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

— Spring 2025 —

Baden-Wuerttemberg Cooperative State University (DHBW)

Prof. Dr. Karl Stroetmann

May 10, 2025

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

In the lecture on algorithms, we have discussed three important properties that an algorithms should have: An algorithm should be

- correct,
- efficient, and
- simple.

The efficiency of algorithms has been discussed 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. A stark example of IT failure consequences is the December 2022 Southwest Airlines meltdown. A number of delayed flights due to a severe winter storm triggered the collapse of the Southwest Airlines crew scheduling system SkySolver, leading to over 16,700 canceled flights and stranding hundreds of thousands of passengers during peak holiday travel. Southwest estimated the financial impact at over $1 billion.

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

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

3. 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 of embezzlement. As a consequences, some of these employees were imprisoned. Four of those that were falsely convicted committed suicide. These tragic incidents are know as the Horizon IT scandal.

4. 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 and the Z3, which is a logic constraint solver, 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

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

This should not be too surprising. After all, there is no reason to believe that a good bricklayer is a good architect and neither is a good architect necessarily a good bricklayer. In computer science, a good programmer need not be a scientist at all, while a computer scientist, by its very name, is a scientist. There is no denying that mathematics in general and logic in particular is an important part of science. Furthermore, these topics form the foundation of computer science. Therefore, you should master them. In addition, this part of your scientific education is much more permanent than the knowledge of any 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, most of you will have to learn new programming languages. Then your ability to quickly grasp new concepts will be much more important than your skills in any particular programming language.

1.2 Overview

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.

1.3 Chapter Review

- Provide three examples of faulty software systems that have caused great harm. Do not cote the examples given in this Chapter but rather search the web to find more examples.
- Provide two examples of automatic theorem provers that are used in industry.
- Is first order logic more expressive as propositional logic or is it the other way around?

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 9. This program contains a `while`-loop in line 18. If there is a natural number $n \geq m$ such that the expression,

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

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

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

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

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

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

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

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

holds. Although there are a number of arguments in support of Legendre's conjecture, to this day nobody has been able to prove it. The answer to the question, whether the invocation of the

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

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

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

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

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

Figure 2.1: A program checking Legendre's conjecture.

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

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

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

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

$$exec(t)$$

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

then this is written as

$$\texttt{name}(t) = f.$$ □

Examples:

1. We define the string s_1 as follows:

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

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

2. We define the string s_2 as

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

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

3. We define the string s_3 as

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

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

```
exec(s3)
```

   yields an error message complaining about the string "++".

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

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

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

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

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

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

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

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

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

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

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

```
def stops(t, a):
    ⋮
```

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

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

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

Theorem 3 (Alan Turing, 1937) *The halting problem is undecidable.*

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

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

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

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

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

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

```
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:

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

Let us consider these cases in turn.

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

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

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

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

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

   is evaluated:

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

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

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

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

   Therefore, this case has lead us to a contradiction.

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

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

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

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

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

(a) Since we have assumed for this case that the expression `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 published in 1937 by Alan Turing (1912 – 1954) [Tur37]. 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 it would be possible to solve the halting problem if we use this more powerful programming language. 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 a bijective 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 a bijective 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:

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

Now try to derive a contradiction. ◇

Exercise 3: Prove that the set $\mathbb{Q}$ of all rational numbers is countable. The set $\mathbb{Q}$ can be defined as follows:

$$\mathbb{Q} = \Big\{ \frac{p}{q} \;\Big|\; p \in \mathbb{Z} \wedge q \in \mathbb{N} \wedge q \geq 1 \Big\}.$$

2.2 Undecidability of the Equivalence Problem

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

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

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

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

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

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

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

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

and, furthermore, it satisfies the following specification:

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

   holds, then we must have:

   $$\texttt{equal}(p_1, p_2, a) \rightsquigarrow 1.$$

3. Otherwise we must have

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

Theorem 5 *The equivalence problem is undecidable.*

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

```
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 a test function with the 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 notion of a test function.

(b) Define the halting problem.

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

(d) How is the notion $\simeq$ defined.

(e) Define the equivalence problem.

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

(g) Define the notion of a countable set.

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

(i) Provide an example of an unsolved mathematical problem that could be easily solved if the halting problem was decidable.

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.

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

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

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

1. The base case.

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

2. The induction step.

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

3. The termination proof.

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

Let us prove the claim

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

by computational induction.

1. **Base case**:

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

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

2. **Induction step**:

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

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

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

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

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

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

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

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

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

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

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

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

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

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.: $m \mathbin{//} 2 \mapsto m$

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:

$$m \mathbin{//} 2 = q \cdot n + r \tag{3.3}$$
$$r < n \tag{3.4}$$

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:

$$2 \cdot m \mathbin{//} 2 = 2 \cdot q \cdot n + 2 \cdot r.$$

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

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

As we have $2 \cdot m \mathbin{//} 2 + m \,\%\, 2 = m$ 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:

$$\begin{array}{ll} & m \mathbin{//} 2 = q \cdot n + r \\ \Rightarrow & 2 \cdot m \mathbin{//} 2 = 2 \cdot q \cdot n + 2 \cdot r \\ \Rightarrow & 2 \cdot m \mathbin{//} 2 + m \,\%\, 2 = 2 \cdot q \cdot n + 2 \cdot r + m \,\%\, 2 \\ \Rightarrow & m = 2 \cdot q \cdot n + 2 \cdot r + m \,\%\, 2 \end{array}$$

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

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

From (3.4) we know that

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

T.: As $m \mathbin{//} 2 < m$ 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

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

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

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

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 $q = a \,//\, b$ 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 $q = a \,//\, b$ 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. □

```
1  def ggt(x, y):
2      if y == 0:
3          return x
4      return ggt(y, x % y)
```

Figure 3.3: The function ggt.

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

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

Figure 3.4: The function isqrt.

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

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

Prove that the function root that is shown in Figure 3.4 on page 23 computes the integer square root of its argument. ◇

Exercise 6: Figure 3.5 shows a *Python* program implementing the function `square`. Show that the equation

$$\texttt{square}(n) = n^2$$

holds for every $n \in \mathbb{N}$. You should use computational induction to verify the claim. ◇

```
def square(n):
    if n == 0:
        return 0
    return square(n-1) + 2 * n - 1
```

Figure 3.5: The function `square`.

Exercise 7: The binomial coefficient $\binom{n}{k}$ is defined as follows:

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

Figure 3.6 shows a *Python* program implementing the function `binomial`. Prove that

$$\texttt{binomial}(n,k) = \binom{n}{k}$$

holds for all $n \in \mathbb{N}$ and all $k \in \{0, \cdots, n\}$. ◇

```
def binomial(n, k):
    if k == 0 or k == n:
        return 1
    else:
        return binomial(n - 1, k - 1) + binomial(n - 1, k)
```

Figure 3.6: The function `binomial`.

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 iteratively via loops instead of recursively, then the method of computational induction is not applicable. Instead, the method of symbolic execution is then used. We 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 3.7.

```
1  def power(x0, y0):
2      r0 = 1
3      while yn > 0:
4          if yn % 2 == 1:
5              rn+1 = rn * xn
6          xn+1 = xn * xn
7          yn+1 = yn // 2
8      return rN
```

Figure 3.7: 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 3.7. 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 $\texttt{r}_N$ 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

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

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

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

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

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

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 $\texttt{power}(a, b) = a^b$ holds. □

3.2.2 Integer Square Root

We continue with another example that demonstrates the method of symbolic execution. Figure 3.8 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 we must have $b_{31} = 1$. Otherwise we have $b_{31} = 0$. Assume now that we have already computed the bits $b_{31}, b_{30}, \cdots, b_{31-n)}$. In order to compute $b_{31-(n+1)}$ we have to check whether

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

If it is, then $b_{31-n+1} = 1$, else $b_{31-(n+1)} = 0$. With this in mind, we can state the invariants that hold when the `while` loop in line 4 is entered:

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

```
1  def root(a):
2      r_0 = 0
3      p_0 = 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 3.8: An annotated programm to compute powers.

We proceed to prove the first invariant by induction.

B.C.: $n = 0$

$p_0 = 2^{31} = 2^{31-0}$.

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

$p_{n+1} = p_n \;//\; 2 \overset{IV}{=} 2^{31-n} \;//\; 2 = 2^{31-n-1} = 2^{31-(n+1)}$.

This holds as long as $n + 1 \leq 32$. Since we have $p_{31} = 2^{31-31} = 2^0 = 1$ it follows that

$$p_{32} = p_{31} \;//\; 2 = 1 \;//\; 2 = 0.$$

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.: $n = 0$

We have $r_0 = 0$ and also $\sum_{i=32-0}^{31} b_i \cdot 2^i = \sum_{i=32}^{31} b_i \cdot 2^i = 0$, since the last sum is empty.

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

Now we need to perform 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 $r_n + p_n \leq \texttt{isqrt}(a)$ and since $p_n = 2^{31-n}$ this shows that

$$r_n + 2^{31-n} \leq \texttt{isqrt}(a).$$

This implies that $b_{31-n} = 1$. Therefore we have

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

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

In this case we have that $r_n + p_n > \texttt{isqrt}(a)$ and since $p_n = 2^{31-n}$ this shows that

$$r_n + 2^{31-n} > \texttt{isqrt}(a).$$

This implies that $b_{31-n} = 0$. Since $b_{32-(n+1)} = b_{31-n}$ we have

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

Since the program terminates when $n = 32$ we have

$$\texttt{root}(a) = r_N = r_{32} = \sum_{i=32-32}^{31} b_i \cdot 2^i = \sum_{i=0}^{31} b_i \cdot 2^i = \texttt{isqrt}(a). \qquad \square$$

Exercise 8: Use the method of symbolic program execution to prove the correctness of the implementation of the Euclidean algorithm that is shown in Figure 3.9 on page 29. During the proof you should make use of the fact that for all positive natural numbers x and y the function `gcd` that computes the greatest common divisor of x and x satisfies the equation

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

Furthermore, the invariant of the `while` loop is

$$\texttt{ggt}(x_n, y_n) = \texttt{ggt}(a, b) \quad \text{where } a := x_0 \text{ and } b := y_0.$$

Using this invariant you should be able to prove that $\texttt{ggt}(a, b) = \texttt{gcd}(a, b)$ 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 `gcd` that computes the greatest common divisor and the *Python* function `ggt` that is implemented in Figure 3.9. ◇

```
def ggt(x_0, y_0):
    while y_n != 0:
        x_{n+1}, y_{n+1} = y_n, x_n % y_n
    return x_N
```

Figure 3.9: An iterative implementation of the Euclidean algorithm.

Exercise 9: Figure 3.10 shows a *Python* program implementing the function `square`. Show that the equation

$$\texttt{square}(s) = s^2$$

holds for every $s \in \mathbb{N}$. You should use symbolic execution to verify the claim. ◇

```
1  def square(s_0):
2      r_0 = 0
3      u_0 = 1
4      while s_n > 0:
5          r_{n+1} += u_n
6          u_{n+1} += 2
7          s_{n+1} -= 1
8      return r_N
```

Figure 3.10: The function `square` with annotations.

3.3 Check Your Understanding

(a) Explain the method of computational induction.

(b) What are the three steps that have to be executed to prove the correctness of the implementation of a *Python* function via computational induction?

(c) What are the two properties of Euclidean division of natural numbers?

(d) Are you able to prove the correctness of an iterative implementation of Euclid's algorithm?

(e) What is the difference between a variable occurring in a mathematical formula and a variable occurring in a *Python* program?

(f) How do we transform a variable occurring in a *Python* program into a mathematical variable?

(g) Explain the method of symbolic execution.

(h) Are you able to prove the correctness of an iterative implementation of Euclid's algorithm?

(i) 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. $\Big(\big((\neg p \to q) \vee (q \to \neg p) \big) \to r \Big) \in \mathcal{F}$. □

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

1. Outermost parentheses are dropped. Therefore, we write
   $$p \wedge q \quad \text{instead of} \quad (p \wedge q).$$
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
   $$p \wedge q \wedge r \quad \text{instead of} \quad (p \wedge q) \wedge r.$$
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 $\to$ is right associative, i.e. we write
   $$p \to q \to r \quad \text{instead of} \quad p \to (q \to r).$$
6. The operators $\vee$ and $\wedge$ have a higher precedence than the operator $\to$. Therefore, we write
   $$p \wedge q \to r \quad \text{instead of} \quad (p \wedge q) \to r.$$
7. The operator $\to$ has a higher precedence than the operator $\leftrightarrow$. Therefore, we write
   $$p \to q \leftrightarrow r \quad \text{instead of} \quad (p \to q) \leftrightarrow r.$$
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*

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

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 $\ominus$, $\oslash$, $\otimes$, $\oplus$, and $\ominus$. These functions have the following signatures:

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

We will use these functions as the valuations of the propositional operators. It is easiest to define the functions $\ominus$, $\oslash$, $\otimes$, $\oplus$, and $\ominus$ via the following truth table (Table 4.1):

p	q	$\ominus(p)$	$\otimes(p,q)$	$\oslash(p,q)$	$\oplus(p,q)$	$\ominus(p,q)$
`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. $\widehat{\mathcal{I}}(f \wedge g) := \oslash\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ for all $f, g \in \mathcal{F}$.
6. $\widehat{\mathcal{I}}(f \vee g) := \oslash\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ for all $f, g \in \mathcal{F}$.
7. $\widehat{\mathcal{I}}(f \rightarrow g) := \ominus\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ for all $f, g \in \mathcal{F}$.
8. $\widehat{\mathcal{I}}(f \leftrightarrow g) := \ominus\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ for all $f, g \in \mathcal{F}$.

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

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

that is defined on all propositional formulas and the function

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

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

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

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

◇

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 $\ominus$ 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 40. 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 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      evaluate(G, I) <=  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 in *Python* the values `True` and `False` are ordered and we have

   `False < True`.

   Now the evaluation of the formula $G \rightarrow H$ is only false if the evaluation of G yields `True` but the evaluation of H yields false. Therefore,

   $$\texttt{evaluate}(G \rightarrow H, \text{I}) \quad \text{has the same value as} \quad \texttt{evaluate}(G), I) \leq \texttt{evaluate}(H, I).$$

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 Caine. 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 Caine 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.

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

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

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

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

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

4. If exactly two of the suspects are guilty, then Caine 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 Caine 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 Caine is innocent, then Aaron is guilty.

   This translates into the following formula:

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

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 Caine 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.

```
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 Caine is guilty.
9  f1 = 'a ∨ b ∨ c'
10 # If Aaron is guilty, he has exactly one accomplice.
11 f2 = 'a → b ∨ c'
12 f3 = 'a → ¬(b ∧ c)'
13 # If Bernard is innocent, then Caine is innocent, too.
14 f4 = '¬b → ¬c'
15 # If exactly two of the suspects are guilty, then Caine is one of them.
16 f5 = '¬(¬c ∧ a ∧ b)'
17 # If Caine is innocent, then Aaron is guilty.
18 f6 = '¬c → a'
19 Fs = { f1, f2, f3, f4, f5, f6 };
20 Fs = { 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.

(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`

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 Caine are guilty, while Aaron is innocent.

Exercise 10: 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

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

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

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*

$$\models f \leftrightarrow g.$$ ◇

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
$\models p \vee \top \leftrightarrow \top$	$\models p \wedge \bot \leftrightarrow \bot$	
$\models p \wedge p \leftrightarrow p$	$\models p \vee p \leftrightarrow p$	idempotency
$\models p \wedge q \leftrightarrow q \wedge p$	$\models p \vee q \leftrightarrow q \vee p$	commutativity
$\models (p \wedge q) \wedge r \leftrightarrow p \wedge (q \wedge r)$	$\models (p \vee q) \vee r \leftrightarrow p \vee (q \vee r)$	associativity
$\models \neg\neg p \leftrightarrow p$		elimination of $\neg\neg$
$\models p \wedge (p \vee q) \leftrightarrow p$	$\models p \vee (p \wedge q) \leftrightarrow p$	absorption
$\models p \wedge (q \vee r) \leftrightarrow (p \wedge q) \vee (p \wedge r)$	$\models p \vee (q \wedge r) \leftrightarrow (p \vee q) \wedge (p \vee r)$	distributivity
$\models \neg(p \wedge q) \leftrightarrow \neg p \vee \neg q$	$\models \neg(p \vee q) \leftrightarrow \neg p \wedge \neg q$	DeMorgan
$\models (p \rightarrow q) \leftrightarrow \neg p \vee q$		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

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

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 $\texttt{collectVars}(f)$ is the set of propositional variables occurring in f. This function can be defined recursively.

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

Figure 4.3 on page 45 shows how to implement this definition. Note that we have been able to combine the last four cases.

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

Figure 4.3: Checking that a formula is a tautology.

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 46 shows the implementation.

1. First, we compute the set P of all propositional variables occurring in f.
2. The function $\texttt{allSubsets}(M)$ takes a set M as its input and returns the list of all subsets of M. The idea behind the definition of $\texttt{allSubsets}(M)$ is as follows:
   (a) Let x be any element from the set M.
   (b) Then there are two kinds of subsets of M:
   - Those subsets $A \subseteq M$ that do not contain x.

- Those subsets $B \subseteq M$ that do contain x.

The set $\mathcal{L}$ of those subsets A of M that do not contain x can be calculated recursively:

$$\mathcal{L} = \texttt{allSubsets}(M - \{x\})$$

(c) Adding x to the subsets in $\mathcal{L}$ yields all those subsets of M that do contain x.

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`.

```
1   def allSubsets(M: set[T]) -> list[set[T]]:
2       "Compute a list containing all subsets of the set M"
3       if M == set():
4           return [ set() ]
5       x = M.pop() # remove x from M and return x
6       L = allSubsets(M)
7       return L + [ A | { x } for A in L ]
8
9   def tautology(f):
10      "Check, whether the formula f is a tautology."
11      P = collectVars(f)
12      for I in allSubsets(P):
13          if not evaluate(f, I):
14              return I
15      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 $\mathcal{L}$. ◇

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 $\mathcal{K}$. ◇

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. ◇

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

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

Since we have $K \wedge \top \Leftrightarrow K$ we have that

$$\{\} \Leftrightarrow \top,$$

when the empty set is interpreted as an empty set of clauses. Furthermore, we have

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

4.5.1 Transforming a Formula into CNF

Next, we present a method that can be used to transform any propositional 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
$$\begin{aligned} & (F_1 \wedge \cdots \wedge F_m) \vee (G_1 \wedge \cdots \wedge G_n) \\ \Leftrightarrow\ & (F_1 \vee G_1) \wedge \cdots \wedge (F_1 \vee G_n)\ \wedge\ \cdots\ \wedge\ (F_m \vee G_1) \wedge \cdots \wedge (F_m \vee G_n) \end{aligned}$$
   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.

4.5.2 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

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

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 definition, 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 $f = g_1 \rightarrow g_2$ 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) $\texttt{elimBiconditional}(g_1 \rightarrow g_2) = \texttt{elimBiconditional}(g_1) \rightarrow \texttt{elimBiconditional}(g_2)$.

5. If f has the form $f = g_1 \leftrightarrow g_2$, we use the equivalence

$$(g_1 \Leftrightarrow g_2) \leftrightarrow \big((g_1 \rightarrow g_2) \wedge (g_2 \rightarrow g_1)\big).$$

   This leads to the equation:

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

   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.

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

Figure 4.5: Elimination of $\leftrightarrow$

Figure 4.5 on page 51 shows the implementation of the function `elimBiconditional`.

1. In line 3, the function call `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 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.

3. In line 7, 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.

4. In line 9 we check the case that f has the form $g \leftrightarrow h$. In this case, we use the equivalence

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

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

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 53 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):
            return ('¬', eliminateConditional(g))
        case ('→', g, h):
            return eliminateConditional(('∨', ('¬', g), h))
        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 54 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. $\texttt{nnf}(p) = p \quad$ für alle $p \in \mathcal{P}$,
2. $\texttt{nnf}(\top) = \top$,
3. $\texttt{nnf}(\bot) = \bot$,
4. $\texttt{nnf}(\neg\texttt{f}) = \texttt{neg}(\texttt{f})$,
5. $\texttt{nnf}(f_1 \wedge f_2) = \texttt{nnf}(f_1) \wedge \texttt{nnf}(f_2)$,
6. $\texttt{nnf}(f_1 \vee f_2) = \texttt{nnf}(f_1) \vee \texttt{nnf}(f_2)$.

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

```python
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. $\mathtt{neg}(\neg f) = \mathtt{nnf}(\neg\neg f) = \mathtt{nnf}(f)$.

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

   Hence we have:

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

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

   Therefore we have:

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

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:
$$\texttt{cnf}(f) := \{\{f\}\} \quad \text{if } f \in \mathcal{P}.$$
2. We saw earlier that the empty set of *clauses* can be interpreted as $\top$. Therefore:
$$\texttt{cnf}(\top) := \{\}.$$
3. We had also seen that the empty set of *literals* can be interpreted as $\bot$. Therefore:
$$\texttt{cnf}(\bot) := \{\{\}\}.$$
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:
$$\texttt{cnf}(\neg p) := \{\{\neg p\}\} \quad \text{if } p \in \mathcal{P}.$$
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:
$$\begin{array}{ll} & g \vee h \\ \Leftrightarrow & (k_1 \wedge \cdots \wedge k_m) \vee (l_1 \wedge \cdots \wedge l_n) \\ \Leftrightarrow & \begin{array}{ccccccc} (k_1 \vee l_1) & \wedge & \cdots & \wedge & (k_m \vee l_1) & \wedge \\ \vdots & & & & \vdots & \\ (k_1 \vee l_n) & \wedge & \cdots & \wedge & (k_m \vee l_n) & \end{array} \\ \Leftrightarrow & \{k_i \vee l_j : i \in \{1, \cdots, m\}, j \in \{1, \cdots, n\}\} \end{array}$$
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
$$\texttt{cnf}(g \vee h) = \{k \cup l \mid k \in \texttt{cnf}(g) \ \wedge \ l \in \texttt{cnf}(h)\}.$$

Figure 4.8 on page 56 shows the implementation of the function `cnf`. (The name `cnf` is the abbreviation of conjunctive normal form).

```
1   def cnf(f: Formula) -> CNF:
2       match f:
3           case p if isinstance(p, str):
4               return { frozenset({p}) }
5           case ('⊤', ):
6               return set()
7           case ('⊥', ):
8               return { frozenset() }
9           case ('¬', p):
10              return { frozenset({ ('¬', p) }) }
11          case ('∧', g, h):
12              return cnf(g) | cnf(h)
13          case ('∨', g, h):
14              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 57 how the functions that have been shown above interact.

1. The function `normalize` first eliminates the operators "↔" with the help of the function `eliminateBiconditional`.
2. The operator "→" 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.

```
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.

Exercise 11: Calculate the conjunctive normal forms of the following formulae and state your result in set notation.

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

(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

Logic is concerned with the validity of proofs. In order to build an automatic theorem prover that can help us with the verification of digital circuit or a program we need to investigate the concept of a proof. In mathematics, a proof is a conclusive argument that starts from given axioms and applies certain rules of derivation to these axioms in order to derive theorems. Our aim is to formalize this process so that we are able to implement it. The formalized version of a proof will be called 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

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

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 $p \to r$ follows from the two formulas $p \to q$ and $q \to r$. We compute the conjunctive normal form of the formula

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

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 $p \to r$ follows from the formulas $p \to q$ and $q \to r$, 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

$$\big\langle \langle f_1, f_2 \rangle, k \big\rangle.$$

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 $\big\langle \langle f_1, f_2 \rangle, k \big\rangle$ 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	Denying the Antecedent
$\dfrac{f \qquad f \to g}{g}$	$\dfrac{\neg g \qquad f \to g}{\neg f}$	$\dfrac{\neg f \qquad f \to g}{\neg g}$

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

$$\models f_1 \wedge f_2 \rightarrow k. \qquad \diamond$$

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 "Denying the Antecedent" is, in fact, a logical fallacy.

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:

$$\begin{aligned}
& (k_1 \vee p) \wedge (\neg p \vee k_2) \rightarrow k_1 \vee k_2 \\
\Leftrightarrow\ & \neg\big((k_1 \vee p) \wedge (\neg p \vee k_2)\big) \vee k_1 \vee k_2 \\
\Leftrightarrow\ & \neg(k_1 \vee p) \vee \neg(\neg p \vee k_2) \vee k_1 \vee k_2 \\
\Leftrightarrow\ & (\neg k_1 \wedge \neg p) \vee (p \wedge \neg k_2) \vee k_1 \vee k_2 \\
\Leftrightarrow\ & (\neg k_1 \vee p \vee k_1 \vee k_2) \wedge (\neg k_1 \vee \neg k_2 \vee k_1 \vee k_2) \wedge (\neg p \vee p \vee k_1 \vee k_2) \wedge (\neg p \vee \neg k_2 \vee k_1 \vee k_2) \\
\Leftrightarrow\ & \top \wedge \top \wedge \top \wedge \top \\
\Leftrightarrow\ & \top
\end{aligned}$$

□

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:*

   $$\textit{If} \quad f \in M, \quad \textit{then} \quad M \vdash f.$$

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:*

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

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 $p \to r$ follows from $p \to q$ and $q \to r$. First, we convert all formulas into clauses:

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

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:

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

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

   $$\mathcal{I}(g \vee h) = \texttt{True}. \quad \surd$$

(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

$$\mathcal{I}(g \vee h) = \texttt{True}. \quad \surd$$ □

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 Satisfiability

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. In this case the propositional interpretation $\mathcal{I}$ is called a solution of M. 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 $\textsc{Ali}$, we formally write

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

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

$$M \not\models \bot.$$ ◇

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. ◇

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 in set notation 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. ◇

4.6.3 Refutational Completeness

We are now ready to state and proof the refutational completeness of the cut rule for propositional logic.

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

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

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

$$M \not\vdash \{\}.$$

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 $\mathtt{var}(C)$. 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 $\mathcal{I}_k(C) = \mathtt{True}$ holds. This is formally written as follows:

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

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

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

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, $\mathcal{I}_0(C) = \mathtt{True}$ holds. The only clause that contains no variable is the empty clause. Since we assumed that $M \not\vdash \{\}$, $\overline{M}$ 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

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

   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 $\texttt{I}_k(D) = \texttt{True}$.

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

$$M \not\vdash \{\},$$

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.

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

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

```
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.

we can derive both the clause

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

using the cut rule.

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:

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

```
27  def saturate(Clauses):
28      while True:
29          Derived = { C for C1 in Clauses
30                        for C2 in Clauses
31                        for C in cutRule(C1, C2)
32                    }
33          if frozenset() in Derived:
34              return { frozenset() }  # This is the set notation of ⊥.
35          Derived -= Clauses
36          if Derived == set():        # no new clauses found
37              return Clauses
38          Clauses |= Derived
```

Figure 4.11: Die Funktion `saturate`

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 $\mathcal{I}(p_i) = \mathtt{True}$ and if $l_i = \neg p_i$, then we have $\mathcal{I}(p_i) = \mathtt{False}$. Now assume that a clause C exists in the set `Clauses` such that

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

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

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

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

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

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 too inefficient 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:

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

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

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

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) $\texttt{card}(C) = 1$ *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:

1. The Cut Rule,
2. Subsumption, and
3. Case Distinction.

We are already familiar with the first of the two possibilities, and we will explain the other two in more detail later. Next, we will consider these possibilities in turn.

4.7.1 Simplification with the Cut 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

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

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

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

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

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

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:

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

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 $K \cup \big\{\{p\}\big\}$ or $K \cup \big\{\{\neg p\}\big\}$ 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 $K \cup \big\{\{p\}\big\}$ is also satisfiable, otherwise $K \cup \big\{\{\neg p\}\big\}$ is satisfiable.

"$\Leftarrow$": Since K is a subset of both $K \cup \big\{\{p\}\big\}$ and of $K \cup \big\{\{\neg p\}\big\}$, 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

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

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
   $$\mathcal{I} := \big\{p \mapsto \texttt{True} \mid p \in \mathcal{P} \wedge \{p\} \in K\big\} \cup \big\{p \mapsto \texttt{False} \mid p \in \mathcal{P} \wedge \{\neg p\} \in K\big\}$$
   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
   $$K \cup \big\{\{p\}\big\}$$
   If successful, we have found a solution for K.
   (b) If $K \cup \big\{\{p\}\big\}$ is unsolvable, we instead try to solve the set of clauses
   $$K \cup \big\{\{\neg p\}\big\}$$
   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:

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

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:

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

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
   $$K_0 := K \cup \big\{\{p\}\big\}$$
   and attempt to solve this set. To do this, we form
   $$K_1 := \texttt{reduce}\big(K_0, p\big) = \Big\{\{r, \neg t\},\ \{r, s\},\ \{\neg r, q\},\ \{\neg q, s, \neg r\},\ \{\neg r, \neg s\},\ \{\neg s\},\ \{p\}\ \Big\}.$$
   The clause set K_1 contains the unit clause $\{\neg s\}$, so next we reduce using this clause:
   $$K_2 := \texttt{reduce}\big(K_1, \neg s\big) = \Big\{\{r, \neg t\},\ \{r\},\ \{\neg r, q\},\ \{\neg q, \neg r\},\ \{\neg s\},\ \{p\}\Big\}.$$
   We have found the new unit clause $\{r\}$. We use this unit clause to reduce K_2:
   $$K_3 := \mathit{reduce}\big(K_2, r\big) = \Big\{\{r\},\ \{q\},\ \{\neg q\},\ \{\neg s\},\ \{p\}\Big\}$$
   Since K_3 contains the unit clause $\{q\}$, we now reduce with q:
   $$K_4 := \mathit{reduce}\big(K_3, q\big) = \Big\{\{r\},\ \{q\},\ \{\},\ \{\neg s\},\ \{p\}\Big\}.$$
   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
   $$K_6 = \mathit{reduce}\big(K_5, \neg p\big) = \Big\{\{q, s\},\ \{r, s\},\ \{\neg s\},\ \{\neg q, s\},\ \{\neg r, \neg s\},\ \{\neg p\}\ \Big\}.$$
   The set K_6 contains the unit clause $\{\neg s\}$. We therefore form
   $$K_7 = \mathit{reduce}\big(K_6, \neg s\big) = \Big\{\{q\},\ \{r\},\ \{\neg s\},\ \{\neg q\},\ \{\neg p\}\ \Big\}.$$
   The set K_7 contains the new unit clause $\{q\}$. We use this unit clause to reduce K_7:
   $$K_8 = \mathit{reduce}\big(K_7, q\big) = \Big\{\{q\},\ \{r\},\ \{\neg s\},\ \{\},\ \{\neg p\}\ \Big\}.$$
   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

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 75. 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   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`.

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.

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

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

   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

$$\texttt{S} \cup \Big\{ \big\{ \overline{\texttt{l}} \big\} \Big\}$$

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 76.

```
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`.

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.

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

Figure 4.15: The function `reduce`.

```
1   def selectLiteral(Clauses, Forbidden):
2       Variables = { extractVariable(l) for C in Clauses for l in C } - Forbidden
3       Scores    = {}
4       for var in Variables:
5           cmp = ('¬', var)
6           Scores[var] = 0.0
7           Scores[cmp] = 0.0
8           for C in Clauses:
9               if var in C:
10                  Scores[var] += 2 ** -len(C)
11              if cmp in C:
12                  Scores[cmp] += 2 ** -len(C)
13      return max(Scores, key=Scores.get)
14
15  def extractVariable(l):
16      match l:
17          case ('¬', p): return p
18          case p       : return p
```

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

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 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:

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

   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 12: The set of clauses M is given as follows:

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

Check whether the set M is contradictory. ◇

4.8 Solving Logical Puzzles

In this section, we demonstrate how certain combinatorial problems can be reduced to the satisfiablity of a set of formulas from propositional logic. These problems can then be solved using the algorithm of Davis and Putnam. We will discuss the solution of two puzzles:

1. We start with the 8-Queens Problem.
2. As a second example we discuss the Zebra Puzzle. This puzzle is also known as Einstein's Riddle, even though there is no evidence that it is related to Albert Einstein in any way.

4.8.1 The 8-Queens Problem

As our first example, we consider the 8-Queens Problem. This problem asks us to place 8 queens on a chessboard such 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 79 shows a chessboard where a queen is placed in the third row of the fourth column. This queen can move to all the fields marked with arrows, and thus can attack pieces that are placed on these fields.

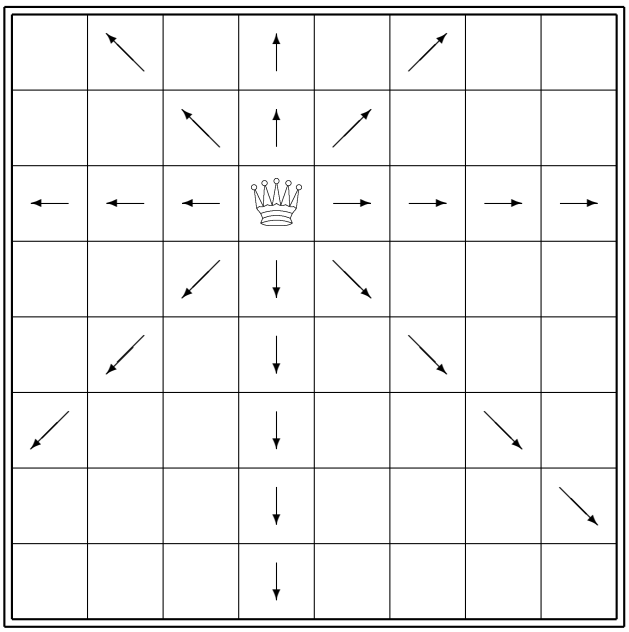

Figure 4.17: The 8-Queens-Problem.

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 column of the 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, while the columns are numbered from left to right. Figure 4.19 on page 80 shows the assignment of variables to the squares. The function shown in Figure 4.18 var(r, c) calculates the variable that expresses that there is a queen in row r and column c.

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

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

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

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.

- "at most one queen in a row",
- "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

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

```
1  def atMostOne(S):
2      return { frozenset({('¬',p), ('¬', q)}) for p in S
3                                              for q in S
4                                              if  p != q
5             }
```

Figure 4.20: The function `atMostOne`.

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

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

We implement these ideas in a *Python* function. The function `atMostOne` shown in Figure 4.20 receives as input a set S of propositional variables. The call `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 given `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 `atMostOne` with this set, which delivers the result as a set of clauses.

```
1  def atMostOneInRow(row, n):
2      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`. We return a set containing a single clause because our implementation of the Davis and Putnam algorithm works with a set of clauses.

```
1  def oneInColumn(col, n):
2      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

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

The indices r and c of the variables $\texttt{Q}(r, c)$ 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.

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

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

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

```
1  def atMostOneInRisingDiagonal(k, n):
2      S = { var(row, col) for row in range(1, n+1)
3                          for col in range(1, n+1)
4                          if  row + col == k
5          }
6      return atMostOne(S)
```

Figure 4.23: The function `atMostOneInUpperDiagonal`

$$r - c = 0.$$

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.

```
1  def atMostOneInFallingDiagonal(k, n):
2      S = { var(row, col) for row in range(1, n+1)
3                          for col in range(1, n+1)
4                          if  row - col == k
5          }
6      return atMostOne(S)
```

Figure 4.24: The function `atMostOneInLowerDiagonal`.

Now we are able to summarize our results: We can construct a set of clauses that fully describe the 8-Queens problem. Figure 4.37 shows the implementation of the function `allClauses`. The call

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

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.

```
def allClauses(n):
    All = [ atMostOneInRow(row, n)           for row in range(1, n+1)        ] \
        + [ atMostOneInFallingDiagonal(k, n) for k in range(-(n-2), (n-2)+1) ] \
        + [ atMostOneInRisingDiagonal(k, n)  for k in range(3, (2*n-1)+1)    ] \
        + [ oneInColumn(col, n)              for col in range(1, n+1)        ]
    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.

The set `solution` calculated by the call `solve`$(\mathit{Clauses}, \{\})$ contains for each of the variables `Q<`r`,`c`>` either the unit clause $\{$`Q<`r`,`c`>`$\}$ (if there is a queen on this square) or the unit clause $\{$`(¬, Q<`r`,`c`>)`$\}$ (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.

```
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()`.

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.

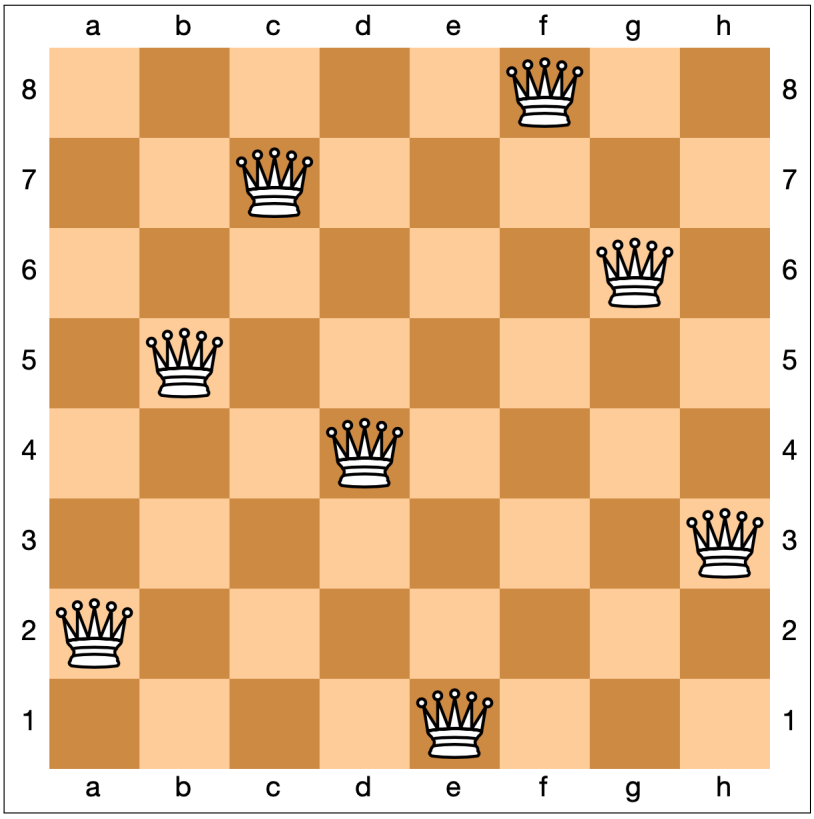

Figure 4.28: A solution of the 8-queens-problem.

4.8.2 The Zebra Puzzle

The following logical puzzle appeared in the magazine Life International on December, 17th in the year 1962:

- There are five houses.
- The Englishman lives in the red house.
- The Spaniard owns the dog.
- Coffee is drunk in the green house.
- The Ukrainian drinks tea.
- The green house is immediately to the right of the ivory house.
- The Old Gold smoker owns snails.
- Kools are smoked in the yellow house.
- Milk is drunk in the middle house.
- The Norwegian lives in the first house.

- The man who smokes Chesterfields lives in the house next to the man with the fox.
- Kools are smoked in the house next to the house where the horse is kept.
- The Lucky Strike smoker drinks orange juice.
- The Japanese smokes Parliaments.
- The Norwegian lives next to the blue house.

Furthermore, each of the five houses is painted in a different colour, their inhabitants are of different nationalities, own different pets, drink different beverages, and smoke different brands of cigarettes.

Our task is to answers the following questions:

1. Who drinks water?
2. Who owns the zebra?

In order to formulate this puzzle as a propositional formula we first have to choose the appropriate propositional variables. In order to express the fact that the Englishman lives in the i^{th} house we can use the variables

$$\texttt{English}\langle i \rangle \quad \text{for } i = 1, \cdots, 5.$$

The idea is that the variable $\texttt{English}\langle i \rangle$ is `True` iff the Englishman lives in the i^{th} house. In a similar way we use the variables

$$\texttt{Spanish}\langle i \rangle, \ \texttt{Ukrainian}\langle i \rangle, \ \texttt{Norwegian}\langle i \rangle, \text{ and } \texttt{Japanese}\langle i \rangle$$

to encode where the people of different nationalities live. The different drinks are encoded by the variables

$$\texttt{Coffee}\langle i \rangle, \ \texttt{Milk}\langle i \rangle, \ \texttt{OrangeJuice}\langle i \rangle, \ \texttt{Tea}\langle i \rangle, \text{ and } \texttt{Water}\langle i \rangle.$$

The colours are encoded using the variables

$$\texttt{Red}\langle i \rangle, \ \texttt{Green}\langle i \rangle, \ \texttt{Ivory}\langle i \rangle, \ \texttt{Yellow}\langle i \rangle, \text{ and } \texttt{Blue}\langle i \rangle.$$

The pets are encoded using the variables

$$\texttt{Dog}\langle i \rangle, \ \texttt{Snails}\langle i \rangle, \ \texttt{Fox}\langle i \rangle, \ \texttt{Horse}\langle i \rangle, \text{ and } \texttt{Zebra}\langle i \rangle.$$

The cigarette brands are encoded as

$$\texttt{OldGold}\langle i \rangle, \ \texttt{Kools}\langle i \rangle, \ \texttt{Chesterfields}\langle i \rangle, \ \texttt{LuckyStrike}\langle i \rangle, \text{ and } \texttt{Parliaments}\langle i \rangle.$$

We are using the angular brackets "⟨" and "⟩" as part of the variable names because our parser for propositional logic accepts these symbols as part of variable names. The parser would be confused if we would use the normal parenthesis.

In order to be able to write the conditions concisely, we introduce the function $\texttt{var}(f, i)$, which is shown in Figure 4.29. This function takes a string f, where f is a property like, e.g. `Japanese` and an integer $i \in \{1, \cdots, 5\}$ and returns the string $f\langle i \rangle$.

The function $\texttt{somewhere}(x)$ that is shown in Figure 4.30 takes a string x as its argument, where x denotes a property like, e.g. `Japanese`. It returns a clause that specifies that x has to be located in one of the five houses. For example, the call `somewhere('Japanese')` returns the clause

```
1  def var(f, i):
2      return f + "<" + str(i) + ">"
```

Figure 4.29: The function var(f, i).

```
1  def somewhere(x):
2      return frozenset({ var(x, i) for i in range(1, 5+1) })
```

Figure 4.30: The function somewhere.

```
{'Japanese<1>', 'Japanese<2>', 'Japanese<3>', 'Japanese<4>', 'Japanese<5>'}
```

as a frozenset.

```
1  def atMostOneAt(S, i):
2      return atMostOne({var(x, i) for x in S})
```

Figure 4.31: The function atMostOneAt(S, i)

Given an set of properties S and a house number i, the function atMostOne(S, i) returns a set of clauses that specifies that the person living in house number i has at most one of the properties from the set S. For example, if

$$S = \{\texttt{"Japanese"}, \texttt{"Englishman"}, \texttt{"Spaniard"}, \texttt{"Norwegian"}, \texttt{"Ukranian"}\},$$

then atMostOne$(S, 3)$ specifies that the inhabitant of house number 3 has at most one of the nationalities from the set S. The implementation makes use of the function atMostOne that is shown in Figure 4.20 on page 81.

```
1  def onePerHouse(S):
2      Clauses  = { somewhere(x) for x in S }
3      Clauses |= { C for i in range(1, 5+1) for C in atMostOneAt(S, i) }
4      return Clauses
```

Figure 4.32: The function onePerHouse.

The function onePerHouse(S) takes a set S of properties. For example, S could be the set of the five nationalities. In this case, the function would create a set of clauses that expresses that there has to be

a house where the Japanese lives, a house where the Englishman lives, a house where the Spaniard lives, a house where the Norwegian lives, and a house where the Ukranian lives. Furthermore, the set of clauses would contain clauses that express the fact that these five persons live in different houses.

```
def sameHouse(a, b):
    return { C for i in range(1,5+1)
               for C in parseKNF(f"{var(a, i)} ↔ {var(b, i)}")
           }
```

Figure 4.33: The function `sameHouse`(a, b).

Given two properties a and b the function `sameHouse`(a, b) that is shown in Figure 4.33 computes a set of clauses that specifies that if the inhabitant of house number i has the property a, then he also has the the property b and vice versa. For example, `sameHouse("Japanese","Dog")` specifies that the Japanese guy keeps a dog. The function `parseKNF` that is used here takes a string, converts it into a formula from propositional logic, and finally converts this formula into CNF.

```
def nextTo(a, b):
    Result  = parseKNF(f"{var(a,1)} → {var(b,2)}")
    Result |= { C for i in [2,3,4]
                  for C in parseKNF(f"{var(a,i)} → {var(b,i-1)} ∨ {var(b,i+1)}") }
    Result |= parseKNF(f"{var(a,5)} → {var(b,4)}")
    return Result
```

Figure 4.34: nextTo

Given to properties a and b the function `nextTo`(a, b) computes a set of clauses that specifies that the inhabitants with properties a and b are direct neighbours. For example, `nextTo('Japanese','Dog')` specifies that the Japanese guy lives next to the guy who keeps a dog.

```
def leftTo(a, b):
    Result = set()
    for i in [1, 2, 3, 4]:
        Result |= parseKNF(f"{var(a,i)} ↔ {var(b,i+1)}")
    Result |= parseKNF(f"¬{var(a,5)}")
    return Result
```

Figure 4.35: The function `leftTo`.

Given to properties a and b the function `leftTo`(a, b) computes a list of clauses that specifies that the inhabitants with properties a lives in the house to the left of the inhabitant who has property b.

For example, `livesTo('Japanese','Dog')` specifies that the Japanese guy lives in the house to the left of the house where there is a dog.

```
1 Nations  = { "English", "Spanish", "Ukrainian", "Norwegian", "Japanese" }
2 Drinks   = { "Coffee", "Tea", "Milk", "OrangeJuice", "Water" }
3 Pets     = { "Dog", "Snails", "Horse", "Fox", "Zebra" }
4 Brands   = { "LuckyStrike", "Parliaments", "Kools", "Chesterfields", "OldGold" }
5 Colours  = { "Red", "Green", "Ivory", "Yellow", "Blue" }
```

Figure 4.36: Definition of the properties.

In order to be able to formulate the different conditions conveniently, we define the sets shown in Figure 4.36.

The function `allClauses` in Figure 4.37 computes a set of clauses that encodes the conditions of the zebra puzzle. I we solve these clauses with the algorithm of Davis and Putnam, we get the solution that is shown in Figure 4.38.

```
def allClauses():
    # Every house has exactly one inhabitant.  This inhabitant has exactly one
    # nationality, one pet, smokes one brand of cigarettes, and has one type
    # of drink.  Furthermore, every house has exactly one color.
    Clauses  = onePerHouse(Nations)
    Clauses |= onePerHouse(Drinks)
    Clauses |= onePerHouse(Pets)
    Clauses |= onePerHouse(Brands)
    Clauses |= onePerHouse(Colours)
    # The Englishman lives in the red house.
    Clauses |= sameHouse("English", "Red")
    # The Spaniard owns the dog.
    Clauses |= sameHouse("Spanish", "Dog")
    # Coffee is drunk in the green house.
    Clauses |= sameHouse("Coffee", "Green")
    # The Ukrainian drinks tea.
    Clauses |= sameHouse("Ukrainian", "Tea")
    # The green house is immediately to the right of the ivory house.
    Clauses |= leftTo("Ivory", "Green")
    # The Old Gold smoker owns snails.
    Clauses |= sameHouse("OldGold", "Snails")
    # Kools are smoked in the yellow house.
    Clauses |= sameHouse("Kools", "Yellow")
    # Milk is drunk in the middle house.
    Clauses |= parseKNF("Milk<3>")
    # The Norwegian lives in the first house.
    Clauses |= parseKNF("Norwegian<1>")
    # The man who smokes Chesterfields lives in the house next
    # to the man with the fox.
    Clauses |= nextTo("Chesterfields", "Fox")
    # Kools are smoked in the house next to the house where the horse is kept.
    Clauses |= nextTo("Kools", "Horse")
    # The Lucky Strike smoker drinks orange juice.
    Clauses |= sameHouse("LuckyStrike", "OrangeJuice")
    # The Japanese smokes Parliaments.
    Clauses |= sameHouse("Japanese", "Parliaments")
    # The Norwegian lives next to the blue house.
    Clauses |= nextTo("Norwegian", "Blue")
```

Figure 4.37: The function `allClauses`.

House	Nationality	Drink	Animal	Brand	Colour
1	Norwegian	Water	Fox	Kools	Yellow
2	Ukrainian	Tea	Horse	Chesterfields	Blue
3	English	Milk	Snails	OldGold	Red
4	Spanish	OrangeJuice	Dog	LuckyStrike	Ivory
5	Japanese	Coffee	Zebra	Parliaments	Green

Figure 4.38: The solution of the zebra puzzle.

Exercise 13: Once upon a time, there was a king who wanted to marry his daughter to a prince. He had it proclaimed throughout the land that he was seeking a husband for his daughter. One day, a prince came by to apply for the position.

Since the king did not want to marry his daughter to some dullard, he led the prince into a room with nine doors. The king told the prince that the princess was in one of the rooms, but that there were other rooms behind which hungry tigers were waiting. Some rooms were also empty. If the prince opened a door with a tiger behind it, it would probably be his last mistake.

The king went on to say that there were signs on all the doors with statements on them. These statements behave as follows:

1. In the rooms where there is a tiger, the statement on the sign is false.
2. In the room where the princess is, the statement is true.
3. With regards to the the empty rooms, the situation is a bit more complicated, because there are two possibilities:
   - Either all the inscriptions on the empty rooms are true,
   - or all the inscriptions on the empty rooms are false.

The prince then read the inscriptions. These were as follows:

1. Room: The princess is in a room with an odd number and there is no tiger in the rooms with an even number.
2. Room: This room is empty.
3. Room: The inscription on Room No. 5 is true, the inscription on Room No. 7 is false, and there is a tiger in Room No. 3.
4. Room: The inscription on Room No. 1 is false, there is no tiger in Room No. 8, and the inscription on Room No. 9 is true.
5. Room: If the inscription on Room No. 2 or on Room No. 4 is true, then there is no tiger in Room No. 1.
6. Room: The inscription on Room No. 3 is false, the princess is in Room No. 2, and there is no tiger in Room No. 2.
7. Room: The princess is in Room No. 1 and the inscription on Room No. 5 is true.
8. Room: There is no tiger in this room and Room No. 9 is empty.
9. Room: Neither in this room nor in Room No. 1 is there a tiger, and moreover, the inscription on Room No. 6 is true.

I have provided the framework of a notebook at the following address:

github.com/karlstroetmann/Logic/blob/master/Python/Chapter-4/Prince-Tiger.ipynb

You should edit this notebook in order to solve the given problem. ◇

Exercise 14: 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.
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?

I have provided the framework of a notebook at the following address:

github.com/karlstroetmann/Logic/blob/master/Python/Chapter-4/Zweistein.ipynb

You should edit this notebook in order to solve the given problem.

4.9 Sudoku and PicoSat

As a final example we will solve a Sudoku puzzle. In particular, we will solve an instance of a Sudoku puzzle that was created by Arto Inkala. He claims that this is the hardest solvable instance of a Sudoku puzzle. Figure 4.39 shows the Sudoku created by Arto Inkala.

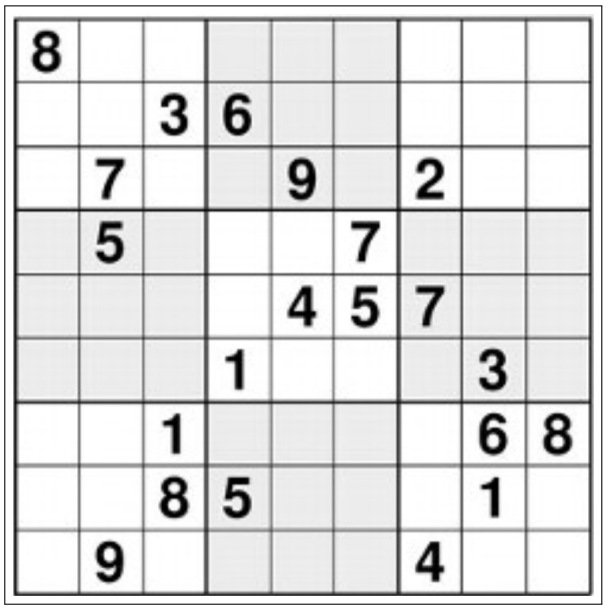

Figure 4.39: The Sudoku puzzle created by Arto Inkala.

A Sudoku puzzle is a logic-based, combinatorial number-placement puzzle. The objective is to fill a 9×9 grid with digits from the set $\{1, \cdots, 9\}$ so that each column, each row, and each of the nine 3×3 subgrids that compose the grid (also called "boxes") contain all of these digits exactly once. The puzzle is given as a partially completed grid. A well-posed puzzle must have a single unique solution.

In order to formulate a Sudoku as a satisfiablity problem from propositional logic, we will use the following set of propositional variables:

$$\mathcal{P} = \big\{ Q_{r,c,d} \mid r \in \{1, \cdots, 9\} \wedge c \in \{1, \cdots, 9\} \wedge r \in \{1, \cdots, 9\} \big\}$$

The variable $Q_{r,c,d}$ is supposed to be `True` if d is the digit in row r and column c of the Sudoku. Note that this means that we have $9^3 = 729$ different propositional variables. In our implementation we will use the function `var` shown in Figure 4.40 to create these variables.

```
1  def var(row, col, digit):
2      return f'Q<{row},{col},{digit}>'
```

Figure 4.40: The function `var`.

The Sudoku itself is created by the function `create_puzzle` shown in Figure 4.41 on page 96.

In this function, the puzzle is represented as a list of lists. Each of these lists represents a row. The inner lists contain digits and the characters "*". These characters represent the digits that are initially unknown and whose value have to be computed.

```
def create_puzzle():
    return [ [ 8 , "*", "*", "*", "*", "*", "*", "*", "*"],
             ["*", "*",  3 ,  6 , "*", "*", "*", "*", "*"],
             ["*",  7 , "*", "*",  9 , "*",  2 , "*", "*"],
             ["*",  5 , "*", "*", "*",  7 , "*", "*", "*"],
             ["*", "*", "*", "*",  4 ,  5 ,  7 , "*", "*"],
             ["*", "*", "*",  1 , "*", "*", "*",  3 , "*"],
             ["*", "*",  1 , "*", "*", "*", "*",  6 ,  8 ],
             ["*", "*",  8 ,  5 , "*", "*", "*",  1 , "*"],
             ["*",  9 , "*", "*", "*", "*",  4 , "*", "*"]
           ]
```

Figure 4.41: The function `create_puzzle`.

In our implementation we will use functions `atMostOne` that we have already seen in Figure 4.20 on page 81 that expresses that at most one of a given set of propositional variables is `True`. Furthermore, we will use the auxiliary function `atLeastOne`(V) which takes a set of propositional variables V and which returns a formula in conjunctive normal form in set notation that is `True` if and only if at least one of the variables in V is `True`. Note that in set notation we have

$$\texttt{atLeastOne}(V) = \big\{V\big\}.$$

The reason is that in set notation V is interpreted as a clause, i.e. as a disjunction of its variables. Using these two functions we can then define a function `exactlyOne`(V) that takes a set of variables and returns `True` iff exactly one of the variables in V is `True`. The implementation is shown in Figure 4.42.

```
def atLeastOne(S):
    return { frozenset(S) }

def exactlyOne(S):
    return atMostOne(S) | atLeastOne(S)
```

Figure 4.42: The functions `atLeastOne` and `exactlyOne`.

Figure 4.43 show the implementation of the function `exactlyOnce`(L). This function takes a set L of pairs as arguments. Every pair $(r, c) \in L$ is interpreted as a position in the Sudoku grid whre r specifies the row and c specifies the column of the position. The function `exactlyOnce` computes a set of formulas that express that every digit $d \in \{1, \cdots, 9\}$ occurs exactly once in the set of coordinates

L:

$$\texttt{exactlyOnce}(L) = \bigcup_{d \in \{1,\cdots,9\}} \texttt{exactlyOne}\big(\{Q_{r,c,d} \mid (r,c) \in L\}\big)$$

```
def exactlyOnce(L):
    Clauses = set()
    for digit in range(1, 10):
        Clauses |= exactlyOne({ var(col, row, digit)  for col, row in L })
    return Clauses
```

Figure 4.43: The function `exactlyOnce`.

The function $\texttt{exactlyOneDigit}(r,c)$ takes a row r and a column c as its argument. It returns a formula expressing that exactly one of the variables from the set

$$\{Q_{r,c,d} \mid d \in \{1,\cdots,9\}\}$$

is `True`. The implementation is shown in Figure 4.44.

```
def exactlyOneDigit(row, col):
    return exactlyOne({ var(row, col, digit) for digit in range(1, 10) })
```

Figure 4.44: The function `exactlyOneDigit`.

Furthermore, we need a function encoding the constraints form the puzzle itself. This function is called `constraintsFromPuzzle` and its implementation is shown in Figure 4.45 on page 98. For example, the given puzzle demands that the digit in row 2 and column 4 is the digit 6. This can be expressed as the formula $Q_{2,3,6}$. In set notation this formula takes the form

$$\big\{\{Q_{2,3,6}\}\big\}.$$

Of course, we have to take the union of all these formula. This leads to the code shown in Figure 4.45 on page 98. Note that we have to provide `row+1` and `col+1` as arguments to the function `var` since in Python list indices start with 0.

Finally, we have to combine all constraints of the puzzle. This is done with the function `allConstraints` shown in Figure 4.46 shown on page 98.

The Jupyter notebook showing the resulting program is available at:

github.com/karlstroetmann/Logic/blob/master/Python/Chapter-4/10-Sudoku.ipynb

```
def constraintsFromPuzzle():
    Puzzle = create_puzzle()
    Variables = [ var(row+1, col+1, Puzzle[row][col]) for row in range(9)
                                                       for col in range(9)
                                                       if  Puzzle[row][col] != '*'
                ]
    return { frozenset({ var }) for var in Variables }
```

Figure 4.45: The function `constraintsFromPuzzle`

```
def allConstraints():
    L = [1, 2, 3, 4, 5, 6, 7, 8, 9]
    # the constraints from the puzzle have to be satisfied
    Clauses = constraints_from_puzzle()
    # there is exactly one digit in every field
    for row in L:
        for col in L:
            Clauses |= exactlyOneDigit(row, col)
    # all entries in a row are unique
    for row in L:
        Clauses |= exactlyOnce([ (row, col) for col in L ])
    # all entries in a column are unique
    for col in L:
        Clauses |= exactlyOnce([ (row, col) for row in L ])
    # all entries in a 3x3 box are unique
    for r in range(3):
        for c in range(3):
            S = [ (r * 3 + row, c * 3 + col) for row in range(1, 4)
                                             for col in range(1, 4)
                ]
            Clauses |= exactlyOnce(S)
    return Clauses
```

Figure 4.46: The function `allConstraints`.

4.9.1 PicoSat

Unfortunately, solving the resulting set of clauses takes about 10 minutes. Fortunately, there is a version of the Davis-Putnam algorithm that is implemented in `C` and that, furthermore, incorporates a number of improvements to the original algorithm. This solver is called PicoSat and has been implemented by Armine Biere A Python API to this solver is available as pycosat. We can install it via `pip` using the following command:

```
pip install pycosat
```

In *PicoSat* propositional variables are represented as follows:

1. A propositional variable p is represented as a positive natural number.
2. If n is the natural number representing the propositional variable p, then $\neg p$ is represented as the integer $-n$.
3. A clause is represented as a list of integers.
4. A formula in CNF is represented as a list of clauses and hence as a list of list of integers.

For example, if the propositional variables p, q, and r are represented as the natural numbers 1, 2, and 3 respectively, then

1. $p \vee \neg q \vee r$ is represented as the list `[1, -2, 3]`,
2. $\neg p \vee q \vee \neg r$ is represented as the list `[-1, 2, -3]`,
3. $\neg p \vee \neg q \vee r$ is represented as the list `[-1, -2, 3]`, and
4. $p \vee q \vee \neg r$ is represented as the list `[1, 2, -3]`.

Finally, the formula

$$f = (p \vee \neg q \vee r) \wedge (\neg p \vee q \vee \neg r) \wedge (\neg p \vee \neg q \vee r) \wedge (p \vee q \vee \neg r),$$

which is in conjunctive normal form, is represented as follows:

```
f = [ [1, -2, 3], [-1, 2, -3], [-1, -2, 3], [1, 2, -3] ]
```

In order so find a solution to this formula we can call `pycosat.solve`(f).

Transforming Clauses into PyCoSat Format

In order to use *PicoSat* to solve the Sudoku discussed earlier, we need a function that transforms a formula that is in conjunctive normal form into the format of *PicoSat*. Furthermore, we need a function that can translate a solution found by *PicoSat* back into our format.

The function `findVariables(Clauses)` in Figure 4.47 on page 100 takes a set of clauses and returns the set of all propositional variables occurring in this set.

```
1  def findVariables(Clauses):
2      Variables = set()
3      for Clause in Clauses:
4          for literal in Clause:
5              match literal:
6                  case ('¬', var): Variables |= { var }
7                  case var       : Variables |= { var }
8      return Variables
```

Figure 4.47: The function `findVariables`.

The function `numberVariables(Clauses)` takes a set of clauses as input. It returns two dictionaries:

1. The dictionary `Var2Int` maps every propositional variable occurring in `Clauses` to a unique natural number.

2. The dictionary `Int2Var` is the mapping that is inverse to the dictionary `Var2Int`. Hence it maps the natural numbers back to propositional variables.

```
1  def numberVariables(Clauses):
2      Variables = findVariables(Clauses)
3      count     = 1
4      Var2Int   = {}
5      Int2Var   = {}
6      for variable in Variables:
7          Var2Int[variable] = count
8          Int2Var[count   ] = variable
9          count += 1
10     return Var2Int, Int2Var
```

Figure 4.48: The function `numberVariables`

Figure 4.49 shows the remaining auxiliary functions that are necessary to transform a formula represented as a set of clauses into the format of PicoSat.

1. The function `literal2int` takes a literal and transforms this literal into an integer representing the literal. If the literal is a negated variable, the integer is negative, else it is positive. `Var2Int` is a dictionary mapping propositional variables to natural numbers.
2. The function `clause2pyco(Clause, Var2Int)` transforms a set of literals into a list of integers. `Var2Int` is a dictionary mapping the propositional variables to natural numbers.
3. The function `clauses2pyco(Clauses, Var2Int)` transforms a set of clauses into a list of lists of integers. `Var2Int` is a dictionary mapping the propositional variables to natural numbers.
4. The function `int2var(Numbers, Int2Var)` takes a list of numbers representing a set of literals and returns the associated list of literals. `Int2Var` is a dictionary mapping natural numbers to propositional variables.

```
1  def literal2int(literal, Var2Int):
2      match literal:
3          case ('¬', var): return -Var2Int[var]
4          case var       : return  Var2Int[var]
5
6  def clause2pyco(Clause, Var2Int):
7      return [literal2int(literal, Var2Int) for literal in Clause]
8
9  def clauses2pyco(Clauses, Var2Int):
10     return [clause2pyco(clause, Var2Int) for clause in Clauses ]
11
12 def int2var(Numbers, Int2Var):
13     Result = set()
14     for n in Numbers:
15         if n > 0:
16             Result |= {frozenset({Int2Var[n]})}
17         else:
18             Result |= {frozenset({ ('¬', Int2Var[-n]) })}
19     return Result
```

Figure 4.49: Some axilliary functions to transform clauses into the format of PicoSat.

The Jupyter notebook showing the resulting program is available at:

github.com/karlstroetmann/Logic/blob/master/Python/Chapter-4/11-PicoSat-Sudoku.ipynb

4.10 Check Your Comprehension

(a) How have we defined the set of propositional logic formulas?

(b) How have we defined the semantics of propositional logic formulas?

(c) How do we represent propositional logic formulas in *Python*?

(d) What is a tautology?

(e) How can we transform propositional logic formula into conjunctive normal form?

(f) Define the notion of an inference rule.

(g) when is an inference rule correct?

(h) Define the cut rule.

(i) How did we define the proof concept $M \vdash C$?

(j) What is a saturated set of clauses?

(k) What are the two most important properties of the proof concept $\vdash$?

(l) Assume that M is a saturated set of clauses. How can we define a propositional interpretation $\mathcal{I}$ such that $\mathcal{I}(C) = \texttt{True}$ for all $C \in M$?

(m) How did we define the notion of a set of clauses being solvable?

(n) Do you know how the algorithm of Davis and Putnam works?

(o) Can you code a logical puzzle as a propositional formula?

Chapter 5

First-Order Logic

In propositional logic, we have studied the combination of atomic propositions using the propositional connectives "$\neg$", "$\vee$", "$\wedge$", "$\rightarrow$" and "$\leftrightarrow$". In first-order logic we additionally examine the structure of these atomic propositions. The following additional concepts are introduced in first-order logic:

1. Terms are used as designations for objects.
2. These terms are composed of object variables and function symbols. In the following examples, "x" is an object variable, while "*father*" and "*mother*" are unary function symbols and "*isaac*" is a nullary function symbol:

   $$\mathit{father}(x), \quad \mathit{mother}(\mathit{isaac}).$$

   Nullary function symbols are also referred to as constants and instead of object variables we simply talk about variables.
3. Different objects are related by predicate symbols. In the following example, we use the predicate symbols "*isBrother*" and "$<$". Additionally, "*albert*" and "*bruno*" are constants:

   $$\mathit{isBrother}\big(\mathit{albert}, \mathit{father}(\mathit{bruno})\big), \quad x + 7 < x \cdot 7.$$

   The resulting formulas are referred to as atomic formulas , since they do not contain subformulas.
4. Atomic formulas can be combined using the propositional connectives as shown the following example:

   $$x > 1 \rightarrow x + 7 < x \cdot 7.$$

5. Finally, the quantifiers "$\forall$" (*for all*) and "$\exists$" (*there exists*) are introduced to distinguish between variables that are existentially quantified and those variables that are universally quantified. For example, the formula

   $$\forall x \in \mathbb{R} : \exists n \in \mathbb{N} : x < n$$

   is read as: "*For all real numbers x there exists a natural number n such that x is less than n.*"

This chapter is structured as follows:

(a) In the next section, we will define the syntax of first-order logic formulas, i.e., we will specify those strings that are regarded as first-order formulas.

(b) In the following section, we deal with the semantics of these formulas, that is we specify the meaning of first-order formulas.

(c) After that, we show how to implement syntax and semantics in *Python*.

(d) As an application of first-order logic we discuss constraint programming. In constraint programming, a given problem is described using first-order logic formulas. To solve the problem, a so-called constraint solver is used.

(e) We develop a simple constraint solver that works by the means of backtracking.

(f) Then we discuss the constraint solver `Z3`, which is a state-of-the-art constraint solver developed by Microsoft Corporation.

(g) Furthermore, we consider normal forms of first-order logic formulas and show how formulas can be transformed into first-order logic clauses.

(h) We also discuss a first-order logic calculus, which is the basis of automatic reasoning in first-order logic.

(i) To conclude the chapter, we discuss the automatic theorem prover *Vampire*.

5.1 Syntax of First-Order Logic

First, we define the concept of a signature. Essentially, this is a structured summary of variables, function symbols, and predicate symbols, along with a specification of the arity of these symbols.

Definition 28 (Signature) *A signature is a 4-tuple*

$$\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle,$$

such that the following holds:

1. $\mathcal{V}$ *is the set of object variables, which we often simply refer to as variables for brevity.*
2. $\mathcal{F}$ *is the set of function symbols.*
3. $\mathcal{P}$ *is the set of predicate symbols.*
4. *arity is a function that assigns each function and predicate symbol its arity:*

   $$arity : \mathcal{F} \cup \mathcal{P} \to \mathbb{N}.$$

   We say that the function or predicate symbol f is an n-ary symbol if $arity(f) = n$.
   A function symbol c such that $arity(c) = 0$ is called a constant.
   A predicate symbol p such that $arity(p) = 0$ is called a propositional variable.

5. *Since we need to be able to distinguish between variables, function symbols, and predicate symbols, we agree that the sets $\mathcal{V}$, $\mathcal{F}$, and $\mathcal{P}$ have to be pairwise disjoint:*

$$\mathcal{V} \cap \mathcal{F} = \{\}, \quad \mathcal{V} \cap \mathcal{P} = \{\}, \quad \text{and} \quad \mathcal{F} \cap \mathcal{P} = \{\}.$$ ◇

Example: Next, we define the signature Σ_G of group theory. Let

1. $\mathcal{V} := \{x, y, z\}$ be the set of variables,
2. $\mathcal{F} := \{\mathsf{e}, \circ\}$ be the set of function symbols,
3. $\mathcal{P} := \{=\}$ be the set of predicate symbols,
4. $\mathit{arity} := \{\mathsf{e} \mapsto 0, \circ \mapsto 2, = \mapsto 2\}$ specifes the arities of function and predicate symbols.

Then the signature Σ_G of group theory is defined as follows:

$$\Sigma_G := \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle.$$ ◇

We use expressions built from variables and function symbols as identifiers for objects. These expressions are called terms. The formal definition follows.

Definition 29 (Terms, $\mathcal{T}_\Sigma$)
If $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle$ is a signature, then we define the set $\mathcal{T}_\Sigma$ of Σ-terms, inductively:

1. *For every variable $x \in \mathcal{V}$, we have that $x \in \mathcal{T}_\Sigma$. Thus, every variable is also a term.*
2. *If $c \in \mathcal{F}$ is a 0-ary function symbol, i.e. $\mathit{arity}(c) = 0$, then $c \in \mathcal{T}_\Sigma$.*
   Therefore, every constant is a term.
3. *If $f \in \mathcal{F}$ is an n-ary function symbol such that $n > 0$ and $t_1, \cdots, t_n \in \mathcal{T}_\Sigma$, then*
   $$f(t_1, \cdots, t_n) \in \mathcal{T}_\Sigma,$$
   thus the expression $f(t_1, \cdots, t_n) \in \mathcal{T}_\Sigma$ is a term. ◇

Example: Let

1. $\mathcal{V} := \{x, y, z\}$ be the set of variables,
2. $\mathcal{F} := \{0, 1, +, -, *\}$ be the set of function symbols,
3. $\mathcal{P} := \{=, \leq\}$ be the set of predicate symbols,
4. $\mathit{arity} := \{0 \mapsto 0, 1 \mapsto 0, + \mapsto 2, - \mapsto 2, * \mapsto 2, = \mapsto 2, \leq \mapsto 2\}$ specify the arity of function and predicate symbols, and
5. $\Sigma_{\text{arith}} := \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle$ be a signature.

Then we can construct Σ_{arith}-terms as follows:

1. $x, y, z \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
   since all variables are also Σ_{arith}-terms.

2. $0, 1 \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
   since 0 and 1 are nullary function symbols.
3. $+(0, x) \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
   since $0 \in \mathcal{T}_{\Sigma_{\text{arith}}}$, $x \in \mathcal{T}_{\Sigma_{\text{arith}}}$ and $+$ is a binary function symbol.
4. $*(+(0, x), 1) \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
   since $+(0, x) \in \mathcal{T}_{\Sigma_{\text{arith}}}$, $1 \in \mathcal{T}_{\Sigma_{\text{arith}}}$ and $*$ is a binary function symbol.

In practice, for certain binary functions, we use an infix notation, i.e., we write binary function symbols between their arguments. For example, we write $x + y$ instead of $+(x, y)$. The infix notation is then to be understood as an abbreviation for the above-defined representation. This, of course, works only if we also set precedence levels for the various operators. ◇

Next, we define the concept of atomic formulas. These are formulas that cannot be decomposed into smaller formulas: atomic formulas thus contain neither propositional connectives nor quantifiers.

Definition 30 (Atomic Formulas, $\mathcal{A}_\Sigma$) *Given a signature $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle$. The set of atomic Σ-formulas $\mathcal{A}_\Sigma$ is defined as follows:*

1. *$\top \in \mathcal{A}_\Sigma$ and $\bot \in \mathcal{A}_\Sigma$.*
2. *If $p \in \mathcal{P}$ and $arity(p) = 0$, then $p \in \mathcal{A}_\Sigma$.*
   Therefore, propositional variables are atomic formulas.
3. *If $p \in \mathcal{P}$ is an n-ary predicate symbol such that $n > 0$ and, furthermore, n Σ-terms $t_1, \cdots, t_n$ are given, then $p(t_1, \cdots, t_n)$ is an atomic Σ-formula:*
   $$p(t_1, \cdots, t_n) \in \mathcal{A}_\Sigma.$$ □

Example: Continuing from the last example, we can see that

$$=(*(+(0, x), 1), 0)$$

is an atomic Σ_{arith}-formula. Note that we have not yet discussed the truth value of such formulas. The question of whether a formula is considered true or false will be explored in the next section.

In practice we will often use infix notation for binary predicate symbols. The atomic formula given above is then written as

$$(0 + x) * 1 = 0.$$ ◇

5.1.1 Bound Variables and Free Variables

In defining first-order logic formulas, it is necessary to distinguish between so-called bound and free variables. We introduce these concepts informally using an example from calculus. Consider the following equation:

$$\int_0^x y \cdot t \, dt = \frac{1}{2} \cdot x^2 \cdot y$$

In this equation, the variables x and y occur free, while the variable t is bound by the integral. This means the following: We can substitute arbitrary values for x and y in this equation without changing the validity of the formula. For example, substituting 2 for x, we get

$$\int_0^2 y \cdot t \, dt = \frac{1}{2} \cdot 2^2 \cdot y$$

and this equation is also valid. Conversely, it makes no sense to substitute a number for the bound variable t. The left side of the resulting equation would simply be undefined. We can only replace t with another variable. For example, replacing variable t with u, we get

$$\int_0^x y \cdot u \, du = \frac{1}{2} \cdot x^2 \cdot y$$

and this is effectively the same statement as above. However, this does not work with every variable. If we substitute the variable y for t, we get

$$\int_0^x y \cdot y \, dy = \frac{1}{2} \cdot x^2 \cdot y.$$

This statement is incorrect! The problem is that the previously free variable y becomes bound during the substitution.

A similar problem arises when we substitute arbitrary terms for y. As long as these terms do not contain the variable t, everything is fine. For example, if we substitute the term x^2 for y, we get

$$\int_0^x x^2 \cdot t \, dt = \frac{1}{2} \cdot x^2 \cdot x^2$$

and this formula is valid. However, if we substitute the term t^2 for y, we get

$$\int_0^x t^2 \cdot t \, dt = \frac{1}{2} \cdot x^2 \cdot t^2$$

and this formula is no longer valid. These examples show that we have to distinguish between bound and free variables.

In first-order logic, the quantifiers "$\forall$" (for all) and "$\exists$" (there exists) bind variables in a similar way as the integral operator "$\int \cdot \, \mathrm{d}t$". The above explanations show that there are two different types of variables: free variables and bound variables. To clarify these concepts, we first define for a Σ-term t the set of variables contained in t.

Definition 31 (*Var*(t)) *Given a signature $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle$ and t a Σ-term, we define the set $Var(t)$ of variables that occur in t by induction on the structure of the term:*

1. $Var(x) := \{x\}$ *for all* $x \in \mathcal{V}$,
2. $Var(c) := \{\}$ *for all* $c \in \mathcal{F}$ *such that* $arity(c) = 0$,
3. $Var\big(f(t_1, \cdots, t_n)\big) := Var(t_1) \cup \cdots \cup Var(t_n)$. ◇

5.1.2 Σ-Formulas

Definition 32 (Σ-Formula, $\mathbb{F}_\Sigma$, bound and free variables, *BV*(F), *FV*(F))
Let $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle$ be a signature. We denote the set of Σ-Formulas by $\mathbb{F}_\Sigma$. We define this set inductively. Simultaneously, for each formula $F \in \mathbb{F}_\Sigma$, we define the set $BV(F)$ of variables that occur bound in F and the set $FV(F)$ of variables that occur free in F.

1. $\bot \in \mathbb{F}_\Sigma$ *and* $\top \in \mathbb{F}_\Sigma$*, and we define*
$$FV(\bot) := FV(\top) := BV(\bot) := BV(\top) := \{\}.$$
2. *If* $p \in \mathcal{A}_\Sigma$ *and* $arity(p) = 0$*, then* $p \in \mathbb{F}_\Sigma$ *and*
$$FV(p) := \{\} \quad \text{and} \quad BV(p) := \{\}.$$
3. *If* $F = p(t_1, \cdots, t_n)$ *is an atomic* Σ*-formula, then* $F \in \mathbb{F}_\Sigma$*. Furthermore, we have*
   (a) $FV\big(p(t_1, \cdots, t_n)\big) := Var(t_1) \cup \cdots \cup Var(t_n)$,
   (b) $BV\big(p(t_1, \cdots, t_n)\big) := \{\}$.
4. *If* $F \in \mathbb{F}_\Sigma$*, then* $\neg F \in \mathbb{F}_\Sigma$*. Furthermore, we have*
   (a) $FV\big(\neg F\big) := FV(F)$,
   (b) $BV\big(\neg F\big) := BV(F)$.
5. *If* $F, G \in \mathbb{F}_\Sigma$ *and additionally*
$$\big(FV(F) \cup FV(G)\big) \cap \big(BV(F) \cup BV(G)\big) = \{\},$$
   then it also holds that
   (a) $(F \wedge G) \in \mathbb{F}_\Sigma$,
   (b) $(F \vee G) \in \mathbb{F}_\Sigma$,
   (c) $(F \to G) \in \mathbb{F}_\Sigma$,
   (d) $(F \leftrightarrow G) \in \mathbb{F}_\Sigma$.

   Furthermore, we define for all propositional connectives $\odot \in \{\wedge, \vee, \to, \leftrightarrow\}$*:*
   (a) $FV\big((F \odot G)\big) := FV(F) \cup FV(G)$.
   (b) $BV\big((F \odot G)\big) := BV(F) \cup BV(G)$.
6. *Let* $x \in \mathcal{V}$ *and* $F \in \mathbb{F}_\Sigma$ *with* $x \notin BV(F)$*. Then:*
   (a) $(\forall x : F) \in \mathbb{F}_\Sigma$,
   (b) $(\exists x : F) \in \mathbb{F}_\Sigma$.

   Furthermore, we define
   (a) $FV\big((\forall x : F)\big) := FV\big((\exists x : F)\big) := FV(F) \backslash \{x\}$,
   (b) $BV\big((\forall x : F)\big) := BV\big((\exists x : F)\big) := BV(F) \cup \{x\}$.

If the signature Σ *is clear from the context or irrelevant, we may also write* $\mathbb{F}$ *instead of* $\mathbb{F}_\Sigma$ *and simply refer to formulas instead of* Σ*-formulas.* ◇

In the definition given above, we have taken care that a variable cannot appear both free and bound in a formula simultaneously, because a straightforward induction on the structure of the formulas shows that for all $F \in \mathbb{F}_\Sigma$ the following holds:

$$FV(F) \cap BV(F) = \{\}.$$

Example: Continuing the example started above, we see that

$$(\exists x\colon\ \leq(+(y,x),y))$$

is a formula from $\mathbb{F}_{\Sigma_{\text{arith}}}$. The set of bound variables is $\{x\}$, the set of free variables is $\{y\}$. $\diamond$

If we would always write formulas in the prefix notation defined above, then readability would suffer disproportionately. To abbreviate, we agree that in first-order logic the same rules for saving parentheses apply that we have already used in propositional logic. In addition, the same quantifiers are combined: for example, we write

$$\forall x, y\colon p(x,y) \quad \text{instead of} \quad \forall x\colon (\forall y\colon p(x,y)).$$

Moreover, we agree that we may also indicate binary predicate and function symbols in infix notation. In order to guarantee a unique readability, we must then define the precedence and the associativity of the function symbols. For example, we write

$$\mathtt{n}_1 = \mathtt{n}_2 \quad \text{instead of} \quad =(\mathtt{n}_1, \mathtt{n}_2).$$

The formula $(\exists x\colon\ \leq(+(y,x),y))$ becomes more readable as

$$\exists x\colon y + x \leq y$$

Moreover, in the literature, you will often find expressions of the form

$$\forall x \in M : F \quad \text{or} \quad \exists x \in M : F.$$

These are abbreviations defined as

$$\big(\forall x \in M : F\big) \overset{\text{def}}{\Longleftrightarrow} \forall x : \big(x \in M \to F\big) \quad \text{and} \quad \big(\exists x \in M : F\big) \overset{\text{def}}{\Longleftrightarrow} \exists x : \big(x \in M \wedge F\big).$$

Finally, we agree that quantifiers bind more tightly than the operators $\wedge$, $\vee$, $\to$, and $\leftrightarrow$.

Exercise 15: **Consider the following signature for formalizing family relationships:**

- **Variables:** $\{u, v, w, x, y, z\}$.
- **Function Symbols:**
  - $\mathtt{mother}(x)$: the mother of x.
  - $\mathtt{father}(x)$: the father of x.
- **Unary Predicates:**
  - $\mathtt{female}(x)$: x is female.
  - $\mathtt{male}(x)$: x is male.
- **Binary Predicates:**
  - $\mathtt{brother}(x,y)$: x is a brother of y.
  - $\mathtt{sister}(x,y)$: x is a sister of y.
  - $\mathtt{sibling}(x,y)$: x is a sibling of y.
  - $\mathtt{married}(x,y)$: x is married to y.
  - $\mathtt{grandmother}(x,y)$: x is a grandmother of y.

- `grandfather`(x, y): x is a grandfather of y.
- `grandchild`(x, y): x is a grandchild of y,
- `uncle`(x, y): x is an uncle of y.
- `aunt`(x, y): x is an aunt of y.
- `cousin`(x, y): x is a cousin of y.

In the following, your task is to define certain predicates. In general, a definition of an n-ary predicate p has the form

$$\forall x_1, \cdots, x_n : \big(p(x_1, \cdots, x_n) \leftrightarrow F\big),$$

where F is a formula that does not contain the predicate p. In this exercise, this formula may only contain the function symbols `mother` and `father` and the predicate symbols `male` and `female`. For the purpose of this exercise we make the simplifying assumption that every person is either femal or male.

As an example, to `define` the predicate `brother` we can use the following formula:

$$\forall x, y : \big(\texttt{brother}(x, y) \leftrightarrow \texttt{male}(x) \wedge \texttt{father}(x) = \texttt{father}(y) \wedge \texttt{mother}(x) = \texttt{mother}(y)\big)$$

Define the following predicates:

(a) `sister`(x, y),

(b) `aunt`(x, y),

(c) `uncle`,

(d) `sibling`(x, y),

(e) `grandfather`(x, y),

(f) `grandmother`(x, y),

(g) `grandchild`(x, y),

(h) `cousin`(x, y). ◇

5.2 Semantics of First-Order Logic

Next, we define the meaning of the formulas. For this purpose, we introduce the concept of a Σ-structure. This kind of structure specifies how the function and predicate symbols of the signature Σ are to be interpreted.

Definition 33 (Structure) *Let a signature*

$$\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle$$

be given. A Σ-structure $\mathcal{S}$ is a pair $\langle \mathcal{U}, \mathcal{J} \rangle$, such that the following holds:

1. *$\mathcal{U}$ is a non-empty set. This set is also called the universe of the Σ-structure. This universe contains the values, which will later result from the evaluation of terms.*

2. *$\mathcal{J}$ is the interpretation of the function and predicate symbols. Formally, we define $\mathcal{J}$ as a mapping with the following properties:*

   (a) *Every function symbol $c \in \mathcal{F}$ with $arity(f) = 0$ is mapped to an element of $\mathcal{U}$, i.e. we have $\mathcal{J}(c) \in \mathcal{U}$. Instead of writing $\mathcal{J}(c)$ we will write $c^{\mathcal{J}}$.*

   (b) *Every function symbol $f \in \mathcal{F}$ with $arity(f) = m$ and $m > 0$ is mapped to an m-ary function*

   $$f^{\mathcal{J}} : \mathcal{U}^m \to \mathcal{U}$$

   that maps m-tuples of the universe $\mathcal{U}$ into the universe $\mathcal{U}$.

   (c) *Every predicate symbol $p \in \mathcal{P}$ with $arity(p) = 0$ is mapped to the set $\mathbb{B}$ of truth values, i.e. $p^{\mathcal{J}} \in \{\texttt{True}, \texttt{False}\}$.*

   (d) *Every predicate symbol $p \in \mathcal{P}$ with $arity(p) = n$ such that $n > 0$ is mapped to a subset*

   $$p^{\mathcal{J}} \subseteq \mathcal{U}^n.$$

   The idea is that an atomic formula of the form $p(t_1, \cdots, t_n)$ is interpreted as `True` exactly if the interpretation of the tuple $\langle t_1, \cdots, t_n \rangle$ is an element of the set $p^{\mathcal{J}}$.

   (e) *If the symbol "=" is a member of the set of predicate symbols $\mathcal{P}$, then the interpretation of the equality symbol "=" has to be canonical, i.e. we must have*

   $$=^{\mathcal{J}} = \big\{\langle u, u\rangle \mid u \in \mathcal{U}\big\}.$$

   A formula of the type $s = t$ is thus interpreted as true exactly when the interpretation of the term s yields the same value as the interpretation of the term t. ◇

Example: The signature Σ_G of group theory is defined as

$$\Sigma_G = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle \quad \text{with}$$

1. $\mathcal{V} := \{x, y, z\}$
2. $\mathcal{F} := \{\mathrm{e}, *\}$
3. $\mathcal{P} := \{=\}$
4. $arity = \big\{\langle \mathrm{e}, 0\rangle, \langle *, 2\rangle, \langle =, 2\rangle\big\}$

We can then define a Σ_G structure $\mathcal{Z} = \langle \{0, 1\}, \mathcal{J}\rangle$ by defining the interpretation $\mathcal{J}$ as follows:

1. $\mathrm{e}^{\mathcal{J}} := 0$,
2. $*^{\mathcal{J}} := \Big\{\langle 0, 0\rangle \mapsto 0,\ \langle 0, 1\rangle \mapsto 1,\ \langle 1, 0\rangle \mapsto 1,\ \langle 1, 1\rangle \mapsto 0\Big\}$,
3. $=^{\mathcal{J}} := \big\{\langle 0, 0\rangle, \langle 1, 1\rangle\big\}$.

   Note that we have no leeway in the interpretation of the equality symbol! ◇

If we want to evaluate terms that contain variables, we have to substitute values from the universe for these variables. Which values we substitute is determined by a variable assignment. We define this concept next.

Definition 34 (Variable Assignment) *Assume a signature*

$$\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle$$

is given. Furthermore, $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ *is a* Σ*-structure. Then a function of the form*

$$\mathcal{I} : \mathcal{V} \to \mathcal{U}$$

is called an $\mathcal{S}$ variable assignment.

If $\mathcal{I}$ *is an* $\mathcal{S}$ *variable assignment,* $x \in \mathcal{V}$ *and* $c \in \mathcal{U}$*, then* $\mathcal{I}[x/c]$ *denotes the variable assignment that maps the variable* x *to the value* c *and that otherwise agrees with* $\mathcal{I}$*:*

$$\mathcal{I}[x/c](y) := \begin{cases} c & \text{if } y = x; \\ \mathcal{I}(y) & \text{otherwise.} \end{cases}$$ ◇

Definition 35 (Interpretation of Terms) *If* $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle$ *is a signature* $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ *is a* Σ*-structure and* $\mathcal{I}$ *is an* $\mathcal{S}$ *variable assignment, then for every term* t *the value* $\mathcal{S}(\mathcal{I}, t)$ *is defined by induction on* t*:*

1. *If* $x \in \mathcal{V}$ *we define:*
$$\mathcal{S}(\mathcal{I}, x) := \mathcal{I}(x).$$
2. *If* $c \in \mathcal{F}$ *and* $arity(c) = 0$*, then* $\mathcal{S}(\mathcal{I}, c) := c^{\mathcal{J}}$.
3. *If* t *is a term of the form* $f(t_1, \cdots, t_n)$*, then we define*
$$\mathcal{S}\big(\mathcal{I}, f(t_1, \cdots, t_n)\big) := f^{\mathcal{J}}\big(\mathcal{S}(\mathcal{I}, t_1), \cdots, \mathcal{S}(\mathcal{I}, t_n)\big).$$ ◇

Example: Given the Σ_G-structure $\mathcal{Z}$ we define a $\mathcal{Z}$ variable assignment $\mathcal{I}$ as follows:

$$\mathcal{I} := \big\{x \mapsto 0,\ y \mapsto 1,\ z \mapsto 0\big\}.$$

The previous line is interpreted as follows:

$$\mathcal{I}(\mathtt{x}) := 0, \quad \mathcal{I}(\mathtt{y}) := 1, \quad \text{and} \quad \mathcal{I}(\mathtt{z}) := 0.$$

Then we have

$$\mathcal{Z}\big(\mathcal{I}, \mathtt{x} * \mathtt{y}\big) = 1.$$ ◇

Example: If we continue our previous example we have

$$\mathcal{Z}(\mathcal{I}, x * y = y * x) = \mathtt{True}.$$ ◇

To define the semantics of arbitrary Σ-formulas, we assume that we have the following functions available, just like in propositional logic:

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

The definitions of these functions have been given by the table in Figure 4.1 on page 36.

Definition 36 (Semantics of Σ-formulas) *Let $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ be a Σ-structure and $\mathcal{I}$ a $\mathcal{S}$-variable assignment. For each Σ-formula F the value $\mathcal{S}(\mathcal{I}, F)$ is defined by induction:*

1. $\mathcal{S}(\mathcal{I}, \top) := \texttt{True}$ *and* $\mathcal{S}(\mathcal{I}, \bot) := \texttt{False}$.
2. *If p is a propositional variable, then we define the value of p as follows:*
$$\mathcal{S}(\mathcal{I}, p) := p^{\mathcal{J}}.$$
3. *For each atomic Σ-formula of the form $p(t_1, \cdots, t_n)$ we define the value as follows:*
$$\mathcal{S}\big(\mathcal{I}, p(t_1, \cdots, t_n)\big) := \Big(\big\langle \mathcal{S}(\mathcal{I}, t_1), \cdots, \mathcal{S}(\mathcal{I}, t_n) \big\rangle \in p^{\mathcal{J}}\Big).$$
4. $\mathcal{S}(\mathcal{I}, \neg F) := \ominus\big(\mathcal{S}(\mathcal{I}, F)\big)$.
5. $\mathcal{S}(\mathcal{I}, F \wedge G) := \oslash\big(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G)\big)$.
6. $\mathcal{S}(\mathcal{I}, F \vee G) := \oslash\big(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G)\big)$.
7. $\mathcal{S}(\mathcal{I}, F \to G) := \ominus\big(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G)\big)$.
8. $\mathcal{S}(\mathcal{I}, F \leftrightarrow G) := \oslash\big(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G)\big)$.
9. $\mathcal{S}\big(\mathcal{I}, \forall x\colon F\big) := \begin{cases} \texttt{True} & \text{if } \mathcal{S}(\mathcal{I}[x/c], F) = \texttt{True} \quad \text{for all } c \in \mathcal{U} \text{ holds;} \\ \texttt{False} & \text{otherwise.} \end{cases}$
10. $\mathcal{S}\big(\mathcal{I}, \exists x\colon F\big) := \begin{cases} \texttt{True} & \text{if } \mathcal{S}(\mathcal{I}[x/c], F) = \texttt{True} \quad \text{for some } c \in \mathcal{U} \text{ holds;} \\ \texttt{False} & \text{otherwise.} \end{cases}$ ◇

Example: Continuing from the above example, we have

$$\mathcal{Z}\big(\mathcal{I}, \forall \mathtt{x} : \mathrm{e} * x = x\big) = \texttt{True}.$$ ◇

Definition 37 (Universally valid) *If F is a Σ-formula such that for every Σ-structure $\mathcal{S}$ and for every $\mathcal{S}$-variable assignment $\mathcal{I}$*

$$\mathcal{S}(\mathcal{I}, F) = \texttt{True}$$

holds, we call F universally valid. In this case, we write

$$\models F.$$ ◇

If F is a formula such that $FV(F) = \{\}$, then the value $\mathcal{S}(\mathcal{I}, F)$ obviously does not depend on the interpretation $\mathcal{I}$. We also refer to such formulas as closed formulas. In this case, we write $\mathcal{S}(F)$ instead of $\mathcal{S}(\mathcal{I}, F)$. If additionally $\mathcal{S}(F) = \texttt{True}$, we also say that $\mathcal{S}$ is a model of F. This is written as

$$\mathcal{S} \models F.$$

The definitions of the terms "satisfiable" and "equivalent" can now be transferred from propositional logic. To avoid unnecessary complexity in the definitions, we assume a fixed signature Σ as given. Hence, in the following definitions when we talk about terms, formulas, structures, etc., we mean Σ-terms, Σ-formulas, and Σ-structures.

Definition 38 (Equivalent) *Two formulas F and G, in which the variables $x_1, \cdots, x_n$ appear free, are called equivalent if and only if*

$$\models \forall x_1 : \cdots \forall x_n : (F \leftrightarrow G)$$

holds. If no variables appear free in F and G, then F is equivalent to G if and only if

$$\models F \leftrightarrow G$$

holds. If F and G are equivalent, then this will be written as

$$F \Leftrightarrow G.$$ ◇

Remark: All propositional logic equivalences are also first-order logic equivalences. ◇

Definition 39 (Satisfiable) *A set $M \subseteq \mathbb{F}_\Sigma$ is satisfiable, if there exists a structure $\mathcal{S}$ and a variable assignment $\mathcal{I}$ such that*

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

holds. Otherwise, M is called unsatisfiable or contradictory. This is written as

$$M \models \bot$$ ◇

Our goal is to provide a method that allows us to check whether a set M of formulas is contradictory, i.e., whether $M \models \bot$ holds. It turns out that this is generally not possible; the question of whether $M \models \bot$ holds is undecidable. A proof of this fact is based on the unsolvability of the halting problem but the details are beyond the scope of this lecture. However, as in propositional logic, it is possible to specify a calculus $\vdash$ such that:

$$M \vdash \bot \quad \text{iff} \quad M \models \bot.$$

This calculus can then be used to implement a semi-decision procedure: To check if $M \models \bot$ holds, we try to derive the formula $\bot$ from the set M. If we proceed systematically by trying all possible proofs, if indeed $M \models \bot$ holds, we will eventually find a proof that demonstrates $M \vdash \bot$. However, if M is satisfiable, i.e. if we have

$$M \not\models \bot$$

we generally will not be able to detect this, because the set of all possible proofs is infinite and we can never try all proofs. We can only ensure that we attempt every proof eventually. But if there is no proof, we can never be certain of this fact, because at any given time we have only tried a part of the possible proofs.

The situation is similar to that in verifying certain number-theoretical questions. Consider the following specific example: A number n is called a perfect number, if the sum of all proper divisors of n equals n itself. For example, the number 6 is perfect, since the set of proper divisors of 6 is $\{1, 2, 3\}$ and

$$1 + 2 + 3 = 6.$$

So far, all known perfect numbers are divisible by 2. The question of whether there are odd numbers that are perfect is an open problem. To solve this problem, we could write a program that sequentially checks whether each odd number is perfect. Figure 5.1 on page 115 shows such a program. If there is an odd perfect number, then this program will eventually find it. However, if no odd perfect number

exists, then the program will run indefinitely and we will never know for sure that there are no odd perfect numbers.

```
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: The search for an odd perfect number.

5.3 Implementing Σ-Structures in *Python*

The concept of a Σ-structure that has been presented in the last section is quite abstract. In this section, we implement Σ-structures in *Python*. This helps to illustrate the concept. As a concrete example, we will consider Σ-structures related to group theory. We proceed in four steps:

1. First, we define the concept of a group.
2. Then, we discuss how the formulas of group theory are represented in *Python*.
3. Next, we define a Σ-structure in which the formulas of group theory are valid.
4. Finally, we show how to evaluate formulas from first-order logic in *Python*.

5.3.1 Group Theory

In mathematics, a group $\mathcal{G}$ is defined as a triple of the form

$$\mathcal{G} = \langle G, \mathrm{e}, \circ \rangle$$

where:

1. G is a set,
2. e is an element of the set G, and
3. $\circ : G \times G \to G$ is a binary function on G, which we henceforth refer to as the multiplication of the group.

4. Moreover, the following three axioms hold:

   (a) $\forall x : \mathrm{e} \circ x = x$,
   e is a left-neutral element with respect to multiplication.

   (b) $\forall x : \exists y : y \circ x = \mathrm{e}$
   for every $x \in G$ there is a left-inverse element.

   (c) $\forall x : \forall y : \forall z : (x \circ y) \circ z = x \circ (y \circ z)$
   the operator $\circ$ is associative.

   (d) The group $\mathcal{G}$ is a commutative group if and only if additionally the following axiom holds:
   $\forall x : \forall y : x \circ y = y \circ x$
   This is know as the law of commutativity. ◇

   Note that the law of commutativity does not have to hold in a group. It is only required for commutative groups.

5.3.2 Representation of Formulas in *Python*

In the last section, we have defined the signature Σ_G of group theory as follows:

$$\Sigma_G = \big\langle \{x, y, z\},\ \{\mathrm{e}, \circ\},\ \{=\},\ \{\langle \mathrm{e}, 0\rangle, \langle \circ, 2\rangle, \langle =, 2\rangle\}\big\rangle.$$

Here, "e" is a 0-arity function symbol representing the identity element, "$\circ$" is the binary function symbol representing the group multiplication, and "$=$" is the binary predicate symbol to denote equality. We will use a parser for first-order logic formulas that does not support binary infix operators like "$\circ$" or "$=$". With this parser, terms can only be written in prefix form, i.e. as

$$f(t_1, \cdots, t_n)$$

where f is a function symbol and $t_1, \cdots, t_n$ are terms. Similarly, atomic formulas have to be represented by expressions in the form

$$p(t_1, \cdots, t_n)$$

where p is a predicate symbol. Variables are distinguished from function and predicate symbols by the fact that variables begin with a lowercase letter, while function and predicate symbols begin with an uppercase letter. To be able to represent the formulas of group theory, we therefore agree on the following:

1. The neutral element e is written as `E()`.

   The parser we use requires that a constant is followed by parentheses.

2. For the operator $\circ$, we use the binary function symbol `Multiply`. Thus, the expression $x \circ y$ is written as `Multiply`(x, y).

3. The equality sign "$=$" is represented by the binary predicate symbol `Equals`. Thus, for example, the formula $x = y$ is written as `Equals`(x, y).

Figure 5.2 shows the formulas of group theory as strings. We can transform the formulas into nested tuples using the function `parse`(s) shown in Figure 5.3. The result of this transformation is displayed in Figure 5.4.

```
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: The string representation of the formulas of group theory.

```
% run FOL-Parser.ipynb

def parse(s):
    p = LogicParser(s)
    return p.parse()

F1 = parse(G1)
F2 = parse(G2)
F3 = parse(G3)
F4 = parse(G4)
```

Figure 5.3: The function `parse`

```
F1 = ('∀', 'x', ('Equals', ('Multiply', ('E',), 'x'), 'x'))
F2 = ('∀', 'x', ('∃', 'y', ('Equals', ('Multiply', 'x', 'y'), ('E',))))
F3 = ('∀', 'x', ('∀', 'y', ('∀', 'z',
          ('Equals', ('Multiply', ('Multiply', 'x', 'y'), 'z'),
                     ('Multiply', 'x', ('Multiply', 'y', 'z'))
          )
     )))
F4 = ('∀', 'x', ('∀', 'y',
          ('Equals', ('Multiply', 'x', 'y'),
                     ('Multiply', 'y', 'x')
          )
     ))
```

Figure 5.4: Die Axiome einer kommutativen Gruppe als geschachtelte Tupel

5.3.3 Representation of Σ-Structures in *Python*

In the section defininig the semantics of first-order logic in Section 5.2, we have already specied a Σ-structure $\mathcal{Z}$ for group theory. The universe of $\mathcal{Z}$ consists of the set $\{0,1\}$. In *Python*, we can implement this structure using the code shown in Figure 5.5 on page 118.

```
1  U = { 0, 1 }
2  NeutralElement = { (): 0 }
3  Product        = { (0, 0): 0,  (0, 1): 1,  (1, 0): 1,  (1, 1): 0 }
4  Identity       = { (0, 0), (1, 1) }
5  J = { "E": NeutralElement, "Multiply": Product, "Equals": Identity }
6  S = (U, J)
7  I = { "x": 0, "y": 1, "z": 0 }
```

Figure 5.5: Implementierung einer Struktur zur Gruppen-Theorie

1. The universe U defined in line 1 consists of the two numbers 0 and 1.
2. In line 2, we define the interpretation of the zero-arity function symbol `E` as the *Python* dictionary that maps the empty tuple to the number 0.
3. In line 3, we define a function `Product` as a *Python* dictionary. We have
   $$\texttt{Product}(0,0) = 0, \quad \texttt{Product}(0,1) = 1,$$
   $$\texttt{Product}(1,0) = 1, \quad \texttt{Product}(1,1) = 0.$$
   We use this function later as the interpretation $\texttt{Multiply}^{\mathcal{J}}$ of the function symbol "`Multiply`".
4. In line 4, we have defined the interpretation $\texttt{Equals}^{\mathcal{J}}$ of the predicate symbol "`Equals`" as the set $\{\langle 0,0\rangle, \langle 1,1\rangle\}$.
5. In line 5, we combine the interpretations of the function symbols "`E`" and "`Multiply`" and the predicate symbol "`Equals`" into the dictionary `J`, so that for a function or predicate symbol f, the interpretation $f^{\mathcal{J}}$ is given by the value $\texttt{J}[f]$.
6. The interpretation `J` is then combined with the universe `U` into the structure `S`, which is simply represented as a pair in *Python*.
7. Finally, line 7 shows that a variable assignment can also be represented as a dictionary. The keys are the variables, and the values are the objects from the universe to which these variables are mapped.

Next, we consider how we can evaluate terms within a Σ-structure. Figure 5.6 shows the implementation of the procedure `evalTerm`$(t, \mathcal{S}, \mathcal{I})$, which takes as arguments a term t, a Σ-structure $\mathcal{S}$, and a variable assignment $\mathcal{I}$. The term t is represented in *Python* as a nested tuple.

```
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  = tuple(evalTerm(arg, S, I) for arg in Args)
8      return fJ[ArgVals]
```

Figure 5.6: Evaluation of a term.

1. In line 2, we check whether the term t is a variable. This can be done by noting that variables are represented as strings, while all other terms are represented as tuples. If t is a variable, then we return the value stored in the variable assignment $\mathcal{I}$ for this variable.
2. Otherwise, in line 4, we extract the dictionary $\mathcal{J}$ that contains the interpretations of the function and predicate symbols from the structure $\mathcal{S}$.
3. The function symbol f of the term t is the first component of the tuple t, the arguments are collected in the list `Args`.
4. The interpretation $f^{\mathcal{J}}$ of this function symbol is looked up in line 6 in the dictionary $\mathcal{J}$.
5. The arguments of f are recursively evaluated in line 7. As a result, we obtain a tuple of values.
6. This tuple then serves in line 8 as the argument for the dictionary $f^{\mathcal{J}}$. The value stored in this dictionary for the given tuple of arguments is the result of evaluating the term 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  = tuple(evalTerm(arg, S, I) for arg in Args)
6      return ArgVals in pJ
```

Figure 5.7: Evaluation of an atomic formula.

Figure 5.7 shows the evaluation of an atomic formula. An atomic formula a is represented in Python as a tuple of the form

$$a = (p, t_1, \cdots, t_n).$$

We can decompose this tuple into its components by the assignment

```
p, *Args = a
```

where `Args` is then the list $[t_1, \cdots, t_n]$. To verify whether the atomic formula a is true, we need to check whether

$$\big(\texttt{evalTerm}(t_1, \mathcal{S}, \mathcal{I}), \cdots, \texttt{evalTerm}(t_n, \mathcal{S}, \mathcal{I})\big) \in p^{\mathcal{J}}$$

holds. This test is conducted in line 6. Note that the implementation of the function `evalAtomic` is very similar to the implementation of the function `evalTerm`.

```
1  def evalFormula(F, S, I):
2      U, _ = S # U is the universe
3      match F:
4          case ('⊤', ):     return True
5          case ('⊥', ):     return False
6          case ('¬', G):    return not evalFormula(G, S, I)
7          case ('∧', G, H): return evalFormula(G, S, I) and evalFormula(H, S, I)
8          case ('∨', G, H): return evalFormula(G, S, I) or evalFormula(H, S, I)
9          case ('→', G, H): return not evalFormula(G, S, I) or evalFormula(H, S, I)
10         case ('↔', G, H): return evalFormula(G, S, I) == evalFormula(H, S, I)
11         case ('∀', x, G): return all(evalFormula(G, S, modify(I, x, c)) for c in U)
12         case ('∃', x, G): return any(evalFormula(G, S, modify(I, x, c)) for c in U)
13     return evalAtomic(F, S, I)
```

Figure 5.8: The function `evalFormula`.

Figure 5.8 on page 120 shows the implementation of the function $\texttt{evalFormula}(F, \mathcal{S}, \mathcal{I})$, which receives as arguments a first-order formula F, a Σ-structure $\mathcal{S}$, and a variable assignment $\mathcal{I}$. The function computes the result $\mathcal{S}(\mathcal{I}, F)$. The evaluation of the formula F proceeds analogously to the evaluation of propositional logic formulas shown in Figure 4.1 on page 40. The novelty here is the handling of quantifiers. In line 11, we deal with the evaluation of universally quantified formulas. If F is a formula of the form $\forall x : G$, then the formula F is represented by the tuple

$$F = (\text{'}\forall\text{'}, x, G).$$

The evaluation of $\forall x \colon G$ implements the formula

$$\mathcal{S}\big(\mathcal{I}, \forall x \colon G\big) := \begin{cases} \texttt{True} & \text{if } \mathcal{S}(\mathcal{I}[x/c], G) = \texttt{True} \quad \text{for all } c \in \mathcal{U} \text{ holds;} \\ \texttt{False} & \text{otherwise.} \end{cases}$$

To implement this, we use the procedure `modify()`, which modifies the variable assignment $\mathcal{I}$ at position x to return c, hence

$$\texttt{modify}(\mathcal{I}, x, c) = \mathcal{I}[x/c].$$

The implementation of this procedure is shown in Figure 5.9 on page 121. In the evaluation of a universal quantifier, we can take advantage of the fact that the *Python* language supports the quantifier "∀" through the function `all`. Thus, we can directly test whether the formula is true for all possible values c that we can substitute for the variable x. For a set S of truth values, the expression

`all`(S)

is true exactly when all elements of S are `True`.

The evaluation of $\exists x\colon G$ implements the formula

$$\mathcal{S}\big(\mathcal{I}, \exists x\colon G\big) := \begin{cases} \texttt{True} & \text{if } \mathcal{S}(\mathcal{I}[x/c], G) = \texttt{True} \quad \text{for any } c \in \mathcal{U} \text{ holds;} \\ \texttt{False} & \text{otherwise.} \end{cases}$$

The evaluation of $\exists x\colon G$ is then similar to the evaluation of $\forall x\colon G$, except for the fact that we now have to use the function `any` instead of the function `all`. For a list, set, tuple, or indeed any iterable L the expression `any(L)` is true iff `True` is an element of L.

In the implementation of the procedure `modify`(I, x, c), which calculates the result as the variable assignment $\mathcal{I}[x/c]$, we take advantage of the fact that for a function stored as a dictionary, the value assigned to an argument x can be changed by an assignment of the form

$$\mathcal{I}[x] = c.$$

However, we must not change the variable assignment $\mathcal{I}$. Instead, we must return a new variable assignment $\mathcal{J}$ that returns the same values as the variable assignment $\mathcal{I}$, except for the argument x, where it should return c instead of $\mathcal{I}[x]$. Therefore, we have to create a copy of $\mathcal{I}$ in line 2. This copy is then modified in line 3 and returned in line 4.

```
1  def modify(I, x, c):
2      J = I.copy() # do not modify I
3      J[x] = c
4      return J
```

Figure 5.9: Implementation of the function `modify`.

The script shown in Figure 5.10 can check whether the Σ-structure defined in 5.10 on page 121 is a group. The output shown in Figure 5.11 allows us to conclude that this structure is indeed a commutative group.

```
1  f"evalFormula({G1}, S, I) = {evalFormula(F1, S, I)}"
2  f"evalFormula({G2}, S, I) = {evalFormula(F2, S, I)}"
3  f"evalFormula({G3}, S, I) = {evalFormula(F3, S, I)}"
4  f"evalFormula({G4}, S, I) = {evalFormula(F4, S, I)}"
```

Figure 5.10: Checking whether the Σ-structure shown in Figure 5.5 is a group.

Remark: The program shown above is available as a Jupyter Notebook on GitHub at the address:

`github.com/karlstroetmann/Logic/blob/master/Python/Chapter-5/01-FOL-Evaluation.ipynb`

With this program we can check, whether a first-order formula is satisfied in a given finite structure. However, we cannot use it to check whether a formula is universally valid because, on one hand, we

```
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: Output of the script shown in Figure 5.10.

cannot apply the program if the Σ-structure has an infinite universe, and on the other hand, even the number of different finite Σ-structures we would need to test is infinitely large. ◇

Exercise 16:

(a) Show that the formula

$$\forall x : \exists y : p(x,y) \rightarrow \exists y : \forall x : p(x,y)$$

is not universally valid.

In order to do this, implement a suitable Σ-structure $\mathcal{S}$ in *Python* such that this formula is false.

(b) Consider how many different structures there are for the signature of group theory if we assume that the universe is of the form $\{1, \cdots, n\}$.

(c) Provide a satisfiable first-order formula F that is always false in a Σ-structure $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ if the universe $\mathcal{U}$ is finite.

Hint: Let $f : U \rightarrow U$ be a function. Think about how the statements "*f is injective*" and "*f is surjective*" are related when the universe is finite. ◇

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 some 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 and provide two examples:
   - We introduce the map coloring problem for the case of Australia.
   - We formulate the eight queens puzzle as a constraint satisfaction problem.

2. We discuss a simple constraint solver that is based on backtracking.

   More efficient constraint solvers will be discussed in the lecture on artificial intelligence in the 6th semester. Furthermore, later in this chapter we will introduce Z3, which is a very powerful constraint solver developed by Microsoft.

3. Finally, we demonstrate a number of puzzles that 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 set of free variables. Furthermore, a Σ-structure $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ consisting of a universe $\mathcal{U}$ and the interpretation $\mathcal{J}$ of the function and predicate symbols used in these formulas is assumed to be specified by the context of the problem. The goal is to find a variable assignment such that the given first-order formulas are evaluated as true. The formal definition follows.

Definition 40 (CSP)
A constraint satisfaction problem (abbreviated as CSP*) is defined as a triple*

$$\mathbb{P} := \langle \Sigma, \mathcal{S}, \mathcal{C} \rangle$$

where

1. $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle$ *is signature,*
2. $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ *is* Σ*-structure,*
3. $\mathcal{C} \subseteq \mathbb{F}_\Sigma$*, i.e.* $\mathcal{C}$ *is a set of* Σ*-formulas from first order logic. These formulas are called the constraints of* $\mathbb{P}$*.* ◇

In the following, we will often specify a constraint satisfaction problem by just specifying the variables $\mathcal{V}$, the set of values $\mathcal{U}$ that these variables can take, and the set $\mathcal{C}$ of constraints. In this case, we assume that the sets $\mathcal{F}$ and $\mathcal{P}$ of function and predicate symbols and their interpretation $\mathcal{J}$ are implicitly given. In this case the constraint satisfaction problem is given as the triple

$$\mathbb{P} = \langle \mathcal{V}, \mathcal{U}, \mathcal{C} \rangle.$$

Given a CSP

$$\mathbb{P} = \langle \mathcal{V}, \mathcal{U}, \mathcal{C} \rangle,$$

a variable assignment for $\mathbb{P}$ is a function

$$\mathcal{I} : \mathcal{V} \to \mathcal{U}.$$

A variable assignment $\mathcal{I}$ is a solution of the CSP $\mathbb{P} = \langle \mathcal{V}, \mathcal{U}, \mathcal{C} \rangle$ if, given the assignment $\mathcal{I}$, all constraints from $\mathcal{C}$ are satisfied, i.e. we have

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

Finally, a partial variable assignment $\mathcal{B}$ for $\mathcal{P}$ is a function

$$\mathcal{B} : \mathcal{V} \to \mathcal{U} \cup \{\Omega\} \quad \text{where } \Omega \text{ denotes the undefined value.}$$

Hence, a partial variable assignment does not assign values to all variables. Instead, it assigns values only to a subset of the set $\mathcal{V}$. 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

$$\texttt{dom}(\mathcal{B}) := \{x \in \mathcal{V} \mid \mathcal{B}(x) \neq \Omega\}.$$

We proceed to illustrate the definitions given so far by presenting two examples.

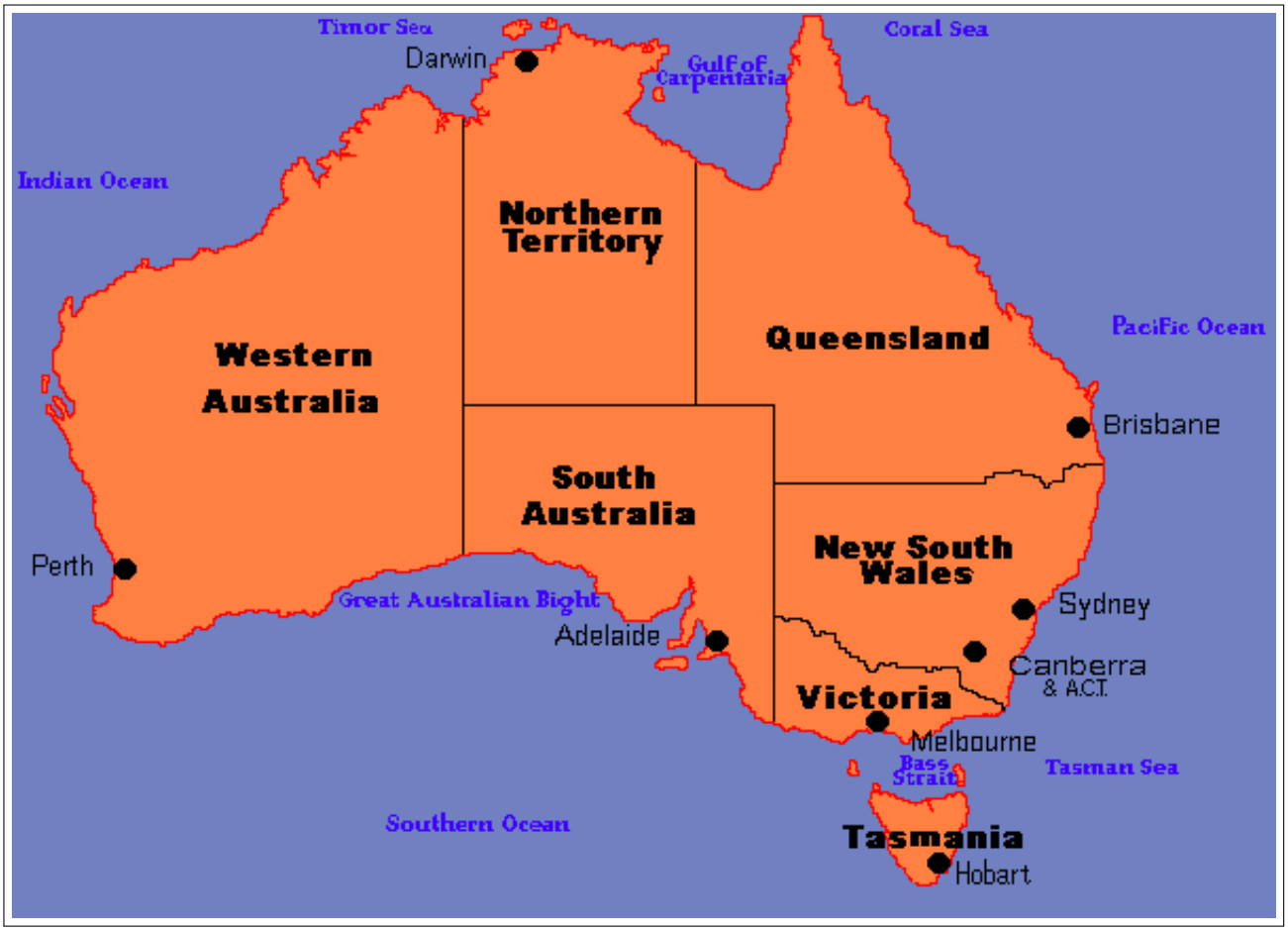

Figure 5.12: A map of Australia.

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 124 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. $\mathcal{V} := \{\texttt{WA}, \texttt{NT}, \texttt{SA}, \texttt{Q}, \texttt{NSW}, \texttt{V}, \texttt{T}\}$,
2. $\mathcal{U} := \{\texttt{red}, \texttt{green}, \texttt{blue}\}$,
3. $\mathcal{C} := \big\{\texttt{WA} \neq \texttt{NT},\ \texttt{WA} \neq \texttt{SA}, \texttt{NT} \neq \texttt{SA},\ \texttt{NT} \neq \texttt{Q}, \texttt{SA} \neq \texttt{Q},\ \texttt{SA} \neq \texttt{NSW},\ \texttt{SA} \neq \texttt{V},\ \texttt{Q} \neq \texttt{NSW},\ \texttt{NSW} \neq \texttt{V}\big\}$.
   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 $\mathbb{P} := \langle \mathcal{V}, \mathcal{U}, \mathcal{C} \rangle$ 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 126 shows this solution.

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:

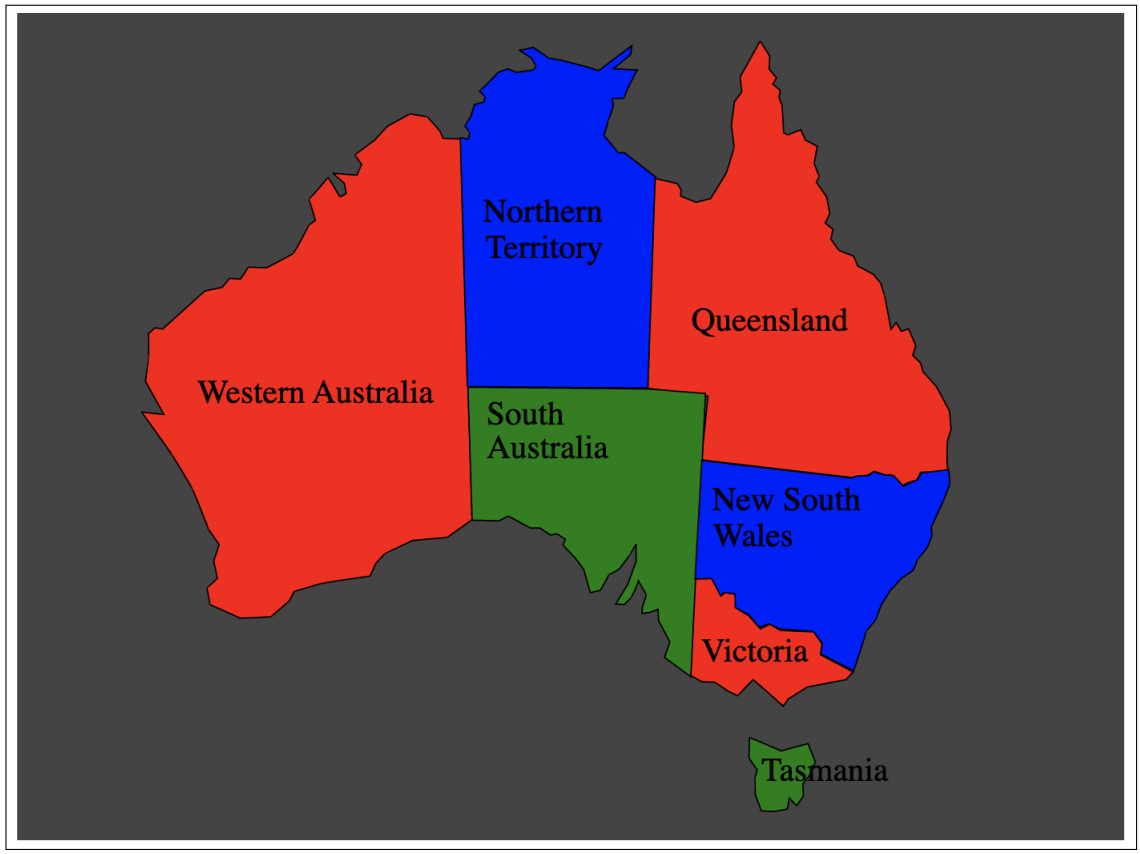

Figure 5.13: A map coloring for Australia.

$$\mathcal{V} := \{\mathtt{Q}_1, \mathtt{Q}_2, \mathtt{Q}_3, \mathtt{Q}_4, \mathtt{Q}_5, \mathtt{Q}_6, \mathtt{Q}_7, \mathtt{Q}_8\},$$

where for $i \in \{1, \cdots, 8\}$ the variable $\mathtt{Q}_i$ specifies the column of the queen that is placed in row i. As the columns run from one to eight, we define the set $\mathcal{U}$ as

$$\mathcal{U} := \{1, 2, 3, 4, 5, 6, 7, 8\}.$$

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

   $$\mathtt{SameColumn} := \big\{\mathtt{Q}_i \neq \mathtt{Q}_j \;\big|\; i, j \in \{1, \cdots, 8\} \wedge j < i\big\}.$$

   Here the condition $i < j$ ensures that, for example, we have the constraint $\mathtt{Q}_2 \neq \mathtt{Q}_1$ but not the constraint $\mathtt{Q}_1 \neq \mathtt{Q}_2$, 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

   $$|i - j| = |\mathtt{Q}_i - \mathtt{Q}_j|$$

   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 $|\mathtt{Q}_i - \mathtt{Q}_j|$ is the absolute value of the difference of the

columns of these queens. To capture these constraints, we define

$$\texttt{SameDiagonal} := \big\{ |i-j| \neq |\texttt{Q}_i - \texttt{Q}_j| \;\big|\; i,j \in \{1,\cdots,8\} \wedge j < i \big\}.$$

Then, the set of constraints is defined as

$$\mathcal{C} := \texttt{SameColumn} \cup \texttt{SameDiagonal}$$

and the eight queens problem can be stated as the constraint satisfaction problem

$$\mathbb{P} := \langle \mathcal{V}, \mathcal{U}, \mathcal{C} \rangle.$$

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 127.

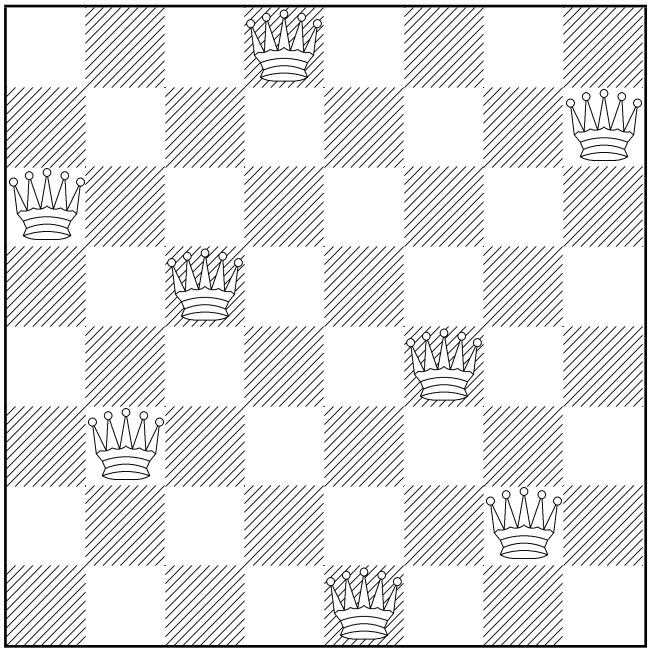

Figure 5.14: A solution of the eight queens problem.

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 in *Python* 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 128.

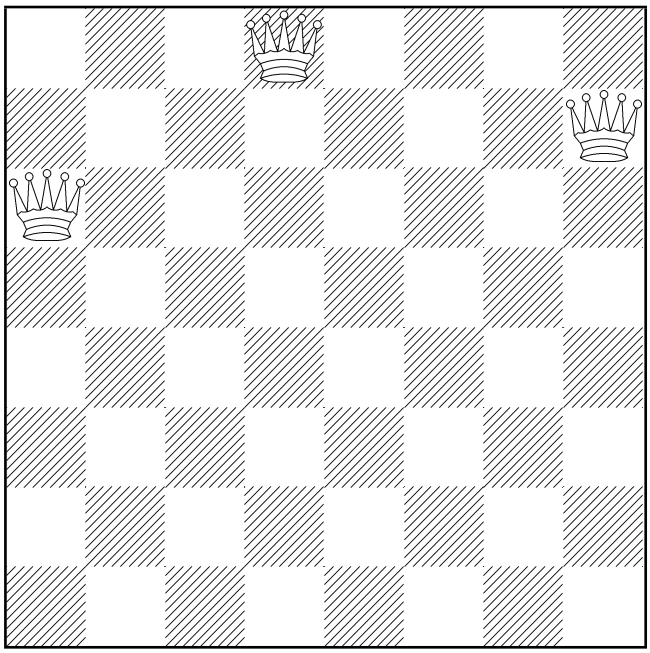

Figure 5.15: The partial assignment $\{\mathtt{Q}_1 \mapsto 4, \mathtt{Q}_2 \mapsto 8, \mathtt{Q}_3 \mapsto 1)\}$.

Figure 5.16 on page 128 shows a *Python* program that can be used to create the eight queens puzzle as a CSP.

```
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.

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 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

$$\mathcal{B} = \{x_1 : v_1, \cdots, x_k : v_k\}$$

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 129 shows a simple CSP solver that employs backtracking. We discuss this program next.

```
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

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.

   The details of this implementation are quite technical and are not important in the following.

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.

   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`.

```
1  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`

Next, we discuss the implementation of the procedure `backtrack_search` that is shown in Figure 5.18 on page 130. This procedure receives a partial assignment `Assignment` as input together with an

augmented CSP. This partial assignment is consistent with CSP: If f is a constraint of CSP such that all the variables occurring in f are assigned to in Assignment, then evaluating f using Assignment yields True. Initially, this partial assignment is empty and hence trivially consistent. The idea is to extend this partial assignment until it is a complete assignment that satisfies all constraints of the given CSP.

1. First, the augmented CSP is split into its components.
2. Next, if Assignment is already a complete variable assignment, i.e. if the dictionary Assignment has as many elements as there are variables, then the fact that Assignment is partially consistent implies that it is a solution of the CSP and, therefore, it is returned.
3. Otherwise, we have to extend the partial Assignment. In order to do so, we first have to select a variable var that has not yet been assigned a value in Assignment so far. We pick the first variable in the list Variables that is yet unassigned. This variable is called var.
4. Next, we try to assign a value to the selected variable var. After assigning a value to var, we immediately check whether this assignment would be consistent with the constraints using the procedure isConsistent. If the partial Assignment turns out to be consistent, the partial Assignment is extended to the new partial assignment NewAssign that satisfies

   ```
   NewAssign[var] = value
   ```

   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.

```
1  def isConsistent(var, value, Assignment, Constraints):
2      NewAssign      = Assignment.copy()
3      NewAssign[var] = value
4      return all(eval(f, NewAssign) for (f, Vs) in Constraints
5                                    if var in Vs and Vs <= NewAssign.keys()
6                 )
```

Figure 5.19: The procedure isConsistent

We still need to discuss the implementation of the auxiliary procedure isConsistent shown in Figure 5.19 on page 131. This procedure takes a variable var, a value, a partial Assignment and a set of constraints 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

$$\texttt{NewAssign} := \texttt{Assignment} \cup \{\langle \texttt{var}, \texttt{value} \rangle\}$$

that assigns `value` to the variable `var` is still partially consistent with $\mathcal{C}$. To this end, the `for`-loop iterates over all `Formula`s in $\mathcal{C}$. However, we only have to check those `Formula`s that contain the variable `var` and, furthermore, have the property that

$$\mathcal{V}(\texttt{Formula}) \subseteq \texttt{dom}(\texttt{NewAssign}),$$

i.e. all variables occurring in `Formula` need to have a value assigned in `NewAssign`. 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`: As `Assignment` is assumed to be partially consistent with respect to `Formula`, `NewAssign` is also partially consistent with respect to `Formula`.

2. If the formula `f` contains variables that have not been assigned yet, then `f` can not be evaluated anyway.

If we use backtracking, we can solve the 8 queens puzzle in less than a second. For the 8 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`.

Exercise 17: We have already discussed the Zebra Puzzle in section 4.8.2 of the previous chapter. Your task is to reformulate this puzzle as a constraint programming problem. You should start from the following notebook:

`github.com/karlstroetmann/Logic/blob/master/Python/Chapter-5/Zebra-CSP.ipynb`

While it is important to order the variables in a sensible way, you shouldn't spend to much time with this task. ◇

Exercise 18: Figure 5.20 shows a cryptarithmetic puzzle. The idea is that the letters "`S`", "`E`", "`N`", "`D`", "`M`", "`O`", "`R`", "`Y`" are interpreted as variables ranging over the set of decimal digits, i.e. these variables can take values in the set $\{0,1,2,3,4,5,6,7,8,9\}$. Then, the string "`SEND`" is interpreted as a decimal number, i.e. it is interpreted as the number

$$\texttt{S} \cdot 10^3 + \texttt{E} \cdot 10^2 + \texttt{N} \cdot 10^1 + \texttt{D} \cdot 10^0.$$

The strings "`MORE` and "`MONEY`" are interpreted similarly. To make the problem interesting, the assumption is that different variables have different values. Furthermore, the digits at the beginning of a number should be different from 0.

Figure 5.20: A cryptarithmetic puzzle

Your task is to reformulate this puzzle as a constraint programming problem. You should start from the following notebook:

github.com/karlstroetmann/Logic/blob/master/Python/Chapter-5/Crypto-Arithmetic.ipynb

It is importtant that you add the digits one by one as in elementary school. This way, the addition is separated into a number of constraints that can then be solved by backtracking. ◇

5.5 Solving Search Problems by Constraint Programming

In this section we show how we can formulate certain search problems as CSPs. 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 41 (Symbolic Transition System) *A symbolic transition system is a 6-tuple*

$$\mathcal{T} = \langle \mathcal{V}, \mathcal{U}, \texttt{Start}, \texttt{Goal}, \texttt{Invariant}, \texttt{Transition} \rangle$$

such that:

(a) $\mathcal{V}$ *is a set of variables.*

These variables are strings. For every variable $x \in \mathcal{V}$ *there is a primed variable* x' *which does not occur in* $\mathcal{V}$*. The set of these primed variables is denoted as* $\mathcal{V}'$*.*

(b) $\mathcal{U}$ *is a set of values that these variables can take.*

(c) `Start`*,* `Goal`*, and* `Invariant` *are first-order formulas such that all free variables occurring in these formulas are elements from the set* $\mathcal{V}$*.*

- `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 first-order formula. The free variables of this formula are elements of the set* $\mathcal{V} \cup \mathcal{V}'$*, i.e. they are either variables from the set* $\mathcal{V}$ *or they are primed variables from the set* $\mathcal{V}'$*.*

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) $\mathcal{V} := \{\mathtt{M}, \mathtt{C}, \mathtt{B}\}$.

The value of $\mathtt{M}$ is the number of missionaries on the western shore, the value of $\mathtt{C}$ is the number of infidels on that shore, while the value of $\mathtt{B}$ is the number of boats on the western shore.

(b) $\mathcal{U} := \{0, 1, 2, 3\}$.

(c) $\mathtt{Start} := (\mathtt{M} = 3 \wedge \mathtt{C} = 3 \wedge \mathtt{B} = 1)$.

At the beginning, there are 3 missionaries, 3 infidels, and 1 boat on the western shore.

(d) $\mathtt{Goal} := (\mathtt{M} = 0 \wedge \mathtt{C} = 0 \wedge B = 0)$.

The goal is to transfer everybody to the eastern shore. Hence the number of missionaries, infidels, and boats on the western shore will then be 0.

(e) $\mathtt{Invariant} := \big((\mathtt{M} = 3 \vee \mathtt{M} = 0 \vee \mathtt{M} = \mathtt{C}) \ \wedge \ \mathtt{B} \leq 1\big)$.

The first part of the invariant, i.e. the formulas $\mathtt{M} = 3 \vee \mathtt{M} = 0 \vee \mathtt{M} = \mathtt{C}$, describes those states where the missionaries are not threatened by the infidels.

- $\mathtt{M} = 3$: All missionaries are on the western shore.
- $\mathtt{M} = 0$: All missionaries are together on the eastern shore.
- $\mathtt{M} = \mathtt{C}$: On both shores the numbers of missionaries and cannibals are the same.
  If neither $\mathtt{M} = 3$ nor $\mathtt{M} = 0$ holds, then the number of missionaries and cannibals have to be the same on the western shorem because if there were more missionaries on the western shore than cannibals, then there would be less missionaries than cannibals on the eastern shore and hence there would be a problem on the eastern shore.

The condition $\mathtt{B} \leq 1$ has to be true because there is just one boat.

(f) $$\begin{aligned}\mathtt{Transition} := \quad & \mathtt{B}' = 1 - \mathtt{B} \\ \wedge \quad & \big(\mathtt{B} = 1 \ \rightarrow \ 1 \leq \mathtt{M} - \mathtt{M}' + \mathtt{C} - \mathtt{C}' \leq 2 \ \wedge \ \mathtt{M}' \leq \mathtt{M} \ \wedge \ \mathtt{C}' \leq \mathtt{C}\big) \\ \wedge \quad & \big(\mathtt{B} = 0 \ \rightarrow \ 1 \leq \mathtt{M}' - \mathtt{M} + \mathtt{C}' - \mathtt{C} \leq 2 \ \wedge \ \mathtt{M}' \geq \mathtt{M} \ \wedge \ \mathtt{C}' \geq \mathtt{C}\big)\end{aligned}$$

Let us explain the details of this formula:

- $\mathtt{B}' = 1 - \mathtt{B}$
  If the boat is initially on the western shore, i.e. $\mathtt{B} = 1$, it will be on the eastern shore afterwards, i.e. we will then have $\mathtt{B}' = 0$. If, instead, the boat is initially on the eastern shore, i.e. $\mathtt{B} = 0$, it will be on the western shore afterwards and then we have $\mathtt{B}' = 1$.
- $\mathtt{B} = 1 \ \rightarrow \ 1 \leq \mathtt{M} - \mathtt{M}' + \mathtt{C} - \mathtt{C}' \leq 2 \ \wedge \ \mathtt{M}' \leq \mathtt{M} \ \wedge \ \mathtt{C}' \leq \mathtt{C}$
  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 $\mathtt{M} - \mathtt{M}'$ is the number of missionaries on the boat, while $\mathtt{C} - \mathtt{C}'$ 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.
  The condition $\mathtt{M}' \leq \mathtt{M}$ is true because when missionaries are traveling from the western shore to the eastern shore, the number of missionaries on the western shore can not increase. Similarly, $\mathtt{C}' \leq \mathtt{C}$ has to be true.

- $\mathtt{B} = 0 \;\rightarrow\; 1 \leq \mathtt{M}' - \mathtt{M} + \mathtt{C}' - \mathtt{C} \leq 2 \;\wedge\; \mathtt{M}' \geq \mathtt{M} \;\wedge\; \mathtt{C}' \geq \mathtt{C}$
  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 5.21: Coding the *missionaries and cannibals problem* as a symbolic transition system.

Figure 5.21 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 $\mathcal{V}$, while we append β to these names to get the primed versions of the corresponding variable.

Figure 5.22 shows how we can turn the symbolic transition system into a CSP.

1. The function `flatten(LoL)` receives a list of lists `LoL` as its argument. This list has the form

   $$\mathtt{LoL} = [L_1, \cdots, L_k]$$

   where the L_i are lists for $i = 1, \cdots, k$.

   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

   $\mathtt{M}_i$, $\mathtt{C}_i$, and $\mathtt{B}_i$, where $i = 0, \cdots, n$.

```
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 5.22: Turning the symbolic transition system into a CSP.

$\texttt{M}_i$ 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 of crossings has to be odd, we increment n by two at the end of the `while` loop.

   The function `find_solution` needs less than 2 seconds to find the solution.

Exercise 19: An agricultural economist has to sell a wolf, a goat, and a cabbage on a market place. In order to reach the market place, she has to cross a river. The boat that she can use is so small that it can only accommodate either the goat, the wolf, or the cabbage in addition to the agricultural economist herself. Now if the agricultural economist leaves the wolf alone with the goat, the wolf will eat the goat. If, instead, the agricultural economist leaves the goat with the cabbage, the goat will eat the cabbage. Is it possible for the agricultural economist to develop a schedule that allows her to cross the river without either the goat or the cabbage being eaten?

Encode this problem as a symbolic transition system and then solve it with the help of the constraints solver developed earlier. Assume that the problem can be solved with $n \in \mathbb{N}$ crossing of the river. Use the following variables:

- `F`i for $i \in \{0, \cdots, n\}$ is the number of farmers on the western shore after the i^{th} crossing.
- `W`i for $i \in \{0, \cdots, n\}$ is the number of wolves on the western shore after the i^{th} crossing.
- `G`i for $i \in \{0, \cdots, n\}$ is the number of goats on the western shore after the i^{th} crossing.
- `C`i for $i \in \{0, \cdots, n\}$ is the number of cabbages on the western shore after the i^{th} crossing.

You should start from the following notebook:

`github.com/karlstroetmann/Logic/blob/master/Python/Chapter-5/Wolf-Goat-Cabbage-STS.ipynb`

◇

5.6 The Z3 Solver

We proceed with a discussion of the solver Z3, which has beed developd at Microsoft. 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 8^{th} 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?*

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 have not been mentioned.

We can now infer the number of children. If we denote the number of boys by the variable b and the number of girls by the variable g, the problem statements are equivalent to the following equations:

- $b - 1 = g$.
- $2 \cdot (g - 1) = b$.

Before we can start to solve this problem, we have to install Z3 via `pip` using the following command:

```
pip install z3-solver
```

```
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 5.23: Solving a simple text problem.

Figure 5.23 on page 138 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`.

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 20: 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. You should start from the following notebook:

`github.com/karlstroetmann/Logic/blob/master/Python/Chapter-5/Parrot-and-Penguin.ipynb`

◇

Exercise 21: 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.

You should start from the following notebook:

github.com/karlstroetmann/Logic/blob/master/Python/Chapter-5/Train-Z3.ipynb ◇

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

$$\mathtt{R}_i \text{ and } \mathtt{C}_i \quad \text{for } i \in \{0, 1, \cdots, 63\}.$$

Here $\mathtt{R}_i$ denotes the row of the knight after its i^{th} move, while $\mathtt{C}_i$ 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 $\langle \mathtt{R}_i, \mathtt{C}_i \rangle$ to the position $\langle \mathtt{R}_{i+1}, \mathtt{C}_{i+1} \rangle$ 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 5.24 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 $i \neq j$.

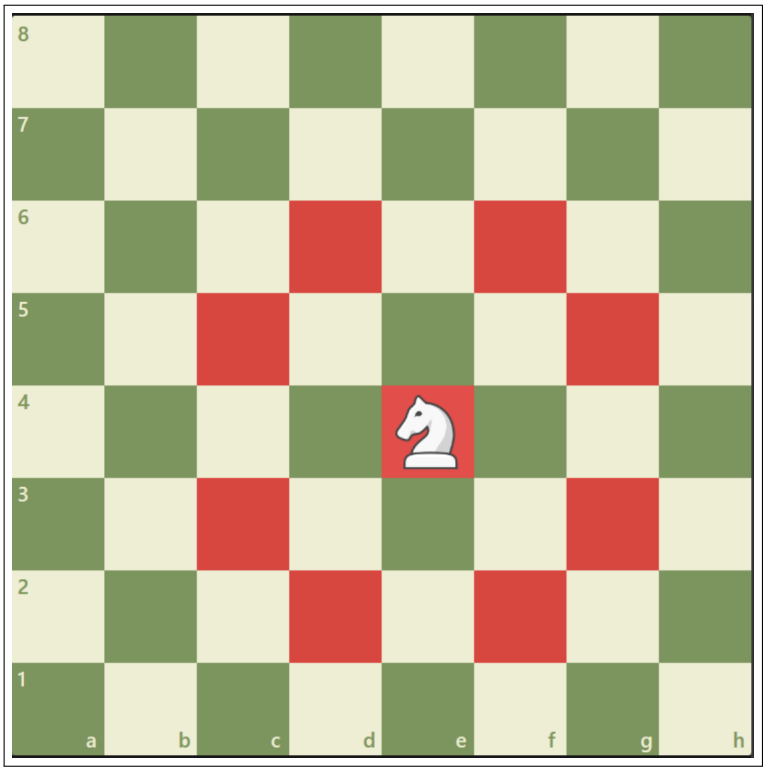

Figure 5.24: The moves of a knight, courtesy of chess.com.

Figure 5.25 shows how we can formulate the puzzle using `Z3`.

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 $\texttt{R}_i$ and $\texttt{C}_i$.
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

```python
import z3

def row(i): return f'R{i}'
def col(i): return f'C{i}'

def is_knight_move(row, col, rowX, colX):
    Formulas = set()
    S = {1, 2, -1, -2}
    DeltaSet = {(x, y) for x in S for y in S if abs(x) != abs(y)}
    for delta_r, delta_c in DeltaSet:
        Formulas.add(z3.And(rowX == row + delta_r, colX == col + delta_c))
    return z3.Or(Formulas)

def all_different(Rows, Cols):
    Result = set()
    for i in range(62+1):
        for j in range (i+1, 63+1):
            Result.add(z3.Or(Rows[i] != Rows[j], Cols[i] != Cols[j]))
    return Result

def all_constraints(Rows, Cols):
    Constraints = all_different(Rows, Cols)
    Constraints.add(Rows[0] == 0)
    Constraints.add(Cols[0] == 0)
    for i in range(62+1):
        Constraints.add(is_knight_move(Rows[i], Cols[i], Rows[i+1], Cols[i+1]))
    for i in range(63+1):
        Constraints.add(Rows[i] >= 0)
        Constraints.add(Cols[i] >= 0)
    return Constraints
```

Figure 5.25: The Knight's Tour: Computing the constraints.

$$f_1 \vee \cdots \vee f_n.$$

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 $\langle \mathtt{R}_i, \mathtt{C}_i \rangle$ for $i = 0, 1 \cdots, 63$ are all different from each other. Note that the position $\langle \mathtt{R}_i, \mathtt{C}_i \rangle$ is different from the position $\langle \mathtt{R}_j, \mathtt{C}_j \rangle$ 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.

```
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 5.26: The function `solve`.

Finally, the function `solve` that is shown in Figure 5.26 on page 143 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.

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 5.27: 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 corresponding 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 5.27 on page 144 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 22: Table 5.1 on page 144 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. ◇

5.6.3 Solving Search Problems with Z3

In this subsection we show how to solve a search problem using Z3. The idea is to formulate the search problem as a symbolic transition system, convert the symbolic transition system into a CSP, and then solve the CSP using Z3. We will explain our method by solving the missionaries and cannibals problem that has already been discussed earlier.

```
1  M = z3.Int('M')
2  C = z3.Int('C')
3  B = z3.Int('B')
4
5  def start(M, C, B):
6      return z3.And(M == 3, C == 3, B == 1)
7
8  def goal(M, C, B):
9      return z3.And(M == 0, C == 0, B == 0)
10
11 def invariant(M, C, B):
12     return z3.And(z3.Or(M == 0, M == 3, M == C),
13                   0 <= M, M <= 3,
14                   0 <= C, C <= 3,
15                   0 <= B, B <= 1)
16 def transition(M, C, B, Mx, Cx, Bx):
17     Formulas  = { Bx == 1 - B }
18     Formulas |= { z3.Implies(B == 1,
19                              z3.And(1 <= M - Mx + C - Cx,
20                                     2 >= M - Mx + C - Cx,
21                                     Mx <= M,
22                                     Cx <= C)
23                             ),
24                   z3.Implies(B == 0,
25                              z3.And(1 <= Mx - M + Cx - C,
26                                     2 >= Mx - M + Cx - C,
27                                     Mx >= M,
28                                     Cx >= C)
29                             )
30                 }
31     return z3.And(Formulas)
```

Figure 5.28: Solving The Missionaries-and-Cannibals-Problem with Z3, part I.

Figure 5.28 on page 145 shows how we can define the associated symbolic transition system using the constraint solver Z3.

1. In line 1 – 3 we define the variables M, C, and B. These represent the number of missionaries,

cannibals, and boats on the eastern shore. We have defined these variables as integers, although the set of values really is the set $\{0, 1, 2, 3\}$. We will later add appropriate inequalities to restrict these variables to this set.

2. The definition of `start` and `goal` in line 6 and 9 are analogous the corresponding definitions in Abbildung 5.21 auf Seite 135. The describe that initially everybody and the boat is on the eastern shore, while in the end everybody is on the western shore.

   Note that we have to use the `Z3` function `And` to form the conjunction of the conditions. The function takes an arbitrary number of arguments. The constraint solver `Z3` also provides the functions `Or` and `Implies`. The function `Or` computes the disjunction of its arguments, while the function `Implies`(A, B) represents the formula $A \rightarrow B$.

3. The definition of the invariant in 11 – 15 has become more complicated because in addition to the formula

   $$M = 0 \vee M = 3 \vee M = C$$

   we also have to add constraints that restrict the values of M and C to the set $\{0, 1, 2, 3\}$, while B has to be an element of the set $\{0, 1\}$.

4. The definition of the formula describing the transitions of the symbolic transition system is analogous to the corresponding definition in Abbildung 5.21 auf Seite 135.

Figure 5.29 on page 147 show the implementation of the function `missionaries_CSP` that takes a natural number n as its input and then creates a CSP that is solvable if and only if the symbolic transition system shown in Abbildung 5.28 auf Seite 145 has a solution involving n transitions.

1. Line 2 creates the solver object `S`.
2. Line 3–5 create the associated variables.
3. Line 6 creates the constraint corresponding to the initial state of the symbolic transition system.
4. Line 7 creates the constraint corresponding to the goal state.
5. The `for`-loop in line 8 ensures two things:
   (a) Every intermediate state has to satisfy the invariant.
   (b) Every transition from a state to the successor state has to satisfy the transition formula.
6. Line 12 feeds the constraints to the solver.
7. Line 13 checks whether the resulting CSP is solvable.
8. If it is solvable, we return a dictionary that associates every variable $\texttt{M}_i$, $\texttt{C}_i$, $\texttt{B}_i$ with the corresponding value of the solution.
9. The function `find_solution` searches for a value of $n \in \mathbb{N}$ such that the problem is solvable in n steps.

```
def missionaries_CSP(n):
    S = z3.Solver()
    Ms = [z3.Int(f'M{i}') for i in range(n+1)]
    Is = [z3.Int(f'C{i}') for i in range(n+1)]
    Bs = [z3.Int(f'B{i}') for i in range(n+1)]
    Constraints  = { start(Ms[0], Is[0], Bs[0]) }  # start state
    Constraints |= { goal( Ms[n], Is[n], Bs[n]) }  # goal state
    for i in range(n):
        Constraints.add(invariant(Ms[i], Is[i], Bs[i]))
        Constraints.add(transition(Ms[i  ], Is[i  ], Bs[i  ],
                                   Ms[i+1], Is[i+1], Bs[i+1]))
    S.add(Constraints)
    result = str(S.check())
    if result == 'sat':
        Model = S.model()
        Solution = (   { f'M{i}': Model[Ms[i]] for i in range(n+1) }
                     | { f'C{i}': Model[Is[i]] for i in range(n+1) }
                     | { f'B{i}': Model[Bs[i]] for i in range(n+1) }
                   )
        return { key: Solution[key].as_long() for key in Solution }
    else:
        return None

def find_solution():
    n = 1
    while True:
        print(n)
        Solution = missionaries_CSP(n)
        if Solution != None:
            return n, Solution
        n += 2
```

Figure 5.29: Solving The Missionaries-and-Cannibals-Problem with Z3, part II.

At this point you might well ask whether there is any benefit if we solve this problem with Z3. The answer is that it is about efficiency. While our simple backtracking solver took 1.5 seconds to solve this problem, Z3 takes only 0.113 seconds to compute the solution. In the following exercise you will solve a more complicated search problem which would take much too long to solve if we would only use backtracking.

Exercise 23: The following puzzle is part of a recruitment test in Japan.

A policeman, a convict, a father and his two sons Anton and Bruno, and a mother with her two daughters Cindy and Doris have to cross a river. On the boat there is only room for two passengers. During the crossing, the following conditions have to be observed:

(a) *The father is not allowed to be on a shore with one of the daughters if the mother is on the other shore.*

(b) *The mother is not allowed to be on a shore with one of the sons if the father is on the other shore.*

(c) *If the convict is not alone, then the policeman must watch him.*

(d) *However the convict can be alone on a shore, as his shackles prevent him from running away.*

(e) *Only the father, the mother, and the policeman are able to steer the boat.*

Your task is to solve this puzzle by coding it as a symbolic transition system. You should then solve the transition system using the constraint solver Z3. Start with the following file:

https://github.com/karlstroetmann/Logic/blob/master/Python/Chapter-5/Japanese-Z3.ipynb. ◇

Exercise 24: The following puzzle offers a new twist to the river crossing puzzles encountered before, because we have to take the time used for a crossing into account.

> *In the darkness of the night, a group of four individuals encounters a river. A slender bridge stretches before them, capable of accommodating just two people simultaneously. Equipped with a single torch, they must rely on its flickering light to navigate the bridge. Each person possesses a distinct crossing time: Ariela takes 1 minute, Brian takes 2 minutes, Charly takes 5 minutes, and Dumpy takes 8 minutes. It is crucial to note that when two people cross together, they must synchronize their steps with the slower individual's pace. Given the torch's limited lifespan of 15 minutes, the pressing question arises: can all four individuals successfully traverse the bridge?*

Your task is to solve this puzzle by coding it as a symbolic transition system. You should then solve the transition system using the constraint solver Z3. Start with the following file:

https://github.com/karlstroetmann/Logic/blob/master/Python/Chapter-5/Bridge-Torch.ipynb. ◇

5.7 Normalizing First-Order Formulas

In the next section, we will define a calculus $\vdash$ for first-order logic. Just as in the case of propositional logic, this will be much easier if we restrict ourselves to formulas that are in a normal form. This normal form consists of so-called first-order clauses. These are defined similarly as in propositional logic: A first-order literal is an atomic formula or the negation of an atomic formula. A first-order clause is a disjunction of first-order literals. We show in this section that every set of formulas M can be transformed into a set of first-order clauses K such that M is satisfiable if and only if K is satisfiable. Therefore, the restriction to first-order clauses is not a real restriction. First, we specify some equivalences that can be used to manipulate quantifiers.

Satz 42 *The following equivalences are valid in first-order logic:*

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. *If x is a variable such that $x \notin FV(f)$, then*
   $$\models (\forall x\colon f) \leftrightarrow f \quad \text{and} \quad \models (\exists x\colon f) \leftrightarrow f.$$
8. *If x is a variable such that $x \notin FV(g)$, then:*
   (a) $\models (\forall x\colon f) \vee g \leftrightarrow \forall x\colon (f \vee g)$ and $\models g \vee (\forall x\colon f) \leftrightarrow \forall x\colon (g \vee f)$,
   (b) $\models (\exists x\colon f) \wedge g \leftrightarrow \exists x\colon (f \wedge g)$ and $\models g \wedge (\exists x\colon f) \leftrightarrow \exists x\colon (g \wedge f)$.

In order to apply the equivalences of the last group, it may be necessary to rename bound variables. If f is a first-order logic formula and x and y are two variables, where y does not appear in f, then $f[x/y]$ denotes the formula that results from f by replacing every occurrence of the variable x in f with y. For example,

$$\big(\forall u : \exists v : p(u,v)\big)[u/z] = \forall z : \exists v : p(z,v)$$

With this we can specify one last equivalence: If f is a first-order logic formula, $x \in BV(f)$ and y is a variable that does not appear in f, then

$$\models f \leftrightarrow f[x/y].$$

Using the equivalences given above together with the equivalences from propositional logic we are able to transform every first order formula into a form where the quantifieres are all at the beginning of the formula, i.e. the formula has the form

$$Q_1 x_1 : Q_2 x_2 : \cdots Q_n x_n : G \quad \text{where } Q_i \in \{\forall, \exists\} \text{ f.a. } i = 1, \cdots, n$$

and G does not contain quantifieres. A formula of this form is in prenex normal form. We demonstrate this procedure with an example and show that the formula

$$\big(\forall x\colon p(x)\big) \to \big(\exists x\colon p(x)\big)$$

is universally valid:

$$\begin{aligned}
& \big(\forall x\colon p(x)\big) \to \big(\exists x\colon p(x)\big) \\
\Leftrightarrow\ & \neg\big(\forall x\colon p(x)\big) \vee \big(\exists x\colon p(x)\big) \\
\Leftrightarrow\ & \big(\exists x\colon \neg p(x)\big) \vee \big(\exists x\colon p(x)\big) \\
\Leftrightarrow\ & \exists x\colon \big(\neg p(x) \vee p(x)\big) \\
\Leftrightarrow\ & \exists x\colon \top \\
\Leftrightarrow\ & \top
\end{aligned}$$

In this case, we were lucky that we managed to recognise the formula as a tautology. In general, however, the above transformations are not sufficient to recognise whether a first order formula is universally valid. To be able to simplify formulas even more, we introduce another concept of equivalence. We want to motivate this term with an example. We look at the following formulas

$$f_1 = \forall x\colon \exists y\colon p(x,y) \quad \text{und} \quad f_2 = \forall x\colon p\big(x, s(x)\big).$$

The two formulae f_1 and f_2 are not equivalent, because they do not even originate from the same signature. In the formula f_2, the function character s is used, which does not appear at all in the formula f_1. Even if the two formulae f_1 and f_2 are not equivalent, the following relationship exists between them: If S_1 is a Σ-structure in which the formula f_1 holds, i.e.

$$\mathcal{S}_1 \models f_1,$$

then we can extend the Σ-structure to a Σ'-structure S_2 such that f_2 is valid in $\mathcal{S}_2$:

$$\mathcal{S}_2 \models f_2.$$

To do this, the interpretation of the function character s has to be chosen such that $p\big(x, s(x)\big)$ actually holds for each x. This is possible because the formula f_1 states that for every x there is a value y such that $p(x,y)$ holds. The function s therefore has to return this y for every x.

Definition 43 (Skolemisation)

Assume $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle$ *is a signature. Furthermore, let* f *be a closed* Σ *formula of the form*

$$f = \forall x_1, \cdots, x_n \colon \exists y \colon g.$$

Then we choose an new n*-digit function character* s*, i.e. we take a character* s *that is not used in the signature* Σ *and extend the signature* Σ *to the signature*

$$\Sigma' := \Big\langle \mathcal{V}, \mathcal{F} \cup \{s\}, \mathcal{P}, arity \cup \big\{\langle s, n\rangle\big\} \Big\rangle,$$

where s *is declared as a new* n*-digit function character. Then we define the* Σ' *formula* f' *as follows:*

$$f' := \texttt{Skolem}(f) := \forall x_1 \colon \cdots \forall x_n \colon g\big[y \mapsto s(x_1, \cdots, x_n)\big]$$

Here, the expression $g\big[y \mapsto s(x_1, \cdots, x_n)\big]$ *denotes the formula that we obtain from* g *by replacing each occurrence of the variable* y *in the formula* g *by the term* $s(x_1, \cdots, x_n)$*. We say that the formula* f' *results from the formula* f *through one Skolemisation step.* ◇

Example: It f the following formula from group theory:

$$f := \forall x : \exists y : y * x = 1.$$

Then the following holds:

$$\texttt{Skolem}(f) = \forall x : s(x) * x = 1.$$ ◇

In what sense are a formula f and a formula f', which are derived from f by a Skolemisation step equivalent? We answer this question with the following definition.

Definition 44 (Satisfiability Equivalence)
Two closed formulas f *and* g *are called satisfiability-equivalent if* f *and* g *are either both satisfiable or both unsatisfiable. If* f *and* g *are satisfiability-equivalent, we write*

$$f \approx_e g.$$ ◇

Observation: If the formula f' is derived from the formula f by a skolemisation step, i.e. if we have that $f' = \texttt{Skolem}(f)$ then f and f' are satisfiability-equivalent, because we can transform any structure $\mathcal{S}$ such that $\mathcal{S} \models f$ into a structure $\mathcal{S}'$ such that $\mathcal{S}' \models f'$. Therefore we have

$$f \approx_e \texttt{Skolem}(f).$$ ◇

We can now specify a simple procedure to eliminate existential quantifiers from a formula. This procedure consists of two steps:

- First, we put the formula into prenex normal form.
- Then we can eliminate the existential quantifiers one by one using Skolemisation steps.

The resulting formula is then satisfiability-equivalent to the original formula. This process of eliminating existential quantifiers through the introduction of new function function characters is called Skolemisation. If we have a formula F into prenex normal form and then skolemised it, the result has the form

$$\forall x_1, \cdots, x_n : g$$

and there are no more quantifiers in the formula g. The formula g is also known as the matrix of the above formula. We can now transform g with the help of of the propositional equivalences known

from the last chapter into conjunctive normal form. We then have a formula of the form

$$\forall x_1, \cdots, x_n : (k_1 \wedge \cdots \wedge k_m).$$

The k_i are disjunctions of first-order literals. Next, we apply the equivalence

$$\forall x \colon (f_1 \wedge f_2) \leftrightarrow (\forall x \colon f_1) \wedge (\forall x \colon f_2).$$

In this way we can distribute the universal-quantifiers to the individual k_i. The resulting formula has the form

$$\big(\forall x_1, \cdots, x_n : k_1\big) \wedge \cdots \wedge \big(\forall x_1, \cdots, x_n : k_m\big).$$

If a formula F has this form, we say that F is in first-order clause normal form and a formula of the form

$$\forall x_1, \cdots, x_n : k,$$

where k is a disjunction of first-order literals, is called a first-order clause. If M is is a set of formulas whose satisfiability we want to analyse, we can always transform M into a satisfiability-equivalent set of first-order clauses. Since then only universal-quantifiers occur, we can simplify the notation by agreeing that all formulae are implicitly universally quantified, i.e. we omit the universal quantifiers. The Jupyter notebook

https://github.com/karlstroetmann/Logic/blob/master/Python/Chapter-5/FOL-CNF.ipynb.

contains a *Python* programme that we can use to transform predicate logic formulae into a satisfiability-equivalent set of predicate logic clauses.

Our goal is to develop a method that can be used to verify that a first-order formula f is universally valid, i.e. we want to be able to prove that

$$\models f$$

holds. We know that

$$\models f \quad \text{if and only if} \quad \{\neg f\} \models \bot,$$

because the formula f is universally valid iff there is no Σ-structure $\mathcal{S}$ such that

$$\mathcal{S} \models \neg f,$$

i.e. f is universally valid iff $\neg f$ is unsatisfiable. Therefore, in oder to prove that f is universally valid, we transform the formula $\neg f$ into first-order normal form. By doing this we obtain clauses $k_1, \cdots, k_n$ such that

$$\neg f \approx_e k_1 \wedge \cdots \wedge k_n$$

holds. Next, we try to derive a contradiction from the clauses $k_1, \cdots, k_n$:

$$\{k_1, \cdots, k_n\} \vdash \bot$$

If this succeeds, then we know that the set $\{k_1, \cdots, k_n\}$ is unsatisfiable. This means that $\neg f$ is also unsatisfiable and hence f must be universally valid. In order to derive a contradiction from the clauses $k_1, \cdots, k_n$ we need a calculus $\vdash$ that works with first-order clauses. We will present this calculus in section 5.9.

To explain the procedure in more detail, we will demonstrate it using an example. We want to investigate whether

$$\models \big(\exists x\colon \forall y\colon p(x,y)\big) \to \big(\forall y\colon \exists x\colon p(x,y)\big)$$

holds. We know that this is equivalent to

$$\Big\{\neg\Big(\big(\exists x\colon \forall y\colon p(x,y)\big) \to \big(\forall y\colon \exists x\colon p(x,y)\big)\Big)\Big\} \models \bot.$$

Therefore we transform the negated formula in prenex normal form.

$$\begin{aligned}
& \neg\Big(\big(\exists x\colon \forall y\colon p(x,y)\big) \to \big(\forall y\colon \exists x\colon p(x,y)\big)\Big) \\
\leftrightarrow\ & \neg\Big(\neg\big(\exists x\colon \forall y\colon p(x,y)\big) \vee \big(\forall y\colon \exists x\colon p(x,y)\big)\Big) \\
\leftrightarrow\ & \big(\exists x\colon \forall y\colon p(x,y)\big) \wedge \neg\big(\forall y\colon \exists x\colon p(x,y)\big) \\
\leftrightarrow\ & \big(\exists x\colon \forall y\colon p(x,y)\big) \wedge \big(\exists y\colon \neg\exists x\colon p(x,y)\big) \\
\leftrightarrow\ & \big(\exists x\colon \forall y\colon p(x,y)\big) \wedge \big(\exists y\colon \forall x\colon \neg p(x,y)\big)
\end{aligned}$$

In order to proceed we have to rename the variables in the second part of the formula. We replace x with u and y with v and are left with the following:

$$\begin{aligned}
& \big(\exists x\colon \forall y\colon p(x,y)\big) \wedge \big(\exists y\colon \forall x\colon \neg p(x,y)\big) \\
\leftrightarrow\ & \big(\exists x\colon \forall y\colon p(x,y)\big) \wedge \big(\exists v\colon \forall u\colon \neg p(u,v)\big) \\
\leftrightarrow\ & \exists v\colon \Big(\big(\exists x\colon \forall y\colon p(x,y)\big) \wedge \big(\forall u\colon \neg p(u,v)\big)\Big) \\
\leftrightarrow\ & \exists v\colon \exists x\colon \Big(\big(\forall y\colon p(x,y)\big) \wedge \big(\forall u\colon \neg p(u,v)\big)\Big) \\
\leftrightarrow\ & \exists v\colon \exists x\colon \forall y\colon \Big(p(x,y) \wedge \big(\forall u\colon \neg p(u,v)\big)\Big) \\
\leftrightarrow\ & \exists v\colon \exists x\colon \forall y\colon \forall u\colon \Big(p(x,y) \wedge \neg p(u,v)\Big)
\end{aligned}$$

Now we have to skolemize in order to get rid of the existential quantifiers. In order to do this, we introduce two new function symbols s_1 and s_2. We have both $\texttt{arity}(s_1) = 0$ and $\texttt{arity}(s_2) = 0$, because the existential quantifiers are not preceded by universal quantifiers.

$$\begin{aligned}
& \exists v\colon \exists x\colon \forall y\colon \forall u\colon \Big(p(x,y) \wedge \neg p(u,v)\Big) \\
\approx_e\ & \exists x\colon \forall y\colon \forall u\colon \Big(p(x,y) \wedge \neg p(u,s_1)\Big) \\
\approx_e\ & \forall y\colon \forall u\colon \Big(p(s_2,y) \wedge \neg p(u,s_1)\Big)
\end{aligned}$$

At this point we are left with universal quantifiers. These can be omitted, as we have already agreed that all free variables are implicitly universally quantified. We proceed to present the first order CNF of our formula in set notation:

$$M := \Big\{\big\{p(s_2,y)\big\}, \big\{\neg p(u,s_1)\big\}\Big\}.$$

Next, we show that the set M is contradictory. To do this, we first consider the clause $\big\{p(s_2,y)\big\}$ and insert the constant s_1 in this clause for y. This gives us the clause

$$\big\{p(s_2,s_1)\big\}. \tag{1}$$

We justify the replacement of y by s_1 by the fact that the above clause is implicitly universally quantified and if something is true for all y, then it is certainly also true for $y = s_1$.

Next, we take the clause $\big\{\neg p(u,s_1)\big\}$ and insert the constant s_2 for the variable u. We obtain the clause

$$\{\neg p(s_2, s_1)\} \tag{2}$$

Now we apply the cut rule to the clauses (1) and (2) and have

$$\{p(s_2, s_1)\}, \quad \{\neg p(s_2, s_1)\} \quad \vdash \quad \{\}.$$

We have thus derived a contradiction and shown that the set M is unsatisfiable. Therefore, we also know that

$$\Big\{\neg\Big(\big(\exists x\colon \forall y\colon p(x,y)\big) \to \big(\forall y\colon \exists x\colon p(x,y)\big)\Big)\Big\}$$

is unsatisfiable and hence we have shown

$$\models \big(\exists x\colon \forall y\colon p(x,y)\big) \to \big(\forall y\colon \exists x\colon p(x,y)\big).$$

5.8 Unification

This section introduces the notion of a most general unifier of two terms. First, we define the notion of a Σ-substitution.

Definition 45 (Σ-Substitution) *Assume that a signature $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle$ is given. A Σ-substitution σ is a map of the form*

$$\sigma : \mathcal{V} \to \mathcal{T}_\Sigma$$

such that the set $dom(\sigma) := \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ is finite. If we have $dom(\sigma) = \{x_1, \cdots, x_n\}$ and $t_i = \sigma(x_i)$ for all $i = 1, \cdots, n$, then we use the following notation:

$$\sigma = \{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\}.$$

The set of all Σ-Substitutions is denoted as $\texttt{Subst}(\Sigma)$. ◇

A substitution $\sigma = \{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\}$ can be applied to a term t by replacing the variables x_i with the terms t_i. We will use the postfix notation $t\sigma$ to denote the application of the substitution σ to the term t. Formally, the notation $t\sigma$ is defined by induction on t:

1. $x\sigma := \sigma(x)$ for all $x \in \mathcal{V}$.
2. $c\sigma = c$ for every constant $c \in \mathcal{F}$.
3. $f(t_1, \cdots, t_n)\sigma := f\big(t_1\sigma, \cdots, t_n\sigma\big)$.

Next, we define the composition of two Σ-substitutions.

Definition 46 (Composition of Substitutions) *Assume that*

$$\sigma = \{x_1 \mapsto s_1, \cdots, x_m \mapsto s_m\} \quad \text{and} \quad \tau = \{y_1 \mapsto t_1, \cdots, y_n \mapsto t_n\}$$

are two substitutions such that $dom(\sigma) \cap dom(\tau) = \{\}$. We define the composition $\sigma\tau$ of σ and τ as

$$\sigma\tau := \{x_1 \mapsto s_1\tau, \cdots, x_m \mapsto s_m\tau,\ y_1 \mapsto t_1, \cdots, y_n \mapsto t_n\}$$ ◇

Example: If we define

$$\sigma := \{x_1 \mapsto c,\ x_2 \mapsto f(x_3)\} \quad \text{and} \quad \tau := \{x_3 \mapsto h(c,c),\ x_4 \mapsto d\},$$

then we have

$$\sigma\tau = \{x_1 \mapsto c,\ x_2 \mapsto f(h(c,c)),\ x_3 \mapsto h(c,c),\ x_4 \mapsto d\}. \qquad \diamond$$

Proposition 47 *If t is a term and σ and τ are substitutions such that $dom(\sigma) \cap dom(\tau) = \{\}$ holds, then we have*

$$(t\sigma)\tau = t(\sigma\tau). \qquad \diamond$$

This proposition may be proven by induction on t.

Definition 48 (Syntactical Equation) *A syntactical equation is a pair $\langle s, t\rangle$ of terms. It is written as $s \doteq t$. A system of syntactical equations is a set of syntactical equations.* $\diamond$

Definition 49 (Unifier) *A substitution σ solves a syntactical equation $s \doteq t$ iff we have $s\sigma = t\sigma$. If E is a system of syntactical equations and σ is a substitution that solves every syntactical equations in E, then σ is a unifier of E.* $\diamond$

If $E = \{s_1 \doteq t_1, \cdots, s_n \doteq t_n\}$ is a system of syntactical equations and σ is a substitution, then we define

$$E\sigma := \{s_1\sigma \doteq t_1\sigma, \cdots, s_n\sigma \doteq t_n\sigma\}.$$

Example: Let us consider the syntactical equation

$$p(x_1, f(x_4)) \doteq p(x_2, x_3)$$

and define the substitution

$$\sigma := \{x_1 \mapsto x_2,\ x_3 \mapsto f(x_4)\}.$$

Then σ solves the given syntactical equation because we have

$$p(x_1, f(x_4))\sigma = p(x_2, f(x_4)) \quad \text{und}$$
$$p(x_2, x_3)\sigma \quad = p(x_2, f(x_4)). \qquad \diamond$$

Next we develop an algorithm for solving a system of syntactical equations. The algorithm we present was published by Martelli and Montanari [MM82]. To begin, we first consider the cases where a syntactical equation $s \doteq t$ is unsolvable. There are two cases: A syntactical equation of the form

$$f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n)$$

is certainly unsolvable if f and g are different function symbols. The reason is that for any substitution σ we have that

$$f(s_1, \cdots, s_m)\sigma = f(s_1\sigma, \cdots, s_m\sigma) \quad \text{und} \quad g(t_1, \cdots, t_n)\sigma = g(t_1\sigma, \cdots, t_n\sigma).$$

If $f \neq g$, then the terms $f(s_1, \cdots, s_m)\sigma$ and $g(t_1, \cdots, t_n)\sigma$ start with different function symbols and hence they can't be identical.

The other case where a syntactical equation is unsolvable, is a syntactical equation of the following form:

$$x \doteq f(t_1, \cdots, t_n) \quad \text{where } x \in \mathtt{var}\big(f(t_1, \cdots, t_n)\big).$$

This syntactical equation is unsolvable because the term $f(t_1, \cdots, t_n)\sigma$ will always contain at least one more occurrence of the function symbol f than the term $x\sigma$.

Now we are able to present an algorithm for solving a system of syntactical equations, provided the system is solvable. The algorithm will also discover if a system of syntactical equations is unsolvable. The algorithm works on pairs of the form $\langle F, \tau \rangle$ where F is a system of syntactical equations and τ is a substitution. The algorithm starts with the pair $\langle E, \{\} \rangle$. Here E is the system of syntactical equations that is to be solved and $\{\}$ represents the empty substitution. The system works by simplifying the pairs $\langle F, \tau \rangle$ using certain reduction rules that are presented below. These reduction rules are applied until we either discover that the system of syntactical equations is unsolvable or else we reduce the pairs until we finally arrive at a pair of the form $\langle \{\}, \mu \rangle$. In this case μ is a unifier of the system of syntactical equations E. The reduction rules are as follows:

1. If $y \in \mathcal{V}$ is a variable that does not occur in the term t, then we can perform the following reduction:

   $$\Big\langle E \cup \{y \doteq t\}, \sigma \Big\rangle \quad \rightsquigarrow \quad \Big\langle E\{y \mapsto t\}, \sigma\{y \mapsto t\} \Big\rangle \qquad \text{if } y \in \mathcal{V} \text{ and } y \notin \mathtt{var}(t)$$

   This reduction rule can be understood as follows: If the system of syntactical equations that is to be solved contains a syntactical equation of the form $y \doteq t$, where the variable y does not occur in the term t, then the syntactical equation $y \doteq t$ can be removed if we apply the substitution $\{y \mapsto t\}$ to both components of the pair

   $$\Big\langle E \cup \{y \doteq t\}, \sigma \Big\rangle.$$

2. If the variable y occurs in the term t, i.e. if $y \in \mathit{Var}(t)$ and, furthermore, $t \neq y$, then the system of syntactical equations $E \cup \{y \doteq t\}$ has no solution. We write this as

   $$\Big\langle E \cup \{y \doteq t\}, \sigma \Big\rangle \rightsquigarrow \Omega \quad \text{if } y \in \mathtt{var}(t) \text{ and } y \neq t.$$

3. If $y \in \mathcal{V}$ and $t \notin \mathcal{V}$, then we have:

   $$\Big\langle E \cup \{t \doteq y\}, \sigma \Big\rangle \quad \rightsquigarrow \quad \Big\langle E \cup \{y \doteq t\}, \sigma \Big\rangle \qquad \text{if } y \in \mathcal{V} \text{ and } t \notin \mathcal{V}.$$

   After we apply this rule, we can apply either the first or the second reduction rule thereafter.

4. Trivial syntactical equations can be deleted:

   $$\Big\langle E \cup \{x \doteq x\}, \sigma \Big\rangle \quad \rightsquigarrow \quad \Big\langle E, \sigma \Big\rangle \qquad \text{if } x \in \mathcal{V}.$$

5. If f is an n-ary function symbol we have

   $$\Big\langle E \cup \{f(s_1, \cdots, s_n) \doteq f(t_1, \cdots, t_n)\}, \sigma \Big\rangle \rightsquigarrow \Big\langle E \cup \{s_1 \doteq t_1, \cdots, s_n \doteq t_n\}, \sigma \Big\rangle.$$

   This rule is the reason that we have to work with a system of syntactical equations, because even if we start with a single syntactical equation the rule given above can increase the number of syntactical equations.

A special case of this rule is the following:

$$\Big\langle E \cup \big\{c \doteq c\big\}, \sigma \Big\rangle \rightsquigarrow \Big\langle E, \sigma \Big\rangle.$$

Here c is a nullary function symbol.

6. The system of syntactical equations $E \cup \big\{f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n)\big\}$ has no solution if the function symbols f and g are different. Hence we have

$$\Big\langle E \cup \big\{f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n)\big\}, \sigma \Big\rangle \rightsquigarrow \Omega \qquad \text{provided } f \neq g.$$

If a system of syntactical equations E is given and we start with the pair $\langle E, \{\}\rangle$, then we can apply the rules given above until one of the following two cases happens:

1. We use the second or the sixth of the reduction rules given above. In this case the system of syntactical equations E is unsolvable.

2. The pair $\langle E, \{\}\rangle$ is reduced into a pair of the form $\langle \{\}, \mu\rangle$. Then μ is a unifier of E. In this case we write $\mu = \texttt{mgu}(E)$. If $E = \{s \doteq t\}$, we write $\mu = \texttt{mgu}(s, t)$. The abbreviation mgu is short for "most general unifier".

Example: We show how to solve the syntactical equation

$$p(x_1, f(x_4)) \doteq p(x_2, x_3).$$

We have the following reductions:

$$\begin{array}{ll} & \big\langle \big\{p(x_1, f(x_4)) \doteq p(x_2, x_3)\big\}, \{\}\big\rangle \\ \rightsquigarrow & \big\langle \big\{x_1 \doteq x_2, f(x_4) \doteq x_3\big\}, \{\}\big\rangle \\ \rightsquigarrow & \big\langle \big\{f(x_4) \doteq x_3\big\}, \{x_1 \mapsto x_2\}\big\rangle \\ \rightsquigarrow & \big\langle \big\{x_3 \doteq f(x_4)\big\}, \{x_1 \mapsto x_2\}\big\rangle \\ \rightsquigarrow & \big\langle \big\{\big\}, \{x_1 \mapsto x_2,\ x_3 \mapsto f(x_4)\}\big\rangle \end{array}$$

Hence the method is successful and we have that the substitution

$$\{x_1 \mapsto x_2,\ x_3 \mapsto f(x_4)\}$$

is a solution of the syntactical equation given above. ◇

Example: Next we try to solve the following system of syntactical equations:

$$E = \big\{p(h(x_1, c)) \doteq p(x_2),\ q(x_2, d) \doteq q(h(d, c), x_4)\big\}$$

We have the following reductions:

$$
\begin{array}{ll}
 & \big\langle \big\{ p(h(x_1, c)) \doteq p(x_2),\ q(x_2, d) \doteq q(h(d, c), x_4) \big\}, \{\} \big\rangle \\
\leadsto & \big\langle \big\{ p(h(x_1, c)) \doteq p(x_2),\ x_2 \doteq h(d, c),\ d \doteq x_4 \big\}, \{\} \big\rangle \\
\leadsto & \big\langle \big\{ p(h(x_1, c)) \doteq p(x_2),\ x_2 \doteq h(d, c),\ x_4 \doteq d \big\}, \{\} \big\rangle \\
\leadsto & \big\langle \big\{ p(h(x_1, c)) \doteq p(x_2),\ x_2 \doteq h(d, c) \big\}, \{x_4 \mapsto d\} \big\rangle \\
\leadsto & \big\langle \big\{ p(h(x_1, c)) \doteq p(h(d, c)) \big\}, \{x_4 \mapsto d,\ x_2 \mapsto h(d, c)\} \big\rangle \\
\leadsto & \big\langle \big\{ h(x_1, c) \doteq h(d, c) \big\}, \{x_4 \mapsto d,\ x_2 \mapsto h(d, c)\} \big\rangle \\
\leadsto & \big\langle \big\{ x_1 \doteq d,\ c \doteq c \big\}, \{x_4 \mapsto d,\ x_2 \mapsto h(d, c)\} \big\rangle \\
\leadsto & \big\langle \big\{ x_1 \doteq d, \big\}, \{x_4 \mapsto d,\ x_2 \mapsto h(d, c)\} \big\rangle \\
\leadsto & \big\langle \{\}, \{x_4 \mapsto d,\ x_2 \mapsto h(d, c),\ x_1 \mapsto d\} \big\rangle
\end{array}
$$

Hence the substitution $\{x_4 \mapsto d,\ x_2 \mapsto h(d, c),\ x_1 \mapsto d\}$ is a solution of the system of syntactical equations given above. ◇

The Jupyter notebook

https://github.com/karlstroetmann/Logic/blob/master/Python/Chapter-5/10-Unification.ipynb.

implements the algorithm that has been described above.

5.9 A Deductive System for First-Order Logic without Equality

In this section, we assume that our signature Σ does not use the equality sign, because this restriction makes it significantly easier to introduce a complete deductive system for first-order logic. Although there is also a complete deductive system for the case where the signature Σ contains the equality sign, it is considerably more complex than the system we are about to introduce and is therefore out of the scope of these lecture notes.

Definition 50 (Resolution) *Assume that*

1. k_1 *and* k_2 *are first-order logic clauses,*
2. $p(s_1, \cdots, s_n)$ *and* $p(t_1, \cdots, t_n)$ *are atomic formulas, and*
3. *the syntactic equation* $p(s_1, \cdots, s_n) \doteq p(t_1, \cdots, t_n)$ *is solvable with most general unifier*

$$\mu = \texttt{mgu}\big(p(s_1, \cdots, s_n), p(t_1, \cdots, t_n)\big).$$

Then the following deduction rule

$$\frac{k_1 \cup \{p(s_1, \cdots, s_n)\} \qquad \{\neg p(t_1, \cdots, t_n)\} \cup k_2}{k_1\mu \cup k_2\mu}$$

is an application of the resolution rule. ◇

The resolution rule is a combination of the substitution rule and the cut rule. The substitution rule has the form

$$\frac{k}{k\sigma}.$$

Here, k is a first-order logic clause and σ is a substitution. In some cases, we may need to rename the variables in one of the two clauses before we can apply the resolution rule. Let us consider an example. The set of clauses

$$M = \Big\{\{p(x)\}, \{\neg p(f(x))\}\Big\}$$

is contradictory. However, we cannot immediately apply the resolution rule because the syntactic equation

$$p(x) \doteq p(f(x))$$

is unsolvable. This is because the same variable happens to be used in both clauses. If we rename the variable x in the second clause to y, we get the set of clauses

$$\Big\{\{p(x)\}, \{\neg p(f(y))\}\Big\}.$$

Here, we can apply the resolution rule because the syntactic equation

$$p(x) \doteq p(f(y))$$

has the solution $[x \mapsto f(y)]$. Then we obtain

$$\{p(x)\}, \quad \{\neg p(f(y))\} \quad \vdash \quad \{\}.$$

and thus we have proven the inconsistency of the clause set M.

The resolution rule by itselft is not sufficient to derive the empty clause from a clause set M that is inconsistent in every case: we need a second rule. To see this, consider following set of clauses:

$$M = \Big\{\{p(f(x), y), p(u, g(v))\}, \{\neg p(f(x), y), \neg p(u, g(v))\}\Big\}$$

We will soon show that the set M is contradictory. It can be shown that the resolution rule is not sufficient to prove that M is contradictory. A simple but tedious proof of this fact can be given by computing all possible resolution steps starting from the set M. In doing so, we would see that the empty clause can never be derived. Because of the incompleteness of the resolution rule by itselft, we now introduce the factorization rule. Using this rule we will be able to show that M is contradictory.

Definition 51 (Factorization) *Assume that the following holds:*

1. *k is a first-order logic clause,*
2. *$p(s_1, \cdots, s_n)$ and $p(t_1, \cdots, t_n)$ are atomic formulas,*
3. *the syntactic equation $p(s_1, \cdots, s_n) \doteq p(t_1, \cdots, t_n)$ is solvable,*
4. *$\mu = \mathtt{mgu}\big(p(s_1, \cdots, s_n), p(t_1, \cdots, t_n)\big)$.*

Then the following rules

$$\frac{k \cup \{p(s_1, \cdots, s_n),\ p(t_1, \cdots, t_n)\}}{k\mu \cup \{p(s_1, \cdots, s_n)\mu\}} \quad \textit{and} \quad \frac{k \cup \{\neg p(s_1, \cdots, s_n),\ \neg p(t_1, \cdots, t_n)\}}{k\mu \cup \{\neg p(s_1, \cdots, s_n)\mu\}}$$

are instances of the factorization rule. ◇

We will demonstrate how the inconsistency of the set M can be proven using the resolution and factorization rules.

1. First, we apply the factorization rule to the first clause. For this, we compute the unifier

   $$\mu = \mathtt{mgu}\big(p(f(x), y), p(u, g(v))\big) = [y \mapsto g(v), u \mapsto f(x)].$$

   Thus, we can apply the factorization rule:

   $$\{p(f(x), y), p(u, g(v))\} \quad \vdash \quad \{p(f(x), g(v))\}.$$

2. Next, we apply the factorization rule to the second clause. For this, we compute the unifier

   $$\mu = \mathtt{mgu}\big(\neg p(f(x), y), \neg p(u, g(v))\big) = [y \mapsto g(v), u \mapsto f(x)].$$

   Thus, we can apply the factorization rule:

   $$\{\neg p(f(x), y), \neg p(u, g(v))\} \quad \vdash \quad \{\neg p(f(x), g(v))\}.$$

3. We conclude the proof with an application of the resolution rule. The unifier used here is the empty substitution, thus $\mu = []$.

   $$\frac{\{p(f(x), g(v))\} \quad \{\neg p(f(x), g(v))\}}{\{\}}$$

If M is a set of first-order logic clauses and k is a first-order logic clause that can be derived from M by applying the resolution rule and the factorization rule, we write

$$M \vdash k.$$

This is read as *M derives k* .

Definition 52 (Universal closure) *If k is a first-order logic clause and $\{x_1, \cdots, x_n\}$ is the set of all variables that occur in k, we define the universal closure $\forall(k)$ of the clause k as*

$$\forall(k) := \forall x_1 \colon \cdots \forall x_n \colon k.$$ ◇

The essential properties of the notion $M \vdash k$ are summarized in the following two theorems.

Satz 53 (Correctness Theorem)
If $M = \{k_1, \cdots, k_n\}$ is a set of clauses and $M \vdash k$, then

$$\models \forall(k_1) \wedge \cdots \wedge \forall(k_n) \rightarrow \forall(k).$$

Thus, if a clause k can be derived from a set M, then k is indeed a consequence of M. □

The converse of the above correctness theorem holds only for the empty clause. It was proven in 1965 by John A. Robinson [Rob65].

Satz 54 (Refutational Completeness (Robinson, 1965))
If $M = \{k_1, \cdots, k_n\}$ is a set of clauses and M is unsatisfiable, i.e. we have

$$\models \forall(k_1) \wedge \cdots \wedge \forall(k_n) \rightarrow \bot,$$

then the empty clause can be derived from M, i.e. we have

$$M \vdash \{\}.$$ □

We now have a method to investigate whether $\models f$ holds for a given first-order logic formula f.

1. First, we compute the Skolem normal form of $\neg f$ and obtain something like

   $$\neg f \approx_e \forall x_1, \cdots, x_m \colon g.$$

2. Next, we convert the matrix g into conjunctive normal form:

   $$g \Leftrightarrow k_1 \wedge \cdots \wedge k_n.$$

   Hence, we now have

   $$\neg f \approx_e k_1 \wedge \cdots \wedge k_n$$

   and it holds that:

   $$\models f \quad \text{if and only if} \quad \{\neg f\} \models \bot \quad \text{if and only if} \quad \{k_1, \cdots, k_n\} \models \bot.$$

3. According to the correctness theorem and the theorem on refutation completeness, it holds that

   $$\{k_1, \cdots, k_n\} \models \bot \quad \text{if and only if} \quad \{k_1, \cdots, k_n\} \vdash \bot.$$

   Therefore, we now try to show the inconsistency of the set $M = \{k_1, \cdots, k_n\}$ by deriving the empty clause from M. If this succeeds, we have demonstrated the validity of the formula f.

Example: To conclude, we demonstrate the outlined method with an example. We turn to the zoology of dragonkind and start with the following axioms [Sch08]:

1. Every dragon is happy if all its children can fly.
2. Red dragons can fly.
3. The children of a red dragon are always red.

We will show that these axioms imply that all red dragons are happy. First, we formalize the axioms and the claim in first-order logic. We choose the signature

$$\Sigma_{Drache} := \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle$$

where the sets $\mathcal{V}$, $\mathcal{F}$, $\mathcal{P}$, and *arity* are defined as follows:

1. $\mathcal{V} := \{x, y, z\}$.

2. $\mathcal{F} = \{\}$.
3. $\mathcal{P} := \{red, flies, happy, child\}$.
4. $arity := \{red \mapsto 1, flies \mapsto 1, happy \mapsto 1, child \mapsto 2\}$

The predicate $child(x, y)$ is true if and only if x is a child of y. Formalizing the axioms and the claim, we obtain the following formulas $f_1, \cdots, f_4$:

1. $f_1 := \forall x : \Big(\forall y : \big(child(y, x) \rightarrow flies(y)\big) \rightarrow happy(x)\Big)$
2. $f_2 := \forall x : \big(red(x) \rightarrow flies(x)\big)$
3. $f_3 := \forall x : \big(red(x) \rightarrow \forall y : \big(child(y, x) \rightarrow red(y)\big)\big)$
4. $f_4 := \forall x : \big(red(x) \rightarrow happy(x)\big)$

We want to show that the formula

$$f := f_1 \wedge f_2 \wedge f_3 \rightarrow f_4$$

is valid. We thus consider the formula $\neg f$ and note

$$\neg f \Leftrightarrow f_1 \wedge f_2 \wedge f_3 \wedge \neg f_4.$$

Next, we need to convert the formula on the right side of this equivalence into a set of clauses. Since this is a conjunction of several formulas, we can convert the individual formulas f_1, f_2, f_3, and $\neg f_4$ into clauses separately.

1. The formula f_1 can be transformed as follows:

$$\begin{array}{rcl}
f_1 & = & \forall x : \Big(\forall y : \big(child(y, x) \rightarrow flies(y)\big) \rightarrow happy(x)\Big) \\
& \Leftrightarrow & \forall x : \Big(\neg\forall y : \big(child(y, x) \rightarrow flies(y)\big) \vee happy(x)\Big) \\
& \Leftrightarrow & \forall x : \Big(\neg\forall y : \big(\neg child(y, x) \vee flies(y)\big) \vee happy(x)\Big) \\
& \Leftrightarrow & \forall x : \Big(\exists y : \neg\big(\neg child(y, x) \vee flies(y)\big) \vee happy(x)\Big) \\
& \Leftrightarrow & \forall x : \Big(\exists y : \big(child(y, x) \wedge \neg flies(y)\big) \vee happy(x)\Big) \\
& \Leftrightarrow & \forall x : \exists y : \Big(\big(child(y, x) \wedge \neg flies(y)\big) \vee happy(x)\Big) \\
& \approx_e & \forall x : \Big(\big(child(s(x), x) \wedge \neg flies(s(x))\big) \vee happy(x)\Big)
\end{array}$$

   In the last step, we have introduced the Skolem function s with $arity(s) = 1$. Intuitively, this function computes for each dragon x, who is not happy, a child $s(x)$, that cannot fly. If we distribute the "or" in the matrix of this formula, we obtain the following two clauses:

$$k_1 := \big\{child(s(x), x), happy(x)\big\},$$
$$k_2 := \big\{\neg flies(s(x)), happy(x)\big\}.$$

2. Similarly, for f_2 we find:
$$\begin{array}{rcl} f_2 & = & \forall x : \big(red(x) \rightarrow flies(x)\big) \\ & \Leftrightarrow & \forall x : \big(\neg red(x) \vee flies(x)\big) \end{array}$$
Thus, f_2 is equivalent to the following clause:
$$k_3 := \big\{\neg red(x), flies(x)\big\}.$$

3. For f_3, we see:
$$\begin{array}{rcl} f_3 & = & \forall x : \Big(red(x) \rightarrow \forall y : \big(child(y,x) \rightarrow red(y)\big)\Big) \\ & \Leftrightarrow & \forall x : \Big(\neg red(x) \vee \forall y : \big(\neg child(y,x) \vee red(y)\big)\Big) \\ & \Leftrightarrow & \forall x : \forall y : \big(\neg red(x) \vee \neg child(y,x) \vee red(y)\big) \end{array}$$
This yields the following clause:
$$k_4 := \big\{\neg red(x), \neg child(y,x), red(y)\big\}.$$

4. Transforming the negation of f_4 gives:
$$\begin{array}{rcl} \neg f_4 & = & \neg\forall x : \big(red(x) \rightarrow happy(x)\big) \\ & \Leftrightarrow & \neg\forall x : \big(\neg red(x) \vee happy(x)\big) \\ & \Leftrightarrow & \exists x : \neg\big(\neg red(x) \vee happy(x)\big) \\ & \Leftrightarrow & \exists x : \big(red(x) \wedge \neg happy(x)\big) \\ & \approx_e & red(d) \wedge \neg happy(d)\big) \end{array}$$
The Skolem constant d introduced here stands for an unhappy red dragon. This leads to the following two clauses:
$$k_5 = \big\{red(d)\big\},$$
$$k_6 = \big\{\neg happy(d)\big\}.$$

Therefore, we have to investigate whether the set M, which consists of the following clauses, is contradictory:

1. $k_1 = \big\{child(s(x), x),\ happy(x)\big\}$
2. $k_2 = \big\{\neg flies(s(x)),\ happy(x)\big\}$
3. $k_3 = \big\{\neg red(x),\ flies(x)\big\}$
4. $k_4 = \big\{\neg red(x),\ \neg child(y,x),\ red(y)\big\}$
5. $k_5 = \big\{red(d)\big\}$
6. $k_6 = \big\{\neg happy(d)\big\}$

In order to do this, define $M := \big\{k_1, k_2, k_3, k_4, k_5, k_6\big\}$. We will show that $M \vdash \bot$ holds:

1. It holds that

   $$\texttt{mgu}\big(\mathit{red}(d), \mathit{red}(x)\big) = [x \mapsto d].$$

   Therefore, we can apply the resolution rule to the clauses k_5 and k_4 as follows:

   $$\big\{\mathit{red}(d)\big\},\ \big\{\neg\mathit{red}(x), \neg\mathit{child}(y,x), \mathit{red}(y)\big\} \vdash \big\{\neg\mathit{child}(y,d), \mathit{red}(y)\big\}.$$

2. We now apply the resolution rule to the resulting clause and the clause k_1. For this, we first compute

   $$\texttt{mgu}\big(\mathit{child}(y,d), \mathit{child}(s(x),x)\big) = [y \mapsto s(d), x \mapsto d].$$

   Then we have

   $$\big\{\neg\mathit{child}(y,d), \mathit{red}(y)\big\},\ \big\{\mathit{child}(s(x),x), \mathit{happy}(x)\big\} \vdash \big\{\mathit{happy}(d), \mathit{red}(s(d))\big\}.$$

3. Now we apply the resolution rule to the derived clause and the clause k_6. We have:

   $$\texttt{mgu}\big(\mathit{happy}(d), \mathit{happy}(d)\big) = []$$

   Thus, we obtain

   $$\big\{\mathit{happy}(d), \mathit{red}(s(d))\big\},\ \big\{\neg\mathit{happy}(d)\big\} \vdash \big\{\mathit{red}(s(d))\big\}.$$

4. We apply the resolution rule to the clause $\big\{\mathit{red}(s(d))\big\}$ and the clause k_3. First, we have

   $$\texttt{mgu}\big(\mathit{red}(s(d)), \mathit{red}(x)\big) = [x \mapsto s(d)]$$

   Thus, the application of the resolution rule yields:

   $$\big\{\mathit{red}(s(d))\big\},\ \big\{\neg\mathit{red}(x), \mathit{flies}(x)\big\} \vdash \big\{\mathit{flies}(s(d))\big\}.$$

5. To resolve the clause $\big\{\mathit{flies}(s(d))\big\}$ with the clause k_2, we compute

   $$\texttt{mgu}\big(\mathit{flies}(s(d)), \mathit{flies}(s(x))\big) = [x \mapsto d]$$

   Then, the resolution rule gives

   $$\big\{\mathit{flies}(s(d))\big\},\ \big\{\neg\mathit{flies}(s(x)), \mathit{happy}(x)\big\} \vdash \big\{\mathit{happy}(d)\big\}.$$

6. We now apply the resolution rule to the result $\big\{\mathit{happy}(d)\big\}$ and the clause k_6:

   $$\big\{\mathit{happy}(d)\big\},\ \big\{\neg\mathit{happy}(d)\big\} \vdash \big\{\big\}.$$

Since we obtained the empty clause in the last step, we have proven that $M \vdash \bot$, and thus we have shown that all communist dragons are happy. ◇

Exercise 25: The *Russell set* R defined by Bertrand Russell is the set of all sets that do not contain themselves. Therefore, it holds that

$$\forall x : \big(x \in R \leftrightarrow \neg x \in x\big).$$

Using the calculus defined in this section, show that this formula is contradictory. ◇

Exercise 26: Given the following axioms:

1. Every barber shaves all persons who do not shave themselves.
2. No barber shaves anyone who shaves themselves.

These two axioms have a truly surprising consequence: Using only these axioms, show that all barbers are gay. ◇

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 5.30 on page 165 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 5.30: 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*`).`

   - `fof` is an abbreviation for *first-order 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 first-order 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 5.30 is stored in a file with the name `group-right-identity.tptp`, then we can invoke *Vampire* with the following command

```
vampire group-right-identity.tptp
```

This will produce the output shown in Figure 5.31. This output shows that *Vampire* is trying to perform an indirect proof, i.e. *Vampire* negates the conjecture and then tries to derive a contradiction from the negated conjecture and the axioms.

```
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 5.31: Vampire proof that the left identity is a right identity.

If we want to prove that the left inverse is also a right inverse we can simply change the last line in Figure 5.30 to

```
fof(right, conjecture, ![X]: ?[Y]: mult(X, Y) = e).
```

Exercise 27: 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.
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 5.32 on page 169 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.

5.11 *Prover9* and *Mace4**

The deductive system described in the last section can be automated and forms the basis of modern automatic provers. At the same time, the search for counterexamples can also be automated. In this section, we introduce two systems that serve these purposes.

```
% 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 5.32: Who killed Agatha?

1. *Prover9* is used to automatically prove first-order logic formulas.
2. *Mace4* examines whether a given set of first-order logic formulas is satisfiable in a finite structure. If so, this structure is computed.

The two programs, *Prover9* and *Mace4*, were developed by William McCune [McC10], are available under the GPL (*GNU General Public License*), and can be downloaded in source code form from

http://www.cs.unm.edu/~mccune/prover9/download/

First, we will discuss *Prover9* and then look at *Mace4*.

5.11.1 The Automatic Prover *Prover9*

Prover9 is a program that takes two sets of formulas as input. The first set of formulas is interpreted as a set of *axioms*, and the second set of formulas are the *theorems* to be proven from the axioms. For example, if we want to show that in group theory, the existence of a left-inverse element implies the existence of a right-inverse element, and that the left-neutral element is also right-neutral, we can axiomatize group theory as follows:

1. $\forall x : e \cdot x = x$,
2. $\forall x : \exists y : y \cdot x = e$,
3. $\forall x : \forall y : \forall z : (x \cdot y) \cdot z = x \cdot (y \cdot z)$.

We must now show that these axioms logically imply the two formulas

$$\forall x : x \cdot e = x \quad \text{and} \quad \forall x : \exists y : x \cdot y = e.$$

We can represent these formulas for *Prover9* as shown in Figure 5.33 on page 170.

```
1  formulas(sos).
2  all x (e * x = x).                              % left neutral
3  all x exists y (y * x = e).                     % left inverse
4  all x all y all z ((x * y) * z = x * (y * z)).  % associativity
5  end_of_list.
6
7  formulas(goals).
8  all x (x * e = x).                              % right neutral
9  all x exists y (x * y = e).                     % right inverse
10 end_of_list.
```

Figure 5.33: The axioms of group theory given in the syntax of Prover9.

The beginning of the axioms in this file is indicated by "`formulas(sos)`" and ends with the keyword "`end_of_list`." Note that both the keywords and each formula are terminated by a period "`.`". The axioms in lines 2, 3, and 4 express that:

1. `e` is a left-neutral element,
2. for every element x, there exists a left-inverse element y, and
3. the associative law holds.

From these axioms, it follows that `e` is also a right-neutral element and that for every element x, there exists a right-inverse element y. These two formulas are the goals to be proven and are marked in the file by "`formulas(goal)`." If the file shown in Figure 5.33 is named "`group2.in`," we can run the *Prover9* program with the command

```
prover9 -f group2.in
```

and obtain the information that the two formulas shown in lines 8 and 9 do indeed follow from the previously given axioms. If a formula cannot be proven, there are two possibilities: In certain cases, *Prover9* can actually recognize that a proof is impossible. In this case, the program stops the search for a proof with a corresponding message. If the situation is unfavorable, due to the undecidability of first-order logic, it is not possible to recognize that the search for a proof must fail. In such a case, the program runs until no more memory is available and then terminates with an error message.

Prover9 attempts to conduct an indirect proof. First, the axioms are converted into first-order logic clauses. Then, each theorem to be proven is negated, and the negated formula is also converted

into clauses. Subsequently, *Prover9* attempts to derive the empty clause from the set of all axioms along with the clauses resulting from the negation of one of the theorems to be proven. If this succeeds, it is proven that the respective theorem indeed follows from the axioms. Figure 5.34 shows an input file for *Prover9*, in which an attempt is made to deduce the commutative law from the axioms of group theory. However, the proof attempt with *Prover9* fails. In this case, the proof search is not continued indefinitely. This is because *Prover9* manages to derive all formulas that follow from the given premises in finite time. Unfortunately, such a case is the exception. Most of the time *Prover9* will abort the attempt because it runs out of memory.

```
1  formulas(sos).
2  all x (e * x = x).                                % left neutral
3  all x exists y (y * x = e).                       % left inverse
4  all x all y all z ((x * y) * z = x * (y * z)).    % associativity
5  end_of_list.
6
7  formulas(goals).
8  all x all y (x * y = y * x).                      % * is commutative
9  end_of_list.
```

Figure 5.34: Does the commutative law apply to all groups?

5.11.2 *Mace4*

If a proof attempt with *Prover9* takes forever, it is unclear whether the theorem to be proven holds. To be sure that a formula does not follow from a given set of axioms, it is sufficient to construct a Σ-structure in which all the axioms are satisfied, but the theorem to be proven is false. The program *Mace4* is specifically designed to find such structures. This, of course, only works as long as the structures are finite. Figure 5.35 shows an input file that we can use to answer the question of whether finite non-commutative groups exist using *Mace4*. Lines 2, 3, and 4 contain the axioms of group theory. The formula in line 5 postulates that for the two elements a and b, the commutative law does not hold, that is, $a \cdot b \neq b \cdot a$. If the text shown in Figure 5.35 is saved in a file named "*group.in*", we can start *Mace4* with the command

```
mace4 -f group.in
```

Mace4 searches for all positive natural numbers $n = 1, 2, 3, \cdots$, to see if there is a Σ-structure $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ with $card(U) = n$ in which the specified formulas hold. For $n = 6$, *Mace4* succeeds and indeed computes a group with 6 elements in which the commutative law is violated.

Figure 5.36 shows a part of the output produced by *Mace4*. The elements of the group are the numbers $0, \cdots, 5$, the constant a is the element 0, b is the element 1, and e is the element 2. Furthermore, we see that the inverse of 0 is 0, the inverse of 1 is 1, the inverse of 2 is 2, the inverse of 3 is 4, the inverse of 4 is 3, and the inverse of 5 is 5. The multiplication is realized by the following group

```
1  formulas(theory).
2  all x (e * x = x).                                % left neutral
3  all x exists y (y * x = e).                       % left inverse
4  all x all y all z ((x * y) * z = x * (y * z)).    % associativity
5  a * b != b * a.                                   % a and b do not commute
6  end_of_list.
```

Figure 5.35: Is there a non-commutative group?

table:

$\circ$	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

This group table shows that

$$a \circ b = 0 \circ 1 = 3, \quad \text{but} \quad b \circ a = 1 \circ 0 = 4.$$

Therefore, the commutative law is indeed violated.

Remark: The theorem prover *Prover9* is a successor of the theorem prover *Otter*. With the help of *Otter*, William McCune succeeded in proving the Robbins conjecture in 1996 [McC97]. This proof was even featured in the New York Times headline, which can be read at

`http://www.nytimes.com/library/cyber/week/1210math.html`.

This shows that automated theorem provers can indeed be useful tools. Nevertheless, first-order logic is undecidable, and so far, only a few open mathematical problems have been solved with the help of automated provers. It remains to be seen whether this will change in the future. ◇

```
============================== 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 5.36: Output of *Mace4*.

5.12 Check Your Comprehension

1. What is a signature?
2. How did we define the set $\mathcal{T}_\Sigma$ of Σ-terms?
3. Define atomic formulas!
4. How did we define the set $\mathbb{F}_\Sigma$ of Σ-formulas?
5. What is a Σ-structure?
6. Let $\mathcal{S}$ be a Σ-structure. How did we define the concept of an $\mathcal{S}$-variable assignment?
7. How did we define the semantics of Σ-formulas?
8. When is a first-order logic formula valid?
9. What does the notation $\mathcal{S} \models F$ mean for a Σ-structure $\mathcal{S}$ and a Σ-formula F?
10. When is a set of first-order logic formulas unsatisfiable?
11. What is a constraint satisfaction problem?
12. How does backtracking work?
13. Why does the order in which the different variables are instantiated matter in backtracking?
14. What are first-order logic clauses and what steps must we take to convert a given first-order logic formula into a satisfiability-equivalent set of clauses?
15. What is a substitution?
16. What is a unifier?
17. State the rules of Martelli and Montanari!
18. How is the resolution rule defined and why might it be necessary to rename variables before the resolution rule can be applied?
19. What is the factorization rule?
20. How do we proceed when we want to prove the validity of a first-order logic formula f?

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.

[Sch08] Uwe Schöning. *Logic for Computer Scientists*. Birkhäuser, 2008.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1996.

[Tur37] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1937.

Index