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

— Summer 2024 —

Baden-Wuerttemberg Cooperative State University (DHBW)

Prof. Dr. Karl Stroetmann

April 22, 2024

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.
```

As the lecture notes are constantly changing, you should do so regularly.

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 the efficiency of algorithms in the lecture on algorithms. This lecture will therefore focus on the correctness of algorithms. The rest of this section will further motivate the importance of the 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, history shows that these failures happen. Here is a list of software problems that have made it to the headlines in recent years.

1. In 2018 and 2019 two Boeing 737 MAX planes crashed because of a software problem in the Maneuvering Characteristics Augmentation System system.

   This error lead to the death of 346 passengers and crew.

2. Between 1999 and 2015 the British Post Office used a faulty accounting software provided by Fujitsu. As a result of the buggy accounting, over 900 employees of the British Post Office were falsely convicted. Some of them were even imprisoned. Four of those that were falsely convicted committed suicide. These tragic incidents are know as the Horizon IT scandal.

3. In 1996, the very first Ariane 5 rocket self-destructed as a result of a software error.

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 part of this foundation that has immediate applications in computer science.

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

(b) Logic is used to build interactive theorem provers that are able to establish the correctness of software. For example, *MicroSoft*™ has build the Lean Prover as part of their research in software engineering.

(c) The correctness of digital circuits can be verified using automatic theorem provers that are based on propositional logic. For example, *Cadence*™ has built the Jasper Formal Verification Platform.

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

good programmer $\neq$ 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, 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 discusses two different methods that can be used to prove the correctness of a program:

- Computational induction is the method of choice for proving the correctness of recursive algorithms.
- Symbolic verification 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

- $\neg$ (not),
- $\wedge$ (and),
- $\vee$ (or),
- $\rightarrow$ (if $\cdots$ then),
- $\leftrightarrow$ (if and only if).

First-order logic also investigates the quantifiers

- $\forall$ (for all),
- $\exists$ (there 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 Vampire, 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 8. 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

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

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.

```python
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

`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*

$$\texttt{name}(t) = f. \qquad \Box$$

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*".* $\Box$

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. $t \notin \mathit{TF} \quad \Leftrightarrow \quad \texttt{stops}(t, a) \rightsquigarrow 2.$

   If the first argument of `stops` is not a test function, then `stops(`t`, `a`)` returns the number 2.

2. $t \in \mathit{TF} \,\wedge\, \texttt{name}(t) = n \,\wedge\, n(a) \downarrow \quad \Leftrightarrow \quad \texttt{stops}(t, a) \rightsquigarrow 1.$

   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. $t \in \mathit{TF} \,\wedge\, \texttt{name}(t) = n \,\wedge\, n(a) \uparrow \quad \Leftrightarrow \quad \texttt{stops}(t, a) \rightsquigarrow 0.$

   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

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

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*`)`

```
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*.

Since we have already noted that *turing* is test function, according to the specification of the function `stops` there are only two cases left:

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

Let us consider these cases in turn.

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

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

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

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

   `alan(turing)`

   is evaluated:

   (a) Since we have assumed for this case that the expression $\texttt{stops}(turing, turing)$ 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. $\texttt{stops}(turing, turing) \leadsto 1$.

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

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

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

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

(a) Since we have assumed for this case that the expression `stops`(*turing*, *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 machine 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:

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

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. $\exists r : \Bigl(n_1(a_1, \cdots, a_k) \rightsquigarrow r \quad \wedge \quad n_2(a_1, \cdots, a_k) \rightsquigarrow r\Bigr)$

   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`.

```
1   def stops(t, a):
2       l = """def loop(x):
3                while True:
4                    x = 1
5           """
6       e = equal(l, t, a);
7       if e == 2:
8           return 2
9       else:
10          return 1 - e
```

Figure 2.3: An implementation 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 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

Correctness Proofs

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

```
def power(m, n):
    if n == 0:
        return 1
    p = power(m, n // 2)
    if n % 2 == 0:
        return p * p
    else:
        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

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

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) $n \,\%\, 2 = 0$, 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:

   $$\begin{array}{lcl} \texttt{power}(m,n) & = & \texttt{power}(m,k) \cdot \texttt{power}(m,k) \\ & \stackrel{\mathrm{IV}}{=} & m^k \cdot m^k \\ & = & m^{2 \cdot k} \\ & = & m^n. \end{array}$$

   (b) $n \,\%\, 2 = 1$, 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. $\surd$

3. **Termination proof**: Every time the function `power` is invoked as $\texttt{power}(m,n)$ and $n > 0$, the recursive invocation has the form $\texttt{power}(m, n \,/\!/\, 2)$ and, since $n \,/\!/\, 2 < n$ 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. $\surd$ □

```python
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`.

Example: The function `div_mod` that is shown in Figure 3.2 satisfies the specification

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

Proof: Assume that $m, n \in \mathbb{N}$, where $n > 0$. Furthermore, assume

$$\bar{q}, \bar{r} = \texttt{div\_mod}(m,n).$$

In order to prove the correctness of `div_mod`, we have to show two formulas:

$$m = \bar{q}\cdot n + \bar{r} \tag{3.1}$$

$$\bar{r} < n \tag{3.2}$$

Since `div_mod` is defined recursively, the proof of these formulas is done by computational induction.

B.C.: $m < n$

In this case we have $\bar{q} = 0$ and $\bar{r} = m$. In order to prove (3.1) we note that

$$\begin{aligned}
& m = \bar{q}\cdot n + \bar{r} \\
\Leftrightarrow\;& m = 0\cdot n + m \quad \surd
\end{aligned}$$

To prove (3.2) we note that

$$\begin{aligned} & \bar{r} < n \\ \Leftrightarrow \quad & m < n \quad \surd \end{aligned}$$

Here $m < n$ is true because this condition is the assumption of the base case.

I.S.: $m \,//\, 2 \mapsto m$

By induction hypotheses we know that our claim is true for the recursive invocation of `div_mod` in line 4. Therefore we have the following:

$$m \,//\, 2 = q \cdot n + r \tag{3.3}$$

$$r < n \tag{3.4}$$

In order to complete the induction step we have to perform a case distinction that is analogous to the test of the second `if`-statement in the implementation of `div_mod`.

(a) $2 \cdot r + m \,\%\, 2 < n$

In this case we have $\bar{q} = 2 \cdot q$ and $\bar{r} = 2 \cdot r + m \,\%\, 2$. In order to prove (3.1) we note the following:

$$m = \bar{q} \cdot n + \bar{r} \tag{3.5}$$

$$\Leftrightarrow \quad m = 2 \cdot q \cdot n + 2 \cdot r + m \,\%\, 2 \tag{3.6}$$

We will derive equation (3.6) from equation (3.3). To this end, we multiply equation (3.3) by 2. This yields:

$$2 \cdot m \,//\, 2 = 2 \cdot q \cdot n + 2 \cdot r.$$

If we add $m \,\%\, 2$ to this equation we get

$$2 \cdot m \,//\, 2 + m \,\%\, 2 = 2 \cdot q \cdot n + 2 \cdot r + m \,\%\, 2.$$

As we have $2 \cdot m \,//\, 2 + m \,\%\, 2 = m$ the last equation can be simplified to

$$m = 2 \cdot q \cdot n + 2 \cdot r + m \,\%\, 2.$$

However, this is just equation (3.6) which we had to prove. $\surd$

Next, we show that $\bar{r} < n$. This is equivalent to

$$2 \cdot r + m \,\%\, 2 < n.$$

However, this inequation is the condition of this case of the case distinction and is therefore valid. $\surd$

(b) $2 \cdot r + m \,\%\, 2 \geq n$

In this case we have $\bar{q} = 2 \cdot q + 1$ and $\bar{r} = 2 \cdot r + m \,\%\, 2 - n$. We start with the proof of (3.1).

$$\begin{aligned} & m = \bar{q} \cdot n + \bar{r} \\ \Leftrightarrow \quad & m = (2 \cdot q + 1) \cdot n + 2 \cdot r + m \,\%\, 2 - n \\ \Leftrightarrow \quad & m = 2 \cdot q \cdot n + 2 \cdot r + m \,\%\, 2 \end{aligned}$$

This last equation follows from equation (3.3) as follows:

$$\begin{aligned}
& m \,//\, 2 = q \cdot n + r \\
\Rightarrow\quad & 2 \cdot m \,//\, 2 = 2 \cdot q \cdot n + 2 \cdot r \\
\Rightarrow\quad & 2 \cdot m \,//\, 2 + m \,\%\, 2 = 2 \cdot q \cdot n + 2 \cdot r + m \,\%\, 2 \\
\Rightarrow\quad & m = 2 \cdot q \cdot n + 2 \cdot r + m \,\%\, 2
\end{aligned}$$

Next, we show that $\bar{r} < n$. This is equivalent to

$$2 \cdot r + m \,\%\, 2 - n < n$$

From (3.4) we know that

$$\begin{aligned}
& r < n \\
\Rightarrow\quad & r + 1 \leq n \\
\Rightarrow\quad & 2 \cdot r + 2 \leq 2 \cdot n \\
\Rightarrow\quad & 2 \cdot r + m \,\%\, 2 + 1 \leq 2 \cdot n \quad \text{since } m \,\%\, 2 \leq 1 \\
\Rightarrow\quad & 2 \cdot r + m \,\%\, 2 < 2 \cdot n \\
\Rightarrow\quad & 2 \cdot r + m \,\%\, 2 - n < n \surd
\end{aligned}$$

T.: As $m \,//\, 2 < m$ for all $m \geq n$ and $n > 0$ it is obvious that we will eventually have $m < n$. But then the function `div_mod` terminates.

Before we can tackle the next exercise, we need to prove the following lemma.

Lemma 6 (Euclid) *Assume $a, b \in \mathbb{N}$ such that $b > 0$. Then we have*

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

Proof: The function $\texttt{cd}(a, b)$ computes the set of common divisors of a and b and is therefore defined as

$$\texttt{cd}(a, b) := \bigl\{ t \in \mathbb{N} \mid a \,\%\, t = 0 \wedge b \,\%\, t = 0 \bigr\}.$$

The function `gcd` is related to the function `cd` by the equation

$$\texttt{gcd}(a, b) = \max\bigl(\texttt{cd}(a, b)\bigr).$$

Hence it is sufficient if we can show that

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

This is an equation between two sets and therefore is equivalent to showing that both

$$\texttt{cd}(a, b) \subseteq \texttt{cd}(b, a \,\%\, b) \quad \text{and} \quad \texttt{cd}(b, a \,\%\, b) \subseteq \texttt{cd}(a, b)$$

holds. We show these two statements separately.

1. We show that $\texttt{cd}(a, b) \subseteq \texttt{cd}(b, a \,\%\, b)$.

   Assume $t \in \texttt{cd}(a, b)$. Then there are $u, v \in \mathbb{N}$ such that

   $$a = t \cdot u \quad \text{and} \quad b = t \cdot v.$$

Since $a = q \cdot b + a \,\%\, b$ where $q = a \mathbin{//} b$ we have

$$a \,\%\, b = a - q \cdot b = t \cdot u - q \cdot t \cdot v = t \cdot (u - q \cdot v).$$

This shows that t divides $a \,\%\, b$. Since t also divides b we therefore have $t \in \texttt{cd}(b, a \,\%\, b)$. $\surd$

2. We show that $\texttt{cd}(b, a \,\%\, b) \subseteq \texttt{cd}(a, b)$.

   Assume $t \in \texttt{cd}(b, a \,\%\, b)$. Then there are $u, v \in \mathbb{N}$ such that

   $$b = t \cdot u \quad \text{and} \quad a \,\%\, b = t \cdot v.$$

   Since $a = q \cdot b + a \,\%\, b$ where $q = a \mathbin{//} b$ we have

   $$a = q \cdot t \cdot u + t \cdot v = t \cdot (q \cdot u + v).$$

   This shows that t divides a. Since t also divides b we therefore have $t \in \texttt{cd}(a, b)$. $\surd$

This concludes the proof. □

```python
def ggt(x, y):
    if y == 0:
        return x
    return ggt(y, x % y)
```

Figure 3.3: The function ggt.

Exercise 3: Prove that the function ggt that is shown in Figure 3.3 computes the greatest common divisor of its arguments. ◇

```python
def isqrt(n):
    if n == 0:
        return 0
    r = isqrt(n // 4)
    if (2 * r + 1) ** 2 <= n:
        return 2 * r + 1
    else:
        return 2 * r
```

Figure 3.4: The function isqrt.

Exercise 4: The integer square root of a natural number n is defined as

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

Prove that the function isqrt that is shown in Figure 3.4 on page 21 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 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 two examples.

3.2.1 Iterative Squaring

In the previous section we have implemented a recursive version of iterative squaring and verified its correctness via computational induction. In this section we implement an iterative version of iterative squaring and verify its correctness via symbolic execution. Consider the program shown in Figure 3.5.

```
1  def power(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          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 given 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 such that the index includes a counter that counts the number of loop iterations. For concreteness, consider the program shown in Figure 3.5. Here, in line 5 the variable `r` has the index n on the right side of the assignment, while it has the index $n+1$ on the left side of the assignment in line 5. The index n denotes the number of times that the test `y > 0` of the `while` loop has been executed. After the `while`-loop finishes, the variable `r` is indexed as $\mathtt{r}_N$ 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

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

We will show that, provided $a \geq 1$, the `while` loop satisfies the invariant

$$r_n \cdot x_n^{y_n} = a^b. \tag{3.7}$$

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

(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

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

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

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

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

This shows the validity of equation (3.7). If the `while` loop terminates, we must have $y_N = 0$. If $n = N$, then equation (3.7) yields:

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

In the step from the second line to the third line we have used the fact that, since $x_0 \neq 0$, we also have that $x_n \neq 0$ for all $n \in \{0, 1, \cdots, N\}$ and therefore $x_N^0 = 1$. Hence we have shown that $r_N = a^b$. The `while` loop terminates because we have

$$y_{n+1} = y_n \,//\, 2 < y_n \quad \text{as long as} \quad y_n > 0$$

and therefore y_n must eventually become 0. Thus we have proven that `power`$(a, b) = a^b$ holds. $\square$

3.2.2 Integer Square Root

We continue with another example that demonstrates the method of symbolic execution. Figure 3.6 shows a function that is able to compute the integer square root of its argument a as long as a is less than 2^{64}, i.e. if a is represented as an unsigned number, then it can be represented with 64 bits. In the implementation of the function `root` we have already indexed the variables. Note that there is no need to index the variable `a` since this variable is never updated. To prove the correctness of this program, we first have to establish invariants for the variables p_n and r_n. In order to be able to formulate the invariant for the variable `r`, we have to understand that the function `root` computes the integer square root of its argument a bit by bit starting at the most significant bit. Let us denote the mathematical function that computes the integer square root of a number a with `isqrt`. If we represent $\texttt{isqrt}(a)$ in the binary system, we have

$$\texttt{isqrt}(a) = \sum_{i=0}^{31} b_i \cdot 2^i, \quad \text{where } b_i \in \{0, 1\}.$$

In order to compute $\texttt{isqrt}(a)$ we start with the last bit b_{31}. If the square of 2^{31} is less than a, then $b_{31} = 1$. Otherwise we have $b_{31} = 0$. Assume now that we have already computed the bits b_{32-1}, $b_{32-2}, \cdots, b_{32-(n-1)}$. In order to compute b_{32-n} we have to check whether

$$\left(2^{32-n} + \sum_{i=32-(n-1)}^{31} b_i \cdot 2^i\right)^2 \leq a.$$

If it is, then $b_{32-n} = 1$, else $b_{32-n} = 0$. With this in mind, we can state the invariants that hold when the `while` loop in line 4 is entered:

1. $p_n = 2^{32-n} \quad \text{for } n = 1, \cdots, 32 \text{ and } p_{33} = 0.$
2. $r_n = \displaystyle\sum_{i=32-(n-1)}^{31} b_i \cdot 2^i$ where the b_i are the bits of $\texttt{isqrt}(a)$ in the binary representation.

```
1 def root(a):
2     r_1 = 0
3     p_1 = 2 ** 31
4     while p_n > 0:
5         if a >= (r_n + p_n) ** 2:
6             r_{n+1} = r_n + p_n
7         p_{n+1} = p_n // 2
8     return r_N
```

Figure 3.6: An annotated programm to compute powers.

We proceed to prove the first invariant by induction.

B.C.: $n = 1$

$p_1 = 2^{31} = 2^{32-1}$.

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

$p_{n+1} = p_n \ // \ 2 \stackrel{IV}{=} 2^{32-n} \ // \ 2 \ = \ 2^{32-n-1} \ = \ 2^{32-(n+1)}$.

This holds as long as $n+1 \leq 32$. Since we have $p_{32} = 2^{32-32} = 2^0 = 1$ it follows that

$$p_{33} \ = \ p_{32} \ // \ 2 = 1 \ // \ 2 = 0.$$

This shows that the body of the `while` loop is executed 32 times. This was to be expected, as each execution of this loop computes one bit of the result.

We proceed to prove the second invariant by induction.

B.C.: $n = 1$

We have $r_1 = 0$ and also $\sum\limits_{i=33-1}^{31} b_i \cdot 2^i = \sum\limits_{i=32}^{31} b_i \cdot 2^i = 0$, since the last sum is empty.

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

Now we need a case distinction with respect to the test $a \geq (r_n + p_n)^2$.

(a) $a \geq (r_n + p_n)^2$.

In this case we have that $r_n + p_n \leq \mathtt{isqrt}(a)$ and since $p_n = 2^{32-n}$ this shows that

$$r_n + 2^{32-n} \leq \mathtt{isqrt}(a).$$

This implies that $b_{32-n} = 1$. Therefore we have

$$\begin{aligned} r_{n+1} &= r_n + p_n \\ &\stackrel{IV}{=} \sum_{i=33-n}^{31} b_i \cdot 2^i + 2^{32-n} \\ &= \sum_{i=33-n}^{31} b_i \cdot 2^i + b_{32-n} \cdot 2^{32-n} \\ &= \sum_{i=33-(n+1)}^{31} b_i \cdot 2^i \end{aligned}$$

(b) $a < (r_n + p_n)^2$.

In this case we have that $r_n + p_n > \mathtt{isqrt}(a)$ and since $p_n = 2^{32-n}$ this shows that

$$r_n + 2^{32-n} > \mathtt{isqrt}(a).$$

This implies that $b_{32-n} = 0$. Therefore we have

$$\begin{aligned} r_{n+1} &= r_n \\ &\stackrel{IV}{=} \sum_{i=33-n}^{31} b_i \cdot 2^i + 0 \\ &\stackrel{IV}{=} \sum_{i=33-n}^{31} b_i \cdot 2^i + 0 \cdot 2^{32-n} \\ &= \sum_{i=33-n}^{31} b_i \cdot 2^i + b_{32-n} \cdot 2^{32-n} \\ &= \sum_{i=33-(n+1)}^{31} b_i \cdot 2^i \end{aligned}$$

Since the program terminates when $n = 33$ we have

$$\texttt{root}(a) = r_N = r_{33} = \sum_{i=33-33}^{31} b_i \cdot 2^i = \sum_{i=0}^{31} b_i \cdot 2^i = \texttt{isqrt}(a). \qquad \Box$$

Exercise 5: Use the method of symbolic program execution to prove the correctness of the implementation of the Euclidean algorithm that is shown in Figure 3.7 on page 26. 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 x and x 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 $a > 0$. Note that in order to carry out the proof you have to distinguish between the mathematical function ggt that computes the greatest common divisor and the *Python* function gcd that is implemented in Figure 3.7. ◇

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

Figure 3.7: 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?

Chapter 4

Propositional Calculus

4.1 Introduction

Propositional calculus (also known as propositional logic) deals with the connection of propositions (a.k.a. simple statements) through logical connectives. Here, logical connectives are words like "and", "or", "not", "if $\cdots$, then", and "exactly if". To start with, we define the notion of an atomic propositions: An atomic proposition is a sentence that

- expresses a fact that is either true or false and
- that does not contain any logical connectives.

  This last property is the reason for calling the proposition atomic.

Examples of atomic propositions are the following:

1. "*The sun is shining*"
2. "*It is raining.*"
3. "*There is a rainbow in the sky.*"

Atomic propositions can be combined by means of logical connectives into composite propositions. An example for a composite propositions would be

If the sun is shining and it is raining, then there is a rainbow in the sky. (1)

This statement is composed of the three atomic propositions

- "*The sun is shining.*",
- "*It is raining.*", and
- "*There is a rainbow in the sky.*"

using the logical connectives "*and*" and "*if* $\cdots$*, then*". Propositional calculus investigates how the truth value of composite statements is calculated from the truth values of the propositions. Furthermore, it investigates how new statements can be derived from given statements.

In order to analyze the structure of complex statements we introduce propositional variables. These propositional variables are just names that denote atomic propositions. Furthermore, we introduce symbols serving as mathematical operators for the logical connectives "not", "and", "or", "if, $\cdots$ then", and "if and only if".

1. $\neg a$ is read as not a
2. $a \wedge b$ is read as a and b
3. $a \vee b$ is read as a or b
4. $a \rightarrow b$ is read as if a, then b
5. $a \leftrightarrow b$ is read as a if and only if b

Propositional formulas are built from propositional variables using the propositional operators shown above and can have an arbitrary complexity. Using propositional operators, the statement (1) can be written as follows:

$$\texttt{sunny} \wedge \texttt{raining} \rightarrow \texttt{rainBow}.$$

Here, we have used `sunny`, `rainy` and `rainBow` as propositional variables.

Some propositional formulas are always true, no matter how the propositional variables are interpreted. For example, the propositional formula

$$p \vee \neg p$$

is always true, it does not matter whether the proposition denoted by p is true or false. A propositional formula that is always true is known as a tautology. There are also propositional formulas that are never true. For example, the propositional formula

$$p \wedge \neg p$$

is always false. A propositional formula is called satisfiable, if there is at least one way to assign truth values to the variables such that the formula is true. Otherwise the formula is called unsatisfiable. In this lecture we will discuss a number of different algorithms to check whether a formula is satisfiable. These algorithms are very important in a number of industrial applications. For example, a very important application is the design of digital circuits. Furthermore, a number of logic puzzles can be translated into propositional formulas and finding a solution to these puzzles amounts to checking the satisfiable of these formulas. For example, we will solve the eight queens puzzle in this way.

The rest of this chapter is structured as follows:

1. We list several applications of propositional logic.
2. We define the notion of propositional formulas, i.e. we define the set of strings that are propositional formulas.

   This is known as the syntax of propositional formulas.
3. Next, we discuss the evaluation of propositional formulas and implement the evaluation in *Python*.

   This is known as the semantics of propositional formulas.
4. Then we formally define the notions tautology and satisfiability for propositional formulas.

5. We discuss algebraic manipulations of propositional formulas and introduce the conjunctive normal form.

   Some algorithms discussed later require that the propositional formulas have conjunctive normal form.

6. After that we discuss the concept of a logical derivation. The purpose of a logical derivation is to derive new formulas from a given set of formulas.

7. Finally, we discuss the Davis-Putnam algorithm for checking the satisfiability of a set of propositional formulas. As an application, we solve the eight queens puzzle using this algorithm.

4.2 Applications of Propositional Logic

Propositional logic is not only the basis of first order logic, but it also has important practical applications. As there are many different applications of propositional logic, I will only list those applications which I have seen myself during the years when I did work in industry.

1. Analysis and design of electronic circuits.

   Modern digital circuits are comprised of hundreds of millions of logical gates.[1] A logical gate is a building block, that represents a logical connective such as "and", "or", "not" as an electronic circuit.

   The complexity of modern digital circuits would be unmanageable without the use of computer-aided verification methods. The methods used are applications of propositional logic. A very concrete application is circuit comparison. Here two digital circuits are represented as propositional formulas. Afterwards it is tried to show the equivalence of these formulas by means of propositional logic. Software tools, which are used for the verification of digital circuits sometimes cost more than 100 000 \$. For example, the company Magma offers the equivalence checker Quartz Formal at a price of 150 000 \$ per license. Such a license is then valid for three years.

2. Controlling the signals and switches of railroad stations.

   At a large railway station, there are several hundred switches and signals that have to be reset all the time to provide routes for the trains. For safety reasons, different routes must not cross each other. The individual routes are described by so-called closure plans. The correctness of these closure plans can be analyzed via propositional formulas.

3. A number of logical puzzles can be coded as propositional formulas and can then be solved with the algorithm of Davis and Putnam. For example, we will discuss the eight queens puzzle in this lecture. This puzzle asks to place eight queens on a chess board such that no two queens can attack each other.

[1] The web page `https://en.wikipedia.org/wiki/Transistor_count` gives an overview of the complexity of modern processors.

4.3 The Formal Definition of Propositional Formulas

In this section we first cover the syntax of propositional formulas. After that, we discuss their semantics. The syntax defines the way in which we represent formulas as strings and how we can combine formulas into a proof. The semantics of propositional logic is concerned with the meaning of propositional formulas. We will first define the semantics of propositional logic with the help of set theory. Then, we implement this semantics in *Python*.

4.3.1 The Syntax of Propositional Formulas

We define propositional formulas as strings. To this end we assume a set $\mathcal{P}$ of so called propositional variables as given. Typically, $\mathcal{P}$ is the set of all lower case Latin characters, which additionally may be indexed. For example, we will use

p, q, r, p_1, p_2, p_3

as propositional variables. Then, propositional formulas are strings that are formed from the alphabet

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

We define the set $\mathcal{F}$ of propositional formulas by induction:

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

   Here $\top$ denotes the formula that is always true, while $\bot$ denotes the formula that is always false. The formula $\top$ is called "verum"[2], while $\bot$ is called "falsum"[3].

2. If $p \in \mathcal{P}$, then $p \in \mathcal{F}$.

   Every propositional variable is also a propositional formula.

3. If $f \in \mathcal{F}$, then $\neg f \in \mathcal{F}$.

   The formula $\neg f$ (read: not f) is called the negation of f.

4. If $f_1, f_2 \in \mathcal{F}$, then we also have

$(f_1 \vee f_2) \in \mathcal{F}$	(*read*:	f_1 or f_2	also: disjunction	of f_1 and f_2),
$(f_1 \wedge f_2) \in \mathcal{F}$	(*read*:	f_1 and f_2	also: conjunction	of f_1 and f_2),
$(f_1 \rightarrow f_2) \in \mathcal{F}$	(*read*:	if f_1, then f_2	also: implication	of f_1 and f_2),
$(f_1 \leftrightarrow f_2) \in \mathcal{F}$	(*read*:	f_1 if and only if f_2	also: biconditional	of f_1 and f_2).

The set $\mathcal{F}$ of propositional formulas is the smallest set of those strings formed from the characters in the alphabet $\mathcal{A}$ that has the closure properties given above.

Example: Assume that $\mathcal{P} := \{p, q, r\}$. Then we have the following:

1. $p \in \mathcal{F}$,

2. $(p \wedge q) \in \mathcal{F}$,

[2] "Verum" is the Latin word for "true".
[3] "Falsum" is the Latin word for "false".

3. $\Big(\big((\neg p \rightarrow q) \vee (q \rightarrow \neg p)\big) \rightarrow r\Big) \in \mathcal{F}$. $\Box$

In order to save parentheses we agree on the following rules:

1. Outermost parentheses are dropped. Therefore, we write

   $p \wedge q \quad$ instead of $\quad (p \wedge q)$.

2. The negation operator $\neg$ has a higher precedence than all other operators.

3. The operators $\vee$ and $\wedge$ associate to the left. Therefore, we write

   $p \wedge q \wedge r \quad$ instead of $\quad (p \wedge q) \wedge r$.

4. **In this lecture** the logical operators $\wedge$ and $\vee$ have the same precedence. This is different from the programming language *Python*. In *Python* the operator "`and`" has a higher precedence than the operator "`or`".

   In the programming languages `C` and *Java* the operator "`&&`" also has a higher precedence than the operator "`||`".

5. The operator $\rightarrow$ is right associative, i.e. we write

   $p \rightarrow q \rightarrow r \quad$ instead of $\quad p \rightarrow (q \rightarrow r)$.

6. The operators $\vee$ and $\wedge$ have a higher precedence than the operator $\rightarrow$. Therefore, we write

   $p \wedge q \rightarrow r \quad$ instead of $\quad (p \wedge q) \rightarrow r$.

7. The operator $\rightarrow$ has a higher precedence than the operator $\leftrightarrow$. Therefore, we write

   $p \rightarrow q \leftrightarrow r \quad$ instead of $\quad (p \rightarrow q) \leftrightarrow r$.

8. You should note that the operator $\leftrightarrow$ neither associates to the left nor to the right. Therefore, the expression

   $p \leftrightarrow q \leftrightarrow r$

   is **ill-defined** and has to be parenthesised. If you encounter this type of expression in a book it is usually meant as an abbreviation for the expression

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

   We will not use this kind of abbreviation.

Remark: Later, we will conduct a series of proofs that prove mathematical statements about formulas. In these proofs we will make use of propositional connectives. In order to distinguish these connectives from the connectives of propositional logic we agree on the following:

1. Inside a propositional formula, the propositional connective "not" is written as "$\neg$".

   When we prove a statement about propositional formulas, we use the word "`not`" instead.

2. Inside a propositional formula, the propositional connective "and" is written as "$\wedge$".

   When we prove a statement about propositional formulas, we use the word "`and`" instead.

3. Inside a propositional formula, the propositional connective "or" is written as "$\vee$".
   When we prove a statement about propositional formulas, we use the word "`or`" instead.
4. Inside a propositional formula, the propositional connective "if $\cdots$, then" is written as "$\rightarrow$".
   When we prove a statement about propositional formulas, we use the symbol "$\Rightarrow$" instead.
5. Inside a propositional formula, the propositional connective "if and only if" is written as "$\leftrightarrow$".
   When we prove a statement about propositional formulas, we use the symbol "$\Leftrightarrow$" instead. $\diamond$

4.3.2 Semantics of Propositional Formulas

In this section we define the meaning a.k.a. the semantics of propositional formulas. To this end we assign truth values to propositional formulas. First, we define the set $\mathbb{B}$ of truth values:

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

Next, we define the notion of a propositional valuation.

Definition 7 (Propositional Valuation) *A propositional valuation is a function*

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

that maps the propositional variables $p \in \mathcal{P}$ to truth values $\mathcal{I}(p) \in \mathbb{B}$. $\diamond$

A propositional valuation $\mathcal{I}$ maps only the propositional variables to truth values. In order to map propositional formulas to truth values we need to interpret the propositional operators "$\neg$", "$\wedge$", "$\vee$", "$\rightarrow$", and "$\leftrightarrow$" as functions on the set $\mathcal{B}$. To this end we define the functions $\textcircled{$\neg$}$, $\textcircled{$\wedge$}$, $\textcircled{$\vee$}$, $\textcircled{$\rightarrow$}$, and $\textcircled{$\leftrightarrow$}$. These functions have the following signatures:

1. $\textcircled{$\neg$} : \mathbb{B} \rightarrow \mathbb{B}$
2. $\textcircled{$\wedge$} : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$
3. $\textcircled{$\vee$} : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$
4. $\textcircled{$\rightarrow$} : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$
5. $\textcircled{$\leftrightarrow$} : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

We will use these functions as the valuations of the propositional operators. It is easiest to define the functions $\textcircled{$\neg$}$, $\textcircled{$\wedge$}$, $\textcircled{$\vee$}$, $\textcircled{$\rightarrow$}$, and $\textcircled{$\leftrightarrow$}$ via the following truth table (Table 4.1):

p	q	$\textcircled{$\neg$}(p)$	$\textcircled{$\vee$}(p,q)$	$\textcircled{$\wedge$}(p,q)$	$\textcircled{$\rightarrow$}(p,q)$	$\textcircled{$\leftrightarrow$}(p,q)$
`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 of the propositional operators.

Then the truth value of a propositional formula f under a given propositional valuation $\mathcal{I}$ is defined via induction of f. We will denote the truth value as $\widehat{\mathcal{I}}(f)$. We have

1. $\widehat{\mathcal{I}}(\bot) := \texttt{False}$.
2. $\widehat{\mathcal{I}}(\top) := \texttt{True}$.
3. $\widehat{\mathcal{I}}(p) := \mathcal{I}(p)$ for all $p \in \mathcal{P}$.
4. $\widehat{\mathcal{I}}(\neg f) := \ominus\big(\widehat{\mathcal{I}}(f)\big)$ for all $f \in \mathcal{F}$.
5. $\widehat{\mathcal{I}}(f \wedge g) := \owedge\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ for all $f, g \in \mathcal{F}$.
6. $\widehat{\mathcal{I}}(f \vee g) := \ovee\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ for all $f, g \in \mathcal{F}$.
7. $\widehat{\mathcal{I}}(f \rightarrow g) := \ominus\!\!\rightarrow\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ for all $f, g \in \mathcal{F}$.
8. $\widehat{\mathcal{I}}(f \leftrightarrow g) := \ominus\!\!\leftrightarrow\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ for all $f, g \in \mathcal{F}$.

In order to simplify the notation we will not distinguish between the function

$\widehat{\mathcal{I}} : \mathcal{F} \to \mathbb{B}$

that is defined on all propositional formulas and the function

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

Hence, from here on we will write $\mathcal{I}(f)$ instead of $\widehat{\mathcal{I}}(f)$.

Example: We show how to compute the truth value of the formula

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

for the propositional valuation

$\mathcal{I} := \{p \mapsto \texttt{True}, q \mapsto \texttt{False}\}$.

$$\begin{aligned}
\mathcal{I}\Big((p \rightarrow q) \rightarrow (\neg p \rightarrow q) \rightarrow q\Big) &= \ominus\!\!\rightarrow\Big(\mathcal{I}\big((p \rightarrow q)\big), \mathcal{I}\big((\neg p \rightarrow q) \rightarrow q\big)\Big) \\
&= \ominus\!\!\rightarrow\Big(\ominus\!\!\rightarrow\big(\mathcal{I}(p), \mathcal{I}(q)\big), \mathcal{I}\big((\neg p \rightarrow q) \rightarrow q\big)\Big) \\
&= \ominus\!\!\rightarrow\Big(\ominus\!\!\rightarrow\big(\texttt{True}, \texttt{False}\big), \mathcal{I}\big((\neg p \rightarrow q) \rightarrow q\big)\Big) \\
&= \ominus\!\!\rightarrow\Big(\texttt{False}, \mathcal{I}\big((\neg p \rightarrow q) \rightarrow q\big)\Big) \\
&= \texttt{True}
\end{aligned}$$ ◇

Note that we did just evaluate some parts of the formula. The reason is that as soon as we know that the first argument of $\ominus\!\!\rightarrow$ is `False` the value of the corresponding formula is already determined. Nevertheless, this approach is too cumbersome. In practice, we do not evaluate large propositional formulas ourselves. Instead, we will next implement a *Python* program that evaluates these formulas for us.

4.3.3 Implementation

In this section we develop a *Python* program that can evaluate propositional formulas. Every time when we develop a program to compute something useful we have to decide which data structures are most appropriate to represent the information that is to be processed by the program. In this case we want to process propositional formulas. Therefore, we have to decide how to represent propositional formulas in *Python*. One obvious possibility would be to use strings. However, this would be a bad choice as it would then be difficult to access the parts of a given formula. It is far more suitable to represent propositional formulas as nested tuples. A nested tuple is a tuple that contains both strings and nested tuples. For example,

```
('∧', ('¬', 'p'), 'q')
```

is a nested tuple that represents the propositional formula $\neg p \wedge q$.

Formally, the representation of propositional formulas is defined by a function

$$rep : \mathcal{F} \to Python$$

that maps a propositional formula f to the corresponding nested tuple $rep(f)$. We define $rep(f)$ inductively by induction on f.

1. $\top$ is represented as the tuple `('⊤',)`.

   This is possible because `'⊤'` is a unicode symbol and *Python* supports the use of unicode symbols in strings. Alternatively, in *Python* the string `'⊤'` can be written as `'\N{up tack}'` since "`up tack`" is the name of the unicode symbol "$\top$" and any unicode symbol that has the name u can be written as `'\N{`u`}'` in *Python*. Therefore, we have

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

2. $\bot$ is represented as the tuple `('⊥',)`.

   The unicode symbol `'⊥'` has the name "`down tack`". Therefore, we have

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

3. Since propositional variables are strings we can represents these variables by themselves:

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

4. If f is a propositional formula, the negation $\neg f$ is represented as a pair where we put the unicode symbol `'¬'` at the first position, while the representation of f is put at the second position. As the name of the unicode symbol `'¬'` is "`not sign`" we have

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

5. If f_1 and f_2 are propositional formulas, we represent $f_1 \wedge f_2$ with the help of the unicode symbol `'∧'`. This symbol has the name "`logical and`". Hence we have

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

6. If f_1 and f_2 are propositional formulas, we represent $f_1 \vee f_2$ with the help of the unicode symbol `'∨'`. This symbol has the name "`logical or`". Hence we have

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

7. If f_1 and f_2 are propositional formulas, we represent $f_1 \to f_2$ with the help of the unicode symbol `'→'`. This symbol has the name "`rightwards arrow`". Hence we have

   $$rep(f \to g) := \bigl(\texttt{'→'}, rep(f), rep(g)\bigr).$$

8. If f_1 and f_2 are propositional formulas, we represent $f_1 \leftrightarrow f_2$ with the help of the unicode symbol `'↔'`. This symbol has the name "`left right arrow`". Hence we have

$$rep(f \leftrightarrow g) := \bigl(\text{'}\leftrightarrow\text{'}, rep(f), rep(g)\bigr).$$

When choosing the representation of a formula in *Python* we have a lot of freedom. We could as well have represented formulas as objects of different classes. A good representation should have the following properties:

1. It should be intuitive, i.e. we do not want to use any obscure encoding.
2. It should be adequate.
   (a) It should be easy to recognize whether a formula is a propositional variable, a negation, a conjunction, etc.
   (b) It should be easy to access the components of a formula.
   (c) Given a formula f, it should be easy to generate the representation of f.
3. It should be memory efficient.

A propositional valuation is a function

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

mapping the set of propositional variables $\mathcal{P}$ into the set of truth values $\mathbb{B} = \{\texttt{True}, \texttt{False}\}$. We represent a propositional valuation $\mathcal{I}$ as the set of all propositional variables that are mapped to `True` by $\mathcal{I}$:

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

This enables us to implement a simple function that evaluates a propositional formula f with a given propositional valuation $\mathcal{I}$. The *Python* function `evaluate` is shown in Figure 4.1 on page 36. The function `evaluate` takes two arguments.

1. The first argument F is a propositional formula that is represented as a nested tuple.
2. The second argument I is a propositional evaluation. This evaluation is represented as a set of propositional variables. Given a propositional variables p, the value of $\mathcal{I}(p)$ is computed by the expression "p `in` I".

Next, we discuss the implementation of the function `evaluate()`.

1. We make use of the `match` statement, which is available since *Python* 3.10. This new control structure is explained in the tutorial "PEP 636: Structural Pattern Matching" available at

   https://peps.python.org/pep-0636/.

2. Line 3 deals with the case that the argument F is a propositional variable. We can recognize this by the fact that F is a string, which we can check with the predefined function `isinstance`.

   In this case we have to check whether the variable p is an element of the set I, because p is interpreted as `True` if and only if $p \in I$.

3. If F is $\top$, then evaluating F always yields `True`.

```
def evaluate(F, I):
    match F:
        case p if isinstance(p, str):
            return p in I
        case ('⊤', ):     return True
        case ('⊥', ):     return False
        case ('¬', G):    return not evaluate(G, I)
        case ('∧', G, H): return     evaluate(G, I) and evaluate(H, I)
        case ('∨', G, H): return     evaluate(G, I) or  evaluate(H, I)
        case ('→', G, H): return not evaluate(G, I) or  evaluate(H, I)
        case ('↔', G, H): return     evaluate(G, I) ==  evaluate(H, I)
```

Figure 4.1: Evaluation of a propositional formula.

4. If F is $\bot$, then evaluating F always yields `False`.
5. If F has the form $\neg G$, we recursively evaluate G given the evaluation I and negate the result.
6. If F has the form $G \wedge H$, we recursively evaluate G and H using I. The results are then combined with the *Python* operator "`and`".
7. If F has the form $G \vee H$, we recursively evaluate G and H using I. The results are then combined with the *Python* operator "`or`".
8. If F has the form $G \rightarrow H$, we recursively evaluate G and H using I. Then we exploit the fact that the two formulas

   $$G \rightarrow H \quad \text{und} \quad \neg G \vee H$$

   are equivalent.
9. If F has the form $G \leftrightarrow H$, we recursively evaluate G and H using I. Then we exploit the fact that the formula $G \leftrightarrow H$ is true if and only if G and H have the same truth value.

4.3.4 An Application

Next, we showcase a playful application of propositional logic. Inspector Watson is called to investigate a burglary at a jewelry store. Three suspects have been detained in the vicinity of the jewelry store. Their names are Aaron, Bernard, and Cain. The evaluation of the files reveals the following facts.

1. At least one of these suspects must have been involved in the crime.

   If the propositional variable a is interpreted as claiming that Aaron is guilty, while b and c stand for the guilt of Bernard and Cain respectively, then this statement is captured by the following formula:

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

2. If Aaron is guilty, then he has exactly one accomplice.

   To formalize this statement, we decompose it into two statements.

   (a) If Aaron is guilty, then he has at least one accomplice.

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

   (b) If Aaron is guilty, then he has at most one accomplice.

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

3. If Bernard is innocent, then Cain is innocent too.

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

4. If exactly two of the suspects are guilty, then Cain is one of them.

   It is not straightforward to translate this statement into a propositional formula. One trick we can try is to negate the statement and then try to translate the negation. Now the statement given above is wrong if Cain is innocent but both Aaron and Bernard are guilty. Therefore we can translate this statement as follows:

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

5. If Cain is innocent, then Aaron is guilty.

   This translates into the following formula:

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

We now have a set $F = \{f_1, f_2, f_3, f_4, f_5, f_6\}$ of propositional formulas. The question then is to find all propositional valuations $\mathcal{I}$ that evaluate all formulas from F as `True`. If there is exactly one such propositional valuation, then this valuation gives us the culprits. As it is too time consuming to try all possible valuations by hand we will write a program that performs the required computations. Figure 4.2 shows the program `02-Usual-Suspects.ipynb`. We discuss this program next.

1. We input propositional formulas as strings. However, the function `evaluate` needs nested tuples as input. Therefore, we first import our parser for propositional formulas.

2. Next, we define the function `transform`. This function takes a propositional formula that is represented as a string and transforms it into a nested tuple.

3. Line 7 defines the set P of propositional variables. We use the propositional variable `a` to express that Aaron is guilty, `b` is short for Bernard is guilty and `c` is true if and only if Cain is guilty.

4. Next, we define the propositional formulas $f_1, \cdots, f_6$.

5. `Fs` is the set of all propositional formulas.

6. In line 20 these formulas are transformed into nested tuples.

7. The function `allTrue`(Fs, I) takes two inputs.

   (a) `Fs` is a set of propositional formulas that are represented as nested tuples.

   (b) `I` is a propositional evaluation that is represented as a set of propositional variables. Hence `I` is a subset of `P`

```
1  %run Propositional-Logic-Parser.ipynb
2
3  def parse(s):
4      parser = LogicParser(s)
5      return parser.parse()
6
7  P = { 'a', 'b', 'c' }
8  # Aaron, Bernard, or Cain 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 Cain is innocent, too.
14 f4 = '¬b → ¬c'
15 # If exactly two of the suspects are guilty, then Cain is one of them.
16 f5 = '¬(¬c ∧ a ∧ b)'
17 # If Cain is innocent, then Aaron is guilty.
18 f6 = '¬c → a'
19 Fs = { f1, f2, f3, f4, f5, f6 };
20 Fs = { parse(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: A program to investigate the burglary.

If all propositional formulas f from the set `Fs` evaluate as `True` given the evaluation `I`, then `allTrue` returns the result `True`, otherwise `False` is returned.

8. Line 25 computes the set of all propositional variables that render all formulas from `Fs` true. The function `power` takes a set M and returns the power set of M, i.e. it returns the set 2^M.

When we run this program we see that there is just a single propositional valuation I such that all formulas from `Fs` are rendered `True` under I. This propositional valuation has the form

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

Thus, the given problem is solvable and both Bernard and Cain are guilty, while Aaron is innocent.

Exercise 6: Solve the following puzzle by editing the notebook Python/Chapter-4/The-Visit.ipynb.

A portion of the Smith family is visiting the Walton family. **John** Smith and his wife **Helen** have three children. Their oldest child is a boy named **Thomas**. Thomas has two younger sisters named **Amy** and **Jennifer**. Jennifer is the youngest child. The following facts are given:

1. If John is going, he will take his wife Helen along.
2. At least one of the two older children will visit the Waltons.
3. Either Helen or Jennifer will visit the Waltons.
4. Either both daughters will visit the Waltons together or neither of them will.
5. If Thomas visits the Waltons, both John and Amy will also visit.

What part of the Smith family will visit the Walton family?

4.4 Tautologies

Using the program to evaluate a propositional formula we can see that the formula

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

is true for every propositional valuation $\mathcal{I}$. This property gives rise to a definition.

Definition 8 (Tautology) *If f is a propositional formula and we have*

$\mathcal{I}(f) = \texttt{True}$ *for every propositional valuation $\mathcal{I}$,*

then f is a tautology. This is written as

$$\models f.$$ ◇

If f is a tautology, then we say that f is universally valid .

Examples:

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

One way to prove that a formula F is universally valid is to evaluate it for every possible valuation. However, if there are n propositional variables occurring in f, then there are 2^n different possible valuations. Hence for values of n that are greater than fourty this method is hopelessly inefficient. Therefore, our goal in the rest of this chapter is to develop a method that can often deal with hundreds of propositional variables.

Definition 9 (Equivalent) *Two formulas f and g are equivalent if and only if*

$$\models f \leftrightarrow g. \qquad \diamond$$

Examples: We have the following equivalences:

$\models \neg\bot \leftrightarrow \top$	$\models \neg\top \leftrightarrow \bot$	
$\models p \vee \neg p \leftrightarrow \top$	$\models p \wedge \neg p \leftrightarrow \bot$	tertium-non-datur
$\models p \vee \bot \leftrightarrow p$	$\models p \wedge \top \leftrightarrow p$	identity element
$\models p \vee \top \leftrightarrow \top$	$\models p \wedge \bot \leftrightarrow \bot$	
$\models p \wedge p \leftrightarrow p$	$\models p \vee p \leftrightarrow p$	idempotent
$\models p \wedge q \leftrightarrow q \wedge p$	$\models p \vee q \leftrightarrow q \vee p$	commutative
$\models (p \wedge q) \wedge r \leftrightarrow p \wedge (q \wedge r)$	$\models (p \vee q) \vee r \leftrightarrow p \vee (q \vee r)$	Assoziativität
$\models \neg\neg p \leftrightarrow p$		elimination of $\neg\neg$
$\models p \wedge (p \vee q) \leftrightarrow p$	$\models p \vee (p \wedge q) \leftrightarrow p$	absorption
$\models p \wedge (q \vee r) \leftrightarrow (p \wedge q) \vee (p \wedge r)$	$\models p \vee (q \wedge r) \leftrightarrow (p \vee q) \wedge (p \vee r)$	distributive
$\models \neg(p \wedge q) \leftrightarrow \neg p \vee \neg q$	$\models \neg(p \vee q) \leftrightarrow \neg p \wedge \neg q$	DeMorgan's rules
$\models (p \rightarrow q) \leftrightarrow \neg p \vee q$		elimination of $\rightarrow$
$\models (p \leftrightarrow q) \leftrightarrow (\neg p \vee q) \wedge (\neg q \vee p)$		elimination of $\leftrightarrow$

Notation: If the formulas f and g are equivalent, we can write this as

$$\models f \leftrightarrow g.$$

However, since this notation is rather clumsy, we will denote this fact as $f \Leftrightarrow g$ instead. Furthermore, in this context we use *chaining* for the operator "$\Leftrightarrow$", that is we write

$$f \Leftrightarrow g \Leftrightarrow h$$

to denote the fact that we have both

$$\models f \leftrightarrow g \quad \text{and} \quad \models g \leftrightarrow h. \qquad \diamond$$

4.4.1 *Python* Implementation

In this section we develop a *Python* program that is able to decide whether a given propositional formula f is a tautology. The idea is that the program evaluates f for all possible propositional interpretations. Hence we have to compute the set of all propositional interpretations for a given set of propositional variables. We have already seen that the propositional interpretations are in a 1-to-1 correspondence with the subsets of the set $\mathcal{P}$ of all propositional variables because we can represent a propositional interpretation $\mathcal{I}$ as the set of all propositional variables that evaluate as `True`:

$$\{q \in \mathcal{P} \mid \mathcal{I}(q) = \texttt{True}\}.$$

If we have a propositional formula f and want to check whether f is a tautology, we first have to determine the set of propositional variables occurring in f. To this end we define a function

$$\texttt{collectVars} : \mathcal{F} \rightarrow 2^{\mathcal{P}}$$

such that collectVars(f) is the set of propositional variables occurring in f. This function can be defined recursively.

1. collectVars$(p) = \{p\}$ for all propositional variables p.
2. collectVars$(\top) = \{\}$.
3. collectVars$(\bot) = \{\}$.
4. collectVars$(\neg f) :=$ collectVars(f).
5. collectVars$(f \wedge g) :=$ collectVars$(f) \cup$ collectVars(g).
6. collectVars$(f \vee g) :=$ collectVars$(f) \cup$ collectVars(g).
7. collectVars$(f \rightarrow g) :=$ collectVars$(f) \cup$ collectVars(g).
8. collectVars$(f \leftrightarrow g) :=$ collectVars$(f) \cup$ collectVars(g).

Figure 4.3 on page 41 shows how to implement this definition. Note that we have been able to combine the last four cases.

```
def collectVars(f):
    "Collect all propositional variables occurring in the formula f."
    match f:
        case p if isinstance(p, str): return { p }
        case ('⊤', ):   return set()
        case ('⊥', ):   return set()
        case ('¬', g):  return collectVars(g)
        case (_, g, h): return collectVars(g) | collectVars(h)
```

Figure 4.3: Überprüfung der Allgemeingültigkeit einer aussagenlogischen Formel

Now we are able to implement the function

$$\texttt{tautology} : \mathcal{F} \rightarrow \mathbb{B}$$

that takes a formula f and returns True if and only if $\models f$ holds. Figure 4.4 on page 42 shows the implementation.

1. First, we compute the set P of all propositional variables occurring in f.
2. Next, the function allSubsets computes a list containing all subsets of the set P. Every propositional interpretation $\mathcal{I}$ is contained in this list.
3. Then we try to find a propositional interpretation $\mathcal{I}$ such that $\mathcal{I}(f)$ False. In this case $\mathcal{I}$ is returned.
4. Otherwise f is a tautology and we return True.

```
def tautology(f):
    "Check, whether the formula f is a tautology."
    P = collectVars(f)
    for I in power.allSubsets(P):
        if not evaluate(f, I):
            return I
    return True
```

Figure 4.4: Checking that f is a tautology.

4.5 Conjunctive Normal Form

The following section discusses algebraic manipulations of propositional formulas. Concretely, we will define the notion of a conjunctive normal form and show how a propositional formula can be turned into conjunctive normal form. The Davis-Putnam algorithm that is discussed later requires us to put the given formulas into conjunctive normal form.

Definition 10 (Literal) *A propositional formula f is a literal if and only if we have one of the following cases:*

1. $f = \top$ or $f = \bot$.
2. *$f = p$, where p is a propositional variable.*

   In this case f is a positive literal.
3. *$f = \neg p$, where p is a propositional variable.*

   In this case f is a negative literal.

The set of all literals is denoted as $\mathcal{L}$. $\diamond$

If l is a literal, then the complement l of is denoted as $\overline{l}$. It is defined by a case distinction.

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

The complement $\overline{l}$ of a literal l is equivalent to the negation of l, we have

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

However, the complement of a literal l is also a literal, while the negation of a literal is in general not a literal. For example, if p is a propositional variable, then the complement of $\neg p$ is p, while the negation of $\neg p$ is $\neg\neg p$ and this is not a literal.

Definition 11 (Clause) *A propositional formula K is a clause when it has the form*

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

where l_i is a literal for all $i = 1, \cdots, r$. Hence a clause is a disjunction of literals. The set of all clauses is denoted as $\mathcal{K}$. $\diamond$

Often, a clause is seen as a set of its literals. Interpreting a clause as a set of its literals abstracts from both the order and the number of the occurrences of its literals. This is possible because the operator "$\vee$" is associative, commutative, and idempotent. Hence the clause $l_1 \vee \cdots \vee l_r$ will be written as the set

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

This notation is called the set notation for clauses. The following example shows how the set notation is beneficial. We take the clauses

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

Although these clauses are equivalent, they are not syntactically identical. However, if we transform these clauses into set notation, we get

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

In a set every element occurs at most once. Furthermore, the order of the elements does not matter. Hence the sets given above are the same!

Let us now transfer the propositional equivalence

$$l_1 \vee \cdots \vee l_r \vee \bot \;\Leftrightarrow\; l_1 \vee \cdots \vee l_r$$

into set notation. We get

$$\{l_1, \cdots, l_r, \bot\} \;\Leftrightarrow\; \{l_1, \cdots, l_r\}.$$

This shows, that the element $\bot$ can be discarded from a clause. If we write this equivalence for $r = 0$, we get

$$\{\bot\} \Leftrightarrow \{\}.$$

Hence the empty set of literals is to be interpreted as $\bot$.

Definition 12 *A clause K is trivial, if and only if one of the following cases occurs:*

1. *$\top \in K$.*
2. *There is a variable $p \in \mathcal{P}$, such that we have both $p \in K$ as well as $\neg p \in K$.*

   In this case p and $\neg p$ are called complementary literals. $\diamond$

Proposition 13 *A clause K is a tautology if and only if it is trivial.*

Proof: We first assume that the clause K is trivial. If $\top \in K$, then because we have $f \vee \top \Leftrightarrow \top$ obviously $K \Leftrightarrow \top$. If p is a propositional variable, so that both $p \in K$ and $\neg p \in K$ are valid, then due to the equivalence $p \vee \neg p \Leftrightarrow \top$ we immediately have $K \Leftrightarrow \top$.

Next we assume that the clause K is a tautology. We carry out the proof indirectly and assume that K is non-trivial. This means that $\top \notin K$ and K cannot contain any complementary literals. Thus

K must have the form

$$K = \{\neg p_1, \cdots, \neg p_m, q_1, \cdots, q_n\} \quad \text{with } p_i \neq q_j \text{ for all } i \in \{1, \cdots, m\} \text{ and } j \in \{1, \cdots, n\}.$$

Then we can define an propositional valuation $\mathcal{I}$ as follows:

1. $\mathcal{I}(p_i) = \texttt{True}$ for all $i = 1, \cdots, m$ and
2. $\mathcal{I}(q_j) = \texttt{False}$ for all $j = 1, \cdots, n$,

We have $\mathcal{I}(K) = \texttt{False}$ with this valuation and therefore K isn't a tautology. So the assumption that K is not trivial is false. □

Definition 14 (Conjunctive Normal Form) *A formula F is in conjunctive normal form (short CNF) if and only if F is a conjunction of clauses, i.e. if*

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

where the K_i are clauses for all $i = 1, \cdots, n$. ◇

The following is an immediately corollary of the definition of a CNF.

Corollary 15 *If $F = K_1 \wedge \cdots \wedge K_n$ is in conjunctive normal form, then*

$$\models F \quad \text{if and only if} \quad \models K_i \quad \text{for all } i = 1, \cdots, n.$$ □

Thus, for a formula $F = K_1 \wedge \cdots \wedge K_n$ in conjunctive normal form, we can easily decide whether F is a tautology, because F is a tautology iff all clauses K_i are trivial.

Since the associative, commutative and idempotent law applies to a conjunction in the same way as to a disjunction it is beneficial to also use a set notation: If the formula

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

is in conjunctive normal form, we represent this formula by the set of its clauses and write

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

The clauses themselves are also specified in set notation. We provide an example: If p, q and r are propositional variables, the formula

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

is in conjunctive normal form. In set notation, this becomes

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

We now present a method that can be used to transform any formula F into CNF. As we have seen above, we can then easily decide whether F is a tautology.

1. Eliminate all occurrences of the junction "$\leftrightarrow$" using the equivalence

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

2. Eliminate all occurrences of the junction "$\rightarrow$" using the equivalence

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

3. Move the negation signs inwards as far as possible. Use the following equivalences:
   (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 the result that we obtain after this step, the negation signs occurs only immediately before propositional variables. Formulas with this property are also referred to as formulas in negation-normal-form.

4. If the formula contains "$\vee$" junctors on top of "$\wedge$" junctors, we use the distributive law
$$\begin{array}{rl} & (F_1 \wedge \cdots \wedge F_m) \vee (G_1 \wedge \cdots \wedge G_n) \\ \Leftrightarrow & (F_1 \vee G_1) \wedge \cdots \wedge (F_1 \vee G_n) \;\wedge\; \cdots \;\wedge\; (F_m \vee G_1) \wedge \cdots \wedge (F_m \vee G_n) \end{array}$$
   to move the disjunction "$\vee$" inwards.

5. In the final step we convert the formula into set notation.

We should note that the formula can grow considerably when using the distributive law. This is because the formula F occurs twice on the right-hand side of the equivalence $F \vee (G \wedge H) \leftrightarrow (F \vee G) \wedge (F \vee H)$, while it occurs only occurs once on the left.

We demonstrate the procedure using the example of the formula

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

1. Since the formula does not contain the operator "$\leftrightarrow$" there is nothing to do in the first step.
2. The elimination of the operator "$\rightarrow$" yields
$$\neg(\neg p \vee q) \vee (\neg\neg p \vee \neg q).$$
3. The conversion to negation normal form results in
$$(p \wedge \neg q) \vee (p \vee \neg q).$$
4. By using the distributive law we get
$$\big(p \vee (p \vee \neg q)\big) \wedge \big(\neg q \vee (p \vee \neg q)\big).$$
5. The conversion to set notation yields the following clauses
$$\{p, p, \neg q\} \quad \text{and} \quad \{\neg q, p, \neg q\}.$$
   However, since the order of the elements of a set is unimportant and, moreover, a set contains each element only once, we realize that these two clauses are the same. Therefore, if we combine these clauses into a set, we get
$$\big\{\{p, \neg q\}\big\}.$$
   Note that the formula is significantly simplified by converting it into set notation.

The formula is now converted to CNF.

4.5.1 Computing the Conjunctive Normal Form in *Python*

Next, we specify anumber of functions that can be used to convert a given formula formula f into conjunctive normal form. These functions are part of the Jupyter notebook

Python/Chapter-4/04-CNF.ipynb.

We start with the function

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

which has the task of transforming a given propositional formula f into an equivalent formula which no longer contains the operator "$\leftrightarrow$". The function $\texttt{elimBiconditional}(f)$ is defined by induction with respect to the propositional formula f. In order to present this inductive defined, we set up recursive equations that describe the behavior of the function `elimBiconditional`:

1. If f is a propositional variable p, there is nothing to do:
   $$\texttt{elimBiconditional}(p) = p \quad \text{for all } p \in \mathcal{P}.$$
2. The cases in which f is equal to Verum or Falsum are also trivial:
   $$\texttt{elimBiconditional}(\top) = \top \quad \text{and} \quad \texttt{elimBiconditional}(\bot) = \bot.$$
3. If f has the form $f = \neg g$, we eliminate the operator "$\leftrightarrow$" recursively from the formula g and negate the resulting formula:
   $$\texttt{elimBiconditional}(\neg g) = \neg\texttt{elimBiconditional}(g).$$
4. In the cases $f = g_1 \wedge g_2$, $f = g_1 \vee g_2$ and $f = g_1 \to g_2$ we recursively eliminate the operator "$\leftrightarrow$" from the formulas g_1 and g_2 and then reassemble the formula together again:
   (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. If f has the form $f = g_1 \leftrightarrow g_2$, we use the equivalence
   $$(g_1 \Leftrightarrow g_2) \leftrightarrow \big((g_1 \to g_2) \wedge (g_2 \to g_1)\big).$$
   This leads to the equation:
   $$\texttt{elimBiconditional}(g_1 \leftrightarrow g_2) = \texttt{elimBiconditional}\big((g_1 \to g_2) \wedge (g_2 \to g_1)\big).$$
   It is necessary to call the function `elimBiconditional` on the right-hand side of the equation, because the operator "$\leftrightarrow$" can still occur in g_1 and g_2.

Figure 4.5 on page 47 shows the implementation of the function `elimBiconditional`.

1. In line 3, the function call $\texttt{isinstance}(p, \texttt{str})$ checks whether p is a string. In this case f must be a propositional variable, because all other propositional formulas are represented as nested lists. Therefore, f is returned unchanged in this case.
2. In line 5 we check the case that f has the form $g \leftrightarrow h$. In this case, we use the equivalence
   $$(g \leftrightarrow h) \Leftrightarrow (g \to h) \wedge (h \to g).$$

```
1  def eliminateBiconditional(f):
2      match f:
3          case p if isinstance(p, str):   # This case covers variables.
4              return p
5          case ('↔', g, h):
6              return eliminateBiconditional( ('∧', ('→', g, h), ('→', h, g)) )
7          case ('⊤', ) | ('⊥', ):
8              return f
9          case ('¬', g):
10             return ('¬', eliminateBiconditional(g))
11         case (op, g, h):   # This case covers '→', '∧', and '∨'.
12             return (op, eliminateBiconditional(g), eliminateBiconditional(h))
```

Figure 4.5: Elimination of $\leftrightarrow$

Furthermore, we have to eliminate the operator "$\leftrightarrow$" from g and h by recursively calling the function function `elimBiconditional`.

3. In line 5 we deal with the cases where f is equal to Verum or Falsum. Note that these formulas are also represented as nested tuples. For example, verum is represented as the tuple `('⊤',)`, while falsum is represented in *Python* by the tuple `('⊥',)`. In this case, f is returned unchanged.

4. In line 9, we consider the case where f is a negation. Then f has the form

   `('¬',` g`)`

   and we have to recursively remove the operator "$\leftrightarrow$" from g.

5. In the remaining cases, f has the form

   `('`o`',` g`,` h`)` with $o \in \{\rightarrow, \wedge, \vee\}$.

   In these cases, the operator "$\leftrightarrow$" has to be removed recursively from the subformulas g and h.

Next, we look at the function for eliminating the "$\rightarrow$" operator. Figure 4.6 on page 48 shows the implementation of the function `eliminateConditional`. The underlying idea of the implementation is the same as for the elimination of the operator "$\leftrightarrow$". The only difference is that we now use the equivalence

$$(g \rightarrow h) \Leftrightarrow (\neg g \vee h).$$

Furthermore, when implementing this function, we can assume that the operator "$\leftrightarrow$" has already been eliminated from the propositional formula `f`, which is passed as an argument. This eliminates one case in the implementation.

```
def eliminateConditional(f: Formula) -> Formula:
    'Eliminate the logical operator "→" from f.'
    match f:
        case p if isinstance(p, str):
            return p
        case ('⊤', ) | ('⊥', ):
            return f
        case ('→', g, h):
            return eliminateConditional(('∨', ('¬', g), h))
        case ('¬', g):
            return ('¬', eliminateConditional(g))
        case (op, g, h):      # This case covers '∧' and '∨'.
            return (op, eliminateConditional(g), eliminateConditional(h))
```

Figure 4.6: Elimination of $\rightarrow$

Next, we present the functions for calculating the negation normal form. Figure 4.7 on page 49 shows the implementation of the functions `nnf` and `neg`, which call each other. Here, $\texttt{nnf}(f)$ calculates the negation normal form of f, while $\texttt{neg}(f)$ calculates the negation normal form of $\neg f$, so the following holds

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

The actual work is done in the function `neg`, because this is where DeMorgan's laws

$$\neg(f \wedge g) \Leftrightarrow (\neg f \vee \neg g) \quad \text{and} \quad \neg(f \vee g) \Leftrightarrow (\neg f \wedge \neg g)$$

are applied. We describe the transformation into negation normal form by the following equations:

1. $\texttt{nnf}(p) = p$ 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. $\texttt{nnf}(f_1 \vee f_2) = \texttt{nnf}(f_1) \vee \texttt{nnf}(f_2)$.

The auxiliary procedure `neg`, which calculates the negation normal form of $\neg f$, is also specified by recursive equations:

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. $\texttt{neg}(\bot) = \texttt{nnf}(\neg\bot) = \texttt{nnf}(\top) = \top$,

```
def nnf(f: Formula) -> Formula:
    match f:
        case p if isinstance(p, str): return p
        case ('⊤', ) | ('⊥', ):      return f
        case ('¬', g):               return neg(g)
        case (op, g, h):             return (op, nnf(g), nnf(h))

def neg(f: Formula) -> Formula:
    match f:
        case p if isinstance(p, str): return ('¬', p)
        case ('⊤', ):     return ('⊥', )
        case ('⊥', ):     return ('⊤', )
        case ('¬', g):    return nnf(g)
        case ('∧', g, h): return ('∨', neg(g), neg(h))
        case ('∨', g, h): return ('∧', neg(g), neg(h))
```

Figure 4.7: Computing the negation normal form.

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

5. $$\begin{aligned} & \texttt{neg}\big(f_1 \wedge f_2\big) \\ = \; & \texttt{nnf}\big(\neg(f_1 \wedge f_2)\big) \\ = \; & \texttt{nnf}\big(\neg f_1 \vee \neg f_2\big) \\ = \; & \texttt{nnf}\big(\neg f_1\big) \vee \texttt{nnf}\big(\neg f_2\big) \\ = \; & \texttt{neg}(f_1) \vee \texttt{neg}(f_2). \end{aligned}$$

   Hence we have:
   $$\texttt{neg}\big(f_1 \wedge f_2\big) = \texttt{neg}(f_1) \vee \texttt{neg}(f_2).$$

6. $$\begin{aligned} & \texttt{neg}\big(f_1 \vee f_2\big) \\ = \; & \texttt{nnf}\big(\neg(f_1 \vee f_2)\big) \\ = \; & \texttt{nnf}\big(\neg f_1 \wedge \neg f_2\big) \\ = \; & \texttt{nnf}\big(\neg f_1\big) \wedge \texttt{nnf}\big(\neg f_2\big) \\ = \; & \texttt{neg}(f_1) \wedge \texttt{neg}(f_2). \end{aligned}$$

   Therefore we have:
   $$\texttt{neg}\big(f_1 \vee f_2\big) = \texttt{neg}(f_1) \wedge \texttt{neg}(f_2).$$

Finally, we present the function `cnf` that allows us to transform a propositional formula f from negation normal form into conjunctive normal form. Furthermore, `cnf` converts the resulting formula into set notation, i.e. the formulas are represented as sets of sets of literals. of literals. We interpret a set of literals as a disjunction of the literals and a set of clauses is interpreted as a conjunction of the clauses. Mathematically, our goal is therefore to find a function

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

so that $\texttt{cnf}(f)$ for a formula f, which is in negation normal form, returns a set of clauses as a result whose conjunction is equivalent to f. The definition of $\texttt{cnf}(f)$ is given recursively.

1. If f is a propositional variable, we return as result a set that contains exactly one clause. This clause is itself a set of literals, the only literal of which is the propositional variable f:

   $$\texttt{cnf}(f) := \bigl\{\{f\}\bigr\} \quad \text{if } f \in \mathcal{P}.$$

2. We saw earlier that the empty set of *clauses* can be interpreted as $\top$. Therefore:

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

3. We had also seen that the empty set of *literals* can be interpreted as $\bot$. Therefore:

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

4. If f is a negation, then the following holds:

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

   because f is in negation normal form and in such a formula the negation operator can only be applied to a propositional variable. Therefore, f is a literal and we return as a result a set that contains exactly one clause. This clause is itself a set of literals, which contains the formula f as the only literal:

   $$\texttt{cnf}(\neg p) := \bigl\{\{\neg p\}\bigr\} \quad \text{if } p \in \mathcal{P}.$$

5. If f is a conjunction and therefore $f = g \wedge h$, then we first transform the formulas g and h into CNF. We then obtain sets of clauses $\texttt{cnf}(g)$ and $\texttt{cnf}(h)$. Since we interpret a set of clauses as a conjunction of the clauses contained in the set, it is sufficient to form the union of the sets $\texttt{cnf}(f)$ and $\texttt{cnf}(g)$, so we have

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

6. If $f = g \vee h$, we first transform g and h into CNF. This gives us

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

   The formulas g_i and the h_j are clauses. In order to form the CNF of $g \vee h$ we proceed as follows:

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

   If we take into account that clauses in the set notation are understood as sets of literals that are implicitly linked disjunctively, then we can also write $k_i \vee l_j$ as the union $k_i \cup l_j$. Therefore, we obtain

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

Figure 4.8 on page 51 shows the implementation of the function `cnf`. (The name `cnf` is the abbreviation of conjunctive normal form).

```
def cnf(f: Formula) -> CNF:
    match f:
        case p if isinstance(p, str):
            return { frozenset({p}) }
        case ('⊤', ):
            return set()
        case ('⊥', ):
            return { frozenset() }
        case ('¬', p):
            return { frozenset({ ('¬', p) }) }
        case ('∧', g, h):
            return cnf(g) | cnf(h)
        case ('∨', g, h):
            return { k1 | k2 for k1 in cnf(g) for k2 in cnf(h) }
```

Figure 4.8: Berechnung der konjunktiven Normalform.

Lastly, we show in figure 4.9 on page 52 how the functions that have been shown above interact.

1. The function `normalize` first eliminates the operators "$\leftrightarrow$" with the help of the function `eliminateBiconditional`.
2. The operator "$\rightarrow$" is then replaced with the help of the function `eliminateConditional`.
3. Calling `nnf` converts the formula to negation-normal form.
4. The negation normal form is now converted to conjunctive normal form using the function `cnf`, whereby the formula is simultaneously converted to set notation.
5. Finally, the function `simplify` removes all clauses from the set `N4` that are trivial.
6. The function `isTrivial` checks whether a clause C, which is in set notation, contains both a variable p and the negation $\neg p$ of this variable, because then this clause is equivalent to $\top$ and can be omitted.

The complete program for calculating the conjunctive normal form can be found as the file

`Python/Chapter-4/04-CNF.ipynb`

on my GitHub repository.

Exercise 7: Calculate the conjunctive normal forms of the following formulae and state your result in set notation.

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

```
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 4.9: Normalizing a Formula.

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

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

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

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

4.6 The Concept of a Derivation

If $\{f_1, \cdots, f_n\}$ is a set of formulas and g is another formula, then we might ask whether the formula g is a logical consequence of the formulas $f_1, \cdots, f_n$, i.e. whether

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

holds. There are various ways to answer this question. We already know one method: First, we transform the formula $f_1 \wedge \cdots \wedge f_n \rightarrow g$ into conjunctive normal form. Thereby we obtain a set of clauses $\{k_1, \cdots, k_m\}$, whose conjunction is equivalent to the formula

$$f_1 \wedge \cdots \wedge f_n \rightarrow g$$

This formula is a tautology if and only if each of the clauses $k_1, \cdots, k_m$ is trivial.

However, the above method is quite inefficient. We demonstrate this with an example and apply the method to decide whether $p \rightarrow r$ follows from the two formulas $p \rightarrow q$ and $q \rightarrow r$. We compute the conjunctive normal form of the formula

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

and after a laborious calculation, we obtain

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

Although we can now see that the formula h is a tautology, given the fact that we can see with the naked eye that $p \rightarrow r$ follows from the formulas $p \rightarrow q$ and $q \rightarrow r$, this calculation is far too inefficient.

Therefore, we now present another method that can help us decide whether a formula follows from a given set of formulas. The idea of this method is to derive the formula to be proved using inference rules from the given formulas. The concept of an inference rule is based on the following definition.

Definition 16 (Inference Rule)
A propositional inference rule is a pair of the form $\langle\langle f_1, f_2\rangle, k\rangle$. Here, $\langle f_1, f_2\rangle$ is a pair of propositional formulas, and k is a single propositional formula. The formulas f_1 and f_2 are referred to as premises, and the formula k is called the conclusion of the inference rule. If the pair $\langle\langle f_1, f_2\rangle, k\rangle$ is an inference rule, then this is written as:

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

We read this inference rule as: "From f_1 and f_2 we conclude k." ⋄

Examples of inference rules:

Modus Ponens	Modus Tollens	Nonsense
$\dfrac{f \qquad f \rightarrow g}{g}$	$\dfrac{\neg g \qquad f \rightarrow g}{\neg f}$	$\dfrac{\neg f \qquad f \rightarrow g}{\neg g}$

Initially, the definition of the notion of an inference rule does not limit the formulas that can be used as premises or conclusion. However, it is certainly not useful to allow arbitrary inference rules. If we want to use inference rules in proofs, they should be correct in the sense explained in the following definition.

Definition 17 (Correct Inference Rule) *An inference rule of the form*

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

is correct if and only if

$$\models f_1 \wedge f_2 \rightarrow k.$$ ⋄

With this definition, we see that the rules of inference referred to above as "Modus Ponens" and "Modus Tollens" are correct, while the one called "Nonsense" is not correct.

From now on, we assume that all formulas are clauses. On one hand, this is not a real restriction, as we can transform any formula into an equivalent set of clauses. On the other hand, the formulas in many practical propositional logic problems are already in the form of clauses. Therefore, we now present an inference rule where both the premises and the conclusion are clauses.

Definition 18 (Cut Rule) *If p is a propositional variable and k_1 and k_2 are sets of literals, which we interpret as clauses, then we refer to the following inference rule as the cut rule:*

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

The cut rule is very general. If we set $k_1 = \{\}$ and $k_2 = \{q\}$ in the definition above, we obtain the following rule as a special case:

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

Interpreting the sets of literals as disjunctions, we have:

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

Taking into account that the formula $\neg p \vee q$ is equivalent to the formula $p \rightarrow q$, this instance of the cut rule is none other than Modus Ponens. The rule Modus Tollens is also a special case of the cut rule. We obtain this rule if we set $k_1 = \{\neg q\}$ and $k_2 = \{\}$ in the cut rule.

Theorem 19 *The cut rule is correct.*

Proof: We need to show that

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

holds. To do this, we convert the above formula into conjunctive normal form:

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

Definition 20 (Derivation, $\vdash$)
Let M be a set of clauses and f a single clause. The formulas from M are referred to as our axioms. Our goal is to derive the formula f using the axioms from M. For this, we define the relation

$$M \vdash f.$$

We read "$M \vdash f$" as "M derives f". The inductive definition is as follows:

1. *From a set M of assumptions, any of the assumptions can be derived:*

   If $\quad f \in M, \quad$ then $\quad M \vdash f$.

2. *If $k_1 \cup \{p\}$ and $\{\neg p\} \cup k_2$ are clauses that can be derived from M, then using the cut rule, the clause $k_1 \cup k_2$ can also be derived from M:*

   If both $\quad M \vdash k_1 \cup \{p\} \quad$ and $\quad M \vdash \{\neg p\} \cup k_2 \quad$ hold, then $\quad M \vdash k_1 \cup k_2 \quad$ holds, too. $\diamond$

Example: To illustrate the concept of a derivation, we provide an example and show

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

At the same time, we demonstrate through this example how proofs are presented on paper:

1. From $\{\neg p, q\}$ and $\{\neg q, \neg p\}$ it follows by the cut rule that $\{\neg p, \neg p\}$ holds. Since $\{\neg p, \neg p\} = \{\neg p\}$, we write this as

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

   Remark: This example shows that the clause $k_1 \cup k_2$ might contain fewer elements than the sum $\texttt{card}(k_1) + \texttt{card}(k_2)$. This situation occurs when there are literals that appear in both k_1 and k_2.

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

As another example, we demonstrate that $p \rightarrow r$ follows from $p \rightarrow q$ and $q \rightarrow r$. First, we convert all formulas into clauses:

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

Thus, we have $M = \big\{ \{\neg p, q\},\ \{\neg q, r\} \big\}$ and must show that

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

holds. The proof consists of a single application of the Cut Rule:

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

4.6.1 Properties of the Concept of a Formal Derivation

The relation $\vdash$ has two important properties:

Theorem 21 (Correctness) *If $\{k_1, \cdots, k_n\}$ is a set of clauses and k is a single clause, then:*

$$\textit{If } \{k_1, \cdots, k_n\} \vdash k \textit{ holds, then } \models k_1 \wedge \cdots \wedge k_n \rightarrow k \textit{ also holds.}$$

In other words: If we can prove a clause k using the assumptions $k_1, \cdots, k_n$, then the clause k logically follows from these assumptions.

Proof: The proof of the correctness theorem proceeds by induction on the definition of the relation $\vdash$.

1. Case: It holds that $\{k_1, \cdots, k_n\} \vdash k$ because $k \in \{k_1, \cdots, k_n\}$. Then there exists an $i \in \{1, \cdots, n\}$ such that $k = k_i$. In this case, we need to show

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

   which is obvious.

2. Case: It holds that $\{k_1, \cdots, k_n\} \vdash k$ because there is a propositional variable p and clauses g and h such that
$$\{k_1, \cdots, k_n\} \vdash g \cup \{p\} \quad \text{and} \quad \{k_1, \cdots, k_n\} \vdash h \cup \{\neg p\}$$
hold and from this we have concluded using the cut rule that
$$\{k_1, \cdots, k_n\} \vdash g \cup h$$
where $k = g \cup h$. We have to show that
$$\models k_1 \wedge \cdots \wedge k_n \rightarrow g \vee h$$
holds. Let $\mathcal{I}$ be a propositional interpretation such that
$$\mathcal{I}(k_1 \wedge \cdots \wedge k_n) = \texttt{True}$$
Then we need to show that we have
$$\mathcal{I}(g) = \texttt{True} \quad \text{or} \quad \mathcal{I}(h) = \texttt{True}.$$
By the induction hypothesis, we know
$$\models k_1 \wedge \cdots \wedge k_n \rightarrow g \vee p \quad \text{and} \quad \models k_1 \wedge \cdots \wedge k_n \rightarrow h \vee \neg p.$$
Since $\mathcal{I}(k_1 \wedge \cdots \wedge k_n) = \texttt{True}$, it follows that
$$\mathcal{I}(g \vee p) = \texttt{True} \quad \text{and} \quad \mathcal{I}(h \vee \neg p) = \texttt{True}.$$
Now there are two cases:
   (a) Case: $\mathcal{I}(p) = \texttt{True}$.
   Then $\mathcal{I}(\neg p) = \texttt{False}$ and hence from the fact that $\mathcal{I}(h \vee \neg p) = \texttt{True}$, we must have
   $$\mathcal{I}(h) = \texttt{True}.$$
   This immediately implies
   $$\mathcal{I}(g \vee h) = \texttt{True}. \ \surd$$
   (b) Case: $\mathcal{I}(p) = \texttt{False}$.
   Since $\mathcal{I}(g \vee p) = \texttt{True}$ we must have
   $$\mathcal{I}(g) = \texttt{True}.$$
   Thus, we also have
   $$\mathcal{I}(g \vee h) = \texttt{True}. \ \surd$$ □

The converse of this theorem only holds in a weakened form, specifically when k is the empty clause, which corresponds to Falsum. Therefore, we say that the cut rule is refutation-complete.

4.6.2 Proof of Refutation Completeness

To succinctly state the theorem of refutation completeness in propositional logic, we need the concept of satisfiability, which we introduce next.

Definition 22 (Satisfiability)
Let M be a set of propositional formulas. If there exists a propositional interpretation $\mathcal{I}$ that satisfies

all formulas from M, i.e. we have

$$\mathcal{I}(f) = \texttt{True} \quad \text{for all } f \in M,$$

then we call M satisfiable. Furthermore, we say that M is unsatisfiable and write

$$M \models \bot,$$

if there is no propositional interpretation $\mathcal{I}$ that simultaneously satisfies all formulas from M. Denoting the set of propositional interpretations as ALI*, we formally write*

$$M \models \bot \quad \text{iff} \quad \forall \mathcal{I} \in \text{ALI} : \exists C \in M : \mathcal{I}(C) = \texttt{False}.$$

If a set M of propositional formulas is satisfiable, we also write

$$M \not\models \bot. \qquad \diamond$$

Remark: If $M = \{f_1, \cdots, f_n\}$ is a set of propositional formulas, it is straightforward to show that M is unsatisfiable if and only if

$$\models f_1 \wedge \cdots \wedge f_n \rightarrow \bot$$

holds. $\diamond$

Definition 23 (Saturated Sets of Clauses)
A set M of clauses is saturated if every clause that can be derived from two clauses $C_1, C_2 \in M$ using the cut rule is already itself an element of the set M, i.e. if

$$C_1 \in M, \quad C_2 \in M, \quad \text{and} \quad C_1,\ C_2 \vdash C,$$

then we must also have

$$C \in M.$$

Remark: If M is a finite set of clauses, we can extend M to a saturated set of clauses $\overline{M}$ by initializing $\overline{M}$ as the set M and then continually applying the cut rule to clauses from $\overline{M}$ and adding them to the set $\overline{M}$ as long as this is possible. Since there is only a limited number of clauses that can be formed from a given finite set of propositional variables, this process must terminate, and the set of formulas we then obtain is saturated. $\diamond$

4.6.3 Proof of Refutational Completeness

To succinctly state the theorem of refutational completeness in propositional logic, we need the concept of satisfiability, which we introduce formally.

Satz 24 (Refutation Completeness)
Assume M is a set of clauses that is unsatisfiable. Then M derives the empty clause:

$$\text{If } M \models \bot, \text{ then } M \vdash \{\}.$$

Proof: We prove this by contraposition. Assume that M is a finite set of clauses such that

(a) M is unsatisfiable, thus

$$M \models \bot,$$

(b) but such that the empty clause cannot be derived from M, i.e. we have

$$M \nvdash \{\}.$$

We will show that then there has to exist a propositional interpretation $\mathcal{I}$ under which all clauses from M become true, which contradicts the unsatisfiability of M.

Following the previous remark, we saturate the set M to obtain the saturated set $\overline{M}$. Let

$$\{p_1, \cdots, p_N\}$$

be the set of all propositional variables appearing in clauses from M. We denote the set of propositional variables appearing in a clause C by $\texttt{var}(C)$. For all $k = 0, 1, \cdots, N$ we now define a propositional interpretation $\mathcal{I}_k$ by induction on k. For all $k = 0, 1, \cdots, N$ the propositional interpretations $\mathcal{I}_k$ has the property that for each clause $C \in \overline{M}$, which contains only the variables $p_1, \cdots, p_k$, we have that $\mathcal{I}_k(C) = \texttt{True}$ holds. This is formally written as follows:

$$\forall C \in \overline{M} : \Big(\texttt{var}(C) \subseteq \{p_1, \cdots, p_k\} \Rightarrow \mathcal{I}_k(C) = \texttt{True}\Big). \qquad (*)$$

Additionally, the propositional interpretation $\mathcal{I}_k$ only defines values for the variables $p_1, \cdots, p_k$, so

$$\texttt{dom}(\mathcal{I}_k) = \{p_1, \cdots, p_k\}.$$

I.A.: $k = 0$.

We define $\mathcal{I}_0$ as the empty propositional interpretation, which assigns no values to any variables. To prove $(*)$, we must show that for each clause $C \in \overline{M}$ that contains no propositional variable, $\mathcal{I}_0(C) = \texttt{True}$ holds. The only clause that contains no variable is the empty clause. Since we assumed that $M \nvdash \{\}$, $\overline{M}$ cannot contain the empty clause. Thus, there is nothing to prove.

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

$\mathcal{I}_k$ is already defined by the induction hypothesis. We first set

$$\mathcal{I}_{k+1}(p_i) := \mathcal{I}_k(p_i) \quad \text{for all } i = 1, \cdots, k$$

and must now define $\mathcal{I}_{k+1}(p_{k+1})$. This is done by case analysis.

(a) There exists a clause

$$C \cup \{p_{k+1}\} \in \overline{M},$$

such that C contains at most the variables $p_1, \cdots, p_k$ and moreover $\mathcal{I}_k(C) = \texttt{False}$. In this case, we set

$$\mathcal{I}_{k+1}(p_{k+1}) := \texttt{True},$$

since otherwise $\mathcal{I}_{k+1}\big(C \cup \{p_{k+1}\}\big) = \texttt{False}$ would hold.

We must now show that for every clause $D \in \overline{M}$ that contains only the variables $p_1, \cdots, p_k, p_{k+1}$, the statement $\mathcal{I}_{k+1}(D) = \texttt{True}$ holds. There are three possibilities:

1. Case: $\texttt{var}(D) \subseteq \{p_1, \cdots, p_k\}$
   Then the claim holds by the induction hypothesis, since the interpretations $\mathcal{I}_{k+1}$ and $\mathcal{I}_k$ agree on these variables.
2. Case: $p_{k+1} \in D$.
   Since we defined $\mathcal{I}_{k+1}(p_{k+1})$ as $\texttt{True}$, $\mathcal{I}_{k+1}(D) = \texttt{True}$ holds.

3. Case: $(\neg p_{k+1}) \in D$.
   Then D has the form

$$D = E \cup \{\neg p_{k+1}\}.$$

   At this point, we need the fact that $\overline{M}$ is saturated. We apply the cut rule to the clauses $C \cup \{p_{k+1}\}$ and $E \cup \{\neg p_{k+1}\}$:

$$C \cup \{p_{k+1}\}, \quad E \cup \{\neg p_{k+1}\} \quad \vdash \quad C \cup E$$

   Since $\overline{M}$ is saturated, $C \cup E \in \overline{M}$. $C \cup E$ contains only the variables $p_1, \cdots, p_k$. Therefore by the induction hypothesis,

$$\mathcal{I}_{k+1}(C \cup E) = \mathcal{I}_k(C \cup E) = \texttt{True}.$$

   Since $\mathcal{I}_k(C) = \texttt{False}$, $\mathcal{I}_k(E) = \texttt{True}$ must hold. Thus we have

$$\begin{aligned} \mathcal{I}_{k+1}(D) &= \mathcal{I}_{k+1}\big(E \cup \{\neg p_{k+1}\}\big) \\ &= \mathcal{I}_k(E) \circledvee \mathcal{I}_{k+1}(\neg p_{k+1}) \\ &= \texttt{True} \circledvee \texttt{False} = \texttt{True} \end{aligned}$$

   and that was to be shown.

(b) There is no clause

$$C \cup \{p_{k+1}\} \in \overline{M},$$

such that C contains at most the variables $p_1, \cdots, p_k$ and moreover $\mathcal{I}_k(C) = \texttt{False}$. In this case, we set

$$\mathcal{I}_{k+1}(p_{k+1}) := \texttt{False}.$$

In this case, for clauses $D \in \overline{M}$, there are three cases:

1. D does not contain the variable p_{k+1}.
   In this case, $\mathcal{I}_{k+1} = \texttt{True}$ already holds by the induction hypothesis.
2. $D = E \cup \{p_{k+1}\}$.
   Then we must have $\mathcal{I}_k(E) = \texttt{True}$ since otherwise we would be in case (a) of the outer case distinction. Obviously, $\mathcal{I}_k(E) = \texttt{True}$ implies $\mathtt{I}_k(D) = \texttt{True}$.
3. $D = E \cup \{\neg p_{k+1}\}$.
   In this case, by definition of $\mathcal{I}$, $\mathcal{I}_{k+1}(p_{k+1}) = \texttt{False}$ holds and hence $\mathcal{I}_{k+1}(\neg p_{k+1}) = \texttt{True}$ follows, resulting in $\mathcal{I}_{k+1}(D) = \texttt{True}$.

Through the induction, we have thus shown that the propositional interpretation $\mathcal{I}_N$ renders all clauses $C \in \overline{M}$ true, as only the propositional variables $p_1, \cdots, p_N$ occur in $\overline{M}$. Since $M \subseteq \overline{M}$, $\mathcal{I}_N$ also renders all clauses from M true, contradicting the assumption $M \models \bot$. □

4.6.4 Constructive Interpretation of the Proof of Refutation Completeness

In this section, we implement a program that takes a set of clauses M as its input. If the empty set of clauses cannot be derived from M, i.e. if

$$M \nvdash \{\},$$

then it returns a propositional valuation $\mathcal{I}$ that satisfies all clauses in M. This program is shown in Figures 4.10, 4.11, and 4.12 on the following pages. You can find this program at the address

github.com/karlstroetmann/Logic/blob/master/Python/Chapter-4/05-Completeness.ipynb
online.

```
def complement(l):
    "Compute the complement of the literal l."
    match l:
        case p if isinstance(p, str):  return ('¬', p)
        case ('¬', p):                 return p

 def extractVariable(l):
    "Extract the variable of the literal l."
    match l:
        case p if isinstance(p, str): return p
        case ('¬', p):                return p

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 4.10: Auxiliary function that are used in Figure 4.11.

The basic idea of this program is that we attempt to derive all clauses from a given set M of clauses that can be derived from M by using the cut rule. Let us call this set $\widetilde{M}$. If $\widetilde{M}$ contains the empty clause, then, due to the correctness of the cut rule, M has to be unsatisfiable. However, if we fail to derive the empty clause from M, we construct a propositional interpretation $\mathcal{I}$ from $\widetilde{M}$ such that $\mathcal{I}$ satisfies all clauses from M. We first discuss the auxiliary procedures shown in Figure 4.10.

1. The function `complement` takes as argument a literal l and computes the complement $\overline{l}$ of this literal. If the literal l is a propositional variable p, which we recognize by l being a string, then we have $\overline{p} = \neg p$. If l is in the form $\neg p$ with a propositional variable p, then $\overline{\neg p} = p$.

2. The function `extractVariable` extracts the propositional variable contained in a literal l. The implementation is analogous to the previous implementation of the function `complement` with

a case distinction, considering whether l is either in the form p or in the form $\neg p$, where p is the propositional variable to be extracted.

3. The function `collectVars` takes as an argument a set M of clauses, where each clause $C \in M$ is represented as a set of literals. The task of the function `collectVars` is to compute the set of all propositional variables that occur in any of the clauses C from M. During the implementation, we first iterate over the clauses C of the set M and then for each clause C over the literals l occurring in C, where the literals are transformed into propositional variables using the function `extractVariable`.

4. The function `cutRule` takes two clauses C_1 and C_2 as its arguments and calculates the set of all clauses that can be derived using an application of the cut rule from C_1 and C_2. For example, from the two clauses

   $$\{p, q\} \quad \text{and} \quad \{\neg p, \neg q\}$$

   we can derive both the clause

   $$\{q, \neg q\} \quad \text{as well as the clause} \quad \{p, \neg p\}$$

   using the cut rule.

```
def saturate(Clauses):
    while True:
        Derived = { C for C1 in Clauses
                      for C2 in Clauses
                      for C in cutRule(C1, C2)
                  }
        if frozenset() in Derived:
            return { frozenset() }  # This is the set notation of ⊥.
        Derived -= Clauses
        if Derived == set():        # no new clauses found
            return Clauses
        Clauses |= Derived
```

Figure 4.11: Die Funktion `saturate`

Figure 4.11 shows the function `saturate`. This function takes as input a set `Clauses` of propositional clauses, represented as sets of literals. The task of the function is to derive all clauses that can be derived directly or indirectly from the set `Clauses` using the cut rule. Specifically, the set S of clauses returned by the function `saturate` is saturated under the application of the cut rule, meaning:

1. If S contains the empty clause $\{\}$, then S is saturated.

2. Otherwise, `Clauses` must be a subset of S and additionally the following must hold: If for a literal l both the clause $C_1 \cup \{l\}$ and the clause $C_2 \cup \{\overline{l}\}$ are contained in S, then the clause $C_1 \cup C_2$ is also an element of the set of clauses S:

   $$C_1 \cup \{l\} \in S \ \wedge \ C_2 \cup \{\overline{l}\} \in S \ \Rightarrow \ C_1 \cup C_2 \in S$$

We now explain the implementation of the function `saturate`.

1. The `while` loop that starts in line 28 is tasked with applying the cut rule as long as possible to derive new clauses from the given clauses using the cut rule. Since this loop's condition has the value `True`, it can only be terminated by executing one of the two `return` commands in line 34 or line 37.

2. In line 29, the set `Derived` is defined as the set of clauses that can be inferred using the cut rule from two of the clauses in the set `Clauses`.

3. If the set `Derived` contains the empty clause, then the set `Clauses` is contradictory, and the function `saturate` returns as a result the set $\{\{\}\}$, where the inner set must be represented as `frozenset`. Note that the set $\{\{\}\}$ corresponds to Falsum.

4. Otherwise, in line 35, we first subtract from the set `Derived` the clauses that were already present in the set `Clauses`, as we are concerned with determining whether we actually found new clauses in the last step in line 31, or whether all clauses that we derived in the last step were already known.

5. If we find in line 36 that we have not derived any new clauses, then the set `Clauses` is saturated and we return this set in line 37.

6. Otherwise, we add the newly found clauses to the set `Clauses` in line 38 and continue the `while` loop.

At this point, we must verify that the `while` loop will eventually terminate. This is due to two reasons:

1. In each iteration of the loop, the number of elements of the set `Clauses` is increased by at least one, as we know that the set `Derived`, which we add to the set `Clauses` in line 38, is neither empty nor does it contains only clauses that are already present in `Clauses`.

2. The set `Clauses`, with which we originally start, contains a certain number n of propositional variables. However, the application of the cut rule does not generate any new variables. Therefore, the number of propositional variables that appear in `Clauses` always remains the same. This limits the number of literals that can appear in `Clauses`: If there are only n propositional variables, then there can be at most $2 \cdot n$ different literals. However, each clause from `Clauses` is a subset of the set of all literals. Since a set with k elements has a total of 2^k subsets, there can be at most $2^{2 \cdot n}$ different clauses that can appear in `Clauses`.

From the two reasons given above, we can conclude that the `while` loop that starts in line 28 will terminate after at most $2^{2 \cdot n}$ iterations.

```
39 def findValuation(Clauses):
40     "Given a set of Clauses, find an interpretation satisfying all clauses."
41     Variables = collectVariables(Clauses)
42     Clauses   = saturate(Clauses)
43     if frozenset() in Clauses:  # The set Clauses is inconsistent.
44         return False
45     Literals = set()
46     for p in Variables:
47         if any(C for C in Clauses
48                  if  p in C and C - {p} <= { complement(l) for l in Literals }
49               ):
50             Literals |= { p }
51         else:
52             Literals |= { ('¬', p) }
53     return Literals
```

Figure 4.12: Die Funktion `findValuation`.

Next, we discuss the implementation of the function `findValuation`, which is shown in Figure 4.12. This function receives a set `Clauses` of clauses as input. If this set is contradictory, the function should return the result `False`. Otherwise, the function should compute a propositional valuation $\mathcal{I}$ that satisfies all clauses from the set `Clauses`. In detail, the function `findValuation` works as follows.

1. First, in line 41, we compute the set of all propositional variables that appear in the set `Clauses`. We need this set because in the propositional interpretation, which we want to return as a result, we must map these variables to the set $\{\texttt{True}, \texttt{False}\}$.

2. In line 42, we saturate the set `Clauses` and compute all clauses that can be derived from the original set of clauses using the cut rule. Here, two cases can occur:

   (a) If the empty clause can be derived, then it follows from the correctness of the cut rule, that the original set of clauses is contradictory, and we return `False` instead of an assignment, as a contradictory set of clauses is certainly not satisfiable.

   (b) Otherwise, we now compute a propositional assignment under which all clauses from the set `Clauses` become true. For this purpose, we first compute a set of literals, which we store in the variable `Literals`. The idea is that we include the propositional variable `p` in the set `Literals` exactly when the sought-after assignment $\mathcal{I}$ evaluates the propositional variable `p` to `True`. Otherwise, we include the literal $\neg$`p` in the set `Literals`. As a result, in line 53, we return the set `Literals`. The sought-after propositional assignment $\mathcal{I}$ can then be computed according to the formula

$$\mathcal{I}(\mathtt{p}) = \begin{cases} \mathtt{True} & \text{if } \ \mathtt{p} \in \mathtt{Literals} \\ \mathtt{False} & \text{if } \neg\mathtt{p} \in \mathtt{Literals}. \end{cases}$$

3. The computation of the set `Literals` is done using a `for` loop. The idea is that for a propositional variable `p`, we add the literal `p` to the set `Literals` exactly when the assignment $\mathcal{I}$ has to

map the variable `p` to `True` in order to satisfy the clauses. Otherwise, we add the literal $\neg$`p` to this set instead.

The condition for adding the literal p is as follows: Suppose we have already found values for the variables $p_1, \cdots, p_n$ in the set `Literals`. The values of these variables are determined by the literals $l_1, \cdots, l_n$ in the set `Literals` as follows: If $l_i = p_i$, then $\mathcal{I}(p_i) = \texttt{True}$ and if $l_i = \neg p_i$, then we have $\mathcal{I}(p_i) = \texttt{False}$. Now assume that a clause C exists in the set `Clauses` such that

$$C\backslash\{p\} \subseteq \{\overline{l_1}, \cdots, \overline{l_n}\} \quad \text{and} \quad \texttt{p} \in C$$

holds. If $\mathcal{I}(C) = \texttt{True}$ is to hold, then $\mathcal{I}(\texttt{p}) = \texttt{True}$ must hold, because by construction of $\mathcal{I}$ we have

$$\mathcal{I}(\overline{l_i}) = \texttt{False} \quad \text{for all } i \in \{1, \cdots, n\}$$

and thus `p` is the only literal in the clause C that we can still make true with the help of the assignment $\mathcal{I}$. In this case, we thus add the literal `p` to the set `Literals`. Otherwise, the literal $\neg p$ is added to the set `Literals`.

The definition of the function `findValuation` implements the inductive definition of the assignments $\mathcal{I}_k$ that we specified in the proof of refutation completeness.

4.7 The Algorithm of Davis and Putnam

In practice, the we often have to compute a propositional assignment $\mathcal{I}$ for a given set of clauses K such that

$$\texttt{evaluate}(C, \mathcal{I}) = \texttt{True} \quad \text{for all } C \in K$$

holds. In this case, the assignment $\mathcal{I}$ is a solution of the set of clauses K. In the last section, we have already introduced the function `findValuation` which can be used to compute such a solution of a set of clauses. Unfortunately, this function is sufficiently efficient to be useful in practice. Therefore, in this section, we will introduce an algorithm that enables us to compute a solution for a set of propositional clauses in many practically relevant cases, even when the number of variables is large. This procedure is based on Davis and Putnam [DP60, DLL62]. Refinements of this procedure [MMZ+01] are used, for example, to verify the correctness of digital electronic circuits.

To motivate the algorithm, let us first consider those cases where it is immediately clear whether there is an assignment that solves a set of clauses K. Consider the following example:

$$K_1 = \{\ \{p\},\ \{\neg q\},\ \{r\},\ \{\neg s\},\ \{\neg t\}\ \}$$

The clause set K_1 corresponds to the propositional formula

$$p \wedge \neg q \wedge r \wedge \neg s \wedge \neg t.$$

Therefore, K_1 is solvable and the assignment

$$\mathcal{I} = \{\ p \mapsto \texttt{True},\ q \mapsto \texttt{False},\ r \mapsto \texttt{True},\ s \mapsto \texttt{False},\ t \mapsto \texttt{False}\ \}$$

is a solution of K_1. Consider another example:

$$K_2 = \{\ \{\}, \{p\},\ \{\neg q\},\ \{r\}\ \}$$

This clause set corresponds to the formula

$$\bot \wedge p \wedge \neg q \wedge r.$$

Obviously, K_2 is unsolvable. As a final example, consider

$$K_3 = \big\{\{p\}, \{\neg q\}, \{\neg p\}\big\}.$$

This clause set encodes the formula

$$p \wedge \neg q \wedge \neg p$$

and is obviously also unsolvable, as a solution $\mathcal{I}$ would have to make the propositional variable p both true and false simultaneously. We take the observations made in the last three examples as the basis for two definitions.

Definition 25 (Unit Clause) *A clause C is a unit clause if C consists of only one literal. Then either*

$$C = \{p\} \quad \text{or} \quad C = \{\neg p\}$$

for a propositional variable p. ◇

Definition 26 (Trivial Set of Clauses) *A set of clauses K is a trivial set of clauses if one of the following two cases applies:*

1. *K contains the empty clause, so $\{\} \in K$.*

   In this case, K is obviously unsolvable.

2. *K contains only unit clauses with different propositional variables. Denoting the set of propositional variables by $\mathcal{P}$, this condition is expressed as*

   (a) $\texttt{card}(C) = 1 \quad$ *for all $C \in K$ and*

   (b) There is no $p \in \mathcal{P}$ such that:

   $$\{p\} \in K \quad \text{and} \quad \{\neg p\} \in K.$$

   In this case, we can define the propositional assignment $\mathcal{I}$ as follows:

   $$\mathcal{I}(p) = \begin{cases} \texttt{True} & \text{if} \quad \{p\} \in K, \\ \texttt{False} & \text{if} \quad \{\neg p\} \in K. \end{cases}$$

   Then $\mathcal{I}$ is a solution of the set of clauses K. ◇

How can we transform a given set of clauses into a trivial set of clauses? There are three ways to simplify a set of clauses. We are already familiar with the first of the two possibilities, and we will explain the other two in more detail later.

1. Resolution Rule,

2. Subsumption, and

3. Case Distinction.

Next, we will consider these possibilities in turn.

4.7.1 Simplification with the Resolution Rule

A typical application of the resolution rule has the form:

$$\frac{C_1 \cup \{p\} \quad \{\neg p\} \cup C_2}{C_1 \cup C_2}$$

Usually, the clause $C_1 \cup C_2$ that is generated here will contain more literals than the premises $C_1 \cup \{p\}$ and $\{\neg p\} \cup C_2$. If the clause $C_1 \cup \{p\}$ contains a total of $m+1$ literals and the clause $\{\neg p\} \cup C_2$ contains a total of $n+1$ literals, then the conclusion $C_1 \cup C_2$ can contain up to $m+n$ literals. Of course, it can also contain fewer literals if there are literals that occur in both C_1 and C_2. Often, $m+n$ is greater than both $m+1$ and $n+1$. We are only certain that the clauses do not grow if we have $n=0$ or $m=0$. This case occurs when one of the two clauses consists of a single literal and is consequently a unit clause. Since our goal is to simplify the set of clauses, we allow only applications of the resolution rule where one of the clauses is a unit clause. Such cuts are referred to as unit cuts. To perform all possible cuts with a given unit clause $\{l\}$, we define a function

$$\texttt{unitCut} : 2^{\mathcal{K}} \times \mathcal{L} \rightarrow 2^{\mathcal{K}}$$

such that for a set of clauses K and a literal l the function $\texttt{unitCut}(K,l)$ simplifies the set of clauses K as much as possible with unit cuts with the clause $\{l\}$:

$$\texttt{unitCut}(K,l) = \Big\{ C \backslash \{\overline{l}\} \;\Big|\; C \in K \Big\}.$$

Note that the set $\texttt{unitCut}(K,l)$ contains the same number of clauses as the set K. However, those clauses from the set K that contain the literal $\overline{l}$ have been reduced. All other clauses from K remain unchanged.

Of course, we only simplify a set of clauses K using the expression $\texttt{unitCut}(K,l)$ if the unit clause $\{l\}$ is an element of the set K.

4.7.2 Simplification via Subsumption

We start by demonstrating the principle of subsumption with an example. Consider the set of clauses

$$K = \big\{ \{p, q, \neg r\}, \{p\} \big\} \cup M.$$

Clearly, the clause $\{p\}$ implies the clause $\{p, q, \neg r\}$, because whenever $\{p\}$ is satisfied, $\{q, p, \neg r\}$ is automatically satisfied as well. This is because

$$\models p \rightarrow q \vee p \vee \neg r$$

holds. Generally, we say that a clause C is subsumed by a unit clause U when

$$U \subseteq C$$

applies. If K is a set of clauses with $C \in K$, the unit clause $U \in K$, and C is subsumed by U, then we can reduce the set K by unit subsumption to the set $K - \{C\}$, thus we can delete the clause C from K. In order to implement this, we define a function

$$\texttt{subsume} : 2^{\mathcal{K}} \times \mathcal{L} \rightarrow 2^{\mathcal{K}},$$

which simplifies a given set of clauses K, containing the unit clause $\{l\}$, through subsumption by deleting all clauses subsumed by $\{l\}$ from K. The unit clause $\{l\}$ itself is, of course, retained. Therefore, we define:

$$\texttt{subsume}(K, l) := \bigl(K \backslash \{C \in K \mid l \in C\}\bigr) \cup \bigl\{\{l\}\bigr\} = \bigl\{C \in K \mid l \notin C\bigr\} \cup \bigl\{\{l\}\bigr\}.$$

In the above definition, the unit clause $\{l\}$ must be included in the result because the set $\{C \in K \mid l \notin C\}$ does not contain the unit clause $\{l\}$. The two sets of clauses $\texttt{subsume}(K, l)$ and K are equivalent if and only if $\{l\} \in K$. Therefore, a set of clauses K will only be simplified using the expression $\texttt{subsume}(K, l)$ if the unit clause $\{l\}$ is contained in the set K.

4.7.3 Simplification through Case Distinction

A calculus that only uses unit cuts and subsumption is not refutation-complete. Therefore, we need another method to simplify sets of clauses. Such a simplification is case distinction. This principle is based on the following theorem.

Satz 27 *If K is a set of clauses and p is a propositional variable, then K is satisfiable if and only if $K \cup \bigl\{\{p\}\bigr\}$ or $K \cup \bigl\{\{\neg p\}\bigr\}$ is satisfiable.*

Proof:

"$\Rightarrow$": If K is satisfied by an assignment $\mathcal{I}$, then there are two possibilities for $\mathcal{I}(p)$, as $\mathcal{I}(p)$ is either true or false. If $\mathcal{I}(p) = \texttt{True}$, then the set $K \cup \bigl\{\{p\}\bigr\}$ is also satisfiable, otherwise $K \cup \bigl\{\{\neg p\}\bigr\}$ is satisfiable.

"$\Leftarrow$": Since K is a subset of both $K \cup \bigl\{\{p\}\bigr\}$ and of $K \cup \bigl\{\{\neg p\}\bigr\}$, it is clear that K is satisfiable if any of these sets is satisfiable. □

We can now simplify a set of clauses K by selecting a propositional variable p that occurs in K. Then we form the sets

$$K_1 := K \cup \bigl\{\{p\}\bigr\} \quad \text{and} \quad K_2 := K \cup \bigl\{\{\neg p\}\bigr\}$$

and recursively investigate whether K_1 is satisfiable. If we find a solution for K_1, this is also a solution for the original set of clauses K, and we have achieved our goal. Otherwise, we recursively investigate whether K_2 is satisfiable. If we find a solution, this is also a solution for K. If we find no solution for either K_1 or K_2, then K cannot have a solution either, since any solution $\mathcal{I}$ for K must make the variable p either true or false. The recursive examination of K_1 or K_2 is easier than the examination of K, because in K_1 and K_2, with the unit clauses $\{p\}$ and $\{\neg p\}$, we can perform both unit subsumptions and unit cuts, thereby simplifying these sets.

4.7.4 The Algorithm

We can now outline the Davis and Putnam algorithm . Given a set of clauses K, the goal is to find a solution for K. We are looking for an assignment $\mathcal{I}$, such that:

$$\mathcal{I}(C) = \texttt{True} \quad \text{for all } C \in K.$$

The algorithm of Davis and Putnam consists of the following steps.

1. Perform all unit cuts and unit subsumptions that are possible with clauses from K.
2. Then, If K is trivial, the procedure ends.

(a) If $\{\} \in K$, then K is unsolvable.

(b) Otherwise, the propositional interpretation

$$\mathcal{I} := \bigl\{p \mapsto \texttt{True} \mid p \in \mathcal{P} \wedge \{p\} \in K\bigr\} \cup \bigl\{p \mapsto \texttt{False} \mid p \in \mathcal{P} \wedge \{\neg p\} \in K\bigr\}$$

is a solution for K.

3. If K is not trivial, select a propositional variable p that appears in K.

(a) We then try recursively to solve the set of clauses

$$K \cup \bigl\{\{p\}\bigr\}$$

If successful, we have found a solution for K.

(b) If $K \cup \bigl\{\{p\}\bigr\}$ is unsolvable, we instead try to solve the set of clauses

$$K \cup \bigl\{\{\neg p\}\bigr\}$$

If this also fails, K is unsolvable. Otherwise, we have found a solution for K.

In order to implement this algorithm, it is convenient to combine the two functions `unitCut`() and `subsume`() into a single function. Therefore, we define the function

$$\texttt{reduce} : 2^{\mathcal{K}} \times \mathcal{L} \to 2^{\mathcal{K}}$$

as follows:

$$\texttt{reduce}(K, l) = \Bigl\{ C\backslash\{\overline{l}\} \mid C \in K \wedge \overline{l} \in C \Bigr\} \cup \Bigl\{ C \in K \mid \overline{l} \notin C \wedge l \notin C \Bigr\} \cup \bigl\{\{l\}\bigr\}.$$

Thus, the set contains the results of cuts with the unit clause $\{l\}$ and we have removed those clauses that are subsumed by the unit clause $\{l\}$. The two sets K and $\texttt{reduce}(K, l)$ are logically equivalent, if $\{l\} \in K$. Therefore, K will only be replaced by $\texttt{reduce}(K, l)$ if $\{l\} \in K$.

4.7.5 An Example

To illustrate, we demonstrate the algorithm of Davis and Putnam with an example. The set K is defined as follows:

$$K := \Bigl\{ \{p, q, s\},\ \{\neg p, r, \neg t\},\ \{r, s\},\ \{\neg r, q, \neg p\}, \{\neg s, p\},\ \{\neg p, \neg q, s, \neg r\},\ \{p, \neg q, s\},\ \{\neg r, \neg s\},\ \{\neg p, \neg s\} \Bigr\}.$$

We will show that K is unsolvable. Since the set K contains no unit clauses, there is nothing to do in the first step. Since K is not trivial, we are not finished yet. Thus, we proceed to step 3 and choose a propositional variable that occurs in K. At this point, it makes sense to choose a variable that appears in as many clauses of K as possible. We therefore choose the propositional variable p.

1. First, we form the set

$$K_0 := K \cup \bigl\{\{p\}\bigr\}$$

and attempt to solve this set. To do this, we form

$$K_1 := \texttt{reduce}\bigl(K_0, p\bigr) = \Bigl\{ \{r, \neg t\},\ \{r, s\},\ \{\neg r, q\},\ \{\neg q, s, \neg r\},\ \{\neg r, \neg s\},\ \{\neg s\},\ \{p\} \Bigr\}.$$

The clause set K_1 contains the unit clause $\{\neg s\}$, so next we reduce using this clause:

$$K_2 := \texttt{reduce}\bigl(K_1, \neg s\bigr) = \Bigl\{ \{r, \neg t\},\ \{r\},\ \{\neg r, q\},\ \{\neg q, \neg r\},\ \{\neg s\},\ \{p\} \Bigr\}.$$

We have found the new unit clause $\{r\}$. We use this unit clause to reduce K_2:

$$K_3 := \mathit{reduce}(K_2, r) = \Big\{\{r\},\ \{q\},\ \{\neg q\},\ \{\neg s\},\ \{p\}\Big\}$$

Since K_3 contains the unit clause $\{q\}$, we now reduce with q:

$$K_4 := \mathit{reduce}(K_3, q) = \Big\{\{r\},\ \{q\},\ \{\},\ \{\neg s\},\ \{p\}\Big\}.$$

The clause set K_4 contains the empty clause and is therefore unsolvable.

2. Thus, we now form the set

$$K_5 := K \cup \big\{\{\neg p\}\big\}$$

and attempt to solve this set. We form

$$K_6 = \mathit{reduce}(K_5, \neg p) = \Big\{\{q, s\},\ \{r, s\},\ \{\neg s\},\ \{\neg q, s\},\ \{\neg r, \neg s\},\ \{\neg p\}\Big\}.$$

The set K_6 contains the unit clause $\{\neg s\}$. We therefore form

$$K_7 = \mathit{reduce}(K_6, \neg s) = \Big\{\{q\},\ \{r\},\ \{\neg s\},\ \{\neg q\},\ \{\neg p\}\Big\}.$$

The set K_7 contains the new unit clause $\{q\}$. We use this unit clause to reduce K_7:

$$K_8 = \mathit{reduce}(K_7, q) = \Big\{\{q\},\ \{r\},\ \{\neg s\},\ \{\},\ \{\neg p\}\Big\}.$$

Since K_8 contains the empty clause, K_8 and thus the originally given set K is unsolvable.

In this example, we were fortunate as we only had to make a single case distinction. In more complex examples, it is often necessary to perform more than one case distinction.

4.7.6 Implementation of the Davis and Putnam Algorithm

Next, we provide the implementation of the function `solve`, which can answer the question of whether a set of clauses is satisfiable. The implementation is shown in Figure 4.13 on page 70. The function receives two arguments: The sets `Clauses` and `Variables`. Here, `Clauses` is a set of clauses and `Variables` is a set of variables. If the set `Clauses` is satisfiable, then the call

```
solve(Clauses, Variables)
```

returns a set of unit clauses `Result`, such that any assignment $\mathcal{I}$, which satisfies all unit clauses from `Result`, also satisfies all clauses from the set `Clauses`. If the set `Clauses` is not satisfiable, the call

```
solve(Clauses, Variables)
```

returns the set $\big\{\{\}\big\}$, as the empty clause represents the unsatisfiable formula $\bot$.

You might wonder why we need the set `Variables` in the function `solve`. The reason is that we need to keep track of which variables we have already used for case distinctions during the recursive calls. These variables are collected in the set `Variables`.

1. In line 2, we reduce the given set of clauses `Clauses` as much as possible using the method `saturate`, applying unit cuts and removing all clauses that are subsumed by unit clauses.
2. Next, we test in line 5 if the simplified set of clauses `S` contains the empty clause, and if so, we return the set $\big\{\{\}\big\}$ as the result.

```
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
9      l       = selectLiteral(S, Variables)
10     lBar    = complement(l)
11     p       = extractVariable(l)
12     newVars = Variables | { p }
13     Result  = solve(S | { frozenset({l}) }, newVars)
14     if Result != Falsum:
15         return Result
16     return solve(S | { frozenset({lBar}) }, newVars)
```

Figure 4.13: The function `solve`.

3. Then, in line 7, we check if all clauses C from the set `S` are unit clauses. If this is the case, the set `S` is trivial and we return this set as the result.
4. Otherwise, in line 9 we select a literal l that appears in a clause from the set `S` but has not yet been used. We then recursively investigate in line 13 whether the set

   $$\texttt{S} \cup \{\{\texttt{l}\}\}$$

   is solvable. There are two cases:

   (a) If this set is solvable, we return the solution of this set as the result.

   (b) Otherwise, we recursively check whether the set

   $$\texttt{S} \cup \left\{\{\overline{\texttt{l}}\}\right\}$$

   is solvable. If this set is solvable, then this solution is also a solution of the set `Clauses`, and we return this solution. If the set is unsolvable, then the set `Clauses` must also be unsolvable.

We now discuss the auxiliary procedures used in the implementation of the function `solve`. First, we discuss the function `saturate`. This function receives a set S of clauses as input and performs all possible unit cuts and unit subsumptions. The function `saturate` is shown in Figure 4.14 on page 71.

1. Initially, we copy the set `Clauses` into the variable `S`. This is necessary because we will later modify the set `S`. The function `saturate` should not alter the argument `Clauses` and therefore it has to create a copy of the set `S`.
2. Then, in line 3, we compute the set `Units` of all unit clauses.

```
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 4.14: The function `saturate`.

3. Next, in line 4, we initialize the set `Used` as an empty set. In this set, we record which unit clauses we have already used for unit cuts and subsumptions.

4. As long as the set `Units` of unit clauses is not empty, we select in line 6 an arbitrary unit clause `unit` from the set `Units` using the function `pop`, and remove this unit clause from the set `Units`.

5. In line 7, we add the clause `unit` to the set `Used` of used clauses.

6. In line 8, we extract the literal `l` from the clause `Unit` using the function `arb`. The function `arb` returns an arbitrary element of the set passed to it as an argument. If this set contains only one element, then that element is returned.

7. In line 9, the actual work is done by a call to the function `reduce`. This function performs all possible unit cuts with the unit clause $\{\texttt{l}\}$ and also removes all clauses that are subsumed by the unit clause $\{\texttt{l}\}$.

8. When the unit cuts with the unit clause $\{\texttt{l}\}$ are computed, new unit clauses can emerge, which we collect in line 10. Of course, we collect only those unit clauses that have not yet been used.

9. The loop in lines 5 – 10 is continued as long as we find unit clauses that have not been used.

10. At the end, return the remaining set of clauses is returned.

The function `reduce` is shown in Figure 4.15. In the previous section, we defined the function $\mathit{reduce}(S, l)$, which reduces a set of clauses `Cs` using the literal l, as

$$\mathit{reduce}(\texttt{Cs}, l) = \Big\{ C \backslash \big\{ \overline{l} \big\} \mid C \in \texttt{Cs} \wedge \overline{l} \in C \Big\} \cup \Big\{ C \in \texttt{Cs} \mid \overline{l} \notin C \wedge l \notin C \Big\} \cup \Big\{ \{l\} \Big\}$$

The code is an immediate implementation of this definition.

1. The function `selectLiteral` selects a literal from a given set `Clauses` of clauses, where the variable of the literal must not appear in the set `Forbidden` of variables that have already been used. To do this, we first iterate over all clauses C from the set `Clauses` and then over all literals

```
def reduce(Clauses, l):
    lBar = complement(l)
    return   { C - { lBar } for C in Clauses if lBar in C }          \
           | { C for C in Clauses if lBar not in C and l not in C }  \
           | { frozenset({l}) }
```

Figure 4.15: The function `reduce`.

```
def selectLiteral(Clauses, Forbidden):
    Variables = { extractVariable(l) for C in Clauses for l in C } - Forbidden
    Scores    = {}
    for var in Variables:
        cmp = ('¬', var)
        Scores[var] = 0.0
        Scores[cmp] = 0.0
        for C in Clauses:
            if var in C:
                Scores[var] += 2 ** -len(C)
            if cmp in C:
                Scores[cmp] += 2 ** -len(C)
    return max(Scores, key=Scores.get)

def extractVariable(l):
    match l:
        case ('¬', p): return p
        case p       : return p
```

Figure 4.16: The functions `select` and `negateLiteral`.

l in the clause C. From these literals, we extract the variable contained within using the function `extractVariable`. Subsequently, we return the literal for which the value of the Jeroslow-Wang Heuristic [JW90] is maximal. For a set of clauses K and a literal l, we define the Jeroslow-Wang heuristic $\texttt{jw}(K,l)$ as follows:

$$\texttt{jw}(K,l) := \sum_{\{C \in K \,|\, l \in C\}} \frac{1}{2^{|C|}}$$

Here, $|C|$ denotes the number of literals appearing in the clause C. The idea is that we want to select a literal l that appears in as many clauses as possible, as these clauses are then subsumed by the literal. However, it is more important to subsume clauses that contain as few literals as possible, as these clauses are harder to satisfy. This is because a clause is satisfied if even a single literal from the clause is satisfied. The more literals the clause contains, the easier it is to satisfy this clause.

The implementation of the other auxiliary function discussed here, `extractVariable`, is straightforward.

The version of the Davis and Putnam procedure presented above can be improved in many ways. Due to time constraints, we cannot discuss these improvements. Interested readers are referred to the following paper by Moskewicz et.al. [MMZ+01]:

> *Chaff: Engineering an Efficient SAT Solver*
> by M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik

Exercise 8: The set of clauses M is given as follows:

$$M := \{ \{r,p,s\}, \{r,s\}, \{q,p,s\}, \{\neg p, \neg q\}, \{\neg p, s, \neg r\}, \{p, \neg q, r\}, \\ \{\neg r, \neg s, q\}, \{p,q,r,s\}, \{r, \neg s, q\}, \{\neg r, s, \neg q\}, \{s, \neg r\} \}$$

Check whether the set M is contradictory. $\diamond$

4.8 The 8-Queens Problem

In this section, we demonstrate how certain combinatorial problems can be reduced to the question whether a set of formulas from propositional logic is satisfiable. These problems can then be solved using the algorithm of Davis and Putnam. As a specific example, we consider the 8-Queens Problem. This involves placing 8 queens on a chessboard in such a way that no queen can attack another queen. In the game of chess, a queen can attack another piece if that piece is either

- in the same row,
- in the same column, or
- on the same diagonal

as the queen. Figure 4.17 on page 74 shows a chessboard where in the third row and fourth column a queen is placed. This queen can move to all the fields marked with arrows, and thus can attack pieces that are on these fields.

First, let us consider how we can represent a chessboard with queens placed on it in propositional logic. One possibility is to introduce a propositional logic variable for each square. This variable expresses that there is a queen on the corresponding square. We assign names to these variables as follows: The variable that denotes the j-th square in the i-th row is represented by the string

> "Q<i,j>" with $i, j \in \{1, \cdots, 8\}$

We number the rows from top to bottom from 1 to 8, while the columns are numbered from left to right. Figure 4.19 on page 75 shows the assignment of variables to the squares. The function shown in Figure 4.18 $\texttt{var}(r, c)$ calculates the variable that expresses that there is a queen in row r and column c.

Next, we consider how to encode the individual conditions of the 8-Queens problem as formulas from propositional logic. Ultimately, all statements of the form

- "at most one queen in a row",

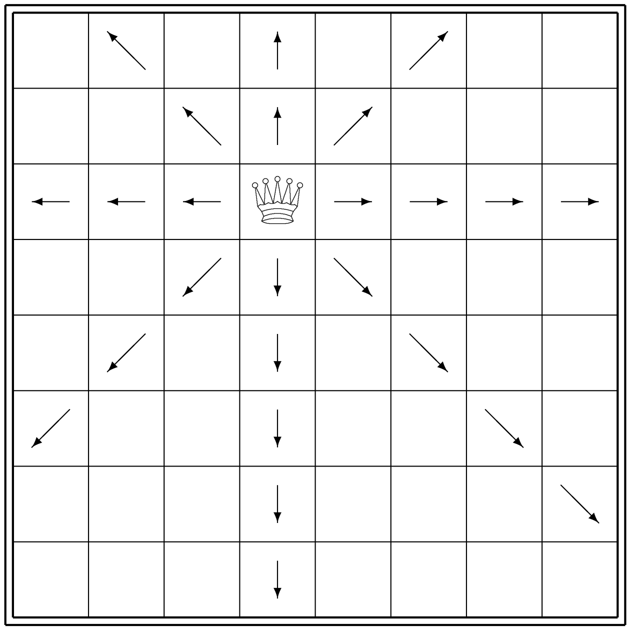

Figure 4.17: The 8-Queens-Problem.

```
def var(row, col):
    return 'Q<' + str(row) + ',' + str(col) + '>'
```

Figure 4.18: The function `var` to compute a propositional variable.

- "at most one queen in a column", or
- "at most one queen in a diagonal"

can be reduced to the same basic pattern: Given a set of propositional variables

$$V = \{x_1, \cdots, x_n\}$$

we need a formula that states that at most one of the variables in V has the value `True`. This is equivalent to the statement that for every pair $x_i, x_j \in V$ with $x_i \neq x_j$, the following formula holds:

$$\neg(x_i \wedge x_j).$$

This formula expresses that the variables x_i and x_j cannot both take the value `True` at the same time. According to De Morgan's laws, we have

$$\neg(x_i \wedge x_j) \Leftrightarrow \neg x_i \vee \neg x_j$$

and the clause on the right side of this equivalence can be written in set notation as

$$\{\neg x_i, \neg x_j\}.$$

Therefore, the formula for a set of variables V that expresses that no two different variables are simultaneously true can be written as a set of clauses in the form

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 4.19: Interpretation of the variables.

```
def atMostOne(S):
    return { frozenset({('¬',p), ('¬', q)}) for p in S
                                            for q in S
                                            if  p != q
           }
```

Figure 4.20: The function `atMostOne`.

$$\big\{ \{\neg p, \neg q\} \mid p \in V \wedge\ q \in V \wedge p \neq q \big\}$$

We implement these considerations in a *Python* function. The function `atMostOne()` shown in Figure 4.20 receives as input a set S of propositional variables. The call $\texttt{atMostOne}(S)$ calculates a set of clauses. These clauses are true if and only if at most one of the variables in S has the value `True`.

With the help of the function `atMostOne`, we can now implement the function `atMostOneInRow`. The call

`atMostOneInRow(row, n)`

calculates for a given row `row` and a board size of `n` a formula that expresses that at most one queen is in the row `row`. Figure 4.21 shows the function `atMostOneInRow`: We collect all the variables of the row specified by `row` in the set

$$\big\{\texttt{var}(\texttt{row}, j) \mid j \in \{1, \cdots, n\}\big\}$$

and call the function $\texttt{atMostOne}()$ with this set, which delivers the result as a set of clauses.

```python
def atMostOneInRow(row, n):
    return atMostOne({ var(row, col) for col in range(1,n+1) })
```

Figure 4.21: The function `atMostOneInRow`.

Next, we compute a formula that states that at least one queen is in a given column. For the first column, this formula would look like

$$\texttt{Q<1,1>} \vee \texttt{Q<2,1>} \vee \texttt{Q<3,1>} \vee \texttt{Q<4,1>} \vee \texttt{Q<5,1>} \vee \texttt{Q<6,1>} \vee \texttt{Q<7,1>} \vee \texttt{Q<8,1>}$$

and generally, for a column c where $c \in \{1, \cdots, 8\}$, the formula is

$$\texttt{Q<1,c>} \vee \texttt{Q<2,c>} \vee \texttt{Q<3,c>} \vee \texttt{Q<4,c>} \vee \texttt{Q<5,c>} \vee \texttt{Q<6,c>} \vee \texttt{Q<7,c>} \vee \texttt{Q<8,c>}.$$

When we write this formula in set notation as a set of clauses, we obtain

$$\big\{\{\texttt{Q<1,c>},\texttt{Q<2,c>},\texttt{Q<3,c>},\texttt{Q<4,c>},\texttt{Q<5,c>},\texttt{Q<6,c>},\texttt{Q<7,c>},\texttt{Q<8,c>}\}\big\}.$$

Figure 4.22 shows a *Python* function that calculates the corresponding set of clauses for a given column `col` and a given board size `n`. The step of going from a single clause to a set of clauses is necessary because our implementation of the Davis and Putnam algorithm works with a set of clauses.

```python
def oneInColumn(col, n):
    return { frozenset({ var(row, col) for row in range(1,n+1) }) }
```

Figure 4.22: The Function `oneInColumn`

At this point, you might expect us to provide formulas that express that there is at most one queen in a given column and that there is at least one queen in every row. However, such formulas are unnecessary because if we know that there is at least one queen in every column, we already know that there are at least 8 queens on the board. If we additionally know that there is at most one queen in each row, it is automatically clear that there are at most 8 queens on the board. Thus, there are exactly 8 queens on the board. Therefore, there can only be at most one queen per column, otherwise, we would have more than 8 queens on the board, and similarly, there must be at least one queen in each row, otherwise, we would not reach a total of 8 queens.

Next, let's consider how we can characterize the variables that are on the same diagonal. There are basically two types of diagonals: descending diagonals and ascending diagonals. We first consider the ascending diagonals. The longest ascending diagonal, also called the main diagonal, in the case of an 8×8 board consists of the variables

`Q<8,1>, Q<7,2>, Q<6,3>, Q<5,4>, Q<4,5>, Q<3,6>, Q<2,7>, Q<1,8>.`

The indices r and c of the variables $\mathtt{Q}(r,c)$ clearly satisfy the equation

$$r + c = 9.$$

In general, the indices of the variables of an ascending diagonal that contains more than one square satisfy the equation

$$r + c = k,$$

where k for an 8×8 chess board takes a value from the set $\{3, \cdots, 15\}$. We pass the value k as an argument in the function `atMostOneInRisingDiagonal`. This function is shown in Figure 4.23.

```
def atMostOneInRisingDiagonal(k, n):
    S = { var(row, col) for row in range(1, n+1)
                        for col in range(1, n+1)
                        if  row + col == k
        }
    return atMostOne(S)
```

Figure 4.23: The function `atMostOneInUpperDiagonal`

To see how the variables of a descending diagonal can be characterized, let's consider the main descending diagonal, which consists of the variables

`Q<1,1>, Q<2,2>, Q<3,3>, Q<4,4>, Q<5,5>, Q<6,6>, Q<7,7>, Q<8,8>`

The indices r and c of these variables clearly satisfy the equation

$$r - c = 0.$$

Generally, the indices of the variables on a descending diagonal satisfy the equation

$$r - c = k,$$

where k takes a value from the set $\{-6, \cdots, 6\}$. We pass the value k as an argument in the function `atMostOneInLowerDiagonal`. This function is shown in Figure 4.24.

Now we are able to summarize our results: We can construct a set of clauses that fully describe the 8-Queens problem. Figure 4.25 shows the implementation of the function `allClauses`. The call

```
def atMostOneInFallingDiagonal(k, n):
    S = { var(row, col) for row in range(1, n+1)
                        for col in range(1, n+1)
                        if  row - col == k
        }
    return atMostOne(S)
```

Figure 4.24: The function `atMostOneInLowerDiagonal`.

`allClauses`(n)

computes for a chessboard of size n a set of clauses that are satisfied if and only if on the chessboard

1. there is at most one queen in each row (line 2),
2. there is at most one queen in each descending diagonal (line 3),
3. there is at most one queen in each ascending diagonal (line 4), and
4. there is at least one queen in each column (line 5).

The expressions in the individual lines yield lists, whose elements are sets of clauses. What we need as a result, however, is a set of clauses and not a list of sets of clauses. Therefore, in line 6 we convert the list `All` into a set of clauses.

```
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 4.25: The function `allClauses`.

Finally, we show in Figure 4.26 the function `queens`, with which we can solve the 8-Queens problem.

1. First, we encode the problem as a set of clauses that is solvable only if the problem has a solution.
2. Then, we compute the solution using the function `solve` from the notebook that implements the algorithm of Davis and Putnam.

3. Finally, the computed solution is printed using the function `printBoard`.

   Here, `printBoard` is a function that prints the solution in a more readable format. However, this only works properly if a font is used where all characters have the same width. This function is shown for completeness in Figure 4.27, but we will not discuss the implementation further.

The complete program is available as a Jupyter Notebook on GitHub as

`karlstroetmann/Logic/blob/master/Python/Chapter-4/08-N-Queens.ipynb`.

```
def queens(n):
    "Solve the n queens problem."
    Clauses  = allClauses(n)
    Solution = solve(Clauses, set())
    if Solution != { frozenset() }:
        return Solution
    else:
        print(f'The problem is not solvable for {n} queens!')
```

Figure 4.26: The function `queens` that solves the n-queens-problem.

```
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 4.27: The function `show_solution`().

The set `solution` calculated by the call `solve`$(\textit{Clauses}, \{\})$ contains for each of the variables $\texttt{Q<}r\texttt{,}c\texttt{>}$ either the unit clause $\{\texttt{Q<}r\texttt{,}c\texttt{>}\}$ (if there is a queen on this square) or the unit clause $\{\texttt{(}\neg\texttt{, Q<}r\texttt{,}c\texttt{>)}\}$ (if the square remains empty). A graphical representation of a computed solution can be seen in Figure 4.28. This graphical representation was generated using the library `python-chess` and the function `show_solution`, which is shown in Figure 4.27.

Jessica Roth and Koen Loogman (who are two former DHBW students) implemented an animation of the Davis and Putnam procedure. You can find and try this animation at the address https://koenloogman.github.io/Animation-Of-N-Queens-Problem-In-JavaScript/

on the internet.

The 8-Queens problem is of course only a playful application of propositional logic. Nevertheless, it demonstrates the effectiveness of the Davis and Putnam algorithm very well, as the set of clauses computed by the function `allClauses` consists of 512 different clauses. In this set of clauses, 64 different variables appear.

In practice, there are many problems that can be similarly reduced to solving a set of clauses. For example, creating a timetable that meets certain constraints is one such problem. Generalizations of the timetable problem are referred to in the literature as Scheduling Problems. The efficient solution of such problems is a subject of current research.

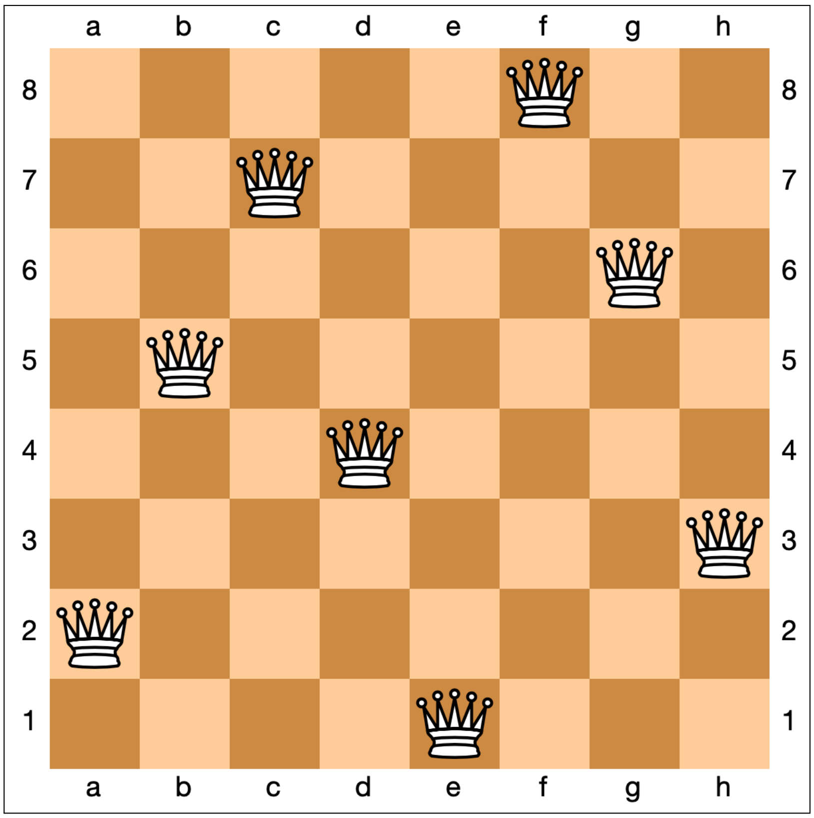

Figure 4.28: A solution of the 8-queens-problem.

Exercise 9: The following exercise is taken from the book 99 Logeleien von Zweistein. This book has been published 1968. It is written by Thomas von Randow.

The gentlemen Amann, Bemann, Cemann *and* Demann *are called - not necessarily in the same order - by their first names* Erich, Fritz, Gustav *and* Heiner. *They are all married to exactly one woman. We also know the following about them and their wives:*

1. *Either* Amann's *first name is* Heiner*, or* Bemann's *wife is* Inge*.*
2. *If* Cemann *is married to* Josefa*, then -* ***and only in this case*** *-* Klara's *husband is* ***not*** *called* Fritz*.*
3. *If* Josefa's *husband is* ***not*** *called* Erich*, then* Inge *is married to* Fritz*.* I
4. *If* Luise's *husband is called* Fritz*, then* Klara's *husband's first name is* ***not*** Gustav*.*
5. *If the wife of* Fritz *is called* Inge*, then* Erich *is* ***not*** *married to* Josefa*.*
6. *If* Fritz *is* ***not*** *married to* Luise*, then* Gustav's *wife's name is* Klara*.*
7. *Either* Demann *is married to* Luise*, or* Cemann *is called* Gustav*.*

What are the full names of these gentlemen, and what are their wives' first names?
have provided a framework notebook at the following address:

`karlstroetmann/Logic/blob/master/Python/Chapter-4/Zweistein.ipynb`

You should edit this notebook in order to solve the given problem.

4.9 Check Your Comprehension

(a) We defined the set of propositional logic formulas as expressions built from propositional variables, logical constants, and logical operators according to the syntax rules of propositional logic.

(b) The semantics of propositional logic formulas were established by defining truth values for these formulas based on the truth values of their constituent propositional variables and the nature of the logical operators used.

(c) In *Python*, propositional logic formulas can be represented using objects like strings or structured data types such as lists or tuples to mimic the logical structure of the formulas.

(d) A tautology is a propositional logic formula that is true under any assignment of truth values to its variables.

(e) Conjunctive normal form (CNF) is a form of representing propositional logic formulas where the formula is a conjunction (AND) of several clauses, and each clause is a disjunction (OR) of literals (variables or their negations).

(f) To calculate the conjunctive normal form of a given propositional logic formula, one can apply logical equivalences such as De Morgan's laws, distributive laws, etc. This calculation can be implemented in *Python* using recursion or iterative transformations based on the structure of the formula.

(g) We defined the proof concept $M \vdash C$ to mean that a clause C can be derived from a set of clauses M through a series of logical inference rules.

(h) The proof concept $\vdash$ has properties such as soundness (if $M \vdash C$, then $M \models C$) and completeness (if $M \models C$, then $M \vdash C$).

(i) A set of clauses is solvable if there exists an assignment of truth values to the variables in the clauses such that all the clauses are satisfied.

(j) The Davis and Putnam procedure is a systematic method for determining the satisfiability of a set of clauses in propositional logic by successive variable elimination and clause simplification.

(k) The 8-Queens problem can be formulated as a propositional logic problem by defining variables that represent the presence of a queen on each square of the board and then formulating clauses that encode the constraints of the queens not attacking each other.

Chapter 5

Prädikatenlogik

In der Aussagenlogik haben wir die Verknüpfung von atomaren Aussagen mit Junktoren untersucht. Die Prädikatenlogik untersucht zusätzlich auch die Struktur dieser atomaren 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) Weiter betrachten wir Normalformen prädikatenlogischer Formeln und zeigen, wie Formeln in erfüllbarkeits-äquivalente prädikatenlogische Klauseln umgewandelt werden können.

(f) Außerdem diskutieren wir einen prädikatenlogischen Kalkül, der die Grundlage des automatischen Beweisens in der Prädikaten-Logik ist.

(g) Zum Abschluss des Kapitels diskutieren wir den automatischen Theorem-Beweiser *Prover9*, sowie das Werkzeug *Mace4*, mit dem die Erfüllbarkeit von Formeln überprüft werden kann.

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 28 (Signatur) *Eine Signatur ist ein 4-Tupel*

$$\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \textsf{arity} \rangle,$$

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:*

   $$\textsf{arity} : \mathcal{F} \cup \mathcal{P} \to \mathbb{N}.$$

   Wir sagen, dass das Funktions- oder Prädikats-Zeichen f ein n-stelliges Zeichen ist, falls $\textsf{arity}(f) = n$ 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:*

   $$\mathcal{V} \cap \mathcal{F} = \{\}, \quad \mathcal{V} \cap \mathcal{P} = \{\}, \quad \text{und} \quad \mathcal{F} \cap \mathcal{P} = \{\}. \qquad \diamond$$

Als Bezeichner für Objekte verwenden wir Ausdrücke, die aus Variablen und Funktions-Zeichen aufgebaut sind. Solche Ausdrücke nennen wir Terme.

Definition 29 (Terme, $\mathcal{T}_\Sigma$) *Ist $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \textsf{arity} \rangle$ eine Signatur, so definieren wir die Menge der Σ-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. $arity := \{0 \mapsto 0, 1 \mapsto 0, + \mapsto 2, - \mapsto 2, * \mapsto 2, = \mapsto 2, \leq \mapsto 2\}$,

   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 Σ_{arith}-Terme konstruieren:

1. $x, y, z \in \mathcal{T}_{\Sigma_{\text{arith}}}$,

   denn alle Variablen sind auch Σ_{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. Dies funktioniert natürlich nur, wenn wir für die einzelnen Operatoren auch Bindungsstärken festlegen. ◇

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 30 (Atomare Formeln, $\mathcal{A}_\Sigma$) *Gegeben sei eine Signatur $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle$. Die Menge der atomaren Σ-Formeln $\mathcal{A}_\Sigma$ wird wie folgt definiert: Ist $p \in \mathcal{P}$ ein n-stelliges Prädikats-Zeichen und sind n Σ-Terme $t_1, \cdots, t_n$ gegeben, so ist $p(t_1, \cdots, t_n)$ eine atomaren Σ-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 Σ_{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 Σ-Term t die Menge der in t enthaltenen Variablen.

Definition 31 ($\textbf{Var}(t)$) *Ist $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle$ eine Signatur und ist t ein Σ-Term, so definieren wir die Menge $\mathit{Var}(t)$ der Variablen, die in t auftreten, durch Induktion nach dem Aufbau des Terms:*

1. *$\mathit{Var}(x) := \{x\}$ für alle $x \in \mathcal{V}$,*
2. *$\mathit{Var}\big(f(t_1, \cdots, t_n)\big) := \mathit{Var}(t_1) \cup \cdots \cup \mathit{Var}(t_n)$.* $\diamond$

Definition 32 (Σ-Formel, $\mathbb{F}_\Sigma$, gebundene und freie Variablen, $\boldsymbol{BV}(F)$, $\boldsymbol{FV}(F)$)
Es sei $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle$ eine Signatur. Die Menge der Σ-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 Σ-Formel, so gilt $F \in \mathbb{F}_\Sigma$. Weiter definieren wir:*
   (a) *$FV\big(p(t_1, \cdots, t_n)\big) := \mathit{Var}(t_1) \cup \cdots \cup \mathit{Var}(t_n)$.*
   (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) *$(F \rightarrow G) \in \mathbb{F}_\Sigma$,*
   (d) *$(F \leftrightarrow G) \in \mathbb{F}_\Sigma$.*

   Weiter definieren wir für alle Junktoren $\odot \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$:
   (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 Σ 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 Σ-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

$$(\exists x\colon\ \le(+(y,x),y))$$

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

$$\forall x,y\colon p(x,y) \quad \text{statt} \quad \forall x\colon (\forall y\colon p(x,y)).$$

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 $(\exists x\colon\ \le(+(y,x),y))$ wird dann lesbarer als

$$\exists x\colon y + x \le y$$

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

$$\big(\forall x \in M : F\big) \stackrel{\text{def}}{\Longleftrightarrow} \forall x : \big(x \in M \rightarrow F\big), \quad \text{und} \quad \big(\exists x \in M : F\big) \stackrel{\text{def}}{\Longleftrightarrow} \exists x : \big(x \in M \wedge F\big).$$

definiert sind.

5.2 Semantik der Prädikatenlogik

Als nächstes legen wir die Bedeutung der Formeln fest. Dazu definieren wir den Begriff einer Σ-Struktur. Eine solche Struktur legt fest, wie die Funktions- und Prädikats-Zeichen der Signatur Σ zu interpretieren sind.

Definition 33 (Struktur) *Es sei eine Signatur*

$$\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle.$$

gegeben. Eine Σ-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 Σ-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 $\mathit{arity}(f) = m$ 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 $\mathit{arity}(p) = n$ 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*
   $$=^{\mathcal{J}} = \bigl\{ \langle u, u \rangle \mid u \in \mathcal{U} \bigr\}.$$
   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 Σ_G der Gruppen-Theorie sei definiert als

$$\Sigma_G = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle \quad \text{mit}$$

1. $\mathcal{V} := \{x, y, z\}$
2. $\mathcal{F} := \{\mathrm{e}, *\}$
3. $\mathcal{P} := \{=\}$
4. $\mathit{arity} = \bigl\{ \langle \mathrm{e}, 0 \rangle, \langle *, 2 \rangle, \langle =, 2 \rangle \bigr\}$

Dann können wir eine Σ_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! $\diamond$

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 34 (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 Σ-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:

$$\mathcal{I}[x/c](y) := \begin{cases} c & \text{falls } y = x; \\ \mathcal{I}(y) & \text{sonst.} \end{cases}$$ $\diamond$

Definition 35 (Semantik der Terme) *Ist $\mathcal{S} = \langle \mathcal{U}, \mathcal{J}\rangle$ eine Σ-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 Σ-Terme der Form $f(t_1, \cdots, t_n)$ definieren wir*

   $$\mathcal{S}\big(\mathcal{I}, f(t_1, \cdots, t_n)\big) := f^{\mathcal{J}}\big(\mathcal{S}(\mathcal{I}, t_1), \cdots, \mathcal{S}(\mathcal{I}, t_n)\big).$$ $\diamond$

Beispiel: Mit der oben definieren Σ_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

$$\mathcal{Z}\big(\mathcal{I}, \mathtt{x} * \mathtt{y}\big) = 1.$$ $\diamond$

Definition 36 (Semantik der atomaren Σ-Formeln) *Ist $\mathcal{S}$ eine Σ-Struktur und $\mathcal{I}$ eine $\mathcal{S}$-Variablen-Belegung, so definieren wir für jede atomare Σ-Formel $p(t_1, \cdots, t_n)$ den Wert $\mathcal{S}\big(\mathcal{I}, p(t_1, \cdots, t_n)\big)$ wie folgt:*

$$\mathcal{S}\big(\mathcal{I}, p(t_1, \cdots, t_n)\big) := \Big(\big\langle \mathcal{S}(\mathcal{I}, t_1), \cdots, \mathcal{S}(\mathcal{I}, t_n)\big\rangle \in p^{\mathcal{J}}\Big).$$ $\diamond$

Beispiel: In Fortführung des obigen Beispiels gilt:

$$\mathcal{Z}(\mathcal{I}, x * y = y * x) = \texttt{True}.$$ ◇

Um die Semantik beliebiger Σ-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. $\circledvee\!: \mathbb{B} \times \mathbb{B} \to \mathbb{B}$,
3. $\circledwedge\!: \mathbb{B} \times \mathbb{B} \to \mathbb{B}$,
4. $\ominus\!\!\!\rightarrow : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$,
5. $\ominus\!\!\!\leftrightarrow : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$.

Die Semantik dieser Funktionen hatten wir durch die Tabelle in Abbildung 4.1 auf Seite 32 gegeben.

Definition 37 (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. $\mathcal{S}(\mathcal{I}, F \wedge G) \;:=\; \circledwedge\big(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G)\big)$.
4. $\mathcal{S}(\mathcal{I}, F \vee G) \;:=\; \circledvee\big(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G)\big)$.
5. $\mathcal{S}(\mathcal{I}, F \rightarrow G) \;:=\; \ominus\!\!\!\rightarrow\big(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G)\big)$.
6. $\mathcal{S}(\mathcal{I}, F \leftrightarrow G) \;:=\; \ominus\!\!\!\leftrightarrow\big(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G)\big)$.
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

$$\mathcal{Z}\big(\mathcal{I}, \forall \mathtt{x} : \mathtt{e} * x = x\big) = \texttt{True}.$$ ◇

Definition 38 (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 39 (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 40 (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 93 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.

```python
def perfect(n):
    return sum({ x for x in range(1, n) if n % x == 0 }) == n

def findOddPerfect():
    n = 1
    while True:
        if perfect(n):
            return n
        n += 2

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. $\diamond$

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 Σ_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.

```
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

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.

```
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`

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 96 gezeigten Code implementieren.

1. Das in Zeile 1 definierte Universum U besteht aus den beiden Zahlen 0 und 1.

```
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

```
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

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.

```
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

Als nächstes überlegen wir uns, wie wir prädikatenlogische Terme in einer solchen Struktur auswerten können. Abbildung 5.6 zeigt die Implementierung der Prozedur `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. 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`.

```
def evalFormula(F, S, I):
    U = S[0]   # universe
    if F[0] == '⊤': return True
    if F[0] == '⊥': return False
    if F[0] == '¬': return not evalFormula(F[1], S, I)
    if F[0] == '∧': return evalFormula(F[1], S, I) and evalFormula(F[2], S, I)
    if F[0] == '∨': return evalFormula(F[1], S, I) or evalFormula(F[2], S, I)
    if F[0] == '→': return not evalFormula(F[1], S, I) or evalFormula(F[2], S, I)
    if F[0] == '↔': return evalFormula(F[1], S, I) == evalFormula(F[2], S, I)
    if F[0] == '∀':
        x, G = F[1:]
        return all({ evalFormula(G, S, modify(I, x, c)) for c in U} )
    if F[0] == '∃':
        x, G = F[1:]
        return any({ evalFormula(G, S, modify(I, x, c)) for c in U} )
    return evalAtomic(F, S, I)
```

Figure 5.8: Die Funktion `evalFormula`.

Abbildung 5.8 auf Seite 98 zeigt die Implementierung der Funktion `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 36 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

$$\mathcal{S}(\mathcal{I}, \forall x\colon G) \;:=\; \begin{cases} \texttt{True} & \text{falls } \mathcal{S}(\mathcal{I}[x/c], G) = \texttt{True} \quad \text{für alle } c \in \mathcal{U} \text{ gilt;} \\ \texttt{False} & \text{sonst.} \end{cases}$$

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 99 gezeigt. Bei der Auswertung eines All-Quantors können wir ausnutzen, dass die Sprache *Python* den Quantor "$\forall$" 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

$$\mathcal{I}[x] \texttt{ = } c$$

abändern können.

```
def modify(I, x, c):
    I[x] = c
    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 100 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.

```
f"evalFormula({G1}, S, I) = {evalFormula(F1, S, I)}"
f"evalFormula({G2}, S, I) = {evalFormula(F2, S, I)}"
f"evalFormula({G3}, S, I) = {evalFormula(F3, S, I)}"
f"evalFormula({G4}, S, I) = {evalFormula(F4, S, I)}"
```

Figure 5.10: Überprüfung, ob die in Abbildung 5.5 definierte Struktur eine Gruppe ist

```
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

Bemerkung: Das oben vorgestellte Programm finden sie als Jupyter Notebook auf GitHub unter der Adresse:

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. $\diamond$

Aufgabe 10:

1. Zeigen Sie, dass die Formel

   $$\forall x : \exists y : p(x,y) \rightarrow \exists y : \forall x : p(x,y)$$

   <u>nicht</u> 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 <u>erfüllbare</u> 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. $\diamond$

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 first order logic structure $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ (later abbreviated as FOL structure) 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 41 (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 FOL *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 Ω, 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.

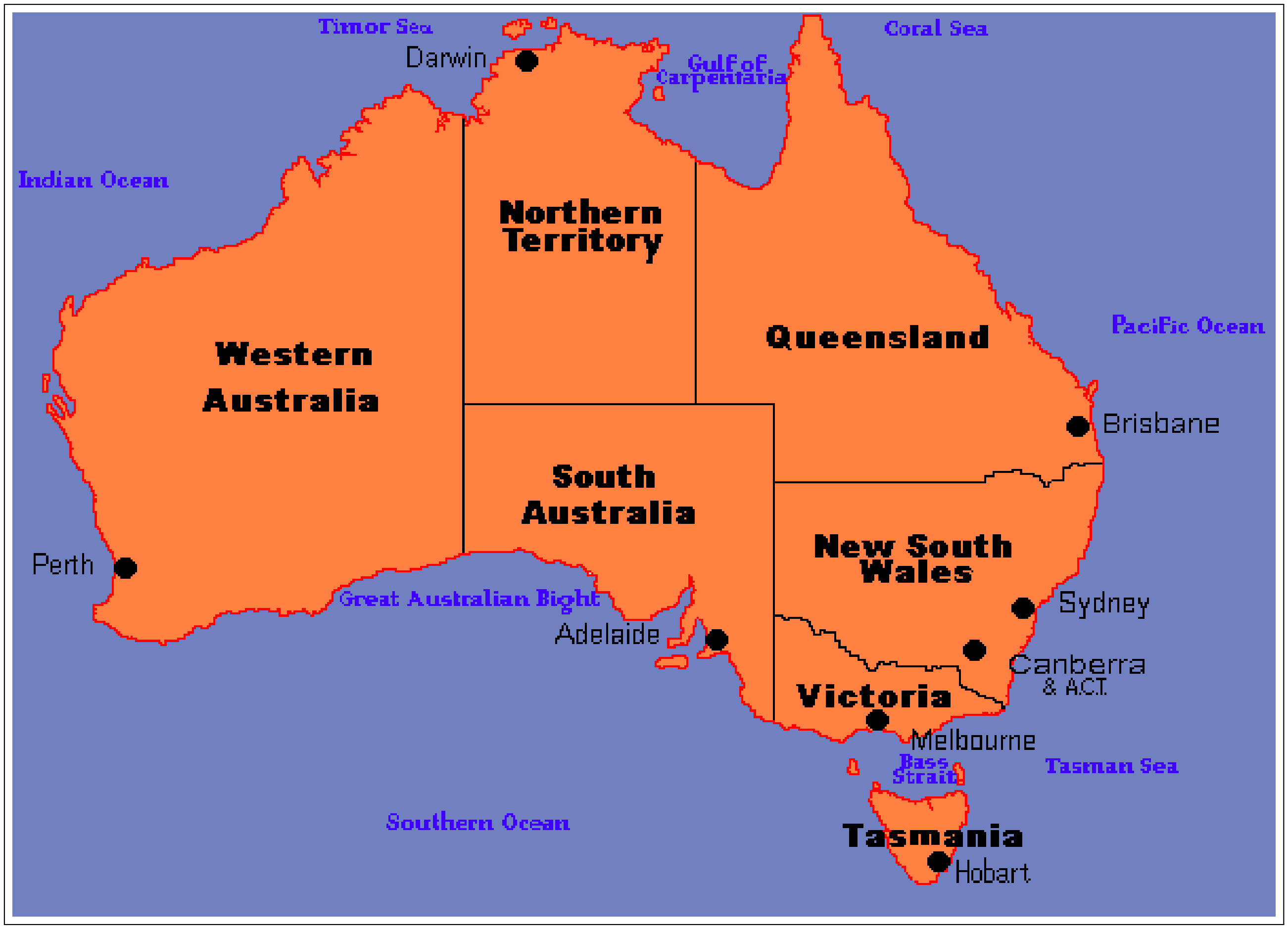

Figure 5.12: A map of Australia.

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 102 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 the three colours red, green, and blue 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. $\texttt{Constraints} :=$
$$\{\texttt{WA} \neq \texttt{NT}, \texttt{WA} \neq \texttt{SA}, \texttt{NT} \neq \texttt{SA}, \texttt{NT} \neq \texttt{Q}, \texttt{SA} \neq \texttt{Q}, \texttt{SA} \neq \texttt{NSW}, \texttt{SA} \neq \texttt{V}, \texttt{Q} \neq \texttt{NSW}, \texttt{NSW} \neq \texttt{V}\}.$$
The constraints do not mention the variable T for Tasmania, as Tasmania does not share a common border with any of the other states.

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. $\mathcal{I}(\texttt{WA}) = \texttt{red}$,
2. $\mathcal{I}(\texttt{NT}) = \texttt{blue}$,
3. $\mathcal{I}(\texttt{SA}) = \texttt{green}$,
4. $\mathcal{I}(\texttt{Q}) = \texttt{red}$,
5. $\mathcal{I}(\texttt{NSW}) = \texttt{blue}$,
6. $\mathcal{I}(\texttt{V}) = \texttt{red}$,
7. $\mathcal{I}(\texttt{T}) = \texttt{green}$,

then you can check that the assignment $\mathcal{I}$ is indeed a solution to the constraint satisfaction problem $\mathcal{P}$. Figure 5.13 on page 104 shows this solution.

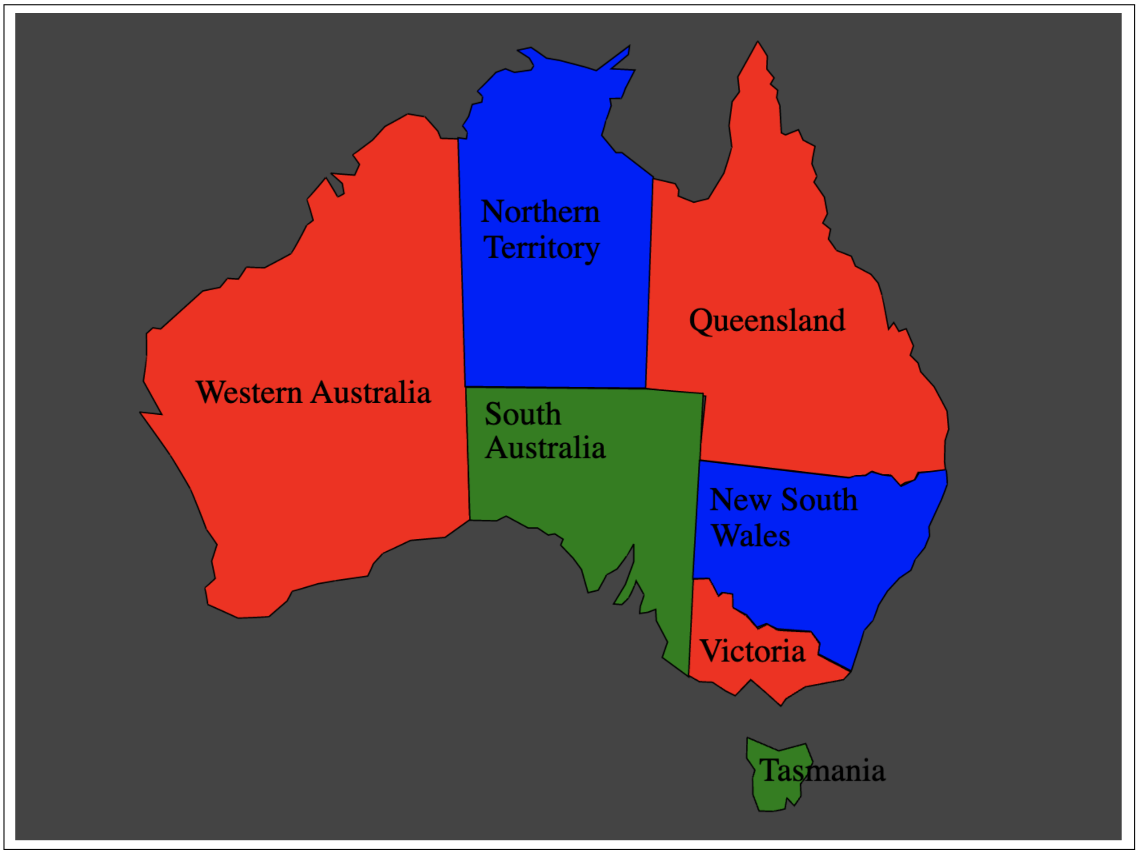

Figure 5.13: A map coloring for Australia.

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. We have already discussed this problem in the previous chapter. Let us recapitulate: 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

$$\texttt{SameColumn} := \big\{ \texttt{Q}_i \neq \texttt{Q}_j \bigm| i \in \{1, \cdots, 8\} \wedge j \in \{1, \cdots, 8\} \wedge j < i \big\}.$$

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

$$\texttt{SameDiagonal} := \big\{ |i - j| \neq |\texttt{Q}_i - \texttt{Q}_j| \bigm| i \in \{1, \cdots, 8\} \wedge j \in \{1, \cdots, 8\} \wedge j < i \big\}.$$

Then, the set of constraints is defined as

$$\texttt{Constraints} := \texttt{SameColumn} \cup \texttt{SameDiagonal}$$

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 5.14 on page 106.

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 5.15 on page 106.

Figure 5.16 on page 107 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 constraints that mention only the variable x_1 and check whether these constraints are satisfied. If any

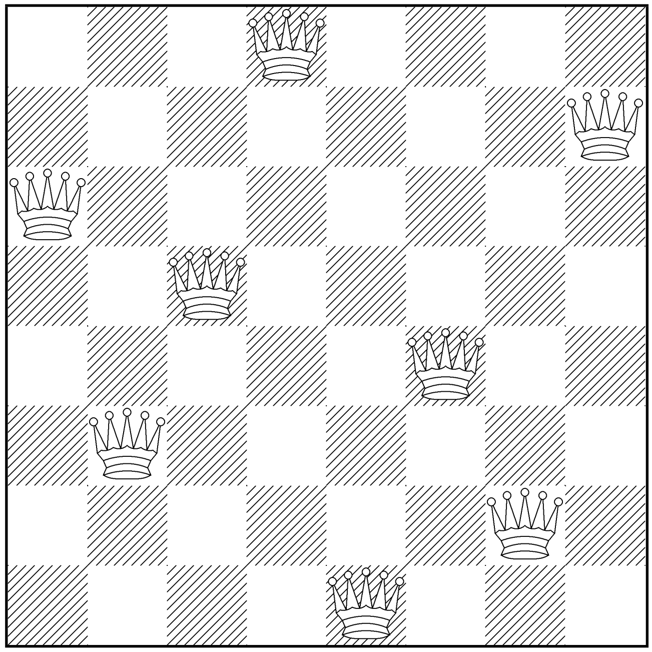

Figure 5.14: A solution of the eight queens problem.

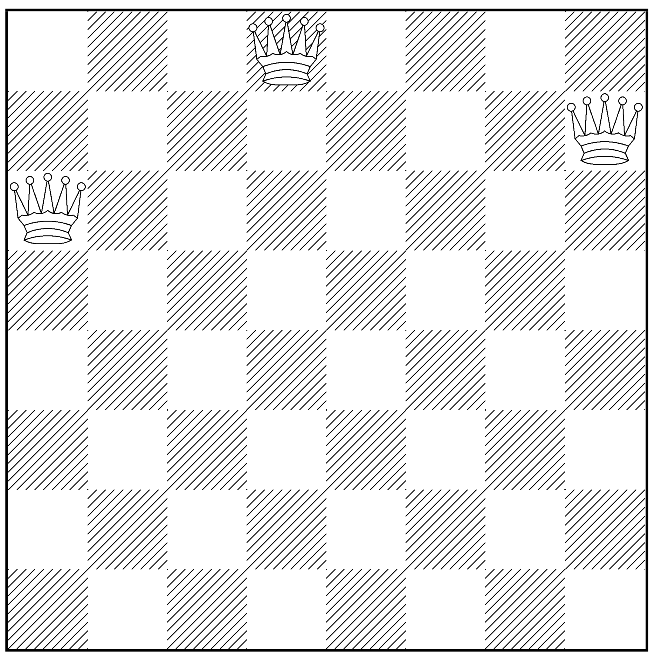

Figure 5.15: The partial assignment $\{\texttt{Q}_1 \mapsto 4, \texttt{Q}_2 \mapsto 8, \texttt{Q}_3 \mapsto 1)\}$.

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

$$\mathcal{B} = \{x_1 : v_1, \cdots, x_k : v_k\}$$

```python
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 }
    SameColumn   = { 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, SameColumn | SameDiagonal)
```

Figure 5.16: *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 5.17 on page 108 shows a simple CSP solver that employs backtracking. We discuss this program next.

1. As we need to determine the variables occurring in a given constraint, we import the module `ast`. This module implements the function `parse`(e) that takes a *Python* expression e. This expression is parsed and the resulting syntax tree is returned.

2. The function `collect_variables(expr)` takes a *Python* expression as its input. It returns the set of variable names occurring in this expression.

3. The procedure `solve` takes a constraint satisfaction problem `CSP` as input and tries to find a solution.

   (a) First, in line 11 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 tremendously.

   (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 `collect_variables`. 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.

```
import ast

def collect_variables(expr):
    tree = ast.parse(expr)
    return { node.id for node in ast.walk(tree)
                     if  isinstance(node, ast.Name)
                     if  node.id not in dir(__builtins__)
           }

def solve(CSP):
    'Compute a solution for the given constraint satisfaction problem.'
    Variables, Values, Constraints = CSP
    CSP = (Variables,
           Values,
           [(f, collect_variables(f) & set(Variables)) for f in Constraints]
          )
    return backtrack_search({}, CSP)
```

Figure 5.17: A backtracking CSP solver

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`.

Next, we discuss the implementation of the procedure `backtrack_search` that is shown in Figure 5.18 on page 109. 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`.

```
   def backtrack_search(Assignment, CSP):
2      '''
3      Given a partial variable assignment, this function tries to
4      complete this assignment towards a solution of the CSP.
5      '''
6      Variables, Values, Constraints = CSP
7      if len(Assignment) == len(Variables):
8          return Assignment
9      var = [x for x in Variables if x not in Assignment][0]
10     for value in Values:
11         if isConsistent(var, value, Assignment, Constraints):
12             NewAssign      = Assignment.copy()
13             NewAssign[var] = value
14             Solution = backtrack_search(NewAssign, CSP)
15             if Solution != None:
16                 return Solution
17     return None
```

Figure 5.18: The function `backtrack_search`

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`

    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 `for`-loop in line 10 tries the next `value`. If all possible values have been tried and none was successful, the `for`-loop ends and the function returns `None`.

```
def isConsistent(var, value, Assignment, Constraints):
    NewAssign      = Assignment.copy()
    NewAssign[var] = value
    return all(eval(f, NewAssign) for (f, Vs) in Constraints
                                  if var in Vs and Vs <= NewAssign.keys()
              )
```

Figure 5.19: The procedure `isConsistent`

We still need to discuss the implementation of the auxiliary procedure `isConsistent` shown in Figure 5.19 on page 109. 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 check, whether the extended assignment

$$\texttt{NA := Assignment} \cup \{\langle \texttt{var}, \texttt{value} \rangle\}$$

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 Solving Search Problems by Constraint Programming

In this section we show how we can formulate certain search problems as `CSP`s. We will explain our method by solving the missionaries and cannibals problem, which is explained in the following: Three missionaries and three infidels have to cross a river in order to get to a church where the infidels can be baptized. According to ancient catholic mythology, baptizing the infidels is necessary to save them from the eternal tortures of hell fire. In order to cross the river, the missionaries and infidels have a small boat available that can take at most two passengers. If at any moments at any shore there are more infidels than missionaries, then the missionaries have a problem, since the infidels have a diet that is rather unhealthy for the missionaries.

In order to solve this problem via constraint programming, we first introduce the notion of a symbolic transition system.

Definition 42 (Symbolic Transition System) *A symbolic transition system is a 6-tuple*

$$\mathcal{T} = \langle \texttt{Vars}, \texttt{Values}, \texttt{Start}, \texttt{Goal}, \texttt{Invariant}, \texttt{Transition} \rangle$$

such that:

(a) *`Vars` is a set of variables.*

These variables are strings. For every variable $x \in$ `Vars` there is a primed variable x' which does not occur in `Vars`. The set of these primed Variables is denoted as `Vars`$'$.

(b) *`Values` is a set of values that these variables can take.*

(c) *`Start`, `Goal`, and `Invariant` are* FOL *formulas such that all free variables occurring in these formulas are elements from the set `Vars`.*

- *`Start` describes the initial state of the transition system.*
- *`Goal` describes a state that should be reached by the transition system.*
- *`Invariant` is a formula that has to be true for every state of the transition system.*

(d) *`Transition` is a* FOL *formula. The free variables of this formula are elements of the set `Vars` $\cup$ `Vars`$'$, i.e. they are either variables from the set `Vars` or they are primed variables from the set `Vars`$'$.*

The formula `Transition` describes how the variables in the transition system change during a state transition. The primed variables refer to the values of the original variables after the state transition.

Every state of a transition system is a mapping of the variable to values. The idea is that the formula `Start` describes the start state of our search problem, `Goal` describes the state that we want to reach, while `Invariant` is a formula that must be true initially and that has to remain true after every transition of our system.

In order to clarify this definition we show how the *missionaries and cannibals* problem can be formulated as a symbolic transition system.

(a) $\texttt{Vars} := \{\texttt{M},\texttt{C},\texttt{B}\}$.

`M` is the number of missionaries on the western shore, `C` is the number of infidels on that shore, while `B` is the number of boats.

(b) $\texttt{Values} := \{0,1,2,3\}$.

(c) $\texttt{Start} := (\texttt{M} = 3 \wedge \texttt{C} = 3 \wedge \texttt{B} = 1)$.

(d) $\texttt{Goal} := (\texttt{M} = 0 \wedge \texttt{C} = 0 \wedge B = 0)$.

(e) $\texttt{Invariant} := \big((\texttt{M} = 3 \vee \texttt{M} = 0 \vee \texttt{M} = C) \ \wedge\ \texttt{B} \leq 1\big)$,

since there is no problem when all missionaries are either on the western shore or on the eastern shore or when the number of missionaries is the same as the number of infidels on the western shore, because then these numbers have to agree on the eastern shore as well.

Furthermore, there is just one boat.

(f)
$$\begin{array}{lll} \texttt{Transition} := & & \texttt{B}' = 1 - \texttt{B} \\ & \wedge & \big(\texttt{B} = 1 \ \rightarrow\ 1 \leq \texttt{M} - \texttt{M}' + \texttt{C} - \texttt{C}' \leq 2 \ \wedge\ \texttt{M}' \leq \texttt{M} \ \wedge\ \texttt{C}' \leq \texttt{C}\big) \\ & \wedge & \big(\texttt{B} = 0 \ \rightarrow\ 1 \leq \texttt{M}' - \texttt{M} + \texttt{C}' - \texttt{C} \leq 2 \ \wedge\ \texttt{M}' \geq \texttt{M} \ \wedge\ \texttt{C}' \geq \texttt{C}\big) \end{array}$$

Let us explain the details of this formula:

- $\texttt{B}' = 1 - \texttt{B}$
  If the boat is initially on the western shore, i.e. $\texttt{B} = 1$, it will be on the eastern shore afterwards, i.e. we will then have $\texttt{B}' = 0$. If, instead, the boat is initially on the eastern shore, i.e. $\texttt{B} = 0$, it will be on the western shore afterwards and then we have $\texttt{B}' = 1$.
- $\texttt{B} = 1 \;\rightarrow\; 1 \leq \texttt{M} - \texttt{M}' + \texttt{C} - \texttt{C}' \leq 2 \,\wedge\, \texttt{M}' \leq \texttt{M} \,\wedge\, \texttt{C}' \leq \texttt{C}$
  If the boat is initially on the western shore, then afterwards the number of missionaries and infidels will decrease, as they leave for the eastern shore. In this case $\texttt{M} - \texttt{M}'$ is the number of missionaries on the boat, while $\texttt{C} - \texttt{C}'$ is the number of infidels. The sum of these numbers has to be between 1 and 2 because the boat can not travel empty and can take at most two passengers.
- $\texttt{B} = 0 \;\rightarrow\; 1 \leq \texttt{M}' - \texttt{M} + \texttt{C}' - \texttt{C} \leq 2 \,\wedge\, \texttt{M}' \geq \texttt{M} \,\wedge\, \texttt{C}' \geq \texttt{C}$
  This formula describes the transition from the eastern shore to the western shore and is analogous to the previous formula.

```
def start(M, C, B):
    return M == 3 and C == 3 and B == 1

def goal(M, C, B):
    return M == 0 and C == 0 and B == 0

def invariant(M, C, B):
    return (M == 0 or M == 3 or M == C) and B <= 1

def transition(Mα, Cα, Bα, Mβ, Cβ, Bβ):
    if not (Bβ == 1 - Bα):
        return False
    if Bα == 1:
        return 1 <= Mα - Mβ + Cα - Cβ <= 2 and Mβ <= Mα and Cβ <= Cα
    else:
        return 1 <= Mβ - Mα + Cβ - Cα <= 2 and Mβ >= Mα and Cβ >= Cα
```

Figure 5.20: Coding the *missionaries and cannibals problem* as a symbolic transition system.

Figure 5.20 shows how the *missionaries and cannibals problem* can be represented as a symbolic transition system in *Python*. In the function `transition` we use the following convention: Since variables cannot be primed in *Python* we append the character α to the names of the original variables from the set `Vars`, while we append β to these names to get the primed versions of the corresponding variable.

Figure 5.21 shows how we can turn the symbolic transition system into a CSP.

1. The function `flatten(LoL)` receives a list `LoL` of lists as its argument. This list has the form

   $$\texttt{LoL} = [L_1, \cdots, L_k]$$

   where the L_i are lists for $i = 1, \cdots, k$.

```
1   def flatten(LoL):
2       return [x for L in LoL for x in L]
3
4   def missionaries_CSP(n):
5       "Returns a CSP encoding the problem."
6       Lists        = [[f'M{i}', f'C{i}', f'B{i}'] for i in range(n+1)]
7       Variables    = flatten(Lists)
8       Values       = { 0, 1, 2, 3 }
9       Constraints  = {  'start(M0, C0, B0)'      }  # start state
10      Constraints |= { f'goal(M{n}, C{n}, B{n})' }  # goal state
11      for i in range(n):
12          Constraints.add(f'invariant(M{i}, C{i}, B{i})')
13          Constraints.add(f'transition(M{i}, C{i}, B{i}, M{i+1}, C{i+1}, B{i+1})')
14      return Variables, Values, Constraints
15
16  def find_solution():
17      n = 1
18      while True:
19          print(n)
20          CSP = missionaries_CSP(n)
21          Solution = solve(CSP)
22          if Solution != None:
23              return n, Solution
24          n += 2
```

Figure 5.21: Turning the symbolic transition system into a CSP.

It returns the list

$$L_1 + \cdots + L_k,$$

i.e. it appends these lists and returns the result.

2. The function `missionaries_CSP`(n) receives a natural number n as its argument. It returns a CSP that has a solution if there is a solution of the *missionaries and cannibals* problem that crosses the river exactly n times. It uses the variables

   $\texttt{M}_i$, $\texttt{C}_i$, and $\texttt{B}_i$, where $i = 0, \cdots, n$.

   $\texttt{M}_i$ is the number of missionaries on the western shore after the boat has crossed the river i times. The variables $\texttt{C}_i$ and $\texttt{B}_i$ denote the number of infidels and boats respectively.

   Line 12 ensures that the invariant of the transition system is valid after every crossing of the boat. Line 13 describes the mechanics of the crossing.

3. The function `find_solution` tries to find a natural number n such that problem can be solved with n crossings. As the number off crossings has to be odd, we increment n by two.

5.6 Z3

We conclude this chapter with a discussion of the solver Z3. Z3 implements most of the state-of-the-art constraint solving algorithms and is exceptionally powerful. We introduce Z3 via a series of examples.

5.6.1 A Simple Text Problem

The following is a simple text problem from my old 8th grade math book.

- *I have as many brothers as I have sisters.*
- *My sister has twice as many brothers as she has sisters.*
- *How many children does my father have?*

However, in order to solve this puzzle we need two additional assumptions.

1. My father has no illegitimate children.
2. All of my fathers children identify themselves as either male or female.

Strangely, in my old math book these assumptions are not mentioned.

We can now infer the number of children. If we denote the number of boys with the variable b and the number of girls with g, the problem statements are equivalent to the following two equations:

(a) $b - 1 = g$.

(b) $2 \cdot (g - 1) = b$.

Before we can start to solve this problem, we have to install `Z3` via `pip` using the following command:

```
pip install z3-solver
```

Figure 5.22 on page 115 shows how we can solve the given problem using the *Python* interface of Z3.

1. In line 1 we import the module `z3` so that we can use the Python API of Z3. The documentation of this API is available at the following address:

   https://ericpony.github.io/z3py-tutorial/guide-examples.htm

2. Lines 3 and 4 creates the `Z3` variables `boys` and `girls` as integer valued variables. The function `Int` takes one argument, which has to be a string. This string is the name of the variable. We store these variables in Python variables of the same name. It would be possible to use different names for the Python variables, but that would be very confusing.
3. Line 6 creates an object of the class `Solver`. This is the constraint solver provided by `Z3`.
4. Lines 8 and 9 add the constraints expressing that the number of girls is one less than the number of boys and that my sister has twice as many brothers as she has sisters as constraints to the solver `S`.

```
1  import z3
2
3  boys  = z3.Int('boys')
4  girls = z3.Int('girls')
5
6  S = z3.Solver()
7
8  S.add(boys - 1 == girls)
9  S.add(2 * (girls - 1) == boys)
10 S.check()
11 Solution = S.model()
12
13 b = Solution[boys ].as_long()
14 g = Solution[girls].as_long()
15
16 print(f'My father has {b + g} children.')
```

Figure 5.22: Solving a simple text problem.

5. In line 10 the method `check` examines whether the given set of constraints is satisfiable. In general, this method returns one of the following results:

   (a) `sat` is returned if the problem is solvable, (`sat` is short for *satisfiable*)

   (b) `unsat` is returned if the problem is unsolvable,

   (c) `unknown` is returned if `Z3` is not powerful enough to solve the given problem.

6. Since in our case the method `check` returns `sat`, we can extract the solution that is computed via the method `model` in line 11.

7. In order to extract the values that have been computed by `Z3` for the variables `boys` and `girls`, we can use dictionary syntax and write `Solution[boys]` and `Solution[girls]` to extract these values. However, these values are not stored as integers but rather as objects of the class `IntNumRef`, which is some internal class of `Z3` to store integers. This class provides the method `as_long` that converts its argument into an integer number.

Exercise 11: Solve the following text problem using Z3.

(a) *A Japanese deli offers both penguins and parrots.*

(b) *A parrot and a penguin together cost 666 bucks.*

(c) *The penguin costs 600 bucks more than the parrot.*

What is the price of the parrot? You may assume that the prizes of these delicacies are integer valued. ◇

Exercise 12: Solve the following text problem using Z3.

(a) *A train travels at a uniform speed for 360 miles.*

(b) *The train would have taken 48 minutes less to travel the same distance if it had been faster by 5 miles per hour.*

Find the speed of the train!
Hints:

(a) As the speed is a real number you should declare this variable via the Z3 function `Real` instead of using the function `Int`.

(b) Be careful to not mix up different units. In particular, the time 48 minutes should be expressed as a fraction of an hour.

(c) When you formulate the information given above, you will get a system of **non-linear** equations, which is equivalent to a quadratic equation. This quadratic equation has two different solutions. One of these solutions is negative. In order to exclude the negative solution you need to add a constraint stating that the speed of the train has to be greater than zero.

5.6.2 The Knight's Tour

In this subsection we will solve the puzzle The Knight's Tour using Z3. This puzzle asks whether it is possible for a knight to visit all 64 squares of a chess board in 63 moves. We will start the tour in the upper left corner of the board.

In order to model this puzzle as a constraint satisfaction problem we first have to decide on the variables that we want to use. The idea is to have 64 variables that describe the position of the knight after its i^{th} move where $i = 0, 1, \cdots, 63$. However, it turns out that it is best to split the values of these positions up into a row and a column. If we do this, we end up with 128 variables of the form

$$\texttt{R}_i \text{ and } \texttt{C}_i \quad \text{for } i \in \{0, 1, \cdots, 63\}.$$

Here $\texttt{R}_i$ denotes the row of the knight after its i^{th} move, while $\texttt{C}_i$ denotes the corresponding column. Next, we have to formulate the constraints. In this case, there are two kinds of constraints:

1. We have to specify that the move from the position $\langle \texttt{R}_i, \texttt{C}_i \rangle$ to the position $\langle \texttt{R}_{i+1}, \texttt{C}_{i+1} \rangle$ is legal move for a knight. In chess, there are two ways for a knight to move:

(a) The knight can move two squares horizontally left or right followed by moving vertically one square up or down, or
(b) the knight can move two squares vertically up or down followed by moving one square left or right.

Figure 5.23 shows all legal moves of a knight that is positioned in the square `e4`. Therefore, a formula that expresses that the i^{th} move is a legal move of the knight is a disjunction of the following eight formulas that each describe one possible way for the knight to move:

(a) $\mathtt{R}_{i+1} = \mathtt{R}_i + 2 \;\wedge\; \mathtt{C}_{i+1} = \mathtt{C}_i + 1$,
(b) $\mathtt{R}_{i+1} = \mathtt{R}_i + 2 \;\wedge\; \mathtt{C}_{i+1} = \mathtt{C}_i - 1$,
(c) $\mathtt{R}_{i+1} = \mathtt{R}_i - 2 \;\wedge\; \mathtt{C}_{i+1} = \mathtt{C}_i + 1$,
(d) $\mathtt{R}_{i+1} = \mathtt{R}_i - 2 \;\wedge\; \mathtt{C}_{i+1} = \mathtt{C}_i - 1$,
(e) $\mathtt{R}_{i+1} = \mathtt{R}_i + 1 \;\wedge\; \mathtt{C}_{i+1} = \mathtt{C}_i + 2$,
(f) $\mathtt{R}_{i+1} = \mathtt{R}_i + 1 \;\wedge\; \mathtt{C}_{i+1} = \mathtt{C}_i - 2$,
(g) $\mathtt{R}_{i+1} = \mathtt{R}_i - 1 \;\wedge\; \mathtt{C}_{i+1} = \mathtt{C}_i + 2$,
(h) $\mathtt{R}_{i+1} = \mathtt{R}_i - 1 \;\wedge\; \mathtt{C}_{i+1} = \mathtt{C}_i - 2$.

2. Furthermore, we have to specify that the position $\langle \mathtt{R}_i, \mathtt{C}_i \rangle$ is different from the position $\langle \mathtt{R}_j, \mathtt{C}_j \rangle$ if $i \neq j$.

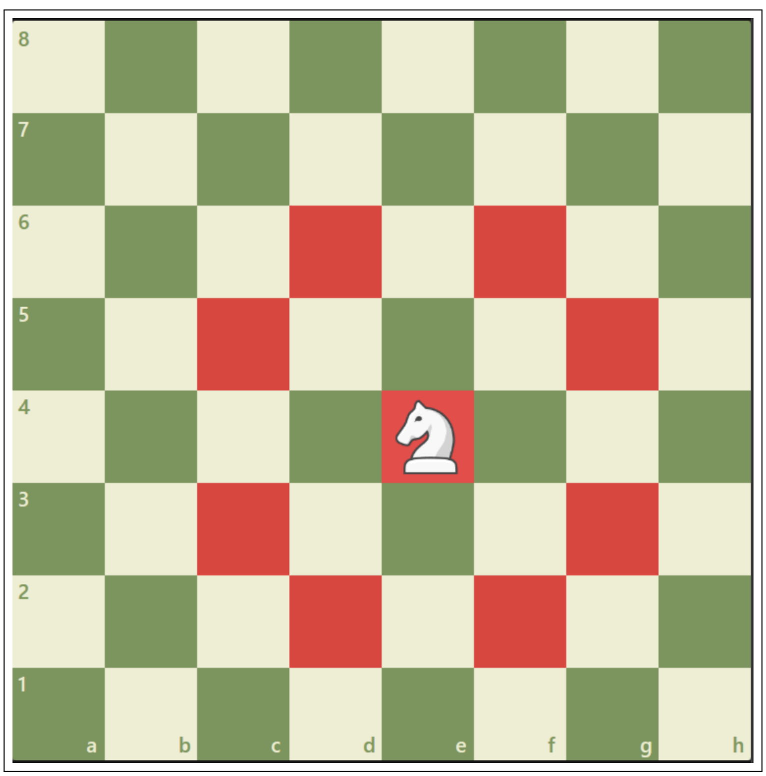

Figure 5.23: The moves of a knight, courtesy of chess.com.

Figure 5.24 shows how we can formulate the puzzle using `Z3`.

```
1  import z3
2
3  def row(i): return f'R{i}'
4  def col(i): return f'C{i}'
5
6  def is_knight_move(row, col, rowX, colX):
7      Formulas = set()
8      for delta_r, delta_c in [(1, 2), (2, 1)]:
9          Formulas.add(z3.And(rowX == row + delta_r, colX == col + delta_c))
10         Formulas.add(z3.And(rowX == row + delta_r, colX + delta_c == col))
11         Formulas.add(z3.And(rowX + delta_r == row, colX == col + delta_c))
12         Formulas.add(z3.And(rowX + delta_r == row, colX + delta_c == col))
13     return z3.Or(*Formulas)
14
15 def all_different(Rows, Cols):
16     Result = set()
17     for i in range(62+1):
18         for j in range (i+1, 63+1):
19             Result.add(z3.Or(Rows[i] != Rows[j], Cols[i] != Cols[j]))
20     return Result
21
22 def all_constraints(Rows, Cols):
23     Constraints = all_different(Rows, Cols)
24     Constraints.add(Rows[0] == 0)
25     Constraints.add(Cols[0] == 0)
26     for i in range(62+1):
27         Constraints.add(is_knight_move(Rows[i], Cols[i], Rows[i+1], Cols[i+1]))
28     for i in range(63+1):
29         Constraints.add(Rows[i] >= 0)
30         Constraints.add(Cols[i] >= 0)
31     return Constraints
```

Figure 5.24: The Knight's Tour: Computing the constraints.

1. In line 1 we import the library `z3`.
2. We define the auxiliary functions `row` and `col` in line 3 and 4. Given a natural number i, the expression `row`(i) returns the string `'R`i`'` and `col`(i) returns the string `'C`i`'`. These strings in turn represent the variables $\mathtt{R}_i$ and $\mathtt{C}_i$.
3. The function `is_knight_move` takes four parameters:
   (a) `row` is a `Z3` variable that specifies the row of the position of the knight before the move.
   (b) `col` is a `Z3` variable that specifies the column of the position of the knight before the move.
   (c) `rowX` is a `Z3` variable that specifies the row of the position of the knight after the move.

(d) `colX` is a `Z3` variable that specifies the column of the position of the knight after the move.

The function checks whether the move from position $\langle \texttt{R}_i, \texttt{C}_i \rangle$ to the position $\langle \texttt{R}_{i+1}, \texttt{C}_{i+1} \rangle$ is a legal move for a knight. In line 13 we use the fact that the function `z3.Or` can take any number of arguments. If `Formulas` is the set

$$\texttt{Formulas} = \{f_1, \cdots, f_n\},$$

then the notation `z3.Or(*Formulas)` is expanded into the call

$$\texttt{z3.Or}(f_1, \cdots, f_n),$$

which computes the logical disjunction

$$f_1 \vee \cdots \vee f_n.$$

4. The function `all_different` takes two parameters:

   (a) `Rows` is a list of `Z3` variables. The `Z3` variable `Rows[i]` specifies the row of the position of the knight after the i^{th} move.

   (b) `Cols` is a list of `Z3` variables. The `Z3` variable `Cols[i]` specifies the column of the position of the knight after the i^{th} move.

   The function computes a set of formulas that state that the positions $\langle \texttt{R}_i, \texttt{C}_i \rangle$ for $i = 0, 1 \cdots, 63$ are all different from each other. Note that the position $\langle \texttt{R}_i, \texttt{C}_i \rangle$ is different from the position $\langle \texttt{R}_j, \texttt{C}_j \rangle$ iff R_i is different from R_j or C_i is different from C_j.

5. The function `all_constraints` computes the set of all constraints. The parameters for this function are the same as those for the function `all_different`. In addition to the constraints already discussed this function specifies that the knight starts its tour at the upper left corner of the board.

   Furthermore, there are constraints that the variables $\texttt{R}_i$ and $\texttt{C}_i$ are all non-negative. These constraints are needed as we will model the variables with bit vectors of length 4. These bit vectors store integers in two's complement representation. In two's complement representation of a bit vector of length 4 we can model integers from the set $\{-8, \cdots, 7\}$. If we add the number 1 to a 4-bit bit vector v that represents the number 7, then an overflow will occur and the result will be -8 instead of 8. This could happen in the additions that are performed in the formulas computed by the function `is_knight_move`. We can exclude these cases by adding the constraints that all variables are non-negative.

Finally, the function `solve` that is shown in Figure 5.25 on page 120 can be used to solve the puzzle. The purpose of the function `solve` is to construct a CSP encoding the puzzle and to find a solution of this CSP unsing `Z3`. If successful, it returns a dictionary that maps every variable name to the corresponding value of the solution that has been found.

1. In line 2 and 3 we create the `Z3` variables that specify the positions of the knight after its i^{th} move. $\texttt{Rows}[i]$ specifies the row of the knight after the its i^{th} move, while $\texttt{Cols}[i]$ specifies the column.

2. We compute the set of all constraints in line 4.

3. We create a solver object in line 5 and add the constraints to this solver in the following line.

```
1  def solve():
2      Rows = [z3.BitVec(row(i), 4) for i in range(63+1)]
3      Cols = [z3.BitVec(col(i), 4) for i in range(63+1)]
4      Constraints = all_constraints(Rows, Cols)
5      S = z3.Solver()
6      S.add(Constraints)
7      result = str(S.check())
8      if result == 'sat':
9          Model    = S.model()
10         Solution = (  { row(i): Model[Rows[i]] for i in range(63+1) }
11                     | { col(i): Model[Cols[i]] for i in range(63+1) })
12         return Solution
13     elif result == 'unsat':
14         print('The problem is not solvable.')
15     else:
16         print('Z3 cannot determine whether the problem is solvable.')
```

Figure 5.25: The function `solve`.

0	37	40	51	2	17	42	25
39	50	1	18	41	24	3	16
36	19	38	23	52	55	26	43
49	22	53	30	57	44	15	4
20	35	58	45	54	31	56	27
59	48	21	10	29	12	5	14
34	9	46	61	32	7	28	63
47	60	33	8	11	62	13	6

Figure 5.26: A solution of the knight's problem.

4. The function `check` tries to build a model satisfying the constraints, while the function `model` extracts this model if it exists.

5. Finally, in line 10 and 11 we create a dictionary that maps all of our variables to the correspond-

ing values that are found in the model. Note that `row`(i) returns the name of the `Z3` variable `Rows`$[i]$ and similarly `col`(i) returns the name of the `Z3` variable `Cols`$[i]$. This dictionary is then returned.

Figure 5.26 on page 120 shows a solution that has been computed by the program discussed above.

	3	9						7
			7			4	9	2
				6	5		8	3
			6		3	2	7	
				4		8		
5	6							
		5	2		9			1
	2	1					4	
7						5		

Table 5.1: A super hard sudoku from the magazine "Zeit Online".

Exercise 13: Table 5.1 on page 121 shows a sudoku that I have taken from the Zeit Online magazine. Solve this sudoku using `Z3`. You should start with the following file:

https://github.com/karlstroetmann/Logic/blob/master/Python/Chapter-5/Sudoku-Z3.ipynb. ◇

5.7 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 43 *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*

$$\models (\forall x\colon f) \leftrightarrow f \quad \textit{und} \quad \models (\exists x\colon f) \leftrightarrow f.$$

8. *Falls x eine Variable ist, für die $x \notin FV(g)$ gilt, so haben wir die folgenden Äquivalenzen:*
   (a) $\models (\forall x\colon f) \vee g \leftrightarrow \forall x\colon (f \vee g) \quad \textit{und} \quad \models g \vee (\forall x\colon f) \leftrightarrow \forall x\colon (g \vee f)$,
   (b) $\models (\exists x\colon f) \wedge g \leftrightarrow \exists x\colon (f \wedge g) \quad \textit{und} \quad \models g \wedge (\exists x\colon f) \leftrightarrow \exists x\colon (g \wedge f)$.

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 $\mathcal{S}_2$ 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 44 (Skolemisierung)
Es sei $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity\rangle$ eine Signatur. Ferner sei f eine geschlossene Σ-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 Σ nicht auftritt und erweitern die Signatur Σ 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 Σ'-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 45 (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 hervorgegangen 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. ◇

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.

Das Jupyter Notebook

https://github.com/karlstroetmann/Logic/blob/master/Python/Chapter-5/FOL-CNF.ipynb.

enthält ein *Python*-Programm, mit dessen Hilfe wir prädikatenlogische Formeln in eine erfüllbarkeits-äquivalente Menge von prädikatenlogischen Klauseln umformen können.

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

$$\models \big(\exists x\colon \forall y\colon p(x,y)\big) \rightarrow \big(\forall y\colon \exists x\colon p(x,y)\big)$$

gilt. Wir wissen, dass dies äquivalent dazu ist, dass

$$\Big\{\neg\Big(\big(\exists x\colon \forall y\colon p(x,y)\big) \rightarrow \big(\forall y\colon \exists x\colon p(x,y)\big)\Big)\Big\} \models \bot$$

gilt. Wir bringen zunächst die negierte Formel in pränexe Normalform.

$$\begin{array}{ll} & \neg\Big(\big(\exists x\colon \forall y\colon p(x,y)\big) \rightarrow \big(\forall y\colon \exists x\colon p(x,y)\big)\Big) \\ \leftrightarrow & \neg\Big(\neg\big(\exists x\colon \forall y\colon p(x,y)\big) \vee \big(\forall y\colon \exists x\colon p(x,y)\big)\Big) \\ \leftrightarrow & \big(\exists x\colon \forall y\colon p(x,y)\big) \wedge \neg\big(\forall y\colon \exists x\colon p(x,y)\big) \\ \leftrightarrow & \big(\exists x\colon \forall y\colon p(x,y)\big) \wedge \big(\exists y\colon \neg\exists x\colon p(x,y)\big) \\ \leftrightarrow & \big(\exists x\colon \forall y\colon p(x,y)\big) \wedge \big(\exists y\colon \forall x\colon \neg p(x,y)\big) \end{array}$$

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

$$\begin{array}{ll} & \big(\exists x\colon \forall y\colon p(x,y)\big) \wedge \big(\exists y\colon \forall x\colon \neg p(x,y)\big) \\ \leftrightarrow & \big(\exists x\colon \forall y\colon p(x,y)\big) \wedge \big(\exists v\colon \forall u\colon \neg p(u,v)\big) \\ \leftrightarrow & \exists v\colon \Big(\big(\exists x\colon \forall y\colon p(x,y)\big) \wedge \big(\forall u\colon \neg p(u,v)\big)\Big) \\ \leftrightarrow & \exists v\colon \exists x\colon \Big(\big(\forall y\colon p(x,y)\big) \wedge \big(\forall u\colon \neg p(u,v)\big)\Big) \\ \leftrightarrow & \exists v\colon \exists x\colon \forall y\colon \Big(p(x,y) \wedge \big(\forall u\colon \neg p(u,v)\big)\Big) \\ \leftrightarrow & \exists v\colon \exists x\colon \forall y\colon \forall u\colon \Big(p(x,y) \wedge \neg p(u,v)\Big) \end{array}$$

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.

$$\begin{array}{ll} & \exists v\colon \exists x\colon \forall y\colon \forall u\colon \Big(p(x,y) \wedge \neg p(u,v)\Big) \\ \approx_e & \exists x\colon \forall y\colon \forall u\colon \Big(p(x,y) \wedge \neg p(u,s_1)\Big) \\ \approx_e & \forall y\colon \forall u\colon \Big(p(s_2,y) \wedge \neg p(u,s_1)\Big) \end{array}$$

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

$$M := \Big\{\big\{p(s_2,y)\big\}, \big\{\neg p(u,s_1)\big\}\Big\}.$$

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

$$\Big\{\neg\Big(\big(\exists x\colon \forall y\colon p(x,y)\big) \rightarrow \big(\forall y\colon \exists x\colon p(x,y)\big)\Big)\Big\}$$

unerfüllbar und folglich gilt

$$\models \big(\exists x\colon \forall y\colon p(x,y)\big) \rightarrow \big(\forall y\colon \exists x\colon p(x,y)\big).$$

5.8 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 46 (Substitution) *Es sei eine Signatur*

$$\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \textit{arity} \rangle$$

gegeben. Eine Σ-Substitution ist eine endliche Menge von Paaren der Form

$$\sigma = \big\{\langle x_1, t_1\rangle, \cdots, \langle x_n, t_n\rangle\big\}.$$

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 $\sigma = \big\{\langle x_1, t_1\rangle, \cdots, \langle x_n, t_n\rangle\big\}$ eine Σ-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 σ 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 47 (Anwendung einer Substitution)
Es sei t ein Term und es sei $\sigma = [x_1 \mapsto t_1, \cdots, x_n \mapsto t_n]$ eine Substitution. Wir definieren die Anwendung von σ 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* $x_i\sigma := t_i$.
   (b) *$t = y$ mit $y \in \mathcal{V}$, aber $y \notin \{x_1, \cdots, x_n\}$. Dann definieren wir* $y\sigma := y$.
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. $\diamond$

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 σ durch
$$\sigma := [x_1 \mapsto c,\ x_2 \mapsto f(d)].$$
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. $\{p(x_2), q(d, h(x_3, x_1))\}\sigma = \{p(f(d)),\ q(d, h(x_3, c))\}$.

Als nächstes zeigen wir, wie Substitutionen miteinander verknüpft werden können.

Definition 48 (Komposition von Substitutionen) *Es seien*
$$\sigma = [x_1 \mapsto s_1, \cdots, x_m \mapsto s_m] \quad \text{und} \quad \tau = [y_1 \mapsto t_1, \cdots, y_n \mapsto t_n]$$
zwei Substitutionen mit $\mathrm{dom}(\sigma) \cap \mathrm{dom}(\tau) = \{\}$. Dann definieren wir die Komposition von Substitutionen $\sigma\tau$ von σ und τ als
$$\sigma\tau := [x_1 \mapsto s_1\tau, \cdots, x_m \mapsto s_m\tau,\ y_1 \mapsto t_1, \cdots, y_n \mapsto t_n]$$ $\diamond$

Beispiel: Wir führen das obige Beispiel fort und setzen
$$\sigma := [x_1 \mapsto c,\ x_2 \mapsto f(x_3)] \quad \text{und} \quad \tau := [x_3 \mapsto h(c, c),\ x_4 \mapsto d].$$
Dann gilt:

$$\sigma\tau = \big[x_1 \mapsto c,\ x_2 \mapsto f(h(c,c)),\ x_3 \mapsto h(c,c),\ x_4 \mapsto d\big]. \qquad \Box$$

Die Definition der Komposition von Substitutionen ist mit dem Ziel gewählt worden, dass der folgende Satz gilt.

Satz 49 *Ist t ein Term und sind σ und τ Substitutionen mit $dom(\sigma) \cap dom(\tau) = \{\}$, so gilt*

$$(t\sigma)\tau = t(\sigma\tau). \qquad \Box$$

Der Satz kann durch Induktion über den Aufbau des Termes t bewiesen werden.

Definition 50 (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 51 (Unifikator) *Eine Substitution σ löst eine syntaktische Gleichung $s \doteq t$ genau dann, wenn $s\sigma = t\sigma$ ist, wenn also durch die Anwendung von σ auf s und t tatsächlich identische Objekte entstehen. Ist E ein syntaktisches Gleichungs-System, so sagen wir, dass σ ein Unifikator von E ist wenn σ 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 σ 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 σ löst die obige syntaktische Gleichung, denn es gilt

$$\begin{aligned} p(x_1, f(x_4))\sigma &= p(x_2, f(x_4)) \quad \text{und} \\ p(x_2, x_3)\sigma &= p(x_2, f(x_4)). \end{aligned} \qquad \diamond$$

Als nächstes entwickeln wir ein Verfahren, mit dessen Hilfe wir von einer vorgegebenen Menge E von syntaktischen Gleichungen entscheiden können, ob es einen Unifikator σ 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 σ 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

$$x \doteq f(t_1, \cdots, t_n) \quad \text{falls } x \in Var\big(f(t_1, \cdots, t_n)\big).$$

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 τ 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 σ 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 σ 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 $y \in Var(t)$ ist und wenn außerdem $t \neq y$ ist, dann hat das Gleichungs-System $E \cup \{y \doteq t\}$ keine Lösung, wir schreiben

   $$\Big\langle E \cup \{y \doteq t\}, \sigma \Big\rangle \rightsquigarrow \Omega \quad \text{falls } y \in Var(t) \text{ und } y \neq t.$$

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:

$$\Big\langle E \cup \{x \doteq x\}, \sigma \Big\rangle \quad \leadsto \quad \Big\langle E, \sigma \Big\rangle.$$

5. Ist f ein n-stelliges Funktions-Zeichen, so gilt

$$\Big\langle E \cup \{f(s_1, \cdots, s_n) \doteq f(t_1, \cdots, t_n)\}, \sigma \Big\rangle \leadsto \Big\langle E \cup \{s_1 \doteq t_1, \cdots, s_n \doteq t_n\}, \sigma \Big\rangle.$$

   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

$$\Big\langle E \cup \{c \doteq c\}, \sigma \Big\rangle \leadsto \Big\langle E, \sigma \Big\rangle.$$

   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 $E \cup \{f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n)\}$ hat keine Lösung, falls die Funktions-Zeichen f und g verschieden sind, wir schreiben

$$\Big\langle E \cup \{f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n)\}, \sigma \Big\rangle \leadsto \Omega \qquad \text{falls } f \neq g.$$

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 Ω zurück gegeben.

2. Das Paar $\langle E, [] \rangle$ wird reduziert zu einem Paar $\langle \{\}, \sigma \rangle$. Dann ist σ 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:

$$\begin{aligned}
& \big\langle \{p(x_1, f(x_4)) \doteq p(x_2, x_3)\}, [] \big\rangle \\
\leadsto \; & \big\langle \{x_1 \doteq x_2, f(x_4) \doteq x_3\}, [] \big\rangle \\
\leadsto \; & \big\langle \{f(x_4) \doteq x_3\}, [x_1 \mapsto x_2] \big\rangle \\
\leadsto \; & \big\langle \{x_3 \doteq f(x_4)\}, [x_1 \mapsto x_2] \big\rangle \\
\leadsto \; & \big\langle \{\}, [x_1 \mapsto x_2,\; x_3 \mapsto f(x_4)] \big\rangle
\end{aligned}$$

In diesem Fall ist das Verfahren also erfolgreich und wir erhalten die Substitution

$$[x_1 \mapsto x_2,\; x_3 \mapsto f(x_4)]$$

als Lösung der oben gegebenen syntaktischen Gleichung. ◇

Beispiel: Wir geben ein weiteres Beispiel und betrachten das Gleichungs-System

$$E = \{p(h(x_1,c)) \doteq p(x_2),\; q(x_2,d) \doteq q(h(d,c),x_4)\}$$

Wir haben folgende Reduktions-Schritte:

$$\begin{array}{ll}
& \langle \{p(h(x_1,c)) \doteq p(x_2),\; q(x_2,d) \doteq q(h(d,c),x_4)\},\, []\rangle \\
\leadsto & \langle \{p(h(x_1,c)) \doteq p(x_2),\; x_2 \doteq h(d,c),\; d \doteq x_4\},\, []\rangle \\
\leadsto & \langle \{p(h(x_1,c)) \doteq p(x_2),\; x_2 \doteq h(d,c),\; x_4 \doteq d\},\, []\rangle \\
\leadsto & \langle \{p(h(x_1,c)) \doteq p(x_2),\; x_2 \doteq h(d,c)\},\, [x_4 \mapsto d]\rangle \\
\leadsto & \langle \{p(h(x_1,c)) \doteq p(h(d,c))\},\, [x_4 \mapsto d,\; x_2 \mapsto h(d,c)]\rangle \\
\leadsto & \langle \{h(x_1,c) \doteq h(d,c)\},\, [x_4 \mapsto d,\; x_2 \mapsto h(d,c)]\rangle \\
\leadsto & \langle \{x_1 \doteq d,\; c \doteq c\},\, [x_4 \mapsto d,\; x_2 \mapsto h(d,c)]\rangle \\
\leadsto & \langle \{x_1 \doteq d,\},\, [x_4 \mapsto d,\; x_2 \mapsto h(d,c)]\rangle \\
\leadsto & \langle \{\},\, [x_4 \mapsto d,\; x_2 \mapsto h(d,c),\; x_1 \mapsto d]\rangle
\end{array}$$

Damit haben wir die Substitution $[x_4 \mapsto d,\; x_2 \mapsto h(d,c),\; x_1 \mapsto d]$ als Lösung des anfangs gegebenen syntaktischen Gleichungs-Systems gefunden. ◇

Das Jupyter Notebook

https://github.com/karlstroetmann/Logic/blob/master/Python/Chapter-5/Unification.ipynb.

enthält ein *Python*-Programm, das den oben beschriebenen Algorithmus umsetzt.

5.9 Ein Kalkül für die Prädikatenlogik ohne Gleichheit

In diesem Abschnitt setzen wir voraus, dass unsere Signatur Σ 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 Σ das Gleichheits-Zeichen enthält, einen vollständigen Kalkül. Dieser ist allerdings deutlich aufwendiger als der Kalkül, den wir gleich einführen werden.

Definition 52 (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*

$$\mu = \mathtt{mgu}\big(p(s_1, \cdots, s_n), p(t_1, \cdots, t_n)\big).$$

Dann ist

$$\frac{k_1 \cup \{p(s_1, \cdots, s_n)\} \qquad \{\neg p(t_1, \cdots, t_n)\} \cup k_2}{k_1\mu \cup k_2\mu} \text{ eine Anwendung der Resolutions-Regel.}$$

◇

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 σ 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 53 (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. *$\mu = \mathtt{mgu}\big(p(s_1, \cdots, s_n), p(t_1, \cdots, t_n)\big)$.*

Dann sind

$$\frac{k \cup \{p(s_1, \cdots, s_n),\ p(t_1, \cdots, t_n)\}}{k\mu \cup \{p(s_1, \cdots, s_n)\mu\}} \quad \text{und} \quad \frac{k \cup \{\neg p(s_1, \cdots, s_n),\ \neg p(t_1, \cdots, t_n)\}}{k\mu \cup \{\neg p(s_1, \cdots, s_n)\mu\}}$$

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:

   $$\{p(f(x), y), p(u, g(v))\} \quad \vdash \quad \{p(f(x), g(v))\}.$$

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:

   $$\{\neg p(f(x), y), \neg p(u, g(v))\} \quad \vdash \quad \{\neg p(f(x), g(v))\}.$$

3. Wir schließen den Beweis mit einer Anwendung der Resolutions-Regel ab. Der dabei verwendete Unifikator ist die leere Substitution, es gilt also $\mu = []$.

   $$\{p(f(x), g(v))\}, \quad \{\neg p(f(x), g(v))\} \quad \vdash \quad \{\}.$$

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 54 (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 55 (Korrektheits-Satz)
Ist $M = \{k_1, \cdots, k_n\}$ eine Menge von Klauseln und gilt $M \vdash k$, so folgt

$$\models \forall(k_1) \land \cdots \land \forall(k_n) \rightarrow \forall(k).$$

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 56 (Widerlegungs-Vollständigkeit (Robinson, 1965))
Ist $M = \{k_1, \cdots, k_n\}$ eine Menge von Klauseln und gilt $\models \forall(k_1) \land \cdots \land \forall(k_n) \rightarrow \bot$, 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

   $$\neg f \approx_e \forall x_1, \cdots, x_m \colon g.$$

2. Anschließend bringen wir die Matrix g in konjunktive Normalform:

   $$g \leftrightarrow k_1 \land \cdots \land k_n.$$

   Daher haben wir nun

   $$\neg f \approx_e k_1 \land \cdots \land k_n$$

   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. $\mathit{arity} := \big\{\mathit{rot} \mapsto 1, \mathit{fliegt} \mapsto 1, \mathit{glücklich} \mapsto 1, \mathit{kind} \mapsto 2\big\}$

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

$$k_1 := \{\mathit{kind}(s(x), x), \mathit{glücklich}(x)\},$$

$$k_2 := \{\neg\mathit{fliegt}(s(x)), \mathit{glücklich}(x)\}.$$

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:

$$k_3 := \{\neg\mathit{rot}(x), \mathit{fliegt}(x)\}.$$

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:

$$k_4 := \{\neg\mathit{rot}(x), \neg\mathit{kind}(y, x), \mathit{rot}(y)\}.$$

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

$$k_5 = \{\mathit{rot}(d)\},$$

$$k_6 = \{\neg\mathit{glücklich}(d)\}.$$

Wir müssen also untersuchen, ob die Menge M, die aus den folgenden Klauseln besteht, widersprüchlich ist:

1. $k_1 = \{\mathit{kind}(s(x), x),\ \mathit{glücklich}(x)\}$
2. $k_2 = \{\neg\mathit{fliegt}(s(x)),\ \mathit{glücklich}(x)\}$

3. $k_3 = \{\neg \mathit{rot}(x),\ \mathit{fliegt}(x)\}$
4. $k_4 = \{\neg \mathit{rot}(x),\ \neg \mathit{kind}(y,x),\ \mathit{rot}(y)\}$
5. $k_5 = \{\mathit{rot}(d)\}$
6. $k_6 = \{\neg \mathit{glücklich}(d)\}$

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
$$\mathtt{mgu}\big(\mathit{rot}(d), \mathit{rot}(x)\big) = [x \mapsto d].$$
Daher können wir die Resolutions-Regel auf die Klauseln k_5 und k_4 wie folgt anwenden:
$$\{\mathit{rot}(d)\},\ \{\neg \mathit{rot}(x), \neg \mathit{kind}(y,x), \mathit{rot}(y)\}\ \vdash\ \{\neg \mathit{kind}(y,d), \mathit{rot}(y)\}.$$
2. Wir wenden nun auf die resultierende Klausel und auf die Klausel k_1 die Resolutions-Regel an. Dazu berechnen wir zunächst
$$\mathtt{mgu}\big(\mathit{kind}(y,d), \mathit{kind}(s(x),x)\big) = [y \mapsto s(d), x \mapsto d].$$
Dann haben wir
$$\{\neg \mathit{kind}(y,d), \mathit{rot}(y)\},\ \{\mathit{kind}(s(x),x), \mathit{glücklich}(x)\}\ \vdash\ \{\mathit{glücklich}(d), \mathit{rot}(s(d))\}.$$
3. Jetzt wenden wir auf die eben abgeleitete Klausel und die Klausel k_6 die Resolutions-Regel an. Wir haben:
$$\mathtt{mgu}\big(\mathit{glücklich}(d), \mathit{glücklich}(d)\big) = []$$
Also erhalten wir
$$\{\mathit{glücklich}(d), \mathit{rot}(s(d))\},\ \{\neg \mathit{glücklich}(d)\}\ \vdash\ \{\mathit{rot}(s(d))\}.$$
4. Auf die Klausel $\{\mathit{rot}(s(d))\}$ und die Klausel k_3 wenden wir die Resolutions-Regel an. Zunächst haben wir
$$\mathtt{mgu}\big(\mathit{rot}(s(d)), \neg \mathit{rot}(x)\big) = [x \mapsto s(d)]$$
Also liefert die Anwendung der Resolutions-Regel:
$$\{\mathit{rot}(s(d))\},\ \{\neg \mathit{rot}(x), \mathit{fliegt}(x)\}\ \vdash\ \{\mathit{fliegt}(s(d))\}$$
5. Um die so erhaltenen Klausel $\{\mathit{fliegt}(s(d))\}$ mit der Klausel k_3 resolvieren zu können, berechnen wir
$$\mathtt{mgu}\big(\mathit{fliegt}(s(d)), \mathit{fliegt}(s(x))\big) = [x \mapsto d]$$
Dann liefert die Resolutions-Regel
$$\{\mathit{fliegt}(s(d))\},\ \{\neg \mathit{fliegt}(s(x)), \mathit{glücklich}(x)\}\ \vdash\ \{\mathit{glücklich}(d)\}.$$
6. Auf das Ergebnis $\{\mathit{glücklich}(d)\}$ und die Klausel k_6 können wir nun die Resolutions-Regel anwenden:
$$\{\mathit{glücklich}(d)\},\ \{\neg \mathit{glücklich}(d)\}\ \vdash\ \{\}.$$

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. $\diamond$

Aufgabe 14: 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 15: 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.

5.10 Vampire

The logical calculus described in the last section can be automated and forms the basis of modern automatic provers. This section presents the theorem prover Vampire [KV13]. We introduce this theorem prover via a small example from group theory.

5.10.1 Proving Theorems in Group Theory

A group is a triple $\mathcal{G} = \langle G, \mathrm{e}, \circ \rangle$ such that

1. G is a set.
2. e is an element of G.
3. $\circ$ is a binary operation on G, i.e. we have
$$\circ : G \times G \to G.$$
4. Furthermore, the following axioms hold:
   (a) $\forall x : e \circ x = x,$ (e is a left identity)
   (b) $\forall x : \exists y : y \circ x = \mathrm{e},$ (every element has a left inverse)
   (c) $\forall x : \forall y : \forall z : (x \circ y) \circ z = x \circ (y \circ z).$ ($\circ$ is associative)

It is a well known fact that the given axioms imply the following:

1. The element e is also a right identity, i.e. we have
$$\forall x : x \circ \mathrm{e} = x.$$

2. Every element has a right inverse, i.e. we have

   $$\forall x : \exists y : x \circ y = e.$$

We will show both these claims with the help of *Vampire*. Figure 5.27 on page 139 shows the input file for *Vampire* that is used to prove that the left identity element e is also a right identity. We discuss this file line by line.

```
1  fof(identity, axiom, ! [X] : mult(e,X) = X).
2  fof(inverse,  axiom, ! [X] : ? [Y] : mult(Y, X) = e).
3  fof(assoc,    axiom, ! [X,Y,Z] : mult(mult(X, Y), Z) = mult(X, mult(Y, Z))).
4
5  fof(right, conjecture, ! [X] : mult(X, e) = X).
```

Figure 5.27: Prove that the left identity is also a right identity.

1. Line 1 states the axiom $\forall x : \mathrm{e} \circ x = x$. Every formula is written in the form

   `fof(`*name*, *type*, *formula* `).`

   - *name* is a string giving the name of the formula. This name can be freely chosen, but should contain only letters, digits, and underscores. Furthermore, it should start with a letter.
   - *type* is either the string "`axiom`" or the string "`conjecture`". Every file must hold exactly one conjecture. The conjecture is the formula that has to be proven from the axioms.
   - *formula* is FOL formula. The precise syntax of formulas will be described below.

2. Line 2 states the axiom $\forall x : \exists y : y \circ x = \mathrm{e}$.

3. Line 3 states the axiom $\forall x : \forall y : \forall z : (x \circ y) \circ z = x \circ (y \circ z)$.

4. Line 5 states the conjecture $\forall x : x \circ \mathrm{e} = x$. The keyword `conjecture` signifies that we want to prove this formula.

In order to understand the syntax of *Vampire* formulas we first have to note that all variables start with a capital letter, while function symbols and predicate symbols start with a lower case letter. As *Vampire* does not support binary operators, we had to introduce the function symbol `mult` to represent the operator $\circ$. Therefore, the term `mult`(x, y) is interpreted as $x \circ y$. Instead of `mult` we could have chosen any other name. Furthermore, *Vampire* uses the following operators:

(a) `! [X]`: F is interpreted as $\forall x : F$.

(b) `? [X]`: F is interpreted as $\exists x : F$.

(c) `$true` is interpreted as $\top$.

(d) `$false` is interpreted as $\bot$.

(e) `~`F is interpreted as $\neg F$.

(f) F `&` G is interpreted as $F \wedge G$.

(g) F `|` G is interpreted as $F \vee G$.

(h) F `=>` G is interpreted as $F \rightarrow G$.

(i) F `<=>` G is interpreted as $F \leftrightarrow G$.

When the text shown in Figure 5.27 is stored in a file with the name `group.tptp`, then we can invoke *Vampire* with the following command

```
vampire group.tptp
```

This will produce the output shown in Figure 5.28. If we want to prove that the left inverse is also a right inverse we can simply change the last line in Figure 5.27 to

```
fof(right, conjecture, ![X]: ?[Y]: mult(X, Y) = e).
```

Exercise 16: Use *Vampire* to show that in every group the left inverse is unique. ⋄

5.10.2 Who killed Agatha?

Next, we solve the following puzzle.

> Someone who lives in Dreadbury Mansion killed Aunt Agatha. Agatha, the butler, and Charles live in Dreadbury Mansion, and are the only people who live therein. A killer always hates his victim, and is never richer than his victim. Charles hates no one that Aunt Agatha hates. Agatha hates everyone except the butler. The butler hates everyone not richer than Aunt Agatha. The butler hates everyone Aunt Agatha hates. No one hates everyone. Agatha is not the butler.

The question then is: Who killed Agatha? Let us first solve the puzzle by hand. As there are only three suspects who could have killed Agatha, we proceed with a case distinction.

1. Charles killed Agatha.

   (a) As a killer always hates its victim, Charles must then have hated Agatha.

   (b) As Charles hates no one that Agatha hates, Agatha can not have hated herself.

   (c) But Agatha hates everybody with the exception of the butler and since Agatha is not the butler, she must have hated herself.
   This contradiction shows that Charles has not killed Agatha.

2. The butler killed Agatha.

   (a) As a killer is never richer than his victim, the butler can then not be not richer than Agatha.

   (b) But as the butler hates every one not richer than Agatha, he would then hate himself.

   (c) As the butler also hates everyone that Agatha hates and Agatha hates everyone except the butler, the butler would then hate everyone.

   (d) However, we know that no one hates everyone.
   This contradiction shows, that the butler has not killed Agatha.

```
vampire group-right-identity.tptp
% Running in auto input_syntax mode. Trying TPTP
% Refutation found. Thanks to Tanya!
% SZS status Theorem for group-right-identity
% SZS output start Proof for group-right-identity
1. ! [X0] : mult(e,X0) = X0 [input]
2. ! [X0] : ? [X1] : e = mult(X1,X0) [input]
3. ! [X0,X1,X2] : mult(mult(X0,X1),X2) = mult(X0,mult(X1,X2)) [input]
4. ! [X0] : mult(X0,e) = X0 [input]
5. ~! [X0] : mult(X0,e) = X0 [negated conjecture 4]
6. ? [X0] : mult(X0,e) != X0 [ennf transformation 5]
7. ! [X0] : (? [X1] : e = mult(X1,X0) => e = mult(sK0(X0),X0)) [choice axiom]
8. ! [X0] : e = mult(sK0(X0),X0) [skolemisation 2,7]
9. ? [X0] : mult(X0,e) != X0 => sK1 != mult(sK1,e) [choice axiom]
10. sK1 != mult(sK1,e) [skolemisation 6,9]
11. mult(e,X0) = X0 [cnf transformation 1]
12. e = mult(sK0(X0),X0) [cnf transformation 8]
13. mult(mult(X0,X1),X2) = mult(X0,mult(X1,X2)) [cnf transformation 3]
14. sK1 != mult(sK1,e) [cnf transformation 10]
16. mult(sK0(X2),mult(X2,X3)) = mult(e,X3) [superposition 13,12]
18. mult(sK0(X2),mult(X2,X3)) = X3 [forward demodulation 16,11]
22. mult(sK0(sK0(X1)),e) = X1 [superposition 18,12]
24. mult(X5,X6) = mult(sK0(sK0(X5)),X6) [superposition 18,18]
35. mult(X3,e) = X3 [superposition 24,22]
55. sK1 != sK1 [superposition 14,35]
56. $false [trivial inequality removal 55]
% SZS output end Proof for group-right-identity
% ------------------------------
% Version: Vampire 4.7 (commit )
% Termination reason: Refutation
```

Figure 5.28: Vampire proof that the left identity is a right identity.

3. Hence we must conclude that Agatha has killed herself.

Next, we show how *Vampire* can solve the puzzle. As there are only three possibilities, we try to prove the following conjectures one by one:

1. Charles killed Agatha.
2. The butler killed Agatha.
3. Agatha killed herself.

Figure 5.29 on page 142 shows the axioms and the conjecture of the last proof attempt. The first two proof attempts fail, but the last one is successful. Hence we have shown again that Agatha committed suicide.

```
% Someone who lives in Dreadbury Mansion killed Aunt Agatha.
fof(a1, axiom, ?[X] : (lives_at_dreadbury(X) & killed(X, agatha))).
% Agatha, the butler, and Charles live in Dreadbury Mansion, and are
% the only people who live therein.
fof(a2, axiom, ![X] : (lives_at_dreadbury(X) <=>
                       (X = agatha | X = butler | X = charles))).
% A killer always hates his victim.
fof(a3, axiom, ![X, Y]: (killed(X, Y) => hates(X, Y))).
% A killer is never richer than his victim.
fof(a4, axiom, ![X, Y]:  (killed(X, Y) => ~richer(X, Y))).
% Charles hates no one that Aunt Agatha hates.
fof(a5, axiom, ![X]: (hates(agatha, X) => ~hates(charles, X))).
% Agatha hates everyone except the butler.
fof(a6, axiom, ![X]: (hates(agatha, X) <=> X != butler)).
% The butler hates everyone not richer than Aunt Agatha.
fof(a7, axiom, ![X]: (~richer(X, agatha) => hates(butler, X))).
% The butler hates everyone Aunt Agatha hates.
fof(a8, axiom, ![X]: (hates(agatha, X) => hates(butler, X))).
%  No one hates everyone.
fof(a9, axiom, ![X]: ?[Y]: ~hates(X, Y)).
% Agatha is not the butler.
fof(a0, axiom, agatha != butler).

fof(c, conjecture, killed(agatha, agatha)).
```

Figure 5.29: Who killed Agatha?

5.11 *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.

5.11.1 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 5.30 auf Seite 144 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 5.30 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.

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 5.31 zeigt eine Eingabe-Datei für *Prover9*, bei

```
formulas(sos).
all x (e * x = x).                                 % left neutral
all x exists y (y * x = e).                        % left inverse
all x all y all z ((x * y) * z = x * (y * z)).  % associativity
end_of_list.

formulas(goals).
all x (x * e = x).                                 % right neutral
all x exists y (x * y = e).                        % right inverse
end_of_list.
```

Figure 5.30: Textuelle Darstellung der Axiome der Gruppentheorie.

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.

```
formulas(sos).
all x (e * x = x).                                 % left neutral
all x exists y (y * x = e).                        % left inverse
all x all y all z ((x * y) * z = x * (y * z)).  % associativity
end_of_list.

formulas(goals).
all x all y (x * y = y * x).                       % * is commutative
end_of_list.
```

Figure 5.31: Gilt das Kommutativ-Gesetz in allen Gruppen?

5.11.2 *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 5.32 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 5.32 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.

```
formulas(theory).
all x (e * x = x).                             % left neutral
all x exists y (y * x = e).                    % left inverse
all x all y all z ((x * y) * z = x * (y * z)). % associativity
a * b != b * a.                                % a and b do not commute
end_of_list.
```

Figure 5.32: Gibt es eine Gruppe, in der das Kommutativ-Gesetz nicht gilt?

Abbildung 5.33 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:

$\circ$	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. ◇

```
=============================== 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 5.33: Ausgabe von *Mace4*.

5.12 Reflexion

1. Was ist eine Signatur?
2. Wie haben wir die Menge $\mathcal{T}_\Sigma$ der Σ-Terme definiert?
3. Was ist eine atomare Formel?
4. Wie haben wir die Menge $\mathbb{F}_\Sigma$ der Σ-Formeln definiert?
5. Was ist eine Σ-Struktur?
6. Es sei $\mathcal{S}$ eine Σ-Struktur. Wie haben wir den Begriff der $\mathcal{S}$-Variablen-Belegung definiert?
7. Wie haben wir die Semantik von Σ-Formeln definiert?
8. Wann ist eine prädikatenlogische Formel allgemeingültig?
9. Was bedeutet die Schreibweise $\mathcal{S} \models F$ für eine Σ-Struktur $\mathcal{S}$ und eine Σ-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

[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. ISSN 0001-0782.

[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.

[JW90] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[KV13] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 1–35. Springer, 2013.

[McC97] William McCune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19: 263–276, December 1997. ISSN 0168-7433.

[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)*. ACM, 2001. URL `http://www.princeton.edu/\symbol{126}chaff/publication/DAC2001v56.pdf`.

[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