapsto h^{-1}(x)\big].$$

Here, $F\sigma$ denotes the application of the substitution $\sigma$ to the formula $F$. The expression $F\sigma$ is computed from the expression $F$ by replacing every occurrence of the variable $x$ by the term $h^{-1}(x)$. Therefore, the substitution $\sigma$ undoes the effect of the assignment and restores the variables in $F$ to the state before the assignment.

In order to understand why this is the correct way to compute the postcondition, we consider the assignment "`x := x + 1`" again and choose the formula $x = 7$ as precondition. Since $h^{-1}(x) = x - 1$, the substitution $\sigma$ is given as $\sigma = [x \mapsto x - 1]$. Therefore, $F\sigma$ has the form

$$(x = 7)[x \mapsto x - 1] \;\equiv\; (x - 1 = 7).$$

I have used the symbol "$\equiv$" here in order to express that these formulæ are syntactically identical. Therefore, we have

$$\{x = 7\} \quad \texttt{x := x + 1;} \quad \{x - 1 = 7\}.$$

Since the formula $x - 1 = 7$ is equivalent to the formula $x = 8$ the Hoare triple above can be rewritten as

$$\{x = 7\} \quad \texttt{x := x + 1;} \quad \{x = 8\}$$

and this is obviously correct: If the value of $x$ is 7 before the assignment

"`x := x + 1;`"

is executed, then after the assignment is executed, $x$ will have the value 8.

Let us try to understand why

$$\{F\} \quad \texttt{x := h(x);} \quad \{F\sigma\} \quad \text{where} \quad \sigma = \big[x \mapsto h^{-1}(x)\big]$$

is, indeed, correct: Before the assignment "x := h(x);" is executed, the variable $x$ has some fixed value $x_0$. The precondition $F$ is valid for $x_0$. Therefore, the formula $F[x \mapsto x_0]$ is valid before the assignment is executed. However, the variable $x$ does not occur in the formula $F[x \mapsto x_0]$ because it has been replaced by the fixed value $x_0$. Therefore, the formula

$$F[x \mapsto x_0]$$

remains valid after the assignment "`x = h(x);`" is executed. After this assignment, the variable $x$ is set to $h(x_0)$. Therefore, we have

$$x = h(x_0).$$

Let us solve this equation for $x_0$. We find

$$h^{-1}(x) = x_0.$$

Therefore, after the assignment the formula

$$F[x \mapsto x_0] \equiv F[x \mapsto h^{-1}(x)]$$

is valid and this is the formula that is written as $F\sigma$ above.

We conclude this discussion with another example. The unary predicate *prime* checks whether its argument is a prime number. Therefore, $\mathit{prime}(x)$ is true if $x$ is a prime number. Then we have

$$\{\mathit{prime}(x)\} \quad \texttt{x := x + 1;} \quad \{\mathit{prime}(x - 1)\}.$$

The correctness of this Hoare triple should be obvious: If $x$ is a prime and if $x$ is then incremented by 1, then afterwards $x - 1$ is prime.

**Different Forms of Assignments** Not all assignments can be written in the form "`x := h(x);`" where the function $h$ is invertible. Often, a constant $c$ is assigned to some variable $x$. If $x$ does not occur in the precondition $F$, then we have

$$\{F\} \quad \texttt{x := c;} \quad \{F \wedge x = c\}.$$

The formula $F$ can be used to restrict the values of other variables occurring in the program under consideration.

**General Form of the Assignment Rule** In the literature the rule for specifying an assignment is given as

$$\{F[x \mapsto t]\} \quad \texttt{x := t;} \quad \{F\}.$$

Here, $t$ is an arbitrary term that can contain the variable $x$. This rule can be read as follows:

> "*If the formula $F(t)$ is valid in some state and $t$ is assigned to $x$, then after this assignment we have $F(x)$.*"

This rule is obviously correct. However, it is not very useful because in order to apply this rule we first have to rewrite the precondition as $F(t)$. If $t$ is some complex term, this is often very difficult to do.

### 3.4.3 The Weakening Rule

If a program fragment $P$ satisfies the specification

$$\{F\} \quad \texttt{P} \quad \{G\}$$

and if, furthermore, the formula $G$ implies the validity of the formula $H$, that is if

$$G \to H$$

holds, then the program fragment $P$ satisfies

$$\{F\} \quad \texttt{P} \quad \{H\}.$$

The reasoning is as follows: If after executing $P$ we know that $G$ is valid, then, since $G$ implies $H$, the formula $H$ has to be valid, too. Therefore, the following *verification rule*, which is known as the *weakening rule*, is valid:

$$\frac{\{F\} \quad \texttt{P} \quad \{G\}, \qquad G \to H}{\{F\} \quad \texttt{P} \quad \{H\}}$$

The formulæ written over the fraction line are called the *premisses* and the formula under the fraction line is called the *conclusion*. The conclusion and the first premiss are Hoare triples, the second premiss is a formula of first order logic. The interpretation of this rule is that the conclusion is true if the premisses are true.

### 3.4.4 Compound Statements

If the program fragments `P` and `Q` have the specifications

$$\{F_1\} \quad \texttt{P} \quad \{G_1\} \quad \text{and} \quad \{F_2\} \quad \texttt{Q} \quad \{G_2\}$$

and if, furthermore, the postcondition $G_1$ implies the precondition $F_2$, then the composition `P;Q` of `P` and `Q` satisfies the specification

$$\{F_1\} \quad \texttt{P;Q} \quad \{G_2\}.$$

The reasoning is as follows: If, initially, $F_1$ is satisfied and we execute $P$ then we have $G_1$ afterwards. Therefore we also have $F_2$ and if we now execute $Q$ then afterwards we will have $G_2$. This chain of

thoughts is combined in the following verification rule:

$$\frac{\{F_1\}\quad \texttt{P}\quad \{G_1\}, \qquad G_1 \rightarrow F_2, \qquad \{F_2\}\quad \texttt{Q}\quad \{G_2\}}{\{F_1\}\quad \texttt{P;Q}\quad \{G_2\}}$$

If the formulæ $G_1$ and $F_2$ are identical, then this rule can be simplified as follows:

$$\frac{\{F_1\}\quad \texttt{P}\quad \{G_1\}, \qquad \{G_1\}\quad \texttt{Q}\quad \{G_2\}}{\{F_1\}\quad \texttt{P;Q}\quad \{G_2\}}$$

**Example**: Let us analyse the program fragment shown in Figure 3.7. We start our analysis by using the precondition

$$\texttt{x} = a \wedge \texttt{y} = b.$$

Here, $a$ and $b$ are two variables that we use to store the initial values of x and y. The first assignment yields the Hoare triple

$$\{\texttt{x} = a \wedge \texttt{y} = b\} \quad \texttt{x := x - y;} \quad \{(\texttt{x} = a \wedge \texttt{y} = b)\sigma\}$$

where $\sigma = [x \mapsto x + y]$. The form of $\sigma$ follows from the fact that the function $x \mapsto x + y$ is the inverse of the function $x \mapsto x - y$. If we apply $\sigma$ to the formula $x = a \wedge y = b$ we get

$$\{\texttt{x} = a \wedge \texttt{y} = b\} \quad \texttt{x := x - y;} \quad \{\texttt{x + y} = a \wedge \texttt{y} = b\}. \tag{3.2}$$

The second assignment yields the Hoare triple

$$\{\texttt{x + y} = a \wedge \texttt{y} = b\} \quad \texttt{y := y + x;} \quad \{(\texttt{x + y} = a \wedge \texttt{y} = b)\sigma\}$$

where $\sigma = [y \mapsto y - x]$. The reason is that the function $y \mapsto y - x$ is the inverse of the function $y \mapsto y + x$. This time, we get

$$\{\texttt{x + y} = a \wedge \texttt{y} = b\} \quad \texttt{y := y + x;} \quad \{\texttt{x + y - x} = a \wedge \texttt{y - x} = b\}.$$

Simplifying the postcondition yields

$$\{\texttt{x + y} = a \wedge \texttt{y} = b\} \quad \texttt{y := y + x;} \quad \{\texttt{y} = a \wedge \texttt{y - x} = b\}. \tag{3.3}$$

Let us consider the last assignment. We have

$$\{\texttt{y} = a \wedge \texttt{y - x} = b\} \quad \texttt{x := y - x;} \quad \{(\texttt{y} = a \wedge \texttt{y - x} = b)\sigma\}$$

where $\sigma = [x \mapsto y - x]$, since the function $x \mapsto y - x$ is the inverse of the function $x \mapsto y - x$. This yields

$$\{\texttt{y} = a \wedge \texttt{y - x} = b\} \quad \texttt{x := y - x;} \quad \{\texttt{y} = a \wedge \texttt{y - (y - x)} = b\}$$

Simplifying the postcondition gives

$$\{\texttt{y} = a \wedge \texttt{y - x} = b\} \quad \texttt{x := y - x;} \quad \{\texttt{y} = a \wedge \texttt{x} = b\}. \tag{3.4}$$

Combining the Hoare triples (3.2), (3.3) and (3.4) we get

$$\{\texttt{x} = a \wedge \texttt{y} = b\} \quad \texttt{x:=x-y; y:=y+x; x:=y-x;} \quad \{\texttt{y} = a \wedge \texttt{x} = b\}. \tag{3.5}$$

The Hoare triple (3.5) shows that the program fragment shown in Figure 3.7 swaps the values of the variables $x$ and $y$: If the value of $x$ is $a$ and $y$ has the value $b$ before the program is executed, then afterwards $y$ has the value $a$ and $x$ has the value $b$. The trick shown in Figure 3.7 can be used to swap variables without using an auxiliary variable. This is useful because when this code is compiled into machine language, the resulting code will only use two registers.

```
1  x := x - y;
2  y := y + x;
3  x := y - x;
```

Figure 3.7: A tricky way to swap variables.

### 3.4.5 Conditional Statements

In order to compute the effect of a conditional of the form

`if (`$B$`) { P } else { Q }`

let us assume that before the conditional statement is executed, the precondition $F$ is satisfied. We have to analyse the effect of the program fragments $P$ and $Q$. The program fragment $P$ is only executed when $B$ is true. Therefore, the precondition for $P$ is $F \wedge B$. On the other hand, the precondition for the program fragment $Q$ is $F \wedge \neg B$, since $Q$ is only executed if $B$ is false. Hence, we have the following verification rule:

$$\frac{\{F \wedge B\} \quad \texttt{P} \quad \{G\}, \qquad \{F \wedge \neg B\} \quad \texttt{Q} \quad \{G\}}{\{F\} \quad \texttt{if (}B\texttt{) P else Q} \quad \{G\}} \tag{3.6}$$

In this form, the rule is not always applicable. The reason is that the analysis of the program fragments `P` and `Q` yields Hoare triple of the form

$$\{F \wedge B\} \quad \texttt{P} \quad \{G_1\} \qquad \text{and} \qquad \{F \wedge \neg B\} \quad \texttt{Q} \quad \{G_2\}, \tag{3.7}$$

and in general $G_1$ and $G_2$ will be different from each other. In order to be able to apply the rule for conditionals we have to find a formula $G$ that is a consequence of $G_1$ and also a consequence of $G_2$, i. e. we want to have

$$G_1 \rightarrow G \qquad \text{and} \qquad G_2 \rightarrow G.$$

If we find $G$, then the weakening rule can be applied to conclude the validity of

$$\{F \wedge B\} \quad \texttt{P} \quad \{G\} \qquad \text{and} \qquad \{F \wedge \neg B\} \quad \texttt{Q} \quad \{G\},$$

and this gives us the premisses that are needed for the rule (3.6).

**Example**: Let us analyze the following program fragment:

```
if (x < y) { z := x; } else { z := y; }
```

We start with the precondition

$$F = \big(\mathtt{x} = a \wedge \mathtt{y} = b\big)$$

and want to show that the execution of the conditional establishes the postcondition

$$G = \big(z = \mathit{min}(a, b)\big).$$

The first assignment "`z := x;`" gives the Hoare triple

$$\big\{\mathtt{x} = a \wedge \mathtt{y} = b \wedge \mathtt{x} < \mathtt{y}\big\} \quad \mathtt{z := x} \quad \big\{\mathtt{x} = a \wedge \mathtt{y} = b \wedge \mathtt{x} < \mathtt{y} \wedge \mathtt{z} = \mathtt{x}\big\}.$$

In the same way, the second assignment "`z := y`" yields

$$\big\{\mathtt{x} = a \wedge \mathtt{y} = b \wedge \mathtt{x} \geq \mathtt{y}\big\} \quad \mathtt{z := y} \quad \big\{\mathtt{x} = a \wedge \mathtt{y} = b \wedge \mathtt{x} \geq \mathtt{y} \wedge \mathtt{z} = \mathtt{y}\big\}.$$

Since we have

$$\mathtt{x} = a \wedge \mathtt{y} = b \wedge \mathtt{x} < \mathtt{y} \wedge \mathtt{z} = \mathtt{x} \rightarrow \mathtt{z} = \min(a, b)$$

and also

$$\mathtt{x} = a \wedge \mathtt{y} = b \wedge \mathtt{x} \geq \mathtt{y} \wedge \mathtt{z} = \mathtt{y} \rightarrow \mathtt{z} = \min(a, b).$$

Using the weakening rule we conclude that

$$\big\{\mathtt{x} = a \wedge \mathtt{y} = b \wedge \mathtt{x} < \mathtt{y}\big\} \quad \mathtt{z := x;} \quad \big\{\mathtt{z} = \min(a, b)\big\} \qquad \text{and}$$
$$\big\{\mathtt{x} = a \wedge \mathtt{y} = b \wedge \mathtt{x} \geq \mathtt{y}\big\} \quad \mathtt{z := y;} \quad \big\{\mathtt{z} = \min(a, b)\big\}$$

holds. Now we can apply the rule for the conditional and conclude that

$$\big\{\mathtt{x} = a \wedge \mathtt{y} = b\big\} \quad \mathtt{if\ (x < y)\ \{\ z := x;\ \}\ else\ \{\ z := y;\ \}} \quad \big\{\mathtt{z} = \min(a, b)\big\}$$

holds. Thus we have shown that the program fragment above computes the minimum of the numbers $a$ and $b$.

### 3.4.6 Loops

Finally, let us analyze the effect of a loop of the form

```
while (B) { P }
```

The important point here is that the postcondition of the $n$-th execution of the body of the loop $P$ is the precondition of the $(n+1)$-th execution of $P$. Basically this means that the precondition and the postcondition of $P$ have to be more or less the same. Hence, this condition is called the *loop invariant*. Therefore, the details of the verification rule for `while` loops are as follows:

$$\frac{\big\{I \wedge B\big\} \quad \mathtt{P} \quad \big\{I\big\}}{\big\{I\big\} \quad \mathtt{while\ (}B\mathtt{)\ \{\ P\ \}} \quad \big\{I \wedge \neg B\big\}}$$

The premiss of this rule expresses the fact that the invariant $I$ remains valid on execution of $P$. However, since $P$ is only executed as long as $B$ is `true`, the precondition for $P$ is actually the formula $I \wedge B$. The conclusion of the rule says that if the invariant $I$ is true before the loop is executed, then $I$ will be true after the loop has finished. This result is intuitive since every time $P$ is executed $I$ remains

valid. Furthermore, the loop only terminates once $B$ gets `false`. Therefore, the postcondition of the loop can be strengthened by adding $\neg B$.

### 3.4.7 The Euclidean Algorithm

In this section we show how the verification rules of the last section can be used to prove the correctness of a non-trivial program. We will show that the algorithm shown in Figure 3.11 on page 33 is correct. The procedure shown in this figure implements the *Euclidean algorithm* to compute the greatest common divisor of two natural numbers. Our proof is based on the following property of the function `gcd`:

$$\texttt{gcd}(x+y,y) = \texttt{gcd}(x,y) \quad \text{for all } x,y \in \mathbb{N}.$$

```
1  gcd := procedure(x, y) {
2      while (x != y) {
3          if (x < y) {
4              y := y - x;
5          } else {
6              x := x - y;
7          }
8      }
9      return x;
10 };
```

Figure 3.8: The Euclidean Algorithm to compute the greatest common divisor.

#### Correctness Proof of the Euclidean Algorithm

To start our correctness proof we formulate the invariant of the `while` loop. Let us define

$$I := \big(x > 0 \wedge y > 0 \wedge \texttt{gcd}(x,y) = \texttt{gcd}(a,b)\big)$$

In this formula we have defined the initial values of $x$ and $y$ as $a$ and $b$. In order to establish the invariant at the beginning we have to ensure that the function `gcd` is only called with positive natural numbers. If we denote these numbers as $a$ and $b$, then the invariant $I$ is valid initially. The reason is that $x = a$ and $y = b$ implies $\texttt{gcd}(x,y) = \texttt{gcd}(a,b)$.

In order to prove that the invariant $I$ is maintained in the loop we formulate the Hoare triples for both alternatives of the conditional. For the first conditional we know that

$$\big\{I \wedge x \neq y \wedge x < y\big\} \quad \texttt{y := y - x;} \quad \big\{(I \wedge x \neq y \wedge x < y)\sigma\big\}$$

holds, where $\sigma$ is defined as $\sigma = [y \mapsto y + x]$. Here, the condition $x \neq y$ is the condition controlling the execution of the `while` loop and the condition $x < y$ is the condition of the `if` conditional. We

rewrite the formula $(I \wedge x \neq y \wedge x < y)\sigma$:

$$\begin{aligned}
& (I \wedge x \neq y \wedge x < y)\sigma \\
\leftrightarrow\ & (I \wedge x < y)\sigma \qquad \text{because } x < y \text{ implies } x \neq y \\
\leftrightarrow\ & (x > 0 \wedge y > 0 \wedge \mathtt{gcd}(x,y) = \mathtt{gcd}(a,b) \wedge x < y)[y \mapsto y + x] \\
\leftrightarrow\ & x > 0 \wedge y + x > 0 \wedge \mathtt{gcd}(x, y + x) = \mathtt{gcd}(a,b) \wedge x < y + x \\
\leftrightarrow\ & x > 0 \wedge y + x > 0 \wedge \mathtt{gcd}(x,y) = \mathtt{gcd}(a,b) \wedge 0 < y
\end{aligned}$$

In the last step we have used the formula

$$\mathtt{gcd}(x, y + x) = \mathtt{gcd}(x,y)$$

and we have simplified the inequality $x < y + x$ as $0 < y$. The last formula implies

$$x > 0 \wedge y > 0 \wedge \mathtt{gcd}(x,y) = \mathtt{gcd}(a,b).$$

However, this is precisely the invariant $I$. Therefore we have shown that

$$\{I \wedge x \neq y \wedge x < y\} \quad \texttt{y := y - x;} \quad \{I\} \tag{3.8}$$

holds. Next, let us consider the second alternative of the `if` conditional. We have

$$\{I \wedge x \neq y \wedge x \geq y\} \quad \texttt{x := x - y;} \quad \{(I \wedge x \neq y \wedge x \geq y)\sigma\}$$

where $\sigma = [x \mapsto x + y]$. The expression $(I \wedge x \neq y \wedge x \geq y)\sigma$ is rewritten as follows:

$$\begin{aligned}
& (I \wedge x \neq y \wedge x \geq y)\sigma \\
\leftrightarrow\ & (I \wedge x > y)\sigma \\
\leftrightarrow\ & (x > 0 \wedge y > 0 \wedge \mathtt{gcd}(x,y) = \mathtt{gcd}(a,b) \wedge x > y)[x \mapsto x + y] \\
\leftrightarrow\ & x + y > 0 \wedge y > 0 \wedge \mathtt{gcd}(x + y, y) = \mathtt{gcd}(a,b) \wedge x + y > y \\
\leftrightarrow\ & x + y > 0 \wedge y > 0 \wedge \mathtt{gcd}(x,y) = \mathtt{gcd}(a,b) \wedge x > 0
\end{aligned}$$

The last formula implies that

$$x > 0 \wedge y > 0 \wedge \mathtt{gcd}(x,y) = \mathtt{gcd}(a,b).$$

holds. Again, this is our invariant $I$. Therefore we have shown that

$$\{I \wedge x \neq y \wedge x \geq y\} \quad \texttt{x := x - y;} \quad \{I\} \tag{3.9}$$

holds. If we use the Hoare triples (3.8) and (3.9) as premisses for the rule for conditionals we have shown that

$$\{I \wedge x \neq y\} \quad \texttt{if (x < y) \{ y := y - x; \} else \{ x := x - y; \}} \quad \{I\}$$

holds. Now the verification rule for `while` loops yields

$$\{I\}$$

```
while (x != y ) {
     if (x < y) { y := y - x; } else { x := x - y;}
}
```

$$\{I \wedge x = y\}.$$

Expanding the invariant $I$ in the formula $I \wedge x = y$ shows that the postcondition of the `while` loop is given as

$$x > 0 \wedge y > 0 \wedge \mathtt{gcd}(x,y) = \mathtt{gcd}(a,b) \wedge x = y.$$

Now the correctness of the Euclidean algorithm can be established as follows:

$$\begin{aligned} & x > 0 \wedge y > 0 \wedge \texttt{gcd}(x,y) = \texttt{gcd}(a,b) \wedge x = y \\ \Rightarrow \quad & \texttt{gcd}(x,y) = \texttt{gcd}(a,b) \wedge x = y \\ \Rightarrow \quad & \texttt{gcd}(x,x) = \texttt{gcd}(a,b) \\ \Rightarrow \quad & x = \texttt{gcd}(a,b) \qquad \text{because} \quad \texttt{gcd}(x,x) = x. \end{aligned}$$

All in all we have shown the following: If the `while` loop terminates, then the variable $x$ will be set to the greatest common divisor of $a$ and $b$, where $a$ and $b$ are the initial values of the variables $x$ and $y$. In order to finish our correctness proof we have to show that the `while` loop does indeed terminate for all choices of $a$ and $b$. To this end let us define the variable $s$ as follows:

$$s := x + y.$$

The variables $x$ and $y$ are natural numbers. Therefore $s$ is a natural number, too. Every iteration of the loop reduces the number $s$: either $x$ is subtracted from $s$ or $y$ is subtracted from $s$ and the invariant $I$ shows that both $x$ and $y$ are positive. Therefore, if the `while` loop would run forever, at some point $s$ would get negative. Since $s$ can not be negative, the loop must terminate. Hence we have shown the correctness of the Euclidean algorithm.

**Exercise 7**: Show that the function `power`$(x, y)$ that is defined in Figure 3.9 does compute $x^y$, i. e. show that `power`$(x, y) = x^y$ for all natural numbers $x$ and $y$.

```
1  power := procedure(x, y) {
2      r := 1;
3      while (y > 0) {
4          if (y % 2 == 1) {
5              r := r * x;
6          }
7          x := x * x;
8          y := y \ 2;
9      }
10     return r;
11 };
```

Figure 3.9: A program to compute $x^y$ iteratively.

**Hints**:

1. If the initial values of $x$ and $y$ are called $a$ and $b$, then an invariant for the `while` loop is given as
   $$I := \left(r \cdot x^y = a^b\right).$$
2. The verification rule for the conditional without `else` is given as
   $$\frac{\{F \wedge B\} \quad \texttt{P} \quad \{G\}, \qquad F \wedge \neg B \rightarrow G}{\{F\} \quad \texttt{if (}B\texttt{) \{ P \}} \quad \{G\}}$$
   This rule is interpreted as follows:
   (a) If both the precondition $F$ and the condition $B$ is valid, then execution of the program fragment $P$ has to establish the validity of the postcondition $G$.
   (b) If the precondition $F$ is valid but we have $\neg B$, then this must imply the postcondition $G$.

**Bemerkung**: Proving the correctness of a nontrivial program is very tedious. Therefore, various attempts have been made to automate the task. For example, *KeY Hoare* is a tool that can be used to verify the correctness of programs. It is based on Hoare calculus.

## 3.5 Symbolic Program Execution

The last section has shown that using Hoare logic to verify a program can be quite difficult. There is another method to prove the correctness of imperative programs. This method is called *symbolic program execution*. Let us demonstrate this method. Consider the program shown in Figure 3.10.

The main difference between a mathematical formula and a program is that in a formula all occurrences of a variable refer to the same value. This is different in a program because the variables change their values dynamically. In order to deal with this property of program variables we have

```
1   power := procedure(x_0, y_0) {
2       r_0 := 1;
3       while (y_n > 0) {
4           if (y_n % 2 == 1) {
5               r_{n+1} := r_n * x_n;
6           }
7           x_{n+1} := x_n * x_n;
8           y_{n+1} := y_n \ 2;
9       }
10      return r_N;
11  };
```

Figure 3.10: An annotated programm to compute powers.

to be able to distinguish the different occurrences of a variable. To this end, we index the program variables. When doing this we have to be aware of the fact that the same occurrence of a program variable can still denote different values if the variable occurs inside a loop. In this case we have to index the variables in a way that the index includes a counter that counts the number of loop iterations. For concreteness, consider the program shown in Figure 3.10. Here, in line 5 the variable `r` has the index $n$ on the right side of the assignment, while it has the index $r_{n+1}$ on the left side of the assignment in line 5. Here, $n$ denotes the number of times the `while` loop has been iterated. After the loop in line 10 the variable is indexed as $\mathtt{r}_N$, where $N$ denotes the total number of loop iterations. We show the correctness of the given program next. Let us define

$$a := x_0, \quad b := y_0.$$

We show, that the `while` loop satisfies the invariant

$$r_n \cdot x_n^{y_n} = a^b. \tag{3.10}$$

This claim is proven by induction on the number of loop iterations.

B.C. $n = 0$: Since we have $r_0 = 1$, $x_0 = a$, and $y_0 = b$ we have

$$r_n \cdot x_n^{y_n} = r_0 \cdot x_0^{y_0} = 1 \cdot a^b = a^b.$$

I.S. $n \mapsto n + 1$: We need a case distinction with respect to $y \mod 2$:

(a) $y_n \mod 2 = 1$. Then we have $y_n = 2 \cdot (y_n \backslash 2) + 1$ and $r_{n+1} = r_n \cdot x_n$. Hence

$$\begin{aligned} & r_{n+1} \cdot x_{n+1}^{y_{n+1}} \\ = \; & (r_n \cdot x_n) \cdot (x_n \cdot x_n)^{y_n \backslash 2} \\ = \; & r_n \cdot x_n^{2 \cdot (y_n \backslash 2) + 1} \\ = \; & r_n \cdot x_n^{y_n} \\ \overset{i.h.}{=} \; & a^b \end{aligned}$$

(b) $y_n \mod 2 = 0$. Then we have $y_n = 2 \cdot (y_n \backslash 2)$ and $r_{n+1} = r_n$. Therefore

$$\begin{aligned} & r_{n+1} \cdot x_{n+1}^{y_{n+1}} \\ = \;& r_n \cdot (x_n \cdot x_n)^{y_n \backslash 2} \\ = \;& r_n \cdot x_n^{2 \cdot (y_n \backslash 2)} \\ = \;& r_n \cdot x_n^{y_n} \\ \overset{i.h.}{=} \;& a^b \end{aligned}$$

This shows the validity of the equation (3.10). If the `while` loop terminates, we must have $y_N = 0$. If $n = N$, then equation (3.10) yields:

$$r_N \cdot x_N^{y_N} = x_0^{y_0} \iff r_N \cdot x_N^0 = a^b \iff r_N \cdot 1 = a^b \iff r_N = a^b$$

This shows $r_N = a^b$ and since we already know that the `while` loop terminates, we have proven that $\texttt{power}(a, b) = a^b$.

**Exercise 8**: Use the method of symbolic program execution to prove the correctness of the implementation of the Euclidean algorithm that is shown in Figure 3.11. During the proof you should make use of the fact that for all positive natural numbers $a$ and $b$ the equation

$$\texttt{gcd}(a, b) = \texttt{gcd}(a \,\%\, b, b)$$

is valid.

```
1  gcd := procedure(a, b) {
2      while (b != 0) {
3          [a, b] := [b, a % b];
4      }
5      return a;
6  };
```

Figure 3.11: An efficient version of the Euclidean algorithm.

# Chapter 4

# Aussagenlogik

## 4.1 Überblick

Die Aussagenlogik beschäftigt sich mit der Verknüpfung einfacher Aussagen durch Junktoren. Dabei sind Junktoren Worte wie "und", "oder", "nicht", "wenn $\cdots$, dann", und "genau dann, wenn". Unter einfachen Aussagen verstehen wir Sätze, die

- einen Tatbestand ausdrücken, der entweder wahr oder falsch ist und
- selber keine Junktoren enthalten.

Beispiele für einfache Aussagen sind

1. "*Die Sonne scheint.*"
2. "*Es regnet.*"
3. "*Am Himmel ist ein Regenbogen.*"

Einfache Aussagen dieser Art bezeichnen wir auch als atomare Aussagen, weil sie sich nicht weiter in Teilaussagen zerlegen lassen. Atomare Aussagen lassen sich mit Hilfe der eben angegebenen Junktoren zu zusammengesetzten Aussagen verknüpfen. Ein Beispiel für eine zusammengesetzte Aussage wäre

*Wenn die Sonne scheint und es regnet, dann ist ein Regenbogen am Himmel.* (1)

Die Aussage ist aus den drei atomaren Aussagen "*Die Sonne scheint.*", "*Es regnet.*", und "*Am Himmel ist ein Regenbogen.*" mit Hilfe der Junktoren "*und*" und "*wenn $\cdots$, dann*" zusammen gesetzt worden. Die Aussagenlogik untersucht, wie sich der Wahrheitswert zusammengesetzter Aussagen aus dem Wahrheitswert der einzelnen Teilaussagen berechnen lässt. Darauf aufbauend wird dann gefragt, in welcher Art und Weise wir aus gegebenen Aussagen neue Aussagen folgern können.

Um die Struktur komplexerer Aussagen übersichtlich darstellen zu können, führen wir in der Aussagenlogik zunächst sogenannte Aussage-Variablen ein. Diese Variablen sind Namen, die für atomare Aussagen stehen. Zusätzlich führen wir für die Junktoren "nicht", "und", "oder", "wenn, $\cdots$ dann", und "genau dann, wenn" die Symbole "$\neg$", "$\wedge$", "$\vee$", "$\rightarrow$" und "$\leftrightarrow$" als Abkürzungen ein:

1. $\neg a$ steht für nicht $a$

2. $a \wedge b$ steht für $a$ und $b$
3. $a \vee b$ steht für $a$ oder $b$
4. $a \rightarrow b$ steht für wenn $a$, dann $b$
5. $a \leftrightarrow b$ steht für $a$ genau dann, wenn $b$

Aussagenlogische Formeln werden aus Aussage-Variablen mit Hilfe von Junktoren aufgebaut und können beliebig komplex sein. Die Aussage (1) können wir mit Hilfe der Junktoren kürzer als

$$\texttt{SonneScheint} \wedge \texttt{EsRegnet} \rightarrow \texttt{Regenbogen}$$

schreiben. Hier haben wir `SonneScheint`, `EsRegnet` und `Regenbogen` als Aussage-Variablen verwendet. Durch die Benutzung der Junktoren wird die logische Struktur der Aussage klarer. Bestimmte aussagenlogische Formeln sind offenbar immer wahr, egal was wir für die einzelnen Teilaussagen einsetzen. Beispielsweise ist eine Formel der Art

$$p \vee \neg p$$

unabhängig von dem Wahrheitswert der Aussage $p$ immer wahr. Eine aussagenlogische Formel, die immer wahr ist, bezeichnen wir als eine Tautologie. Andere aussagenlogische Formeln sind nie wahr, beispielsweise ist die Formel

$$p \wedge \neg p$$

immer falsch. Eine Formel heißt erfüllbar, wenn es wenigstens eine Möglichkeit gibt, dass die Formel wahr wird, ansonsten heißt sie unerfüllbar Im Rahmen der Vorlesung werden wir verschiedene Verfahren entwickeln, mit denen es möglich ist zu entscheiden, ob eine aussagenlogische Formel eine Tautologie ist oder ob sie erfüllbar ist. Solche Verfahren spielen in der Praxis eine wichtige Rolle.

Der Rest dieses Kapitels ist wir folgt strukturiert:

1. Wir diskutieren Anwendungen der Aussagenlogik.
2. Danach definieren wir formal, wie aussagenlogische Formeln aufgebaut sind.

   Formal sprechen wir in diesem Zusammenhang von der Syntax aussagenlogische Formeln.
3. Anschließend diskutieren wir die Auswertung aussagenlogischer Formeln. Wir legen also fest, welchen Wahrheitswert eine aussagenlogische Formel unter einer gegebenen Belegung annimmt. Wir zeigen in diesem Zusammenhang auch, wie die Auswertung aussagenlogischer Formeln sich in *Python* implementieren lässt.

   Hier sprechen wir von der Semantik der aussagenlogischen Formeln.
4. Danach führen wir die Begriffe der Tautologie und der Erfüllbarkeit einer aussagenlogischen Formel formal ein.
   (a) Eine aussagenlogische Formel ist eine Tautologie genau dann, wenn Sie unter jeder möglichen Interpretation der aussagenlogischen Variablen wahr ist.
   (b) Eine aussagenlogischen Formel ist erfüllbar genau dann, wenn es eine Interpretation ihrer aussagenlogischen Variablen gibt, unter der die Formel wahr wird.
5. Wir diskutieren, wie sich aussagenlogische Formeln algebraisch umformen lassen und diskutieren in diesem Zusammenhang den Begriff der konjunktiven Normalform. Formeln, die in konjunktiver Normalform vorliegen, lassen sich auf dem Rechner leichter verarbeiten.

6. Anschließend diskutieren wir den Herleitungs-Begriff. Dabei geht es darum, aus Formeln, die als Axiome vorgegebenen sind, neue Formeln herzuleiten.

7. Schließlich stellen wir das Verfahren von Davis und Putnam vor. Mit diesem Verfahren lässt sich überprüfen, ob eine aussagenlogische Formel erfüllbar ist. Als Anwendung zeigen wir, wie das 8-Damen-Problem mit Hilfe des Verfahrens von Davis und Putnam gelöst werden kann.

## 4.2 Anwendungen der Aussagenlogik

Die Aussagenlogik bildet nicht nur die Grundlage für die Prädikatenlogik, die wir im nächsten Kapitel diskutieren werden, sondern sie hat auch wichtige praktische Anwendungen. Aus der großen Zahl der industriellen Anwendungen möchte ich stellvertretend vier Beispiele nennen:

1. Analyse und Design digitaler Schaltungen.

   Komplexe digitale Schaltungen bestehen heute aus Milliarden von logischen Gattern.[1] Ein Gatter ist dabei, aus logischer Sicht betrachtet, ein Baustein, der einen der logischen Junktoren wie "und", "oder", "nicht", etc. auf elektronischer Ebene repräsentiert.

   Die Komplexität solcher Schaltungen wäre ohne den Einsatz rechnergestützter Verfahren zur Verifikation nicht mehr beherrschbar. Die dabei eingesetzten Verfahren sind Anwendungen der Aussagenlogik.

   Eine ganz konkrete Anwendung ist der Schaltungs-Vergleich. Hier werden zwei digitale Schaltungen als aussagenlogische Formeln dargestellt. Anschließend wird versucht, mit aussagenlogischen Mitteln die Äquivalenz dieser Formeln zu zeigen. Software-Werkzeuge, die für die Verifikation digitaler Schaltungen eingesetzt werden, kosten zum Teil mehr als 100 000 $. Die Firma Magma bietet beispielsweise den Equivalence-Checker Quartz Formal zum Preis von 150 000 $ pro Lizenz an. Eine solche Lizenz ist dann drei Jahre lang gültig.

2. Erstellung von Verschlussplänen für die Weichen und Signale von Bahnhöfen.

   Bei einem größeren Bahnhof gibt es einige hundert Weichen und Signale, die ständig neu eingestellt werden müssen, um für die Züge sogenannte Fahrstraßen zu realisieren. Verschiedene Fahrstraßen dürfen sich aus Sicherheitsgründen nicht kreuzen. Die einzelnen Fahrstraßen werden durch sogenannte Verschlusspläne beschrieben. Die Korrektheit solcher Verschlusspläne kann durch aussagenlogische Formeln ausgedrückt werden.

3. Eine Reihe kombinatorischer Puzzles lassen sich als aussagenlogische Formeln kodieren und können dann mit Hilfe aussagenlogischer Methoden gelöst werden. Als ein Beispiel werden wir in der Vorlesung das 8-Damen-Problem behandeln. Dabei geht es um die Frage, ob 8 Damen so auf einem Schachbrett angeordnet werden können, dass keine der Damen eine andere Dame bedroht.

## 4.3 Formale Definition der aussagenlogischen Formeln

Wir behandeln zunächst die Syntax der Aussagenlogik und besprechen anschließend die Semantik. Die Syntax gibt an, wie Formeln geschrieben werden und wie sich Formeln zu Beweisen verknüpfen

[1] Die Seite `https://en.wikipedia.org/wiki/Transistor_count` gibt einen Überblick über die Komplexität moderner Prozessoren.

lassen. Die Semantik befasst sich mit der Bedeutung der Formeln. Nachdem wir die Semantik der aussagenlogischen Formeln mit Hilfe der Mengenlehre definiert haben, zeigen wir anschließend, wie sich diese Semantik in *Python* implementieren lässt.

### 4.3.1 Syntax der aussagenlogischen Formeln

In diesem Abschnitt legen wir fest, was aussagenlogische Formeln sind: Dazu werden wir aussagenlogische Formeln als Menge von Strings definieren, wobei die Strings in der Menge bestimmte Eigenschaften haben müssen, damit wir von aussagenlogischen Formeln sprechen können.

Zunächst betrachten wir eine Menge $\mathcal{P}$ von sogenannten Aussage-Variablen als gegeben. Typischerweise besteht $\mathcal{P}$ aus der Menge der kleinen lateinischen Buchstaben, die zusätzlich noch indiziert sein dürfen. Beispielsweise werden wir

$$p,\ q,\ r,\ p_1,\ p_2,\ p_3$$

als Aussage-Variablen verwenden. Aussagenlogische Formeln sind dann Wörter, die aus dem Alphabet

$$\mathcal{A} := \mathcal{P} \cup \big\{ \top, \bot, \neg, \vee, \wedge, \rightarrow, \leftrightarrow, (, ) \big\}$$

gebildet werden. Wir definieren die Menge der aussagenlogischen Formeln $\mathcal{F}$ durch Induktion:

1. $\top \in \mathcal{F}$ und $\bot \in \mathcal{F}$.

   Hier steht $\top$ für die Formel, die immer wahr ist, während $\bot$ für die Formel steht, die immer falsch ist. Die Formel $\top$ trägt den Namen Verum[2], für $\bot$ sagen wir Falsum[3].

2. Ist $p \in \mathcal{P}$, so gilt auch $p \in \mathcal{F}$.

   Jede aussagenlogische Variable ist also auch eine aussagenlogische Formel.

3. Ist $f \in \mathcal{F}$, so gilt auch $\neg f \in \mathcal{F}$.

   Die Formel $\neg f$ bezeichnen wir auch als die Negation von $f$.

4. Sind $f_1, f_2 \in \mathcal{F}$, so gilt auch

   | | | | | |
   |---|---|---|---|---|
   | $(f_1 \vee f_2) \in \mathcal{F}$ | (*gelesen*: | $f_1$ oder $f_2$ | auch: Disjunktion | von $f_1$ und $f_2$), |
   | $(f_1 \wedge f_2) \in \mathcal{F}$ | (*gelesen*: | $f_1$ und $f_2$ | auch: Konjunktion | von $f_1$ und $f_2$), |
   | $(f_1 \rightarrow f_2) \in \mathcal{F}$ | (*gelesen*: | wenn $f_1$, dann $f_2$ | auch: Implikation | von $f_1$ und $f_2$), |
   | $(f_1 \leftrightarrow f_2) \in \mathcal{F}$ | (*gelesen*: | $f_1$ genau dann, wenn $f_2$ | auch: Bikonditional | von $f_1$ und $f_2$). |

Die Menge $\mathcal{F}$ der aussagenlogischen Formeln ist nun die kleinste Teilmenge der aus dem Alphabet $\mathcal{A}$ gebildeten Wörter, welche die oben aufgelisteten Abschluss-Eigenschaften hat.

**Beispiel**: Es sei $\mathcal{P} := \{p, q, r\}$. Dann gilt:

1. $p \in \mathcal{F}$,
2. $(p \wedge q) \in \mathcal{F}$,
3. $\Big( \big( (\neg p \rightarrow q) \vee (q \rightarrow \neg p) \big) \rightarrow r \Big) \in \mathcal{F}$. □

[2] Verum ist das lateinische Wort für "wahr".
[3] Falsum ist das lateinische Wort für "falsch".

Um Klammern zu sparen, vereinbaren wir die folgenden Regeln:

1. Äußere Klammern werden weggelassen, wir schreiben also beispielsweise

   $$p \wedge q \quad \text{statt} \quad (p \wedge q).$$

2. Der Junktor $\neg$ bindet stärker als alle anderen Junktoren.

3. Die Junktoren $\vee$ und $\wedge$ werden implizit links geklammert, d.h. wir schreiben

   $$p \wedge q \wedge r \quad \text{statt} \quad (p \wedge q) \wedge r.$$

   Operatoren, die implizit nach links geklammert werden, nennen wir links-assoziativ.

   Beachten Sie, dass wir für diese Vorlesung vereinbaren, dass die Junktoren $\wedge$ und $\vee$ dieselbe Bindungsstärke haben. Das ist anders als in der Sprache *Python*, denn dort bindet der Operator "`and`" stärker als der Operator "`or`". In den Sprachen `C` und *Java* bindet der Operator "`&&`" ebenfalls stärker als der Operator "`||`".

4. Der Junktor $\rightarrow$ wird implizit rechts geklammert, d.h. wir schreiben

   $$p \rightarrow q \rightarrow r \quad \text{statt} \quad p \rightarrow (q \rightarrow r).$$

   Operatoren, die implizit nach rechts geklammert werden, nennen wir rechts-assoziativ.

5. Die Junktoren $\vee$ und $\wedge$ binden stärker als $\rightarrow$, wir schreiben also

   $$p \wedge q \rightarrow r \quad \text{statt} \quad (p \wedge q) \rightarrow r.$$

6. Der Junktor $\rightarrow$ bindet stärker als $\leftrightarrow$, wir schreiben also

   $$p \rightarrow q \leftrightarrow r \quad \text{statt} \quad (p \rightarrow q) \leftrightarrow r.$$

7. Beachten Sie, dass der Junktor $\leftrightarrow$ weder rechts- noch links-assoziativ ist. Daher ist ein Ausdruck der Form

   $$p \leftrightarrow q \leftrightarrow r$$

   undefiniert und muss geklammert werden. Wenn Sie eine solche Formel in einem Buch sehen, ist dies in der Regel als Abkürzung für die Formel

   $$(p \leftrightarrow q) \wedge (q \leftrightarrow r)$$

   zu verstehen. Wir werden diese Form der Abkürzung aber nicht verwenden.

**Bemerkung**: Wir werden im Rest dieser Vorlesung eine Reihe von Beweisen führen, bei denen es darum geht, mathematische Aussagen über Formeln nachzuweisen. Bei diesen Beweisen werden wir natürlich ebenfalls aussagenlogische Junktoren verwenden. Dabei entsteht dann die Gefahr, dass wir die Junktoren, die wir in unseren Beweisen verwenden, mit den Junktoren, die in den aussagenlogischen Formeln auftreten, verwechseln. Um dieses Problem zu umgehen vereinbaren wir:

1. Innerhalb einer aussagenlogischen Formel wird der Junktor "nicht" als "$\neg$" geschrieben.

   Bei den Beweisen, die wir über aussagenlogische Formeln führen, schreiben wir den Junktor statt dessen als das Wort "nicht".

2. Innerhalb einer aussagenlogischen Formel wird der Junktor "und" als "$\wedge$" geschrieben.

   Bei den Beweisen, die wir über aussagenlogische Formeln führen, verwenden wir stattdessen das Wort "und".

3. Innerhalb einer aussagenlogischen Formel wird der Junktor "oder" als "$\vee$" geschrieben.

   Bei den Beweisen, die wir über aussagenlogische Formeln führen, verwenden wir stattdessen das Wort "oder".

4. Innerhalb einer aussagenlogischen Formel wird der Junktor "wenn $\cdots$, dann" als "$\rightarrow$" geschrieben.

   Bei den Beweisen, die wir über aussagenlogische Formeln führen, verwenden wir stattdessen das Symbol "$\Rightarrow$".

5. Analog wird der Junktor "genau dann, wenn" innerhalb einer aussagenlogischen Formel als "$\leftrightarrow$" geschrieben, aber wenn wir diesen Junktor als Teil eines Beweises verwenden, schreiben wir stattdessen "$\Leftrightarrow$". ◇

### 4.3.2 Semantik der aussagenlogischen Formeln

In diesem Abschnitt definieren wir die Semantik, also die Bedeutung, aussagenlogischer Formeln. Wir legen also die Interpretation oder auch Bedeutung dieser Formeln fest. Dazu ordnen wir den aussagenlogischen Formeln Wahrheitswerte zu. Damit dies möglich ist, definieren wir zunächst die Menge $\mathbb{B}$ der Wahrheitswerte:

$$\mathbb{B} := \{\texttt{True}, \texttt{False}\}.$$

Damit können wir nun den Begriff einer aussagenlogischen Interpretation festlegen.

**Definition 6 (Aussagenlogische Interpretation)** *Eine aussagenlogische Interpretation ist eine Funktion*

$$\mathcal{I} : \mathcal{P} \to \mathbb{B},$$

*die jeder Aussage-Variablen $p \in \mathcal{P}$ einen Wahrheitswert $\mathcal{I}(p) \in \mathbb{B}$ zuordnet.* ◇

Eine aussagenlogische Interpretation wird oft auch als Belegung der Aussage-Variablen mit Wahrheits-Werten bezeichnet.

Eine aussagenlogische Interpretation $\mathcal{I}$ interpretiert die Aussage-Variablen. Um nicht nur Variablen sondern auch aussagenlogische Formeln interpretieren zu können, benötigen wir eine Interpretation der Junktoren "$\neg$", "$\wedge$", "$\vee$", "$\rightarrow$" und "$\leftrightarrow$". Zu diesem Zweck definieren wir auf der Menge $\mathbb{B}$ der Wahrheits-Werte Funktionen $\ominus$, $\oslash$, $\oplus$, $\oplus$ und $\ominus$, mit deren Hilfe wir die aussagenlogischen Junktoren interpretieren können:

1. $\ominus : \mathbb{B} \to \mathbb{B}$
2. $\oslash : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$
3. $\oplus : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$
4. $\oplus : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$
5. $\ominus : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

Wir könnten die Funktionen $\ominus$, $\oslash$, $\oplus$, $\oplus$ und $\ominus$ am einfachsten durch die folgende Wahrheits-Tafel (Tabelle 4.1) definieren:

| $p$ | $q$ | $\ominus(p)$ | $\oslash(p,q)$ | $\otimes(p,q)$ | $\ominus(p,q)$ | $\oplus(p,q)$ |
|---|---|---|---|---|---|---|
| True | True | False | True | True | True | True |
| True | False | False | True | False | False | False |
| False | True | True | True | False | True | False |
| False | False | True | False | False | True | True |

Table 4.1: Interpretation der Junktoren

Nun können wir den Wahrheits-Wert, den eine aussagenlogische Formel $f$ unter einer gegebenen aussagenlogischen Interpretation $\mathcal{I}$ annimmt, durch Induktion nach dem Aufbau der Formel $f$ definieren. Wir werden diesen Wert mit $\widehat{\mathcal{I}}(f)$ bezeichnen. Wir setzen:

1. $\widehat{\mathcal{I}}(\bot) := \texttt{False}$.
2. $\widehat{\mathcal{I}}(\top) := \texttt{True}$.
3. $\widehat{\mathcal{I}}(p) := \mathcal{I}(p)$ für alle $p \in \mathcal{P}$.
4. $\widehat{\mathcal{I}}(\neg f) := \ominus\big(\widehat{\mathcal{I}}(f)\big)$ für alle $f \in \mathcal{F}$.
5. $\widehat{\mathcal{I}}(f \wedge g) := \otimes\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ für alle $f, g \in \mathcal{F}$.
6. $\widehat{\mathcal{I}}(f \vee g) := \oslash\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ für alle $f, g \in \mathcal{F}$.
7. $\widehat{\mathcal{I}}(f \to g) := \ominus\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ für alle $f, g \in \mathcal{F}$.
8. $\widehat{\mathcal{I}}(f \leftrightarrow g) := \oplus\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ für alle $f, g \in \mathcal{F}$.

Um die Schreibweise nicht übermäßig kompliziert werden zu lassen, unterscheiden wir in Zukunft nicht mehr zwischen der Funktion $\widehat{\mathcal{I}}$ und der Funktion $\mathcal{I}$, wir werden das Hütchen über dem $\mathcal{I}$ also weglassen.

**Beispiel**: Wir zeigen, wie sich der Wahrheits-Wert der Formel

$$(p \to q) \to (\neg p \to q) \to q$$

für die aussagenlogische Interpretation $\mathcal{I}$, die durch $\mathcal{I}(p) = \texttt{True}$ und $\mathcal{I}(q) = \texttt{False}$ definiert ist, berechnen lässt:

$$\begin{aligned}
\mathcal{I}\Big((p \to q) \to (\neg p \to q) \to q\Big) &= \ominus\Big(\mathcal{I}\big((p \to q)\big), \mathcal{I}\big((\neg p \to q) \to q\big)\Big) \\
&= \ominus\Big(\ominus\big(\mathcal{I}(p), \mathcal{I}(q)\big), \mathcal{I}\big((\neg p \to q) \to q\big)\Big) \\
&= \ominus\Big(\ominus\big(\texttt{True}, \texttt{False}\big), \mathcal{I}\big((\neg p \to q) \to q\big)\Big) \\
&= \ominus\Big(\texttt{False}, \mathcal{I}\big((\neg p \to q) \to q\big)\Big) \\
&= \texttt{True} \qquad \diamond
\end{aligned}$$

Beachten Sie, dass wir bei der Berechnung gerade soviele Teile der Formel ausgewertet haben, wie notwendig waren, um den Wert der Formel zu bestimmen. Trotzdem ist die eben durchgeführte

Rechnung für die Praxis zu umständlich. Stattdessen wird der Wert einer Formel direkt mit Hilfe der Tabelle 4.1 auf Seite 40 berechnet. Wir zeigen exemplarisch, wie wir den Wahrheits-Wert der Formel

$$(p \to q) \to (\neg p \to q) \to q$$

für beliebige Belegungen $\mathcal{I}$ über diese Tabelle berechnen können. Um nun die Wahrheitswerte dieser Formel unter einer gegebenen Belegung der Aussage-Variablen bestimmen zu können, bauen wir eine Tabelle auf, die für jede in der Formel auftretende Teilformel eine Spalte enthält. Tabelle 4.2 auf Seite 41 zeigt die entstehende Tabelle.

| $p$ | $q$ | $\neg p$ | $p \to q$ | $\neg p \to q$ | $(\neg p \to q) \to q$ | $(p \to q) \to (\neg p \to q) \to q$ |
|---|---|---|---|---|---|---|
| `True` | `True` | `False` | `True` | `True` | `True` | `True` |
| `True` | `False` | `False` | `False` | `True` | `False` | `True` |
| `False` | `True` | `True` | `True` | `True` | `True` | `True` |
| `False` | `False` | `True` | `True` | `False` | `True` | `True` |

Table 4.2: Berechnung der Wahrheitswerte von $(p \to q) \to (\neg p \to q) \to q$

Betrachten wir die letzte Spalte der Tabelle, so sehen wir, dass dort immer der Wert `True` auftritt. Also liefert die Auswertung der Formel $(p \to q) \to (\neg p \to q) \to q$ für jede aussagenlogische Belegung $\mathcal{I}$ den Wert `True`. Eine Formel, die immer wahr ist, wird als Tautologie bezeichnet.

Wir erläutern die Aufstellung dieser Tabelle anhand der zweiten Zeile. In dieser Zeile sind zunächst die aussagenlogischen Variablen $p$ auf `True` und $q$ auf `False` gesetzt. Bezeichnen wir die aussagenlogische Interpretation mit $\mathcal{I}$, so gilt also

$$\mathcal{I}(p) = \texttt{True} \text{ und } \mathcal{I}(q) = \texttt{False}.$$

Damit erhalten wir folgende Rechnung:

1. $\mathcal{I}(\neg p) = \ominus(\mathcal{I}(p)) = \ominus(\texttt{True}) = \texttt{False}$
2. $\mathcal{I}(p \to q) = \ominus\!\!\to(\mathcal{I}(p), \mathcal{I}(q)) = \ominus\!\!\to(\texttt{True}, \texttt{False}) = \texttt{False}$
3. $\mathcal{I}(\neg p \to q) = \ominus\!\!\to(\mathcal{I}(\neg p), \mathcal{I}(q)) = \ominus\!\!\to(\texttt{False}, \texttt{False}) = \texttt{True}$
4. $\mathcal{I}\big((\neg p \to q) \to q\big) = \ominus\!\!\to\big(\mathcal{I}(\neg p \to q), \mathcal{I}(q)\big) = \ominus\!\!\to(\texttt{True}, \texttt{False}) = \texttt{False}$
5. $\mathcal{I}\big((p \to q) \to (\neg p \to q) \to q\big) = \ominus\!\!\to\big(\mathcal{I}(p \to q), \mathcal{I}((\neg p \to q) \to q)\big) = \ominus\!\!\to(\texttt{False}, \texttt{False}) = \texttt{True}$

Für komplexe Formeln ist die Auswertung von Hand viel zu mühsam und fehleranfällig, um praktikabel zu sein. Wir zeigen deshalb später, wie sich dieser Prozess mit Hilfe von *Python* automatisieren lässt.

### 4.3.3 Extensionale und intensionale Interpretationen der Aussagenlogik

Die Interpretation der aussagenlogischen Junktoren ist rein extensional: Wenn wir den Wahrheitswert der Formel

$$\mathcal{I}(f \to g)$$

berechnen wollen, so müssen wir die Details der Teilformeln $f$ und $g$ nicht kennen, es reicht, wenn

wir die Werte $\mathcal{I}(f)$ und $\mathcal{I}(g)$ kennen. Das ist problematisch, denn in der Umgangssprache hat der Junktor "*wenn* $\cdots$*, dann*" auch eine kausale Bedeutung.

Obwohl der folgende Satz mit der extensionalen Implikation

"*Wenn* $3 \cdot 3 = 8$*, dann schneit es.*"

als wahr interpretiert wird, erscheint er in ausgesprochener Form sinnlos.

Insofern ist die extensionale Interpretation des sprachlichen Junktors "*wenn* $\cdots$*, dann*" nur eine Approximation der umgangssprachlichen Interpretation, die sich für die Mathematik und die Informatik aber als ausreichend erwiesen hat.

Es gibt durchaus auch andere Logiken, in denen die Interpretation des Operators "$\rightarrow$" von der hier gegebenen Definition abweicht. Solche Logiken werden als intensionale Logiken bezeichnet. Diese Logiken spielen zwar in der Informatik auch eine Rolle, aber da die Untersuchung intensionaler Logiken wesentlich aufwändiger ist als die Untersuchung der extensionalen Logik, werden wir uns auf die Analyse letzterer beschränken.

### 4.3.4 Implementierung in *Python*

Um die bisher eingeführten Begriffe nicht zu abstrakt werden zu lassen, entwickeln wir in *Python* ein Programm, mit dessen Hilfe sich Formeln auswerten lassen. Jedes Mal, wenn wir ein Programm zur Berechnung irgendwelcher Werte entwickeln wollen, müssen wir uns als erstes fragen, wie wir die Argumente der zu implementierenden Funktion und die Ergebnisse dieser Funktion in der verwendeten Programmier-Sprache darstellen können. In diesem Fall müssen wir uns also überlegen, wie wir eine aussagenlogische Formel in *Python* repräsentieren können, denn die Ergebniswerte `True` und `False` stehen ja als Wahrheitswerte unmittelbar zur Verfügung. Zusammengesetzte Daten-Strukturen können in *Python* am einfachsten als geschachtelte Tupel dargestellt werden und das ist auch der Weg, den wir für die aussagenlogischen Formeln beschreiten werden. Ein geschachteltes Tupel ist dabei ein Tupel, dass sowohl Strings als auch geschachtelte Tupel enthalten kann. Beispielsweise ist

```
('∧', ('¬', 'p'), 'q')
```

ein geschachteltes Tupel, das die aussagenlogische Formel $\neg p \wedge q$ repräsentiert.

Wir legen die Repräsentation von aussagenlogischen Formeln nun formal dadurch fest, dass wir eine Funktion

$$\mathit{rep} : \mathcal{F} \rightarrow \mathit{Python}$$

definieren, die einer aussagenlogischen Formel $f$ ein geschachteltes Tupel $\mathit{rep}(f)$ zuordnet. Diese Funktion definieren wir durch Induktion nach der Formel $f$:

1. $\top$ wird repräsentiert durch das Tupel `('⊤',)`. Dies ist möglich, da `'⊤'` ein Unicode-Zeichen ist und Python die Verwendung von Unicode-Zeichen in Strings erlaubt. Wir können diesen String alternativ in *Python* auch in der Form `'\N{up tack}'` schreiben, denn "`up tack`" ist der Name des Unicode-Zeichens "$\top$" und ein Unicode Zeichen, das den Namen $u$ hat, kann in *Python* als `'\N{`$u$`}'` geschrieben werden. Also haben wir

   $$\mathit{rep}(\top) := \texttt{('\N\{up tack\}',)}.$$

2. $\bot$ wird repräsentiert durch das Tupel `('⊥',)`. Das Unicode-Zeichen `'⊥'` trägt den Namen "`down tack`". Also haben wir

$$rep(\bot) := \texttt{('\N\{down tack\}',)}.$$

3. Da aussagenlogische Variablen nichts anderes als Strings sind, können wir eine aussagenlogische Variable durch sich selbst repräsentieren:

   $$rep(p) := p \quad \text{für alle } p \in \mathcal{P}.$$

4. Ist $f$ eine aussagenlogische Formel, so repräsentieren wir die Negation $\neg f$ als geschachteltes Tupel, bei dem wir das Unicode-Zeichen `'¬'` an die erste Stelle setzen und anschließend rekursiv die Formel $f$ in ihre *Python*-Repräsentierung umwandeln. Der Name des Unicode-Zeichen `'¬'` ist "`not sign`". Also haben wir

   $$rep(\neg f) := \big(\texttt{'¬'}, rep(f)\big).$$

5. Sind $f_1$ und $f_2$ aussagenlogische Formeln, so repräsentieren wir $f_1 \wedge f_2$ mit Hilfe des Unicode-Zeichens `'∧'`, das den Namen "`logical and`" hat:

   $$rep(f \wedge g) := \big(\texttt{'∧'}, rep(f), rep(g)\big).$$

6. Sind $f_1$ und $f_2$ aussagenlogische Formeln, so repräsentieren wir $f_1 \vee f_2$ mit Hilfe des Unicode-Zeichens `'∨'`, das den Namen "`logical or`" hat:

   $$rep(f \vee g) := \big(\texttt{'∨'}, rep(f), rep(g)\big).$$

7. Sind $f_1$ und $f_2$ aussagenlogische Formeln, so repräsentieren wir $f_1 \rightarrow f_2$ mit Hilfe des Unicode-Zeichens `'→'`, das den Namen "`rightwards arrow`" hat:

   $$rep(f \rightarrow g) := \big(\texttt{'→'}, rep(f), rep(g)\big).$$

8. Sind $f_1$ und $f_2$ aussagenlogische Formeln, so repräsentieren wir $f_1 \leftrightarrow f_2$ mit Hilfe des Unicode-Zeichens `'↔'`, das den Namen "`left right arrow`" hat:

   $$rep(f \leftrightarrow g) := \big(\texttt{'↔'}, rep(f), rep(g)\big).$$

Bei der Wahl der Repräsentation, mit der wir eine Formel in *Python* repäsentieren, sind wir weitgehend frei. Wir hätten oben sicher auch eine andere Repräsentation verwenden können. Eine gute Repräsentation sollte einerseits möglichst intuitiv sein, andererseits ist es auch wichtig, dass die Repräsentation für die zu entwickelnden Algorithmen adäquat ist. Im Wesentlichen heißt dies, dass es einerseits einfach sein sollte, auf die Komponenten einer Formel zuzugreifen, andererseits sollte es auch leicht sein, die entsprechende Repräsentation zu erzeugen.

Als nächstes geben wir an, wie wir eine aussagenlogische Interpretation in *Python* darstellen. Eine aussagenlogische Interpretation ist eine Funktion

$$\mathcal{I} : \mathcal{P} \rightarrow \mathbb{B}$$

von der Menge der Aussage-Variablen $\mathcal{P}$ in die Menge der Wahrheitswerte $\mathbb{B}$. Eine Möglichkeit eine solche aussagenlogische Interpretation darzustellen besteht darin, dass wir die Menge aller aussagenlogischen Variablen angeben, die unter der aussagenlogischen Interpretation $\mathcal{I}$ den Wert `True` annehmen:

$$rep(\mathcal{I}) := \big\{x \in \mathcal{P} \mid \mathcal{I}(x) = \texttt{True}\big\}.$$

Damit können wir jetzt eine einfache Funktion implementieren, die den Wahrheitswert einer aussagenlogischen Formel $f$ unter einer gegebenen aussagenlogischen Interpretation $\mathcal{I}$ berechnet. Die Funktion `evaluate` ist in Abbildung 4.1 auf Seite 44 gezeigt. Die Funktion `evaluate` erwartet zwei Argumente:

1. Das erste Argument $F$ ist eine aussagenlogische Formel, die als geschachteltes Tupel dargestellt wird.

2. Das zweite Argument $I$ ist eine aussagenlogische Interpretation, die als Menge von aussagenlogischen Variablen dargestellt wird. Für eine aussagenlogische Variable mit dem Namen $p$ können wir den Wert, der dieser Variablen durch $I$ zugeordnet wird, mittels des Ausdrucks "$p$ `in` $I$" berechnen.

```
1  def evaluate(F, I):
2      "Evaluate the propositional formula F using the interpretation I"
3      if isinstance(F, str): # F is a propositional variable
4          return F in I
5      if F[0] == '⊤': return True
6      if F[0] == '⊥': return False
7      if F[0] == '¬': return not evaluate(F[1], I)
8      if F[0] == '∧': return     evaluate(F[1], I) and evaluate(F[2], I)
9      if F[0] == '∨': return     evaluate(F[1], I) or  evaluate(F[2], I)
10     if F[0] == '→': return not evaluate(F[1], I) or  evaluate(F[2], I)
11     if F[0] == '↔': return     evaluate(F[1], I) ==  evaluate(F[2], I)
```

Figure 4.1: Auswertung einer aussagenlogischen Formel

Wir diskutieren jetzt die Implementierung der Funktion `evaluate()` Zeile für Zeile:

1. In Zeile 3 betrachten wir den Fall, dass das Argument $F$ eine aussagenlogische Variable repräsentiert. Dies erkennen wir daran, dass $F$ ein String ist. Die vordefinierte Funktion `isinstance` führt diese Überprüfung durch.

   In diesem Fall müssen wir wissen, ob die Variable $F$ ein Element der Menge $I$ ist, denn genau dann wird $F$ als wahr interpretiert.

2. Falls die Formel $F$ den Wert $\top$ hat, so ist das Ergebnis der Auswertung unabhängig von der aussagenlogischen Interpretation $I$ immer `True`. Um zu erkennen, ob $F$ die Formel $\top$ repräsentiert, betrachten wir die erste Komponente des Tupels $F$ und überprüfen also, ob $F[0]$ das Zeichen `'⊤'` ist.

3. Falls die Formel $F$ den Wert $\bot$ hat, so ist das Ergebnis der Auswertung unabhängig von der aussagenlogischen Interpretation $I$ immer `False`.

4. In Zeile 7 betrachten wir den Fall, dass $F$ die Form $\neg G$ hat. In diesem Fall werten wir erst $G$ unter der Belegung $I$ aus und negieren dann das Ergebnis.

5. In Zeile 8 betrachten wir den Fall, dass $F$ die Form $G_1 \wedge G_2$ hat. In diesem Fall werten wir zunächst $G_1$ und $G_2$ unter der Belegung $I$ aus und verknüpfen das Ergebnis mit dem Operator "`and`".

6. In Zeile 9 betrachten wir den Fall, dass $F$ die Formel $G_1 \vee G_2$ repräsentiert. In diesem Fall werten wir zunächst $G_1$ und $G_2$ unter der Belegung $I$ aus und verknüpfen das Ergebnis mit dem Operator "`or`".

7. In Zeile 10 betrachten wir den Fall, dass $F$ die Form $G_1 \rightarrow G_2$ hat. In diesem Fall werten wir zunächst $G_1$ und $G_2$ unter der Belegung $I$ aus und nutzen dann aus, dass die Formeln

   $$G_1 \rightarrow G_2 \quad \text{und} \quad \neg G_1 \vee G_2$$

   äquivalent sind.

8. In Zeile 11 führen wir die Auswertung einer Formel $F \leftrightarrow G$ darauf zurück, dass diese Formel genau dann wahr ist, wenn $F$ und $G$ den selben Wahrheitswert haben.

### 4.3.5 Eine Anwendung

Wir betrachten eine spielerische Anwendung der Aussagenlogik. Inspektor Watson wird zu einem Juweliergeschäft gerufen, in das eingebrochen worden ist. In der unmittelbaren Umgebung werden drei Verdächtige Aaron, Bernard und Caine festgenommen. Die Auswertung der Akten ergibt folgendes:

1. Einer der drei Verdächtigen muss die Tat begangen haben:

   $$f_1 := a \vee b \vee c.$$

2. Wenn Aaron schuldig ist, so hat er genau einen Komplizen.

   Diese Aussage zerlegen wir zunächst in zwei Teilaussagen:

   (a) Wenn Aaron schuldig ist, dann hat er mindestens einen Komplizen:

   $$f_2 := a \rightarrow b \vee c$$

   (b) Wenn Aaron schuldig ist, dann hat er höchstens einen Komplizen:

   $$f_3 := a \rightarrow \neg(b \wedge c)$$

3. Wenn Bernard unschuldig ist, dann ist auch Caine unschuldig:

   $$f_4 := \neg b \rightarrow \neg c$$

4. Wenn genau zwei schuldig sind, dann ist Caine einer von ihnen.

   Es ist nicht leicht zu sehen, wie sich diese Aussage aussagenlogisch formulieren lässt. Wir behelfen uns mit einem Trick und überlegen uns, wann die obige Aussage falsch ist. Wir sehen, die Aussage ist dann falsch, wenn Caine nicht schuldig ist und wenn gleichzeitig Aaron und Bernard schuldig sind. Damit lautet die Formalisierung der obigen Aussage:

   $$f_5 := \neg(\neg c \wedge a \wedge b)$$

5. Wenn Caine unschuldig ist, ist Aaron schuldig.

   $$f_6 := \neg c \rightarrow a$$

Wir haben nun eine Menge $F = \{f_1, f_2, f_3, f_4, f_5, f_6\}$ von Formeln. Wir fragen uns nun, für welche Belegungen $\mathcal{I}$ alle Formeln aus der Menge $F$ wahr werden. Wenn es genau eine Belegung gibt, für die dies der Fall ist, dann liefert uns die Belegung den oder die Täter. Eine Belegung entspricht dabei 1-zu-1 der Menge der Täter. Da es zu zeitraubend ist, alle Belegungen von Hand auszuprobieren, schreiben wir besser ein Programm, das die notwendigen Berechnungen für uns durchführt. Abbildung 4.2 zeigt das Programm `Usual-Suspects.ipynb`. Wir diskutieren dieses Programm nun Zeile für Zeile.

```
1  import propLogParser as plp
2
3  def transform(s):
4      "transform the string s into a nested tuple"
5      return plp.LogicParser(s).parse()
6
7  P = { 'a', 'b', 'c' }
8  # Aaron, Bernard, or Caine is guilty.
9  f1 = 'a ∨ b ∨ c'
10 # If Aaron is guilty, he has exactly one accomplice.
11 f2 = 'a → b ∨ c'
12 f3 = 'a → ¬(b ∧ c)'
13 # If Bernard is innocent, then Caine is innocent, too.
14 f4 = '¬b → ¬c'
15 # If exactly two are guilty, then Caine is one of them.
16 f5 = '¬(¬c ∧ a ∧ b)'
17 # If Caine is innocent, then Aaron is guilty.
18 f6 = '¬c → a'
19 Fs = { f1, f2, f3, f4, f5, f6 };
20 Fs = { transform(f) for f in Fs }
21
22 def allTrue(Fs, I):
23     return all({evaluate(f, I) for f in Fs})
24
25 print({ I for I in power(P) if allTrue(Fs, I) })
```

Figure 4.2: Programm zur Aufklärung des Einbruchs

1. Da wir die aussagenlogischen Formeln als Strings eingeben, unsere Funktion `evaluate` aber geschachtelte Tupel verarbeitet, importieren wir zunächst den Parser für aussagenlogische Formeln und definieren außerdem die Funktion `transform`, die eine aussagenlogische Formel, die als String vorliegt, in ein geschachteltes Tupel umwandelt.
2. In Zeile 7 definieren wir die Menge $P$ der aussagenlogischen Variablen. Wir benutzen `a` als Abkürzung dafür, dass Aaron schuldig ist, `b` steht für Bernard und `c` ist wahr, wenn Caine schuldig ist.
3. In den Zeilen 7 – 17 definieren wir die Formeln $f_1, \cdots, f_6$.
4. `Fs` ist die Menge aller Formeln.
5. Die Formeln werden in Zeile 20 in geschachtelte Tupel transformiert.
6. Die Funktion `allTrue`$(Fs, I)$ bekommt als Eingabe eine Menge von aussagenlogischen Formeln $Fs$ und eine aussagenlogische Belegung $I$, die als Teilmenge von $P$ dargestellt wird. Falls alle Formeln $f$ aus der Menge $Fs$ unter der Belegung $I$ wahr sind, gibt diese Funktion als Ergebnis `True` zurück.

7. In Zeile 25 berechnen wir alle Belegungen, für die alle Formeln wahr werden.

Lassen wir das Programm laufen, so sehen wir, dass es nur eine einzige Belegung gibt, bei der alle Formeln wahr werden. Dies ist die Belegung

```
{'b', 'c'}.
```

Damit ist das Problem eindeutig lösbar und Bernard und Caine sind schuldig.

## 4.4 Tautologien

Die Tabelle in Abbildung 4.2 zeigt, dass die Formel

$$(p \to q) \to (\neg p \to q) \to q$$

für jede aussagenlogische Interpretation wahr ist, denn in der letzten Spalte dieser Tabelle steht immer der Wert `True`. Formeln mit dieser Eigenschaft bezeichnen wir als Tautologie.

**Definition 7 (Tautologie)** *Ist $f$ eine aussagenlogische Formel und gilt*

$$\mathcal{I}(f) = \texttt{True} \quad \textit{für jede aussagenlogische Interpretation } \mathcal{I},$$

*dann ist $f$ eine Tautologie. In diesem Fall schreiben wir*

$$\models f.$$ ◇

Ist eine Formel $f$ eine Tautologie, so sagen wir auch, dass $f$ allgemeingültig ist.
**Beispiele**:

1. $\models p \vee \neg p$
2. $\models p \to p$
3. $\models p \wedge q \to p$
4. $\models p \to p \vee q$
5. $\models (p \to \bot) \leftrightarrow \neg p$
6. $\models p \wedge q \leftrightarrow q \wedge p$

Wir können die Tatsache, dass es sich bei diesen Formeln um Tautologien handelt, durch eine Tabelle nachweisen, die analog zu der auf Seite 41 gezeigten Tabelle 4.2 aufgebaut ist. Dieses Verfahren ist zwar konzeptuell sehr einfach, allerdings zu ineffizient, wenn die Anzahl der aussagenlogischen Variablen groß ist. Ziel dieses Kapitels ist daher die Entwicklung eines effizienteren Verfahren.

Die letzten beiden Beispiele in der obigen Aufzählung geben Anlass zu einer neuen Definition.

**Definition 8 (Äquivalent)** *Zwei Formeln $f$ und $g$ heißen äquivalent g.d.w.*

$$\models f \leftrightarrow g$$

*gilt.* ◇

**Beispiele**: Es gelten die folgenden Äquivalenzen:

| | | |
|---|---|---|
| $\models \neg\bot \leftrightarrow \top$ | $\models \neg\top \leftrightarrow \bot$ | |
| $\models p \vee \neg p \leftrightarrow \top$ | $\models p \wedge \neg p \leftrightarrow \bot$ | Tertium-non-Datur |
| $\models p \vee \bot \leftrightarrow p$ | $\models p \wedge \top \leftrightarrow p$ | Neutrales Element |
| $\models p \vee \top \leftrightarrow \top$ | $\models p \wedge \bot \leftrightarrow \bot$ | |
| $\models p \wedge p \leftrightarrow p$ | $\models p \vee p \leftrightarrow p$ | Idempotenz |
| $\models p \wedge q \leftrightarrow q \wedge p$ | $\models p \vee q \leftrightarrow q \vee p$ | Kommutativität |
| $\models (p \wedge q) \wedge r \leftrightarrow p \wedge (q \wedge r)$ | $\models (p \vee q) \vee r \leftrightarrow p \vee (q \vee r)$ | Assoziativität |
| $\models \neg\neg p \leftrightarrow p$ | | Elimination von $\neg\neg$ |
| $\models p \wedge (p \vee q) \leftrightarrow p$ | $\models p \vee (p \wedge q) \leftrightarrow p$ | Absorption |
| $\models p \wedge (q \vee r) \leftrightarrow (p \wedge q) \vee (p \wedge r)$ | $\models p \vee (q \wedge r) \leftrightarrow (p \vee q) \wedge (p \vee r)$ | Distributivität |
| $\models \neg(p \wedge q) \leftrightarrow \neg p \vee \neg q$ | $\models \neg(p \vee q) \leftrightarrow \neg p \wedge \neg q$ | DeMorgan'sche Regeln |
| $\models (p \to q) \leftrightarrow \neg p \vee q$ | | Elimination von $\to$ |
| $\models (p \leftrightarrow q) \leftrightarrow (\neg p \vee q) \wedge (\neg q \vee p)$ | | Elimination von $\leftrightarrow$ |

Wir können diese Äquivalenzen nachweisen, indem wir in einer Tabelle sämtliche Belegungen durchprobieren. Eine solche Tabelle heißt auch Wahrheits-Tafel. Wir demonstrieren dieses Verfahren anhand der ersten DeMorgan'schen Regel. Wir erkennen, dass in Abbildung 4.3 in den letzten beiden

| $p$ | $q$ | $\neg p$ | $\neg q$ | $p \wedge q$ | $\neg(p \wedge q)$ | $\neg p \vee \neg q$ |
|---|---|---|---|---|---|---|
| True | True | False | False | True | False | False |
| True | False | False | True | False | True | True |
| False | True | True | False | False | True | True |
| False | False | True | True | False | True | True |

Table 4.3: Nachweis der ersten DeMorgan'schen Regel

Spalten in jeder Zeile dieselben Werte stehen. Daher sind die Formeln, die zu diesen Spalten gehören, äquivalent.

### 4.4.1 Testen der Allgemeingültigkeit in *Python*

Die manuelle Überprüfung der Frage, ob eine gegebene Formel $f$ eine Tautologie ist, läuft auf die Erstellung umfangreicher Wahrheitstafeln heraus. Solche Wahrheitstafeln von Hand zu erstellen ist viel zu zeitaufwendig. Wir wollen daher nun ein *Python*-Programm entwickeln, mit dessen Hilfe wir die obige Frage automatisch beantworten können. Die Grundidee ist, dass wir die zu untersuchende Formel für alle möglichen Belegungen auswerten und überprüfen, ob sich bei der Auswertung jedes Mal der Wert `True` ergibt. Dazu müssen wir zunächst einen Weg finden, alle möglichen Belegungen einer Formel zu berechnen. Wir haben früher schon gesehen, dass Belegungen $\mathcal{I}$ zu Teilmengen $M$ der Menge der aussagenlogischen Variablen $\mathcal{P}$ korrespondieren, denn für jedes $M \subseteq \mathcal{P}$ können wir eine aussagenlogische Belegung $\mathcal{I}(M)$ wie folgt definieren:

$$\mathcal{I}(M)(p) := \begin{cases} \texttt{True} & \text{falls } p \in M; \\ \texttt{False} & \text{falls } p \notin M. \end{cases}$$

Wir stellen daher eine aussagenlogische Belegung durch die Menge $M_{\mathcal{I}}$ der aussagenlogischen Variablen $x$ dar, für die $\mathcal{I}(x)$ den Wert `True` hat. Bei gegebener aussagenlogischer Belegung $\mathcal{I}$ können wir die Menge $M_{\mathcal{I}}$ wie folgt definieren:

$$M_{\mathcal{I}} := \{p \in \mathcal{P} \mid \mathcal{I}(p) = \texttt{True}\}.$$

Damit können wir nun eine Funktion implementieren, die für eine gegebene aussagenlogische Formel $f$ testet, ob $f$ eine Tautologie ist. Hierzu müssen wir zunächst die Menge $\mathcal{P}$ der aussagenlogischen Variablen bestimmen, die in $f$ auftreten. Abstrakt definieren wir dazu eine Funktion $\texttt{collectVars}(f)$, welche die Menge aller aussagenlogischen Variablen berechnet, die in einer aussagenlogischen Formel $f$ auftreten. Diese Funktion ist durch die folgenden rekursiven Gleichugen spezifiziert:

1. $\texttt{collectVars}(p) = \{p\}$ für alle aussagenlogischen Variablen $p$.
2. $\texttt{collectVars}(\top) = \{\}$.
3. $\texttt{collectVars}(\bot) = \{\}$.
4. $\texttt{collectVars}(\neg f) := \texttt{collectVars}(f)$.
5. $\texttt{collectVars}(f \wedge g) := \texttt{collectVars}(f) \cup \texttt{collectVars}(g)$.
6. $\texttt{collectVars}(f \vee g) := \texttt{collectVars}(f) \cup \texttt{collectVars}(g)$.
7. $\texttt{collectVars}(f \rightarrow g) := \texttt{collectVars}(f) \cup \texttt{collectVars}(g)$.
8. $\texttt{collectVars}(f \leftrightarrow g) := \texttt{collectVars}(f) \cup \texttt{collectVars}(g)$.

Die in Abbildung 4.3 auf Seite 49 zeigt, dass wir diese Gleichungen unmittelbar in *Python* umsetzen können, wobei wir die letzten vier Fälle zusammengefasst haben, denn die Berechnung der Variablen verläuft in diesen Fällen analog.

```
1  def collectVars(f):
2      "Collect all propositional variables occurring in the formula f."
3      if isinstance(f, str):
4          return { f }
5      if f[0] in ['⊤', '⊥']:
6          return set()
7      if f[0] == '¬':
8          return collectVars(f[1])
9      return collectVars(f[1]) | collectVars(f[2])
```

Figure 4.3: Überprüfung der Allgemeingültigkeit einer aussagenlogischen Formel

Damit sind wir nun in der Lage, eine Funktion $\texttt{tautology}(f)$ zu implementieren, die für eine gegebene aussagenlogische Formel $f$ überprüft, ob $f$ eine Tautologie ist. Diese in Abbildung 4.4 auf Seite 50 gezeigte Funktion arbeitet wie folgt:

1. Zunächst berechnen wir in Zeile 3 die Menge `P` der aussagenlogischen Variablen, die in $f$ auftreten.
2. Sodann berechnen wir mit Hilfe der in dem Modul `power` definierten Funktion `allSubsets` die Liste `A` aller Teilmengen von `P`. Jede als Menge dargestellte aussagenlogische Belegung `I` ist ein Element dieser Liste.
3. Anschließend prüfen wir für jede mögliche Belegung `I`, ob die Auswertung der Formel $f$ für die Belegung `I` den Wert `True` ergibt und geben gegebenenfalls `True` zurück.
4. Andernfalls geben wir die erste Belegung `I` zurück, für welche die Formel $f$ den Wert `False` hat.

```
1  def tautology(f):
2      "Check, whether the formula f is a tautology."
3      P = collectVars(f)
4      A = power.allSubsets(P)
5      if { evaluate(f, I) for I in A } == { True }:
6          return True
7      else:
8          return [I for I in A if not evaluate(f, I)][0]
```

Figure 4.4: Überprüfung der Allgemeingültigkeit einer aussagenlogischen Formel

### 4.4.2 Nachweis der Allgemeingültigkeit durch Äquivalenz-Umformungen

Wollen wir nachweisen, dass eine Formel eine Tautologie ist, können wir uns prinzipiell immer einer Wahrheits-Tafel bedienen. Aber diese Methode hat einen Haken: Kommen in der Formel $n$ verschiedene Aussage-Variablen vor, so hat die Tabelle $2^n$ Zeilen. Beispielsweise hat die Tabelle zum Nachweis eines der Distributiv-Gesetze bereits 8 Zeilen, da hier 3 verschiedene Variablen auftreten. Das gleiche Problem tritt auch in der im letzten Abschnitt diskutierten Funktion `tautology` auf, denn dort berechnen wir die Potenz-Menge der Menge aller aussagenlogischen Variablen, die in der dort vorgegebenen aussagenlogischen Formel `F` auftreten. Auch hier gilt: Treten in der Formel `F` insgesamt $n$ verschiedene aussagenlogische Variablen auf, so hat die Potenz-Menge $2^n$ verschiedene Elemente und daher ist dieses Programm für solche Formeln, in denen viele verschiedenen Variablen auftreten, unbrauchbar.

Eine andere Möglichkeit nachzuweisen, dass eine Formel eine Tautologie ist, ergibt sich dadurch, dass wir die Formel mit Hilfe der im letzten Abschnitt aufgeführten Äquivalenzen vereinfachen. Wenn es gelingt, eine Formel $F$ unter Verwendung dieser Äquivalenzen zu $\top$ zu vereinfachen, dann ist gezeigt, dass $F$ eine Tautologie ist. Wir demonstrieren das Verfahren zunächst an einem Beispiel. Mit Hilfe einer Wahrheits-Tafel hatten wir schon gezeigt, dass die Formel

$$(p \rightarrow q) \rightarrow (\neg p \rightarrow q) \rightarrow q$$

eine Tautologie ist. Wir zeigen nun, wie wir diesen Tatbestand auch durch eine Kette von Äquivalenz-Umformungen einsehen können:

$$
\begin{array}{lcl}
 & (p \to q) \to (\neg p \to q) \to q & \text{(Elimination von } \to\text{)} \\
\Leftrightarrow & (\neg p \vee q) \to (\neg p \to q) \to q & \text{(Elimination von } \to\text{)} \\
\Leftrightarrow & (\neg p \vee q) \to (\neg\neg p \vee q) \to q & \text{(Elimination der Doppelnegation)} \\
\Leftrightarrow & (\neg p \vee q) \to (p \vee q) \to q & \text{(Elimination von } \to\text{)} \\
\Leftrightarrow & \neg(\neg p \vee q) \vee ((p \vee q) \to q) & \text{(DeMorgan)} \\
\Leftrightarrow & (\neg\neg p \wedge \neg q) \vee ((p \vee q) \to q) & \text{(Elimination der Doppelnegation)} \\
\Leftrightarrow & (p \wedge \neg q) \vee ((p \vee q) \to q) & \text{(Elimination von } \to\text{)} \\
\Leftrightarrow & (p \wedge \neg q) \vee (\neg(p \vee q) \vee q) & \text{(DeMorgan)} \\
\Leftrightarrow & (p \wedge \neg q) \vee ((\neg p \wedge \neg q) \vee q) & \text{(Distributivität)} \\
\Leftrightarrow & (p \wedge \neg q) \vee ((\neg p \vee q) \wedge (\neg q \vee q)) & \text{(Tertium-non-Datur)} \\
\Leftrightarrow & (p \wedge \neg q) \vee ((\neg p \vee q) \wedge \top) & \text{(Neutrales Element)} \\
\Leftrightarrow & (p \wedge \neg q) \vee (\neg p \vee q) & \text{(Distributivität)} \\
\Leftrightarrow & (p \vee (\neg p \vee q)) \wedge (\neg q \vee (\neg p \vee q)) & \text{(Assoziativität)} \\
\Leftrightarrow & ((p \vee \neg p) \vee q) \wedge (\neg q \vee (\neg p \vee q)) & \text{(Tertium-non-Datur)} \\
\Leftrightarrow & (\top \vee q) \wedge (\neg q \vee (\neg p \vee q)) & \text{(Neutrales Element)} \\
\Leftrightarrow & \top \wedge (\neg q \vee (\neg p \vee q)) & \text{(Neutrales Element)} \\
\Leftrightarrow & \neg q \vee (\neg p \vee q) & \text{(Assoziativität)} \\
\Leftrightarrow & (\neg q \vee \neg p) \vee q & \text{(Kommutativität)} \\
\Leftrightarrow & (\neg p \vee \neg q) \vee q & \text{(Assoziativität)} \\
\Leftrightarrow & \neg p \vee (\neg q \vee q) & \text{(Tertium-non-Datur)} \\
\Leftrightarrow & \neg p \vee \top & \\
\Leftrightarrow & \top &
\end{array}
$$

Die Umformungen in dem obigen Beweis sind nach einem bestimmten System durchgeführt worden. Um dieses System präzise formulieren zu können, benötigen wir noch einige Definitionen.

**Definition 9 (Literal)** *Eine aussagenlogische Formel $f$ heißt Literal g.d.w. einer der folgenden Fälle vorliegt:*

1. $f = \top$ *oder* $f = \bot$.
2. $f = p$, *wobei $p$ eine aussagenlogische Variable ist.*
   *In diesem Fall sprechen wir von einem positiven Literal.*
3. $f = \neg p$, *wobei $p$ eine aussagenlogische Variable ist.*
   *In diesem Fall sprechen wir von einem negativen Literal.*

*Die Menge aller Literale bezeichnen wir mit $\mathcal{L}$.* ◇

Später werden wird noch den Begriff des Komplements eines Literals benötigen. Ist $l$ ein Literal, so wird das Komplement von $l$ mit $\overline{l}$ bezeichnet. Das Komplement wird durch Fall-Unterscheidung definiert:

1. $\overline{\top} = \bot$ und $\overline{\bot} = \top$.
2. $\overline{p} := \neg p$, falls $p \in \mathcal{P}$.
3. $\overline{\neg p} := p$, falls $p \in \mathcal{P}$.

Wir sehen, dass das Komplement $\overline{l}$ eines Literals $l$ äquivalent zur Negation von $l$ ist, wir haben also

$$\models \overline{l} \leftrightarrow \neg l.$$

**Definition 10 (Klausel)** *Eine aussagenlogische Formel $K$ ist eine Klausel wenn $K$ die Form*

$$K = l_1 \vee \cdots \vee l_r$$

*hat, wobei $l_i$ für alle $i = 1, \cdots, r$ ein Literal ist. Eine Klausel ist also eine Disjunktion von Literalen. Die Menge aller Klauseln bezeichnen wir mit $\mathcal{K}$.* ◇

Oft werden Klauseln auch einfach als Mengen von Literalen betrachtet. Durch diese Sichtweise abstrahieren wir von der Reihenfolge und der Anzahl des Auftretens der Literale in der Disjunktion. Dies ist möglich aufgrund der Assoziativität, Kommutativität und Idempotenz des Junktors "$\vee$". Für die Klausel $l_1 \vee \cdots \vee l_r$ schreiben wir also in Zukunft auch

$$\{l_1, \cdots, l_r\}.$$

Diese Art, eine Klausel als Menge ihrer Literale darzustellen, bezeichnen wir als Mengen-Schreibweise. Das folgende Beispiel illustriert die Nützlichkeit der Mengen-Schreibweise von Klauseln. Wir betrachten die beiden Klauseln

$$p \vee (q \vee \neg r) \vee p \quad \text{und} \quad \neg r \vee q \vee (\neg r \vee p).$$

Die beiden Klauseln sind zwar äquivalent, aber rein syntaktisch sind die Formeln verschieden. Überführen wir die beiden Klauseln in Mengen-Schreibweise, so erhalten wir

$$\{p, q, \neg r\} \quad \text{und} \quad \{\neg r, q, p\}.$$

In einer Menge kommt jedes Element höchstens einmal vor und die Reihenfolge, in der die Elemente auftreten, spielt auch keine Rolle. Daher sind die beiden obigen Mengen gleich! Durch die Tatsache, dass Mengen von der Reihenfolge und der Anzahl der Elemente abstrahieren, implementiert die Mengen-Schreibweise die Assoziativität, Kommutativität und Idempotenz der Disjunktion. Übertragen wir nun die aussagenlogische Äquivalenz

$$l_1 \vee \cdots \vee l_r \vee \bot \leftrightarrow l_1 \vee \cdots \vee l_r$$

in Mengen-Schreibweise, so erhalten wir

$$\{l_1, \cdots, l_r, \bot\} \leftrightarrow \{l_1, \cdots, l_r\}.$$

Dies zeigt, dass wir das Element $\bot$ in einer Klausel getrost weglassen können. Betrachten wir die letzten Äquivalenz für den Fall, dass $r = 0$ ist, so haben wir

$$\{\bot\} \leftrightarrow \{\}.$$

Damit sehen wir, dass die leere Menge von Literalen als $\bot$ zu interpretieren ist.

**Definition 11** *Eine Klausel $K$ ist trivial, wenn einer der beiden folgenden Fälle vorliegt:*

1. $\top \in K$.
2. *Es existiert eine Variable $p \in \mathcal{P}$, so dass sowohl $p \in K$ als auch $\neg p \in K$ gilt.*
   *In diesem Fall bezeichnen wir $p$ und $\neg p$ als komplementäre Literale.* ◇

**Satz 12** *Eine Klausel $K$ ist genau dann eine Tautologie, wenn sie trivial ist.*

**Beweis**: Wir nehmen zunächst an, dass die Klausel $K$ trivial ist. Falls nun $\top \in K$ ist, dann gilt wegen der Gültigkeit der Äquivalenz $f \vee \top \leftrightarrow \top$ offenbar $K \leftrightarrow \top$. Ist $p$ eine Aussage-Variable, so dass sowohl $p \in K$ als auch $\neg p \in K$ gilt, dann folgt aufgrund der Äquivalenz $p \vee \neg p \leftrightarrow \top$ sofort $K \leftrightarrow \top$.

Wir nehmen nun an, dass die Klausel $K$ eine Tautologie ist. Wir führen den Beweis indirekt und nehmen an, dass $K$ nicht trivial ist. Damit gilt $\top \notin K$ und $K$ kann auch keine komplementären Literale enthalten. Damit hat $K$ dann die Form

$$K = \{\neg p_1, \cdots, \neg p_m, q_1, \cdots, q_n\} \quad \text{mit } p_i \neq q_j \text{ für alle } i \in \{1, \cdots, m\} \text{ und } j \in \{1, \cdots, n\}.$$

Dann könnten wir eine Interpretation $\mathcal{I}$ wie folgt definieren:

1. $\mathcal{I}(p_i) = \texttt{True}$ für alle $i = 1, \cdots, m$ und
2. $\mathcal{I}(q_j) = \texttt{False}$ für alle $j = 1, \cdots, n$,

Mit dieser Interpretation würde offenbar $\mathcal{I}(K) = \texttt{False}$ gelten und damit könnte $K$ keine Tautologie sein. Also ist die Annahme, dass $K$ nicht trivial ist, falsch. □

**Definition 13 (Konjunktive Normalform)** *Eine Formel $F$ ist in konjunktiver Normalform (kurz KNF) genau dann, wenn $F$ eine Konjunktion von Klauseln ist, wenn also gilt*

$$F = K_1 \wedge \cdots \wedge K_n,$$

*wobei die $K_i$ für alle $i = 1, \cdots, n$ Klauseln sind.* ◇

Aus der Definition der KNF folgt sofort:

**Korollar 14** *Ist $F = K_1 \wedge \cdots \wedge K_n$ in konjunktiver Normalform, so gilt*

$$\models F \quad \textit{genau dann, wenn} \quad \models K_i \quad \textit{für alle } i = 1, \cdots, n.$$ □

Damit können wir für eine Formel $F = K_1 \wedge \cdots \wedge K_n$ in konjunktiver Normalform leicht entscheiden, ob $F$ eine Tautologie ist, denn $F$ ist genau dann eine Tautologie, wenn alle Klauseln $K_i$ trivial sind.

Da für die Konjunktion analog zur Disjunktion das Assoziativ-, Kommutativ- und Idempotenz-Gesetz gilt, ist es zweckmäßig, auch für Formeln in konjunktiver Normalform wie folgt eine Mengen-Schreibweise einzuführen: Ist die Formel

$$F = K_1 \wedge \cdots \wedge K_n$$

in konjunktiver Normalform, so repräsentieren wir diese Formel durch die Menge ihrer Klauseln und schreiben

$$F = \{K_1, \cdots, K_n\}.$$

Hierbei werden auch die Klauseln ihrerseits in Mengen-Schreibweise angegeben. Wir geben ein Beispiel: Sind $p$, $q$ und $r$ Aussage-Variablen, so ist die Formel

$$(p \vee q \vee \neg r) \wedge (q \vee \neg r \vee p \vee q) \wedge (\neg r \vee p \vee \neg q)$$

in konjunktiver Normalform. In Mengen-Schreibweise wird daraus

$$\big\{\{p, q, \neg r\},\ \{p, \neg q, \neg r\}\big\}.$$

Wir stellen nun ein Verfahren vor, mit dem sich jede Formel $F$ in KNF transformieren lässt. Nach dem oben Gesagten können wir dann leicht entscheiden, ob $F$ eine Tautologie ist.

1. Eliminiere alle Vorkommen des Junktors "$\leftrightarrow$" mit Hilfe der Äquivalenz

$$(F \leftrightarrow G) \leftrightarrow (F \rightarrow G) \wedge (G \rightarrow F).$$

2. Eliminiere alle Vorkommen des Junktors "$\rightarrow$" mit Hilfe der Äquivalenz

$$(F \rightarrow G) \leftrightarrow \neg F \vee G.$$

3. Schiebe die Negationszeichen soweit es geht nach innen. Verwende dazu die folgenden Äquivalenzen:

   (a) $\neg\bot \leftrightarrow \top$
   (b) $\neg\top \leftrightarrow \bot$
   (c) $\neg\neg F \leftrightarrow F$
   (d) $\neg(F \wedge G) \leftrightarrow \neg F \vee \neg G$
   (e) $\neg(F \vee G) \leftrightarrow \neg F \wedge \neg G$

   In dem Ergebnis, das wir nach diesem Schritt erhalten, stehen die Negationszeichen nur noch unmittelbar vor den aussagenlogischen Variablen. Formeln mit dieser Eigenschaft bezeichnen wir auch als Formeln in Negations-Normalform.

4. Stehen in der Formel jetzt "$\vee$"-Junktoren über "$\wedge$"-Junktoren, so können wir durch Ausmultiplizieren, sprich Verwendung des Distributiv-Gesetzes

$$\begin{array}{rl} & (F_1 \wedge \cdots \wedge F_m) \vee (G_1 \wedge \cdots \wedge G_n) \\ \leftrightarrow & (F_1 \vee G_1) \wedge \cdots \wedge (F_1 \vee G_n) \;\wedge\; \cdots \;\wedge\; (F_m \vee G_1) \wedge \cdots \wedge (F_m \vee G_n) \end{array}$$

   den Junktor "$\vee$" nach innen schieben.

5. In einem letzten Schritt überführen wir die Formel nun in Mengen-Schreibweise, indem wir zunächst die Disjunktionen aller Literale als Mengen zusammenfassen und anschließend alle so entstandenen Klauseln wieder in einer Menge zusammen fassen.

Hier sollten wir noch bemerken, dass die Formel beim Ausmultiplizieren stark anwachsen kann. Das liegt daran, dass die Formel $F$ auf der rechten Seite der Äquivalenz $F \vee (G \wedge H) \leftrightarrow (F \vee G) \wedge (F \vee H)$ zweimal auftritt, während sie links nur einmal vorkommt.

Wir demonstrieren das Verfahren am Beispiel der Formel

$$(p \rightarrow q) \rightarrow (\neg p \rightarrow \neg q).$$

1. Da die Formel den Junktor "$\leftrightarrow$" nicht enthält, ist im ersten Schritt nichts zu tun.

2. Die Elimination des Junktors "$\rightarrow$" liefert

$$\neg(\neg p \vee q) \vee (\neg\neg p \vee \neg q).$$

3. Die Umrechnung auf Negations-Normalform ergibt

$$(p \wedge \neg q) \vee (p \vee \neg q).$$

4. Durch "Ausmultiplizieren" erhalten wir

$$\big(p \vee (p \vee \neg q)\big) \wedge \big(\neg q \vee (p \vee \neg q)\big).$$

5. Die Überführung in die Mengen-Schreibweise ergibt zunächst als Klauseln die beiden Mengen

$$\{p, p, \neg q\} \quad \text{und} \quad \{\neg q, p, \neg q\}.$$

   Da die Reihenfolge der Elemente einer Menge aber unwichtig ist und außerdem eine Menge

jedes Element nur einmal enthält, stellen wir fest, dass diese beiden Klauseln gleich sind. Fassen wir jetzt die Klauseln noch in einer Menge zusammen, so erhalten wir

$$\big\{\{p, \neg q\}\big\}.$$

Beachten Sie, dass sich die Formel durch die Überführung in Mengen-Schreibweise noch einmal deutlich vereinfacht hat.

Damit ist die Formel in KNF überführt.

### 4.4.3 Berechnung der konjunktiven Normalform in *Python*

Wir geben nun eine Reihe von Funktionen an, mit deren Hilfe sich eine gegebene Formel $f$ in konjunktive Normalform überführen lässt. Diese Funktionen sind Teil des Jupyter Notebooks CNF.ipynb. Wir beginnen mit der Funktion

$$\texttt{elimBiconditional} : \mathcal{F} \to \mathcal{F}$$

welche die Aufgabe hat, eine vorgegebene aussagenlogische Formel $f$ in eine äquivalente Formel umzuformen, die den Junktor "$\leftrightarrow$" nicht mehr enthält. Die Funktion $\texttt{elimBiconditional}(f)$ wird durch Induktion über den Aufbau der aussagenlogischen Formel $f$ definiert. Dazu stellen wir zunächst rekursive Gleichungen auf, die das Verhalten der Funktion `elimBiconditional` beschreiben:

1. Wenn $f$ eine Aussage-Variable $p$ ist, so ist nichts zu tun:

   $$\texttt{elimBiconditional}(p) = p \quad \text{für alle } p \in \mathcal{P}.$$

2. Die Fälle, in denen $f$ gleich dem Verum oder dem Falsum ist, sind ebenfalls trivial:

   $$\texttt{elimBiconditional}(\top) = \top \quad \text{und} \quad \texttt{elimBiconditional}(\bot) = \bot.$$

3. Hat $f$ die Form $f = \neg g$, so eliminieren wir den Junktor "$\leftrightarrow$" rekursiv aus der Formel $g$ und negieren die resultierende Formel:

   $$\texttt{elimBiconditional}(\neg g) = \neg\texttt{elimBiconditional}(g).$$

4. In den Fällen $f = g_1 \wedge g_2$, $f = g_1 \vee g_2$ und $f = g_1 \to g_2$ eliminieren wir rekursiv den Junktor "$\leftrightarrow$" aus den Formeln $g_1$ und $g_2$ und setzen dann die Formel wieder zusammen:

   (a) $\texttt{elimBiconditional}(g_1 \wedge g_2) = \texttt{elimBiconditional}(g_1) \wedge \texttt{elimBiconditional}(g_2)$.

   (b) $\texttt{elimBiconditional}(g_1 \vee g_2) = \texttt{elimBiconditional}(g_1) \vee \texttt{elimBiconditional}(g_2)$.

   (c) $\texttt{elimBiconditional}(g_1 \to g_2) = \texttt{elimBiconditional}(g_1) \to \texttt{elimBiconditional}(g_2)$.

5. Hat $f$ die Form $f = g_1 \leftrightarrow g_2$, so benutzen wir die Äquivalenz

   $$(g_1 \leftrightarrow g_2) \leftrightarrow \big((g_1 \to g_2) \wedge (g_2 \to g_1)\big).$$

   Das führt auf die Gleichung:

   $$\texttt{elimBiconditional}(g_1 \leftrightarrow g_2) = \texttt{elimBiconditional}\big((g_1 \to g_2) \wedge (g_2 \to g_1)\big).$$

   Der Aufruf der Funktion `elimBiconditional` auf der rechten Seite der Gleichung ist notwendig, denn der Junktor "$\leftrightarrow$" kann ja noch in $g_1$ und $g_2$ auftreten.

Abbildung 4.5 auf Seite 56 zeigt die Implementierung der Funktion `elimBiconditional`.

```
1   def elimBiconditional(f):
2       "Eliminate the logical operator '↔' from the formula f."
3       if isinstance(f, str):   # This case covers variables.
4           return f
5       if f[0] == '↔':
6           g, h = f[1:]
7           ge   = elimBiconditional(g)
8           he   = elimBiconditional(h)
9           return ('∧', ('→', ge, he), ('→', he, ge))
10      if f[0] == '⊤' or f[0] == '⊥':
11          return f
12      if f[0] == '¬':
13          g  = f[1]
14          ge = elimBiconditional(g)
15          return ('¬', ge)
16      else:
17          op, g, h = f
18          ge       = elimBiconditional(g)
19          he       = elimBiconditional(h)
20          return (op, ge, he)
```

Figure 4.5: Elimination von $\leftrightarrow$

1. In Zeile 3 prüft der Funktions-Aufruf `isinstance`$(f,$`str`$)$, ob $f$ ein String ist. In diesem Fall muss $f$ eine aussagenlogische Variable sein, denn alle anderen aussagenlogischen Formeln werden als geschachtelte Listen dargestellt. Daher wird $f$ in diesem Fall unverändert zurück gegeben.

2. In Zeile 5 überprüfen wir den Fall, dass $f$ die Form $g \leftrightarrow h$ hat. In diesem Fall eliminieren wir den Junktor "$\leftrightarrow$" aus $g$ und $h$ durch einen rekursiven Aufruf der Funktion `elimBiconditional`. Anschließend benutzen wir die Äquivalenz

   $$(g \leftrightarrow h) \leftrightarrow (g \rightarrow h) \wedge (h \rightarrow g).$$

3. In Zeile 10 behandeln wir die Fälle, dass $f$ gleich dem Verum oder dem Falsum ist. Hier ist zu beachten, dass diese Formeln ebenfalls als geschachtelte Tupel dargestellt werden, Verum wird beispielsweise als das Tuple `('⊤',)` dargestellt, während Falsum von uns in *Python* durch das Tuple `('⊥',)` repräsentiert wird. In diesem Fall wird $f$ unverändert zurück gegeben.

4. In Zeile 14 betrachten wir den Fall, dass $f$ eine Negation ist. Dann hat $f$ die Form

   $$(\text{'}\neg\text{'},\ g)$$

   und wir müssen den Junktor "$\leftrightarrow$" rekursiv aus $g$ entfernen.

5. In den jetzt noch verbleibenden Fällen hat $f$ die Form

   $$(\text{'}o\text{'},\ g,\ h) \quad \text{mit } o \in \{\rightarrow, \wedge, \vee\}.$$

In diesen Fällen muss der Junktor "$\leftrightarrow$" rekursiv aus den Teilformeln $g$ und $h$ entfernt werden.

Als nächstes betrachten wir die Funktion zur Elimination des Junktors "$\rightarrow$". Abbildung 4.6 auf Seite 57 zeigt die Implementierung der Funktion `elimFolgt`. Die der Implementierung zu Grunde liegende Idee ist dieselbe wie bei der Elimination des Junktors "$\leftrightarrow$". Der einzige Unterschied besteht darin, dass wir jetzt die Äquivalenz

$$(g \rightarrow h) \leftrightarrow (\neg g \vee h)$$

benutzen. Außerdem können wir bei der Implementierung dieser Funktion voraussetzen, dass der Junktor "$\leftrightarrow$" bereits aus der aussagenlogischen Formel `F`, die als Argument übergeben wird, eliminiert worden ist. Dadurch entfällt bei der Implementierung ein Fall.

```
1   def elimConditional(f):
2       "Eliminate the logical operator '→' from f."
3       if isinstance(f, str):
4           return f
5       if f[0] == '⊤' or f[0] == '⊥':
6           return f
7       if f[0] == '→':
8           g, h = f[1:]
9           ge   = elimConditional(g)
10          he   = elimConditional(h)
11          return ('∨', ('¬', ge), he)
12      if f[0] == '¬':
13          g  = f[1]
14          ge = elimConditional(g)
15          return ('¬', ge)
16      else:
17          op, g, h = f
18          ge       = elimConditional(g)
19          he       = elimConditional(h)
20          return (op, ge, he)
```

Figure 4.6: Elimination von $\rightarrow$

Als nächstes zeigen wir die Funktionen zur Berechnung der Negations-Normalform. Abbildung 4.7 auf Seite 58 zeigt die Implementierung der Funktionen `nnf` und `neg`, die sich wechselseitig aufrufen. Dabei berechnet `nnf(`$f$`)` die Negations-Normalform von $f$, während `neg(`$f$`)` die Negations-Normalform von $\neg f$ berechnet, es gilt also

$$\texttt{neg(F)} = \texttt{nnf}(\neg \texttt{f}).$$

Die eigentliche Arbeit wird dabei in der Funktion `neg` erledigt, denn dort werden die beiden DeMorgan'schen Gesetze

$$\neg(f \wedge g) \leftrightarrow (\neg f \vee \neg g) \quad \text{und} \quad \neg(f \vee g) \leftrightarrow (\neg f \wedge \neg g)$$

angewendet. Wir beschreiben die Umformung in Negations-Normalform durch die folgenden Glei-

chungen:

1. $\texttt{nnf}(p) = p \quad$ für alle $p \in \mathcal{P}$,
2. $\texttt{nnf}(\top) = \top$,
3. $\texttt{nnf}(\bot) = \bot$,
4. $\texttt{nnf}(\neg \texttt{f}) = \texttt{neg}(\texttt{f})$,
5. $\texttt{nnf}(f_1 \wedge f_2) = \texttt{nnf}(f_1) \wedge \texttt{nnf}(f_2)$,
6. $\texttt{nnf}(f_1 \vee f_2) = \texttt{nnf}(f_1) \vee \texttt{nnf}(f_2)$.

```
1   def nnf(f):
2       "Compute the negation normal form of f."
3       if isinstance(f, str):
4           return f
5       if f[0] == '⊤' or f[0] == '⊥':
6           return f
7       if f[0] == '¬':
8           g = f[1]
9           return neg(g)
10      if f[0] == '∧':
11          g, h = f[1:]
12          return ('∧', nnf(g), nnf(h))
13      if f[0] == '∨':
14          g, h = f[1:]
15          return ('∨', nnf(g), nnf(h))
16
```

Figure 4.7: Berechnung der Negations-Normalform: Die Funktion `nnf`.

Die Hilfsprozedur `neg`, die die Negations-Normalform von $\neg f$ berechnet, spezifizieren wir ebenfalls durch rekursive Gleichungen:

1. $\texttt{neg}(p) = \texttt{nnf}(\neg p) = \neg p$ für alle Aussage-Variablen $p$.
2. $\texttt{neg}(\top) = \texttt{nnf}(\neg\top) = \texttt{nnf}(\bot) = \bot$,
3. $\texttt{neg}(\bot) = \texttt{nnf}(\neg\bot) = \texttt{nnf}(\top) = \top$,
4. $\texttt{neg}(\neg f) = \texttt{nnf}(\neg\neg f) = \texttt{nnf}(f)$.
5. $$\begin{aligned} & \texttt{neg}\big(f_1 \wedge f_2\big) \\ = \; & \texttt{nnf}\big(\neg(f_1 \wedge f_2)\big) \\ = \; & \texttt{nnf}\big(\neg f_1 \vee \neg f_2\big) \\ = \; & \texttt{nnf}\big(\neg f_1\big) \vee \texttt{nnf}\big(\neg f_2\big) \\ = \; & \texttt{neg}(f_1) \vee \texttt{neg}(f_2). \end{aligned}$$

Also haben wir:

$$\texttt{neg}(f_1 \wedge f_2) = \texttt{neg}(f_1) \vee \texttt{neg}(f_2).$$

6.
$$\begin{array}{rl} & \texttt{neg}(f_1 \vee f_2) \\ = & \texttt{nnf}(\neg(f_1 \vee f_2)) \\ = & \texttt{nnf}(\neg f_1 \wedge \neg f_2) \\ = & \texttt{nnf}(\neg f_1) \wedge \texttt{nnf}(\neg f_2) \\ = & \texttt{neg}(f_1) \wedge \texttt{neg}(f_2). \end{array}$$

Also haben wir:

$$\texttt{neg}(f_1 \vee f_2) = \texttt{neg}(f_1) \wedge \texttt{neg}(f_2).$$

Die in den Abbildungen 4.7 und 4.8 auf Seite 58 und Seite 59 gezeigten Funktionen setzen die oben diskutierten Gleichungen unmittelbar um.

```
1   def neg(f):
2       "Compute the negation normal form of ¬f."
3       if isinstance(f, str):
4           return ('¬', f)
5       if f[0] == '⊤':
6           return ('⊥',)
7       if f[0] == '⊥':
8           return ('⊤',)
9       if f[0] == '¬':
10          g = f[1]
11          return nnf(g)
12      if f[0] == '∧':
13          g, h = f[1:]
14          return ('∨', neg(g), neg(h))
15      if f[0] == '∨':
16          g, h = f[1:]
17          return ('∧', neg(g), neg(h))
```

Figure 4.8: Berechnung der Negations-Normalform: Die Funktion neg.

Als letztes stellen wir die Funktionen vor, mit denen die Formeln, die bereits in Negations-Normalform sind, ausmultipliziert und dadurch in konjunktive Normalform gebracht werden. Gleichzeitig werden die zu normalisierenden Formeln dabei in die Mengen-Schreibweise transformiert, d.h. die Formeln werden als Mengen von Mengen von Literalen dargestellt. Dabei interpretieren wir eine Menge von Literalen als Disjunktion der Literale und eine Menge von Klauseln interpretieren wir als Konjunktion der Klauseln. Mathematisch ist unser Ziel also, eine Funktion

$$\texttt{cnf} : \texttt{NNF} \rightarrow \texttt{KNF}$$

zu definieren, so dass $\texttt{cnf}(f)$ für eine Formel $f$, die in Negations-Normalform vorliegt, eine Menge

von Klauseln als Ergebnis zurück gibt, deren Konjunktion zu $f$ äquivalent ist. Die Definition von $\texttt{cnf}(f)$ erfolgt rekursiv.

1. Falls $f$ eine aussagenlogische Variable ist, geben wir als Ergebnis eine Menge zurück, die genau eine Klausel enthält. Diese Klausel ist selbst wieder eine Menge von Literalen, die als einziges Literal die aussagenlogische Variable $f$ enthält:

   $$\texttt{cnf}(f) := \big\{\{f\}\big\} \quad \text{falls } f \in \mathcal{P}.$$

2. Wir hatten früher gesehen, dass die leere Menge von *Klauseln* als $\top$ interpretiert werden kann. Daher gilt:

   $$\texttt{cnf}(\top) := \big\{\big\}.$$

3. Wir hatten ebenfalls gesehen, dass die leere Menge von *Literalen* als $\bot$ interpretiert werden kann. Daher gilt:

   $$\texttt{cnf}(\bot) := \big\{\{\}\big\}.$$

4. Falls $f$ eine Negation ist, dann muss gelten

   $$f = \neg p \quad \text{mit } p \in \mathcal{P},$$

   denn $f$ ist ja in Negations-Normalform und in einer solchen Formel kann der Negations-Operator nur auf eine aussagenlogische Variable angewendet werden. Daher ist $f$ ein Literal und wir geben als Ergebnis eine Menge zurück, die genau eine Klausel enthält. Diese Klausel ist selbst wieder eine Menge von Literalen, die als einziges Literal die Formel $f$ enthält:

   $$\texttt{cnf}(\neg p) := \big\{\{\neg p\}\big\} \quad \text{falls } p \in \mathcal{P}.$$

5. Falls $f$ eine Konjunktion ist und also $f = g \wedge h$ gilt, dann können wir die zunächst die Formeln $g$ und $h$ in KNF transformieren. Dabei erhalten wir dann Mengen von Klauseln $\texttt{cnf}(g)$ und $\texttt{cnf}(h)$. Da wir eine Menge von Klauseln als Konjunktion der in der Menge enthaltenen Klauseln interpretieren, reicht es aus, die Vereinigung der Mengen $\texttt{cnf}(f)$ und $\texttt{cnf}(g)$ zu bilden, wir haben also

   $$\texttt{cnf}(g \wedge h) = \texttt{cnf}(g) \cup \texttt{cnf}(h).$$

6. Falls $f = g \vee h$ ist, transformieren wir zunächst $g$ und $h$ in KNF. Dabei erhalten wir

   $$\texttt{cnf}(g) = \{g_1, \cdots, g_m\} \quad \text{und} \quad \texttt{cnf}(h) = \{h_1, \cdots, h_n\}.$$

   Dabei sind die $g_i$ und die $h_j$ Klauseln. Um nun die KNF von $g \vee h$ zu bilden, rechnen wir wie folgt:

   $$\begin{array}{lll}
   & g \vee h & \\
   \Leftrightarrow & (k_1 \wedge \cdots \wedge k_m) \vee (l_1 \wedge \cdots \wedge l_n) & \\
   \Leftrightarrow & (k_1 \vee l_1) \quad \wedge \quad \cdots \quad \wedge \quad (k_m \vee l_1) \quad \wedge & \\
   & \quad \vdots \qquad\qquad\qquad\qquad\quad \vdots & \\
   & (k_1 \vee l_n) \quad \wedge \quad \cdots \quad \wedge \quad (k_m \vee l_n) & \\
   \Leftrightarrow & \big\{k_i \vee l_j : i \in \{1, \cdots, m\}, j \in \{1, \cdots, n\}\big\} &
   \end{array}$$

   Berücksichtigen wir noch, dass Klauseln in der Mengen-Schreibweise als Mengen von Literalen aufgefasst werden, die implizit disjunktiv verknüpft werden, so können wir für $k_i \vee l_j$ auch $k_i \cup l_j$ schreiben. Wir sehen, dass jede Klausel aus $\texttt{cnf}(g)$ mit jeder Klausel aus $\texttt{cnf}(g)$ vereinigt

wird. Insgesamt erhalten wir damit

$$\texttt{cnf}(g \vee h) = \big\{k \cup l \mid k \in \texttt{cnf}(g) \ \wedge \ l \in \texttt{cnf}(h)\big\}.$$

Abbildung 4.9 auf Seite 61 zeigt die Implementierung der Funktion `cnf`. (Der Name `cnf` ist die Abkürzung von conjunctive normal form.)

```
1   def cnf(f):
2       if isinstance(f, str):
3           return { frozenset({f}) }
4       if f[0] == '⊤':
5           return set()
6       if f[0] == '⊥':
7           return { frozenset() }
8       if f[0] == '¬':
9           return { frozenset({f}) }
10      if f[0] == '∧':
11          g, h = f[1:]
12          return cnf(g) | cnf(h)
13      if f[0] == '∨':
14          g, h = f[1:]
15          return { k1 | k2 for k1 in cnf(g) for k2 in cnf(h) }
```

Figure 4.9: Berechnung der konjunktiven Normalform.

Zum Abschluss zeigen wir in Abbildung 4.10 auf Seite 62 wie die einzelnen Funktionen zusammenspielen.

1. Die Funktion `normalize` eliminiert zunächst die Junktoren "$\leftrightarrow$" mit Hilfe der Funktion `elimBiconditional`.
2. Anschließend wird der Junktor "$\rightarrow$" mit Hilfe der Funktion `elimConditional` ersetzt.
3. Der Aufruf von `nnf` bringt die Formel in Negations-Normalform.
4. Die Negations-Normalform wird nun mit Hilfe der Funktion `cnf` in konjunktive Normalform gebracht, wobei gleichzeitig die Formel in Mengen-Schreibweise überführt wird.
5. Schließlich entfernt die Funktion `simplify` alle Klauseln aus der Menge `N4`, die trivial sind.
6. Die Funktion `isTrivial` überprüft, ob eine Klausel $C$, die in Mengen-Schreibweise vorliegt, sowohl eine Variable $p$ als auch die Negation $\neg p$ dieser Variablen enthält, denn dann ist diese Klausel zu $\top$ äquivalent und kann weggelassen werden.

Das vollständige Programm zur Berechnung der konjunktiven Normalform finden Sie als die Datei `CNF.ipynb` unter GitHub.

**Aufgabe 9**: Berechnen Sie die konjunktiven Normalformen der folgenden aussagenlogischen Formeln und geben Sie Ihr Ergebnis in Mengenschreibweise an. Überprüfen Sie Ihr Ergebnis mit Hilfe des Jupyter-Notebooks CNF.ipynb.

```
def normalize (f):
    n1 = elimBiconditional(f)
    n2 = elimConditional(n1)
    n3 = nnf(n2)
    n4 = cnf(n3)
    return simplify(n4)

def simplify(Clauses):
    return { C for C in Clauses if not isTrivial(C) }

def isTrivial(Clause):
    return any(('¬', p) in Clause for p in Clause)
```

Figure 4.10: Normalisierung einer Formel

(a) $p \vee q \to r$,

(b) $p \vee q \leftrightarrow r$,

(c) $(p \to q) \leftrightarrow (\neg p \to \neg q)$,

(d) $(p \to q) \leftrightarrow (\neg q \to \neg p)$,

(e) $\neg r \wedge (q \vee p \to r) \to \neg q \wedge \neg p$.

## 4.5 Der Herleitungs-Begriff

Ist $\{f_1, \cdots, f_n\}$ eine Menge von Formeln, und $g$ eine weitere Formel, so können wir uns fragen, ob die Formel $g$ aus $f_1, \cdots, f_n$ folgt, ob also

$$\models f_1 \wedge \cdots \wedge f_n \to g$$

gilt. Es gibt verschiedene Möglichkeiten, diese Frage zu beantworten. Ein Verfahren kennen wir schon: Zunächst überführen wir die Formel $f_1 \wedge \cdots \wedge f_n \to g$ in konjunktive Normalform. Wir erhalten dann eine Menge $\{k_1, \cdots, k_m\}$ von Klauseln, deren Konjunktion zu der Formel

$$f_1 \wedge \cdots \wedge f_n \to g$$

äquivalent ist. Diese Formel ist nun genau dann eine Tautologie, wenn jede der Klauseln $k_1, \cdots, k_m$ trivial ist.

Das oben dargestellte Verfahren ist aber sehr aufwendig. Wir zeigen dies anhand eines Beispiels und wenden das Verfahren an, um zu entscheiden, ob $p \to r$ aus den beiden Formeln $p \to q$ und $q \to r$ folgt. Wir bilden also die konjunktive Normalform der Formel

$$h := (p \to q) \wedge (q \to r) \to p \to r$$

und erhalten nach mühsamer Rechnung

$$(p \vee \neg p \vee r \vee \neg r) \wedge (\neg q \vee \neg p \vee r \vee \neg r) \wedge (\neg q \vee \neg p \vee q \vee r) \wedge (p \vee \neg p \vee q \vee r).$$

Zwar können wir jetzt sehen, dass die Formel $h$ eine Tautologie ist, aber angesichts der Tatsache, dass wir mit bloßem Auge sehen, dass $p \to r$ aus den Formeln $p \to q$ und $q \to r$ folgt, ist die Rechnung doch sehr aufwendig.

Wir stellen daher nun eine weiteres Verfahren vor, mit dessen Hilfe wir entscheiden können, ob eine Formel aus einer gegebenen Menge von Formeln folgt. Die Idee bei diesem Verfahren ist es, die zu beweisende Formel mit Hilfe von Schluss-Regeln aus vorgegebenen Formeln herzuleiten. Das Konzept einer Schluss-Regel wird in der nun folgenden Definition festgelegt.

**Definition 15 (Schluss-Regel)** *Eine aussagenlogische Schluss-Regel ist eine Paar der Form $\langle\langle f_1, f_2\rangle, k\rangle$. Dabei ist $\langle f_1, f_2\rangle$ ein Paar von aussagenlogischen Formeln und $k$ ist eine einzelne aussagenlogische Formel. Die beiden Formeln $f_1$ und $f_2$ bezeichnen wir als Prämissen, die Formel $k$ heißt die Konklusion der Schluss-Regel. Ist das Paar $\langle\langle f_1, f_2\rangle, k\rangle$ eine Schluss-Regel, so schreiben wir dies als:*

$$\frac{f_1 \qquad f_2}{k}.$$

*Wir lesen diese Schluss-Regel wie folgt: "Aus $f_1$ und $f_2$ kann auf $k$ geschlossen werden."* ◇

**Beispiele** für Schluss-Regeln:

| Modus Ponens | Modus Tollens | Unfug |
|---|---|---|
| $\dfrac{f \qquad f \to g}{g}$ | $\dfrac{\neg g \qquad f \to g}{\neg f}$ | $\dfrac{\neg f \qquad f \to g}{\neg g}$ |

Die Definition der Schluss-Regel schränkt zunächst die Formeln, die als Prämissen bzw. Konklusion verwendet werden können, nicht weiter ein. Es ist aber sicher nicht sinnvoll, beliebige Schluss-Regeln zuzulassen. Wollen wir Schluss-Regeln in Beweisen verwenden, so sollten die Schluss-Regeln in dem in der folgenden Definition erklärten Sinne korrekt sein.

**Definition 16 (Korrekte Schluss-Regel)** *Eine Schluss-Regel der Form*

$$\frac{f_1 \qquad f_2}{k}$$

*ist genau dann korrekt, wenn $\models f_1 \wedge f_2 \to k$ gilt.* ◇

Mit dieser Definition sehen wir, dass die oben als "Modus Ponens" und "Modus Tollens" bezeichneten Schluss-Regeln korrekt sind, während die als "Unfug" bezeichnete Schluss-Regel nicht korrekt ist.

Im Folgenden gehen wir davon aus, dass alle Formeln Klauseln sind. Einerseits ist dies keine echte Einschränkung, denn wir können ja jede Formel in eine äquivalente Menge von Klauseln umformen. Andererseits haben die Formeln bei vielen in der Praxis auftretenden aussagenlogischen Problemen ohnehin die Gestalt von Klauseln. Daher stellen wir jetzt eine Schluss-Regel vor, in der sowohl die Prämissen als auch die Konklusion Klauseln sind.

**Definition 17 (Schnitt-Regel)** *Ist $p$ eine aussagenlogische Variable und sind $k_1$ und $k_2$ Mengen von Literalen, die wir als Klauseln interpretieren, so bezeichnen wir die folgende Schluss-Regel als die*

*Schnitt-Regel:*

$$\frac{k_1 \cup \{p\} \quad \{\neg p\} \cup k_2}{k_1 \cup k_2}.$$ ◇

Die Schnitt-Regel ist sehr allgemein. Setzen wir in der obigen Definition für $k_1 = \{\}$ und $k_2 = \{q\}$ ein, so erhalten wir die folgende Regel als Spezialfall:

$$\frac{\{\} \cup \{p\} \qquad \{\neg p\} \cup \{q\}}{\{\} \cup \{q\}}$$

Interpretieren wir nun die Mengen von Literalen als Disjunktionen, so haben wir:

$$\frac{p \qquad \neg p \vee q}{q}$$

Wenn wir jetzt noch berücksichtigen, dass die Formel $\neg p \vee q$ äquivalent zu der Formel $p \to q$ ist, dann ist das nichts anderes als Modus Ponens. Die Regel Modus Tollens ist ebenfalls ein Spezialfall der Schnitt-Regel. Wir erhalten diese Regel, wenn wir in der Schnitt-Regel $k_1 = \{\neg q\}$ und $k_2 = \{\}$ setzen.

**Satz 18** *Die Schnitt-Regel ist korrekt.*

**Beweis**: Wir müssen zeigen, dass

$$\models (k_1 \vee p) \wedge (\neg p \vee k_2) \to k_1 \vee k_2$$

gilt. Dazu überführen wir die obige Formel in konjunktive Normalform:

$$\begin{array}{ll}
 & (k_1 \vee p) \wedge (\neg p \vee k_2) \to k_1 \vee k_2 \\
\Leftrightarrow & \neg\big((k_1 \vee p) \wedge (\neg p \vee k_2)\big) \vee k_1 \vee k_2 \\
\Leftrightarrow & \neg(k_1 \vee p) \vee \neg(\neg p \vee k_2) \vee k_1 \vee k_2 \\
\Leftrightarrow & (\neg k_1 \wedge \neg p) \vee (p \wedge \neg k_2) \vee k_1 \vee k_2 \\
\Leftrightarrow & (\neg k_1 \vee p \vee k_1 \vee k_2) \wedge (\neg k_1 \vee \neg k_2 \vee k_1 \vee k_2) \wedge (\neg p \vee p \vee k_1 \vee k_2) \wedge (\neg p \vee \neg k_2 \vee k_1 \vee k_2) \\
\Leftrightarrow & \top \wedge \top \wedge \top \wedge \top \\
\Leftrightarrow & \top
\end{array}$$

□

**Definition 19 (Herleitungs-Begriff, $\vdash$)** *Es sei $M$ eine Menge von Klauseln und $f$ sei eine einzelne Klausel. Die Formeln aus $M$ bezeichnen wir als unsere Prämissen, die Formel $f$ heißt Konklusion. Unser Ziel ist es, mit den Prämissen aus $M$ die Konklusion $f$ zu beweisen. Dazu definieren wir induktiv die Relation*

$$M \vdash f.$$

*Wir lesen "$M \vdash f$" als "$M$ leitet $f$ her". Die induktive Definition ist wie folgt:*

1. *Aus einer Menge $M$ von Annahmen kann jede der Annahmen hergeleitet werden:*

   *Falls $f \in M$ ist, dann gilt $M \vdash f$.*

2. *Sind $k_1 \cup \{p\}$ und $\{\neg p\} \cup k_2$ Klauseln, die aus $M$ hergeleitet werden können, so kann mit der Schnitt-Regel auch die Klausel $k_1 \cup k_2$ aus $M$ hergeleitet werden:*

   *Falls sowohl $M \vdash k_1 \cup \{p\}$ als auch $M \vdash \{\neg p\} \cup k_2$ gilt, dann gilt auch $M \vdash k_1 \cup k_2$.* ◇

**Beispiel**: Um den Beweis-Begriff zu veranschaulichen geben wir ein Beispiel und zeigen

$$\big\{\ \{\neg p, q\},\ \{\neg q, \neg p\},\ \{\neg q, p\},\ \{q, p\}\ \big\} \vdash \bot.$$

Gleichzeitig zeigen wir anhand des Beispiels, wie wir Beweise zu Papier bringen:

1. Aus $\{\neg p, q\}$ und $\{\neg q, \neg p\}$ folgt mit der Schnitt-Regel $\{\neg p, \neg p\}$. Wegen $\{\neg p, \neg p\} = \{\neg p\}$ schreiben wir dies als

   $$\{\neg p, q\}, \{\neg q, \neg p\} \vdash \{\neg p\}.$$

   **Bemerkung**: Dieses Beispiel zeigt, dass die Klausel $k_1 \cup k_2$ durchaus auch weniger Elemente enthalten kann als die Summe $\texttt{card}(k_1) + \texttt{card}(k_2)$. Dieser Fall tritt genau dann ein, wenn es Literale gibt, die sowohl in $k_1$ als auch in $k_2$ vorkommen.

2. $\{\neg q, \neg p\},\ \{p, \neg q\} \vdash \{\neg q\}$.
3. $\{p, q\},\ \{\neg q\} \vdash \{p\}$.
4. $\{\neg p\},\ \{p\} \vdash \{\}$.

Als weiteres Beipiel zeigen wir nun, dass $p \to r$ aus $p \to q$ und $q \to r$ folgt. Dazu überführen wir zunächst alle Formeln in Klauseln:

$$\texttt{cnf}(p \to q) = \big\{\{\neg p, q\}\big\}, \quad \texttt{cnf}(q \to r) = \big\{\{\neg q, r\}\big\}, \quad \texttt{cnf}(p \to r) = \big\{\{\neg p, r\}\big\}.$$

Wir haben also $M = \big\{\ \{\neg p, q\},\ \{\neg q, r\}\ \big\}$ und müssen zeigen, dass

$$M \vdash \{\neg p, r\}$$

gilt. Der Beweis besteht aus einer einzigen Anwendung der Schnitt-Regel:

$$\{\neg p, q\},\ \{\neg q, r\} \vdash \{\neg p, r\}. \qquad \diamond$$

### 4.5.1 Eigenschaften des Herleitungs-Begriffs

Die Relation $\vdash$ hat zwei wichtige Eigenschaften:

**Satz 20 (Korrektheit)** *Ist $\{k_1, \cdots, k_n\}$ eine Menge von Klauseln und $k$ eine einzelne Klausel, so haben wir:*

$$\textit{Wenn } \{k_1, \cdots, k_n\} \vdash k \textit{ gilt, dann gilt auch } \models k_1 \wedge \cdots \wedge k_n \to k.$$

*Mit anderen Worten: Wenn wir eine Klausel $k$ mit Hilfe der Annahmen $k_1, \cdots, k_n$ beweisen können, dann folgt die Klausel $k$ logisch aus diesen Annahmen.*

**Beweis**: Der Beweis des Korrektheits-Satzes verläuft durch eine Induktion nach der Definition der Relation $\vdash$.

1. Fall: Es gilt $\{k_1, \cdots, k_n\} \vdash k$, weil $k \in \{k_1, \cdots, k_n\}$ ist. Dann gibt es also ein $i \in \{1, \cdots, n\}$, so dass $k = k_i$ ist. In diesem Fall müssen wir

   $$\models k_1 \wedge \cdots \wedge k_i \wedge \cdots \wedge k_n \to k_i$$

   zeigen, was offensichtlich ist.

2. Fall: Es gilt $\{k_1, \cdots, k_n\} \vdash k$, weil es eine aussagenlogische Variable $p$ und Klauseln $g$ und $h$ gibt, so dass

   $$\{k_1, \cdots, k_n\} \vdash g \cup \{p\} \quad \text{und} \quad \{k_1, \cdots, k_n\} \vdash h \cup \{\neg p\}$$

   gilt und daraus haben wir mit der Schnitt-Regel auf

   $$\{k_1, \cdots, k_n\} \vdash g \cup h$$

   geschlossen, wobei $k = g \cup h$ gilt. Wir müssen nun zeigen, dass

   $$\models k_1 \wedge \cdots \wedge k_n \rightarrow g \vee h$$

   gilt. Es sei also $\mathcal{I}$ eine aussagenlogische Interpretation, so dass

   $$\mathcal{I}(k_1 \wedge \cdots \wedge k_n) = \texttt{True}$$

   ist. Dann müssen wir zeigen, dass

   $$\mathcal{I}(g) = \texttt{True} \quad \text{oder} \quad \mathcal{I}(h) = \texttt{True}$$

   ist. Nach Induktions-Voraussetzung wissen wir

   $$\models k_1 \wedge \cdots \wedge k_n \rightarrow g \vee p \quad \text{und} \quad \models k_1 \wedge \cdots \wedge k_n \rightarrow h \vee \neg p.$$

   Wegen $\mathcal{I}(k_1 \wedge \cdots \wedge k_n) = \texttt{True}$ folgt dann

   $$\mathcal{I}(g \vee p) = \texttt{True} \quad \text{und} \quad \mathcal{I}(h \vee \neg p) = \texttt{True}.$$

   Nun gibt es zwei Fälle:

   (a) Fall: $\mathcal{I}(p) = \texttt{True}$.
   Dann ist $\mathcal{I}(\neg p) = \texttt{False}$ und daher folgt aus der Tatsache, dass $\mathcal{I}(h \vee \neg p) = \texttt{True}$ ist, dass

   $$\mathcal{I}(h) = \texttt{True}$$

   sein muss. Daraus folgt aber sofort

   $$\mathcal{I}(g \vee h) = \texttt{True}. \surd$$

   (b) Fall: $\mathcal{I}(p) = \texttt{False}$.
   Nun folgt aus $\mathcal{I}(g \vee p) = \texttt{True}$, dass

   $$\mathcal{I}(g) = \texttt{True}$$

   gelten muss. Also gilt auch in diesem Fall

   $$\mathcal{I}(g \vee h) = \texttt{True}. \surd$$ □

Die Umkehrung dieses Satzes gilt nur in abgeschwächter Form und zwar dann, wenn $k$ die leere Klausel ist, die ja dem Falsum entspricht. Wir sagen daher, dass die Schnitt-Regel widerlegungs-vollständig ist.

**Bemerkung**: Es gibt alternative Definitionen des Herleitungs-Begriffs, die nicht nur widerlegungs-vollständig sondern tatsächlich vollständig sind, d.h. immer wenn $\models f_1 \wedge \cdots \wedge f_n \rightarrow g$ gilt, dann folgt auch

$$\{f_1, \cdots, f_n\} \vdash g.$$

Diese Herleitungs-Begriffe sind allerdings wesentlich komplexer und daher umständlicher zu implementieren. Wir werden später sehen, dass die Widerlegungs-Vollständigkeit für unsere Zwecke ausreichend ist. ◇

### 4.5.2 Beweis der Widerlegungs-Vollständigkeit

Um den Satz von der Widerlegungs-Vollständigkeit der Aussagenlogik kompakt formulieren zu können, benötigen wir den Begriff der Erfüllbarkeit, den wir jetzt formal einführen.

**Definition 21 (Erfüllbarkeit)** *Es sei $M$ eine Menge von aussagenlogischen Formeln. Falls es eine aussagenlogische Interpretation $\mathcal{I}$ gibt, die alle Formeln aus $M$ erfüllt, für die also*

$$\mathcal{I}(f) = \texttt{True} \quad \text{für alle } f \in M$$

*gilt, so nennen wir $M$ erfüllbar. Weiter sagen wir, dass $M$ unerfüllbar ist und schreiben*

$$M \models \bot,$$

*wenn es keine aussagenlogische Interpretation $\mathcal{I}$ gibt, die gleichzeitig alle Formel aus $M$ erfüllt. Bezeichnen wir die Menge der aussagenlogischen Interpretationen mit $\textsc{Ali}$, so schreibt sich das formal als*

$$M \models \bot \quad \textit{g.d.w.} \quad \forall \mathcal{I} \in \textsc{Ali} : \exists C \in M : \mathcal{I}(C) = \texttt{False}.$$

*Falls eine Menge $M$ von aussagenlogischen Formeln erfüllbar ist, schreiben wir auch*

$$M \not\models \bot.$$ ◇

**Bemerkung**: Ist $M = \{f_1, \cdots, f_n\}$ eine Menge von aussagenlogischen Formeln, so können Sie sich leicht überlegen, dass $M$ genau dann unerfüllbar ist, wenn

$$\models f_1 \wedge \cdots \wedge f_n \rightarrow \bot$$

gilt. ◇

**Definition 22 (Saturierte Klausel-Mengen)** *Eine Menge $M$ von Klauseln ist saturiert wenn jede Klausel, die sich aus zwei Klauseln $C_1, C_2 \in M$ mit Hilfe der Schnitt-Regel ableiten lässt, bereits selbst wieder ein Element der Menge $M$ ist.*

**Bemerkung**: Ist $M$ eine endlichen Menge von Klauseln, so können wir $M$ zu einer saturierten Menge von Klauseln $\overline{M}$ erweitern, indem wir $\overline{M}$ als die Menge $M$ initialisieren und dann solange die Schnitt-Regel auf Klauseln aus $\overline{M}$ anwenden und zu der Menge $\overline{M}$ hinzufügen, wie dies möglich ist. Da es zu einer gegebenen endlichen Menge von aussagenlogischen Variablen nur eine beschränkte Menge von Klauseln gibt, die wir aus diesen Variablen bilden können, muss dieses Verfahren mit einer saturierten Menge abbrechen. ◇

**Satz 23 (Widerlegungs-Vollständigkeit)** *Ist $M$ eine Menge von Klauseln, so haben wir:*

$$\textit{Wenn } M \models \bot \textit{ ist, dann gilt auch } M \vdash \{\}.$$

**Beweis**: Wir führen den Beweis durch Kontraposition. Wir nehmen an, dass $M$ eine endliche Menge von Klauseln ist, aus der sich die leere Klausel nicht herleiten lässt. Dies schreiben wir als $M \not\vdash \bot$. Wir zeigen, dass es dann eine aussagenlogische Belegung $\mathcal{I}$ gibt, unter der alle Klauseln aus $M$ wahr werden.

Nach der obigen Bemerkung können wir annehmen, dass $M$ bereits saturiert ist. Es sei

$$\{p_1, \cdots, p_N\}$$

die Menge aller aussagenlogischen Variablen, die in Klauseln aus $M$ auftreten. Für alle $k =$

$0, 1, \cdots, N$ definieren wir nun eine aussagenlogische Belegung $\mathcal{I}_k$ durch Induktion nach $k$. Die aussagenlogischen Belegungen haben für alle $k = 0, 1, \cdots, N$ die Eigenschaft, dass für jede Klausel $C \in M$, die nur die Variablen $p_1, \cdots, p_k$ enthält,

$$\mathcal{I}_k(C) = \texttt{True} \tag{$*$}$$

gilt. Außerdem definiert die aussagenlogische Belegung $\mathcal{I}_k$ nur Werte für die Variablen $p_1, \cdots, p_k$.

I.A.: $k = 0$.

Wir definieren $\mathcal{I}_0$ als die leere aussagenlogische Belegung, die keiner Variablen einen Wert zuweist. Um $(*)$ nachzuweisen müssen wir zeigen, dass für jede Klausel $C \in M$, die keine aussagenlogische Variable enthält, $\mathcal{I}_0(C) = \texttt{True}$ gilt. Die einzige Klausel, die keine Variable enthält, ist die leere Klausel. Da wir vorausgesetzt haben, dass $M \not\vdash \bot$ gilt, kann $M$ die leere Klausel nicht enthalten. Also ist nichts zu zeigen.

I.S.: $k \mapsto k + 1$.

Nach IV ist $\mathcal{I}_k$ bereits definiert. Wir setzen zunächst

$$\mathcal{I}_{k+1}(p_i) := \mathcal{I}_k(p_i) \quad \text{für alle } i = 1, \cdots, k$$

und müssen nun noch $\mathcal{I}_{k+1}(p_{k+1})$ definieren. Dies geschieht über eine Fallunterscheidung.

(a) Es gibt eine Klausel

$$C \cup \{p_{k+1}\} \in \overline{M},$$

so dass $C$ höchstens die Variablen $p_1, \cdots, p_k$ enthält und außerdem $\mathcal{I}_k(C) = \texttt{False}$ ist. Dann setzen wir

$$\mathcal{I}_{k+1}(p_{k+1}) := \texttt{True},$$

denn sonst würde ja insgesamt $\mathcal{I}_{k+1}\big(C \cup \{p_{k+1}\}\big) = \texttt{False}$ gelten.

Wir müssen nun zeigen, dass für jede Klausel $D \in \overline{M}$, die nur die Variablen $p_1, \cdots, p_k, p_{k+1}$ die Aussage $\mathcal{I}_{k+1}(D) = \texttt{True}$ gilt. Hier gibt es drei Möglichkeiten:

1. Fall: $\texttt{var}(D) \subseteq \{p_1, \cdots, p_k\}$
   Dann gilt die Behauptung nach IV, denn auf diesen Variablen stimmen die Belegungen $\mathcal{I}_{k+1}$ und $\mathcal{I}_k$ überein.
2. Fall: $p_{k+1} \in D$.
   Da wir $\mathcal{I}_{k+1}(p_{k+1})$ als $\texttt{True}$ definiert haben, gilt $\mathcal{I}_{k+1}(D) = \texttt{True}$.
3. Fall: $(\neg p_{k+1}) \in D$.
   Dann hat $D$ also die Form

   $$D = E \cup \{\neg p_{k+1}\}.$$

   An dieser Stelle benötigen wir nun die Tatsache, dass $\overline{M}$ saturiert ist. Wir wenden die Schnitt-Regel auf die Klauseln $C \cup \{p_{k+1}\}$ und $E \cup \{\neg p_{k+1}\}$ an:

   $$C \cup \{p_{k+1}\}, \quad E \cup \{\neg p_{k+1}\} \quad \vdash \quad C \cup E$$

   Da $\overline{M}$ saturiert ist, gilt $C \cup E \in \overline{M}$. $C \cup E$ enthält nur die Variablen $p_1, \cdots, p_k$. Nach IV gilt also

   $$\mathcal{I}_{k+1}(C \cup E) = \mathcal{I}_k(C \cup E) = \texttt{True}.$$

   Da $\mathcal{I}_k(C) = \texttt{False}$ ist, muss $\mathcal{I}_k(E) = \texttt{True}$ gelten. Damit haben wir

$$\begin{array}{lcl}\mathcal{I}_{k+1}(D) &=& \mathcal{I}_{k+1}\big(E \cup \{\neg p_{k+1}\}\big) \\ &=& \mathcal{I}_k(E) \mathbin{\textcircled{\vee}} \mathcal{I}_{k+1}(\neg p_{k+1}) \\ &=& \texttt{True} \mathbin{\textcircled{\vee}} \texttt{False} = \texttt{True}\end{array}$$

und das war zu zeigen.

(b) Es gibt keine Klausel

$$C \cup \{p_{k+1}\} \in \overline{M},$$

so dass $C$ höchstens die Variablen $p_1, \cdots, p_k$ enthält und außerdem $\mathcal{I}_k(C) = \texttt{False}$ ist. Dann setzen wir

$$\mathcal{I}_{k+1}(p_{k+1}) := \texttt{False}.$$

In diesem Fall gibt es für Klauseln $C \in \overline{M}$ drei Fälle:

1. $C$ enthält die Variable $p_{k+1}$ nicht.
   In diesem Fall gilt nach IV bereits $\mathcal{I}_{k+1} = \texttt{True}$.
2. $C = D \cup \{p_{k+1}\}$.
   In diesem Fall muss nach der Fallunterscheidung von Fall (b) $\mathcal{I}_k(D) = \texttt{True}$ gelten und daraus folgt sofort, dass auch $\mathcal{I}_{k+1}(C) = \texttt{True}$ ist.
3. $C = D \cup \{\neg p_{k+1}\}$.
   In diesem Fall gilt nach Definition von $\mathcal{I}$, dass $\mathcal{I}_{k+1}(p_{k+1}) = \texttt{False}$ ist und daraus folgt sofort dass $\mathcal{I}_{k+1}(\neg p_{k+1}) = \texttt{True}$, was $\mathtt{i}_{k+1}(C) = \texttt{True}$ zur Folge hat.

Insgesamt haben wir auch im Induktions-Schritt gezeigt, dass die aussagenlogische Belegungen $\mathcal{I}_{k+1}$ alle Klauseln $C \in \overline{M}$ wahr macht. Wir definieren nun die aussagenlogische Belegungen $\mathcal{I}$ als $\mathcal{I} = \mathcal{I}_N$. Da die Klauseln aus $M$ nur die Variablen $p_1, \cdots, p_N$ enthalten, ist damit klar, dass $\mathcal{I}$ alle Klauseln aus $M$ wahr macht und das war zu zeigen. □

### 4.5.3 Konstruktive Interpretation des Beweises der Widerlegungs-Vollständigkeit

In diesem Abschnitt implementieren wir ein Programm, mit dessen Hilfe sich für eine Klausel-Menge $M$, aus der sich die leere Klausel-Menge nicht herleiten lässt, eine aussagenlogische Belegung $\mathcal{I}$ berechnen lässt, die alle Klauseln aus $M$ war macht. Dieses Programm ist in den Abbildungen 4.11, 4.12 und 4.13 auf den folgenden Seiten gezeigt. Sie finden dieses Programm unter der Adresse

https://github.com/karlstroetmann/Logic/blob/master/Python/Completeness.ipynb

im Netz.

Die Grundidee bei diesem Programm besteht darin, dass wir versuchen, aus einer gegebenen Menge $M$ von Klauseln alle Klauseln herzuleiten, die mit der Schnitt-Regel aus $M$ herleitbar sind. Wenn wir dabei auch die leere Klausel herleiten, dann ist $M$ aufgrund der Korrektheit der Schnitt-Regel offenbar unerfüllbar. Falls es uns aber nicht gelingt, die leere Klausel aus $M$ abzuleiten, dann konstruieren wir aus der Menge aller Klauseln, die wir aus $M$ hergeleitet haben, eine aussagenlogische Interpretation $\mathcal{I}$, die alle Klauseln aus $M$ erfüllt, womit $M$ erfüllbar wäre. Wir diskutieren zunächst die Hilfsprozeduren, die in Abbildung 4.11 gezeigt sind.

1. Die Funktion `complement` erhält als Argument ein Literal $l$ und berechnet das Komplement $\overline{l}$ dieses Literals. Falls das Literal $l$ eine aussagenlogische Variable $p$ ist, was wir daran erkennen, dass $l$ ein String ist, so haben wir $\overline{p} = \neg p$. Falls $l$ die Form $\neg p$ mit einer aussagenlogischen Variablen $p$ hat, so gilt $\overline{\neg p} = p$.

```
def complement(l):
    "Compute the complement of the literal l."
    if isinstance(l, str):  # l is a propositional variable
        return ('¬', l)
    else:                   # l = ('¬', 'p')
    return l[1]

def extractVariable(l):
    "Extract the variable of the literal l."
    if isinstance(l, str):  # l is a propositional variable
        return l
    else:                   # l = ('¬', 'p')
        return l[1]

def collectVariables(M):
    "Return the set of all variables occurring in M."
    return { extractVariable(l) for C in M
                                for l in C
           }

def cutRule(C1, C2):
    '''
    Return the set of all clauses that can be deduced with the cut rule
    from the clauses c1 and c2.
    '''
    return { C1 - {l} | C2 - {complement(l)} for l in C1
                                            if  complement(l) in C2
           }
```

Figure 4.11: Hilfsprozeduren, die in Abbildung 4.12 genutzt werden

2. Die Funktion `extractVariable` extrahiert die aussagenlogische Variable, die in einem Literal $l$ enthalten ist. Die Implementierung verläuft analog zur Implementierung der Funktion `complement` über eine Fallunterscheidung, bei der wir berücksichtigen, dass $l$ entweder die Form $p$ oder die Form $\neg p$ hat, wobei $p$ die zu extrahierende aussagenlogische Variable ist.

3. Die Funktion `collectVars` erhält als Argument eine Menge $M$ von Klauseln, wobei die einzelnen Klauseln $C \in M$ als Mengen von Literalen dargestellt werden. Aufgabe der Funktion `collectVars` ist es, die Menge aller aussagenlogischen Variablen zu berechnen, die in einer der Klauseln $C$ aus $M$ vorkommen. Bei der Implementierung iterieren wir zunächst über die Klauseln $C$ der Menge $M$ und dann für jede Klausel $C$ über die in $C$ vorkommenden Literale $l$, wobei die Literale mit Hilfe der Funktion `extractVariable` in aussagenlogische Variablen umgewandelt werden.

4. Die Funktion `cutRule` erhält als Argumente zwei Klauseln $C_1$ und $C_2$ und berechnet die Menge

aller Klauseln, die mit Hilfe einer Anwendung der Schnitt-Regel aus $C_1$ und $C_2$ gefolgert werden können. Beispielsweise können wir aus den beiden Klauseln

$$\{p, q\} \quad \text{und} \quad \{\neg p, \neg q\}$$

mit der Schnitt-Regel sowohl die Klausel

$$\{q, \neg q\} \quad \text{als auch die Klausel} \quad \{p, \neg p\}$$

herleiten.

```
29  def saturate(Clauses):
30      while True:
31          Derived = { C for C1 in Clauses
32                        for C2 in Clauses
33                        for C in cutRule(C1, C2)
34                    }
35          if frozenset() in Derived:
36              return { frozenset() }  # This is the set notation of ⊥.
37          Derived -= Clauses
38          if Derived == set():        # no new clauses found
39              return Clauses
40          Clauses |= Derived
```

Figure 4.12: Die Funktion `saturate`

Abbildung 4.12 zeigt die Funktion `saturate`. Diese Funktion erhält als Eingabe eine Menge `Clauses` von aussagenlogischen Klauseln, die als Mengen von Literalen dargestellt werden. Aufgabe der Funktion ist es, alle Klauseln herzuleiten, die mit Hilfe der Schnitt-Regel auf direktem oder indirekten Wege aus der Menge `Clauses` hergeleitet werden können. Genauer gesagt ist die Menge $S$ der Klauseln, die von der Funktion `saturate` zurück gegeben wird, unter Anwendung der Schnitt-Regel saturiert, es gilt also:

1. Falls $S$ die leere Klausel $\{\}$ enthält, dann ist $S$ saturiert.

2. Andernfalls muss `Clauses` eine Teilmenge von $S$ sein und es muss zusätzlich Folgendes gelten: Falls für ein Literal $l$ sowohl die Klausel $C_1 \cup \{l\}$ als auch die Klausel $C_2 \cup \{\overline{l}\}$ Klausel in $S$ enthalten ist, dann ist auch die Klausel $C_1 \cup C_2$ ein Element der Klauselmenge $S$:

$$C_1 \cup \{l\} \in S \;\wedge\; C_2 \cup \{\overline{l}\} \in S \;\Rightarrow\; C_1 \cup C_2 \in S$$

Wir erläutern nun die Implementierung der Funktion `saturate`.

1. Die `while`-Schleife, die in Zeile 30 beginnt, hat die Aufgabe, die Schnitt-Regel so lange wie möglich anzuwenden, um mit Hilfe der Schnitt-Regel neue Klauseln aus den gegebenen Klauseln herzuleiten. Da die Bedingung dieser Schleife den Wert `True` hat, kann diese Schleife nur durch die Ausführung einer der beiden `return`-Befehle in Zeile 36 bzw. Zeile 39 abgebrochen werden.

2. In Zeile 31 wird die Menge `Derived` als die Menge der Klauseln definiert, die mit Hilfe der Schnitt-Regel aus zwei der Klauseln in der Menge `Clauses` gefolgert werden können.

3. Falls die Menge `Derived` die leere Klausel enthält, dann ist die Menge `Clauses` widersprüchlich und die Funktion `saturate` gibt als Ergebnis die Menge $\{\{\}\}$ zurück, wobei die innere Menge als `frozenset` dargestellt werden muss. Beachten Sie, dass die Menge $\{\{\}\}$ dem Falsum entspricht.

4. Andernfalls ziehen wir in Zeile 37 von der Menge `Derived` zunächst die Klauseln ab, die schon in der Menge `Clauses` vorhanden waren, denn es geht uns darum festzustellen, ob wir im letzten Schritt tatsächlich neue Klauseln gefunden haben, oder ob alle Klauseln, die wir im letzten Schritt in Zeile 31 hergeleitet haben, schon vorher bekannt waren.

5. Falls wir nun in Zeile 38 feststellen, dass wir keine neuen Klauseln hergeleitet haben, dann ist die Menge `Clauses` saturiert und wir geben diese Menge in Zeile 39 zurück.

6. Andernfalls fügen wir in Zeile 40 die Klauseln, die wir neu gefunden haben, zu der Menge `Clauses` hinzu und setzen die `while`-Schleife fort.

An dieser Stelle müssen wir uns überlegen, dass die `while`-Schleife tatsächlich irgendwann abbricht. Das hat zwei Gründe:

1. In jeder Iteration der Schleife wird die Anzahl der Elemente der Menge `Clauses` mindestens um Eins erhöht, denn wir wissen ja, dass die Menge `Derived`, die wir in Zeile 40 zur Menge `Clauses` hinzufügen, einerseits nicht leer ist und andererseits auch nur solche Klauseln enthält, die nicht bereits in `Clauses` auftreten.

2. Die Menge `Clauses`, mit der wir ursprünglich starten, enthält eine bestimmte Anzahl $n$ von aussagenlogischen Variablen. Bei der Anwendung der Schnitt-Regel werden aber keine neuen Variablen erzeugt. Daher bleibt die Anzahl der aussagenlogischen Variablen, die in `Clauses` auftreten, immer gleich. Damit ist natürlich auch die Anzahl der Literale, die in `Clauses` auftreten, beschränkt: Wenn es nur $n$ aussagenlogische Variablen gibt, dann kann es auch höchstens $2 \cdot n$ verschiedene Literale geben. Jede Klausel aus `Clauses` ist aber eine Teilmenge der Menge aller Literale. Da eine Menge mit $k$ Elementen insgesamt $2^k$ Teilmengen hat, gibt es höchstens $2^{2 \cdot n}$ verschiedene Klauseln, die in `Clauses` auftreten können.

Aus den beiden oben angegebenen Gründen können wir schließen, dass die `while`-Schleife in Zeile 30 spätestens nach $2^{2 \cdot n}$ Iterationen abgebrochen wird.

```
41  def findValuation(Clauses):
42      "Given a set of Clauses, find an interpretation satisfying all clauses."
43      Variables = collectVariables(Clauses)
44      Clauses   = saturate(Clauses)
45      if frozenset() in Clauses:  # The set Clauses is inconsistent.
46          return False
47      Literals = set()
48      for p in Variables:
49          if any(C for C in Clauses
50                   if  p in C and C - {p} <= { complement(l) for l in Literals }
51                ):
52              Literals |= { p }
53          else:
54              Literals |= { ('¬', p) }
55      return Literals
```

Figure 4.13: Die Funktion `findValuation`.

Als nächstes diskutieren wir die Implementierung der Funktion `findValuation`, die in Abbildung 4.13 gezeigt ist. Diese Funktion erhält als Eingabe eine Menge `Clauses` von Klauseln. Falls diese Menge widersprüchlich ist, soll die Funktion das Ergebnis `False` zurück geben. Andernfalls soll eine aussagenlogische Belegung $\mathcal{I}$ berechnet werden, unter der alle Klauseln aus der Menge `Clauses` erfüllt sind. Im Detail arbeitet die Funktion `findValuation` wie folgt.

1. Zunächst berechnen wir in Zeile 43 die Menge aller aussagenlogischen Variablen, die in der Menge `Clauses` auftreten. Wir benötigen diese Menge, denn in der aussagenlogischen Interpretation, die wir als Ergebnis zurück geben wollen, müssen wir diese Variablen auf die Menge $\{\texttt{True}, \texttt{False}\}$ abbilden.

2. In Zeile 44 saturieren wir die Menge `Clauses` und berechnen alle Klauseln, die aus der ursprünglich gegebenen Menge von Klauseln mit Hilfe der Schnitt-Regel hergeleitet werden können. Hier können zwei Fälle auftreten:

   (a) Falls die leere Klausel hergeleitet werden kann, dann folgt aus der Korrektheit der Schnitt-Regel, dass die ursprünglich gegebene Menge von Klauseln widersprüchlich ist und wir geben als Ergebnis an Stelle einer Belegung den Wert `False` zurück, denn eine widersprüchliche Menge von Klauseln ist sicher nicht erfüllbar.

   (b) Andernfalls berechnen wir nun eine aussagenlogische Belegung, unter der alle Klauseln aus der Menge `Clauses` wahr werden. Zu diesem Zweck berechnen wir zunächst eine Menge von Literalen, die wir in der Variablen `Literals` abspeichern. Die Idee ist dabei, dass wir die aussagenlogische Variable `p` genau dann in die Menge `Literals` aufnehmen, wenn die gesuchte Belegung $\mathcal{I}$ die aussagenlogische Variable `p` zu `True` auswertet. Andernfalls nehmen wir an Stelle von `p` das Literal $\neg$`p` in der Menge `Literals` auf. Als Ergebnis geben wir daher in Zeile 55 die Menge `Literals` zurück. Die gesuchte aussagenlogische Belegung $\mathcal{I}$ kann dann gemäß der Formel

$$\mathcal{I}(\mathtt{p}) = \begin{cases} \mathtt{True} & \text{falls } \mathtt{p} \in \mathtt{Literals} \\ \mathtt{False} & \text{falls } \neg\mathtt{p} \in \mathtt{Literals} \end{cases}$$

berechnet werden.

3. Die Berechnung der Menge `Literals` erfolgt nun über eine `for`-Schleife. Dabei ist der Gedanke, dass wir für eine aussagenlogische Variable p genau dann das Literal p zu der Menge `Literals` hinzufügen, wenn die Belegung $\mathcal{I}$ die Variable p auf `True` abbilden muss, um die Klauseln zu erfüllen. Andernfalls fügen wir stattdessen das Literal $\neg$p zu dieser Menge hinzu.

   Die Bedingung dafür, dass wir das Literal $p$ hinzufügen müssen ist wie folgt: Angenommen, wir haben bereits Werte für die Variablen $p_1, \cdots, p_n$ in der Menge `Literals` gefunden. Die Werte dieser Variablen seien durch die Literale $l_1, \cdots, l_n$ in der Menge `Literals` wie folgt festgelegt: Wenn $l_i = p_i$ ist, dann gilt $\mathcal{I}(p_i) = \mathtt{True}$ und falls $l_i = \neg p_i$ gilt, so haben wir $\mathcal{I}(p_i) = \mathtt{False}$. Nehmen wir nun weiter an, dass eine Klausel $C$ in der Menge `Clauses` existiert, so dass

   $$C \backslash \{p\} \subseteq \{\overline{l_1}, \cdots, \overline{l_n}\} \quad \text{und} \quad \mathtt{p} \in C$$

   gilt. Wenn $\mathcal{I}(C) = \mathtt{True}$ gelten soll, dann muss $\mathcal{I}(\mathtt{p}) = \mathtt{True}$ gelten, denn nach Konstruktion von $\mathcal{I}$ gilt

   $$\mathcal{I}\big(\overline{l_i}\big) = \mathtt{False} \quad \text{für alle } i \in \{1, \cdots, n\}$$

   und damit ist p das einzige Literal in der Klausel $C$, das wir mit Hilfe der Belegung $\mathcal{I}$ überhaupt noch wahr machen können. In diesem Fall fügen wir also das Literal p in die Menge `Literals` ein. Andernfalls wird das Literal $\neg p$ zu der Menge `Literals` hinzugefügt.

Die Definition der Funktion `findValuation` setzt die induktive Definition der Belegungen $\mathcal{I}_k$ um, die wir im Beweis der Widerlegungs-Vollständigkeit angegeben haben.

## 4.6 Das Verfahren von Davis und Putnam

In der Praxis stellt sich oft die Aufgabe, für eine gegebene Menge von Klauseln $K$ eine aussagenlogische Belegung $\mathcal{I}$ zu berechnen, so dass

$$\mathtt{evaluate}(C, \mathcal{I}) = \mathtt{True} \quad \text{für alle } C \in K$$

gilt. In diesem Fall sagen wir auch, dass die Belegung $\mathcal{I}$ eine Lösung der Klausel-Menge $K$ ist. Im letzten Abschnitt haben wir bereits die Funktion `findValuation` kennengelernt, mit der wir eine solche Belegung berechnen könnten. Bedauerlicherweise ist diese Funktion für eine praktische Anwendung nicht effizient genug, denn das Saturieren einer Klausel-Menge ist im Allgemeinen sehr aufwendig. Wir werden daher in diesem Abschnitt ein Verfahren vorstellen, mit dem die Berechnung einer Lösung einer aussagenlogischen Klausel-Menge in vielen praktisch relevanten Fällen auch dann möglich ist, wenn die Anzahle der Variablen groß ist. Dieses Verfahren geht auf Davis und Putnam [DP60, DLL62] zurück. Verfeinerungen dieses Verfahrens werden beispielsweise eingesetzt, um die Korrektheit digitaler elektronischer Schaltungen nachzuweisen.

Um das Verfahren zu motivieren, überlegen wir zunächst, bei welcher Form der Klausel-Menge $K$ unmittelbar klar ist, ob es eine Belegung gibt, die $K$ löst und wie diese Belegung aussieht. Betrachten wir dazu ein Beispiel:

$$K_1 = \{\ \{p\},\ \{\neg q\},\ \{r\},\ \{\neg s\},\ \{\neg t\}\ \}$$

Die Klausel-Menge $K_1$ entspricht der aussagenlogischen Formel

$$p \wedge \neg q \wedge r \wedge \neg s \wedge \neg t.$$

Daher ist $K_1$ lösbar und die Belegung

$$\mathcal{I} = \{\ \langle p, \texttt{True}\rangle,\ \langle q, \texttt{False}\rangle,\ \langle r, \texttt{True}\rangle,\ \langle s, \texttt{False}\rangle,\ \langle t, \texttt{False}\rangle\ \}$$

ist eine Lösung. Betrachten wir eine weiteres Beispiel:

$$K_2 = \{\ \{\}, \{p\},\ \{\neg q\},\ \{r\}\ \}$$

Diese Klausel-Menge entspricht der Formel

$$\bot \wedge p \wedge \neg q \wedge r.$$

Offensichtlich ist $K_2$ unlösbar. Als letztes Beispiel betrachten wir

$$K_3 = \big\{\{p\}, \{\neg q\}, \{\neg p\}\big\}.$$

Diese Klausel-Menge kodiert die Formel

$$p \wedge \neg q \wedge \neg p$$

und ist offenbar ebenfalls unlösbar, denn eine Lösung $\mathcal{I}$ müsste die aussagenlogische Variable $p$ gleichzeitig wahr und falsch machen. Wir nehmen die an den letzten drei Beispielen gemachten Beobachtungen zum Anlass für zwei Definitionen.

**Definition 24 (Unit-Klausel)** *Eine Klausel $C$ ist eine Unit-Klausel, wenn $C$ nur aus einem Literal besteht. Es gilt dann entweder*

$$C = \{p\} \quad \textit{oder} \quad C = \{\neg p\}$$

*für eine Aussage-Variable $p$.* ◇

**Definition 25 (Triviale Klausel-Mengen)** *Eine Klausel-Menge $K$ ist genau dann eine einfache Klausel-Menge, wenn einer der beiden folgenden Fälle vorliegt:*

1. *$K$ enthält die leere Klausel, es gilt also $\{\} \in K$.*

   *In diesem Fall ist $K$ offensichtlich unlösbar.*

2. *$K$ enthält nur Unit-Klauseln mit verschiedenen Aussage-Variablen, d.h. es kann nicht sein, dass es eine aussagenlogische Variable $p$ gibt, so dass $K$ sowohl die Klausel $\{p\}$, als auch die Klausel $\{\neg p\}$ enthält. Bezeichnen wir die Menge der aussagenlogischen Variablen mit $\mathcal{P}$, so schreibt sich diese Bedingung als*

   $$\big(\forall C \in K : \texttt{card}(C) = 1\big) \ \wedge\ \forall p \in \mathcal{P} : \neg\big(\{p\} \in K \wedge \{\neg p\} \in K\big).$$

   *In diesem Fall können wir die aussagenlogische Belegung $\mathcal{I}$ wie folgt definieren:*

   $$\mathcal{I}(p) = \begin{cases} \texttt{True} & \textit{falls} \quad \{p\} \in K, \\ \texttt{False} & \textit{falls} \quad \{\neg p\} \in K. \end{cases}$$

   *Damit ist $\mathcal{I}$ dann eine Lösung der Klausel-Menge $K$.* ◇

Wie können wir nun eine gegebene Klausel-Menge in eine einfache Klausel-Menge umwandeln? Es gibt drei Möglichkeiten, Klauselmengen zu vereinfachen. Die erste der beiden Möglichkeiten kennen wir schon, die anderen beiden Möglichkeiten werden wir später näher erläutern.

1. Schnitt-Regel,
2. Subsumption und
3. Fallunterscheidung.

Wir betrachten diese Möglickeiten jetzt der Reihe nach.

### 4.6.1 Vereinfachung mit der Schnitt-Regel

Eine typische Anwendung der Schnitt-Regel hat die Form:

$$\frac{C_1 \cup \{p\} \quad \{\neg p\} \cup C_2}{C_1 \cup C_2}$$

Die hierbei erzeugte Klausel $C_1 \cup C_2$ wird in der Regel mehr Literale enthalten als die Prämissen $C_1 \cup \{p\}$ und $\{\neg p\} \cup C_2$. Enthält die Klausel $C_1 \cup \{p\}$ insgesamt $m+1$ Literale und enthält die Klausel $\{\neg p\} \cup C_2$ insgesamt $n+1$ Literale, so kann die Konklusion $C_1 \cup C_2$ bis zu $m+n$ Literale enthalten. Natürlich können es auch weniger Literale sein, und zwar dann, wenn es Literale gibt, die sowohl in $C_1$ als auch in $C_2$ auftreten. Oft ist $m+n$ aber sowohl größer als $m+1$ als auch größer als $n+1$. Die Klauseln wachsen nur dann sicher nicht, wenn $n = 0$ oder $m = 0$ ist. Dieser Fall liegt vor, wenn einer der beiden Klauseln nur aus einem Literal besteht und folglich eine Unit-Klausel ist. Da es unser Ziel ist, die Klausel-Mengen zu vereinfachen, lassen wir nur solche Anwendungen der Schnitt-Regel zu, bei denen eine der Klausel eine Unit-Klausel ist. Solche Schnitte bezeichnen wir als Unit-Schnitte. Um alle mit einer gegebenen Unit-Klausel $\{l\}$ möglichen Schnitte durchführen zu können, definieren wir eine Funktion

$$\texttt{unitCut} : 2^{\mathcal{K}} \times \mathcal{L} \to 2^{\mathcal{K}}$$

so, dass für eine Klausel-Menge $K$ und ein Literal $l$ die Funktion $\texttt{unitCut}(K, l)$ die Klausel-Menge $K$ soweit wie möglich mit Unit-Schnitten mit der Klausel $\{l\}$ vereinfacht:

$$\texttt{unitCut}(K, l) = \Big\{ C \backslash \{\overline{l}\} \;\Big|\; C \in K \Big\}.$$

Beachten Sie, dass die Menge $\texttt{unitCut}(K, l)$ genauso viele Klauseln enthält wie die Menge $K$. Allerdings sind diejenigen Klauseln aus der Menge $K$, die das Literal $\overline{l}$ enthalten, verkleinert worden. Alle anderen Klauseln aus $K$ bleiben unverändert.

Eine Klauselmenge $K$ werden wir nur dann mit Hilfe des Ausdrucks $\texttt{unitCut}(K, l)$ vereinfachen, wenn die Unit-Klausel $\{l\}$ ein Element der Menge $K$ ist.

### 4.6.2 Vereinfachung durch Subsumption

Das Prinzip der Subsumption demonstrieren wir zunächst an einem Beispiel. Wir betrachten

$$K = \big\{\{p, q, \neg r\}, \{p\}\big\} \cup M.$$

Offenbar impliziert die Klausel $\{p\}$ die Klausel $\{p, q, \neg r\}$, denn immer wenn $\{p\}$ erfüllt ist, ist automatisch auch $\{q, p, \neg r\}$ erfüllt. Das liegt daran, dass

$$\models p \to q \vee p \vee \neg r$$

gilt. Allgemein sagen wir, dass eine Klausel $C$ von einer Unit-Klausel $U$ subsumiert wird, wenn

$$U \subseteq C$$

gilt. Ist $K$ eine Klausel-Menge mit $C \in K$ und $U \in K$ und wird $C$ durch $U$ subsumiert, so können wir die Menge $K$ durch Unit-Subsumption zu der Menge $K - \{C\}$ verkleinern, wir können also die Klausel $C$ aus $K$ löschen. Dazu definieren wir eine Funktion

$$\texttt{subsume} : 2^{\mathcal{K}} \times \mathcal{L} \to 2^{\mathcal{K}},$$

die eine gegebene Klauselmenge $K$, welche die Unit-Klausel $\{l\}$ enthält, mittels Subsumption dadurch vereinfacht, dass alle durch $\{l\}$ subsumierten Klauseln aus $K$ gelöscht werden. Die Unit-Klausel $\{l\}$ selbst behalten wir natürlich. Daher definieren wir:

$$\texttt{subsume}(K, l) := \big(K \backslash \big\{C \in K \mid l \in C\big\}\big) \cup \big\{\{l\}\big\} = \big\{C \in K \mid l \notin C\big\} \cup \big\{\{l\}\big\}.$$

In der obigen Definition muss $\{l\}$ in das Ergebnis eingefügt werden, weil die Menge $\big\{C \in K \mid l \notin C\big\}$ die Unit-Klausel $\{l\}$ nicht enthält. Die beiden Klausel-Mengen $\texttt{subsume}(K, l)$ und $K$ sind genau dann äquivalent, wenn $\{l\} \in K$ gilt. Eine Klauselmenge $K$ werden wir daher nur dann mit Hilfe des Ausdrucks $\texttt{subsume}(K, l)$ vereinfachen, wenn die Unit-Klausel $\{l\}$ in der Menge $K$ enthalten ist.

### 4.6.3 Vereinfachung durch Fallunterscheidung

Ein Kalkül, der nur mit Unit-Schnitten und Subsumption arbeitet, ist nicht widerlegungs-vollständig. Wir brauchen daher eine weitere Möglichkeit, Klausel-Mengen zu vereinfachen. Eine solche Möglichkeit bietet das Prinzip der Fallunterscheidung. Dieses Prinzip basiert auf dem folgenden Satz.

**Satz 26** *Ist $K$ eine Menge von Klauseln und ist $p$ eine aussagenlogische Variable, so ist $K$ genau dann erfüllbar, wenn $K \cup \big\{\{p\}\big\}$ oder $K \cup \big\{\{\neg p\}\big\}$ erfüllbar ist.*

**Beweis**:

"$\Rightarrow$": Ist $K$ erfüllbar durch eine Belegung $\mathcal{I}$, so gibt es für $\mathcal{I}(p)$ zwei Möglichkeiten, denn $\mathcal{I}(p)$ ist entweder wahr oder falsch. Falls $\mathcal{I}(p) = \texttt{True}$ ist, ist damit auch die Menge $K \cup \big\{\{p\}\big\}$ erfüllbar, andernfalls ist $K \cup \big\{\{\neg p\}\big\}$ erfüllbar.

"$\Leftarrow$": Da $K$ sowohl eine Teilmenge von $K \cup \big\{\{p\}\big\}$ als auch von $K \cup \big\{\{\neg p\}\big\}$ ist, ist klar, dass $K$ erfüllbar ist, wenn eine dieser Mengen erfüllbar sind. □

Wir können nun eine Menge $K$ von Klauseln dadurch vereinfachen, dass wir eine aussagenlogische Variable $p$ wählen, die in $K$ vorkommt. Anschließend bilden wir die Mengen

$$K_1 := K \cup \big\{\{p\}\big\} \quad \text{und} \quad K_2 := K \cup \big\{\{\neg p\}\big\}$$

und untersuchen rekursiv ob $K_1$ erfüllbar ist. Falls wir eine Lösung für $K_1$ finden, ist dies auch eine Lösung für die ursprüngliche Klausel-Menge $K$ und wir haben unser Ziel erreicht. Andernfalls untersuchen wir rekursiv ob $K_2$ erfüllbar ist. Falls wir eine Lösung finden, ist dies auch eine Lösung von $K$. Wenn wir weder für $K_1$ noch für $K_2$ eine Lösung finden, dann kann auch $K$ keine Lösung haben, denn jede Lösung $\mathcal{I}$ von $K$ muss die Variable $p$ entweder wahr oder falsch machen. Die rekursive Untersuchung von $K_1$ bzw. $K_2$ ist leichter als die Untersuchung von $K$, weil wir ja in $K_1$ und $K_2$ mit den Unit-Klausel $\{p\}$ bzw. $\{\neg p\}$ sowohl Unit-Subsumptionen als auch Unit-Schnitte durchführen können und dadurch diese Mengen vereinfacht werden.

### 4.6.4 Der Algorithmus

Wir können jetzt den Algorithmus von Davis und Putnam skizzieren. Gegeben sei eine Menge $K$ von Klauseln. Gesucht ist dann eine Lösung von $K$. Wir suchen also eine Belegung $\mathcal{I}$, so dass gilt:

$$\mathcal{I}(C) = \texttt{True} \quad \text{für alle } C \in K.$$

Das Verfahren von Davis und Putnam besteht nun aus den folgenden Schritten.

1. Führe alle Unit-Schnitte und Unit-Subsumptionen aus, die mit Klauseln aus $K$ möglich sind.
2. Falls $K$ jetzt trivial ist, sind wir fertig.
3. Andernfalls wählen wir eine aussagenlogische Variable $p$, die in $K$ auftritt.
   (a) Jetzt versuchen wir rekursiv, die Klausel-Menge
   $$K \cup \big\{\{p\}\big\}$$
   zu lösen. Falls diese gelingt, haben wir eine Lösung von $K$.
   (b) Andernfalls versuchen wir, die Klausel-Menge
   $$K \cup \big\{\{\neg p\}\big\}$$
   zu lösen. Wenn auch dies fehlschlägt, ist $K$ unlösbar. Andernfalls haben wir eine Lösung von $K$.

Für die Implementierung ist es zweckmäßig, die beiden oben definierten Funktionen `unitCut()` und `subsume()` zu einer Funktion zusammen zu fassen. Wir definieren daher die Funktion

$$\texttt{reduce} : 2^{\mathcal{K}} \times \mathcal{L} \to 2^{\mathcal{K}}$$

wie folgt:

$$\texttt{reduce}(K, l) = \Big\{ C \backslash \big\{\overline{l}\big\} \mid C \in K \wedge \overline{l} \in C \Big\} \cup \Big\{ C \in K \mid \overline{l} \notin C \wedge l \notin C \Big\} \cup \big\{\{l\}\big\}.$$

Die Menge enthält also einerseits die Ergebnisse von Schnitten mit der Unit-Klausel $\{l\}$ und andererseits nur die Klauseln $C$, die mit $l$ nichts zu tun haben, weil weder $l \in C$ noch $\overline{l} \in C$ gilt. Außerdem fügen wir noch die Unit-Klausel $\{l\}$ hinzu. Dadurch erreichen wir, dass die beiden Mengen $K$ und $\texttt{reduce}(K, l)$ logisch äquivalent sind, falls $\{l\} \in K$ gilt.

### 4.6.5 Ein Beispiel

Zur Veranschaulichung demonstrieren wir das Verfahren von Davis und Putnam an einem Beispiel. Die Menge $K$ sei wie folgt definiert:

$$K := \Big\{\{p,q,s\},\ \{\neg p,r,\neg t\},\ \{r,s\},\ \{\neg r,q,\neg p\},\{\neg s,p\},\ \{\neg p,\neg q,s,\neg r\},\ \{p,\neg q,s\},\ \{\neg r,\neg s\},\ \{\neg p,\neg s\}\Big\}.$$

Wir zeigen nun mit dem Verfahren von Davis und Putnam, dass $K$ nicht lösbar ist. Da die Menge $K$ keine Unit-Klauseln enthält, ist im ersten Schritt nichts zu tun. Da $K$ nicht trivial ist, sind wir noch nicht fertig. Also gehen wir jetzt zu Schritt 3 und wählen eine aussagenlogische Variable, die in $K$ auftritt. An dieser Stelle ist es sinnvoll eine Variable zu wählen, die in möglichst vielen Klauseln von $K$ auftritt. Wir wählen daher die aussagenlogische Variable $p$.

1. Zunächst bilden wir die Menge

   $$K_0 := K \cup \big\{\{p\}\big\}$$

   und versuchen, diese Menge zu lösen. Dazu bilden wir

   $$K_1 := \texttt{reduce}\big(K_0, p\big) = \Big\{\{r, \neg t\},\ \{r, s\},\ \{\neg r, q\},\ \{\neg q, s, \neg r\},\ \{\neg r, \neg s\},\ \{\neg s\},\ \{p\}\ \Big\}.$$

   Die Klausel-Menge $K_1$ enthält die Unit-Klausel $\{\neg s\}$, so dass wir als nächstes mit dieser Klausel reduzieren können:

   $$K_2 := \texttt{reduce}\big(K_1, \neg s\big) = \Big\{\{r, \neg t\},\ \{r\},\ \{\neg r, q\},\ \{\neg q, \neg r\},\ \{\neg s\},\ \{p\}\Big\}.$$

   Hier haben wir nun die neue Unit-Klausel $\{r\}$, mit der wir weiter reduzieren:

   $$K_3 := \mathit{reduce}\big(K_2, r\big) = \Big\{\{r\},\ \{q\},\ \{\neg q\},\ \{\neg s\},\ \{p\}\Big\}$$

   Da $K_3$ die Unit-Klausel $\{q\}$ enthält, reduzieren wir jetzt mit $q$:

   $$K_4 := \mathit{reduce}\big(K_2, q\big) = \Big\{\{r\},\ \{q\},\ \{\},\ \{\neg s\},\ \{p\}\Big\}.$$

   Die Klausel-Menge $K_4$ enthält die leere Klausel und ist damit unlösbar.

2. Also bilden wir jetzt die Menge

   $$K_5 := K \cup \big\{\{\neg p\}\big\}$$

   und versuchen, diese Menge zu lösen. Dazu bilden wir

   $$K_6 = \mathit{reduce}\big(K_5, \neg p\big) = \Big\{\{q, s\},\ \{r, s\},\ \{\neg s\},\ \{\neg q, s\},\ \{\neg r, \neg s\},\ \{\neg p\}\ \Big\}.$$

   Die Menge $K_6$ enthält die Unit-Klausel $\{\neg s\}$. Wir bilden daher

   $$K_7 = \mathit{reduce}\big(K_6, \neg s\big) = \Big\{\{q\},\ \{r\},\ \{\neg s\},\ \{\neg q\},\ \{\neg p\}\ \Big\}.$$

   Die Menge $K_7$ enthält die neue Unit-Klausel $\{q\}$, mit der wir als nächstes reduzieren:

   $$K_8 = \mathit{reduce}\big(K_7, q\big) = \Big\{\{q\},\ \{r\},\ \{\neg s\},\ \{\},\ \{\neg p\}\ \Big\}.$$

   Da $K_8$ die leere Klausel enthält, ist $K_8$ und damit auch die ursprünglich gegebene Menge $K$ unlösbar.

Bei diesem Beispiel hatten wir Glück, denn wir mussten nur eine einzige Fallunterscheidung durchführen. Bei komplexeren Beispielen ist es häufig so, dass wir innerhalb einer Fallunterscheidung eine oder mehrere weitere Fallunterscheidungen durchführen müssen.

### 4.6.6 Implementierung des Algorithmus von Davis und Putnam

Wir zeigen jetzt die Implementierung der Funktion `solve`, mit der die Frage, ob eine Menge von Klauseln erfüllbar ist, beantwortet werden kann. Die Implementierung ist in Abbildung 4.14 auf Seite 80 gezeigt. Die Funktion erhält zwei Argumente: Die Mengen `Clauses` und `Variables`. Hier ist `Clauses` eine Menge von Klauseln und `Variables` ist eine Menge von Variablen. Falls die Menge `Clauses` erfüllbar ist, so liefert der Aufruf

```
solve(Clauses, Variables)
```

eine Menge von Unit-Klauseln `Result`, so dass jede Belegung $\mathcal{I}$, die alle Unit-Klauseln aus `Result`

erfüllt, auch alle Klauseln aus der Menge `Clauses` erfüllt. Falls die Menge `Clauses` nicht erfüllbar ist, liefert der Aufruf

```
solve(Clauses, Variables)
```

als Ergebnis die Menge $\{\{\}\}$ zurück, denn die leere Klausel repräsentiert die unerfüllbare Formel $\bot$.

Sie fragen sich vielleicht, wozu wir in der Funktion `solve` die Menge `Variables` brauchen. Der Grund ist, dass wir uns bei den rekursiven Aufrufen merken müssen, welche Variablen wir schon für Fallunterscheidungen benutzt haben. Diese Variablen sammeln wir in der Menge `Variables`.

```
1  def solve(Clauses, Variables):
2      S      = saturate(Clauses);
3      empty  = frozenset()
4      Falsum = {empty}
5      if empty in S:                  # S is inconsistent
6          return Falsum
7      if all(len(C) == 1 for C in S): # S is trivial,
8          return S                    # hence it is a solution.
9      p      = selectVariable(S, Variables)
10     negP   = complement(p)
11     Result = solve(S | { frozenset({p}) }, Variables | { p })
12     if Result != Falsum:
13         return Result
14     return solve(S | { frozenset({negP}) }, Variables| { p })
```

Figure 4.14: Die Funktion `solve`

Die in Abbildung 4.14 gezeigte Implementierung funktioniert wie folgt:

1. In Zeile 2 reduzieren wir mit Hilfe der Methode `saturate` solange wie möglich die gegebene Klausel-Menge `Clauses` mit Hilfe von Unit-Schnitten und entfernen alle Klauseln, die durch Unit-Klauseln subsumiert werden.

2. Anschließend testen wir in Zeile 5, ob die so vereinfachte Klausel-Menge `S` die leere Klausel enthält und geben in diesem Fall als Ergebnis die Menge $\{\{\}\}$ zurück.

3. Dann testen wir in Zeile 7, ob bereits alle Klauseln $C$ aus der Menge `S` Unit-Klauseln sind. Wenn dies so ist, dann ist die Menge `S` trivial und wir geben diese Menge als Ergebnis zurück.

4. Andernfalls wählen wir in Zeile 9 eine Variable `p`, die in der Menge `S` vorkommt, die wir aber noch nicht benutzt haben. Wir untersuchen dann in Zeile 11 rekursiv, ob die Menge

   $$\texttt{S} \cup \{\{\texttt{p}\}\}$$

   lösbar ist. Dabei gibt es zwei Fälle:

   (a) Falls diese Menge lösbar ist, geben wir die Lösung dieser Menge als Ergebnis zurück.

   (b) Sonst prüfen wir rekursiv, ob die Menge

$$S \cup \Big\{ \{\overline{p}\} \Big\}$$

lösbar ist. Ist diese Menge lösbar, so ist diese Lösung auch eine Lösung der Menge `Clauses` und wir geben diese Lösung zurück. Ist die Menge unlösbar, dann muss auch die Menge `Clauses` unlösbar sein.

Wir diskutieren nun die Hilfsprozeduren, die bei der Implementierung der Funktion `solve` verwendet wurden. Als erstes besprechen wir die Funktion `saturate`. Diese Funktion erhält eine Menge $S$ von Klauseln als Eingabe und führt alle möglichen Unit-Schnitte und Unit-Subsumptionen durch. Die Funktion `saturate` ist in Abbildung 4.15 auf Seite 81 gezeigt.

```
1  def saturate(Clauses):
2      S     = Clauses.copy()
3      Units = { C for C in S if len(C) == 1 }
4      Used  = set()
5      while len(Units) > 0:
6          unit  = Units.pop()
7          Used |= { unit }
8          l     = arb(unit)
9          S     = reduce(S, l)
10         Units = { C for C in S if len(C) == 1 } - Used
11     return S
```

Figure 4.15: Die Funktion `saturate`.

Die Implementierung von `saturate` funktioniert wie folgt:

1. Zunächst kopieren wir die Menge `Clauses` in die Variable `S`. Dies ist notwendig, da wir die Menge `S` später verändern werden. Die Funktion `saturate` soll das Argument `Clauses` aber nicht verändern und muss daher eine Kopie der Menge `S` anlegen.

2. Dann berechnen wir in Zeile 3 die Menge `Units` aller Unit-Klauseln.

3. Anschließend initialisieren wir in Zeile 4 die Menge `Used` als die leere Menge. In dieser Menge merken wir uns, welche Unit-Klauseln wir schon für Unit-Schnitte und Subsumptionen benutzt haben.

4. Solange die Menge `Units` der Unit-Klauseln nicht leer ist, wählen wir in Zeile 6 mit Hilfe der Funktion `pop` eine beliebige Unit-Klausel `unit` aus der Menge `Units` aus und entfernen diese Unit-Klausel aus der Menge `Units`.

5. In Zeile 7 fügen wir die Klausel `unit` zu der Menge `Used` der benutzten Klausel hinzu.

6. In Zeile 8 extrahieren mit der Funktion `arb` das Literal `l` der Klausel `Unit`. Die Funktion `arb` liefert ein beliebiges Element der Menge zurück, das dieser Funktion als Argument übergeben wird. Enthält diese Menge nur ein Element, so wird also dieses Element zurück gegeben.

7. In Zeile 9 wird die eigentliche Arbeit durch einen Aufruf der Funktion `reduce` geleistet. Diese Funktion berechnet alle Unit-Schnitte, die mit der Unit-Klausel $\{\texttt{l}\}$ möglich sind und entfernt darüber hinaus alle Klauseln, die durch die Unit-Klausel $\{\texttt{l}\}$ subsumiert werden.

8. Wenn die Unit-Schnitte mit der Unit-Klausel $\{\texttt{l}\}$ berechnet werden, können neue Unit-Klauseln entstehen, die wir in Zeile 10 aufsammeln. Wir sammeln dort aber nur die Unit-Klauseln auf, die wir noch nicht benutzt haben.

9. Die Schleife in den Zeilen 5 – 10 wird nun solange durchlaufen, wie wir Unit-Klauseln finden, die wir noch nicht benutzt haben.

10. Am Ende geben wir die verbliebene Klauselmenge als Ergebnis zurück.

Die dabei verwendete Funktion `reduce` ist in Abbildung 4.16 gezeigt. Im vorigen Abschnitt hatten wir die Funktion $\mathit{reduce}(S, l)$, die eine Klausel-Menge `Cs` mit Hilfe des Literals $l$ reduziert, als

$$\mathit{reduce}(\texttt{Cs}, l) = \Big\{ C \backslash \{\overline{l}\} \mid C \in \texttt{Cs} \wedge \overline{l} \in C \Big\} \cup \Big\{ C \in \texttt{Cs} \mid \overline{l} \notin C \wedge l \notin C \Big\} \cup \Big\{ \{l\} \Big\}$$

definiert. Die Implementierung setzt diese Definition unmittelbar um.

```
1  def reduce(Clauses, l):
2      lBar = complement(l)
3      return   { C - { lBar } for C in Clauses if lBar in C }          \
4             | { C for C in Clauses if lBar not in C and l not in C }  \
5             | { frozenset({l}) }
```

Figure 4.16: Die Funktion `reduce`.

Die Implementierung des Algorithmus von Davis und Putnam benutzt außer den bisher diskutierten Funktionen noch zwei weitere Hilfsprozeduren, deren Implementierung in Abbildung 4.17 auf Seite 83 gezeigt wird.

1. Die Funktion `selectLiteral` wählt eine beliebige Variable aus einer gegeben Menge `Clauses` von Klauseln aus, das außerdem nicht in der Menge `Forbidden` von den Variablen vorkommen darf, die bereits benutzt worden sind. Dazu iterieren wir zunächst über alle Klauseln $C$ aus der Menge `Clauses` und dann über alle Literale $l$ der Klausel $C$. Aus diesen Literalen extrahieren wir die darin enthaltene Variable mit Hilfe der Funktion `extractVariable`. Anschließend wird eine beliebige Variable zurück gegeben.

2. Die Funktion `arb` gibt ein nicht näher spezifiziertes Element einer Menge zurück.

Die oben dargestellte Version des Verfahrens von Davis und Putnam lässt sich in vielerlei Hinsicht verbessern. Aus Zeitgründen können wir auf solche Verbesserungen nicht weiter eingehen. Der interessierte Leser sei hier auf die folgende Arbeit von Moskewicz et.al. [MMZ+01] verwiesen:

> *Chaff: Engineering an Efficient SAT Solver*
> von M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik

**Aufgabe 10**: Die Klausel-Menge $M$ sei wie folgt gegeben:

```
def selectLiteral(Clauses, Forbidden):
    Variables = { extractVariable(l) for C in Clauses for l in C } - Forbidden
    return arb(Variables)

def arb(S):
    "Return some member from the set S."
    for x in S:
        return x
```

Figure 4.17: Die Funktionen `select` und `negateLiteral`

$$M := \big\{\ \{r,p,s\},\{r,s\},\{q,p,s\},\{\neg p,\neg q\},\{\neg p,s,\neg r\},\{p,\neg q,r\},$$
$$\{\neg r,\neg s,q\},\{p,q,r,s\},\{r,\neg s,q\},\{\neg r,s,\neg q\},\{s,\neg r\}\big\}$$

Überprüfen Sie mit dem Verfahren von Davis und Putnam, ob die Menge $M$ widersprüchlich ist. ◇

## 4.7 Das 8-Damen-Problem

In diesem Abschnitt zeigen wir, wie bestimmte kombinatorische Probleme als aussagenlogische Fragestellungen formuliert werden können. Diese können dann anschließend mit dem Algorithmus von Davis und Putnam gelöst werden. Als konkretes Beispiel betrachten wir das 8-Damen-Problem. Dabei geht es darum, 8 Damen so auf einem Schach-Brett aufzustellen, dass keine Dame eine andere Dame schlagen kann. Beim Schach-Spiel kann eine Dame dann eine andere Figur schlagen, wenn diese Figur entweder

- in derselben Reihe,
- in derselben Spalte oder
- in derselben Diagonale

wie die Dame steht. Abbildung 4.18 auf Seite 84 zeigt ein Schachbrett, in dem sich in der dritten Reihe in der vierten Spalte eine Dame befindet. Diese Dame kann auf alle die Felder ziehen, die mit Pfeilen markierte sind, und kann damit Figuren, die sich auf diesen Feldern befinden, schlagen.

Als erstes überlegen wir uns, wie wir ein Schach-Brett mit den darauf positionierten Damen aussagenlogisch repräsentieren können. Eine Möglichkeit besteht darin, für jedes Feld eine aussagenlogische Variable einzuführen. Diese Variable drückt aus, dass auf dem entsprechenden Feld eine Dame steht. Wir ordnen diesen Variablen wie folgt Namen zu: Die Variable, die das $j$-te Feld in der $i$-ten Reihe bezeichnet, stellen wir durch den String

$$\texttt{'Q<}i\texttt{,}j\texttt{>'} \quad \text{mit } i,j \in \{1,\cdots,8\}$$

dar. Wir nummerieren die Reihen dabei von oben beginnend von 1 bis 8 durch, während die Spalten von links nach rechts numeriert werden. Abbildung 4.20 auf Seite 85 zeigt die Zuordnung der Variablen zu den Feldern. Die in Abbildung 4.19 gezeigte Funktion $\texttt{var}(r,c)$ berechnet die Variable, die ausdrückt, dass sich in Reihe $r$ und Spalte $c$ eine Dame befindet.

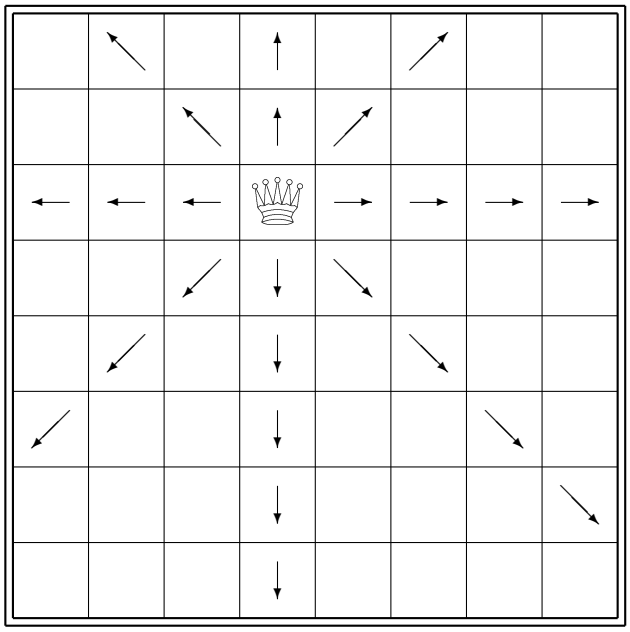

Figure 4.18: Das 8-Damen-Problem

```
1  def var(row, col):
2      return 'Q<' + str(row) + ',' + str(col) + '>'
```

Figure 4.19: Die Funktion `var` Zur Berechnung der aussagenlogischen Variablen

Als nächstes überlegen wir uns, wie wir die einzelnen Bedingungen des 8-Damen-Problems als aussagenlogische Formeln kodieren können. Letztlich lassen sich alle Aussagen der Form

- "in einer Reihe steht höchstens eine Dame",
- "in einer Spalte steht höchstens eine Dame", oder
- "in einer Diagonale steht höchstens eine Dame"

auf dasselbe Grundmuster zurückführen: Ist eine Menge von aussagenlogischen Variablen

$$V = \{x_1, \cdots, x_n\}$$

gegeben, so brauchen wir eine Formel die aussagt, dass höchstens eine der Variablen aus $V$ den Wert `True` hat. Das ist aber gleichbedeutend damit, dass für jedes Paar $x_i, x_j \in V$ mit $x_i \neq x_j$ die folgende Formel gilt:

$$\neg(x_i \wedge x_j).$$

Diese Formel drückt aus, dass die Variablen $x_i$ und $x_j$ nicht gleichzeitig den Wert `True` annehmen. Nach den DeMorgan'schen Gesetzen gilt

$$\neg(x_i \wedge x_j) \leftrightarrow \neg x_i \vee \neg x_j$$

und die Klausel auf der rechten Seite dieser Äquivalenz schreibt sich in Mengen-Schreibweise als

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Q<1,1> | Q<1,2> | Q<1,3> | Q<1,4> | Q<1,5> | Q<1,6> | Q<1,7> | Q<1,8> |
| Q<2,1> | Q<2,2> | Q<2,3> | Q<2,4> | Q<2,5> | Q<2,6> | Q<2,7> | Q<2,8> |
| Q<3,1> | Q<3,2> | Q<3,3> | Q<3,4> | Q<3,5> | Q<3,6> | Q<3,7> | Q<3,8> |
| Q<4,1> | Q<4,2> | Q<4,3> | Q<4,4> | Q<4,5> | Q<4,6> | Q<4,7> | Q<4,8> |
| Q<5,1> | Q<5,2> | Q<5,3> | Q<5,4> | Q<5,5> | Q<5,6> | Q<5,7> | Q<5,8> |
| Q<6,1> | Q<6,2> | Q<6,3> | Q<6,4> | Q<6,5> | Q<6,6> | Q<6,7> | Q<6,8> |
| Q<7,1> | Q<7,2> | Q<7,3> | Q<7,4> | Q<7,5> | Q<7,6> | Q<7,7> | Q<7,8> |
| Q<8,1> | Q<8,2> | Q<8,3> | Q<8,4> | Q<8,5> | Q<8,6> | Q<8,7> | Q<8,8> |

Figure 4.20: Zuordnung der Variablen

$$\{\neg x_i, \neg x_j\}.$$

Die Formel, die für eine Variablen-Menge $V$ ausdrückt, dass keine zwei verschiedenen Variablen gleichzeitig wahr sind, kann daher als Klausel-Menge in der Form

$$\big\{\, \{\neg p, \neg q\} \mid p \in V \wedge\ q \in V \wedge p \neq q \big\}$$

geschrieben werden. Wir setzen diese Überlegungen in eine *Python*-Funktion um. Die in Abbildung 4.21 gezeigte Funktion `atMostOne()` bekommt als Eingabe eine Menge $S$ von aussagenlogischen Variablen. Der Aufruf `atMostOne`$(S)$ berechnet eine Menge von Klauseln. Diese Klauseln sind genau dann wahr, wenn höchstens eine der Variablen aus $S$ den Wert `True` hat.

Mit Hilfe der Funktion `atMostOne` können wir nun die Funktion `atMostOneInRow` implementieren. Der Aufruf

```
1  def atMostOne(S):
2      return { frozenset({('¬',p), ('¬', q)}) for p in S
3                                              for q in S
4                                              if  p != q
5              }
```

Figure 4.21: Die Funktion `atMostOne`

`atMostOneInRow(row, n)`

berechnet für eine gegebene Reihe `row` bei einer Brettgröße von `n` eine Formel, die ausdrückt, dass in der Reihe `row` höchstens eine Dame steht. Abbildung 4.22 zeigt die Funktion `atMostOneInRow`: Wir sammeln alle Variablen der durch `row` spezifizierten Reihe in der Menge

$$\big\{\texttt{var}(\texttt{row}, j) \mid j \in \{1, \cdots, n\}\big\}$$

auf und rufen mit dieser Menge die Funktion `atMostOne()` auf, die das Ergebnis als Menge von Klauseln liefert.

```
1  def atMostOneInRow(row, n):
2      return atMostOne({ var(row, col) for col in range(1,n+1) })
```

Figure 4.22: Die Funktion `atMostOneInRow`

Als nächstes berechnen wir eine Formel die aussagt, dass mindestens eine Dame in einer gegebenen Spalte steht. Für die erste Spalte hätte diese Formel die Form

$$\texttt{Q<1,1>} \vee \texttt{Q<2,1>} \vee \texttt{Q<3,1>} \vee \texttt{Q<4,1>} \vee \texttt{Q<5,1>} \vee \texttt{Q<6,1>} \vee \texttt{Q<7,1>} \vee \texttt{Q<8,1>}$$

und wenn allgemein eine Spalte $c$ mit $c \in \{1, \cdots, 8\}$ gegeben ist, lautet die Formel

$$\texttt{Q<1,c>} \vee \texttt{Q<2,c>} \vee \texttt{Q<3,c>} \vee \texttt{Q<4,c>} \vee \texttt{Q<5,c>} \vee \texttt{Q<6,c>} \vee \texttt{Q<7,c>} \vee \texttt{Q<8,c>}.$$

Schreiben wir diese Formel in der Mengenschreibweise als Menge von Klauseln, so erhalten wir

$$\big\{\{\texttt{Q<1,c>}, \texttt{Q<2,c>}, \texttt{Q<3,c>}, \texttt{Q<4,c>}, \texttt{Q<5,c>}, \texttt{Q<6,c>}, \texttt{Q<7,c>}, \texttt{Q<8,c>}\}\big\}.$$

Abbildung 4.23 zeigt eine *Python*-Funktion, die für eine gegebene Spalte `col` und eine gegebene Brettgröße `n` die entsprechende Klausel-Menge berechnet. Der Schritt, von einer einzelnen Klausel zu einer Menge von Klauseln überzugehen ist notwendig, denn unsere Implementierung des Algorithmus von Davis und Putnam arbeitet mit einer Menge von Klauseln.

An dieser Stelle erwarten Sie vielleicht, dass wir noch Formeln angeben die ausdrücken, dass in einer gegebenen Spalte höchstens eine Dame steht und dass in jeder Reihe mindestens eine Dame steht. Solche Formeln sind aber unnötig, denn wenn wir wissen, dass in jeder Spalte mindestens eine Dame steht, so wissen wir bereits, dass auf dem Brett mindestens 8 Damen stehen. Wenn wir nun zusätzlich wissen, dass in jeder Reihe höchstens eine Dame steht, so ist automatisch klar, dass höchstens 8 Damen auf dem Brett stehen. Damit stehen also insgesamt genau 8 Damen auf dem

```
1   def oneInColumn(col, n):
2       return { frozenset({ var(row, col) for row in range(1,n+1) }) }
```

Figure 4.23: Die Funktion `oneInColumn`

Brett. Dann kann aber in jeder Spalte nur höchstens eine Dame stehen, denn sonst hätten wir mehr als 8 Damen auf dem Brett und genauso muss in jeder Reihe mindestens eine Dame stehen, denn sonst würden wir in der Summe nicht auf 8 Damen kommen.

Als nächstes überlegen wir uns, wie wir die Variablen, die auf derselben Diagonale stehen, charakterisieren können. Es gibt grundsätzlich zwei verschiedene Arten von Diagonalen: Absteigende Diagonalen und aufsteigende Diagonalen. Wir betrachten zunächst die aufsteigenden Diagonalen. Die längste aufsteigende Diagonale, wir sagen dazu auch Hauptdiagonale, besteht im Fall eines $8 \times 8$-Bretts aus den Variablen

`Q<8,1>, Q<7,2>, Q<6,3>, Q<5,4>, Q<4,5>, Q<3,6>, Q<2,7>, Q<1,8>`.

Die Indizes $r$ und $c$ der Variablen $\texttt{Q}(r, c)$ erfüllen offenbar die Gleichung

$$r + c = 9.$$

Allgemein erfüllen die Indizes der Variablen einer aufsteigenden Diagonale, die mehr als ein Feld enthält, die Gleichung

$$r + c = k,$$

wobei $k$ im Falle eines $8 \times 8$ Schach-Bretts einen Wert aus der Menge $\{3, \cdots, 15\}$ annimmt. Den Wert $k$ geben wir als Argument bei der Funktion `atMostOneInRisingDiagonal` mit. Diese Funktion ist in Abbildung 4.24 gezeigt.

```
1   def atMostOneInRisingDiagonal(k, n):
2       S = { var(row, col) for row in range(1, n+1)
3                           for col in range(1, n+1)
4                           if  row + col == k
5           }
6       return atMostOne(S)
```

Figure 4.24: Die Funktion `atMostOneInUpperDiagonal`

Um zu sehen, wie die Variablen einer fallenden Diagonale charakterisiert werden können, betrachten wir die fallende Hauptdiagonale, die aus den Variablen

`Q<1,1>, Q<2,2>, Q<3,3>, Q<4,4>, Q<5,5>, Q<6,6>, Q<7,7>, Q<8,8>`

besteht. Die Indizes $r$ und $c$ dieser Variablen erfüllen offenbar die Gleichung

$$r - c = 0.$$

Allgemein erfüllen die Indizes der Variablen einer absteigenden Diagonale die Gleichung

$$r - c = k,$$

wobei $k$ einen Wert aus der Menge $\{-6, \cdots, 6\}$ annimmt. Den Wert $k$ geben wir als Argument bei der Funktion `atMostOneInLowerDiagonal` mit. Diese Funktion ist in Abbildung 4.25 gezeigt.

```
1  def atMostOneInFallingDiagonal(k, n):
2      S = { var(row, col) for row in range(1, n+1)
3                          for col in range(1, n+1)
4                          if  row - col == k
5          }
6      return atMostOne(S)
```

Figure 4.25: Die Funktion `atMostOneInLowerDiagonal`

Jetzt sind wir in der Lage, unsere Ergebnisse zusammen zu fassen: Wir können eine Menge von Klauseln konstruieren, die das 8-Damen-Problem vollständig beschreiben. Abbildung 4.26 zeigt die Implementierung der Funktion `allClauses`. Der Aufruf

$$\texttt{allClauses}(n)$$

rechnet für ein Schach-Brett der Größe $n$ eine Menge von Klauseln aus, die genau dann erfüllt sind, wenn auf dem Schach-Brett

1. in jeder Reihe höchstens eine Dame steht (Zeile 2),
2. in jeder absteigenden Diagonale höchstens eine Dame steht (Zeile 3),
3. in jeder aufsteigenden Diagonale höchstens eine Dame steht (Zeile 4) und
4. in jeder Spalte mindestens eine Dame steht (Zeile 5).

Die Ausdrücke in den einzelnen Zeilen liefern Listen, deren Elemente Klausel-Mengen sind. Was wir als Ergebnis brauchen, ist aber eine Klausel-Menge und keine Liste von Klausel-Mengen. Daher wandeln wir in Zeile 6 die Liste `All` in eine Menge von Klauseln um.

```
1  def allClauses(n):
2      All = [ atMostOneInRow(row, n)           for row in range(1, n+1)        ] \
3          + [ atMostOneInFallingDiagonal(k, n) for k in range(-(n-2), (n-2)+1) ] \
4          + [ atMostOneInRisingDiagonal(k, n)  for k in range(3, (2*n-1)+1)    ] \
5          + [ oneInColumn(col, n)              for col in range(1, n+1)        ]
6      return { clause for S in All for clause in S }
```

Figure 4.26: Die Funktion `allClauses`

Als letztes zeigen wir in Abbildung 4.27 die Funktion `queens`, mit der wir das 8-Damen-Problem lösen können.

1. Zunächst kodieren wir das Problem als eine Menge von Klauseln, die genau dann lösbar ist, wenn das Problem eine Lösung hat.

2. Anschließend berechnen wir die Lösung mit Hilfe der Funktion `sovle` aus dem Modul `davisPutnam`, das wir als `dp` importiert haben.

3. Zum Schluss wird die berechnete Lösung mit Hilfe der Funktion `printBoard` ausgedruckt.

   Hierbei ist `printBoard` eine Funktion, welche die Lösung in lesbarere Form ausdruckt. Das funktioniert allerdings nur, wenn ein Font verwendet wird, bei dem alle Zeichen die selbe Breite haben. Diese Funktion ist der Vollständigkeit halber in Abbildung 4.28 gezeigt, wir wollen die Implementierung aber nicht weiter diskutieren.

Das vollständige Programm finden Sie als Jupyter Notebook auf meiner Webseite unter dem Namen `N-Queens.ipynb`.

```
def queens(n):
    "Solve the n queens problem."
    Clauses  = allClauses(n)
    Solution = dp.solve(Clauses, set())
    if Solution != { frozenset() }:
        return Solution
    else:
        print(f'The problem is not solvable for {n} queens!')
```

Figure 4.27: Die Funktion `queens` zur Lösung des $n$-Damen-Problems.

```
import chess

def show_solution(Solution, n):
    board = chess.Board(None)  # create empty chess board
    queen = chess.Piece(chess.QUEEN, True)
    for row in range(1, n+1):
        for col in range(1, n+1):
            field_number = (row - 1) * 8 + col - 1
            if frozenset({ var(row, col) }) in Solution:
                board.set_piece_at(field_number, queen)
    display(board)
```

Figure 4.28: Die Funktion `show_solution()`.

Die durch den Aufruf `solve`($Clauses, \{\}$) berechnete Menge `solution` enthält für jede der Variablen `'Q<`$r$`,`$c$`>'` entweder die Unit-Klausel $\{$`'Q<`$r$`,`$c$`>'`$\}$ (falls auf diesem Feld eine Dame steht) oder aber die Unit-Klausel $\{$`('¬', 'Q<`$r$`,`$c$`>')`$\}$ (falls das Feld leer bleibt). Eine graphische Darstellung

einer berechneten Lösungen sehen Sie in Abbildung 4.29. Diese graphische Darstellung habe ich mit Hilfe der Bibliothek `python-chess` und der Funktion `show_solution`, die in Abbildung 4.28 gezeigt ist, erzeugt.

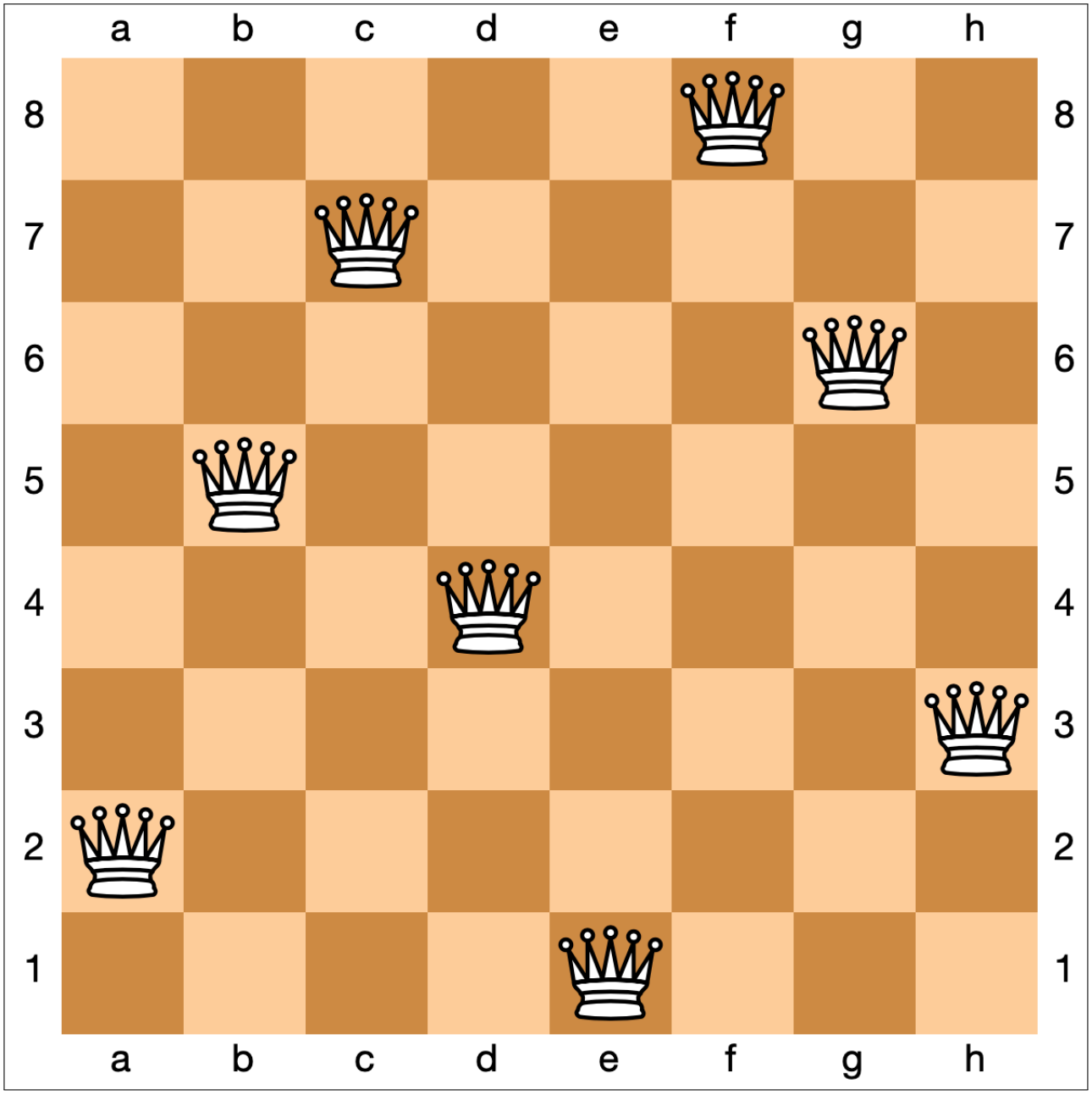


Figure 4.29: Eine Lösung des 8-Damen-Problems.

Jessica Roth und Koen Loogman (das sind zwei ehemalige DHBW-Studenten) haben eine Animation des Verfahren von Davis und Putnam implementiert. Sie können diese Animation unter der Adresse

https://koenloogman.github.io/Animation-Of-N-Queens-Problem-In-JavaScript/

im Netz finden und ausprobieren.

Das 8-Damen-Problem ist natürlich nur eine spielerische Anwendung der Aussagen-Logik. Trotzdem zeigt es die Leistungsfähigkeit des Algorithmus von Davis und Putnam sehr gut, denn die Menge der Klauseln, die von der Funktion `allClauses` berechnet wird, besteht aus 512 verschiedenen Klauseln. In dieser Klausel-Menge kommen 64 verschiedene Variablen vor.

In der Praxis gibt es viele Probleme, die sich in ganz ähnlicher Weise auf die Lösung einer Menge von Klauseln zurückführen lassen. Dazu gehört zum Beispiel das Problem, einen Stundenplan zu erstellen, der gewissen Nebenbedingungen genügt. Verallgemeinerungen des Stundenplan-Problems werden in der Literatur als Scheduling-Probleme bezeichnet. Die effiziente Lösung solcher Probleme ist Gegenstand der aktuellen Forschung.

## 4.8 Reflexion

(a) Wie haben wir die Menge der aussagenlogischen Formeln definiert?

(b) Wie ist die Semantik der aussagenlogischen Formeln festgelegt worden?

(c) Wie können wir aussagenlogische Formeln in *Python* darstellen?

(d) Was ist eine Tautologie?

(e) Was ist eine konjunktive Normalform?

(f) Wie können Sie die konjunktive Normalform einer gegebenen aussagenlogischen Formel berechnen und wie lässt sich diese Berechnung in *Python* implementieren?

(g) Wie haben wir den Beweis-Begriff $M \vdash C$ definiert?

(h) Welche Eigenschaften hat der Beweis-Begriff $\vdash$?

(i) Wann ist eine Menge von Klauseln lösbar?

(j) Wie funktioniert das Verfahren von Davis und Putnam?

(k) Wie können Sie das 8-Damen-Problem als aussagenlogisches Problem formulieren?

# Chapter 5

# Prädikatenlogik

In der Aussagenlogik haben wir die Verknüpfung von elementaren Aussagen mit Junktoren untersucht. Die Prädikatenlogik untersucht zusätzlich auch die Struktur dieser elementaren Aussagen. Dazu werden in der Prädikatenlogik die folgenden zusätzlichen Begriffe eingeführt:

1. Als Bezeichnungen für Objekte werden Terme verwendet.

2. Diese Terme werden aus Objekt-Variablen und Funktions-Zeichen zusammengesetzt. In den folgenden Beispielen ist $x$ eine Objekt-Variable, während *vater* und *mutter* einstellige Funktions-Zeichen sind. *issac* ist ein nullstelliges Funktions-Zeichen:

   $$\mathit{vater}(x), \quad \mathit{mutter}(\mathit{isaac}).$$

   Nullstellige Funktions-Zeichen werden im Folgenden auch als Konstanten bezeichnet und an Stelle von Objekt-Variablen reden wir kürzer nur von Variablen.

3. Verschiedene Objekte werden durch Prädikats-Zeichen in Relation gesetzt. In den folgenden Beispielen benutzen wir die Prädikats-Zeichen *istBruder* und $<$:

   $$\mathit{istBruder}\big(\mathit{albert}, \mathit{vater}(\mathit{bruno})\big), \quad x + 7 < x \cdot 7.$$

   Die dabei entstehenden Formeln werden als atomare Formeln bezeichnet.

4. Atomare Formeln lassen sich durch aussagenlogische Junktoren verknüpfen:

   $$x > 1 \rightarrow x + 7 < x \cdot 7.$$

5. Schließlich werden Quantoren eingeführt, um zwischen existentiell und universell quantifizierten Variablen unterscheiden zu können:

   $$\forall x \in \mathbb{R} : \exists n \in \mathbb{N} : x < n.$$

Dieses Kapitel ist wie folgt aufgebaut:

(a) Wir werden im nächsten Abschnitt die Syntax der prädikatenlogischen Formeln festlegen, wir werden also festlegen, welche Strings wir als aussagenlogische Formeln zulassen.

(b) Im darauf folgenden Abschnitt beschäftigen wir uns mit der Semantik dieser Formeln, dort spezifizieren wir also die Bedeutung der Formeln.

(c) Danach zeigen wir, wie sich die eingeführten Begriffe in *Python* implementieren lassen.

(d) Anschließend diskutieren wir als Anwendung der Prädikaten-Logik das Constraint Programming. Beim Constraint Programming wird ein gegebenes Problem durch prädikatenlogische Formeln beschrieben. Zur Lösung des Problems wird dann ein sogenannter Constraint Solver verwendet.

(e) Anschließend betrachten wir Normalformen prädikatenlogischer Formeln und zeigen, wie Formeln in erfüllbarkeits-äquivalente prädikatenlogische Klauseln umgewandelt werden können.

(f) Zum Abschluss des Kapitels diskutieren wir einen prädikatenlogischen Kalkül, der die Grundlage des automatischen Beweisens in der Prädikaten-Logik ist.

## 5.1 Syntax der Prädikatenlogik

Zunächst definieren wir den Begriff der Signatur. Inhaltlich ist das nichts anderes als eine strukturierte Zusammenfassung von Variablen, Funktions- und Prädikats-Zeichen zusammen mit einer Spezifikation der Stelligkeit dieser Zeichen.

**Definition 27 (Signatur)** *Eine Signatur ist ein 4-Tupel*

$$\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle,$$

*für das Folgendes gilt:*

1. $\mathcal{V}$ *ist die Menge der Objekt-Variablen, die wir der Kürze halber meist nur als Variablen bezeichnen.*
2. $\mathcal{F}$ *ist die Menge der Funktions-Zeichen.*
3. $\mathcal{P}$ *ist die Menge der Prädikats-Zeichen.*
4. *arity ist eine Funktion, die jedem Funktions- und jedem Prädikats-Zeichen seine Stelligkeit zuordnet:*

   $$\mathit{arity} : \mathcal{F} \cup \mathcal{P} \to \mathbb{N}.$$

   *Wir sagen, dass das Funktions- oder Prädikats-Zeichen $f$ ein $n$-stelliges Zeichen ist, falls $\mathit{arity}(f) = n$ gilt.*
5. *Da wir in der Lage sein müssen, Variablen, Funktions- und Prädikats-Zeichen unterscheiden zu können, vereinbaren wir, dass die Mengen $\mathcal{V}$, $\mathcal{F}$ und $\mathcal{P}$ paarweise disjunkt sein müssen:*

   $$\mathcal{V} \cap \mathcal{F} = \{\}, \quad \mathcal{V} \cap \mathcal{P} = \{\}, \quad \text{und} \quad \mathcal{F} \cap \mathcal{P} = \{\}.$$ ◇

Als Bezeichner für Objekte verwenden wir Ausdrücke, die aus Variablen und Funktions-Zeichen aufgebaut sind. Solche Ausdrücke nennen wir Terme.

**Definition 28 (Terme, $\mathcal{T}_\Sigma$)** *Ist $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle$ eine Signatur, so definieren wir die Menge der $\Sigma$-Terme $\mathcal{T}_\Sigma$ induktiv:*

1. *Für jede Variable $x \in \mathcal{V}$ gilt $x \in \mathcal{T}_\Sigma$. Jede Variable ist also auch ein Term.*

2. *Ist $f \in \mathcal{F}$ ein n-stelliges Funktions-Zeichen und sind $t_1, \cdots, t_n \in \mathcal{T}_\Sigma$, so gilt*

   $$f(t_1, \cdots, t_n) \in \mathcal{T}_\Sigma,$$

   *der Ausdruck $f(t_1, \cdots, t_n) \in \mathcal{T}_\Sigma$ ist also ein Term. Falls $c \in \mathcal{F}$ ein 0-stelliges Funktions-Zeichen ist, lassen wir auch die Schreibweise $c$ anstelle von $c()$ zu. In diesem Fall nennen wir $c$ eine Konstante.* ◇

**Beispiel**: Es sei

1. $\mathcal{V} := \{x, y, z\}$ die Menge der Variablen,
2. $\mathcal{F} := \{0, 1, +, -, *\}$ die Menge der Funktions-Zeichen,
3. $\mathcal{P} := \{=, \leq\}$ die Menge der Prädikats-Zeichen,
4. $arity := \{\langle 0, 0\rangle, \langle 1, 0\rangle, \langle +, 2\rangle, \langle -, 2\rangle, \langle *, 2\rangle, \langle =, 2\rangle, \langle \leq, 2\rangle\}$,
   gibt die Stelligkeit der Funktions- und Prädikats-Zeichen an und
5. $\Sigma_{\text{arith}} := \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity\rangle$ sei eine Signatur.

Dann können wir wie folgt $\Sigma_{\text{arith}}$-Terme konstruieren:

1. $x, y, z \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
   denn alle Variablen sind auch $\Sigma_{\text{arith}}$-Terme.
2. $0, 1 \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
   denn 0 und 1 sind 0-stellige Funktions-Zeichen.
3. $+(0, x) \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
   denn es gilt $0 \in \mathcal{T}_{\Sigma_{\text{arith}}}$, $x \in \mathcal{T}_{\Sigma_{\text{arith}}}$ und $+$ ist ein 2-stelliges Funktions-Zeichen.
4. $*(+(0, x), 1) \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
   denn $+(0, x) \in \mathcal{T}_{\Sigma_{\text{arith}}}$, $1 \in \mathcal{T}_{\Sigma_{\text{arith}}}$ und $*$ ist ein 2-stelliges Funktions-Zeichen.

In der Praxis werden wir für bestimmte zweistellige Funktionen eine Infix-Schreibweise verwenden, d.h. wir schreiben zweistellige Funktions-Zeichen zwischen ihren Argumenten. Beispielsweise schreiben wir $x + y$ an Stelle von $+(x, y)$. Die Infix-Schreibweise ist dann als Abkürzung für die oben definierte Darstellung zu verstehen. ◇

Als nächstes definieren wir den Begriff der atomaren Formeln. Darunter verstehen wir solche Formeln, die man nicht in kleinere Formeln zerlegen kann: Atomare Formeln enthalten also weder Junktoren noch Quantoren.

**Definition 29 (Atomare Formeln, $\mathcal{A}_\Sigma$)** *Gegeben sei eine Signatur $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity\rangle$. Die Menge der atomaren $\Sigma$-Formeln $\mathcal{A}_\Sigma$ wird wie folgt definiert: Ist $p \in \mathcal{P}$ ein n-stelliges Prädikats-Zeichen und sind $n$ $\Sigma$-Terme $t_1, \cdots, t_n$ gegeben, so ist $p(t_1, \cdots, t_n)$ eine atomaren $\Sigma$-Formel:*

$$p(t_1, \cdots, t_n) \in \mathcal{A}_\Sigma.$$

*Falls $p$ ein 0-stelliges Prädikats-Zeichen ist, dann schreiben wir auch $p$ anstelle von $p()$. In diesem Fall nennen wir $p$ eine Aussage-Variable.* ◇

**Beispiel**: Setzen wir das letzte Beispiel fort, so können wir sehen, dass

$$=(*(+(0,x),1),0)$$

eine atomare $\Sigma_{\text{arith}}$-Formel ist. Beachten Sie, dass wir bisher noch nichts über den Wahrheitswert von solchen Formeln ausgesagt haben. Die Frage, wann eine Formel als wahr oder falsch gelten soll, wird erst im nächsten Abschnitt untersucht. ◇

Bei der Definition der prädikatenlogischen Formeln ist es notwendig, zwischen sogenannten gebundenen und freien Variablen zu unterscheiden. Wir führen diese Begriffe zunächst informal mit Hilfe eines Beispiels aus der Analysis ein. Wir betrachten die folgende Gleichung:

$$\int_0^x y \cdot t \, dt = \frac{1}{2} \cdot x^2 \cdot y$$

In dieser Gleichung treten die Variablen $x$ und $y$ frei auf, während die Variable $t$ durch das Integral gebunden wird. Damit meinen wir folgendes: Wir können in dieser Gleichung für $x$ und $y$ beliebige Werte einsetzen, ohne dass sich an der Gültigkeit der Formel etwas ändert. Setzen wir zum Beispiel für $x$ den Wert 2 ein, so erhalten wir

$$\int_0^2 y \cdot t \, dt = \frac{1}{2} \cdot 2^2 \cdot y$$

und diese Gleichung ist ebenfalls gültig. Demgegenüber macht es keinen Sinn, wenn wir für die gebundene Variable $t$ eine Zahl einsetzen würden. Die linke Seite der entstehenden Gleichung wäre einfach undefiniert. Wir können für $t$ höchstens eine andere Variable einsetzen. Ersetzen wir die Variable $t$ beispielsweise durch $u$, so erhalten wir

$$\int_0^x y \cdot u \, du = \frac{1}{2} \cdot x^2 \cdot y$$

und das ist inhaltlich dieselbe Aussage wie oben. Das funktioniert allerdings nicht mit jeder Variablen. Setzen wir für $t$ die Variable $y$ ein, so erhalten wir

$$\int_0^x y \cdot y \, dy = \frac{1}{2} \cdot x^2 \cdot y.$$

Diese Aussage ist aber falsch! Das Problem liegt darin, dass bei der Ersetzung von $t$ durch $y$ die vorher freie Variable $y$ gebunden wurde.

Ein ähnliches Problem erhalten wir, wenn wir für $y$ beliebige Terme einsetzen. Solange diese Terme die Variable $t$ nicht enthalten, geht alles gut. Setzen wir beispielsweise für $y$ den Term $x^2$ ein, so erhalten wir

$$\int_0^x x^2 \cdot t \, dt = \frac{1}{2} \cdot x^2 \cdot x^2$$

und diese Formel ist gültig. Setzen wir allerdings für $y$ den Term $t^2$ ein, so erhalten wir

$$\int_0^x t^2 \cdot t \, dt = \frac{1}{2} \cdot x^2 \cdot t^2$$

und diese Formel ist nicht mehr gültig.

In der Prädikatenlogik binden die Quantoren "$\forall$" (für alle) und "$\exists$" (es gibt) Variablen in ähnlicher Weise, wie der Integral-Operator "$\int \cdot \; \mathrm{d}t$" in der Analysis Variablen bindet. Die oben gemachten Ausführungen zeigen, dass es zwei verschiedene Arten von Variable gibt: freie Variablen und gebundene Variablen. Um diese Begriffe präzisieren zu können, definieren wir zunächst für einen $\Sigma$-Term $t$ die Menge der in $t$ enthaltenen Variablen.

**Definition 30 (*Var*$(t)$)** *Ist $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle$ eine Signatur und ist $t$ ein $\Sigma$-Term, so definieren wir die Menge $Var(t)$ der Variablen, die in $t$ auftreten, durch Induktion nach dem Aufbau des Terms:*

1. $Var(x) := \{x\}$ *für alle* $x \in \mathcal{V}$,
2. $Var\big(f(t_1, \cdots, t_n)\big) := Var(t_1) \cup \cdots \cup Var(t_n)$. ◇

**Definition 31 ($\Sigma$-Formel, $\mathbb{F}_\Sigma$, gebundene und freie Variablen, *BV*$(F)$, *FV*$(F)$)**
*Es sei $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle$ eine Signatur. Die Menge der $\Sigma$-Formeln bezeichnen wir mit $\mathbb{F}_\Sigma$. Wir definieren diese Menge induktiv. Gleichzeitig definieren wir für jede Formel $F \in \mathbb{F}_\Sigma$ die Menge $BV(F)$ der in $F$ gebunden auftretenden Variablen und die Menge $FV(F)$ der in $F$ frei auftretenden Variablen.*

1. *Es gilt $\bot \in \mathbb{F}_\Sigma$ und $\top \in \mathbb{F}_\Sigma$ und wir definieren*
   $$FV(\bot) := FV(\top) := BV(\bot) := BV(\top) := \{\}.$$
2. *Ist $F = p(t_1, \cdots, t_n)$ eine atomare $\Sigma$-Formel, so gilt $F \in \mathbb{F}_\Sigma$. Weiter definieren wir:*
   (a) $FV\big(p(t_1, \cdots, t_n)\big) := Var(t_1) \cup \cdots \cup Var(t_n)$.
   (b) $BV\big(p(t_1, \cdots, t_n)\big) := \{\}$.
3. *Ist $F \in \mathbb{F}_\Sigma$, so gilt $\neg F \in \mathbb{F}_\Sigma$. Weiter definieren wir:*
   (a) $FV\big(\neg F\big) := FV(F)$.
   (b) $BV\big(\neg F\big) := BV(F)$.
4. *Sind $F, G \in \mathbb{F}_\Sigma$ und gilt außerdem*
   $$\big(FV(F) \cup FV(G)\big) \cap \big(BV(F) \cup BV(G)\big) = \{\},$$
   *so gilt auch*
   (a) $(F \wedge G) \in \mathbb{F}_\Sigma$,
   (b) $(F \vee G) \in \mathbb{F}_\Sigma$,
   (c) $(F \to G) \in \mathbb{F}_\Sigma$,
   (d) $(F \leftrightarrow G) \in \mathbb{F}_\Sigma$.

   *Weiter definieren wir für alle Junktoren $\odot \in \{\wedge, \vee, \to, \leftrightarrow\}$:*
   (a) $FV\big((F \odot G)\big) := FV(F) \cup FV(G)$.
   (b) $BV\big((F \odot G)\big) := BV(F) \cup BV(G)$.
5. *Sei $x \in \mathcal{V}$ und $F \in \mathbb{F}_\Sigma$ mit $x \notin BV(F)$. Dann gilt:*
   (a) $(\forall x\colon F) \in \mathbb{F}_\Sigma$.
   (b) $(\exists x\colon F) \in \mathbb{F}_\Sigma$.

   *Weiter definieren wir*
   (a) $FV\big((\forall x\colon F)\big) := FV\big((\exists x\colon F)\big) := FV(F) \backslash \{x\}$.
   (b) $BV\big((\forall x\colon F)\big) := BV\big((\exists x\colon F)\big) := BV(F) \cup \{x\}$.

*Ist die Signatur $\Sigma$ aus dem Zusammenhang klar oder aber unwichtig, so schreiben wir auch $\mathbb{F}$ statt $\mathbb{F}_\Sigma$ und sprechen dann einfach von Formeln statt von $\Sigma$-Formeln.* ◇

Bei der oben gegebenen Definition haben wir darauf geachtet, dass eine Variable nicht gleichzeitig frei und gebunden in einer Formel auftreten kann, denn durch eine leichte Induktion nach dem Aufbau der Formeln lässt sich zeigen, dass für alle $F \in \mathbb{F}_\Sigma$ folgendes gilt:

$$FV(F) \cap BV(F) = \{\}.$$

**Beispiel**: Setzen wir das oben begonnene Beispiel fort, so sehen wir, dass

$$(\exists x: \ \leq(+(y,x),y))$$

eine Formel aus $\mathbb{F}_{\Sigma_{\text{arith}}}$ ist. Die Menge der gebundenen Variablen ist $\{x\}$, die Menge der freien Variablen ist $\{y\}$. ◇

Wenn wir Formeln immer in der oben definierten Präfix-Notation anschreiben würden, dann würde die Lesbarkeit unverhältnismäßig leiden. Zur Abkürzung vereinbaren wir, dass in der Prädikatenlogik dieselben Regeln zur Klammer-Ersparnis gelten sollen, die wir schon in der Aussagenlogik verwendet haben. Zusätzlich werden gleiche Quantoren zusammengefasst: Beispielsweise schreiben wir

$$\forall x, y: p(x,y) \quad \text{statt} \quad \forall x: (\forall y: p(x,y)).$$

Außerdem vereinbaren wir, dass wir zweistellige Prädikats- und Funktions-Zeichen auch in Infix-Notation angeben dürfen. Um eine eindeutige Lesbarkeit zu erhalten, müssen wir dann die Präzedenz der Funktions-Zeichen festlegen. Wir schreiben beispielsweise

$$\mathtt{n}_1 = \mathtt{n}_2 \quad \text{anstelle von} \quad =(\mathtt{n}_1, \mathtt{n}_2).$$

Die Formel $(\exists x: \ \leq(+(y,x),y))$ wird dann lesbarer als

$$\exists x: y + x \leq y$$

geschrieben. Außerdem finden Sie in der Literatur häufig Ausdrücke der Form

$$\forall x \in M : F \quad \text{oder} \quad \exists x \in M : F.$$

Hierbei handelt es sich um Abkürzungen, die durch

$$\big(\forall x \in M : F\big) \overset{\text{def}}{\Longleftrightarrow} \forall x : \big(x \in M \rightarrow F\big), \quad \text{und} \quad \big(\exists x \in M : F\big) \overset{\texttt{def}}{\Longleftrightarrow} \exists x : \big(x \in M \wedge F\big).$$

definiert sind.

## 5.2 Semantik der Prädikatenlogik

Als nächstes legen wir die Bedeutung der Formeln fest. Dazu definieren wir den Begriff einer $\Sigma$-Struktur. Eine solche Struktur legt fest, wie die Funktions- und Prädikats-Zeichen der Signatur $\Sigma$ zu interpretieren sind.

**Definition 32 (Struktur)** *Es sei eine Signatur*

$$\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle.$$

*gegeben. Eine $\Sigma$-Struktur $\mathcal{S}$ ist ein Paar $\langle \mathcal{U}, \mathcal{J} \rangle$, so dass folgendes gilt:*

1. *$\mathcal{U}$ ist eine nicht-leere Menge. Diese Menge nennen wir auch das Universum der $\Sigma$-Struktur. Dieses Universum enthält die Werte, die sich später bei der Auswertung der Terme ergeben werden.*

2. *$\mathcal{J}$ ist die Interpretation der Funktions– und Prädikats-Zeichen. Formal definieren wir $\mathcal{J}$ als eine Abbildung mit folgenden Eigenschaften:*

   (a) *Jedem Funktions-Zeichen $f \in \mathcal{F}$ mit $arity(f) = m$ wird eine $m$-stellige Funktion*

   $$f^{\mathcal{J}} : \mathcal{U}^m \to \mathcal{U}$$

   *zugeordnet, die $m$-Tupel des Universums $\mathcal{U}$ in das Universum $\mathcal{U}$ abbildet.*

   (b) *Jedem Prädikats-Zeichen $p \in \mathcal{P}$ mit $arity(p) = n$ wird eine Teilmenge*

   $$p^{\mathcal{J}} \subseteq \mathcal{U}^n$$

   *zugeordnet. Die Idee ist, dass eine atomare Formel der Form $p(t_1, \cdots, t_n)$ genau dann als wahr interpretiert wird, wenn die Interpretation des Tupels $\langle t_1, \cdots, t_n \rangle$ ein Element der Menge $p^{\mathcal{J}}$ ist.*

   (c) *Ist das Zeichen "=" ein Element der Menge der Prädikats-Zeichen $\mathcal{P}$, so gilt*

   $$=^{\mathcal{J}} = \big\{\langle u, u\rangle \mid u \in \mathcal{U}\big\}.$$

   *Eine Formel der Art $s = t$ wird also genau dann als wahr interpretiert, wenn die Interpretation des Terms $s$ den selben Wert ergibt wie die Interpretation des Terms $t$.* ◇

**Beispiel**: Die Signatur $\Sigma_G$ der Gruppen-Theorie sei definiert als

$$\Sigma_G = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity\rangle \quad \text{mit}$$

1. $\mathcal{V} := \{x, y, z\}$
2. $\mathcal{F} := \{\mathrm{e}, *\}$
3. $\mathcal{P} := \{=\}$
4. $arity = \big\{\langle \mathrm{e}, 0\rangle, \langle *, 2\rangle, \langle =, 2\rangle\big\}$

Dann können wir eine $\Sigma_G$ Struktur $\mathcal{Z} = \langle \{0, 1\}, \mathcal{J}\rangle$ definieren, indem wir die Interpretation $\mathcal{J}$ wie folgt festlegen:

1. $\mathrm{e}^{\mathcal{J}} := 0$,
2. $*^{\mathcal{J}} := \Big\{\big\langle\langle 0, 0\rangle, 0\big\rangle, \big\langle\langle 0, 1\rangle, 1\big\rangle, \big\langle\langle 1, 0\rangle, 1\big\rangle, \big\langle\langle 1, 1\rangle, 0\big\rangle\Big\}$,
3. $=^{\mathcal{J}} := \big\{\langle 0, 0\rangle, \langle 1, 1\rangle\big\}$.

   Beachten Sie, dass wir bei der Interpretation des Gleichheits-Zeichens keinen Spielraum haben!

   ◇

Falls wir Terme auswerten wollen, die Variablen enthalten, so müssen wir für diese Variablen irgendwelche Werte aus dem Universum einsetzen. Welche Werte wir einsetzen, kann durch eine Variablen-Belegung festgelegt werden. Diesen Begriff definieren wir nun.

**Definition 33 (Variablen-Belegung)** *Es sei eine Signatur*

$$\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle$$

*gegeben. Weiter sei $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ eine $\Sigma$-Struktur. Dann bezeichnen wir eine Abbildung*

$$\mathcal{I} : \mathcal{V} \to \mathcal{U}$$

*als eine $\mathcal{S}$-Variablen-Belegung.*

*Ist $\mathcal{I}$ eine $\mathcal{S}$-Variablen-Belegung, $x \in \mathcal{V}$ und $c \in \mathcal{U}$, so bezeichnet $\mathcal{I}[x/c]$ die Variablen-Belegung, die der Variablen $x$ den Wert $c$ zuordnet und die ansonsten mit $\mathcal{I}$ übereinstimmt:*

$$\mathcal{I}[x/c](y) := \begin{cases} c & \text{falls } y = x; \\ \mathcal{I}(y) & \text{sonst.} \end{cases}$$ ◇

**Definition 34 (Semantik der Terme)** *Ist $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ eine $\Sigma$-Struktur und $\mathcal{I}$ eine $\mathcal{S}$-Variablen-Belegung, so definieren wir für jeden Term $t$ den Wert $\mathcal{S}(\mathcal{I}, t)$ durch Induktion über den Aufbau von $t$:*

1. *Für Variablen $x \in \mathcal{V}$ definieren wir:*

   $$\mathcal{S}(\mathcal{I}, x) := \mathcal{I}(x).$$

2. *Für $\Sigma$-Terme der Form $f(t_1, \cdots, t_n)$ definieren wir*

   $$\mathcal{S}\big(\mathcal{I}, f(t_1, \cdots, t_n)\big) := f^{\mathcal{J}}\big(\mathcal{S}(\mathcal{I}, t_1), \cdots, \mathcal{S}(\mathcal{I}, t_n)\big).$$ ◇

**Beispiel**: Mit der oben definieren $\Sigma_G$-Struktur $\mathcal{Z}$ definieren wir eine $\mathcal{Z}$-Variablen-Belegung $\mathcal{I}$ durch

$$\mathcal{I} := \big\{\langle x, 0\rangle, \langle y, 1\rangle, \langle z, 0\rangle\big\},$$

es gilt also

$$\mathcal{I}(\mathtt{x}) := 0, \quad \mathcal{I}(\mathtt{y}) := 1, \quad \text{und} \quad \mathcal{I}(\mathtt{z}) := 0.$$

Dann gilt

$$\mathcal{Z}\big(\mathcal{I}, \mathtt{x} * \mathtt{y}\big) = 1.$$ ◇

**Definition 35 (Semantik der atomaren $\Sigma$-Formeln)** *Ist $\mathcal{S}$ eine $\Sigma$-Struktur und $\mathcal{I}$ eine $\mathcal{S}$-Variablen-Belegung, so definieren wir für jede atomare $\Sigma$-Formel $p(t_1, \cdots, t_n)$ den Wert $\mathcal{S}\big(\mathcal{I}, p(t_1, \cdots, t_n)\big)$ wie folgt:*

$$\mathcal{S}\big(\mathcal{I}, p(t_1, \cdots, t_n)\big) := \Big(\big\langle \mathcal{S}(\mathcal{I}, t_1), \cdots, \mathcal{S}(\mathcal{I}, t_n)\big\rangle \in p^{\mathcal{J}}\Big).$$ ◇

**Beispiel**: In Fortführung des obigen Beispiels gilt:

$$\mathcal{Z}(\mathcal{I}, x * y = y * x) = \mathtt{True}.$$ ◇

Um die Semantik beliebiger $\Sigma$-Formeln definieren zu können, nehmen wir an, dass wir, genau wie in der Aussagenlogik, die folgenden Funktionen zur Verfügung haben:

1. $\ominus : \mathbb{B} \to \mathbb{B}$,
2. $\bigvee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$,
3. $\bigwedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$,

4. $\ominus : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$,
5. $\oplus : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$.

Die Semantik dieser Funktionen hatten wir durch die Tabelle in Abbildung 4.1 auf Seite 40 gegeben.

**Definition 36 (Semantik der Σ-Formeln)** *Ist $\mathcal{S}$ eine Σ-Struktur und $\mathcal{I}$ eine $\mathcal{S}$-Variablen-Belegung, so definieren wir für jede Σ-Formel $F$ den Wert $\mathcal{S}(\mathcal{I}, F)$ durch Induktion über den Aufbau von $F$:*

1. $\mathcal{S}(\mathcal{I}, \top) := \texttt{True}$ *und* $\mathcal{S}(\mathcal{I}, \bot) := \texttt{False}$.
2. $\mathcal{S}(\mathcal{I}, \neg F) := \ominus\big(\mathcal{S}(\mathcal{I}, F)\big)$.
3. $\mathcal{S}(\mathcal{I}, F \wedge G) := \otimes\big(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G)\big)$.
4. $\mathcal{S}(\mathcal{I}, F \vee G) := \oplus\big(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G)\big)$.
5. $\mathcal{S}(\mathcal{I}, F \to G) := \ominus\big(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G)\big)$.
6. $\mathcal{S}(\mathcal{I}, F \leftrightarrow G) := \oplus\big(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G)\big)$.
7. $\mathcal{S}\big(\mathcal{I}, \forall x\colon F\big) := \begin{cases} \texttt{True} & \text{falls } \mathcal{S}(\mathcal{I}[x/c], F) = \texttt{True} \quad \text{für alle } c \in \mathcal{U} \text{ gilt;} \\ \texttt{False} & \text{sonst.} \end{cases}$
8. $\mathcal{S}\big(\mathcal{I}, \exists x\colon F\big) := \begin{cases} \texttt{True} & \text{falls } \mathcal{S}(\mathcal{I}[x/c], F) = \texttt{True} \quad \text{für ein } c \in \mathcal{U} \text{ gilt;} \\ \texttt{False} & \text{sonst.} \end{cases}$ ◇

**Beispiel**: In Fortführung des obigen Beispiels gilt

$$\mathcal{Z}\big(\mathcal{I}, \forall \mathtt{x} : \mathrm{e} * x = x\big) = \texttt{True}.$$ ◇

**Definition 37 (Allgemeingültig)** *Ist $F$ eine Σ-Formel, so dass für jede Σ-Struktur $\mathcal{S}$ und für jede $\mathcal{S}$-Variablen-Belegung $\mathcal{I}$*

$$\mathcal{S}(\mathcal{I}, F) = \texttt{True}$$

*gilt, so bezeichnen wir $F$ als allgemeingültig. In diesem Fall schreiben wir*

$$\models F.$$ ◇

Ist $F$ eine Formel für die $FV(F) = \{\}$ ist, dann hängt der Wert $\mathcal{S}(\mathcal{I}, F)$ offenbar gar nicht von der Interpretation $\mathcal{I}$ ab. Solche Formeln bezeichnen wir auch als geschlossene Formeln. In diesem Fall schreiben wir kürzer $\mathcal{S}(F)$ an Stelle von $\mathcal{S}(\mathcal{I}, F)$. Gilt dann zusätzlich $\mathcal{S}(F) = \texttt{True}$, so sagen wir auch, dass $\mathcal{S}$ ein Modell von $F$ ist. Wir schreiben dann

$$\mathcal{S} \models F.$$

Die Definition der Begriffe "erfüllbar" und "äquivalent" lassen sich nun aus der Aussagenlogik übertragen. Um unnötigen Ballast in den Definitionen zu vermeiden, nehmen wir im Folgenden immer eine feste Signatur Σ als gegeben an. Dadurch können wir in den folgenden Definitionen von Termen, Formeln, Strukturen, etc. sprechen und meinen damit Σ-Terme, Σ-Formeln und Σ-Strukturen.

**Definition 38 (Äquivalent)** *Zwei Formeln $F$ und $G$, in denen die Variablen $x_1, \cdots, x_n$ frei auftreten, heißen äquivalent g.d.w.*

$$\models \forall x_1 : \cdots \forall x_n : (F \leftrightarrow G)$$

*gilt. Falls in $F$ und $G$ keine Variablen frei auftreten, dann ist $F$ genau dann äquivalent zu $G$, wenn*

$$\models F \leftrightarrow G$$

*gilt.* ◇

**Bemerkung**: Alle aussagenlogischen Äquivalenzen sind auch prädikatenlogische Äquivalenzen. ◇

**Definition 39 (Erfüllbar)** *Eine Menge $M \subseteq \mathbb{F}_\Sigma$ ist genau dann erfüllbar, wenn es eine Struktur $\mathcal{S}$ und eine Variablen-Belegung $\mathcal{I}$ gibt, so dass*

$$\mathcal{S}(\mathcal{I}, F) = \texttt{True} \quad \text{für alle } F \in M$$

*gilt. Andernfalls heißt $M$ unerfüllbar oder auch widersprüchlich. Wir schreiben dafür auch*

$$M \models \bot$$ ◇

Unser Ziel ist es, ein Verfahren anzugeben, mit dem wir in der Lage sind zu überprüfen, ob eine Menge $M$ von Formeln widersprüchlich ist, ob also $M \models \bot$ gilt. Es zeigt sich, dass dies im Allgemeinen nicht möglich ist, die Frage, ob $M \models \bot$ gilt, ist unentscheidbar. Ein Beweis dieser Tatsache geht allerdings über den Rahmen dieser Vorlesung heraus. Dem gegenüber ist es möglich, ähnlich wie in der Aussagenlogik einen Kalkül $\vdash$ anzugeben, so dass gilt:

$$M \vdash \bot \quad \text{g.d.w.} \quad M \models \bot.$$

Ein solcher Kalkül kann dann zur Implementierung eines Semi-Entscheidungs-Verfahrens benutzt werden: Um zu überprüfen, ob $M \models \bot$ gilt, versuchen wir, aus der Menge $M$ die Formel $\bot$ herzuleiten. Falls wir dabei systematisch vorgehen, indem wir alle möglichen Beweise durchprobieren, so werden wir, falls tatsächlich $M \models \bot$ gilt, auch irgendwann einen Beweis finden, der $M \vdash \bot$ zeigt. Wenn allerdings der Fall

$$M \not\models \bot$$

vorliegt, so werden wir dies im Allgemeinen nicht feststellen können, denn die Menge aller Beweise ist unendlich und wir können nie alle Beweise ausprobieren. Wir können lediglich sicherstellen, dass wir jeden Beweis irgendwann versuchen. Wenn es aber keinen Beweis gibt, so können wir das nie sicher sagen, denn zu jedem festen Zeitpunkt haben wir ja immer nur einen Teil der in Frage kommenden Beweise ausprobiert.

Die Situation ist ähnlich der, wie bei der Überprüfung bestimmter zahlentheoretischer Fragen. Wir betrachten dazu ein konkretes Beispiel: Eine Zahl $n$ heißt eine perfekte Zahl, wenn die Summe aller echten Teiler von $n$ wieder die Zahl $n$ ergibt. Beispielsweise ist die Zahl 6 perfekt, denn die Menge der echten Teiler von 6 ist $\{1, 2, 3\}$ und es gilt

$$1 + 2 + 3 = 6.$$

Bisher sind alle bekannten perfekten Zahlen durch 2 teilbar. Die Frage, ob es auch ungerade Zahlen gibt, die perfekt sind, ist ein offenes mathematisches Problem. Um dieses Problem zu lösen, könnten wir eine Programm schreiben, dass der Reihe nach für alle ungerade Zahlen überprüft, ob die Zahl perfekt ist. Abbildung 5.1 auf Seite 102 zeigt ein solches Programm. Wenn es eine ungerade perfekte Zahl gibt, dann wird dieses Programm diese Zahl auch irgendwann finden. Wenn es aber keine

ungerade perfekte Zahl gibt, dann wird das Programm bis zum St. Nimmerleinstag rechnen und wir werden nie mit Sicherheit wissen, dass es keine ungeraden perfekten Zahlen gibt.

```
1  def perfect(n):
2      return sum({ x for x in range(1, n) if n % x == 0 }) == n
3
4  def findOddPerfect():
5      n = 1
6      while True:
7          if perfect(n):
8              return n
9          n += 2
10
11 findOddPerfect()
```

Figure 5.1: Suche nach einer ungeraden perfekten Zahl.

## 5.3 Implementierung prädikatenlogischer Strukturen in *Python*

Der im letzten Abschnitt präsentierte Begriff einer prädikatenlogischen Struktur erscheint zunächst sehr abstrakt. Wir wollen in diesem Abschnitt zeigen, dass sich dieser Begriff in einfacher Weise in *Python* implementieren lässt. Dadurch gelingt es, diesen Begriff zu veranschaulichen. Als konkretes Beispiel wollen wir Strukturen zu Gruppen-Theorie betrachten. Wir gehen dazu in vier Schritten vor:

1. Zunächst definieren wir mathematisch, was wir unter einer Gruppe verstehen.
2. Anschließend diskutieren wir, wie wir die Formeln der Gruppen-Theorie in *Python* darstellen.
3. Dann definieren wir eine Struktur, in der die Formeln der Gruppen-Theorie gelten.
4. Schließlich zeigen wir, wie wir prädikaten-logische Formeln in *Python* auswerten können und führen dies am Beispiel der für die Gruppen-Theorie definierten Struktur vor.

### 5.3.1 Gruppen-Theorie

In der Mathematik wird eine Gruppe $\mathcal{G}$ als ein Tripel der Form

$$\mathcal{G} = \langle G, \mathrm{e}, * \rangle$$

definiert. Dabei gilt:

1. $G$ ist eine Menge,
2. e ist ein Element der Menge $G$ und
3. $* : G \times G \to G$ ist eine binäre Funktion auf $G$, die wir im Folgenden als die Multiplikation der Gruppe bezeichnen.

4. Außerdem müssen die folgenden drei Axiome gelten:

   (a) $\forall x : \mathrm{e} * x = x$,
   e ist bezüglich der Multiplikation ein links-neutrales Element.

   (b) $\forall x : \exists y : y * x = \mathrm{e}$,
   d.h. für jedes $x \in G$ gibt es ein links-inverses Element.

   (c) $\forall x : \forall y : \forall z : (x * y) * z = x * (y * z)$,
   d.h. es gilt das Assoziativ-Gesetz.

   (d) Die Gruppe $\mathcal{G}$ ist eine kommutative Gruppe genau dann, wenn zusätzlich das folgende Axiom gilt:
   $\forall x : \forall y : x * y = y * x$.
   d.h. es gilt das Kommutativ-Gesetz. ◇

Beachten Sie, dass das Kommutativ-Gesetz in einer Gruppe im Allgemeinen nicht gelten muss.

### 5.3.2 Darstellung der Formeln in *Python*

Im letzten Abschnitt haben wir die Signatur $\Sigma_G$ der Gruppen-Theorie wie folgt definiert:

$$\Sigma_G = \big\langle \{x, y, z\},\ \{\mathrm{e}, *\},\ \{=\},\ \{\langle \mathrm{e}, 0\rangle, \langle *, 2\rangle, \langle =, 2\rangle\}\big\rangle.$$

Hierbei ist also "e" ein 0-stelliges Funktions-Zeichen, "$*$" ist eine 2-stelliges Funktions-Zeichen und "$=$" ist ein 2-stelliges Prädikats-Zeichen. Wir werden für prädikaten-logische Formeln einen Parser verwenden, der keine binären Infix-Operatoren wie "$*$" oder "$=$" unterstützt. Bei diesem Parser können Terme nur in der Form

$$f(t_1, \cdots, t_n)$$

angegeben werden, wobei $f$ eine Funktions-Zeichen ist und $t_1$, $\cdots$, $t_n$ Terme sind. Analog werden atomare Formeln durch Ausdrücke der Form

$$p(t_1, \cdots, t_n)$$

dargestellt, wobei $p$ eine Prädikats-Zeichen ist. Variablen werden von den Funktions- und Prädikats-Zeichen dadurch unterschieden, dass Variablen mit einem kleinen Buchstaben beginnen, während Funktions- und Prädikats-Zeichen mit einem großen Buchstaben beginnen. Um die Formeln der Gruppentheorie darstellen zu können, vereinbaren wir daher das Folgende:

1. Das neutrale Element e schreiben wir als `E()`.
2. Für den Operator $*$ verwenden wir das zweistellige Funktions-Zeichen `Multiply`. Damit wird der Ausdruck $x * y$ also als `Multiply`$(x, y)$ geschrieben.
3. Das Gleichheits-Zeichen "$=$" repräsentieren wir durch das zweistellige Prädikats-Zeichen `Equals`. Damit schreibt sich dann beispielsweise die Formel $x = y$ als `Equals`$(x, y)$.

Abbildung 5.2 zeigt die Formeln der Gruppen-Theorie als Strings.

Wir können die Formeln mit der in Abbildung 5.3 gezeigten Funktion `parse`$(s)$ in geschachtelte Tupel überführen. Das Ergebnis dieser Transformation ist in Abbildung 5.4 zu sehen.

```
G1 = '∀x:Equals(Multiply(E(),x),x)'
G2 = '∀x:∃y:Equals(Multiply(x,y),E())'
G3 = '∀x:∀y:∀z:Equals(Multiply(Multiply(x,y),z), Multiply(x,Multiply(y,z)))'
G4 = '∀x:∀y:Equals(Multiply(x,y), Multiply(y,x))'
```

Figure 5.2: Die Formeln der kommutativen Gruppentheorie als Strings

```
import folParser as fp

def parse(s):
    "Parse string s as fol formula."
    p = fp.LogicParser(s)
    return p.parse()

F1 = parse(G1)
F2 = parse(G2)
F3 = parse(G3)
F4 = parse(G4)
```

Figure 5.3: Die Funktion `parse`

```
F1 = ('∀', 'x', ('Equals', ('Multiply', ('E',), 'x'), 'x'))
F2 = ('∀', 'x', ('∃', 'y', ('Equals', ('Multiply', 'x', 'y'), ('E',))))
F3 = ('∀', 'x', ('∀', 'y', ('∀', 'z',
          ('Equals', ('Multiply', ('Multiply', 'x', 'y'), 'z'),
                     ('Multiply', 'x', ('Multiply', 'y', 'z'))
          )
      )))
F4 = ('∀', 'x', ('∀', 'y',
          ('Equals', ('Multiply', 'x', 'y'),
                     ('Multiply', 'y', 'x')
          )
      ))
```

Figure 5.4: Die Axiome einer kommutativen Gruppe als geschachtelte Tupel

### 5.3.3 Darstellung prädikaten-logischer Strukturen in *Python*

Wir hatten bei der Definition der Semantik der Prädikaten-Logik in Abschnitt 5.2 bereits eine Struktur $\mathcal{S}$ angegeben, deren Universum aus der Menge $\{0,1\}$ besteht. In *Python* können wir diese Struktur durch den in Abbildung 5.5 auf Seite 105 gezeigten Code implementieren.

```
1   U = { 0, 1 }
2   NeutralElement = { (): 0 }
3   Product        = { (0, 0): 0,  (0, 1): 1,  (1, 0): 1,  (1, 1): 0 }
4   Identity       = { (0, 0), (1, 1) }
5   J = { "E": NeutralElement, "Multiply": Product, "Equals": Identity }
6   S = (U, J)
7   I = { "x": 0, "y": 1, "z": 0 }
```

Figure 5.5: Implementierung einer Struktur zur Gruppen-Theorie

1. Das in Zeile 1 definierte Universum $U$ besteht aus den beiden Zahlen 0 und 1.

2. In Zeile 2 definieren wir die Interpretation des nullstelligen Funktions-Zeichens `E` als das *Python*-Dictionary, das dem leeren Tupel die Zahl 0 zuordnet.

3. In Zeile 3 definieren wir eine Funktion `Product` als *Python*-Dictionary. Für die so definierte Funktion gilt

   $$\texttt{Product}(0,0) = 0, \quad \texttt{Product}(0,1) = 1,$$
   $$\texttt{Product}(1,0) = 1, \quad \texttt{Product}(1,1) = 0.$$

   Diese Funktion verwenden wir später als die Interpretation $\texttt{Multiply}^{\mathcal{J}}$ des Funktions-Zeichens "`Multiply`".

4. In Zeile 4 haben wir die Interpretation $\texttt{Equals}^{\mathcal{J}}$ des Prädikats-Zeichens "`Equals`" als die Menge $\{\langle 0,0\rangle, \langle 1,1\rangle\}$ definiert.

5. In Zeile 7 fassen wir die Interpretationen der Funktions-Zeichen "`E`" und "`Multiply`" und des Prädikats-Zeichens "`Equals`" zu dem Dictionary `J` zusammen, so dass für ein Funktions- oder Prädikats-Zeichen $f$ die Interpretation $f^{\mathcal{J}}$ durch den Wert $\texttt{J}[f]$ gegeben ist.

6. Die Interpretation `J` wird dann in Zeile 6 mit dem Universum `U` zu der Struktur `S` zusammengefasst, die in *Python* einfach als Paar dargestellt wird.

7. Schließlich zeigt Zeile 7, dass eine Variablen-Belegung ebenfalls als Dictionary dargestellt werden kann. Die Schlüssel sind die Variablen, die Werte sind dann die Objekte aus dem Universum, auf welche die Variablen abgebildet werden.

Als nächstes überlegen wir uns, wie wir prädikatenlogische Terme in einer solchen Struktur auswerten können. Abbildung 5.6 zeigt die Implementierung der Prozedur $\texttt{evalTerm}(t, \mathcal{S}, \mathcal{I})$, der als Argumente ein prädikatenlogischer Term $t$, eine prädikatenlogische Struktur $\mathcal{S}$ und eine Variablen-Belegung $\mathcal{I}$ übergeben werden. Der Term $t$ wird dabei in *Python* als geschachteltes Tupel dargestellt.

```
1  def evalTerm(t, S, I):
2      if isinstance(t, str):  # t is a variable
3          return I[t]
4      _, J     = S      # J is the dictionary of interpretations
5      f, *args = t      # function symbol and arguments
6      fJ       = J[f]   # interpretation of function symbol
7      argVals  = evalTermTuple(args, S, I)
8      return fJ[argVals]
9
10 def evalTermTuple(Ts, S, I):
11     return tuple(evalTerm(t, S, I) for t in Ts)
```

Figure 5.6: Auswertung von Termen

1. In Zeile 2 überprüfen wir, ob der Term $t$ eine Variable ist. Dies ist daran zu erkennen, dass Variablen als Strings dargestellt werden, während alle anderen Terme Tupel sind. Falls $t$ eine Variable ist, dann geben wir den Wert zurück, der in der Variablen-Belegung $\mathcal{I}$ für diese Variable gespeichert ist.

2. Sonst extrahieren wir in Zeile 4 das Dictionary $\mathcal{J}$, das die Interpretationen der Funktions- und Prädikats-Zeichen enthält, aus der Struktur $\mathcal{S}$.

3. Das Funktions-Zeichen $f$ des Terms $t$ ist die erste Komponente des Tupels $t$, die Argumente werden in der Liste `args` zusammen gefasst.

4. Die Interpretation $f^{\mathcal{J}}$ dieses Funktions-Zeichens schlagen wir in Zeile 6 in dem Dictionary $\mathcal{J}$ nach.

5. Das Tupel, das aus diesen Argumenten besteht, wird in Zeile 7 rekursiv ausgewertet. Als Ergebnis erhalten wir dabei ein Tupel von Werten.

6. Dieses Tupel dient dann in Zeile 8 als Argument für das Dictionary $f^{\mathcal{J}}$. Der in diesem Dictionary für die Argumente abgelegte Wert ist das Ergebnis der Auswertung des Terms $t$.

```
1  def evalAtomic(a, S, I):
2      _, J     = S      # J is the dictionary of interpretations
3      p, *args = a      # predicate symbol and arguments
4      pJ       = J[p]   # interpretation of predicate symbol
5      argVals  = evalTermTuple(args, S, I)
6      return argVals in pJ
```

Figure 5.7: Auswertung atomarer Formeln

Abbildung 5.7 zeigt die Auswertung einer atomaren Formel. Eine atomare Formel $a$ ist in Python als Tupel der Form

$$a = (p, t_1, \cdots, t_n).$$

dargestellt. Wir können diese Tupel durch die Zuweisung

```
p, *args = a
```

in seine Komponenten zerlegen. `args` ist dann die Liste $[t_1, \cdots, t_n]$. Um zu überprüfen, ob die atomare Formel $a$ wahr ist, müssen wir überprüfen, ob

$$\big(\texttt{evalTerm}(t_1, \mathcal{S}, \mathcal{I}), \cdots, \texttt{evalTerm}(t_n, \mathcal{S}, \mathcal{I})\big) \in p^{\mathcal{J}}$$

gilt. Dieser Test wird in Zeile 6 durchgeführt. Der Rest der Implementierung der Funktion `evalAtomic` ist analog zur Implementierung der Funktion `evalTerm`.

```
1  def evalFormula(F, S, I):
2      U = S[0]  # universe
3      if F[0] == '⊤': return True
4      if F[0] == '⊥': return False
5      if F[0] == '¬': return not evalFormula(F[1], S, I)
6      if F[0] == '∧': return evalFormula(F[1], S, I) and evalFormula(F[2], S, I)
7      if F[0] == '∨': return evalFormula(F[1], S, I) or evalFormula(F[2], S, I)
8      if F[0] == '→': return not evalFormula(F[1], S, I) or evalFormula(F[2], S, I)
9      if F[0] == '↔': return evalFormula(F[1], S, I) == evalFormula(F[2], S, I)
10     if F[0] == '∀':
11         x, G = F[1:]
12         return all({ evalFormula(G, S, modify(I, x, c)) for c in U} )
13     if F[0] == '∃':
14         x, G = F[1:]
15         return any({ evalFormula(G, S, modify(I, x, c)) for c in U} )
16     return evalAtomic(F, S, I)
```

Figure 5.8: Die Funktion `evalFormula`.

Abbildung 5.8 auf Seite 107 zeigt die Implementierung der Funktion $\texttt{evalFormula}(F, \mathcal{S}, \mathcal{I})$, die als Argumente eine prädikatenlogische Formel $F$, eine prädikatenlogische Struktur $\mathcal{S}$ und eine Variablen-Belegung $\mathcal{I}$ erhält und die als Ergebnis den Wert $\mathcal{S}(\mathcal{I}, F)$ berechnet. Die Auswertung der Formel $F$ erfolgt dabei analog zu der in Abbildung 4.1 auf Seite 44 gezeigten Auswertung aussagenlogischer Formeln. Neu ist hier nur die Behandlung der Quantoren. In den Zeilen 10, 11 und 12 behandeln wir die Auswertung allquantifizierter Formeln. Ist $F$ eine Formel der Form $\forall x : G$, so wird die Formel $F$ durch das Tupel

$$F = (\text{'}\forall\text{'}, x, G)$$

dargestellt. Die Auswertung von $\forall x\colon G$ geschieht nach der Formel

$$\mathcal{S}\big(\mathcal{I}, \forall x\colon G\big) := \begin{cases} \texttt{True} & \text{falls } \mathcal{S}(\mathcal{I}[x/c], G) = \texttt{True} \quad \text{für alle } c \in \mathcal{U} \text{ gilt;} \\ \texttt{False} & \text{sonst.} \end{cases}$$

Um die Auswertung implementieren zu können, verwenden wir die Prozedur `modify()`, welche die Variablen-Belegung $\mathcal{I}$ an der Stelle $x$ zu $c$ abändert, es gilt also

$$\texttt{modify}(\mathcal{I}, x, c) = \mathcal{I}[x/c].$$

Die Implementierung dieser Prozedur ist in Abbildung 5.9 auf Seite 108 gezeigt. Bei der Auswertung eines All-Quantors können wir ausnutzen, dass die Sprache *Python* den Quantor "∀" durch die Funktion `all` unterstützt. Wir können also direkt testen, ob die Formel für alle möglichen Werte $c$, die wir für die Variable $x$ einsetzen können, richtig ist. Für eine Menge $S$ von Wahrheitswerten ist der Ausdruck

$$\texttt{all}(S)$$

genau dann wahr, wenn alle Elemente von $S$ den Wert `True` haben. Die Auswertung eines Existenz-Quantors ist analog zur Auswertung eines All-Quantors. Der einzige Unterschied besteht darin, dass wir statt der Funktion `all` die Funktion `any` verwenden. Der Ausdruck

$$\texttt{any}(S)$$

ist für eine Menge von Wahrheitswerten $S$ genau dann wahr, wenn es wenigstens ein Element in der Menge $S$ gibt, dass den Wert `True` hat.

Bei der Implementierung der Prozedur `modify`$(I, x, c)$, die als Ergebnis die Variablen-Belegung $\mathcal{I}[x/c]$ berechnet, nutzen wir aus, dass wir bei einer Funktion, die als Dictionary gespeichert ist, den Wert, der für ein Argument $x$ eingetragen ist, durch eine Zuweisung der Form

$$\mathcal{I}[x] = c$$

abändern können.

```
1  def modify(I, x, c):
2      I[x] = c
3      return I
```

Figure 5.9: Die Implementierung der Funktion `modify`.

Mit dem in Abbildung 5.10 gezeigten Skript können wir nun überprüfen, ob die in Abbildung 5.10 auf Seite 108 definierte Struktur eine Gruppe ist. Wir erhalten die in Abbildung 5.11 gezeigte Ausgabe und können daher folgern, dass diese Struktur in der Tat eine kommutative Gruppe ist.

```
1  f"evalFormula({G1}, S, I) = {evalFormula(F1, S, I)}"
2  f"evalFormula({G2}, S, I) = {evalFormula(F2, S, I)}"
3  f"evalFormula({G3}, S, I) = {evalFormula(F3, S, I)}"
4  f"evalFormula({G4}, S, I) = {evalFormula(F4, S, I)}"
```

Figure 5.10: Überprüfung, ob die in Abbildung 5.5 definierte Struktur eine Gruppe ist

**Bemerkung**: Das oben vorgestellte Programm finden sie als Jupyter Notebook auf GitHub unter der Adresse:

```
evalFormula(∀x:Equals(Multiply(E(),x),x), S, I) = True
evalFormula(∀x:∃y:Equals(Multiply(x,y),E()), S, I) = True
evalFormula(∀x:∀y:∀z:Equals(Multiply(Multiply(x,y),z), Multiply(x,Multiply(y,z))), S, I)
= True
evalFormula(∀x:∀y:Equals(Multiply(x,y), Multiply(y,x)), S, I) = True
```

Figure 5.11: Ausgabe des in Abbildung 5.10 gezeigten Skripts

https://github.com/karlstroetmann/Logic/blob/master/FOL-Evaluation.ipynb

Mit diesem Programm können wir überprüfen, ob eine prädikatenlogische Formel in einer vorgegebenen endlichen Struktur erfüllt ist. Wir können damit allerdings nicht überprüfen, ob eine Formel allgemeingültig ist, denn einerseits können wir das Programm nicht anwenden, wenn die Strukturen ein unendliches Universum haben, andererseits ist selbst die Zahl der verschiedenen endlichen Stukturen, die wir ausprobieren müssten, unendlich groß. ◇

**Aufgabe 11**:

1. Zeigen Sie, dass die Formel

   $$\forall x : \exists y : p(x,y) \rightarrow \exists y : \forall x : p(x,y)$$

   nicht allgemeingültig ist, indem Sie in *Python* eine geeignete prädikatenlogische Struktur $\mathcal{S}$ implementieren, in der diese Formel falsch ist.

2. Überlegen Sie, wie viele verschiedene Strukturen es für die Signatur der Gruppen-Theorie gibt, wenn wir davon ausgehen, dass das Universum die Form $\{1, \cdots, n\}$ hat.

3. Geben Sie eine erfüllbare prädikatenlogische Formel $F$ an, die in einer prädikatenlogischen Struktur $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ immer falsch ist, wenn das Universum $\mathcal{U}$ endlich ist.

   **Hinweis**: Es sei $f : U \rightarrow U$ eine Funktion. Überlegen Sie, wie die Aussagen "*$f$ ist injektiv*" und "*$f$ ist surjektiv*" zusammen hängen, wenn das Universum endlich ist. ◇

## 5.4 Constraint Programing

It is time to see a practical application of first order logic. One of these practical applications is constraint programming. Constraint programming is an example of the declarative programming paradigm. In declarative programming, the idea is that in order to solve a given problem, this problem is specified and this specification is given as input to a problem solver which will then compute a solution to the problem. Hence, the task of the programmer is much easier than it normally is: Instead of implementing a program that solves a given problem, the programmer only has to specify the problem precisely, she does not have to explicitely code an algorithm to find the solution. Usually, the specification of a problem is much easier than the coding of an algorithm to solve the problem. This approach works well for those problems that can be specified using first order logic. The remainder of this section is structured as follows:

1. We first define constraint satisfaction problems.

   As an example, we show how the eight queens puzzle can be formulated as a constraint satisfaction problem.

2. We discuss a simple constraint solver that is based on backtracking.
3. Then we show how some puzzles can be solved using constraint programming.

### 5.4.1 Constraint Satisfaction Problems

Conceptually, a constraint satisfaction problem is given by a set of first order logic formulas that contain a number of free variables. Furthermore, a structure $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ consisting of a universe $\mathcal{U}$ and the interpretation $\mathcal{J}$ of the function and predicate symbols used in these formulas is assumed to be understood from the context of the problem. The goal is to find a variable assignment such that the given formulas are evaluated as true.

**Definition 40 (CSP)**
*Formally, a constraint satisfaction problem (abbreviated as* CSP*) is defined as a triple*

$$\mathcal{P} := \langle \texttt{Vars}, \texttt{Values}, \texttt{Constraints} \rangle$$

*where*

1. `Vars` *is a set of strings which serve as variables,*
2. `Values` *is a set of values that can be assigned to the variables in* `Vars`*.*

   *This set of values is assumed to be identical to the universe of the first order structure* $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ *that is given implicitly, i.e. we have*

   $$\texttt{Values} = \mathcal{U}.$$

3. `Constraints` *is a set of formulas from first order logic. Each of these formulas is called a constraint of* $\mathcal{P}$*.* ◇

Given a CSP

$$\mathcal{P} = \langle \texttt{Vars}, \texttt{Values}, \texttt{Constraints} \rangle,$$

a variable assignment for $\mathcal{P}$ is a function

$$\mathcal{I} : \texttt{Vars} \to \texttt{Values}.$$

A variable assignment $\mathcal{I}$ is a solution of the CSP $\mathcal{P}$ if, given the assignment $\mathcal{I}$, all constraints of $\mathcal{P}$ are satisfied, i.e. we have

$$\mathcal{S}(\mathcal{I}, f) = \texttt{True} \quad \text{for all } f \in \texttt{Constraints}.$$

Finally, a partial variable assignment $\mathcal{B}$ for $\mathcal{P}$ is a function

$$\mathcal{B} : \texttt{Vars} \to \texttt{Values} \cup \{\Omega\} \quad \text{where } \Omega \text{ denotes the undefined value.}$$

Hence, a partial variable assignment does not assign values to all variables. Instead, it assigns values only to a subset of the set `Vars`. The domain $\texttt{dom}(\mathcal{B})$ of a partial variable assignment $\mathcal{B}$ is the set of those variables that are assigned a value different from $\Omega$, i.e. we define

$$\texttt{dom}(\mathcal{B}) := \{x \in \texttt{Vars} \mid \mathcal{B}(x) \neq \Omega\}.$$

We proceed to illustrate the definitions given so far by presenting two examples.

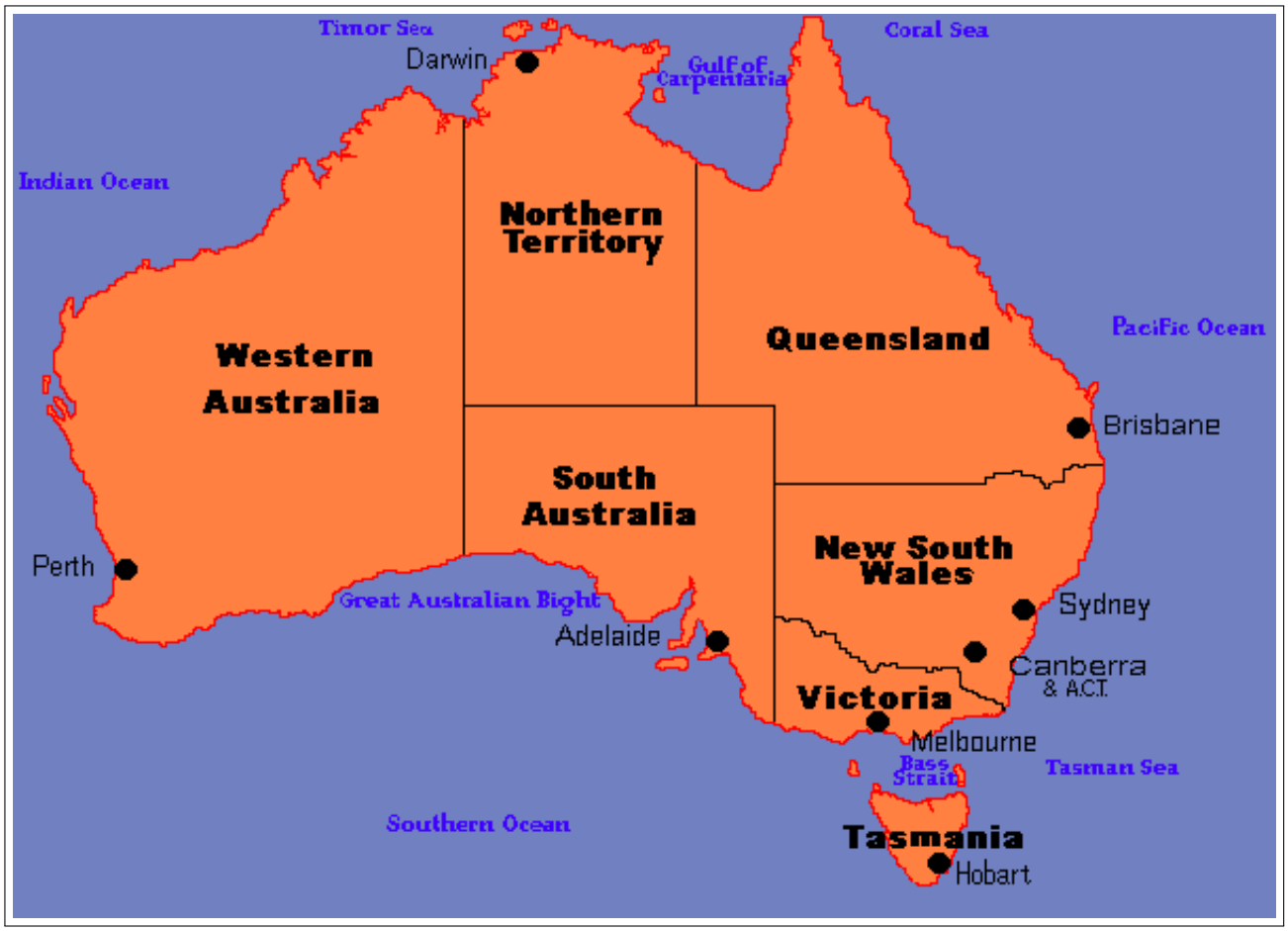


Figure 5.12: A map of Australia.

### 5.4.2 Example: Map Colouring

In map colouring a map showing different state borders is given and the task is to colour the different states such that no two states that have a common border share the same colour. Figure 5.12 on page 111 shows a map of Australia. There are seven different states in Australia:

1. Western Australia, abbreviated as WA,
2. Northern Territory, abbreviated as NT,
3. South Australia, abbreviated as SA,
4. Queensland, abbreviated as Q,
5. New South Wales, abbreviated as NSW,
6. Victoria, abbreviated as V, and
7. Tasmania, abbreviated as T.

Figure 5.12 would certainly look better if different states had been coloured with different colours. For the purpose of this example let us assume that we have only three colours available. The question then is whether it is possible to colour the different states in a way that no two neighbouring states share the same colour. This problem can be formalized as a constraint satisfaction problem. To this end we define:

1. $\texttt{Vars} := \{\texttt{WA}, \texttt{NT}, \texttt{SA}, \texttt{Q}, \texttt{NSW}, \texttt{V}, \texttt{T}\}$,
2. $\texttt{Values} := \{\texttt{red}, \texttt{green}, \texttt{blue}\}$,
3. $\texttt{Constraints} :=$
   $\{\texttt{WA} \neq \texttt{NT}, \texttt{WA} \neq \texttt{SA}, \texttt{NT} \neq \texttt{SA}, \texttt{NT} \neq \texttt{Q}, \texttt{SA} \neq \texttt{Q}, \texttt{SA} \neq \texttt{NSW}, \texttt{SA} \neq \texttt{V}, \texttt{Q} \neq \texttt{NSW}, \texttt{NSW} \neq \texttt{V}, \texttt{V} \neq \texttt{T}\}$

Then $\mathcal{P} := \langle \texttt{Vars}, \texttt{Values}, \texttt{Constraints} \rangle$ is a constraint satisfaction problem. If we define the assignment $\mathcal{I}$ such that

1. $\mathcal{I}(\texttt{WA}) = \texttt{blue}$,
2. $\mathcal{I}(\texttt{NT}) = \texttt{red}$,
3. $\mathcal{I}(\texttt{SA}) = \texttt{green}$,
4. $\mathcal{I}(\texttt{Q}) = \texttt{blue}$,
5. $\mathcal{I}(\texttt{NSW}) = \texttt{red}$,
6. $\mathcal{I}(\texttt{V}) = \texttt{blue}$,
7. $\mathcal{I}(\texttt{T}) = \texttt{red}$,

then you can check that the assignment $\mathcal{I}$ is indeed a solution to the constraint satisfaction problem $\mathcal{P}$.

### 5.4.3 Example: The Eight Queens Puzzle

The eight queens problem asks to put 8 queens onto a chessboard such that no queen can attack another queen. In chess, a queen can attack all pieces that are either in the same row, the same column, or the same diagonal. If we want to put 8 queens on a chessboard such that no two queens can attack each other, we have to put exactly one queen in every row: If we would put more than one queen in a row, the queens in that row can attack each other. If we would leave a row empty, then, given that the other rows contain at most one queen, there would be less than 8 queens on the board. Therefore, in order to model the eight queens problem as a constraint satisfaction problem, we will use the following set of variables:

$$\texttt{Vars} := \{\texttt{Q}_1, \texttt{Q}_2, \texttt{Q}_3, \texttt{Q}_4, \texttt{Q}_5, \texttt{Q}_6, \texttt{Q}_7, \texttt{Q}_8\},$$

where for $i \in \{1, \cdots, 8\}$ the variable $\texttt{Q}_i$ specifies the column of the queen that is placed in row $i$. As the columns run from one to eight, we define the set $\texttt{Values}$ as

$$\texttt{Values} := \{1, 2, 3, 4, 5, 6, 7, 8\}.$$

Next, let us define the constraints. There are two different types of constraints.

1. We have constraints that express that no two queens positioned in different rows share the same column. To capture these constraints, we define

   $$\texttt{SameRow} := \big\{ \texttt{Q}_i \neq \texttt{Q}_j \;\big|\; i \in \{1, \cdots, 8\} \wedge j \in \{1, \cdots, 8\} \wedge j < i \big\}.$$

   Here the condition $i < j$ ensures that, for example, we have the constraint $\texttt{Q}_2 \neq \texttt{Q}_1$ but not the constraint $\texttt{Q}_1 \neq \texttt{Q}_2$, as the latter constraint would be redundant if the former constraint has already been established.

2. We have constraints that express that no two queens positioned in different rows share the same diagonal. The queens in row $i$ and row $j$ share the same diagonal iff the equation

   $$|i - j| = |\texttt{Q}_i - \texttt{Q}_j|$$

   holds. The expression $|i - j|$ is the absolute value of the difference of the rows of the queens in row $i$ and row $j$, while the expression $|\texttt{Q}_i - \texttt{Q}_j|$ is the absolute value of the difference of the columns of these queens. To capture these constraints, we define

   $$\texttt{SameDiagonal} := \big\{ |i - j| \neq |\texttt{Q}_i - \texttt{Q}_j| \;\big|\; i \in \{1, \cdots, 8\} \wedge j \in \{1, \cdots, 8\} \wedge j < i \big\}.$$

Then, the set of constraints is defined as

$$\texttt{Constraints} := \texttt{SameRow} \cup \texttt{SameDiagonal}$$

and the eight queens problem can be stated as the constraint satisfaction problem

$$\mathcal{P} := \langle \texttt{Vars}, \texttt{Values}, \texttt{Constraints} \rangle.$$

If we define the assignment $\mathcal{I}$ such that

$$\mathcal{I}(\texttt{Q}_1) := 4,\ \mathcal{I}(\texttt{Q}_2) := 8,\ \mathcal{I}(\texttt{Q}_3) := 1,\ \mathcal{I}(\texttt{Q}_4) := 2,\ \mathcal{I}(\texttt{Q}_5) := 6,\ \mathcal{I}(\texttt{Q}_6) := 2,\ \mathcal{I}(\texttt{Q}_7) := 7,\ \mathcal{I}(\texttt{Q}_8) := 5,$$

then it is easy to see that this assignment is a solution of the eight queens problem. This solution is shown in Figure 5.13 on page 114.

Later, when we implement procedures to solve CSPs, we will represent variable assignments and partial variable assignments as dictionaries. For example, the variable assignment $\mathcal{I}$ defined above would then be represented as the dictionary

$$\mathcal{I} = \big\{ \texttt{Q}_1 : 4, \texttt{Q}_2 : 8, \texttt{Q}_3 : 1, \texttt{Q}_4 : 3, \texttt{Q}_5 : 6, \texttt{Q}_6 : 2, \texttt{Q}_7 : 7, \texttt{Q}_8 : 5) \big\}.$$

If we define

$$\mathcal{B} := \big\{ \texttt{Q}_1 : 4, \texttt{Q}_2 : 8, \texttt{Q}_3 : 1) \big\},$$

then $\mathcal{B}$ is a partial assignment and $\texttt{dom}(\mathcal{B}) = \{\texttt{Q}_1, \texttt{Q}_2, \texttt{Q}_3\}$. This partial assignment is shown in Figure 5.14 on page 114.

Figure 5.15 on page 115 shows a *Python* program that can be used to create the eight queens puzzle as a CSP.

### 5.4.4 A Backtracking Constraint Solver

One approach to solve a CSP that is both conceptually simple and reasonable efficient is backtracking. The idea is to try to build variable assignments incrementally: We start with an empty dictionary and pick a variable $x_1$ that needs to have a value assigned. For this variable, we choose a value $v_1$ and assign it to this variable. This yields the partial assignment $\{x_1 : v_1\}$. Next, we evaluate all those

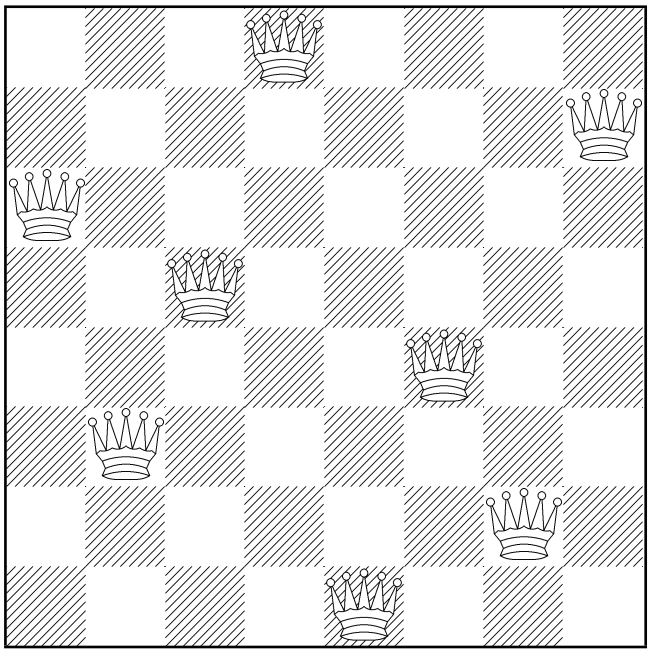

Figure 5.13: A solution of the eight queens problem.

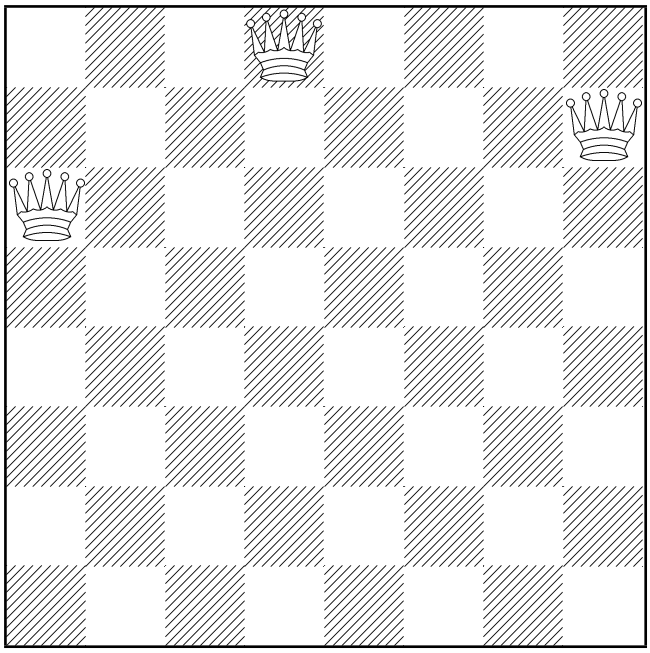

Figure 5.14: The partial assignment $\big\{\mathtt{Q}_1 \mapsto 4, \mathtt{Q}_2 \mapsto 8, \mathtt{Q}_3 \mapsto 1)\big\}$.

constraints that mention only the variable $x_1$ and check whether these constraints are satisfied. If any of these constraints is evaluated as `False`, we try to assign another value to $x_1$ until we find a value that satisfies all constraints that mention only $x_1$.

In general, if we have a partial variable assignment $\mathcal{B}$ of the form

$$\mathcal{B} = \{x_1 : v_1, \cdots, x_k : v_k\}$$

```
def queensCSP():
    'Returns a CSP coding the 8 queens problem.'
    S            = range(1, 8+1)          # used as indices
    Variables    = [ f'Q{i}' for i in S ]
    Values       = { 1, 2, 3, 4, 5, 6, 7, 8 }
    SameRow      = { f'Q{i} != Q{j}' for i in S for j in S if i < j }
    SameDiagonal = { f'abs(Q{i}-Q{j}) != {j-i}' for i in S for j in S if i < j }
    return (Variables, Values, SameRow | SameDiagonal)
```

Figure 5.15: *Python* code to create the CSP representing the eight queens puzzle.

and we already know that all constraints that mention only the variables $x_1, \cdots, x_k$ are satisfied by $\mathcal{B}$, then in order to extend $\mathcal{B}$ we pick another variable $x_{k+1}$ and choose a value $v_{k+1}$ such that all those constraints that mention only the variables $x_1, \cdots, x_k, x_{k+1}$ are satisfied. If we discover that there is no such value $v_{k+1}$, then we have to undo the assignment $x_k : v_k$ and try to find a new value $v_k$ such that, first, those constraints mentioning only the variables $x_1, \cdots, x_k$ are satisfied, and, second, it is possible to find a value $v_{k+1}$ that can be assigned to $x_{k+1}$. This step of going back and trying to find a new value for the variable $x_k$ is called backtracking. It might be necessary to backtrack more than one level and to also undo the assignment of $v_{k-1}$ to $x_{k-1}$ or, indeed, we might be forced to undo the assignments of all variables $x_i, \cdots, x_k$ for some $i \in \{1, \cdots, n\}$. The details of this search procedure are best explained by looking at its implementation. Figure 5.16 on page 115 shows a simple CSP solver that employs backtracking. We discuss this program next.

```
import extractVars as ev

def solve(CSP):
    'Compute a solution for the given constraint satisfaction problem.'
    Variables, Values, Constraints = CSP
    CSP = (Variables,
           Values,
           [(f, ev.extractVars(f) & set(Variables)) for f in Constraints]
          )
    try:
        return backtrack_search({}, CSP)
    except Backtrack:
        return  # no solution found
```

Figure 5.16: A backtracking CSP solver

1. As we need to determine the variables occurring in a given constraint, we import the module `extractVar`. This module implements a function `extractVars`$(f)$ that takes a formula $f$ written as a *Python* expression and returns the set of all variables occurring in $f$.

2. The procedure `solve` takes a constraint satisfaction problem `CSP` as input and tries to find a solution.

   (a) First, in line 5 the `CSP` is split into its three components. However, the first component `Variables` does not have to be a set but rather can also be a list. If `Variables` is a list, then backtracking search will assign these variables in the same order as they appear in this list. This can improve the efficiency of backtracking significantly.

   (b) Next, for every constraint `f` of the given `CSP`, we compute the set of variables that are used in `f`. This is done using the procedure `extractVars`. Of these variables we keep only those variables that also occur in the set `Variables` because we assume that any other *Python* variable occurring in a constraint $f$ has already a value assigned to it and can therefore be regarded as a constant.

   The variables occurring in a constraint `f` are then paired with the constraint `f` and the correspondingly modified data structure is stored in `CSP` and is called an augmented CSP.

   The reason to compute and store these sets of variables is efficiency: When we later check whether a constraint `f` is satisfied for a partial variable assignment `Assignment` where `Assignment` is stored as a dictionary, we only need to check the constraint `f` iff all of the variables occurring in `f` are elements of the domain of `Assignment`. It would be wasteful to compute these sets of all variables occurring in a given formula every time the formula is checked.

   (c) Next, we call the function `backtrack_search` to compute a solution of `CSP`. This function call is enclosed in a `try-except`-block. The function `backtrack_search` either returns a solution or, if it is not able to find a solution, it throws an exception of class `Backtrack`. If this happens, the `except` block silently discards this exception and the procedure `solve` returns without a result.

Next, we discuss the implementation of the procedure `backtrack_search` that is shown in Figure 5.17 on page 117. This procedure receives a partial assignment `Assignment` as input together with an augmented `CSP`. This partial assignment is consistent with `CSP`: If `f` is a constraint of `CSP` such that all the variables occurring in `f` are assigned to in `Assignment`, then evaluating `f` using `Assignment` yields `True`. Initially, this partial assignment is empty and hence trivially consistent. The idea is to extend this partial assignment until it is a complete assignment that satisfies all constraints of the given `CSP`.

1. First, the augmented `CSP` is split into its components.

2. Next, if `Assignment` is already a complete variable assignment, i.e. if the dictionary `Assignment` has as many elements as there are variables, then the fact that `Assignment` is partially consistent implies that it is a solution of the `CSP` and, therefore, it is returned.

3. Otherwise, we have to extend the partial `Assignment`. In order to do so, we first have to select a variable `var` that has not yet been assigned a value in `Assignment` so far. We pick the first variable in the list `Variables` that is yet unassigned. This variable is called `var`.

4. Next, we try to assign a `value` to the selected variable `var`. After assigning a `value` to `var`, we immediately check whether this assignment would be consistent with the constraints using the procedure `isConsistent`. If the partial `Assignment` turns out to be consistent, the partial `Assignment` is extended to the new partial assignment `NewAssign` that satisfies

   ```
   NewAssign[var] = value
   ```

```
def backtrack_search(Assignment, CSP):
    '''
    Given a partial variable assignment, this function tries to complete this
    assignment towards a solution of the CSP.
    '''
    (Variables, Values, Constraints) = CSP
    if len(Assignment) == len(Variables):
        return Assignment
    var = [x for x in Variables if x not in Assignment][0]
    for value in Values:
        try:
            if isConsistent(var, value, Assignment, Constraints):
                NewAssign      = Assignment.copy()
                NewAssign[var] = value
                return backtrack_search(NewAssign, CSP)
        except Backtrack:
            continue
    # all values have been tried without success, no solution has been found
    raise Backtrack()

class Backtrack(Exception):
    pass
```

Figure 5.17: The function `backtrack_search`

and that coincides with `Assignment` for all variables different from `var`. Then, the procedure `backtrack_search` is called recursively to complete this new partial assignment. If this is successful, the resulting assignment is a solution of the CSP and is returned. Otherwise, the recursive call of `backtrack_search` will instead raise an exception. This exception is muted by the `try-except`-block that surrounds the recursive call to `backtrack_search`. In that case, the `for`-loop generates a new possible `value` that can be assigned to the variable `var`. If all possible values have been tried and none was successful, the `for`-loop ends and the statement

```
raise Backtrack()
```

is executed. This raises an exception that signals that the current partial `Assignment` can not be completed into a solution of the `CSP`. This exception is caught by one of the `try-except`-blocks that have been encountered previously.

We still need to discuss the implementation of the auxiliary procedure `isConsistent` shown in Figure 5.18 on page 118. This procedure takes a variable `var`, a `value`, a partial `Assignment` and a set of `Constraints`. It is assumed that `Assignment` is partially consistent with respect to the set `Constraints`, i.e. for every formula `f` occurring in `Constraints` such that

$$\texttt{vars}(\texttt{f}) \subseteq \texttt{dom}(\texttt{Assignment})$$

holds, the formula `f` evaluates to `True` given the `Assignment`. The purpose of `isConsistent` is to

```
1   def isConsistent(var, value, Assignment, Constraints):
2       NewAssign      = Assignment.copy()
3       NewAssign[var] = value
4       return all(eval(f, NewAssign) for (f, Vs) in Constraints
5                                     if var in Vs and Vs <= NewAssign.keys()
6                  )
```

Figure 5.18: The procedure `isConsistent`

check, whether the extended assignment

$$\texttt{NA} := \texttt{Assignment} \cup \{\langle \texttt{var}, \texttt{value} \rangle\}$$

that assigns `value` to the variable `var` is still partially consistent with `Constraints`. To this end, the `for`-loop iterates over all `Formula`s in `Constraints`. However, we only have to check those `Formula`s that contain the variable `var` and, furthermore, have the property that

$$\texttt{Vars}(\texttt{Formula}) \subseteq \texttt{dom}(\texttt{NA}),$$

i.e. all variables occurring in `Formula` need to have a value assigned in `NA`. The reasoning is as follows:

1. If `var` does not occur in `Formula`, then adding `var` to `Assignment` cannot change the result of evaluating `Formula` and as `Assignment` is assumed to be partially consistent with respect to `Formula`, `NA` is also partially consistent with respect to `Formula`.
2. If $\texttt{dom}(\texttt{NA}) \not\subseteq \texttt{Vars}(\texttt{Formula})$, then `Formula` can not be evaluated anyway.

If we use backtracking, we can solve the 8 queens problem in less than a second. For the eight queens puzzle the order in which variables are tried is not particularly important. The reason is that all variables are connected to all other variables. For other problems the ordering of the variables can be very important. The general strategy is that variables that are strongly related to each other should be grouped together in the list `Variables`.

## 5.5 Normalformen für prädikatenlogische Formeln

Im nächsten Abschnitt gehen wir daran, einen Kalkül $\vdash$ für die Prädikaten-Logik zu definieren. Genau wie im Falle der Aussagen-Logik wird dies wesentlich einfacher, wenn wir uns auf Formeln beschränken, die in einer Normalform vorliegen. Bei dieser Normalform handelt es sich nun um sogenannte prädikatenlogische Klauseln. Diese werden ähnlich definiert wie in der Aussagen-Logik: Ein prädikatenlogisches Literal ist eine atomare Formel oder die Negation einer atomaren Formel. Eine prädikatenlogische Klausel ist dann eine Disjunktion prädikatenlogischer Literale. Wir zeigen in diesem Abschnitt, dass jede Formel-Menge $M$ so in eine Menge von prädikatenlogischen Klauseln $K$ transformiert werden kann, dass $M$ genau dann erfüllbar ist, wenn $K$ erfüllbar ist. Daher ist die Beschränkung auf prädikatenlogische Klauseln keine echte Einschränkung. Zunächst geben wir einige Äquivalenzen an, mit deren Hilfe Quantoren manipuliert werden können.

**Satz 41** *Es gelten die folgenden Äquivalenzen:*

1. $\models \neg(\forall x\colon f) \leftrightarrow (\exists x\colon \neg f)$
2. $\models \neg(\exists x\colon f) \leftrightarrow (\forall x\colon \neg f)$
3. $\models \forall x\colon f \wedge \forall x\colon g \leftrightarrow \forall x\colon (f \wedge g)$
4. $\models \exists x\colon f \vee \exists x\colon g \leftrightarrow \exists x\colon (f \vee g)$
5. $\models \forall x\colon \forall y\colon f \leftrightarrow \forall y\colon \forall x\colon f$
6. $\models \exists x\colon \exists y\colon f \leftrightarrow \exists y\colon \exists x\colon f$
7. *Falls $x$ eine Variable ist, für die $x \notin FV(f)$ ist, so haben wir*
   $$\models (\forall x\colon f) \leftrightarrow f \quad \text{und} \quad \models (\exists x\colon f) \leftrightarrow f.$$
8. *Falls $x$ eine Variable ist, für die $x \notin FV(g)$ gilt, so haben wir die folgenden Äquivalenzen:*
   (a) $\models (\forall x\colon f) \vee g \leftrightarrow \forall x\colon (f \vee g)$ und $\models g \vee (\forall x\colon f) \leftrightarrow \forall x\colon (g \vee f)$,
   (b) $\models (\exists x\colon f) \wedge g \leftrightarrow \exists x\colon (f \wedge g)$ und $\models g \wedge (\exists x\colon f) \leftrightarrow \exists x\colon (g \wedge f)$.

Um die Äquivalenzen der letzten Gruppe anwenden zu können, kann es notwendig sein, gebundene Variablen umzubenennen. Ist $f$ eine prädikatenlogische Formel und sind $x$ und $y$ zwei Variablen, wobei $y$ nicht in $f$ auftritt, so bezeichnet $f[x/y]$ die Formel, die aus $f$ dadurch entsteht, dass jedes Auftreten der Variablen $x$ in $f$ durch $y$ ersetzt wird. Beispielsweise gilt

$$\big(\forall u : \exists v : p(u,v)\big)[u/z] = \forall z : \exists v : p(z,v)$$

Damit können wir eine letzte Äquivalenz angeben: Ist $f$ eine prädikatenlogische Formel, ist $x \in BV(f)$ und ist $y$ eine Variable, die in $f$ nicht auftritt, so gilt

$$\models f \leftrightarrow f[x/y].$$

Mit Hilfe der oben stehenden Äquivalenzen und der aussagenlogischen Äquivalenzen, die wir schon kennen, können wir eine Formel so umformen, dass die Quantoren nur noch außen stehen. Eine solche Formel ist dann in pränexer Normalform. Wir führen das Verfahren an einem Beispiel vor: Wir zeigen, dass die Formel

$$\big(\forall x\colon p(x)\big) \rightarrow \big(\exists x\colon p(x)\big)$$

allgemeingültig ist:

$$\begin{array}{ll} & \big(\forall x\colon p(x)\big) \rightarrow \big(\exists x\colon p(x)\big) \\ \Leftrightarrow & \neg\big(\forall x\colon p(x)\big) \vee \big(\exists x\colon p(x)\big) \\ \Leftrightarrow & \big(\exists x\colon \neg p(x)\big) \vee \big(\exists x\colon p(x)\big) \\ \Leftrightarrow & \exists x\colon \big(\neg p(x) \vee p(x)\big) \\ \Leftrightarrow & \exists x\colon \top \\ \Leftrightarrow & \top \end{array}$$

In diesem Fall haben wir Glück gehabt, dass es uns gelungen ist, die Formel als Tautologie zu erkennen. Im Allgemeinen reichen die obigen Umformungen aber nicht aus, um prädikatenlogische Tautologien erkennen zu können. Um Formeln noch stärker vereinfachen zu können, führen wir einen weiteren Äquivalenz-Begriff ein. Diesen Begriff wollen wir vorher durch ein Beispiel motivieren. Wir betrachten die beiden Formeln

$$f_1 = \forall x\colon \exists y\colon p(x,y) \quad \text{und} \quad f_2 = \forall x\colon p\big(x, s(x)\big).$$

Die beiden Formeln $f_1$ und $f_2$ sind nicht äquivalent, denn sie entstammen noch nicht einmal der gleichen Signatur: In der Formel $f_2$ wird das Funktions-Zeichen $s$ verwendet, das in der Formel $f_1$ überhaupt nicht auftritt. Auch wenn die beiden Formeln $f_1$ und $f_2$ nicht äquivalent sind, so besteht zwischen ihnen doch die folgende Beziehung: Ist $S_1$ eine prädikatenlogische Struktur, in der die Formel $f_1$ gilt:

$$\mathcal{S}_1 \models f_1,$$

dann können wir diese Struktur zu einer Struktur $S_2$ erweitern, in der die Formel $f_2$ gilt:

$$\mathcal{S}_2 \models f_2.$$

Dazu muss die Interpretation des Funktions-Zeichens $s$ so gewählt werden, dass für jedes $x$ tatsächlich $p\big(x, s(x)\big)$ gilt. Dies ist möglich, denn die Formel $f_1$ sagt ja aus, dass wir zu jedem $x$ einen Wert $y$ finden, für den $p(x,y)$ gilt. Die Funktion $s$ muss also lediglich zu jedem $x$ dieses $y$ zurück geben.

**Definition 42 (Skolemisierung)**
*Es sei $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle$ eine Signatur. Ferner sei $f$ eine geschlossene $\Sigma$-Formel der Form*

$$f = \forall x_1, \cdots, x_n\colon \exists y\colon g.$$

*Dann wählen wir ein neues $n$-stelliges Funktions-Zeichen $s$, d.h. wir nehmen ein Zeichen $s$, dass in der Signatur $\Sigma$ nicht auftritt und erweitern die Signatur $\Sigma$ zu der Signatur*

$$\Sigma' := \Big\langle \mathcal{V}, \mathcal{F} \cup \{s\}, \mathcal{P}, arity \cup \big\{\langle s, n\rangle\big\} \Big\rangle,$$

*in der wir $s$ als neues $n$-stelliges Funktions-Zeichen deklarieren. Anschließend definieren wir die $\Sigma'$-Formel $f'$ wie folgt:*

$$f' := \texttt{Skolem}(f) := \forall x_1\colon \cdots \forall x_n\colon g\big[y \mapsto s(x_1, \cdots, x_n)\big]$$

*Hierbei bezeichnet der Ausdruck $g\big[y \mapsto s(x_1, \cdots, x_n)\big]$ die Formel, die wir aus $g$ dadurch erhalten, dass wir jedes Auftreten der Variablen $y$ in der Formel $g$ durch den Term $s(x_1, \cdots, x_n)$ ersetzen. Wir sagen, dass die Formel $f'$ aus der Formel $f$ durch einen Skolemisierungs-Schritt hervorgegangen ist.* ◇

**Beispiel**: Es $f$ die folgende Formel aus der Gruppen-Theorie:

$$f := \forall x : \exists y : y * x = 1.$$

Dann gilt

$$\texttt{Skolem}(f) = \forall x : s(x) * x = 1.$$ ◇

In welchem Sinne sind eine Formel $f$ und eine Formel $f'$, die aus $f$ durch einen Skolemisierungs-Schritt hervorgegangen sind, äquivalent? Zur Beantwortung dieser Frage dient die folgende Definition.

**Definition 43 (Erfüllbarkeits-Äquivalenz)**
*Zwei geschlossene Formeln $f$ und $g$ heißen erfüllbarkeits-äquivalent falls $f$ und $g$ entweder beide erfüllbar oder beide unerfüllbar sind. Wenn $f$ und $g$ erfüllbarkeits-äquivalent sind, so schreiben wir*

$$f \approx_e g.$$ ◇

**Beobachtung:** Falls die Formel $f'$ aus der Formel $f$ durch einen Skolemisierungs-Schritt hervorge-

gangen ist, so sind $f$ und $f'$ erfüllbarkeits-äquivalent, denn wir können jede Struktur $\mathcal{S}$, in der die Formel $f$ gilt, zu einer Struktur $\mathcal{S}'$ erweitern, in der auch $f'$ gilt. $\diamond$

Wir können nun ein einfaches Verfahren angeben, um Existenz-Quantoren aus einer Formel zu eliminieren. Dieses Verfahren besteht aus zwei Schritten: Zunächst bringen wir die Formel in pränexe Normalform. Anschließend können wir die Existenz-Quantoren der Reihe nach durch Skolemisierungs-Schritte eliminieren. Nach dem oben gemachten Bemerkungen ist die resultierende Formel zu der ursprünglichen Formel erfüllbarkeits-äquivalent. Dieses Verfahren der Eliminierung von Existenz-Quantoren durch die Einführung neuer Funktions-Zeichen wird als Skolemisierung bezeichnet. Haben wir eine Formel $F$ in pränexe Normalform gebracht und anschließend skolemisiert, so hat das Ergebnis die Gestalt

$$\forall x_1, \cdots, x_n : g$$

und in der Formel $g$ treten keine Quantoren mehr auf. Die Formel $g$ wird auch als die Matrix der obigen Formel bezeichnet. Wir können nun $g$ mit Hilfe der uns aus dem letzten Kapitel bekannten aussagenlogischen Äquivalenzen in konjunktive Normalform bringen. Wir haben dann eine Formel der Gestalt

$$\forall x_1, \cdots, x_n : (k_1 \wedge \cdots \wedge k_m).$$

Dabei sind die $k_i$ Disjunktionen von prädikatenlogischen Literalen. Wenden wir hier die Äquivalenz

$$\forall x \colon (f_1 \wedge f_2) \leftrightarrow (\forall x \colon f_1) \wedge (\forall x \colon f_2)$$

an, so können wir die All-Quantoren auf die einzelnen $k_i$ verteilen und die resultierende Formel hat die Gestalt

$$\big(\forall x_1, \cdots, x_n : k_1\big) \wedge \cdots \wedge \big(\forall x_1, \cdots, x_n : k_m\big).$$

Ist eine Formel $F$ in der obigen Gestalt, so sagen wir, dass $F$ in prädikatenlogischer Klausel-Normalform ist und eine Formel der Gestalt

$$\forall x_1, \cdots, x_n : k,$$

bei der $k$ eine Disjunktion prädikatenlogischer Literale ist, bezeichnen wir als prädikatenlogische Klausel. Ist $M$ eine Menge von Formeln deren Erfüllbarkeit wir untersuchen wollen, so können wir nach dem bisher Gezeigten $M$ immer in eine erfüllbarkeits-äquivalente Menge prädikatenlogischer Klauseln umformen. Da dann nur noch All-Quantoren vorkommen, können wir hier die Notation noch vereinfachen, indem wir vereinbaren, dass alle Formeln implizit allquantifiziert sind, wir lassen also die All-Quantoren weg.

Wozu sind nun die Umformungen in Skolem-Normalform gut? Es geht darum, dass wir ein Verfahren entwickeln wollen, mit dem es möglich ist für eine prädikatenlogische Formel $f$ zu zeigen, dass $f$ allgemeingültig ist, dass also

$$\models f$$

gilt. Wir wissen, dass

$$\models f \quad \text{g.d.w.} \quad \{\neg f\} \models \bot$$

gilt, denn die Formel $f$ ist genau dann allgemeingültig, wenn es keine Struktur gibt, in der die Formel $\neg f$ erfüllbar ist. Wir bilden daher zunächst die Formel $\neg f$ und formen dann diese Formel in prädikatenlogische Klausel-Normalform um. Wir erhalten Klauseln $k_1, \cdots, k_n$, so dass

$$\neg f \approx_e k_1 \wedge \cdots \wedge k_n$$

gilt. Anschließend versuchen wir, aus den Klauseln $k_1, \cdots, k_n$ einen Widerspruch herzuleiten:

$$\{k_1, \cdots, k_n\} \vdash \bot$$

Wenn dies gelingt, dann wissen wir, dass die Menge $\{k_1, \cdots, k_n\}$ unerfüllbar ist. Damit ist auch $\neg f$ unerfüllbar und also ist $f$ allgemeingültig. Damit wir aus den Klauseln $k_1, \cdots, k_n$ einen Widerspruch herleiten können, brauchen wir natürlich noch einen Kalkül $\vdash$, der mit prädikatenlogischen Klauseln arbeitet. Einen solchen Kalkül werden wir im übernächsten Abschnitt vorstellen.

Um das Verfahren näher zu erläutern demonstrieren wir es an einem Beispiel. Wir wollen untersuchen, ob

$$\models (\exists x\colon \forall y\colon p(x,y)) \rightarrow (\forall y\colon \exists x\colon p(x,y))$$

gilt. Wir wissen, dass dies äquivalent dazu ist, dass

$$\Big\{\neg\Big((\exists x\colon \forall y\colon p(x,y)) \rightarrow (\forall y\colon \exists x\colon p(x,y))\Big)\Big\} \models \bot$$

gilt. Wir bringen zunächst die negierte Formel in pränexe Normalform.

$$\begin{array}{rl}
 & \neg\Big((\exists x\colon \forall y\colon p(x,y)) \rightarrow (\forall y\colon \exists x\colon p(x,y))\Big) \\
\leftrightarrow & \neg\Big(\neg(\exists x\colon \forall y\colon p(x,y)) \vee (\forall y\colon \exists x\colon p(x,y))\Big) \\
\leftrightarrow & (\exists x\colon \forall y\colon p(x,y)) \wedge \neg(\forall y\colon \exists x\colon p(x,y)) \\
\leftrightarrow & (\exists x\colon \forall y\colon p(x,y)) \wedge (\exists y\colon \neg\exists x\colon p(x,y)) \\
\leftrightarrow & (\exists x\colon \forall y\colon p(x,y)) \wedge (\exists y\colon \forall x\colon \neg p(x,y))
\end{array}$$

Um an dieser Stelle weitermachen zu können, ist es nötig, die Variablen in dem zweiten Glied der Konjunktion umzubenennen. Wir ersetzen $x$ durch $u$ und $y$ durch $v$ und erhalten

$$\begin{array}{rl}
 & (\exists x\colon \forall y\colon p(x,y)) \wedge (\exists y\colon \forall x\colon \neg p(x,y)) \\
\leftrightarrow & (\exists x\colon \forall y\colon p(x,y)) \wedge (\exists v\colon \forall u\colon \neg p(u,v)) \\
\leftrightarrow & \exists v\colon \Big((\exists x\colon \forall y\colon p(x,y)) \wedge (\forall u\colon \neg p(u,v))\Big) \\
\leftrightarrow & \exists v\colon \exists x\colon \Big((\forall y\colon p(x,y)) \wedge (\forall u\colon \neg p(u,v))\Big) \\
\leftrightarrow & \exists v\colon \exists x\colon \forall y\colon \Big(p(x,y) \wedge (\forall u\colon \neg p(u,v))\Big) \\
\leftrightarrow & \exists v\colon \exists x\colon \forall y\colon \forall u\colon \Big(p(x,y) \wedge \neg p(u,v)\Big)
\end{array}$$

An dieser Stelle müssen wir skolemisieren um die Existenz-Quantoren los zu werden. Wir führen dazu zwei neue Funktions-Zeichen $s_1$ und $s_2$ ein. Dabei gilt $\texttt{arity}(s_1) = 0$ und $\texttt{arity}(s_2) = 0$, denn vor den Existenz-Quantoren stehen keine All-Quantoren.

$$\begin{array}{rl}
 & \exists v\colon \exists x\colon \forall y\colon \forall u\colon \Big(p(x,y) \wedge \neg p(u,v)\Big) \\
\approx_e & \exists x\colon \forall y\colon \forall u\colon \Big(p(x,y) \wedge \neg p(u,s_1)\Big) \\
\approx_e & \forall y\colon \forall u\colon \Big(p(s_2,y) \wedge \neg p(u,s_1)\Big)
\end{array}$$

Da jetzt nur noch All-Quantoren auftreten, können wir diese auch noch weglassen, da wir ja vereinbart haben, dass alle freien Variablen implizit allquantifiziert sind. Damit können wir nun die prädikatenlogische Klausel-Normalform in Mengen-Schreibweise angeben, diese ist

$$M := \Big\{\{p(s_2,y)\}, \{\neg p(u,s_1)\}\Big\}.$$

Wir zeigen, dass die Menge $M$ widersprüchlich ist. Dazu betrachten wir zunächst die Klausel $\{p(s_2, y)\}$ und setzen in dieser Klausel für $y$ die Konstante $s_1$ ein. Damit erhalten wir die Klausel

$$\{p(s_2, s_1)\}. \tag{1}$$

Das Ersetzung von $y$ durch $s_1$ begründen wir damit, dass die obige Klausel ja implizit allquantifiziert ist und wenn etwas für alle $y$ gilt, dann sicher auch für $y = s_1$.

Als nächstes betrachten wir die Klausel $\{\neg p(u, s_1)\}$. Hier setzen wir für die Variablen $u$ die Konstante $s_2$ ein und erhalten dann die Klausel

$$\{\neg p(s_2, s_1)\} \tag{2}$$

Nun wenden wir auf die Klauseln (1) und (2) die Schnitt-Regel an und finden

$$\{p(s_2, s_1)\}, \quad \{\neg p(s_2, s_1)\} \quad \vdash \quad \{\}.$$

Damit haben wir einen Widerspruch hergeleitet und gezeigt, dass die Menge $M$ unerfüllbar ist. Damit ist dann auch

$$\Big\{\neg\Big((\exists x\colon \forall y\colon p(x,y)) \to (\forall y\colon \exists x\colon p(x,y))\Big)\Big\}$$

unerfüllbar und folglich gilt

$$\models (\exists x\colon \forall y\colon p(x,y)) \to (\forall y\colon \exists x\colon p(x,y)).$$

## 5.6 Unifikation

In dem Beispiel im letzten Abschnitt haben wir die Terme $s_1$ und $s_2$ geraten, die wir für die Variablen $y$ und $u$ in den Klauseln $\{p(s_2, y)\}$ und $\{\neg p(u, s_1)\}$ eingesetzt haben. Wir haben diese Terme mit dem Ziel gewählt, später die Schnitt-Regel anwenden zu können. In diesem Abschnitt zeigen wir nun ein Verfahren, mit dessen Hilfe wir die benötigten Terme ausrechnen können. Dazu benötigen wir zunächst den Begriff einer Substitution.

**Definition 44 (Substitution)** *Es sei eine Signatur*

$$\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle$$

*gegeben. Eine $\Sigma$-Substitution ist eine endliche Menge von Paaren der Form*

$$\sigma = \{\langle x_1, t_1\rangle, \cdots, \langle x_n, t_n\rangle\}.$$

*Dabei gilt:*

1. $x_i \in \mathcal{V}$, *die $x_i$ sind also Variablen.*
2. $t_i \in \mathcal{T}_\Sigma$, *die $t_i$ sind also Terme.*
3. *Für $i \neq j$ ist $x_i \neq x_j$, die Variablen sind also paarweise verschieden.*

*Ist $\sigma = \{\langle x_1, t_1\rangle, \cdots, \langle x_n, t_n\rangle\}$ eine $\Sigma$-Substitution, so schreiben wir*

$$\sigma = \big[x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\big].$$

*Außerdem definieren wir den Domain einer Substitution als*

$$dom(\sigma) := \{x_1, \cdots, x_n\}.$$

*Die Menge aller Substitutionen bezeichnen wir mit Subst.* ◇

Substitutionen werden für uns dadurch interessant, dass wir sie auf Terme anwenden können. Ist $t$ ein Term und $\sigma$ eine Substitution, so ist $t\sigma$ der Term, der aus $t$ dadurch entsteht, dass jedes Vorkommen einer Variablen $x_i$ durch den zugehörigen Term $t_i$ ersetzt wird. Die formale Definition folgt.

**Definition 45 (Anwendung einer Substitution)**
*Es sei $t$ ein Term und es sei $\sigma = \big[x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\big]$ eine Substitution. Wir definieren die Anwendung von $\sigma$ auf $t$ (Schreibweise $t\sigma$) durch Induktion über den Aufbau von $t$:*

1. *Falls $t$ eine Variable ist, gibt es zwei Fälle:*
   (a) *$t = x_i$ für ein $i \in \{1, \cdots, n\}$. Dann definieren wir $\quad x_i\sigma := t_i$.*
   (b) *$t = y$ mit $y \in \mathcal{V}$, aber $y \notin \{x_1, \cdots, x_n\}$. Dann definieren wir $\quad y\sigma := y$.*
2. *Andernfalls muss $t$ die Form $t = f(s_1, \cdots, s_m)$ haben. Dann können wir $t\sigma$ durch*
   $$f(s_1, \cdots, s_m)\sigma := f(s_1\sigma, \cdots, s_m\sigma).$$
   *definieren, denn nach Induktions-Voraussetzung sind die Ausdrücke $s_i\sigma$ bereits definiert.* ◇

Genau wie wir Substitutionen auf Terme anwenden können, können wir eine Substitution auch auf prädikatenlogische Klauseln anwenden. Dabei werden Prädikats-Zeichen und Junktoren wie Funktions-Zeichen behandelt. Wir ersparen uns eine formale Definition und geben stattdessen zunächst einige Beispiele. Wir definieren eine Substitution $\sigma$ durch

$$\sigma := \big[x_1 \mapsto c,\ x_2 \mapsto f(d)\big].$$

In den folgenden drei Beispielen demonstrieren wir zunächst, wie eine Substitution auf einen Term angewendet werden kann. Im vierten Beispiel wenden wir die Substitution dann auf eine Klausel in Mengen-Schreibweise an:

1. $x_3\sigma = x_3$,
2. $f(x_2)\sigma = f\big(f(d)\big)$,
3. $h(x_1, g(x_2))\sigma = h\big(c, g(f(d))\big)$.
4. $\big\{p(x_2), q(d, h(x_3, x_1))\big\}\sigma = \big\{p(f(d)),\ q(d, h(x_3, c))\big\}$.

Als nächstes zeigen wir, wie Substitutionen miteinander verknüpft werden können.

**Definition 46 (Komposition von Substitutionen)** *Es seien*

$$\sigma = \big[x_1 \mapsto s_1, \cdots, x_m \mapsto s_m\big] \quad \text{und} \quad \tau = \big[y_1 \mapsto t_1, \cdots, y_n \mapsto t_n\big]$$

*zwei Substitutionen mit $dom(\sigma) \cap dom(\tau) = \{\}$. Dann definieren wir die Komposition von Substitutionen $\sigma\tau$ von $\sigma$ und $\tau$ als*

$$\sigma\tau := \big[x_1 \mapsto s_1\tau, \cdots, x_m \mapsto s_m\tau,\ y_1 \mapsto t_1, \cdots, y_n \mapsto t_n\big]$$ ◇

**Beispiel**: Wir führen das obige Beispiel fort und setzen

$$\sigma := \big[x_1 \mapsto c,\ x_2 \mapsto f(x_3)\big] \quad \text{und} \quad \tau := \big[x_3 \mapsto h(c, c),\ x_4 \mapsto d\big].$$

Dann gilt:

$$\sigma\tau = \big[x_1 \mapsto c,\ x_2 \mapsto f(h(c, c)),\ x_3 \mapsto h(c, c),\ x_4 \mapsto d\big].$$ □

Die Definition der Komposition von Substitutionen ist mit dem Ziel gewählt worden, dass der folgende Satz gilt.

**Satz 47** *Ist $t$ ein Term und sind $\sigma$ und $\tau$ Substitutionen mit $dom(\sigma) \cap dom(\tau) = \{\}$, so gilt*

$$(t\sigma)\tau = t(\sigma\tau).$$ □

Der Satz kann durch Induktion über den Aufbau des Termes $t$ bewiesen werden.

**Definition 48 (Syntaktische Gleichung)** *Unter einer syntaktischen Gleichung verstehen wir in diesem Abschnitt ein Konstrukt der Form $s \doteq t$, wobei einer der beiden folgenden Fälle vorliegen muss:*

1. *$s$ und $t$ sind Terme oder*
2. *$s$ und $t$ sind atomare Formeln.*

*Weiter definieren wir ein syntaktisches Gleichungs-System als eine Menge von syntaktischen Gleichungen.* ◇

Was syntaktische Gleichungen angeht, so machen wir keinen Unterschied zwischen Funktions-Zeichen und Prädikats-Zeichen. Dieser Ansatz ist deswegen berechtigt, weil wir Prädikate ja auch als spezielle Funktionen auffassen können, nämlich als solche Funktionen, die als Ergebnis einen Wahrheitswert aus der Menge $\mathbb{B}$ zurück geben.

**Definition 49 (Unifikator)** *Eine Substitution $\sigma$ löst eine syntaktische Gleichung $s \doteq t$ genau dann, wenn $s\sigma = t\sigma$ ist, wenn also durch die Anwendung von $\sigma$ auf $s$ und $t$ tatsächlich identische Objekte entstehen. Ist $E$ ein syntaktisches Gleichungs-System, so sagen wir, dass $\sigma$ ein Unifikator von $E$ ist wenn $\sigma$ jede syntaktische Gleichung in $E$ löst.* ◇

Ist $E = \{s_1 \doteq t_1, \cdots, s_n \doteq t_n\}$ eine syntaktisches Gleichungs-System und ist $\sigma$ eine Substitution, so definieren wir

$$E\sigma := \{s_1\sigma \doteq t_1\sigma, \cdots, s_n\sigma \doteq t_n\sigma\}.$$

**Beispiel**: Wir verdeutlichen die bisher eingeführten Begriffe anhand eines Beispiels. Wir betrachten die Gleichung

$$p(x_1, f(x_4)) \doteq p(x_2, x_3)$$

und definieren die Substitution

$$\sigma := \big[x_1 \mapsto x_2,\ x_3 \mapsto f(x_4)\big].$$

Die Substitution $\sigma$ löst die obige syntaktische Gleichung, denn es gilt

$$p(x_1, f(x_4))\sigma = p(x_2, f(x_4)) \quad \text{und}$$
$$p(x_2, x_3)\sigma \quad = p(x_2, f(x_4)).$$ ◇

Als nächstes entwickeln wir ein Verfahren, mit dessen Hilfe wir von einer vorgegebenen Menge $E$ von syntaktischen Gleichungen entscheiden können, ob es einen Unifikator $\sigma$ für $E$ gibt. Das Verfahren, das wir entwickeln werden, wurde von Martelli und Montanari veröffentlicht [MM82]. Wir überlegen uns zunächst, in welchen Fällen wir eine syntaktischen Gleichung $s \doteq t$ garantiert nicht lösen können. Da gibt es zwei Möglichkeiten: Eine syntaktische Gleichung

$$f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n)$$

ist sicher dann nicht durch eine Substitution lösbar, wenn $f$ und $g$ verschiedene Funktions-Zeichen sind, denn für jede Substitution $\sigma$ gilt ja

$$f(s_1, \cdots, s_m)\sigma = f(s_1\sigma, \cdots, s_m\sigma) \quad \text{und} \quad g(t_1, \cdots, t_n)\sigma = g(t_1\sigma, \cdots, t_n\sigma).$$

Falls $f \neq g$ ist, haben die Terme $f(s_1, \cdots, s_m)\sigma$ und $g(t_1, \cdots, t_n)\sigma$ verschieden Funktions-Zeichen und können daher syntaktisch nicht identisch werden.

Die andere Form einer syntaktischen Gleichung, die garantiert unlösbar ist, ist

$$x \doteq f(t_1, \cdots, t_n) \quad \text{falls } x \in \mathit{Var}\big(f(t_1, \cdots, t_n)\big).$$

Das diese syntaktische Gleichung unlösbar ist liegt daran, dass die rechte Seite immer mindestens ein Funktions-Zeichen mehr enthält als die linke.

Mit diesen Vorbemerkungen können wir nun ein Verfahren angeben, mit dessen Hilfe es möglich ist, Mengen von syntaktischen Gleichungen zu lösen, oder festzustellen, dass es keine Lösung gibt. Das Verfahren operiert auf Paaren der Form $\langle F, \tau \rangle$. Dabei ist $F$ ein syntaktisches Gleichungs-System und $\tau$ ist eine Substitution. Wir starten das Verfahren mit dem Paar $\langle E, [] \rangle$. Hierbei ist $E$ das zu lösende Gleichungs-System und $[]$ ist die leere Substitution. Das Verfahren arbeitet, indem die im Folgenden dargestellten Reduktions-Regeln solange angewendet werden, bis entweder feststeht, dass die Menge der Gleichungen keine Lösung hat, oder aber ein Paar der Form $\langle \{\}, \sigma \rangle$ erreicht wird. In diesem Fall ist $\sigma$ ein Unifikator der Menge $E$, mit der wir gestartet sind. Es folgen die Reduktions-Regeln:

1. Falls $y \in \mathcal{V}$ eine Variable ist, die nicht in dem Term $t$ auftritt, so können wir die folgende Reduktion durchführen:

   $$\Big\langle E \cup \{y \doteq t\}, \sigma \Big\rangle \quad \rightsquigarrow \quad \Big\langle E[y \mapsto t], \sigma\big[y \mapsto t\big] \Big\rangle$$

   Diese Reduktions-Regel ist folgendermaßen zu lesen: Enthält die zu untersuchende Menge von syntaktischen Gleichungen eine Gleichung der Form $y \doteq t$, wobei die Variable $y$ nicht in $t$ auftritt, dann können wir diese Gleichung aus der gegebenen Menge von Gleichungen entfernen. Gleichzeitig wird die Substitution $\sigma$ in die Substitution $\sigma\big[y \mapsto t\big]$ transformiert und auf die restlichen syntaktischen Gleichungen wird die Substitution $[y \mapsto t]$ angewendet.

2. Wenn die Variable $y$ in dem Term $t$ auftritt, falls also $y \in \mathit{Var}(t)$ ist und wenn außerdem $t \neq y$ ist, dann hat das Gleichungs-System $E \cup \{y \doteq t\}$ keine Lösung, wir schreiben

   $$\Big\langle E \cup \{y \doteq t\}, \sigma \Big\rangle \rightsquigarrow \Omega \quad \text{falls } y \in \mathit{Var}(t) \text{ und } y \neq t.$$

3. Falls $y \in \mathcal{V}$ eine Variable ist und $t$ keine Variable ist, so haben wir folgende Reduktions-Regel:

   $$\Big\langle E \cup \{t \doteq y\}, \sigma \Big\rangle \quad \rightsquigarrow \quad \Big\langle E \cup \{y \doteq t\}, \sigma \Big\rangle.$$

   Diese Regel wird benötigt, um anschließend eine der ersten beiden Regeln anwenden zu können.

4. Triviale syntaktische Gleichungen von Variablen können wir einfach weglassen:

   $$\Big\langle E \cup \{x \doteq x\}, \sigma \Big\rangle \quad \rightsquigarrow \quad \Big\langle E, \sigma \Big\rangle.$$

5. Ist $f$ ein $n$-stelliges Funktions-Zeichen, so gilt

$$\Big\langle E \cup \big\{f(s_1, \cdots, s_n) \doteq f(t_1, \cdots, t_n)\big\}, \sigma \Big\rangle \;\rightsquigarrow\; \Big\langle E \cup \big\{s_1 \doteq t_1, \cdots, s_n \doteq t_n\big\}, \sigma \Big\rangle.$$

Eine syntaktische Gleichung der Form $f(s_1, \cdots, s_n) \doteq f(t_1, \cdots, t_n)$ wird also ersetzt durch die $n$ syntaktische Gleichungen $s_1 \doteq t_1, \cdots, s_n \doteq t_n$ .

Diese Regel ist im übrigen der Grund dafür, dass wir mit Mengen von syntaktischen Gleichungen arbeiten müssen, denn auch wenn wir mit nur einer syntaktischen Gleichung starten, kann durch die Anwendung dieser Regel die Zahl der syntaktischen Gleichungen erhöht werden.

Ein Spezialfall dieser Regel ist

$$\Big\langle E \cup \big\{c \doteq c\big\}, \sigma \Big\rangle \;\rightsquigarrow\; \Big\langle E, \sigma \Big\rangle.$$

Hier steht $c$ für eine Konstante, also ein 0-stelliges Funktions-Zeichen. Triviale Gleichungen über Konstanten können also einfach weggelassen werden.

6. Das Gleichungs-System $E \cup \big\{f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n)\big\}$ hat keine Lösung, falls die Funktions-Zeichen $f$ und $g$ verschieden sind, wir schreiben

$$\Big\langle E \cup \big\{f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n)\big\}, \sigma \Big\rangle \;\rightsquigarrow\; \Omega \qquad \text{falls } f \neq g.$$

Haben wir ein nicht-leeres Gleichungs-System $E$ gegeben und starten mit dem Paar $\langle E, [] \rangle$ , so lässt sich immer eine der obigen Regeln anwenden. Diese geht solange bis einer der folgenden Fälle eintritt:

1. Die 2. oder die 6. Regel ist anwendbar. Dann hat das Gleichungs-System $E$ keine Lösung und als Ergebnis der Unifikation wird $\Omega$ zurück gegeben.
2. Das Paar $\langle E, [] \rangle$ wird reduziert zu einem Paar $\langle \{\}, \sigma \rangle$. Dann ist $\sigma$ ein Unifikator von $E$. In diesem Fall schreiben wir $\sigma = \texttt{mgu}(E)$. Falls $E = \{s \doteq t\}$ ist, schreiben wir auch $\sigma = \texttt{mgu}(s, t)$. Die Abkürzung `mgu` steht hier für "most general unifier".

**Beispiel**: Wir wenden das oben dargestellte Verfahren an, um die syntaktische Gleichung

$$p(x_1, f(x_4)) \doteq p(x_2, x_3)$$

zu lösen. Wir haben die folgenden Reduktions-Schritte:

$$\begin{array}{ll} & \big\langle \big\{p(x_1, f(x_4)) \doteq p(x_2, x_3)\big\}, [] \big\rangle \\ \rightsquigarrow & \big\langle \big\{x_1 \doteq x_2, f(x_4) \doteq x_3\big\}, [] \big\rangle \\ \rightsquigarrow & \big\langle \big\{f(x_4) \doteq x_3\big\}, \big[x_1 \mapsto x_2\big] \big\rangle \\ \rightsquigarrow & \big\langle \big\{x_3 \doteq f(x_4)\big\}, \big[x_1 \mapsto x_2\big] \big\rangle \\ \rightsquigarrow & \big\langle \{\}, \big[x_1 \mapsto x_2,\; x_3 \mapsto f(x_4)\big] \big\rangle \end{array}$$

In diesem Fall ist das Verfahren also erfolgreich und wir erhalten die Substitution

$$\big[x_1 \mapsto x_2,\; x_3 \mapsto f(x_4)\big]$$

als Lösung der oben gegebenen syntaktischen Gleichung. ◇

**Beispiel**: Wir geben ein weiteres Beispiel und betrachten das Gleichungs-System

$$E = \big\{p(h(x_1, c)) \doteq p(x_2),\ q(x_2, d) \doteq q(h(d, c), x_4)\big\}$$

Wir haben folgende Reduktions-Schritte:

$$\begin{array}{rl}
 & \big\langle \big\{p(h(x_1, c)) \doteq p(x_2),\ q(x_2, d) \doteq q(h(d, c), x_4)\big\}, [\,]\big\rangle \\
\rightsquigarrow & \big\langle \big\{p(h(x_1, c)) \doteq p(x_2),\ x_2 \doteq h(d, c),\ d \doteq x_4\big\}, [\,]\big\rangle \\
\rightsquigarrow & \big\langle \big\{p(h(x_1, c)) \doteq p(x_2),\ x_2 \doteq h(d, c),\ x_4 \doteq d\big\}, [\,]\big\rangle \\
\rightsquigarrow & \big\langle \big\{p(h(x_1, c)) \doteq p(x_2),\ x_2 \doteq h(d, c)\big\}, \big[x_4 \mapsto d\big]\big\rangle \\
\rightsquigarrow & \big\langle \big\{p(h(x_1, c)) \doteq p(h(d, c))\big\}, \big[x_4 \mapsto d,\ x_2 \mapsto h(d, c)\big]\big\rangle \\
\rightsquigarrow & \big\langle \big\{h(x_1, c) \doteq h(d, c)\big\}, \big[x_4 \mapsto d,\ x_2 \mapsto h(d, c)\big]\big\rangle \\
\rightsquigarrow & \big\langle \big\{x_1 \doteq d,\ c \doteq c\big\}, \big[x_4 \mapsto d,\ x_2 \mapsto h(d, c)\big]\big\rangle \\
\rightsquigarrow & \big\langle \big\{x_1 \doteq d,\}, \big[x_4 \mapsto d,\ x_2 \mapsto h(d, c)\big]\big\rangle \\
\rightsquigarrow & \big\langle \big\{\big\}, \big[x_4 \mapsto d,\ x_2 \mapsto h(d, c),\ x_1 \mapsto d\big]\big\rangle
\end{array}$$

Damit haben wir die Substitution $\big[x_4 \mapsto d,\ x_2 \mapsto h(d, c),\ x_1 \mapsto d\big]$ als Lösung des anfangs gegebenen syntaktischen Gleichungs-Systems gefunden. ◇

## 5.7 Ein Kalkül für die Prädikatenlogik ohne Gleichheit

In diesem Abschnitt setzen wir voraus, dass unsere Signatur $\Sigma$ das Gleichheits-Zeichen nicht verwendet, denn durch diese Einschränkung wird es wesentlich einfacher, einen vollstängen Kalkül für die Prädikatenlogik einzuführen. Zwar gibt es auch für den Fall, dass die Signatur $\Sigma$ das Gleichheits-Zeichen enthält, einen vollständigen Kalkül. Dieser ist allerdings deutlich aufwendiger als der Kalkül, den wir gleich einführen werden.

**Definition 50 (Resolution)** *Es gelte:*

1. $k_1$ *und* $k_2$ *sind prädikatenlogische Klauseln,*
2. $p(s_1, \cdots, s_n)$ *und* $p(t_1, \cdots, t_n)$ *sind atomare Formeln,*
3. *die syntaktische Gleichung* $p(s_1, \cdots, s_n) \doteq p(t_1, \cdots, t_n)$ *ist lösbar mit*
   $$\mu = \texttt{mgu}\big(p(s_1, \cdots, s_n), p(t_1, \cdots, t_n)\big).$$

*Dann ist*

$$\frac{k_1 \cup \{p(s_1, \cdots, s_n)\} \qquad \{\neg p(t_1, \cdots, t_n)\} \cup k_2}{k_1\mu \cup k_2\mu}$$ *eine Anwendung der Resolutions-Regel.*

◇

Die Resolutions-Regel ist eine Kombination aus der Substitutions-Regel und der Schnitt-Regel. Die Substitutions-Regel hat die Form

$$\frac{k}{k\sigma}.$$

Hierbei ist $k$ eine prädikatenlogische Klausel und $\sigma$ ist eine Substitution. Unter Umständen kann es sein, dass wir vor der Anwendung der Resolutions-Regel die Variablen in einer der beiden Klauseln erst umbenennen müssen bevor wir die Regel anwenden können. Betrachten wir dazu ein Beispiel. Die Klausel-Menge

$$M = \Big\{ \{p(x)\}, \{\neg p(f(x))\} \Big\}$$

ist widersprüchlich. Wir können die Resolutions-Regel aber nicht unmittelbar anwenden, denn die syntaktische Gleichung

$$p(x) \doteq p(f(x))$$

ist unlösbar. Das liegt daran, dass **zufällig** in beiden Klauseln dieselbe Variable verwendet wird. Wenn wir die Variable $x$ in der zweiten Klausel jedoch zu $y$ umbenennen, erhalten wir die Klausel-Menge

$$\Big\{ \{p(x)\}, \{\neg p(f(y))\} \Big\}.$$

Hier können wir die Resolutions-Regel anwenden, denn die syntaktische Gleichung

$$p(x) \doteq p(f(y))$$

hat die Lösung $[x \mapsto f(y)]$. Dann erhalten wir

$$\{p(x)\}, \quad \{\neg p(f(y))\} \quad \vdash \quad \{\}.$$

und haben damit die Inkonsistenz der Klausel-Menge $M$ nachgewiesen.

Die Resolutions-Regel alleine ist nicht ausreichend, um aus einer Klausel-Menge $M$, die inkonsistent ist, in jedem Fall die leere Klausel ableiten zu können: Wir brauchen noch eine zweite Regel. Um das einzusehen, betrachten wir die Klausel-Menge

$$M = \Big\{ \{p(f(x), y), p(u, g(v))\}, \{\neg p(f(x), y), \neg p(u, g(v))\} \Big\}$$

Wir werden gleich zeigen, dass die Menge $M$ widersprüchlich ist. Man kann nachweisen, dass mit der Resolutions-Regel alleine ein solcher Nachweis nicht gelingt. Ein einfacher, aber für die Vorlesung zu aufwendiger Nachweis dieser Behauptung kann geführt werden, indem wir ausgehend von der Menge $M$ alle möglichen Resolutions-Schritte durchführen. Dabei würden wir dann sehen, dass die leere Klausel nie berechnet werden kann. Wir stellen daher jetzt die Faktorisierungs-Regel vor, mir der wir später zeigen werden, dass $M$ widersprüchlich ist.

**Definition 51 (Faktorisierung)** *Es gelte*

1. *$k$ ist eine prädikatenlogische Klausel,*
2. *$p(s_1, \cdots, s_n)$ und $p(t_1, \cdots, t_n)$ sind atomare Formeln,*
3. *die syntaktische Gleichung $p(s_1, \cdots, s_n) \doteq p(t_1, \cdots, t_n)$ ist lösbar,*
4. *$\mu = \texttt{mgu}\big(p(s_1, \cdots, s_n), p(t_1, \cdots, t_n)\big)$.*

*Dann sind*

$$\frac{k \cup \{p(s_1, \cdots, s_n),\ p(t_1, \cdots, t_n)\}}{k\mu \cup \{p(s_1, \cdots, s_n)\mu\}} \quad \textit{und} \quad \frac{k \cup \{\neg p(s_1, \cdots, s_n),\ \neg p(t_1, \cdots, t_n)\}}{k\mu \cup \{\neg p(s_1, \cdots, s_n)\mu\}}$$

*Anwendungen der Faktorisierungs-Regel.* ◇

Wir zeigen, wie sich mit Resolutions- und Faktorisierungs-Regel die Widersprüchlichkeit der Menge $M$ beweisen lässt.

1. Zunächst wenden wir die Faktorisierungs-Regel auf die erste Klausel an. Dazu berechnen wir den Unifikator

   $$\mu = \mathtt{mgu}\big(p(f(x), y), p(u, g(v))\big) = [y \mapsto g(v), u \mapsto f(x)].$$

   Damit können wir die Faktorisierungs-Regel anwenden:

   $$\big\{p(f(x), y), p(u, g(v))\big\} \quad \vdash \quad \big\{p(f(x), g(v))\big\}.$$

2. Jetzt wenden wir die Faktorisierungs-Regel auf die zweite Klausel an. Dazu berechnen wir den Unifikator

   $$\mu = \mathtt{mgu}\big(\neg p(f(x), y), \neg p(u, g(v))\big) = [y \mapsto g(v), u \mapsto f(x)].$$

   Damit können wir die Faktorisierungs-Regel anwenden:

   $$\big\{\neg p(f(x), y), \neg p(u, g(v))\big\} \quad \vdash \quad \big\{\neg p(f(x), g(v))\big\}.$$

3. Wir schließen den Beweis mit einer Anwendung der Resolutions-Regel ab. Der dabei verwendete Unifikator ist die leere Substitution, es gilt also $\mu = []$.

   $$\big\{p(f(x), g(v))\big\}, \quad \big\{\neg p(f(x), g(v))\big\} \quad \vdash \quad \{\}.$$

Ist $M$ eine Menge von prädikatenlogischen Klauseln und ist $k$ eine prädikatenlogische Klausel, die durch Anwendung der Resolutions-Regel und der Faktorisierungs-Regel aus $M$ hergeleitet werden kann, so schreiben wir

$$M \vdash k.$$

Dies wird als $M$ leitet $k$ her gelesen.

**Definition 52 (Allabschluss)** *Ist $k$ eine prädikatenlogische Klausel und ist $\{x_1, \cdots, x_n\}$ die Menge aller Variablen, die in $k$ auftreten, so definieren wir den Allabschluss $\forall(k)$ der Klausel $k$ als*

$$\forall(k) := \forall x_1 \colon \cdots \forall x_n \colon k.$$ ◇

Die für uns wesentlichen Eigenschaften des Beweis-Begriffs $M \vdash k$ werden in den folgenden beiden Sätzen zusammengefasst.

**Satz 53 (Korrektheits-Satz)**
*Ist $M = \{k_1, \cdots, k_n\}$ eine Menge von Klauseln und gilt $M \vdash k$, so folgt*

$$\models \forall(k_1) \wedge \cdots \wedge \forall(k_n) \rightarrow \forall(k).$$

*Falls also eine Klausel $k$ aus einer Menge $M$ hergeleitet werden kann, so ist $k$ tatsächlich eine Folgerung aus $M$.* □

Die Umkehrung des obigen Korrektheits-Satzes gilt nur für die leere Klausel. Sie wurde 1965 von John A. Robinson bewiesen [Rob65].

**Satz 54 (Widerlegungs-Vollständigkeit (Robinson, 1965))**
*Ist $M = \{k_1, \cdots, k_n\}$ eine Menge von Klauseln und gilt $\models \forall(k_1) \wedge \cdots \wedge \forall(k_n) \rightarrow \bot$, so folgt*

$$M \vdash \{\}.$$ □

Damit haben wir nun ein Verfahren in der Hand, um für eine gegebene prädikatenlogischer Formel $f$ die Frage, ob $\models f$ gilt, untersuchen zu können.

1. Wir berechnen zunächst die Skolem-Normalform von $\neg f$ und erhalten dabei so etwas wie

   $$\neg f \approx_e \forall x_1, \cdots, x_m : g.$$

2. Anschließend bringen wir die Matrix $g$ in konjunktive Normalform:

   $$g \leftrightarrow k_1 \wedge \cdots \wedge k_n.$$

   Daher haben wir nun

   $$\neg f \approx_e k_1 \wedge \cdots \wedge k_n$$

   und es gilt:

   $$\models f \quad \text{g.d.w.} \quad \{\neg f\} \models \bot \quad \text{g.d.w.} \quad \{k_1, \cdots, k_n\} \models \bot.$$

3. Nach dem Korrektheits-Satz und dem Satz über die Widerlegungs-Vollständigkeit gilt

   $$\{k_1, \cdots, k_n\} \models \bot \quad \text{g.d.w.} \quad \{k_1, \cdots, k_n\} \vdash \bot.$$

   Wir versuchen also, nun die Widersprüchlichkeit der Menge $M = \{k_1, \cdots, k_n\}$ zu zeigen, indem wir aus $M$ die leere Klausel ableiten. Wenn diese gelingt, haben wir damit die Allgemeingültigkeit der ursprünglich gegebenen Formel $f$ gezeigt.

**Beispiel**: Zum Abschluss demonstrieren wir das skizzierte Verfahren an einem Beispiel. Wir gehen von folgenden Axiomen aus:

1. Jeder Drache ist glücklich, wenn alle seine Kinder fliegen können.
2. Rote Drachen können fliegen.
3. Die Kinder eines roten Drachens sind immer rot.

Wie werden zeigen, dass aus diesen Axiomen folgt, dass alle roten Drachen glücklich sind. Als erstes formalisieren wir die Axiome und die Behauptung in der Prädikatenlogik. Wir wählen die Signatur

$$\Sigma_{Drache} := \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle$$

wobei die Mengen $\mathcal{V}$, $\mathcal{F}$, $\mathcal{P}$ und *arity* wie folgt definiert sind:

1. $\mathcal{V} := \{x, y, z\}$.
2. $\mathcal{F} = \{\}$.
3. $\mathcal{P} := \{\mathit{rot}, \mathit{fliegt}, \mathit{glücklich}, \mathit{kind}\}$.
4. $\mathit{arity} := \{\langle \mathit{rot}, 1\rangle, \langle \mathit{fliegt}, 1\rangle, \langle \mathit{glücklich}, 1\rangle, \langle \mathit{kind}, 2\rangle\}$

Das Prädikat $\mathit{kind}(x, y)$ soll genau dann wahr sein, wenn $x$ ein Kind von $y$ ist. Formalisieren wir die Axiome und die Behauptung, so erhalten wir die folgenden Formeln $f_1, \cdots, f_4$:

1. $f_1 := \forall x : \Big(\forall y : \big(\mathit{kind}(y, x) \rightarrow \mathit{fliegt}(y)\big) \rightarrow \mathit{glücklich}(x)\Big)$
2. $f_2 := \forall x : \big(\mathit{rot}(x) \rightarrow \mathit{fliegt}(x)\big)$
3. $f_3 := \forall x : \big(\mathit{rot}(x) \rightarrow \forall y : \big(\mathit{kind}(y, x) \rightarrow \mathit{rot}(y)\big)\big)$
4. $f_4 := \forall x : \big(\mathit{rot}(x) \rightarrow \mathit{glücklich}(x)\big)$

Wir wollen zeigen, dass die Formel

$$f := f_1 \wedge f_2 \wedge f_3 \rightarrow f_4$$

allgemeingültig ist. Wir betrachten also die Formel $\neg f$ und stellen fest

$$\neg f \leftrightarrow f_1 \wedge f_2 \wedge f_3 \wedge \neg f_4.$$

Als nächstes müssen wir diese Formel in eine Menge von Klauseln umformen. Da es sich hier um eine Konjunktion mehrerer Formeln handelt, können wir die einzelnen Formeln $f_1$, $f_2$, $f_3$ und $\neg f_4$ getrennt in Klauseln umwandeln.

1. Die Formel $f_1$ kann wie folgt umgeformt werden:

$$\begin{aligned}
f_1 &= \forall x : \Big(\forall y : \big(\mathit{kind}(y, x) \rightarrow \mathit{fliegt}(y)\big) \rightarrow \mathit{glücklich}(x)\Big) \\
&\leftrightarrow \forall x : \Big(\neg\forall y : \big(\mathit{kind}(y, x) \rightarrow \mathit{fliegt}(y)\big) \vee \mathit{glücklich}(x)\Big) \\
&\leftrightarrow \forall x : \Big(\neg\forall y : \big(\neg\mathit{kind}(y, x) \vee \mathit{fliegt}(y)\big) \vee \mathit{glücklich}(x)\Big) \\
&\leftrightarrow \forall x : \Big(\exists y : \neg\big(\neg\mathit{kind}(y, x) \vee \mathit{fliegt}(y)\big) \vee \mathit{glücklich}(x)\Big) \\
&\leftrightarrow \forall x : \Big(\exists y : \big(\mathit{kind}(y, x) \wedge \neg\mathit{fliegt}(y)\big) \vee \mathit{glücklich}(x)\Big) \\
&\leftrightarrow \forall x : \exists y : \Big(\big(\mathit{kind}(y, x) \wedge \neg\mathit{fliegt}(y)\big) \vee \mathit{glücklich}(x)\Big) \\
&\approx_e \forall x : \Big(\big(\mathit{kind}(s(x), x) \wedge \neg\mathit{fliegt}(s(x))\big) \vee \mathit{glücklich}(x)\Big)
\end{aligned}$$

   Im letzten Schritt haben wir dabei die Skolem-Funktion $s$ mit $\mathit{arity}(s) = 1$ eingeführt. Anschaulich berechnet diese Funktion für jeden Drachen $x$, der nicht glücklich ist, ein Kind $s(x)$,

das nicht fliegen kann. Wenn wir in der Matrix dieser Formel das "$\vee$" noch ausmultiplizieren, so erhalten wir die beiden Klauseln

$$k_1 := \big\{\mathit{kind}(s(x), x), \mathit{glücklich}(x)\big\},$$
$$k_2 := \big\{\neg\mathit{fliegt}(s(x)), \mathit{glücklich}(x)\big\}.$$

2. Analog finden wir für $f_2$:

$$\begin{array}{rcl} f_2 & = & \forall x : \big(\mathit{rot}(x) \rightarrow \mathit{fliegt}(x)\big) \\ & \leftrightarrow & \forall x : \big(\neg\mathit{rot}(x) \vee \mathit{fliegt}(x)\big) \end{array}$$

Damit ist $f_2$ zu folgender Klauseln äquivalent:

$$k_3 := \big\{\neg\mathit{rot}(x), \mathit{fliegt}(x)\big\}.$$

3. Für $f_3$ sehen wir:

$$\begin{array}{rcl} f_3 & = & \forall x : \Big(\mathit{rot}(x) \rightarrow \forall y : \big(\mathit{kind}(y, x) \rightarrow \mathit{rot}(y)\big)\Big) \\ & \leftrightarrow & \forall x : \Big(\neg\mathit{rot}(x) \vee \forall y : \big(\neg\mathit{kind}(y, x) \vee \mathit{rot}(y)\big)\Big) \\ & \leftrightarrow & \forall x : \forall y : \big(\neg\mathit{rot}(x) \vee \neg\mathit{kind}(y, x) \vee \mathit{rot}(y)\big) \end{array}$$

Das liefert die folgende Klausel:

$$k_4 := \big\{\neg\mathit{rot}(x), \neg\mathit{kind}(y, x), \mathit{rot}(y)\big\}.$$

4. Umformung der Negation von $f_4$ liefert:

$$\begin{array}{rcl} \neg f_4 & = & \neg\forall x : \big(\mathit{rot}(x) \rightarrow \mathit{glücklich}(x)\big) \\ & \leftrightarrow & \neg\forall x : \big(\neg\mathit{rot}(x) \vee \mathit{glücklich}(x)\big) \\ & \leftrightarrow & \exists x : \neg\big(\neg\mathit{rot}(x) \vee \mathit{glücklich}(x)\big) \\ & \leftrightarrow & \exists x : \big(\mathit{rot}(x) \wedge \neg\mathit{glücklich}(x)\big) \\ & \approx_e & \mathit{rot}(d) \wedge \neg\mathit{glücklich}(d) \end{array}$$

Die hier eingeführte Skolem-Konstante $d$ steht für einen unglücklichen roten Drachen. Das führt zu den Klauseln

$$k_5 = \big\{\mathit{rot}(d)\big\},$$
$$k_6 = \big\{\neg\mathit{glücklich}(d)\big\}.$$

Wir müssen also untersuchen, ob die Menge $M$, die aus den folgenden Klauseln besteht, widersprüchlich ist:

1. $k_1 = \big\{\mathit{kind}(s(x), x),\ \mathit{glücklich}(x)\big\}$
2. $k_2 = \big\{\neg\mathit{fliegt}(s(x)),\ \mathit{glücklich}(x)\big\}$
3. $k_3 = \big\{\neg\mathit{rot}(x),\ \mathit{fliegt}(x)\big\}$
4. $k_4 = \big\{\neg\mathit{rot}(x),\ \neg\mathit{kind}(y, x),\ \mathit{rot}(y)\big\}$

5. $k_5 = \{rot(d)\}$
6. $k_6 = \{\neg glücklich(d)\}$

Sei also $M := \{k_1, k_2, k_3, k_4, k_5, k_6\}$. Wir zeigen, dass $M \vdash \bot$ gilt:

1. Es gilt
   $$\texttt{mgu}\big(rot(d), rot(x)\big) = [x \mapsto d].$$
   Daher können wir die Resolutions-Regel auf die Klauseln $k_5$ und $k_4$ wie folgt anwenden:
   $$\{rot(d)\},\ \{\neg rot(x), \neg kind(y,x), rot(y)\} \vdash \{\neg kind(y,d), rot(y)\}.$$
2. Wir wenden nun auf die resultierende Klausel und auf die Klausel $k_1$ die Resolutions-Regel an. Dazu berechnen wir zunächst
   $$\texttt{mgu}\big(kind(y,d), kind(s(x),x)\big) = [y \mapsto s(d), x \mapsto d].$$
   Dann haben wir
   $$\{\neg kind(y,d), rot(y)\},\ \{kind(s(x),x), glücklich(x)\} \vdash \{glücklich(d), rot(s(d))\}.$$
3. Jetzt wenden wir auf die eben abgeleitete Klausel und die Klausel $k_6$ die Resolutions-Regel an. Wir haben:
   $$\texttt{mgu}\big(glücklich(d), glücklich(d)\big) = []$$
   Also erhalten wir
   $$\{glücklich(d), rot(s(d))\},\ \{\neg glücklich(d)\} \vdash \{rot(s(d))\}.$$
4. Auf die Klausel $\{rot(s(d))\}$ und die Klausel $k_3$ wenden wir die Resolutions-Regel an. Zunächst haben wir
   $$\texttt{mgu}\big(rot(s(d)), \neg rot(x)\big) = [x \mapsto s(d)]$$
   Also liefert die Anwendung der Resolutions-Regel:
   $$\{rot(s(d))\},\ \{\neg rot(x), fliegt(x)\} \vdash \{fliegt(s(d))\}$$
5. Um die so erhaltenen Klausel $\{fliegt(s(d))\}$ mit der Klausel $k_3$ resolvieren zu können, berechnen wir
   $$\texttt{mgu}\big(fliegt(s(d)), fliegt(s(x))\big) = [x \mapsto d]$$
   Dann liefert die Resolutions-Regel
   $$\{fliegt(s(d))\},\ \{\neg fliegt(s(x)), glücklich(x)\} \vdash \{glücklich(d)\}.$$
6. Auf das Ergebnis $\{glücklich(d)\}$ und die Klausel $k_6$ können wir nun die Resolutions-Regel anwenden:
   $$\{glücklich(d)\},\ \{\neg glücklich(d)\} \vdash \{\}.$$

Da wir im letzten Schritt die leere Klausel erhalten haben, ist insgesamt $M \vdash \bot$ nachgewiesen worden und damit haben wir gezeigt, dass alle kommunistischen Drachen glücklich sind. ◇

**Aufgabe 12**: Die von Bertrant Russell definierte *Russell-Menge R* ist definiert als die Menge aller der Mengen, die sich nicht selbst enthalten. Damit gilt also

$$\forall x : \big(x \in R \leftrightarrow \neg x \in x\big).$$

Zeigen Sie mit Hilfe des in diesem Abschnitt definierten Kalküls, dass diese Formel widersprüchlich ist.

**Aufgabe 13**: Gegeben seien folgende Axiome:

1. Jeder Barbier rasiert alle Personen, die sich nicht selbst rasieren.
2. Kein Barbier rasiert jemanden, der sich selbst rasiert.

Zeigen Sie, dass aus diesen Axiomen logisch die folgende Aussage folgt:

Alle Barbiere sind blond.

## 5.8 *Prover9* und *Mace4**

Der im letzten Abschnitt beschriebene Kalkül lässt sich automatisieren und bildet die Grundlage moderner automatischer Beweiser. Gleichzeitig lässt sich auch die Suche nach Gegenbeispielen automatisieren. Wir stellen in diesem Abschnitt zwei Systeme vor, die diesen Zwecken dienen.

1. *Prover9* dient dazu, automatisch prädikatenlogische Formeln zu beweisen.
2. *Mace4* untersucht, ob eine gegebene Menge prädikatenlogischer Formeln in einer endlichen Struktur erfüllbar ist. Gegebenenfalls wird diese Struktur berechnet.

Die beiden Programme *Prover9* und *Mace4* wurden von William McCune [McC10] entwickelt, stehen unter der GPL (*Gnu General Public Licence*) und können unter der Adresse

`http://www.cs.unm.edu/~mccune/prover9/download/`

im Quelltext heruntergeladen werden. Wir diskutieren zunächst *Prover9* und schauen uns anschließend *Mace4* an.

### 5.8.1 Der automatische Beweiser *Prover9*

*Prover9* ist ein Programm, das als Eingabe zwei Mengen von Formeln bekommt. Die erste Menge von Formeln wird als Menge von *Axiomen* interpretiert, die zweite Menge von Formeln sind die zu beweisenden *Thereme*, die aus den Axiomen gefolgert werden sollen. Wollen wir beispielsweise zeigen, dass in der Gruppen-Theorie aus der Existenz eines links-inversen Elements auch die Existenz eines rechts-inversen Elements folgt und dass außerdem das links-neutrale Element auch rechts-neutral ist, so können wir zunächst die Gruppen-Theorie wie folgt axiomatisieren:

1. $\forall x : e \cdot x = x$,
2. $\forall x : \exists y : y \cdot x = e$,

3. $\forall x : \forall y : \forall z : (x \cdot y) \cdot z = x \cdot (y \cdot z)$.

Wir müssen nun zeigen, dass aus diesen Axiomen die beiden Formeln

$$\forall x : x \cdot e = x \quad \text{und} \quad \forall x : \exists y : y \cdot x = e$$

logisch folgen. Wir können diese Formeln wie in Abbildung 5.19 auf Seite 136 gezeigt für *Prover9* darstellen. Der Anfang der Axiome wird in dieser Datei durch "`formulas(sos)`" eingeleitet und durch das Schlüsselwort "`end_of_list`" beendet. Zu beachten ist, dass sowohl die Schlüsselwörter als auch die einzelnen Formel jeweils durch einen Punkt "`.`" beendet werden. Die Axiome in den Zeilen 2, 3, und 4 drücken aus, dass

1. `e` ein links-neutrales Element ist,
2. zu jedem Element $x$ ein links-inverses Element $y$ existiert und
3. das Assoziativ-Gesetz gilt.

Aus diesen Axiomen folgt, dass das `e` auch ein rechts-neutrales Element ist und dass außerdem zu jedem Element $x$ ein rechts-neutrales Element $y$ existiert. Diese beiden Formeln sind die zu beweisenden Ziele und werden in der Datei durch "`formulas`(goal)" markiert. Trägt die in Abbildung 5.19 gezeigte Datei den Namen "`group2.in`", so können wir das Programm *Prover9* mit dem Befehl

```
prover9 -f group2.in
```

starten und erhalten als Ergebnis die Information, dass die beiden in Zeile 8 und 9 gezeigten Formeln tatsächlich aus den vorher angegebenen Axiomen folgen. Ist eine Formel nicht beweisbar, so gibt es zwei Möglichkeiten: In bestimmten Fällen kann *Prover9* tatsächlich erkennen, dass ein Beweis unmöglich ist. In diesem Fall bricht das Programm die Suche nach einem Beweis mit einer entsprechenden Meldung ab. Wenn die Dinge ungünstig liegen, ist es auf Grund der Unentscheidbarkeit der Prädikatenlogik nicht möglich zu erkennen, dass die Suche nach einem Beweis scheitern muss. In einem solchen Fall läuft das Programm solange weiter, bis kein freier Speicher mehr zur Verfügung steht und bricht dann mit einer Fehlermeldung ab.

```
1  formulas(sos).
2  all x (e * x = x).                              % left neutral
3  all x exists y (y * x = e).                     % left inverse
4  all x all y all z ((x * y) * z = x * (y * z)).  % associativity
5  end_of_list.
6
7  formulas(goals).
8  all x (x * e = x).                              % right neutral
9  all x exists y (x * y = e).                     % right inverse
10 end_of_list.
```

Figure 5.19: Textuelle Darstellung der Axiome der Gruppentheorie.

*Prover9* versucht, einen indirekten Beweis zu führen. Zunächst werden die Axiome in prädikatenlogische Klauseln überführt. Dann wird jedes zu beweisenden Theorem negiert und die negierte Formel wird ebenfalls in Klauseln überführt. Anschließend versucht *Prover9* aus der Menge

aller Axiome zusammen mit den Klauseln, die sich aus der Negation eines der zu beweisenden Theoreme ergeben, die leere Klausel herzuleiten. Gelingt dies, so ist bewiesen, dass das jeweilige Theorem tatsächlich aus den Axiomen folgt. Abbildung 5.20 zeigt eine Eingabe-Datei für *Prover9*, bei der versucht wird, das Kommutativ-Gesetz aus den Axiomen der Gruppentheorie zu folgern. Der Beweis-Versuch mit *Prover9* schlägt allerdings fehl. In diesem Fall wird die Beweissuche nicht endlos fortgesetzt. Dies liegt daran, dass es *Prover9* gelingt, in endlicher Zeit alle aus den gegebenen Voraussetzungen folgenden Formeln abzuleiten. Leider ist ein solcher Fall eher die Ausnahme als die Regel.

```
1  formulas(sos).
2  all x (e * x = x).                                % left neutral
3  all x exists y (y * x = e).                       % left inverse
4  all x all y all z ((x * y) * z = x * (y * z)).    % associativity
5  end_of_list.
6
7  formulas(goals).
8  all x all y (x * y = y * x).                      % * is commutative
9  end_of_list.
```

Figure 5.20: Gilt das Kommutativ-Gesetz in allen Gruppen?

### 5.8.2 *Mace4*

Dauert ein Beweisversuch mit *Prover9* endlos, so ist zunächst nicht klar, ob das zu beweisende Theorem gilt. Um sicher zu sein, dass eine Formel nicht aus einer gegebenen Menge von Axiomen folgt, reicht es aus, eine Struktur zu konstruieren, in der alle Axiome erfüllt sind, in der das zu beweisende Theorem aber falsch ist. Das Programm *Mace4* dient genau dazu, solche Strukturen zu finden. Das funktioniert natürlich nur, solange die Strukturen endlich sind. Abbildung 5.21 zeigt eine Eingabe-Datei, mit deren Hilfe wir die Frage, ob es endliche nicht-kommutative Gruppen gibt, unter Verwendung von *Mace4* beantworten können. In den Zeilen 2, 3 und 4 stehen die Axiome der Gruppen-Theorie. Die Formel in Zeile 5 postuliert, dass für die beiden Elemente $a$ und $b$ das Kommutativ-Gesetz nicht gilt, dass also $a \cdot b \neq b \cdot a$ ist. Ist der in Abbildung 5.21 gezeigte Text in einer Datei mit dem Namen "*group.in*" gespeichert, so können wir *Mace4* durch das Kommando

*mace4 -f group.in*

starten. *Mace4* sucht für alle positiven natürlichen Zahlen $n = 1, 2, 3, \cdots$, ob es eine Struktur $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ mit $card(U) = n$ gibt, in der die angegebenen Formeln gelten. Bei $n = 6$ wird *Mace4* fündig und berechnet tatsächlich eine Gruppe mit 6 Elementen, in der das Kommutativ-Gesetz verletzt ist.

Abbildung 5.22 zeigt einen Teil der von *Mace4* produzierten Ausgabe. Die Elemente der Gruppe sind die Zahlen $0, \cdots, 5$, die Konstante $a$ ist das Element 0, $b$ ist das Element 1, $e$ ist das Element 2. Weiter sehen wir, dass das Inverse von 0 wieder 0 ist, das Inverse von 1 ist 1 das Inverse von 2 ist 2, das Inverse von 3 ist 4, das Inverse von 4 ist 3 und das Inverse von 5 ist 5. Die Multiplikation wird durch die folgende Gruppen-Tafel realisiert:

```
1  formulas(theory).
2  all x (e * x = x).                                % left neutral
3  all x exists y (y * x = e).                       % left inverse
4  all x all y all z ((x * y) * z = x * (y * z)).    % associativity
5  a * b != b * a.                                   % a and b do not commute
6  end_of_list.
```

Figure 5.21: Gibt es eine Gruppe, in der das Kommutativ-Gesetz nicht gilt?

| $\circ$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 2 | 3 | 0 | 1 | 5 | 4 |
| 1 | 4 | 2 | 1 | 5 | 0 | 3 |
| 2 | 0 | 1 | 2 | 3 | 4 | 5 |
| 3 | 5 | 0 | 3 | 4 | 2 | 1 |
| 4 | 1 | 5 | 4 | 2 | 3 | 0 |
| 5 | 3 | 4 | 5 | 0 | 1 | 2 |

Diese Gruppen-Tafel zeigt, dass

$$a \circ b = 0 \circ 1 = 3, \quad \text{aber} \quad b \circ a = 1 \circ 0 = 4$$

gilt, mithin ist das Kommutativ-Gesetz tatsächlich verletzt.

**Bemerkung**: Der Theorem-Beweiser *Prover9* ist ein Nachfolger des Theorem-Beweisers *Otter*. Mit Hilfe von *Otter* ist es William McCune 1996 gelungen, die Robbin'sche Vermutung zu beweisen [McC97]. Dieser Beweis war damals sogar der New York Times eine Schlagzeile wert, nachzulesen unter

`http://www.nytimes.com/library/cyber/week/1210math.html`.

Dies zeigt, dass automatische Theorem-Beweiser durchaus nützliche Werkzeuge sein können. Nichtsdestoweniger ist die Prädikatenlogik unentscheidbar und bisher sind nur wenige offene mathematische Probleme mit Hilfe von automatischen Beweisern gelöst worden. Das wird sich vermutlich auch in der näheren Zukunft nicht ändern. ◇

## 5.9 Reflexion

1. Was ist eine Signatur?
2. Wie haben wir die Menge $\mathcal{T}_\Sigma$ der $\Sigma$-Terme definiert?
3. Was ist eine atomare Formel?
4. Wie haben wir die Menge $\mathbb{F}_\Sigma$ der $\Sigma$-Formeln definiert?
5. Was ist eine $\Sigma$-Struktur?
6. Es sei $\mathcal{S}$ eine $\Sigma$-Struktur. Wie haben wir den Begriff der $\mathcal{S}$-Variablen-Belegung definiert?
7. Wie haben wir die Semantik von $\Sigma$-Formeln definiert?

```
============================== DOMAIN SIZE 6 ========================

=== Mace4 starting on domain size 6. ===

============================== MODEL ================================

interpretation( 6, [number=1, seconds=0], [

        function(a, [ 0 ]),

        function(b, [ 1 ]),

        function(e, [ 2 ]),

        function(f1(_), [ 0, 1, 2, 4, 3, 5 ]),

        function(*(_,_), [
                            2, 3, 0, 1, 5, 4,
                            4, 2, 1, 5, 0, 3,
                            0, 1, 2, 3, 4, 5,
                            5, 0, 3, 4, 2, 1,
                            1, 5, 4, 2, 3, 0,
                            3, 4, 5, 0, 1, 2 ])
]).

============================== end of model =========================
```

Figure 5.22: Ausgabe von *Mace4*.

8. Wann ist eine prädikatenlogische Formel allgemeingültig?
9. Was bedeutet die Schreibweise $\mathcal{S} \models F$ für eine $\Sigma$-Struktur $\mathcal{S}$ und eine $\Sigma$-Formel $F$?
10. Wann ist eine Menge von prädikatenlogischen Formeln unerfüllbar?
11. Was ist ein Constraint Satisfaction Problem?
12. Wie funktioniert Backtracking?
13. Warum kommt es beim Backtracking auf die Reihenfolge an, in der die verschiedenen Variablen instantiiert werden?
14. Was sind prädikatenlogische Klauseln und welche Schritte müssen wir durchführen, um eine gegebene prädikatenlogische Formel in eine erfüllbarkeits-äquivalente Menge von Klauseln zu überführen?
15. Was ist eine Substitution?
16. Was ist ein Unifikator?

17. Geben Sie die Regeln von Martelli und Montanari an!
18. Wie ist die Resolutions-Regel definiert und warum ist es eventuell erforderlich, Variablen umzubenennen, bevor die Resolutions-Regel angewendet werden kann?
19. Was ist die Faktorisierungs-Regel?
20. Wie gehen wir vor, wenn wir die Allgemeingültigkeit einer prädikatenlogischen Formel $f$ nachweisen wollen?

# Bibliography


[Can95] Georg Cantor. Beiträge zur Begründung der transfiniten Mengenlehre. *Mathematische Annalen*, 46:481–512, 1895.

[Ced18] Naomi R. Ceder. *The Quick Python Book*. Manning Publications, 3rd edition, 2018.

[CPR11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Communications of the ACM*, 54(5):88–98, 2011.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.

[Lip98] Seymour Lipschutz. *Set Theory and Related Topics*. McGraw-Hill, New York, 1998.

[Lut13] Mark Lutz. *Learning Python*. O'Reilly and Associates, 5th edition, 2013.

[McC97] William McCune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19:263–276, December 1997.

[McC10] William McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[MGS96] Martin Müller, Thomas Glaß, and Karl Stroetmann. Automated modular termination proofs for real prolog programs. In Radhia Cousot and David A. Schmidt, editors, *SAS*, volume 1145 of *Lecture Notes in Computer Science*, pages 220–237. Springer, 1996.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[Ric53] Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 83, 1953.

[Rob65] John A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1996.

[Tur36] Alan M. Turing. On computable numbers, with an application to the "Entscheidungsproblem". *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

# Index