



Theoretical Computer Science I: Logic

— Winter 2017 —

DHBW Mannheim

Prof. Dr. Karl Stroetmann

October 16, 2017

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

<https://github.com/karlstroetmann/Logik>.

The **lecture notes** are constantly revised. Provided you have installed **git** on your computer, you can clone my repository using the command

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

Then, the repository can be updated using the command

```
git pull.
```

Contents

1	Introduction	3
1.1	Motivation	3
1.2	Overview	4
2	The Programming Language SETLX	5
2.1	Introductory Examples	5
2.2	Sets in SETLX	9
2.3	Pairs, Relations, and Functions	12
2.4	Lists	14
2.5	Special Functions and Operators on Sets	14
2.5.1	max and min	14
2.5.2	+/ and */	15
2.5.3	first, last, from, and arb	15
2.5.4	Concatenation of Lists	16
2.5.5	The Length Operator “#”	16
2.5.6	List Indexing	17
2.5.7	List Slicing	17
2.5.8	Selection Sort	17
2.6	Control Flow and Boolean Operators	18
2.6.1	Switch-Statements	21
2.6.2	while-Loops	22
2.6.3	for-Loops	23
2.7	Loading a Program	24
2.8	Strings	25
2.9	Numerical Functions	25
2.10	An Application: Fixed-Point Algorithms	26
2.11	Case Study: Computation of Poker Probabilities	28
2.12	Case Study: Finding a Path in a Graph	30
2.12.1	Computing the Transitive Closure of a Relation	31
2.12.2	Computing the Paths	34
2.12.3	The Wolf, the Goat, and the Cabbage	37
2.13	Terms and Matching	39
2.13.1	Constructing and Manipulating Terms	40
2.13.2	Matching	42

2.14	Outlook	46
3	Grenzen der Berechenbarkeit	47
3.1	Das Halte-Problem	47
3.1.1	Informale Betrachtungen zum Halte-Problem	47
3.1.2	Formale Analyse des Halte-Problems	48
3.2	Unl��sbarkeit des ��quivalenz-Problems	51
4	Aussagenlogik	53
4.1	��berblick	53
4.2	Anwendungen der Aussagenlogik	55
4.3	Formale Definition der aussagenlogischen Formeln	56
4.3.1	Syntax der aussagenlogischen Formeln	56
4.3.2	Semantik der aussagenlogischen Formeln	58
4.3.3	Extensionale und intensionale Interpretationen der Aussagenlogik	60
4.3.4	Implementierung in SETLX	60
4.3.5	Eine Anwendung	63
4.4	Tautologien	65
4.4.1	Testen der Allgemeing��ltigkeit in SETLX	67
4.4.2	Nachweis der Allgemeing��ltigkeit durch ��quivalenz-Umformungen	68
4.4.3	Berechnung der konjunktiven Normalform in SETLX	72
4.5	Der Herleitungs-Begriff	76
4.5.1	Eigenschaften des Herleitungs-Begriffs	79
4.5.2	Beweis der Widerlegungs-Vollst��ndigkeit	80
4.6	Das Verfahren von Davis und Putnam	86
4.6.1	Vereinfachung mit der Schnitt-Regel	87
4.6.2	Vereinfachung durch Subsumption	88
4.6.3	Vereinfachung durch Fallunterscheidung	88
4.6.4	Der Algorithmus	89
4.6.5	Ein Beispiel	89
4.6.6	Implementierung des Algorithmus von Davis und Putnam	90
4.7	Das 8-Damen-Problem	93
5	Pr��dikatenlogik	101
5.1	Syntax der Pr��dikatenlogik	101
5.2	Semantik der Pr��dikatenlogik	105
5.2.1	Implementierung pr��dikatenlogischer Strukturen in SETLX	109
5.3	Normalformen f��r pr��dikatenlogische Formeln	114
5.4	Unifikation	118
5.5	Ein Kalk��l f��r die Pr��dikatenlogik ohne Gleichheit	122
5.6	<i>Prover9</i> und <i>Mace4</i>	128
5.6.1	Der automatische Beweiser <i>Prover9</i>	128
5.6.2	<i>Mace4</i>	130

Chapter 1

Introduction

In this short Chapter, I would like to motivate why it is that you have to learn logic when you study computer science. After that, I will give a short overview of the lecture.

1.1 Motivation

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. For example, in the telecommunication industry it is quite common that software projects require more than a thousand developers to collaborate to develop a new system. Obviously, the failure of a project of this size is very costly. The page

Staggering Impact of IT Systems Gone Wrong

presents a number of examples showing big software projects that have failed and have subsequently caused huge financial losses. These 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. Both **mathematical logic** and **set theory** are important parts of this foundation. Furthermore, both set theory and logic have immediate applications in computer science.

1. Logic can be used to specify the interfaces of complex systems.
2. The correctness of digital circuits can be verified using automatic theorem provers.
3. Set theory and the theory of relations is the foundation of the theory of relational databases.

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 his head. The only way to control 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. As logic and set theory are already abstract concepts, engaging students with logic and set theory trains their abilities to work with abstract concepts.

From my past teaching experience I know that many students think that a good programmer already is a good computer scientist. However, a good programmer need not be a scientist, 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, so you should master it. Furthermore, this part of your education is much more permanent than the knowledge of a particular programming language. Nobody knows which programming language will be *en vogue* in 10 years from now. In three years, when you start your professional career, quite a lot of you will have to learn a new programming language. What will count then will be much more your ability to quickly grasp new concepts rather than your skills in a particular programming language.

1.2 Overview

The first lecture in theoretical computer science creates the foundation that is needed for future lectures. As set theory is already covered in the lecture on linear algebra, this lecture deals mostly with mathematical logic. Hence, this lecture is structured as follows.

1. We begin with the programming language SETLX.

SETLX (set language extended) is a programming language that is based on set theory. This language makes **set theory** available to the programmer as it supports the data structure of sets and most of the operations that are used in set theory. As SETLX is set based, it is very easy to implement set theoretical algorithms in SETLX. We will see in the coming lectures that many algorithms from theoretical computer science have a very concise and clear implementation in SETLX. Hence, the second chapter introduces SETLX.

2. Next, we investigate 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 third chapter.

3. 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 , \wedge , \vee , \rightarrow und \leftrightarrow ,

while first-order logic also investigates the **quantifiers**

\forall and \exists ,

where these quantifiers range over the objects of the **domain of discourse**. Finally, in **higher order logic** the quantifiers also range over 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 universally valid. 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 propositional logic and we will then solve this problem using propositional logic.

4. We continue to discuss **first-order logic**.

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

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

Chapter 2

The Programming Language SETLX

The introductory lecture on mathematics starts with set theory. In my experience, the notions of set theory are difficult to master for many students because the concepts introduced in set theory are quite abstract. Fortunately, there is a programming language that is directly based on set theory and logic. This is the language **SETLX**. By programming in SETLX, students can get acquainted with set theory in a playful manner. Furthermore, as many interesting problems have a straightforward solution as SETLX-programs, my experience has shown that students can appreciate the usefulness of abstract notions from set theory better by programming in SETLX.

SETLX is based on the language **SETL** [SDSD86], which was introduced in the late sixties by the renowned mathematician **Jacob T. Schwartz**. However, while the syntax of SETL is similar to **Algol**, SETLX has been designed to be syntactically similar to the programming language **C**. The language SETLX can be downloaded from the website

<http://www.randoom.org/Software/SetlX>.

I would like to mention that SETLX runs on **Android** based smart phones. The version of SETLX for Android is available at **Google Play**.

2.1 Introductory Examples

My goal is to first introduce SETLX via a number of rather simple examples. I will present more advanced features of SETLX in later sections, but this section is intended to provide a first impression of the language.

The language SETLX is an **interpreted** language. Hence, there is no need to **compile** a program. Instead, SETLX-programs can be executed via the interpreter. The interpreter is started with the command:¹

```
setlx
```

After the interpreter is started, the user sees the output that is shown in Figure 2.1 on page 6. The string “=>” is the **prompt**. It signals that the interpreter is waiting for input. If we input the string

```
1 + 2;
```

and press enter, we get the following output:

```
~< Result: 3 >~
```

```
=>
```

¹ While I am usually in the habit of terminating every sentence with either a full stop, a question mark or an exclamation mark, I refrain from doing so when the sentence ends in a SETLX-command that is shown on a separate line. The reason is that I want to avoid confusion as it can otherwise be hard to understand which part of the line is the command that has to be typed verbatim.

```

=====setlX=====v2.7.0==

Welcome to the setlX interpreter!

Open Source Software from http://setlX.random.org/
(c) 2011-2017 by Herrmann, Tom

You can display some helpful information by using '--help' as parameter when
launching this program.

Interactive-Mode:
  The 'exit;' statement terminates the interpreter.

=====Interactive=Mode=====

=>

```

Figure 2.1: The SETLX-Welcome message.

The interpreter has computed the sum $1 + 2$, returned the result, and prints another prompt waiting for more input. The command “`1 + 2;`” is a script. Of course, this is a very small script as it consists only of a single command. By default, just the last result computed by a script is output to the screen. Hence, if we feed the commands

```
1+2; 3*4;
```

to the interpreter, only the number 12 is printed. In order to print arbitrary results to the screen, we can use the function `print`. If we issue the command

```
print("Hello, World!");
```

then the following output is produced:

```

1  Hello, World!
2  ~< Result: om >~
3
4  =>

```

Here, the interpreter has first printed the string “Hello, World!”. After that, the result of the function `print` is shown. However, the function `print` does not return any value and therefore its return value is *undefined*. An undefined value is denoted using the *greek* letter Ω . This letter is then abbreviated as the string “om”.

The function `print` accepts any number of arguments. For example, printing the string “ $36 * 37 / 2$ ” followed by the value of the expression $36 \cdot 37 / 2$ can be achieved via the following command:

```
print("36 * 37 / 2 = ", 36 * 37 / 2);
```

The SETLX interpreter can be executed offline to execute programs. If the program shown in Figure 2.2 on page 7 is stored in a file with the file name “`sum.stlx`”, then we can execute this program via the following command:

```
setlx sum.stlx
```

Executing this command will first print the text

Type a natural number and press return:

to the screen. After entering a natural number n and hitting the **enter** key, the program will compute the set $\{1, \dots, n\}$ of all positive natural number less or equal n , sum the elements of this set, i.e. compute the sum

$$\sum_{i=1}^n i$$

and then print the resulting number.

```

1  // This program reads a number n and computes the sum 1 + 2 + ... + n.
2  n := read("Type a natural number and press return: ");
3  s := +/ { 1 .. n };
4  print("The sum 1 + 2 + ... + ", n, " is equal to ", s, ".");

```

Figure 2.2: A simple program to compute $\sum_{i=1}^n i$.

Let us discuss the program shown in Figure 2.2 on page 7 line by line. Note that the line numbers shown in this figure are not part of the program and have only been added so that I am able to refer to the different lines of the program more easily.

1. The first line is a comment. In SETLX, the string “//” starts a comment that extends to the end of the line. In order to have multi-line comments, we can use the strings “/*” and “*/”. Every text that starts with the string “/*” and ends with the string “*/” is ignored. Of course, this text must only contain the terminating string “*/” at the end.

Note that multi-line comments can not be nested.

2. The second line is an assignment. The expression `read(s)` first prints the string s and then reads and returns the number that is input by the user. This number is then assigned to the variable `n`. This is done using the assignment operator “:=”. It is important to understand that the syntax of SETLX differs from the syntax of the programming language C in one very important way:

SetlX uses the operator “:=” to **assign** a value to a variable, while the programming language C uses the operator “=” instead.

In contrast to the language C, the language SETLX is not **statically typed** but rather is **dynamically typed**. Hence, it is neither necessary nor possible to declare the variable `n`. Of course, in the given program, we expect the function `read` to return a number. If, instead of a number, the user inputs a string, the program would abort with an error message once the third line is executed.

3. The third line shows how a set can be defined as an enumeration. In general, if a and b are integers such that $a < b$, the expression

$$\{ a \dots b \}$$

evaluates to the set

$$\{x \in \mathbb{Z} \mid a \leq x \wedge x \leq b\}.$$

The prefix operator “+/” computes the sum of all elements of the set given to it as an argument. Since this set is equal to

$$\{1, 2, \dots, n\},$$

the operator “+” computes the sum

$$1 + 2 + \cdots + n = \sum_{i=1}^n i.$$

This sum is then assigned to the variable **s**.

4. The last line prints this variable together with some text.

The program `sum-recursive.stlx`, which is shown in Figure 2.3 on page 8 computes the sum $\sum_{i=0}^n i$ recursively.

```

1  sum := procedure(n) {
2      if (n == 0) {
3          return 0;
4      } else {
5          return sum(n-1) + n;
6      }
7  };
8
9  n      := read("Enter a natural number: ");
10 total := sum(n);
11 print("Sum 0 + 1 + 2 + ... + ", n, " = ", total);

```

Figure 2.3: A recursive program to compute $\sum_{i=0}^n i$.

1. The first seven lines define the procedure **sum**. In SETLX, the definition of a procedure is started with the keyword “**procedure**”. This keyword is followed by a list of the **formal arguments**. These arguments are separated by the character “,” and are enclosed in parentheses. As in the programming language C, the body of the procedure is enclosed in the curly braces “{” and “}”. In general, the body of a procedure consists of a list of commands. In Figure 2.3 there is only a single command. This command is a case distinction. The general form of a case distinction is as follows:

```

1      if (test) {
2          body1
3      } else {
4          body2
5      }

```

A case distinction of this form is evaluated as follows:

- (a) First, the expression **test** is evaluated. The evaluation of **test** must either return the value “**true**” or “**false**”.
- (b) If **test** evaluates as “**true**” then the statements in **body₁** are executed. Here, **body₁** is a list of statements.
- (c) Otherwise, the statements in **body₂** are executed.

Note the following **differences** with respect to the programming language C:

- (a) In SETLX, we have to enclose **body₁** and **body₂** in curly braces even if they contain only a single statement.

- (b) The definition of the procedure has to be terminated with the character “;”. The reason is that syntactically the definition of the procedure is part of an [assignment](#) and every assignment ends with a [semicolon](#). In case that we do not intend to assign the procedure to a name, for example if a procedure is used as an argument to another procedure, then the procedure is not terminated with a “;”.
- After defining the procedure `sum`, line 9 reads a number that is assigned to the variable `n`.
 - Next, line 10 calls the procedure `sum` for the given value of `n`. This value is then assigned to the variable `total`.
 - Finally, the result is printed.

The procedure `sum` is an example of a *recursive function*, i.e. the function `sum` calls itself. The logic of this recursion is captured by the following equations:

- $\text{sum}(0) = 0$,
- $n > 0 \rightarrow \text{sum}(n) = \text{sum}(n - 1) + n$.

These equations become evident if we substitute the definition

$$\text{sum}(n) = \sum_{i=0}^n i$$

in these equations, since we have:

- $\text{sum}(0) = \sum_{i=0}^0 i = 0$,
- $\text{sum}(n) = \sum_{i=0}^n i = \left(\sum_{i=0}^{n-1} i \right) + n = \text{sum}(n - 1) + n$.

The first equation deals with the case that the procedure `sum` does not call itself. This case is called the [base case](#). Every recursive function must have a base case, for otherwise the recursion would never stop.

2.2 Sets in SETLX

The most prominent difference between the programming language SETLX and the programming language C is the fact that SETLX has language support for both [sets](#) and [lists](#). In order to demonstrate how sets are supported in SETLX we present a simple program that shows how to compute the [union](#), the [intersection](#), and the [difference](#) of two sets. Furthermore, the program shows how to compute the [power set](#) of a given set and it shows how to compare sets. Figure 2.4 on page 10 shows the file `simple.stlx`. We discuss it line by line.

- The first two lines show that sets can be defined as explicit [enumerations](#) of their elements.
- Line 4, 7, and 10 compute the [union](#), the [intersection](#), and the set [difference](#) of the sets `A` and `B` respectively. Hence, the mathematical operator “ \cup ” corresponds to “+”, “ \cap ” corresponds to “*”, while “ \setminus ” corresponds to “-”.
- Line 13 computes the [power set](#) of the set `A`.
- Line 16 checks whether `A` is a [subset](#) of `B`.
- Line 18 checks whether the number 1 is an element of the set `A`.

```

1  A := { 1, 2, 3 };
2  B := { 2, 3, 4 };
3  // compute the union           A ∪ B
4  C := A + B;
5  print(A, " + ", B, " = ", C);
6  // compute the intersection    A ∩ B
7  C := A * B;
8  print(A, " * ", B, " = ", C);
9  // compute the set difference  A \ B
10 C := A - B;
11 print(A, " - ", B, " = ", C);
12 // compute the power set       2A
13 C := 2 ** A;
14 print("2 ** ", A, " = ", C);
15 // test the subset relation     A ⊆ B
16 print("(", A, " <= ", B, ") = ", (A <= B));
17 // test, whether 1 ∈ A
18 print("1 in ", A, " = ", 1 in A);
19 // compute the cartesian product
20 C := A >< B;
21 print(A, " >< ", B, " = ", C);

```

Figure 2.4: Computation of union, intersection, set difference, and power set.

6. Line 20 computes the **Cartesian product** of A and B. The **Cartesian product** “×” is translated into the operator “><” in SETLX.

If we execute this program, the following results are obtained:

```

{1, 2, 3} + {2, 3, 4} = {1, 2, 3, 4}
{1, 2, 3} * {2, 3, 4} = {2, 3}
{1, 2, 3} - {2, 3, 4} = {1}
2 ** {1, 2, 3} = {{}, {1}, {1, 2}, {1, 2, 3}, {1, 3}, {2}, {2, 3}, {3}}
({1, 2, 3} <= {2, 3, 4}) = false
1 in {1, 2, 3} = true
{1, 2, 3} >< {2, 3, 4} =
  { [1, 2], [1, 3], [1, 4], [2, 2], [2, 3], [2, 4], [3, 2], [3, 3], [3, 4] }

```

In order to be able to present more interesting programs, we present a number of ways to define more complex sets in SETLX.

Defining Sets as Arithmetic Progressions

In the previous example we have defined sets as explicit enumerations of their elements. Of course, this approach is much too tedious when working with sets containing large numbers of elements. An alternative way is to define a set as an **arithmetic progression**. Let us consider an example. The assignment

```
A := { 1 .. 100 };
```

defines A as the set of all positive natural numbers that are less than or equal to 100. The general form of an arithmetic progression is

```
A := { start .. stop };
```

This definition assigns the set of all integer numbers from *start* up to and including *stop* to the variable *A*, i.e. we have

$$A = \{n \in \mathbb{Z} \mid \text{start} \leq n \wedge n \leq \text{stop}\}.$$

We can define arithmetic progressions with a **step size** different from 1. For example, the assignment

```
A := { 1, 3 .. 100 };
```

assigns the set of all odd natural numbers less than or equal to 100 to *a*. Of course, the number 100 is not part of this set as 100 is an even number. The general form of this kind of progression is

```
A := { start, second .. stop }
```

If we define **step** = **second** – **start** and if, furthermore, **step** is positive, then this set can be written as follows:

$$A = \{\text{start} + n \cdot \text{step} \mid n \in \mathbb{Z} \wedge \text{start} + n \cdot \text{step} \leq \text{stop}\}.$$

Note that **stop** does not have to be an element of the set

```
{ start, second .. stop }.
```

For example, we have

```
{ 1, 3 .. 6 } = { 1, 3, 5 }.
```

Defining Sets via Iterators

We can also define sets via **iterators**. Consider the following example:

```
P := { n * m : n in {2..10}, m in {2..10} };
```

After this assignment, *p* is the set of all **non-trivial** products *n* * *m* such that that have both *m* and *n* are at most 10. (A product of the form *a* · *b* is called **trivial** if and only if either of the factors *a* or *b* is equal to 1.) A mathematical definition for the set *P* is as follows:

$$P = \{n \cdot m \mid n \in \mathbb{N} \wedge m \in \mathbb{N} \wedge 2 \leq n \wedge 2 \leq m \wedge n \leq 10 \wedge m \leq 10\}.$$

Iterators can be quite useful. For example, consider the program **primes-difference.stlx** that is shown in Figure 2.5 on page 11. This program computes the set of all **prime numbers** less than *n*. The underlying idea is that a number is prime ² iff it can not be written as a non-trivial product. Hence, if we take the set of all natural numbers less than or equal to *n* and bigger than 1 and subtract the set of all non-trivial products from this set, then the remaining numbers must be prime.

```

1  n := 100;
2  primes := {2 .. n} - { p * q : p in {2..n}, q in {2..n} };
3  print(primes);

```

Figure 2.5: A program to compute prime numbers.

The general form of the definition of a set via iterators is given as

$$\{\text{expr} : x_1 \text{ in } S_1, \dots, x_n \text{ in } S_n\}.$$

Here, *expr* is a term that makes use of the variables *x*₁, ..., *x*_{*n*}. Furthermore, *S*₁, ..., *S*_{*n*} are expressions that return sets (or lists) when they are evaluated. Here, an expression of the form “*x*_{*i*} in *S*_{*i*}” is called a **blueiterator** since the variables *x*_{*i*} **iterate** over the different elements of the sets *S*_{*i*}. The mathematical interpretation of the expression given above is then given as

² Henceforth, the word “iff” is used as an abbreviation for “if and only if”.

$$\{ \text{expr} \mid x_1 \in S_1 \wedge \cdots \wedge x_n \in S_n \}.$$

Hence, the definition of a set via iterators is the same as the definition of a set as an [image set](#) in set theory.

In addition to image sets we can use [selection](#) to define sets. The syntax is:

$$M := \{ \text{expr} : x_1 \text{ in } S_1, \dots, x_n \text{ in } S_n \mid \text{cond} \}.$$

Here, *expr* and S_i are interpreted as above and *cond* is an expression possibly containing the variables x_1, \dots, x_n . The evaluation of *cond* has to return either **true** or **false**. The mathematical interpretation of the expression above is then given as

$$M = \{ \text{expr} \mid x_1 \in S_1 \wedge \cdots \wedge x_n \in S_n \wedge \text{cond} \},$$

i.e. M is defined as the set of all those values that we get when we substitute those values x_i from the sets S_i into *expr* that satisfy *cond*. An example will clarify this. After the assignment

```
Primes := { p : p in {2..100} | { x : x in {1..p} | p % x == 0 } == {1, p} };
```

the variable **Primes** is the set of all prime numbers that are less than 100. The idea is that the number **p** is prime iff 1 and **p** are the only numbers that divide **p** evenly. In order to check whether **p** is evenly dividable by some number **x** we can use the operator **%** in SETLX: The expression **p % x** computes the remainder that is left over when **p** is divided by **x**. Hence,

$$\{ x : x \text{ in } \{1..p\} \mid p \% x == 0 \}$$

is the set of all those numbers that divide **p** evenly and **p** is prime if this set only contains the numbers 1 and **p**. The program **primes-slim.stlx** shown in Figure 2.6 on page 12 uses this method to compute prime numbers.

```

1  dividers := procedure(p) {
2      return { t : t in {1..p} | p % t == 0 };
3  };
4  n      := 100;
5  primes := { p : p in {2..n} | dividers(p) == {1, p} };
6  print(primes);

```

Figure 2.6: Another program to compute prime numbers.

In this program we have first defined the procedure **dividers** that takes a natural number **p** and computes the set of all those natural numbers that divide **p** evenly. Then, the set of prime numbers less than or equal to **n** is the set of those natural numbers **p** bigger than 1 that are only divided by 1 and themselves.

2.3 Pairs, Relations, and Functions

In SETLX the ordered pair $\langle x, y \rangle$ is represented as a list with two elements, i.e. it is written as $[x, y]$, so in order to represent an ordered pair in SETLX we just have to exchange the angle brackets “ \langle ” and “ \rangle ” with the square brackets “[” and “]”. In the [lecture notes on mathematics](#) it is shown that a relation that is both left-total and right-unique can be regarded as a function and hence is called a [functional relation](#). If R is a functional relation and $x \in \text{dom}(R)$, then in SETLX the expression $R[x]$ denotes the [unique](#) element y such that $\langle x, y \rangle \in R$ holds. The program **function.stlx** in Figure 2.7 on page 13 shows this more concretely. Furthermore, the program shows that for a binary relation R , in SETLX we compute $\text{dom}(R)$ as **domain**(R) and $\text{rng}(R)$ as **range**(R). Furthermore, line 2 shows that we can even change the y -value that is associated with a given x -value in a relation.

As a side note, Figure 2.7 shows that SETLX supports [string interpolation](#): Inside a string that is enclosed in double quotes, any substring that is enclosed in dollar symbols is [evaluated](#) as an expression and the substring is then replaced by the result of its evaluation.

```

1  Q := { [n, n**2] : n in {1..10} };
2  Q[5] := 7;
3  print( "Q[3]   = $Q[3]$" );
4  print( "Q[5]   = $Q[5]$" );
5  print( "dom(Q) = $domain(Q)$" );
6  print( "rng(Q) = $range(Q)$" );
7  print( "Q      = $Q$" );
8

```

Figure 2.7: Exercising a functional binary relation.

The relation Q that is computed in line 1 of Figure 2.7 represents the function $x \mapsto x^2$ on the set $\{1, \dots, 10\}$. Line 2 changes the relation Q for the argument $x = 5$ so that $Q[5]$ is 7. After that, both the domain and the range of Q are computed. The program produces the following output:

```

Q[3]   = 9
Q[5]   = 7
dom(Q) = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
rng(Q) = {1, 4, 7, 9, 16, 36, 49, 64, 81, 100}
Q      = {[1, 1], [2, 4], [3, 9], [4, 16], [5, 7], [6, 36],
          [7, 49], [8, 64], [9, 81], [10, 100]}

```

It is an interesting question to ask what happens if we evaluate $R[x]$ but the set $\{y \mid \langle x, y \rangle \in R\}$ is either empty or has more than one element. The program `buggy-function.stlx` shown in Figure 2.8 on page 13 provides us with an answer.

```

1  R := { [1, 1], [1, 4], [3, 3] };
2  print( "R[1] = $R[1]$" );
3  print( "R[2] = $R[2]$" );
4  print( "{ R[1], R[2] } = ${ R[1], R[2] }$" );
5  print( "R{1} = $R{1} $" );
6  print( "R{2} = $R{2} $" );

```

Figure 2.8: Exercising a non-functional binary relation.

If the set $\{y \mid \langle x, y \rangle \in R\}$ is either empty or contains more than one element, the expression $R[x]$ is undefined in SETLX. Trying to insert an undefined expression into a set fails. Hence, line 4 in Figure 2.8 returns the empty set. There is a way to avoid undefined values when working with non-functional binary relations. This is done by replacing the square brackets in the expression $R[x]$ with [curly brackets](#), i.e. we have to write $R\{x\}$ instead of $R[x]$. For a binary relation R , the expression $R\{x\}$ is defined as the following set:

$$R\{x\} := \{y \mid \langle x, y \rangle \in R\},$$

i.e. $R\{X\}$ is the set of all values y such that the pair $\langle x, y \rangle$ is in R . Hence, the program shown in Figure 2.8 yields the following results:

```

R[1] = om
R[2] = om
{ R[1], R[2] } = {}
R{1} = {1, 4}
R{2} = {}

```

2.4 Lists

SETLX supports the data type of a list. Lists can be defined similar to sets by replacing the curly brackets with square brackets. Then, we are able to define lists as arithmetic progressions, iterations, or via selection. The program `primes-tuple.stlx` in Figure 2.9 on page 14 demonstrates how lists can be used instead of sets. This program computes the prime numbers similar to the program shown in Figure 2.6 on page 12 but uses lists instead of sets.

```

1  dividers := procedure(p) {
2      return [ t : t in [1..p] | p % t == 0 ];
3  };
4
5  n := 100;
6  primes := [ p : p in [2 .. n] | dividers(p) == [1, p] ];
7  print(primes);

```

Figure 2.9: Computing prime numbers with lists.

2.5 Special Functions and Operators on Sets

This section discusses various functions and operators that can be applied to both sets and lists.

2.5.1 max and min

The program `sort.stlx` that is shown in Figure 2.10 on page 14 demonstrates a simple algorithm to sort a list of natural numbers. The expression

`max(L)`

computes the biggest element of the list `L`. Therefore, the variable `n` in the iterator

`n in [0 .. max(L)]`

runs from 0 up to the biggest number occurring in `L`. The condition “`n in L`” ensures that the number `n` is only inserted into the resulting list if `n` is an element of the list `L` that is to be sorted. Since the iterator

`n in [0 .. max(L)]`

generates the numbers starting from 0 and increasing, the function `sort` returns a sorted list that contains exactly those elements, that are elements of `L`. Of course, the resulting list will contain every element exactly once, even if it occurs multiple times in `L`.

```

1  sort := procedure(L) {
2      return [n : n in [0 .. max(L)] | n in L];
3  };
4  L := [13, 5, 7, 2, 4];
5  print("sort($L$) = ", sort(L));

```

Figure 2.10: A simple program to sort a given list.

It should be noted that, in general, the algorithm that is implemented in the procedure `sort` is not very efficient. We will discuss several more efficient algorithms later. However, efficiency was not the point of this

example. Rather, the point was to introduce the function `max` via a useful application. Besides `max`, SETLX provides the function `min` that computes the minimum of a list. Both `max` and `min` can also be applied to sets.

2.5.2 `+/` and `*/`

The operators `+/` and `*/` are unary prefix operators that can be applied to either a list or a set. The expression

```
+/ S
```

computes the sum of all elements of `S` and, likewise, the expression

```
*/ S
```

computes the product of the elements of `S`. Hence, we have

```
+/ {1, 2, 3, 4} = 1 + 2 + 3 + 4    and    */ {1, 2, 3, 4} = 1 · 2 · 3 · 4.
```

If either `+/` or `*/` is applied to an empty set or an empty list, the result is the undefined value `om`. To prevent these operators from returning the undefined value, the operators can also be used as binary infix operators. For a number `x` and a set or list `S`, the expression

```
x +/ S
```

returns the result `x` if `S` is empty. If `S` is not empty, the value of `x` is ignored and, instead, the sum of all elements of `S` is returned. An expression of the form

```
x */ S
```

works in a similar way: If `S` is empty, `x` is returned. Otherwise, the expression returns the product of all elements of `S`.

2.5.3 `first`, `last`, `from`, and `arb`

In SETLX, all sets are ordered. This is a notable difference from most other programming languages that support sets. Since sets are ordered, it makes sense to have functions that return the first or the last element of a set. This is achieved via the functions `first` and `last`. Note that `first` and `last` are different from `min` and `max`. The reason is, that the functions `min` and `max` can only be applied to sets that contain numbers. However, the functions `first` and `last` can be applied to any set. For example, if we define

```
S := { "a", "b", "c" };
```

then we have

```
first(S) = "a"    and    last(S) = "c".
```

However, evaluating the expression

```
min(S)
```

yields the following error message:

```
Error in "min(S)":
The set {"a", "b", "c"} is not a set of numbers.
```

```
Replay:
```

```
1.3: min(S) FAILED
```

```
1.2: S <~> {"a", "b", "c"}
```

```
1.1: min <~> procedure(collectionValue) { /* predefined procedure 'min' */ }
```

This error message tells us that the function `min` is only defined for sets of numbers. The same holds for the function `max`.

Another function that can be used to extract elements from a set is the function `from`. This function is called as follows:

```
x := from(S);
```

Here, `S` is supposed to be a set, while `x` is a variable. If the assignment above is executed, the function `from` takes some element from the set `S` and assigns it to `x`. Furthermore, this element is **removed** from the set `S`. If `S` is empty, then the undefined value `om` is assigned to the variable `x` and `S` remains empty.

At this point you might ask: When we call `from(S)`, how do we know which element is taken from `S`? The answer is that we don't know which element is taken by `from` and hence our program should work regardless of which element is removed from `S`. The program `from.stlx` in Figure 2.11 on page 16 shows how `from` can be used to print the elements of a set one by one. Here, every element of `S` is printed in a separate row.

```

1  printSet := procedure(S) {
2      if (S == {}) {
3          return;
4      }
5      x := from(S);
6      print(x);
7      printSet(S);
8  };
9  S := { 13, 5, 7, 2, 4 };
10 printSet(S);

```

Figure 2.11: Printing the elements of a set one by one.

In addition to the function `from`, SETLX provides the function `arb` that takes an arbitrary element from a given set. However, in contrast to `from`, a call to `arb` does not change the set. For example, when executing the statements

```

S := {2, 3, 5, 7, 13};
x := arb(S);
print("x = $x$");
print("S = $$");

```

we get the following output:

```

x = 13
S = {2, 3, 5, 7, 13}

```

2.5.4 Concatenation of Lists

For sets, the operator “+” computes the union. However, this operator can also be applied to lists. If `L1` and `L2` are both lists, the expression

```
L1 + L2
```

concatenates the lists `L1` and `L2`. For example, the assignment

```
L := [1, 2, 3] + [4, 5, 6];
```

creates the list `[1, 2, 3, 4, 5, 6]` and stores it into `L`.

2.5.5 The Length Operator “#”

The unary prefix operator “#” computes the length of a list when it is applied to a list. When applied to a set it returns the number of elements of this set. For example, we have

$\# [2, 3, 5, 7] = 4$ and $\# \{2, 3, 5, 3\} = 3$.

2.5.6 List Indexing

We can access the i -th elements of a list L using the notation $L[i]$, provided that i is a positive natural number that is less than or equal to the length of the list L . The list L can also be changed using this notation, so for example the assignment

$L[3] := 42;$

sets the third element of L to the number 42. Basically, the syntax is the same as the syntax of array access in the programming language C. However, there is one very important difference with respect to C:

In SETLX, list indexing starts with the number 1!

Therefore, after executing the statements

$L := [1, 2, 3];$
 $x := L[1];$

the variable x is set to 1.

In order to retrieve the last element of a list L we can use the expression “ $L[\#L]$ ”, because $\#L$ returns the number of elements of the list L . Alternatively, we can use the expression

$L[-1]$

to retrieve the last element of the list L . Similarly, the expression $L[-2]$ returns the penultimate element of the list L , while $L[-3]$ returns the ante-penultimate element of L .

2.5.7 List Slicing

Often, we have to return a sublist of a given list. This can be done using [slicing](#). If L is a list and i and j are positive natural numbers that are not greater than the length of L , then

$L[i..j]$

is the sublist of L that starts at the i -th element of L and extends to the j -th element of L . If j is less than i , the expression $L[i..j]$ returns the empty list instead. For example, after defining

$L := [5, 4, 3, 2, 1];$

we have

$L[2..4] = [4, 3, 2]$ and $L[4..2] = []$.

2.5.8 Selection Sort

In order to see a practical application of the concepts discussed so far, we present a sorting algorithm that is known as *selection sort*. This algorithm sorts a given list L and works as follows:

1. If L is empty, $\text{sort}(L)$ is also empty:

$\text{sort}([]) = []$.

2. Otherwise, we first compute the minimum of L . Clearly, the minimum needs to be the first element of the sorted list. We remove this minimum from L , sort the remaining elements recursively, and finally attach the minimum at the front of this list:

$\text{sort}(L) = [\min(L)] + \text{sort}([x \in L \mid x \neq \min(L)])$.

Figure 2.12 on page 18 shows the program `min-sort.stlx` that implements selection sort in SETLX.

```

1  minSort := procedure(L) {
2      if (L == []) {
3          return [];
4      }
5      m := min(L);
6      return [m] + minSort([x : x in L | x != m]);
7  };
8  L := [ 13, 5, 13, 7, 2, 4 ];
9  print("sort($L$) = $minSort(L)$");

```

Figure 2.12: Implementing selection sort in SETLX.

2.6 Control Flow and Boolean Operators

The language SETLX provides all those [control flow](#) statements that are used in contemporary programming languages like C or Java. We have already seen [if-then-else](#) statements on several occasions. The most general form of this kind of branching statement is shown in Figure 2.13 on page 18.

```

1      if (test0) {
2          body0
3      } else if (test1) {
4          body1
5          :
6      } else if (testn) {
7          bodyn
8      } else {
9          bodyn+1
10     }

```

Figure 2.13: The general form of a case distinction in SETLX.

Here, `testi` denotes a [Boolean expression](#), i.e. an expression that returns either “true” or “false” when evaluated, while `bodyi` is a list of statements. If the evaluation of `testi` returns “true”, then the statements in `bodyi` are executed. Otherwise, the next test `testi+1` is evaluated. If all the tests `test1`, \dots , `testn` fail, then the statements in `bodyn+1` are executed.

The tests `testi` can use the following relational infix operators:

`==`, `!=`, `>`, `<`, `>=`, `<=`, `in`.

These operators are all infix operators, and with the exception of the operator “in” they work the same way as they work in the programming language C. Hence, the operator “==” compares two objects for equality and “!=” tests whether two objects differ. For example, the Boolean expression

`{1, 2, 2} == {2, 1, 1}`

returns “true”, while the Boolean expression

`[1, 2] == [2, 1]`

returns “false”. If `x` and `y` are numbers, the expression

$x < y$

tests, whether x is less than y . Similarly, the expressions

$x > y$, $x \leq y$, and $x \geq y$

test whether x is bigger than, less than or equal to, or bigger than or equal to y , respectively. If x and y are sets instead, then

1. the expression “ $x < y$ ” is true if x is a **proper subset** of y , i.e. if $x \subset y$ holds.

A set x is a proper subset of a set y (written $x \subset y$) if x is a subset of y , and, furthermore, x is different from y , i.e. we have

$$x \subset y \stackrel{\text{def}}{\iff} x \subseteq y \wedge x \neq y.$$

2. The expression “ $x > y$ ” is true if y is a proper subset of x , i.e. if $y \subset x$ holds.
3. The expression “ $x \leq y$ ” is true if x is a subset of y , i.e. if $x \subseteq y$ holds.
4. The expression “ $x \geq y$ ” is true if y is a subset of x , i.e. if $y \subseteq x$ holds.

If x is an object and S is a set or a list, then the expression

$x \text{ in } S$

returns “**true**” if x is an element of S .

The comparison tests using the previously discussed relational operators can be combined into more complex tests via the following logical operators:

1. “**!**” represents logical **negation**, i.e. a Boolean expression of the form

!b

is **true** iff the evaluation of the Boolean expression **b** returns **false**.

2. “**&&**” represents logical **conjunction**, i.e. a Boolean expression of the form

a && b

is **true** iff the Boolean expressions **a** and **b** both evaluate to **true**.

3. “**|**” represents logical **disjunction**, i.e. a Boolean expression of the form

a || b

is **true** iff at least one of the Boolean expressions **a** or **b** evaluates to **true**.

4. “**=>**” represents logical **implication**, i.e. a Boolean expression of the form

a => b

is **true** iff **a** implies **b**.

5. “**<==>**” represents logical **equivalence**, i.e. a Boolean expression of the form

a <==> b

is **true** iff **a** has the same truth value as **b**.

6. “**<!=>**” represents logical **antivalence**, i.e. a Boolean expression of the form

a <!=> b

is **true** iff the truth values of **a** and **b** are different. Hence, the expression “ $a <!=> b$ ” is an **exclusive or** of the expressions a and b , i.e. it is true if either a or b is true but not both.

Syntactically, the operators “<!=>” and “<==>” have the lowest [precedence](#), the [precedence](#) of “=>” is lower than the [precedence](#) of the operator “||”, the [precedence](#) of “|” is lower than the [precedence](#) of the operator “&&”, and the operator “!” has the highest [precedence](#). Hence, the expression

```
!a == b && b < c || x >= y
```

is read as if it had been parenthesized as follows:

```
((!(a == b)) && b < c) || x >= y.
```

Note that the precedence of these operators is the same as it is in the programming language C.

In addition to these operators, SETLX supports [quantifiers](#). The [universal quantifier](#) is written as follows:

```
forall (x in S | b)
```

Here, x is a variable, S is a set or list and b is a Boolean expression such that the variable x occurs in b . The expression above is to be interpreted as the formula

$$\forall x \in S : b.$$

The evaluation of “forall (x in S | b)” yields **true** if evaluating the expression b yields **true** for every element x from S . For example, the expression

```
forall (x in {1, 2, 3} | x*x < 10)
```

yields **true**, because we have

$$1 \cdot 1 < 10, \quad 2 \cdot 2 < 10, \quad \text{and} \quad 3 \cdot 3 < 10.$$

A more interesting example is shown in Figure 2.14 on page 20. The program `primes-forall.stlx` computes the set of prime numbers less than 100 by making use of a universal quantifier. For a given natural number p , the Boolean expression

```
forall (x in divisors(p) | x in {1, p})
```

evaluates to **true** iff every number x that divides p evenly is either the number 1 or the number p .

```

1  isPrime := procedure(p) {
2      return p != 1 && forall (x in divisors(p) | x in { 1, p });
3  };
4  divisors := procedure(p) {
5      return { t : t in { 1 .. p } | p % t == 0 };
6  };
7  n := 100;
8  primes := [ p : p in [1.. n] | isPrime(p) ];
9  print( primes );

```

Figure 2.14: Computing the prime numbers via a universal quantifier.

Besides the universal quantifier, SETLX supports the [existential quantifier](#). The syntax of this operator is given as follows:

```
exists (x in S | b)
```

Here, x is a variable, S is a set or a list and b is a Boolean expression such that the variable x occurs in b . Mathematically, this expression is interpreted as the formula

$$\exists x \in S : b.$$

If there is at least one value for x in S such that b yields **true**, then the expression

```
exists (x in S | b)
```

is evaluated as **true**.

Remark: If the evaluation of

```
exists (x in S | b)
```

yields **true**, then the variable **x** is bound to the first value from **S** such that the evaluation of **b** returns **true**. Otherwise, **x** is set to the undefined value **om**. For example, evaluating the expression

```
exists (x in [1..10] | 2**x < x**2)
```

returns **true** and, furthermore, assigns the value 3 to the variable **x**. On the other hand, if the evaluation of

```
exists (x in S | b)
```

yields **false**, then the variable **x** is set to the undefined value.

Similarly, if the evaluation of

```
forall (x in S | b)
```

yields **false**, then the variable **x** is set to the first value from **S** that falsifies **b**. For example, after evaluating the expression

```
forall(x in [1 .. 10] | x**2 <= 2**x);
```

the variable **x** is set to the number 3 since this is the first number from the set $\{1, \dots, 10\}$ such that the expression

```
x**2 <= 2**x
```

is false. On the other hand if the evaluation of an expression of the form

```
forall (x in S | b)
```

yields **true**, then the variable **x** is set to the undefined value. ◇

2.6.1 Switch-Statements

Instead of using **if-else**-statements, it is sometimes more convenient to use a **switch**-statement. The syntax of a **switch**-statement is shown in Figure 2.15 on page 21. Here, **test**₁, ..., **test**_n are Boolean expressions, while **body**₁, ..., **body**_n, **body**_{n+1} are lists of statements. When this **switch**-statement is executed, the Boolean expressions **test**₁, ..., **test**_n are evaluated one by one until we find an expression **test**_i that is **true**. Then the corresponding statements in **body**_i are executed and the **switch**-statement ends. The block **body**_{n+1} following the keyword **default** is only executed if all of the tests **test**₁, ..., **test**_n fail. A **switch**-statement can be rewritten as a long chain of **if-else-if** ... **else-if** statements, but often a **switch**-statement is easier to understand.

```

1  switch {
2      case test1 : body1
3          :
4      case testn : bodyn
5      default   : bodyn+1
6  }
```

Figure 2.15: The general form of a **switch**-statement.

Figure 2.16 shows the program **switch.stlx**. The purpose of this program is to print a message that depends on the last digit of a number that is input by the user. In this program, the **switch**-statement results

in code that is much clearer than it would be if we had used `if-else`-statements instead. Later, the chapter on propositional logic will present examples of the `switch`-statement that are even more convincing.

```

1  print("Input a natural number:");
2  n := read();
3  m := n % 10;
4  switch {
5      case m == 0 : print("The last digit is 0.");
6      case m == 1 : print("The last digit is 1.");
7      case m == 2 : print("The last digit is 2.");
8      case m == 3 : print("The last digit is 3.");
9      case m == 4 : print("The last digit is 4.");
10     case m == 5 : print("The last digit is 5.");
11     case m == 6 : print("The last digit is 6.");
12     case m == 7 : print("The last digit is 7.");
13     case m == 8 : print("The last digit is 8.");
14     case m == 9 : print("The last digit is 9.");
15     default      : print("The impossible happened!");
16 }

```

Figure 2.16: A simple example of a `switch`-statement.

Remark: The programming language C has a `switch`-statement that is syntactically similar to the `switch`-statement in SETLX. However, the `switch`-statement is executed **differently** in C. In C, if `bodyi` is executed and `bodyi` does not contain a `break`-statement, then the following block `bodyi+1` is also executed. In contrast, SETLX will **never** execute more than one of the blocks `bodyi`.

2.6.2 while-Loops

The syntax of `while`-loops is shown in Figure 2.17 on page 22. Here, `test` is a Boolean expression and `body` is a list of statements. The evaluation of `test` must return either `true` or `false`. If the evaluation of `test` yields `false`, then the loop is terminated. Otherwise, the statements in `body` are executed. After that, the `while`-loop starts over again, i.e. the Boolean expression `test` is evaluated again and depending on the result of this evaluation the statements in `body` are executed again. This is repeated until the evaluation of `test` finally yields `false`. It should be noted that in SETLX `while`-loops work in exactly the same way as they work in the programming language C.

```

while (test) {
    body
}

```

Figure 2.17: The general form of a `while`-loop.

Figure 2.18 on page 23 shows the program `primes-while.stlx`. This program computes prime numbers using a `while`-loop. The main idea is that a number `p` is prime if there is no prime number `t` less than `p` that divides `p` evenly.

```

1  n := 100;
2  primes := {};
3  p := 2;
4  while (p <= n) {
5      if (forall (t in primes | p % t != 0)) {
6          print(p);
7          primes += { p };
8      }
9      p += 1;
10 }

```

Figure 2.18: Iterative computation of prime numbers.

2.6.3 for-Loops

The syntax of **for**-loops is shown in Figure 2.19 on page 23. Here S is either a set or a list, while x is the name of a variable. Finally, **body** is a list of statements. If S contains n elements, then the **for**-loop is executed n times. Every time the loop is executed, a different value from S is assigned to the variable x and the statements in **body** are executed using the current value of x . A **for**-loop also works if S is a string. In this case, the loop iterates over the different characters of the string S .

```

for (x in S) {
    body
}

```

Figure 2.19: General form of **for**-loops.

Figure 2.20 on page 24 shows the program `primes-for.stlx`. This program computes the prime numbers using a **for**-loop. The algorithm implemented here is known as the *sieve of Eratosthenes*. This algorithm works as follows: If n is a natural and we intend to compute all primes less than or equal to n , then we first compute a list of length n such that the i -th entry of this list is the number i . This list is called **primes** and is computed in line 2. The basic idea is now that for every index $k \leq n$ that is not prime we set **primes**[k] to 0. We know that k is not prime if it can be written as a product of the form $i \cdot j$ where both i and j are natural numbers bigger than 1. In order to set **primes**[k] to 0 for non-prime numbers k we need two loops, where the outer loop iterates over all possible values of i , while the inner loop iterates over j . The smallest value that a proper factor of any number less or equal than n can take is 2, while the largest value is $n/2$. Hence, the outer **for**-loop iterates over all values of i from 2 to $n/2$. The inner **while**-loop takes a given i and iterates over all j such that $2 \leq j$ and $i \cdot j \leq n$ is satisfied. Finally, the last line prints all i such that **primes**[i] $\neq 0$, as these are the prime numbers.

The algorithm shown in Figure 2.20 can be refined if we make use of the following observations:

1. It is sufficient if j is initialized with i because once we start eliminating the multiples of i , all multiples of i of the form $i \cdot j$ where $j < i$ have already been eliminated from the list **primes**.
2. If i is not a prime, then it can be written as $i = i' \cdot j$ where $i' < i$. Hence, any multiples of i are also multiples of i' . Therefore, if i is not prime, then there is no need to eliminate the multiples of i as these multiples have already been eliminated at the time when the multiples of i' were eliminated. For example, there is no point in eliminating the multiples of 6 as these are also multiples of 2 and hence have already been eliminated once i is set to 6.

Figure 2.21 on page 24 shows the program `primes-eratosthenes.stlx`, that makes use of these ideas. In order to skip the inner **while**-loop if i is not a prime number we have used the statement “**continue**”. This statement terminates the current iteration of the innermost loop and proceeds to the next iteration. This works in the same way as in the programming language C.

```

1  n := 100;
2  primes := [1 .. n];
3  for (i in [2 .. n]) {
4      j := 2;
5      while (i * j <= n) {
6          primes[i * j] := 0;
7          j += 1;
8      }
9  }
10 print({ i : i in [2 .. n] | primes[i] != 0 });

```

Figure 2.20: The algorithm of Eratosthenes.

```

1  n := 10000;
2  primes := [1 .. n];
3  for (i in [2 .. n/2]) {
4      if (primes[i] == 0) {
5          continue;
6      }
7      j := i;
8      while (i * j <= n) {
9          primes[i * j] := 0;
10         j += 1;
11     }
12 }
13 print({ i : i in [2 .. n] | primes[i] > 0 });

```

Figure 2.21: A more efficient version of the algorithm of Eratosthenes.

2.7 Loading a Program

The SETLX interpreter can [load](#) programs interactively into a running session. If *file* is the name of a file, then the command

```
load("file");
```

loads the program from *file* and executes the statements given in this program. For example, the command

```
load("primes-forall.stlx");
```

executes the program shown in Figure 2.14 on page 20. After loading the program, the command

```
print(isPrime);
```

shows the following output:

```
procedure (p) { return forall (x in divisors(p) | x in {1, p}); }.
```

This shows that the definitions of user defined function are available at run time once the file defining them has been loaded.

2.8 Strings

SETLX support **strings**. **Strings** are nothing more but sequences of characters. In SETLX, these have to be enclosed either in double quotes or in single quotes. The operator “+” can be used to concatenate strings. For example, the expression

```
"abc" + 'uvw';
```

returns the result

```
"abcuvw".
```

Furthermore, a natural number **n** can be multiplied with a string **s**. The expression

```
n * s;
```

returns a string consisting of **n** concatenations of **s**. For example, the result of

```
3 * "abc";
```

is the string **"abcabcabc"**. When multiplying a string with a number, the order of the arguments does not matter. Hence, the expression

```
"abc" * 3
```

also yields the result **"abcabcabc"**. In order to extract substrings from a given string, we can use the same slicing operator that also works for lists. Therefore, if **s** is a string and **k** and **l** are numbers, then the expression

```
s[k..l]
```

extracts the substring from **s** that starts with the **k**th character of **s** and ends with the **l**th character. For example, if **s** is defined by the assignment

```
s := "abcdefgh";
```

then the expression **s[2..5]** returns the substring

```
"bcde".
```

2.9 Numerical Functions

In order to support numerical computations, SETLX provides **floating point numbers**. These are internally stored as 64 bit numbers according to the **IEEE standard 754**. In order to work with floating point numbers, SETLX provides the following functions:

1. **sin**(*x*) computes the **sine** of *x*. Furthermore, the **trigonometric functions** **cos**(*x*) and **tan**(*x*) are supported. The **inverse trigonometrical functions** are written as **asin**(*x*), **acos**(*x*) and **atan**(*x*).
2. **sinh**(*x*) computes the **hyperbolic sine** of *x*. Similarly, **cosh**(*x*) returns the **hyperbolic cosine** of *x*, while **tanh**(*x*) returns the **hyperbolic tangent** of *x*.
3. **exp**(*x*) computes the **exponential function**, i.e. we have

$$\exp(x) = e^x.$$

Here, **e** denotes **Euler's number**.

4. **log**(*x*) computes the **natural logarithm** of *x*. The logarithm base 10 of *x* is computed as **log10**(*x*).
5. **abs**(*x*) computes the **absolute value** of *x*.
6. **signum**(*x*) computes the **sign function** of *x*.

7. `sqrt`(x) computes the **square root** of x , we have

$$\text{sqrt}(x) = \sqrt{x} \quad \text{and} \quad \text{sqrt}(x)^2 = x.$$

8. `cbrt`(x) computes the **cube root** of x , we have

$$\text{cbrt}(x) = \sqrt[3]{x} \quad \text{and} \quad \text{cbrt}(x)^3 = x$$

9. `ceil`(x) computes the **ceiling function** of x , i.e. `ceil`(x) is the smallest integer that is at least as big as x . We have

$$\text{ceil}(x) = \min(\{z \in \mathbb{Z} \mid z \geq x\}).$$

Hence the function `ceil` rounds up.

10. `floor`(x) computes the **floor function**, that is `floor`(x) is the biggest integer not exceeding x , we have

$$\text{floor}(x) = \max(\{z \in \mathbb{Z} \mid z \leq x\}).$$

Hence, the function `floor` rounds down.

11. `round`(x) returns the nearest integer. Floating point numbers of the form $x.5$ are rounded up. For example, `round`(1.5) = 2 and `round`(-1.5) = -1.

SETLX supports **unlimited precision**³ via rational numbers. For example, the expression

$$1/2 + 1/3;$$

returns the result 5/6. There is no over- or underflow when working with rational numbers, nor are there any rounding errors. For example, to compute **Euler's number** e we can use the formula

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}.$$

In SETLX the expression

$$\text{nDecimalPlaces}(+/ [1/n! : n \text{ in } \{0..50\}], 50);$$

computes e to a precision of 50 decimal digits. The value returned is

$$2.71828182845904523536028747135266249775724709369995.$$

The function `nDecimalPlaces`(x, n) takes a rational number x and converts this number into a decimal number. In this process, the first n decimal places following the decimal point are retained.

2.10 An Application: Fixed-Point Algorithms

Suppose we want to solve the equation

$$x = \cos(x).$$

Here, x is a real number that we seek to compute. A simple approach that works in this case is to use a **fixed-point iteration**. To this end, we define the sequence $(x_n)_{n \in \mathbb{N}}$ inductively as follows:

$$x_0 := 0 \quad \text{and} \quad x_{n+1} := \cos(x_n) \quad \text{for all } n \in \mathbb{N}.$$

With the help of the **Banach fixed-point theorem**⁴ it can be shown that this sequence converges to a solution of the equation $x = \cos(x)$, i.e. if we define

$$\bar{x} := \lim_{n \rightarrow \infty} x_n,$$

then we have

³ In this context, **unlimited precision** means that the precision is only limited by the available memory.

⁴ The Banach fixed-point theorem is discussed in the lecture on **differential calculus**. This lecture is part of the second semester.

$$\cos(\bar{x}) = \bar{x}.$$

Figure 2.22 on page 27 shows the program `solve.stlx` that uses this approach to solve the equation $x = \cos(x)$.

```

1  x := 0.0;
2  while (true) {
3      old_x := x;
4      x := cos(x);
5      print(x);
6      if (abs(x - old_x) < 1.0e-13) {
7          print("x = ", x);
8          break;
9      }
10 }
```

Figure 2.22: Solving the equation $x = \cos(x)$ via fixed-point iteration.

In this program, the iteration stops as soon as the difference between the variables `x` and `old_x` is less than 10^{-13} . Here, `x` corresponds to x_{n+1} , while `old_x` corresponds to x_n . Once the values of x_{n+1} and x_n are sufficiently close, the execution of the `while` loop is stopped using the `break` statement. This statement works the same way as in the programming language C, i.e. it terminates the execution of the innermost loop containing the `break` statement.

```

1  solve := procedure(f, x0) {
2      x := x0;
3      for (n in [1 .. 10000]) {
4          oldX := x;
5          x := f(x);
6          if (abs(x - oldX) < 1.0e-12) {
7              return x;
8          }
9      }
10 };
11 print("solution to x = cos(x): ", solve(cos, 0));
12 print("solution to x = 1/(1+x): ", solve(x |-> 1.0/(1+x), 0));
```

Figure 2.23: A generic implementation of the fixed-point algorithm.

Figure 2.23 on page 27 shows the program `fixpoint.stlx`. In this program we have implemented a function `solve` that takes two arguments.

1. `f` is a unary function. The purpose of the `solve` is to compute the solution of the equation

$$f(x) = x.$$

This equation is solved with the help of a fixed-point algorithm.

2. `x0` is used as the initial value for the fixed-point iteration.

Line 11 calls `solve` to compute the solution of the equation $x = \cos(x)$. Line 12 solves the equation

$$x = \frac{1}{1+x}.$$

This equation is equivalent to the quadratic equation $x^2 + x = 1$. Note that we have defined the function

$x \mapsto \frac{1}{1+x}$ via the expression

`x |-> 1.0/(1+x).`

This expression is called an **anonymous function** since we haven't given a name to the function. It is also important to note that we have used the floating point number 1.0 instead of the integer 1. The reason is that otherwise SETLX would use rational numbers when doing the fixed-point iteration. Although this would work, arithmetic using rational numbers is considerably less efficient than arithmetic using floating point numbers.

Remark: The function `solve` is only able to solve the equation $f(x) = x$ if the function f is a **contraction mapping**. A function $f : \mathbb{R} \rightarrow \mathbb{R}$ is called a **contraction mapping** iff

$$|f(x) - f(y)| < |x - y| \quad \text{for all } x, y \in \mathbb{R}.$$

This notion will be discussed in more detail in the lecture on **differential calculus** in the second semester. ◇

2.11 Case Study: Computation of Poker Probabilities

In this short section we are going to show how to compute probabilities for the **Texas Hold'em** variation of **poker**. Texas Hold'em poker is played with a deck of 52 cards. Every card has a **value**. This value is an element of the set

$$\text{values} = \{2, 3, 4, 5, 6, 7, 8, 9, 10, \text{Jack}, \text{Queen}, \text{King}, \text{Ace}\}.$$

Furthermore, every card has a **suit**. This suit is an element of the set

$$\text{suits} = \{\clubsuit, \heartsuit, \diamondsuit, \spadesuit\}.$$

These suits are pronounced **club**, **heart**, **diamond**, and **spade**. As a card is determined by its value and its suit, a card can be represented as a pair $\langle v, s \rangle$, where v denotes the value while s is the suit of the card. Hence, the set of all cards can be represented as the set

$$\text{deck} = \{\langle v, s \rangle \mid v \in \text{values} \wedge s \in \text{suits}\}.$$

At the start of a game of Texas Hold'em, every player receives two cards. These two cards are known as the **preflop** or the **hole**. Next, there is a **bidding phase** where players can bet on their cards. After this bidding phase, the dealer puts three cards open on the table. These three cards are known as **flop**. Let us assume that a player has been dealt the set of cards

$$\{\langle 3, \clubsuit \rangle, \langle 3, \spadesuit \rangle\}.$$

This set of cards is known as a **pocket pair**. Then the player would like to know the probability that the flop will contain another card with value 3, as this would greatly increase her chance of winning the game. In order to compute this probability we have to compute the number of possible flops that contain a card with the value 3 and we have to divide this number by the number of all possible flops:

$$\frac{\text{number of flops containing a card with value 3}}{\text{number of all possible flops}}$$

The program `poker-triple.stlx` shown in Figure 2.24 performs this computation. We proceed to discuss this program line by line.

1. In line 1 the set **values** is defined to be the set of all possible values that a card can take. In defining this set we have made use of the following abbreviations:





- (a) “T” is short for “**Ten**”,
- (b) “J” is short for “**Jack**”,
- (c) “Q” is short for “**Queen**”,
- (d) “K” is short for “**King**”, and

```

1  values := { "2", "3", "4", "5", "6", "7", "8", "9", "T", "J", "Q", "K", "A" };
2  suits  := { "c", "h", "d", "s" };
3  deck   := { [ v, s ] : v in values, s in suits };
4  hole   := { [ "3", "c" ], [ "3", "s" ] };
5  rest   := deck - hole;
6  flops  := { { k1, k2, k3 } : k1 in rest, k2 in rest, k3 in rest
7                          | #{ k1, k2, k3 } == 3
8                      };
9  trips  := { f : f in flops | [ "3", "d" ] in f || [ "3", "h" ] in f };
10 print(1.0 * #trips / #flops);

```

Figure 2.24: Computing a probability in poker.

- (e) “A” is short for “[Ace](#)”.
2. In line 2 the set `suits` represents the possible suits of a card. Here, we have used the following abbreviations:
 - (a) “c” is short for , which is pronounced as [club](#),
 - (b) “h” is short for , which is pronounced as [heart](#),
 - (c) “d” is short for , which is pronounced as [diamond](#), and
 - (d) “s” is short for , which is pronounced as [spade](#).
 3. Line 3 defines the set of all cards. This set is stored as the variable `deck`. Every card is represented as a pair of the form $[v, s]$. Here, v is the value of the card, while s is its suit.
 4. Line 4 defines the set `hole`. This set represents the two cards that have been given to our player.
 5. The remaining cards are defined as the variable `rest` in line 5.
 6. Line 6 computes the set of all possible flops. Since the order of the cards in the flop does not matter, we use sets to represent these flops. However, we have to take care that the flop does contain three **different** cards. Hence, we have to ensure that the three cards `k1`, `k2`, and `k3` that make up the flop satisfy the inequalities

$$k1 \neq k2, \quad k1 \neq k3, \quad \text{and} \quad k2 \neq k3.$$

These inequalities are satisfied if and only if the set $\{k1, k2, k3\}$ contains exactly three elements. Hence, when choosing `k1`, `k2`, and `k3` we have to make sure that the condition

$$\#\{ k1, k2, k3 \} == 3$$

holds.

7. Line 9 computes the subset of flops that contain at least one card with a value of 3. As the 3 of clubs and the 3 of spades have already been dealt to our player, the only cards with value 3 that are left in the deck are the 3 of diamonds and the 3 of hearts. Therefore, we are looking for those flops that contain one of these two cards.
8. Finally, the probability for obtaining another card with a value of 3 in the flop is computed as the ratio of the number of flops containing a card with a value of 3 to the number of all possible flops.

However, we have to be careful here: The evaluation of the expressions `#trips` and `#flops` produces [integer](#) numbers. Therefore, the division `#trips / #flops` yields a [rational](#) number. As we intend to compute a [floating point](#) number we have to convert the result into a floating point number by multiplying the result with the floating point number 1.0.

When we run the program we see that the probability of improving a **pocket pair** on the flop to **trips** or better is about 11.8%.

Remark: The method to compute probabilities that has been sketched above only works if the sets that have to be computed are small enough to be retained in memory. If this condition is not satisfied we can use the *Monte Carlo method* to compute the probabilities instead. This method will be discussed in the lecture on *algorithms*.

2.12 Case Study: Finding a Path in a Graph

In the following section, I will present an application that is more interesting since it is practically relevant. In order to prepare for this, we will now discuss the problem of finding a **path** in a **directed graph**. Abstractly, a graph consists of **vertices** and **edges** that connect these vertices. In an application, the vertices could be towns and villages, while the edges would be interpreted as streets connecting these villages. To simplify matters, let us assume for now that the vertices are given as natural numbers, while the edges are represented as pairs of natural numbers. Then, the graph can be represented as the set of its edges, as the set of vertices is implicitly given once the edges are known. To make things concrete, let us consider an example. In this case, the set of edges is called R and is defined as follows:

$$R := \{\langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 1, 3 \rangle, \langle 2, 4 \rangle, \langle 4, 5 \rangle\}.$$

In this graph, the set of vertices is given as

$$\{1, 2, 3, 4, 5\}.$$

This graph is shown in Figure 2.25 on page 30. You should note that the connections between vertices that are given in this graph are unidirectional: While there is a connection from vertex 1 to vertex 2, there is no connection from vertex 2 to vertex 1.

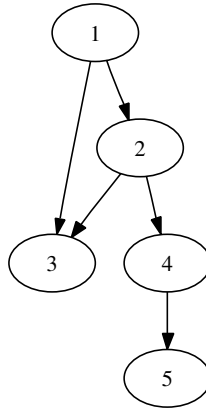


Figure 2.25: A simple graph.

The graph given by the relation R contains only the direct connections of vertices. For example, in the graph shown in Figure 2.25, there is a direct connection from vertex 1 to vertex 2 and another direct connection from vertex 2 to vertex 4. Intuitively, vertex 4 is reachable from vertex 1, since from vertex 1 we can first reach vertex 2 and from vertex 2 we can then reach vertex 4. However, there is no direct connection between the vertices 1 and 4. To make this more formal, define a **path** of a graph R as a list of vertices

$$[x_1, x_2, \dots, x_n] \quad \text{such that} \quad \langle x_i, x_{i+1} \rangle \in R \quad \text{for all } i = 1, \dots, n-1.$$

In this case, the path $[x_1, x_2, \dots, x_n]$ is written as

$$x_1 \mapsto x_2 \mapsto \dots \mapsto x_n$$

and has the **length** $n - 1$. It is important to note that the length of a path $[x_1, x_2, \dots, x_n]$ is defined as the number of edges connecting the vertices and not as the number of vertices appearing in the path.

Furthermore, two vertices a and b are said to be **connected** iff there exists a path

$$[x_1, \dots, x_n] \quad \text{such that} \quad a = x_1 \quad \text{and} \quad b = x_n.$$

The goal of this section is to develop an algorithm that checks whether two vertices a and b are connected. Furthermore, we want to be able to compute the corresponding path connecting the vertices a and b .

2.12.1 Computing the Transitive Closure of a Relation

We have already noted that a graph can be represented as the set of its edges and hence as a **relation**. In order to decide whether there is a path connecting two vertices we have to compute the **transitive closure** R^+ of a relation R . In the **math lecture** we have seen that the transitive closure R^+ can be computed as follows:

$$R^+ = \bigcup_{n=1}^{\infty} R^n = R^1 \cup R^2 \cup R^3 \cup \dots$$

Initially, this formula might look intimidating as it suggests an infinite computation. Fortunately, it turns out that we do not have to compute all powers of the form R^n . Let me explain the reason that allows us to cut the computation short.

1. R is the set of direct connections between two vertices.
2. R^2 is the same as $R \circ R$ and this relational product is defined as

$$R \circ R = \{\langle x, z \rangle \mid \exists y: \langle x, y \rangle \in R \wedge \langle y, z \rangle \in R\}.$$

Hence, $R \circ R$ contains those pairs $\langle x, z \rangle$ that are connected via one intermediate vertex y , i.e. there is a path of the form $x \mapsto y \mapsto z$ that connects x and z . This path has length 2. In general, we can show by induction that R^n connect those pairs that are connected by a path of length n . The induction step of this proof runs as follows:

3. R^{n+1} is defined as $R \circ R^n$ and therefore we have

$$R \circ R^n = \{\langle x, z \rangle \mid \exists y: \langle x, y \rangle \in R \wedge \langle y, z \rangle \in R^n\}.$$

As $\langle y, z \rangle \in R^n$, the induction hypothesis guarantees that the vertices y and z are connected by a path of length n . Hence, this path has the form

$$\underbrace{y \mapsto \dots \mapsto z}_{\text{path of length } n}.$$

Adding x at the front of this path will produce the path

$$x \mapsto y \mapsto \dots \mapsto z.$$

This path has a length of $1 + n = n + 1$ and, furthermore, connects x and z . Hence R^{n+1} contains those pairs $\langle x, z \rangle$ that are connected by a path of length $n + 1$.

Now the important observation is the following. The set of all vertices is finite. For the arguments sake, let us assume there are k vertices. But then every path that has a length of k or greater must contain one vertex that is visited at least twice and hence this path is longer than necessary, i.e. there is a shorter path that connects the same vertices. Therefore, for a finite graph with k vertices, the formula to compute the transitive closure can be simplified as follows:

$$R^+ = \bigcup_{i=1}^{k-1} R^i.$$

While we could use this formula as it stands, it is more efficient to use a [fixed-point iteration](#) instead. To this end, we prove that the transitive closure R^+ satisfies the following equation:

$$R^+ = R \cup R \circ R^+. \quad (2.1)$$

Let me remind you that the precedence of the operator \circ is higher than the precedence of the operator \cup . Therefore, the expression $R \cup R \circ R^+$ is parenthesized as $R \cup (R \circ R^+)$. Equation 2.1 can be proven algebraically. We have:

$$\begin{aligned} & R \cup R \circ R^+ \\ = & R \cup R \circ \bigcup_{i=1}^{\infty} R^i \\ = & R \cup R \circ (R^1 \cup R^2 \cup R^3 \cup \dots) \\ = & R \cup (R \circ R^1 \cup R \circ R^2 \cup R \circ R^3 \cup \dots) \\ = & R \cup (R^2 \cup R^3 \cup R^4 \cup \dots) \\ = & R^1 \cup (R^2 \cup R^3 \cup R^4 \cup \dots) \\ = & \bigcup_{i=1}^{\infty} R^i \\ = & R^+. \end{aligned}$$

Equation 2.1 can now be used to compute R^+ via a fixed-point iteration. To this end, let us define a sequence of relations $(T_n)_{n \in \mathbb{N}}$ by induction on n :

I.A. $n = 0$:

$$T_0 := R$$

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

$$T_{n+1} := R \cup R \circ T_n.$$

The relation T_n can be expressed via the relation R , we have

1. $T_0 = R$.
2. $T_1 = R \cup R \circ T_0 = R \cup R \circ R = R^1 \cup R^2$.
3. $\begin{aligned} T_2 &= R \cup R \circ T_1 \\ &= R \cup R \circ (R^1 \cup R^2) \\ &= R^1 \cup R^2 \cup R^3. \end{aligned}$

In general, we can show by induction that

$$T_n = \bigcup_{i=1}^{n+1} R^i$$

holds for all $n \in \mathbb{N}$. The base case of this proof is immediate from the definition of T_0 . In the induction step we observe the following:

$$\begin{aligned} T_{n+1} &= R \cup R \circ T_n && \text{(by definition)} \\ &= R \cup R \circ \left(\bigcup_{i=1}^{n+1} R^i \right) && \text{(by induction hypothesis)} \\ &= R \cup R \circ (R \cup \dots \cup R^{n+1}) \\ &= R \cup R^2 \cup \dots \cup R^{n+2} && \text{(by the distributivity of } \circ \text{ over } \cup) \\ &= \bigcup_{i=1}^{n+2} R^i && \square \end{aligned}$$

The sequence $(T_n)_{n \in \mathbb{N}}$ has another useful property: It is **monotonically increasing**. In general, a sequence of sets $(X_n)_{n \in \mathbb{N}}$ is called **monotonically increasing** iff we have

$$\forall n \in \mathbb{N} : X_n \subseteq X_{n+1},$$

i.e. the sets X_n get bigger with growing index n . The monotonicity of the sequence $(T_n)_{n \in \mathbb{N}}$ is an immediate consequence of the equation

$$T_n = \bigcup_{i=1}^{n+1} R^i$$

because we have:

$$\begin{aligned} T_n &\subseteq T_{n+1} \\ \Leftrightarrow \bigcup_{i=1}^{n+1} R^i &\subseteq \bigcup_{i=1}^{n+2} R^i \\ \Leftrightarrow \bigcup_{i=1}^{n+1} R^i &\subseteq \bigcup_{i=1}^{n+1} R^i \cup R^{n+2} \end{aligned}$$

If the relation R is finite, then the transitive closure R^+ is finite, too. The sets T_n are all subsets of R^+ because we have

$$T_n = \bigcup_{i=1}^{n+1} R^i \subseteq \bigcup_{i=1}^{\infty} R^i = R^+ \quad \text{for all } n \in \mathbb{N}.$$

Hence the sets T_n can not grow indefinitely. Because of the monotonicity of the sequence $(T_n)_{n \in \mathbb{N}}$ it follows that there exists an index $k \in \mathbb{N}$ such that the sets T_n do not grow any further once n has reached k , i.e. we have

$$\forall n \in \mathbb{N} : (n \geq k \rightarrow T_n = T_k).$$

But this implies that

$$T_n = \bigcup_{i=1}^{n+1} R^i = \bigcup_{i=1}^{\infty} R^i = R^+ \quad \text{holds for all } n \geq k.$$

Therefore, the algorithm for computing R^+ iterates the equation

$$T_{n+1} := R \cup R \circ T_n$$

until the equation $T_{n+1} = T_n$ is satisfied, since this implies that $T_n = R^+$.

The program **transitive-closure.stlx** that is shown in Figure 2.26 on page 34 shows an implementation of this idea. The program produces the following output:

```
R = {[1, 2], [2, 3], [1, 3], [2, 4], [4, 5]}
Computing the transitive closure of R:
R+ = {[1, 2], [1, 3], [1, 4], [1, 5], [2, 3], [2, 4], [2, 5], [4, 5]}
```

The transitive closure R^+ of a relation R has a very intuitive interpretation: R^+ : It contains all pairs $\langle x, y \rangle$ such that there is a path leading from x to y . The function **product**(R_1, R_2) computes the relational product $R_1 \circ R_2$ according to the formula

$$R_1 \circ R_2 = \{ \langle x, z \rangle \mid \exists y : \langle x, y \rangle \in R_1 \wedge \langle y, z \rangle \in R_2 \}.$$

The implementation of the procedure **product** shows the most general way to define a set in SETLX. In general, a set can be defined via an expression of the form

$$\{ \text{expr} : [x_1^{(1)}, \dots, x_{n(1)}^{(1)}] \text{ in } s_1, \dots, [x_1^{(k)}, \dots, x_{n(k)}^{(k)}] \text{ in } s_k \mid \text{cond} \}.$$

Here, for all $i = 1, \dots, k$ the variable s_i denotes a set of lists of length $n(i)$. When the expression given above

```

1  transClosure := procedure(R) {
2      T := R;
3      while (true) {
4          oldT := T;
5          T := R + product(R, T);
6          if (T == oldT) {
7              return T;
8          }
9      }
10 };
11 product := procedure(R1, R2) {
12     return { [x,z] : [x,y] in R1, [y,z] in R2 };
13 };
14 R := { [1,2], [2,3], [1,3], [2,4], [4,5] };
15 print( "R = ", R );
16 print( "Computing the transitive closure of R:" );
17 T := transClosure(R);
18 print( "R+ = ", T );

```

Figure 2.26: Computing the transitive closure.

is evaluated, the variables $x_1^{(i)}, \dots, x_{n(i)}^{(i)}$ are replaced by the corresponding values in the lists from the sets s_i . For example, if we define

```

s1 := { [ 1, 2, 3 ], [ 5, 6, 7 ] };
s2 := { [ "a", "b" ], [ "c", "d" ] };
m := { [ x1, x2, x3, y1, y2 ] : [ x1, x2, x3 ] in s1, [ y1, y2 ] in s2 };

```

then the set m has the following value:

```

{ [1, 2, 3, "a", "b"], [5, 6, 7, "c", "d"],
  [1, 2, 3, "c", "d"], [5, 6, 7, "a", "b"] }

```

2.12.2 Computing the Paths

So far, given a graph represented by a relation R and two vertices x and y , we can only check whether there is a path leading from x to y , but we cannot compute this path. In this subsection we will extend the procedure `transClosure` so that it will also compute the corresponding path. The main idea is to extend the notion of a relational product to the notion of a [path product](#), where a [path product](#) is defined on sets of paths. In order to do so, we introduce three functions for lists.

1. Given a list p , the function `first(p)` returns the first element of p :

$$\text{first}([x_1, \dots, x_m]) = x_1.$$

2. Given a list p , the function `last(p)` returns the last element of p :

$$\text{last}([x_1, \dots, x_m]) = x_m.$$

3. If $p = [x_1, \dots, x_m]$ and $q = [y_1, \dots, y_n]$ are two path such that `first(q) = last(p)`, we define the [join](#) of p and q as

$$p \oplus q := [x_1, \dots, x_m, y_2, \dots, y_n].$$

```

1  transClosure := procedure(R) {
2      P := R;
3      while (true) {
4          oldP := P;
5          P := R + pathProduct(R, P);
6          print(P);
7          if (P == oldP) {
8              return P;
9          }
10     }
11 };
12 pathProduct := procedure(P, Q) {
13     return { add(x, y) : x in P, y in Q | x[-1] == y[1] };
14 };
15 add := procedure(p, q) {
16     return p + q[2..];
17 };
18 R := { [1,2], [2,3], [1,3], [2,4], [4,5] };
19 print( "R = ", R );
20 print( "computing all paths" );
21 P := transClosure(R);
22 print( "P = ", P );

```

Figure 2.27: Computing all connections.

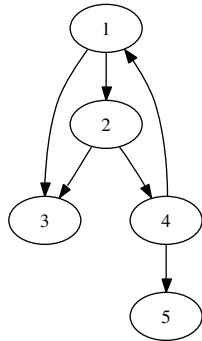


Figure 2.28: A graph with a cycle.

If P_1 and P_2 are sets of paths, we define the **path product** of P_1 and P_2 as follows:

$$P_1 \bullet P_2 := \{ p_1 \oplus p_2 \mid p_1 \in P_1 \wedge p_2 \in P_2 \wedge \text{last}(p_1) = \text{first}(p_2) \}.$$

Using the notion of a **path product** we are able to extend the program shown in Figure 2.26 such that it computes all paths between two vertices. The resulting program **path.stlx** is shown in Figure 2.27 on page 35. Unfortunately, the program does not work any more if the graph is **cyclic**. A graph is defined to be **cyclic** if there is a path of length greater than 1 that starts and ends at the same vertex. This path is then called a **cycle**. Figure 2.28 on page 35 shows a cyclic graph. This graph is cyclic because it contains the path

$[1, 2, 4, 1]$

and this path is a cycle. The problem with this graph is that it contains an infinite number of paths that connect the vertex 1 with the vertex 2:

$[1, 2], [1, 2, 4, 1, 2], [1, 2, 4, 1, 2, 4, 1, 2], [1, 2, 4, 1, 2, 4, 1, 2, 4, 1, 4], \dots$

Of course, there is no point in computing a path that visits a vertex more than once as these paths contain cycles. Our goal is to eliminate all those paths that contain cycles.

```

1  pathProduct := procedure(P, Q) {
2      return { add(x,y) : x in P, y in Q | x[-1] == y[1] && noCycle(x, y) };
3  };
4  noCycle := procedure(L1, L2) {
5      return #({ x : x in L1 } * { x : x in L2 }) == 1;
6  };

```

Figure 2.29: Computing the connections in a cyclic graph.

Figure 2.29 on page shows how the implementation of the function `pathProduct` has to be changed so that the resulting program `path-cyclic.stlx` works also for cyclic graphs.

1. In line 2, we compute only those paths that are not cyclic.
2. Line 5 tests, whether the join $L1 \oplus L2$ is cyclic. The join of $L1$ and $L2$ is cyclic iff the lists $L1$ and $L2$ have more than one common element. The lists $L1$ and $L2$ will always have at least one common element, as we join these lists only if the last element of $L1$ is equal to the first element of $L2$. If there would be an another vertex common to $L1$ and $L2$, then the path $L1 \oplus L2$ would be cyclic.

In general, we are not really interested to compute all possible paths between two given vertices x and y . Instead, we just want to compute the shortest path leading from x to y . Figure 2.30 on page 37 shows the procedure `reachable`. This procedure takes three arguments:

1. x and y are vertices of a graph.
2. R is a binary relation representing a directed graph.

The call `reachable(x, y, R)` checks whether x and y are connected and, furthermore, computes the shortest path from x to y , provided such a path exists. The complete program can be found in the file `find-path.stlx`. Next, we discuss the implementation of the procedure `reachable`.

1. Line 2 initializes the set P . After n iterations, this set will contain all paths that start in the vertex x and that have a length of at most n .
Initially, there is just the trivial path $[x]$ that starts in x and has length 0.
2. Line 5 tries to extend all previously computed paths by one step. If we are lucky, the set P is increased in this step.
3. Line 6 selects all those paths from the set P that lead to the vertex y . These paths are stored in the set `Found`.
4. Line 7 checks whether we have indeed found a path ending at y . This is the case if the set `Found` is not empty. In this case, we return any of these paths.
5. If we have not yet found the vertex y and, furthermore, we have not been able to find any new paths during this iteration, the procedure returns in line 11. As the `return` statement in line 11 does not return a value, the procedure will instead return the undefined value Ω .

The procedure call `reachable(x, y R)` will compute the **shortest** path connecting `x` and `y` because it computes path with increasing length. The first iteration computes all paths starting in `x` that have a length of at most 1, the second iteration computes all paths starting in `x` that have a length of at most 2, and in general the n -th iteration computes all paths starting in `x` that have a length of at most n . Hence, if there is a path of length n , then this path will be found in the n -iteration unless a shorter path has already been found in a previous iteration.

Remark: The algorithm described above is known as **breadth first search**. ◇

```

1  reachable := procedure(x, y, R) {
2      P := { [x] };
3      while (true) {
4          oldP := P;
5          P := P + pathProduct(P, R);
6          Found := { l : l in P | l[-1] == y };
7          if (Found != {}) {
8              return arb(Found);
9          }
10         if (P == oldP) {
11             return;
12         }
13     }
14 };

```

Figure 2.30: Finding the shortest path between two vertices.

2.12.3 The Wolf, the Goat, and the Cabbage

Next, we present an application of the theory developed so far. We solve a problem from that has puzzled the greatest agricultural economists for centuries. The puzzle we want to solve is known as the **wolf-goat-cabbage puzzle**:

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. 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?

In order to compute a schedule, we first have to model the problem. The various **states** of the problem will be regarded as **vertices** of a graph and this graph will be represented as a binary relation. To this end we define the set

$$\text{All} := \{\text{"farmer"}, \text{"wolf"}, \text{"goat"}, \text{"cabbage"}\}.$$

Every node will be represented as a subset S of the set All . The idea is that the set S specifies those objects that are on the left side of the river. We assume that initially the farmer is on the left side of the river. Therefore, the set of all possible states can be defined as the set

$$P := \{ S : S \text{ in } 2 ** \text{All} \mid \text{!problem}(S) \ \&\& \ \text{!problem}(\text{All} - S) \};$$

Here, we have used the procedure `problem` to check whether a given set `S` has a problem. Note that since `S` is the set of objects on the left side, the expression `All - S` computes the set of objects on the right side of the river.

Next, a set `S` of objects has a problem if both of the following conditions are satisfied:

1. The farmer is not an element of `S` and
2. either `S` contains both the goat and the cabbage or `S` contains both the wolf and the goat.

Therefore, we can implement the function `problem` as follows:

```
problem := procedure(S) {
    return !("farmer" in S)
           ({ "goat", "cabbage" } <= S || { "wolf", "goat" } <= S);
};
```

We proceed to compute the relation `R` that contains all possible transitions between different states. We will compute `R` using the formula:

```
R := R1 + R2;
```

Here `R1` describes the transitions that result from the farmer crossing the river from left to right, while `R2` describes the transitions that result from the farmer crossing the river from right to left. We can define the relation `R1` as follows:

```
R1 := { [S, S - B]: S in P, B in 2 ** S
        | S - B in P && "farmer" in B && #B <= 2
      };
```

Let us explain this definition in detail:

1. Initially, `S` is the set of objects on the left side of the river. Hence, `S` is an element of the set of all states that we have defined as `P`.
2. `B` is the set of objects that are put into the boat and that do cross the river. Of course, for an object to go into the boat it has to be on the left side of the river to begin with. Therefore, `B` is a subset of `S` and hence an element of the power set of `S`.
3. Then `S-B` is the set of objects that are left on the left side of the river after the boat has crossed. Of course, the new state `S-B` has to be a state that does not have a problem. Therefore, we check that `S-B` is an element of `P`.

4. Furthermore, the farmer has to be in the boat. This explains the condition

```
"farmer" in B.
```

5. Finally, the boat can only have two passengers. Therefore, we have added the condition

```
#B <= 2.
```

Next, we have to define the relation `R2`. However, as crossing the river from right to left is just the reverse of crossing the river from left to right, `R2` is just the inverse of `R1`. Hence we define:

```
R2 := { [y, x] : [x, y] in R1 };
```

Finally, the start state has all objects on the left side. Therefore, we have

```
start := All;
```

In the end, all objects have to be on the right side of the river. That means that nothing is left on the left side. Therefore, we define

```
goal := {};
```

Figure 2.31 on page 39 shows the program `wolf-goat-cabbage.stlx` that combines the statements shown so far. The solution computed by this program is shown in Figure 2.32.

```

1  problem := procedure(S) {
2      return !("farmer" in S)                                &&
3          ({ "goat", "cabbage" } <= S || { "wolf", "goat" } <= S);
4  };
5
6  All := { "farmer", "wolf", "goat", "cabbage" };
7  P   := { S : S in 2 ** All | !problem(S) && !problem(All - S) };
8  R1  := { [S, S - B]: S in P, B in 2 ** S
9          | S - B in P && "farmer" in B && #B <= 2
10         };
11  R2  := { [y, x] : [x, y] in R1 };
12  R   := R1 + R2;
13
14  start := All;
15  goal  := {};
16
17  path  := reachable(start, goal, R);

```

Figure 2.31: Solving the wolf-goat-cabbage problem.

```

1  {"cabbage", "farmer", "goat", "wolf"}                                {}
2      >>>> {"farmer", "goat"} >>>>
3  {"cabbage", "wolf"}                                                {"farmer", "goat"}
4      <<<< {"farmer"} <<<<
5  {"cabbage", "farmer", "wolf"}                                    {"goat"}
6      >>>> {"farmer", "wolf"} >>>>
7  {"cabbage"}                                                        {"farmer", "goat", "wolf"}
8      <<<< {"farmer", "goat"} <<<<
9  {"cabbage", "farmer", "goat"}                                    {"wolf"}
10     >>>> {"cabbage", "farmer"} >>>>
11 {"goat"}                                                            {"cabbage", "farmer", "wolf"}
12     <<<< {"farmer"} <<<<
13 {"farmer", "goat"}                                                {"cabbage", "wolf"}
14     >>>> {"farmer", "goat"} >>>>
15 {}                                                                    {"cabbage", "farmer", "goat", "wolf"}

```

Figure 2.32: A schedule for the agricultural economist.

2.13 Terms and Matching

So far we have seen the basic data structures of SETLX like numbers, string, sets, and lists. There is one more data structure that is supported by SETLX. This is the data structure of **terms**. This data structure is especially useful when we develop programs that deal with mathematical formulas. For example, in this section we will develop a program that reads a string like

`"x * exp(x)"`,

interprets this string as describing the real valued function

$$x \mapsto x \cdot \exp(x),$$

and then takes the derivative of this function with respect to the variable x . This program is easy to implement if real valued functions are represented as terms. The reason is that SETLX provides **matching** for terms. We will define this notion later. Matching is one of the main ingredients of the programming language **Prolog**. This programming language was quite popular in artificial intelligence during the eighties and has inspired the matching that is available in SETLX.

2.13.1 Constructing and Manipulating Terms

In order to build terms, we first need **functors**. It is important not to confuse functors with function symbols. Therefore, functors have to be preceded by the character “@”. For example, the following strings can be used as functors:

@f, @FabcXYZ, @sum, @Hugo_.

However, in the expression “@f”, the string “f” is the functor. The character “@” is only used as an escape character that tells us that “f” is not a function symbol but rather a functor. Next, we define **terms**. If F is a functor and t_1, t_2, \dots , are any values, i.e. they could be number, strings, lists, sets, or terms themselves, then

$$@F(t_1, t_2, \dots, t_n)$$

is a term. Syntactically, terms look very similar to function calls. The only difference between a function call and a term is the following:

1. A function call starts with a function symbol.
2. A term starts with a functor.

Examples:

1. @Address("Coblitzallee 1-9", 68163, "Mannheim")

is a term that represents an address.

2. @product(@variable("x"), @exp(@variable("x")))

is a term that represents the function $x \mapsto x \cdot \exp(x)$. ◇

At this point you might ask how terms are evaluated. The answer is that terms **are not evaluated!** Terms are used to represent data in a way that is both concise and readable. Hence, terms are values like numbers, sets or strings. As terms are values, they don't need to be evaluated.

Let us demonstrate a very simple application of terms. Imagine that SETLX wouldn't provide lists as a native data type. Then, we could implement lists via terms. First, we would use a functor to represent the empty list. Let us choose the functor **nil** for this purpose. Hence, we have

$$@nil() \hat{=} [],$$

where we read the symbol “ $\hat{=}$ ” as “corresponds to”. Note that the parentheses after the functor **nil** are **necessary!** Next, in order to represent a list with first element x and a list r of remaining elements we use the functor **cons**. Then we have the correspondence

$$@cons(x, r) \hat{=} [x] + r.$$

Concretely, the list $[1, 2, 3]$ is represented as the term

$$@cons(1, @cons(2, @cons(3, @nil()))).$$

The programming language *Prolog* represents lists internally in a similar form.

SETLX provides two functions that allow us to extract the components of a term. Furthermore, there is a function for constructing terms. These functions are described next.

1. The function `fct` returns the functor of a given term. If t is a term of the form $@F(s_1, \dots, s_n)$, then the result returned by the expression

`fct(@F(s1, ..., sn))`

is the functor F of this term. For example the expression

`fct(@cons(1, @cons(2, @cons(3, @nil()))))`

returns the string "cons" as its result.

2. The function `args` returns the arguments of a term. If t is a term of the form $@F(s_1, \dots, s_n)$, then

`args(@F(s1, ..., sn))`

returns the list $[s_1, \dots, s_n]$. For example, the expression

`args(@cons(1, @cons(2, @cons(3, @nil()))))`

is evaluated as

`[1, @cons(2, @cons(3, @nil()))]`.

3. If f is the name of a functor and l is a list, then the function `makeTerm` can be invoked as

`t := makeTerm(f, l).`

This expression generates a term t such that f is the functor and l is the list of its arguments. Therefore we have

`fct(t) = f und args(t) = l.`

For example, the expression

`makeTerm("cons", [1, @nil()])`

returns the result

`@cons(1, @nil()).`

```

1  append := procedure(l, x) {
2      if (fct(l) == "nil") {
3          return @cons(x, @nil());
4      }
5      [head, tail] := args(l);
6      return @cons(head, append(tail, x));
7  };
8  l := @cons(1, @cons(2, @cons(3, @nil()))); // corresponds to [1,2,3]
9  print(append(l, 4));

```

Figure 2.33: Appending an element at the end of a list.

Figure 2.33 on page 41 shows the program `append.stlx`. This program implements the function `append`. As its first arguments, this function takes a list `l` that is represented as a term. As its second argument, it takes an object `x`. The purpose of the expression

`append(l, x)`

is to append the object `x` at the end of the list `l`. The implementation of the function `append` assumes that the list `l` is represented as a term using the functors "cons" and "nil".

1. Line 2 checks whether the list `l` is empty. The list `l` is empty iff we have `l = @nil()`. In the program we merely check the functor of the term `l`. If the name of this functor is "`nil`", then `l` is the empty list.
2. If `l` is not empty, then it must be a term of the form

$$l = @cons(head, tail).$$

Then, conceptually `head` is the first element of the list `l` and `tail` is the list of the remaining elements. In this case, we need to recursively append `t` at the end of the list `tail`. Finally, the first element of the list `l`, which is called `head` in line 5, needs to be prepended to the list that is returned from the recursive invocation of `append`. This is done in line 6 by constructing the term

$$@cons(head, append(tail, x)).$$

2.13.2 Matching

It would be quite tedious if the functions `fct` and `args` were the only means to extract the components of a term. Figure 2.34 on page 42 shows the program `append-match.stlx`, that uses `matching` to implement the function `append`. Line 3 checks, whether the list `l` is empty, i.e. whether `l` is identical to the term `@nil()`. Line 4 is more interesting, as it combines two actions.

1. It checks, whether the list `l` is a term that starts with the functor `cons`.
2. If `l` does indeed starts with the functor `cons`, the arguments of this functor are extracted and assigned to the variables `head` and `tail`.

Hence, if the `match` statement in line 4 is successful, the equation

$$l = @cons(head, tail)$$

holds afterwards.

```

1  append := procedure(l, x) {
2      match (l) {
3          case @nil():          return @cons(x, @nil());
4          case @cons(head, tail): return @cons(head, append(tail, x));
5      }
6  };

```

Figure 2.34: Implementing `append` using a `match` statement.

In general, a `match` statement has the structure that is shown in Figure 2.35. Here, e is any expression that yields a term when evaluated. The expressions t_1, \dots, t_n are so called `patterns` that contain variables. When the `match` statement is executed, SETLX tries to bind the variables occurring in the pattern t_1 such that the resulting expression is equal to e . If this succeeds, the statements in $body_1$ are executed and the execution of the `match` statement ends. Otherwise, the patterns t_2, \dots, t_n are tried one by one. If the pattern t_i is successfully matched to e , the statements in $body_i$ are executed and the execution of the `match` statement ends. If none of the patterns t_1, \dots, t_n can be matched with e , the statements in $body_{n+1}$ are executed.

We close this section by showing an example that demonstrates the power of matching. The function `diff` that is shown in Figure 2.36 on page 43 is part of the program `diff`. This function is called with two arguments.

1. The first argument `t` is a term that represents an arithmetical expression.
2. The second argument `x` is a term that represents a variable.

```

1  match (e) {
2      case  $t_1$  : body1
3          :
4      case  $t_n$  : bodyn
5      default: bodyn+1
6  }
```

Figure 2.35: Struktur eines Match-Blocks

```

1  loadLibrary("termUtilities");
2
3  diff := procedure(t, x) {
4      match (t) {
5          case a + b :
6              return diff(a, x) + diff(b, x);
7          case a - b :
8              return diff(a, x) - diff(b, x);
9          case a * b :
10             return diff(a, x) * b + a * diff(b, x);
11         case a / b :
12             return ( diff(a, x) * b - a * diff(b, x) ) / b * b;
13         case a ** b :
14             return diff( @exp(b * @ln(a)), x);
15         case @ln(a) :
16             return diff(a, x) / a;
17         case @exp(a) :
18             return diff(a, x) * @exp(a);
19         case v | v == x :
20             return 1;
21         case y | isVariable(y) : // must be different from x
22             return 0;
23         case n | isNumber(n):
24             return 0;
25     }
26 };
27 test := procedure(s) {
28     t := parseTerm(s);
29     v := parseTerm("x");
30     d := diff(t, v);
31     print("d/dx($s$) = $d$\n");
32 };
33 test("x ** x");
```

Figure 2.36: A function to perform symbolic differentiation.

The function `diff` interprets its argument `t` as a function of the variable `x`. We take the **derivative** of this function with respect to the variable `x`. For example, in order to compute the derivative of the function

$$x \mapsto x^x,$$

we can call the function `diff` as follows:

```
diff(parseTerm("x ** x"), parseTerm("x"));
```

Here, the function `parseTerm` is a function that is defined in the library `termUtilities`. This function takes a string as input and converts this string into a term. In order to use the function `parseTerm`, we have to load the library that defines it. This happens in line 1 of Figure 2.36.

Let us now discuss the implementation of the function `diff` in more detail.

1. Line 5 makes use of the fact that the operator “+” can be applied to terms. The result is a term that has the functor “@@@sum”. However, this functor is hidden from the user and becomes only visible when we use the function `fct` to expose it. For example, we can define a term `t` as follows:

```
t := @f(1) + @g(2);
```

Then `t` is a term that is displayed as “@f(1) + @g(2)”, but the expression `fct(t)` returns the string “@@@sum”.

There is no need to remember that the internal representation of the operator “+” as a functor is given as the string “@@@sum”. The only thing that you have to keep in mind is the fact, that the operator “+” can be applied to terms. The same is true for the other arithmetical operators “+”, “-”, “*”, “/”, “%”, and “**”. Similarly, the logical operators “&&”, “|”, “!”, “=>”, and “<==>” can be used as functors. Note, however, that the relational operators “<”, “>”, “<=”, “>=”, “can not be used” to combine terms. Finally, the operators “==” and “!=” can be used to check whether two terms are identical or different, respectively. Hence, while these operators can be applied to terms, they return a Boolean value, not a term!

As the operator “+” can be used as a functor, it can also be used in a pattern. The pattern

```
a + b
```

matches any term that can be written as a sum. The derivative of a sum is computed by summing the derivatives of the components of the sum, i.e. we have

$$\frac{d}{dx}(f(x) + g(x)) = \frac{d}{dx}f(x) + \frac{d}{dx}g(x).$$

Therefore, the case where the term `t` has the form `a + b` can be dealt with by recursively computing the derivatives of `a` and `b` and adding them. This happens in line 6.

2. Line 7 deals with the case where `t` is a difference. Mathematically, the rule to take the derivative of a difference is

$$\frac{d}{dx}(f(x) - g(x)) = \frac{d}{dx}f(x) - \frac{d}{dx}g(x).$$

This rule is implemented in line 8.

3. Line 9 deals with the case where `t` is a product. The **product rule** is

$$\frac{d}{dx}(f(x) \cdot g(x)) = \left(\frac{d}{dx}f(x)\right) \cdot g(x) + f(x) \cdot \left(\frac{d}{dx}g(x)\right).$$

This rule is implemented in line 10.

4. Line 11 deals with the case where `t` is a quotient. The **quotient rule** is

$$\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{\left(\frac{d}{dx}f(x)\right) \cdot g(x) - f(x) \cdot \left(\frac{d}{dx}g(x)\right)}{g(x) \cdot g(x)}.$$

This rule is implemented in line 12.

5. Line 13 deals with the case where `t` is a power. Now in order to take the derivative of an expression of the form

$$f(x)^{g(x)}$$

we first need to rewrite it using the following trick:

$$f(x)^{g(x)} = \exp(\ln(f(x)^{g(x)})) = \exp(g(x) \cdot \ln(f(x))),$$

Then, we can recursively call **diff** for this expression. This works, because the function **diff** can deal with both the exponential function $x \mapsto \exp(x)$ and with the natural logarithm $x \mapsto \ln(x)$. This rewriting is done in line 14.

6. Line 15 deals with the case where **t** has the form $\ln(f(x))$. In order to take the derivative of this expression, we first need to know the derivative of the natural logarithm. This derivative is given as

$$\frac{d}{dx} \ln(x) = \frac{1}{x}.$$

Then, using the **chain rule** we have that

$$\frac{d}{dx} \ln(f(x)) = \frac{\frac{d}{dx} f(x)}{f(x)}.$$

This rule is used in line 16.

7. Line 17 deals with the case where **t** has the form $\exp(f(x))$. In order to take the derivative of this expression, we first need to know the derivative of the **exponential function**. This derivative is given as

$$\frac{d}{dx} \exp(x) = \exp(x).$$

Then, using the **chain rule** we have that

$$\frac{d}{dx} \exp(f(x)) = \left(\frac{d}{dx} f(x) \right) \cdot \exp(f(x)).$$

This rule is used in line 18.

8. Line 19 deals with the case where **t** is a variable and happens to be the same variable as **x**. As we have

$$\frac{dx}{dx} = 1,$$

the function **diff** returns 1 in this case.

9. Line 21 deals with the case where **t** is a variable. As line 19 has already covered the case that **t** and **x** are the same variable, in this case the variable **x** must be different from **t**. Therefore, with respect to **x** the term **t** can be seen as a constant and the derivative is 0.

10. Line 23 covers the case where **t** is a number. Note how we call **isNumber** after the condition operator “|”. As a number is a constant, the derivative is 0.

11. Line 27 defines the procedure **test**. This procedure takes a string **s** and transforms it into the term **t** via the function **parseTerm** defined in the library **termUtilities**. Similarly, the string “**x**” is transformed into the term **v** that represents this variable.⁵ Line 30 call the function **diff** using the term **t** and the variable **v** as arguments. The resulting term is printed in line 31.

12. Line 33 shows how the function **test** can be called to compute the derivative $\frac{d}{dx} x^x$.

⁵Internally, this variable is represented as the term “@@@variable(“x”)”.

2.14 Outlook

This introductory chapter covers only a small part of the programming language SETLX. There are some additional features of SETLX that will be discussed in the following chapters as we need them. Furthermore, SETLX is discussed in depth in the tutorial that can be found at the following address:

<http://download.randoom.org/setlX//tutorial.pdf>

Remark: Most of the algorithm that were presented in this chapter are not very efficient. The main purpose of these algorithms is to serve as examples that were presented for two reasons:

1. My first intention was to make the abstract notions introduced in set theory more accessible. For example, the program to compute the transitive closure serves to illustrate both the notion of the relational product and the transitive closure. Furthermore, it shows how these notions are useful in solving real world problems.
2. Second, these programs serve to introduce the programming language SETLX.

Later, the lecture on [algorithms](#) will show how to develop efficient algorithms that are more efficient.

Chapter 3

Grenzen der Berechenbarkeit

In jeder Disziplin der Wissenschaft wird die Frage gestellt, welche Grenzen die verwendeten Methoden haben. Wir wollen daher in diesem Kapitel beispielhaft ein Problem untersuchen, bei dem die Informatik an ihre Grenzen stößt. Es handelt sich um das **Halte-Problem**.

3.1 Das Halte-Problem

Das Halte-Problem ist die Frage, ob eine gegebene Funktion für eine bestimmte Eingabe terminiert. Bevor wir formal beweisen, dass das Halte-Problem im Allgemeinen unlösbar ist, wollen wir versuchen, anschaulich zu verstehen, warum dieses Problem schwer sein muss. Dieser informalen Betrachtung des Halte-Problems ist der nächste Abschnitt gewidmet. Im Anschluss an diesen Abschluss zeigen wir dann die Unlösbarkeit des Halte-Problems.

3.1.1 Informale Betrachtungen zum Halte-Problem

Um zu verstehen, warum das Halte-Problem schwer ist, betrachten wir das in Abbildung 3.1 gezeigte Programm. Dieses Programm ist dazu gedacht, die *Legendresche Vermutung* zu überprüfen. Der französische Mathematiker **Adrien-Marie Legendre** (1752 — 1833) hatte vor etwa 200 Jahren die Vermutung ausgesprochen, dass zwischen zwei positiven Quadratzahlen immer eine Primzahl liegt. Die Frage, ob diese Vermutung richtig ist, ist auch heute noch unbeantwortet. Die in Abbildung 3.1 definierte Funktion `legendre(n)` überprüft für eine gegebene positive natürliche Zahl n , ob zwischen n^2 und $(n+1)^2$ eine Primzahl liegt. Falls dies, wie von Legendre vorhergesagt, der Fall ist, gibt die Funktion als Ergebnis `true` zurück, andernfalls wird `false` zurückgegeben.

Abbildung 3.1 enthält darüber hinaus die Definition der Funktion `findCounterExample(n)`, die versucht, für eine gegebene positive natürliche Zahl n eine Zahl $k \geq n$ zu finden, so dass zwischen k^2 und $(k+1)^2$ keine Primzahl liegt. Die Idee bei der Implementierung dieser Funktion ist einfach: Zunächst überprüfen wir durch den Aufruf `legendre(n)`, ob zwischen n^2 und $(n+1)^2$ eine Primzahl ist. Falls dies der Fall ist, untersuchen wir anschließend das Intervall von $(n+1)^2$ bis $(n+2)^2$, dann das Intervall von $(n+2)^2$ bis $(n+3)^2$ und so weiter, bis wir schließlich eine Zahl m finden, so dass zwischen m^2 und $(m+1)^2$ keine Primzahl liegt. Falls Legendre Recht hatte, werden wir nie ein solches k finden und in diesem Fall wird der Aufruf `findCounterExample(1)` nicht terminieren.

Nehmen wir nun an, wir hätten ein schlaues Programm, nennen wir es `stops`, das als Eingabe eine SETLX Funktion f und ein Argument a verarbeitet und das uns die Frage, ob die Berechnung von $f(a)$ terminiert, beantworten kann. Die Idee wäre, dass für die Funktion `stops` Folgendes gilt:

$$\text{stops}(f, a) = 1 \quad \text{g.d.w.} \quad \text{der Aufruf } f(a) \text{ terminiert.}$$

Falls der Aufruf $f(a)$ nicht terminiert, sollte stattdessen `stops(f, a) = 0` gelten. Wenn wir eine solche Funktion

```

1  findCounterExample := procedure(n) {
2      legendre := procedure(n) {
3          k := n * n + 1;
4          while (k < (n + 1) ** 2) {
5              if (isPrime(k)) {
6                  print("$n**2 < $k < $(n+1)**2");
7                  return true;
8              }
9              k += 1;
10         }
11         return false;
12     };
13     while (true) {
14         if (legendre(n)) {
15             n := n + 1;
16         } else {
17             print("Legendre was wrong, no prime between $n**2$ and $(n+1)**2$!");
18             return;
19         }
20     }
21 };

```

Figure 3.1: Eine Funktion zur Überprüfung der Vermutung von Legendre.

stopps hätten, dann könnten wir

```
stops(findCounterExample, 1)
```

aufrufen und wüssten anschließend, ob die Vermutung von Legendre wahr ist oder nicht: Wenn

```
stops(findCounterExample, 1) = 1
```

ist, dann würde das heißen, dass der Funktions-Aufruf `findCounterExample(1)` terminiert. Das passiert aber nur dann, wenn ein Gegenbeispiel gefunden wird. Würde der Aufruf

```
stops(findCounterExample, 1)
```

stattdessen eine 0 zurück liefern, so könnten wir schließen, dass der Aufruf `findCounterExample(1)` nicht terminiert. Mithin würde die Funktion `findCounterExample` kein Gegenbeispiel finden und das würde heißen, dass die Vermutung von Legendre wahr ist.

Es gibt eine Reihe weiterer offener mathematischer Probleme, die alle auf die Frage abgebildet werden können, ob eine gegebene Funktion terminiert. Daher zeigen die vorhergehenden Überlegungen, dass es sehr nützlich wäre, eine Funktion wie `stops` zur Verfügung zu haben. Andererseits können wir an dieser Stelle schon ahnen, dass die Implementierung der Funktion `stops` nicht ganz einfach sein kann.

3.1.2 Formale Analyse des Halte-Problems

Wir werden in diesem Abschnitt beweisen, dass das Halte-Problem unlösbar ist. Dazu führen wir den Begriff einer Test-Funktion ein.

Definition 1 (Test-Funktion) Ein String t ist genau dann eine *Test-Funktion*, wenn t die Form

```
procedure(x) { ... }
```

hat und sich als SETLX-Funktion parsen lässt. Die Menge der Test-Funktionen bezeichnen wir mit TF . \diamond

Beispiele:

1. $s_1 = \text{"procedure}(x) \{ \text{return } 0; \}$
 s_1 ist eine (sehr einfache) Test-Funktion.
2. $s_2 = \text{"procedure}(x) \{ \text{while } (\text{true}) \{ x := x + 1; \} \}$
 s_2 ist ebenfalls eine Test-Funktion. Offenbar liefert diese Test-Funktion nie ein Ergebnis, aber fur die Frage, ob s_2 eine Test-Funktion ist oder nicht, ist dies irrelevant.
3. $s_3 = \text{"procedure}(x) \{ \text{return } ++x; \}$
 s_3 ist keine Test-Funktion, denn da SETLX den Prefix-Operator “++” nicht unterstutzt, lasst sich der String s_3 nicht fehlerfrei parsen.
4. $s_4 = \text{"procedure}(x, y) \{ \text{return } x + y; \}$
 s_4 ist keine Test-Funktion, denn ein String ist nur dann eine Test-Funktion, wenn die Funktion mit genau einem Parameter aufgerufen wird.

Um das Halte-Problem ubersichtlicher formulieren zu konnen, fuhren wir noch drei zusatzliche Notationen ein.

Notation 2 (\rightsquigarrow , \downarrow , \uparrow) Ist t eine SETLX-Funktion, die k Argumente verarbeitet und sind a_1, \dots, a_k Argumente, so schreiben wir

$$t(a_1, \dots, a_k) \rightsquigarrow r$$

wenn der Aufruf $t(a_1, \dots, a_k)$ das Ergebnis r liefert. Sind wir an dem Ergebnis selbst nicht interessiert, sondern wollen nur angeben, dass ein Ergebnis existiert, so schreiben wir

$$t(a_1, \dots, a_k) \downarrow$$

und sagen, dass der Aufruf $t(a_1, \dots, a_k)$ **terminiert**. Terminiert der Aufruf $t(a_1, \dots, a_k)$ nicht, so schreiben wir

$$t(a_1, \dots, a_k) \uparrow$$

und sagen, dass der Aufruf $t(a_1, \dots, a_k)$ **divergiert**. □

Beispiele: Legen wir die Funktions-Definitionen zugrunde, die wir im Anschluss an die Definition des Begriffs der Test-Funktion gegeben haben, so gilt:

1. $\text{procedure}(x) \{ \text{return } 0; \}(\text{"emil"}) \rightsquigarrow 0$
2. $\text{procedure}(x) \{ \text{return } 0; \}(\text{"emil"}) \downarrow$
3. $\text{procedure}(x) \{ \text{while } (\text{true}) \{ x := x + 1; \} \}(\text{"hugo"}) \uparrow$

Das *Halte-Problem* fur SETLX-Funktionen ist die Frage, ob es eine SetlX-Funktion

$$\text{stops} := \text{procedure}(t, a) \{ \dots \}$$

gibt, die als Eingabe eine Testfunktion t und einen String a erhalt und die folgende Eigenschaft hat:

1. $t \notin TF \Leftrightarrow \text{stops}(t, a) \rightsquigarrow 2$.

Der Aufruf $\text{stops}(t, a)$ liefert genau dann den Wert 2 zuruck, wenn t keine Test-Funktion ist.

2. $t \in TF \wedge t(a) \downarrow \Leftrightarrow \text{stops}(t, a) \rightsquigarrow 1$.

Der Aufruf $\text{stops}(t, a)$ liefert genau dann den Wert 1 zuruck, wenn t eine Test-Funktion ist und der Aufruf $t(a)$ terminiert.

$$3. \ t \in TF \wedge t(a) \uparrow \Leftrightarrow \text{stops}(t, a) \rightsquigarrow 0.$$

Der Aufruf `stops(t, a)` liefert genau dann den Wert 0 zurück, wenn t eine Test-Funktion ist und der Aufruf $t(a)$ nicht terminiert.

Falls eine SETLX-Funktion `stops` mit den obigen Eigenschaften existiert, dann sagen wir, dass das Halte-Problem für SETLX entscheidbar ist.

Theorem 3 (Alan Turing, 1936) Das Halte-Problem ist unentscheidbar.

Beweis: Zunächst eine Vorbemerkung. Um die Unentscheidbarkeit des Halte-Problems nachzuweisen, müssen wir zeigen, dass etwas, nämlich eine Funktion mit gewissen Eigenschaften nicht existiert. Wie kann so ein Beweis überhaupt funktionieren? Wie können wir überhaupt zeigen, dass irgendetwas nicht existiert? Die einzige Möglichkeit zu zeigen, dass etwas nicht existiert ist indirekt: Wir nehmen also an, dass eine Funktion `stops` existiert, die das Halte-Problem löst. Aus dieser Annahme werden wir einen Widerspruch ableiten. Dieser Widerspruch zeigt uns dann, dass eine Funktion `stops` mit den gewünschten Eigenschaften nicht existieren kann. Um zu einem Widerspruch zu kommen, definieren wir den String `turing` wie in Abbildung 3.2 gezeigt.

```

1  turing := "procedure(x) {
2      result := stops(x, x);
3      if (result == 1) {
4          while (true) {
5              print("... looping ...");
6          }
7      }
8      return result;
9  }"
```

Figure 3.2: Die Definition des Strings `turing`.

Mit dieser Definition ist klar, dass `turing` eine Test-Funktion ist:

$$\text{turing} \in TF.$$

Damit sind wir in der Lage, den String `turing` als Eingabe der Funktion `stops` zu verwenden. Wir betrachten nun den folgenden Aufruf:

`stops(turing, turing);`

Da `turing` eine Test-Funktion ist, können nur zwei Fälle auftreten:

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

Diese beiden Fälle analysieren wir nun im Detail:

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

Nach der Spezifikation von `stops` bedeutet dies

$$\text{turing}(\text{turing}) \uparrow$$

Schauen wir nun, was wirklich beim Aufruf `turing(turing)` passiert: In Zeile 2 erhält die Variable `result` den Wert 0 zugewiesen. In Zeile 3 wird dann getestet, ob `result` den Wert 1 hat. Dieser Test schlägt fehl. Daher wird der Block der `if`-Anweisung nicht ausgeführt und die Funktion liefert als nächstes in Zeile 8 den Wert 0 zurück. Insbesondere terminiert der Aufruf also, im Widerspruch zu dem, was die Funktion `stops` behauptet hat. ζ

Damit ist der erste Fall ausgeschlossen.

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

Aus der Spezifikation der Funktion **stops** folgt, dass der Aufruf **turing(turing)** terminiert:

turing(turing) ↓

Schauen wir nun, was wirklich beim Aufruf **turing(turing)** passiert: In Zeile 2 erh  lt die Variable **result** den Wert 1 zugewiesen. In Zeile 3 wird dann getestet, ob **result** den Wert 1 hat. Diesmal gelingt der Test. Daher wird der Block der **if**-Anweisung ausgef  hrt. Dieser Block besteht aber nur aus einer Endlos-Schleife, aus der wir nie wieder zur  ck kommen. Das steht im Widerspruch zu dem, was die Funktion **stops** behauptet hat.   

Damit ist der zweite Fall ausgeschlossen.

Insgesamt haben wir also in jedem Fall einen Widerspruch erhalten. Damit muss die Annahme, dass die SETLX-Funktion **stops** das Halte-Problem l  st, falsch sein, denn diese Annahme ist die Ursache f  r die Widerspr  che, die wir erhalten haben. Insgesamt haben wir daher gezeigt, dass es keine SETLX-Funktion geben kann, die das Halte-Problem l  st.   

Bemerkung: Der Nachweis, dass das Halte-Problem unl  sbar ist, wurde 1936 von Alan Turing (1912 – 1954) [Tur36] erbracht. Turing hat das Problem damals nat  rlich nicht f  r die Sprache SETLX gel  st, sondern f  r die heute nach ihm benannten *Turing-Maschinen*. Eine Turing-Maschine ist abstrakt gesehen nichts anderes als eine Beschreibung eines Algorithmus. Turing hat also gezeigt, dass es keinen Algorithmus gibt, der entscheiden kann, ob ein gegebener anderer Algorithmus terminiert.

Bemerkung: An dieser Stelle k  nnen wir uns fragen, ob es vielleicht eine andere Programmier-Sprache gibt, in der wir das Halte-Problem dann vielleicht doch l  sen k  nnten. Wenn es in dieser Programmier-Sprache Prozeduren, **if**-Verzweigungen und **while**-Schleifen gibt, und wenn wir dort Programm-Texte als Argumente von Funktionen   bergeben k  nnen, dann ist leicht zu sehen, dass der obige Beweis der Unl  sbarkeit des Halte-Problems sich durch geeignete syntaktische Modifikationen auch auf die andere Programmier-Sprache   bertragen l  sst.

3.2 Unl  sbarkeit des   quivalenz-Problems

Es gibt noch eine ganze Reihe anderer Funktionen, die nicht berechenbar sind. In der Regel werden wir den Nachweis, dass eine bestimmte Funktion nicht berechenbar ist, indirekt f  hren und annehmen, dass die gesuchte Funktion doch berechenbar ist. Unter dieser Annahme konstruieren wir dann eine Funktion, die das Halte-Problem l  st, was im Widerspruch zu der Unl  sbarkeit des Halte-Problems steht. Dieser Widerspruch zwingt uns zu der Folgerung, dass die gesuchte Funktion nicht berechenbar ist. Wir werden dieses Verfahren an einem Beispiel demonstrieren. Vorweg ben  tigen wir aber noch eine Definition.

Definition 4 (\simeq) Es seien t_1 und t_2 zwei SETLX-Funktionen und a_1, \dots, a_k seien Argumente, mit denen wir diese Funktionen f  ttern k  nnen. Wir definieren

$$t_1(a_1, \dots, a_k) \simeq t_2(a_1, \dots, a_k)$$

g.d.w. einer der beiden folgen F  lle auftritt:

$$1. \ t_1(a_1, \dots, a_k) \uparrow \ \wedge \ t_2(a_1, \dots, a_k) \uparrow,$$

beide Funktionen divergieren also f  r die gegebenen Argumente.

$$2. \ \exists r : \left(t_1(a_1, \dots, a_k) \rightsquigarrow r \ \wedge \ t_2(a_1, \dots, a_k) \rightsquigarrow r \right),$$

die Funktionen liefern also f  r die gegebenen Argumente das gleiche Ergebnis.

In diesem Fall sagen wir, dass die beiden Funktions-Aufrufe $t_1(a_1, \dots, a_k) \simeq t_2(a_1, \dots, a_k)$ *partiell   quivalent* sind.   

Wir kommen jetzt zum * quivalenz-Problem*. Die Funktion **equal**, die die Form

equal := procedure(p1, p2, a) { ... }

hat, m ge folgender Spezifikation gen igen:

$$1. \ p_1 \notin TF \vee p_2 \notin TF \Leftrightarrow \text{equal}(p_1, p_2, a) \rightsquigarrow 2.$$

2. Falls

- (a) $p_1 \in TF$,
- (b) $p_2 \in TF$ und
- (c) $p_1(a) \simeq p_2(a)$

gilt, dann muss gelten:

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

3. Ansonsten gilt

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

Wir sagen, dass eine Funktion, die der eben angegebenen Spezifikation gen igt, das * quivalenz-Problem* l st.

Theorem 5 (Rice, 1953) *Das  quivalenz-Problem ist unl sbar.*

Beweis: Wir f hren den Beweis indirekt und nehmen an, dass es doch eine Implementierung der Funktion **equal** gibt, die das  quivalenz-Problem l st. Wir betrachten die in Abbildung 3.3 angegebenen Implementierung der Funktion **stops**.

```

1  stops := procedure(p, a) {
2      f := "procedure(x) { while (true) { x := x + x; } }";
3      e := equal(f, p, a);
4      if (e == 2) {
5          return 2;
6      } else {
7          return 1 - e;
8      }
9  };

```

Figure 3.3: Eine Implementierung der Funktion **stops**.

Zu beachten ist, dass in Zeile 3 die Funktion **equal** mit einem String aufgerufen wird, der eine Test-Funktion ist. Diese Test-Funktion hat die folgende Form:

procedure(x) { while (true) { x := x + x; } };

Es ist offensichtlich, dass diese Funktion f r kein Ergebnis terminiert. Ist also das Argument p eine Test-Funktion, so liefert die Funktion **equal** immer dann den Wert 1, wenn $p(a)$ nicht terminiert, andernfalls muss sie den Wert 0 zur ck geben. Damit liefert die Funktion **stops** aber f r eine Test-Funktion p und ein Argument a genau dann 1, wenn der Aufruf $p(a)$ terminiert und w rde folglich das Halte-Problem l sen. Das kann nicht sein, also kann es keine Funktion **equal** geben, die das  quivalenz-Problem l st. \square

Die Unl sbarkeit des  quivalenz-Problems und vieler weiterer praktisch interessanter Probleme folgen aus dem 1953 von Henry G. Rice [Ric53] bewiesenen **Satz von Rice**.

Chapter 4

Aussagenlogik

4.1 Überblick

Die Aussagenlogik beschäftigt sich mit der Verknüpfung einfacher Aussagen durch *Junktoren*. Dabei sind Junktoren Worte wie “und”, “oder”, “nicht”, “wenn \dots , dann”, und “genau dann, wenn”. Einfache Aussagen sind dabei Sätze, die

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

Beispiele für einfache Aussagen sind

1. “Die Sonne scheint.”
2. “Es regnet.”
3. “Am Himmel ist ein Regenbogen.”

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

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

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

Um die Struktur komplexerer Aussagen übersichtlich werden zu lassen, führen wir in der Aussagenlogik zunächst sogenannte *Aussage-Variablen* ein. Diese stehen für atomare Aussagen. Zusätzlich führen wir für die Junktoren “nicht”, “und”, “oder”, “wenn, \dots dann”, und “genau dann, wenn” die folgenden Abkürzungen ein:

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

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

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

schreiben. Aus den Aussagen

1. SonneScheint
2. EsRegnet
3. SonneScheint \wedge EsRegnet \rightarrow Regenbogen

folgt *logisch* die Aussage

Regenbogen.

Diesen Beweis können wir übersichtlicher wie folgt angeben:

$$\frac{\text{SonneScheint} \quad \text{EsRegnet} \quad \text{SonneScheint} \wedge \text{EsRegnet} \rightarrow \text{Regenbogen}}{\text{Regenbogen}}$$

Die Aussagen über dem Bruchstrich bezeichnen wir als *Prämissen*, die Aussage unter dem Bruchstrich ist die *Konklusion*. Statt Beweis-Prinzip sagen wir oft auch *Schluss-Regel*.

Wir stellen fest, dass die obige Schluss-Regel unabhängig von dem Wahrheitswert der Aussagen in dem folgenden Sinne gültig ist: Wenn alle Prämissen gültig sind, dann folgt aus rein logischen Gründen auch die Gültigkeit der Konklusion. Um dieses weiter formalisieren zu können, ersetzen wir die Aussage-Variablen **SonneScheint**, **EsRegnet** und **Regenbogen** durch die *Meta-Variablen* p , q und r , die für beliebige aussagenlogische Formeln stehen. Die obige Schluss-Regel ist dann eine Instanz der folgenden allgemeinen Schluss-Regel:

$$\frac{p \quad q \quad p \wedge q \rightarrow r}{r}$$

Aufgabe 1: Formalisieren Sie die Schluss-Regel, die in dem folgenden Argument verwendet wird.

Wenn es regnet, ist die Straße nass. Es regnet nicht.
Also ist die Straße nicht nass. \diamond

Lösung: Es wird die folgende Schluss-Regel verwendet:

$$\frac{p \rightarrow q \quad \neg p}{\neg q}$$

Diese Schluss-Regel ist nicht logisch korrekt. Wenn Sie das nicht einsehen, sollten Sie bei strahlendem Sonnenschein einen Eimer Wasser auf die Straße kippen. \square

Dadurch, dass wir ausgehend von Beobachtungen und als wahr erkannten Tatsachen und Zusammenhängen mehrere *logische Schlüsse* aneinander fügen, erhalten wir einen *Beweis*. Die als wahr erkannten Tatsachen und Beobachtungen bezeichnet wir dabei als *Axiome*. Wir verwenden in diesem Zusammenhang die folgende Notation:

$$M \vdash r.$$

Hierbei gilt:

- M ist eine Menge von Aussagen.
- \vdash bezeichnet ein System von Schluss-Regeln. Ein solches System bezeichnen wir auch als *Kalkül*.
- r ist eine Aussage.

Die Schreibweise $M \vdash r$ wäre dann als

“Aus den Axiomen der Menge M kann die Aussage r hergeleitet werden”

zu interpretieren. Wir lesen $M \vdash r$ als “ M leitet r her”. Damit ist gemeint, dass wir ausgehend von den Axiomen in M durch sukzessives Anwenden verschiedener Schluss-Regeln die Aussage r beweisen können. Das Zeichen \vdash symbolisiert dabei den *Herleitungs-Begriff*, den wir auch als *Kalkül* bezeichnen. Wir werden in einem späteren Abschnitt den Herleitungs-Begriff formal definieren. Für den Moment können Sie einfach annehmen, dass ein Kalkül nichts anderes als ein System von *Spielregeln* ist, mit dem wir aus gegebenen Formeln rein mechanisch neue Formeln herleiten können, ohne dass wir dabei die Formeln im Detail verstehen müssen. Durch den Umstand, dass sich die Spielregeln rein schematisch anwenden lassen, ist es dann möglich einen solchen Kalkül durch ein Programm zu implementieren.

Parallel zu dem Herleitungs-Begriff gibt es auch den sogenannten *Folgerungs-Begriff*. Wir schreiben

$$M \models r,$$

wenn die Aussage r logisch aus den Aussagen M folgt. Der Folgerungs-Begriff ist ein inhaltlicher Begriff, wir sprechen auch von einem *semantischen* Begriff, denn beim Folgerungs-Begriff geht es um die *Bedeutung* der Formeln. Die Notation $M \models r$ wird gelesen als “ r folgt aus M ”. Das können wir anders auch so formulieren: Immer wenn alle Aussagen aus M wahr sind, dann ist auch die Aussage r wahr. Wir können den Begriff der *logischen Folgerung* aber erst dann präzise definieren, wenn wir die Semantik der Junktoren mathematisch festgelegt haben.

Ziel der Aussagenlogik ist es, einen Herleitungs-begriff zu finden, der die folgenden beiden Bedingungen erfüllt:

1. Der Herleitungs-begriff sollte *korrekt* sein, es sollte also nicht möglich sein, Unsinn zu beweisen. Es sollte also gelten:

$$\text{Aus } M \vdash r \text{ folgt } M \models r.$$

Wenn wir die Aussage r aus den Axiomen der Menge M herleiten können, dann soll r auch aus M folgen.

2. Der Herleitungs-begriff sollte *vollständig* sein, d.h. wenn eine Aussage r aus einer Menge von anderen Aussagen M logisch folgt, dann sollte sie auch aus M herleitbar sein:

$$\text{Aus } M \models r \text{ folgt } M \vdash r.$$

Wenn die Aussage r aus M folgt, dann soll r auch aus der Menge M hergeleitet werden können.

Bestimmte aussagenlogische Formeln sind offenbar immer wahr, egal was wir für die einzelnen Teilaussagen einsetzen. Beispielsweise ist eine Formel der Art

$$p \vee \neg p$$

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

$$p \wedge \neg p$$

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

4.2 Anwendungen der Aussagenlogik

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

1. Analyse und Design digitaler Schaltungen.

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

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

Eine ganz konkrete Anwendung ist der Schaltungs-Vergleich. Hier werden zwei digitale Schaltungen als aussagenlogische Formeln dargestellt. Anschließend wird versucht, mit aussagenlogischen Mitteln die Äquivalenz dieser Formeln zu zeigen. Software-Werkzeuge, die für die Verifikation digitaler Schaltungen eingesetzt werden, kosten heutzutage über 100 000 \$². Dies zeigt die wirtschaftliche Bedeutung der Aussagenlogik.

2. Erstellung von Einsatzplänen (*crew scheduling*).

International tätige Fluggesellschaften müssen bei der Einteilung ihrer Crews einerseits gesetzlich vorgesehene Ruhezeiten einhalten, wollen aber ihr Personal möglichst effizient einsetzen. Das führt zu Optimierungsproblemen, die sich mit Hilfe aussagenlogischer Formeln beschreiben und lösen lassen.

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

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

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

4.3 Formale Definition der aussagenlogischen Formeln

Wir behandeln zunächst die *Syntax* der Aussagenlogik und besprechen anschließend die *Semantik*. Die *Syntax* gibt an, wie Formeln geschrieben werden. Die *Semantik* befasst sich mit der Bedeutung der Formeln. Nachdem wir die Semantik der aussagenlogischen Formeln definiert haben, zeigen wir, wie sich diese Semantik in SETLX implementieren lässt.

4.3.1 Syntax der aussagenlogischen Formeln

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

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

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

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

gebildet werden. Wir definieren die Menge der aussagenlogischen Formeln \mathcal{F} durch eine induktive Definition:

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

² Die Firma Magma bietet beispielsweise den *Equivalence-Checker Quartz Formal* zum Preis von 150 000 \$ pro Lizenz an. Eine solche Lizenz ist dann drei Jahre lang gültig.

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

Hier steht \top für die Formel, die immer wahr ist, während \perp für die Formel steht, die immer falsch ist. Die Formel \top trägt auch den Namen *Verum*, für \perp sagen wir auch *Falsum*.

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

Jede aussagenlogische Variable ist also eine aussagenlogische Formel.

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

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

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

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

Die Menge \mathcal{F} ist nun die kleinste Teilmenge der aus dem Alphabet \mathcal{A} gebildeten Wörter, die den oben aufgestellten Forderungen genügt.

Beispiel: Gilt $\mathcal{P} = \{p, q, r\}$, so haben wir beispielsweise:

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

□

Um Klammern zu sparen, vereinbaren wir:

1. Äußere Klammern werden weggelassen, wir schreiben also beispielsweise

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

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

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

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

Beachten Sie, dass die Junktoren \wedge und \vee dieselbe Bindungsstärke haben. Das ist anders als in der Sprache SETLX, denn dort bindet der Operator “&&” stärker als der Operator “|”.

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

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

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

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

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

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

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

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

1. Innerhalb einer aussagenlogischen Formel wird der Junktor “*wenn ... , dann*” als “ \rightarrow ” geschrieben.
2. Bei den Beweisen, die wir über aussagenlogische Formeln führen, schreiben wir für diesen Junktor stattdessen “ \Rightarrow ”.

Analog wird der Junktor “*genau dann, wenn*” innerhalb einer aussagenlogischen Formel als “ \leftrightarrow ” geschrieben, aber wenn wir dieser Junktor als Teil eines Beweises verwenden, schreiben wir stattdessen “ \Leftrightarrow ”. \diamond

4.3.2 Semantik der aussagenlogischen Formeln

Um aussagenlogischen Formeln einen Wahrheitswert zuordnen zu können, definieren wir zunächst die Menge \mathbb{B} der Wahrheitswerte:

$$\mathbb{B} := \{\text{true}, \text{false}\}.$$

Damit können wir nun den Begriff einer *aussagenlogischen Interpretation* festlegen.

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

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

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

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

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

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

Wir haben in der Mengenlehre gesehen, dass Funktionen als spezielle Relationen aufgefasst werden können. Die Funktion \neg dreht die Wahrheits-Werte um und kann daher als Relation wie folgt geschrieben werden:

$$\neg = \{(\text{true}, \text{false}), (\text{false}, \text{true})\}.$$

Wir könnten auch die Funktionen \wedge , \vee , \rightarrow und \leftrightarrow als Relationen definieren, es ist aber anschaulicher, wenn wir die Werte dieser Funktionen durch eine Tabelle festlegen. Diese Tabelle ist oben auf Seite 58 abgebildet.

p	q	$\neg(p)$	$\vee(p, q)$	$\wedge(p, q)$	$\rightarrow(p, q)$	$\leftrightarrow(p, q)$
true	true	false	true	true	true	true
true	false	false	true	false	false	false
false	true	true	true	false	true	false
false	false	true	false	false	true	true

Table 4.1: Interpretation der Junktoren.

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

1. $\widehat{\mathcal{I}}(\perp) := \mathbf{false}$.
2. $\widehat{\mathcal{I}}(\top) := \mathbf{true}$.
3. $\widehat{\mathcal{I}}(p) := \mathcal{I}(p)$ für alle $p \in \mathcal{P}$.
4. $\widehat{\mathcal{I}}(\neg f) := \ominus(\widehat{\mathcal{I}}(f))$ für alle $f \in \mathcal{F}$.
5. $\widehat{\mathcal{I}}(f \wedge g) := \oslash(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g))$ für alle $f, g \in \mathcal{F}$.
6. $\widehat{\mathcal{I}}(f \vee g) := \oslash(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g))$ für alle $f, g \in \mathcal{F}$.
7. $\widehat{\mathcal{I}}(f \rightarrow g) := \ominus(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g))$ für alle $f, g \in \mathcal{F}$.
8. $\widehat{\mathcal{I}}(f \leftrightarrow g) := \oplus(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g))$ für alle $f, g \in \mathcal{F}$.

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

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

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

für die aussagenlogische Interpretation \mathcal{I} , die durch $\mathcal{I}(p) = \mathbf{true}$ und $\mathcal{I}(q) = \mathbf{false}$ definiert ist, berechnen lässt:

$$\begin{aligned}
 \mathcal{I}\left((p \rightarrow q) \rightarrow (\neg p \rightarrow q) \rightarrow q\right) &= \ominus\left(\mathcal{I}((p \rightarrow q)), \mathcal{I}((\neg p \rightarrow q) \rightarrow q)\right) \\
 &= \ominus\left(\ominus(\mathcal{I}(p), \mathcal{I}(q)), \mathcal{I}((\neg p \rightarrow q) \rightarrow q)\right) \\
 &= \ominus\left(\ominus(\mathbf{true}, \mathbf{false}), \mathcal{I}((\neg p \rightarrow q) \rightarrow q)\right) \\
 &= \ominus\left(\mathbf{false}, \mathcal{I}((\neg p \rightarrow q) \rightarrow q)\right) \\
 &= \mathbf{true}
 \end{aligned}$$

Beachten Sie, dass wir bei der Berechnung gerade so viele Teile der Formel ausgewertet haben, wie notwendig waren um den Wert der Formel zu bestimmen. Trotzdem ist die eben durchgeführte Rechnung für die Praxis zu umständlich. Stattdessen wird der Wert einer Formel direkt mit Hilfe der Tabelle 4.1 auf Seite 58 berechnet. Wir zeigen exemplarisch, wie wir den Wahrheits-Wert der Formel

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

für beliebige Belegungen \mathcal{I} über diese Tabelle berechnen können. Um nun die Wahrheitswerte dieser Formel unter einer gegebenen Belegung der Aussage-Variablen bestimmen zu können, bauen wir eine Tabelle auf, die für jede in der Formel auftretende Teilformel eine Spalte enthält. Tabelle 4.2 auf Seite 59 zeigt die entstehende Tabelle.

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

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

Betrachten wir die letzte Spalte der Tabelle so sehen wir, dass dort immer der Wert **true** auftritt. Also liefert die Auswertung der Formel $(p \rightarrow q) \rightarrow (\neg p \rightarrow q) \rightarrow q$ für jede aussagenlogische Belegung \mathcal{I} den Wert

true. Formeln, die immer wahr sind, haben in der Aussagenlogik eine besondere Bedeutung und werden als *Tautologien* bezeichnet.

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

$$\mathcal{I}(p) = \text{true} \text{ und } \mathcal{I}(q) = \text{false}.$$

Damit erhalten wir folgende Rechnung:

1. $\mathcal{I}(\neg p) = \ominus(\mathcal{I}(p)) = \ominus(\text{true}) = \text{false}$
2. $\mathcal{I}(p \rightarrow q) = \ominus(\mathcal{I}(p), \mathcal{I}(q)) = \ominus(\text{true}, \text{false}) = \text{false}$
3. $\mathcal{I}(\neg p \rightarrow q) = \ominus(\mathcal{I}(\neg p), \mathcal{I}(q)) = \ominus(\text{false}, \text{false}) = \text{true}$
4. $\mathcal{I}((\neg p \rightarrow q) \rightarrow q) = \ominus(\mathcal{I}(\neg p \rightarrow q), \mathcal{I}(q)) = \ominus(\text{true}, \text{false}) = \text{false}$
5. $\mathcal{I}((p \rightarrow q) \rightarrow (\neg p \rightarrow q) \rightarrow q) = \ominus(\mathcal{I}(p \rightarrow q), \mathcal{I}((\neg p \rightarrow q) \rightarrow q)) = \ominus(\text{false}, \text{false}) = \text{true}$

Für komplexe Formeln ist die Auswertung von Hand viel zu mühsam und fehleranfällig um praktikabel zu sein. Wir zeigen deshalb im übernächsten Abschnitt, wie sich dieser Prozess automatisieren lässt.

4.3.3 Extensionale und intensionale Interpretationen der Aussagenlogik

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

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

berechnen wollen, so müssen wir die Details der Teilformeln f und g nicht kennen, es reicht, wenn wir die Werte $\mathcal{I}(f)$ und $\mathcal{I}(g)$ kennen. Das ist problematisch, denn in der Umgangssprache hat der Junktor “wenn \dots , dann” auch eine *kausale* Bedeutung. Mit der extensionalen Implikation wird der Satz

“Wenn $3 \cdot 3 = 8$, dann schneit es.”

als wahr interpretiert, denn die Formel $3 \cdot 3 = 8$ ist ja falsch. Das ist problematisch, weil wir diesen Satz in der Umgangssprache als sinnlos erkennen. Insofern ist die extensionale Interpretation des sprachlichen Junktors “wenn \dots , dann” nur eine Approximation der umgangssprachlichen Interpretation, die sich für die Mathematik und die Informatik aber als ausreichend erwiesen hat.

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

4.3.4 Implementierung in SETLX

Um die bisher eingeführten Begriffe nicht zu abstrakt werden zu lassen, entwickeln wir in SETLX ein Programm, mit dessen Hilfe sich Formeln auswerten lassen. Jedes Mal, wenn wir ein Programm zur Berechnung irgendwelcher Werte entwickeln wollen, müssen wir uns als erstes fragen, wie wir die Argumente der zu implementierenden Funktion und die Ergebnisse dieser Funktion in der verwendeten Programmier-Sprache darstellen können. In diesem Fall müssen wir uns also überlegen, wie wir eine aussagenlogische Formel in SETLX repräsentieren können, denn Ergebnisswerte **true** und **false** stehen ja als Wahrheitswerte unmittelbar zur Verfügung. Zusammengesetzte Daten-Strukturen können in SETLX am einfachsten als Terme dargestellt werden und das ist auch der Weg, den wir für die aussagenlogischen Formeln beschreiten werden. Wir definieren die Repräsentation von aussagenlogischen Formeln formal dadurch, dass wir eine Funktion

$$\text{rep} : \mathcal{F} \rightarrow \text{SETLX}$$

definieren, die einer aussagenlogischen Formel f einen Term $\text{rep}(f)$ zuordnet. Wir werden dabei die in SETLX bereits vorhandenen logischen Operatoren “!”, “&&”, “||”, “=>” und “<==>” benutzen, denn damit können wir die aussagenlogischen Formeln in sehr natürlicher Weise darstellen.

1. \top wird repräsentiert durch den Wahrheitswert **true**.

$$\text{rep}(\top) := \text{true}$$

2. \perp wird repräsentiert durch den Wahrheitswert **false**.

$$\text{rep}(\perp) := \text{false}$$

3. Eine aussagenlogische Variable $p \in \mathcal{P}$ repräsentieren wir durch einen Term der Form

$$\text{@@@variable}(p).$$

Der Grund für diese zunächst seltsam anmutende Darstellung der Variable liegt darin, dass SETLX intern Variablen in der obigen Form darstellt. Wenn wir später einen **String** mit Hilfe der SETLX-Funktion **parseTerm** in eine aussagenlogische Formel umwandeln wollen, dann werden Variablen automatisch in dieser Form dargestellt. Damit haben wir also

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

Da wir später die Funktion **parseTerm** verwenden um einen String in einen Term umzuwandeln, können wir lateinische Buchstaben als Variablen verwenden.

4. Ist f eine aussagenlogische Formel, so repräsentieren wir $\neg f$ mit Hilfe des Operators “!”:

$$\text{rep}(\neg f) := !\text{rep}(f).$$

5. Sind f_1 und f_2 aussagenlogische Formeln, so repräsentieren wir $f_1 \vee f_2$ mit Hilfe des Operators “||”:

$$\text{rep}(f \vee g) := \text{rep}(f) \mid \mid \text{rep}(g).$$

6. Sind f_1 und f_2 aussagenlogische Formeln, so repräsentieren wir $f_1 \wedge f_2$ mit Hilfe des Operators “&&”:

$$\text{rep}(f \wedge g) := \text{rep}(f) \&\& \text{rep}(g).$$

7. Sind f_1 und f_2 aussagenlogische Formeln, so repräsentieren wir $f_1 \rightarrow f_2$ mit Hilfe des Operators “=>”:

$$\text{rep}(f \rightarrow g) := \text{rep}(f) \Rightarrow \text{rep}(g).$$

8. Sind f_1 und f_2 aussagenlogische Formeln, so repräsentieren wir $f_1 \leftrightarrow f_2$ mit Hilfe des Operators “<==>”:

$$\text{rep}(f \leftrightarrow g) := \text{rep}(f) <==> \text{rep}(g).$$

Bei der Wahl der Repräsentation, mit der wir eine Formel in SETLX repräsentieren, sind wir weitgehend frei. Wir hätten oben sicher auch eine andere Repräsentation verwenden können. Beispielsweise wurden in einer früheren Version dieses Skriptes die aussagenlogischen Formeln als Listen repräsentiert. Eine gute Repräsentation sollte einerseits möglichst intuitiv sein, andererseits ist es auch wichtig, dass die Repräsentation für die zu entwickelnden Algorithmen **adäquat** ist. Im Wesentlichen heißt dies, dass es einerseits einfach sein sollte, auf die Komponenten einer Formel zuzugreifen, andererseits sollte es auch leicht sein, die entsprechende Repräsentation zu erzeugen. Da wir zur Darstellung der aussagenlogischen Formeln dieselben Operatoren verwenden, die auch in SETLX selber benutzt werden, können wir die in SETLX in der Bibliothek **termUtilities** vordefinierte Funktion **parseTerm** benutzen um einen String in eine Formel umzuwandeln. Beispielsweise liefert der Aufruf

```
f := parseTerm("p => p || !q");
```

für f die Formel

```
p => p || !q.
```

Mit Hilfe der SETLX-Funktion **canonical** können wir uns anschauen, wie die Formel in SETLX intern als

Term dargestellt wird. Die Eingabe

```
canonical(f);
```

in der Kommandozeile liefert uns das Ergebnis

```
@@@implication(@@@variable("p"),
               @@@disjunction(@@@variable("p"), @@@not(@@@variable("q"))))
```

Wir erkennen, dass in SETLX der Operator “=>” intern durch das Funktions-Zeichen “@@@implication” dargestellt wird. Dem Operator “||” entspricht das Funktions-Zeichen “@@@disjunction”, der Operator “&&” wird durch “@@@conjunction” repräsentiert und der Operator “!” wird intern als “@@@not” geschrieben.

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

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

von der Menge der Aussage-Variablen \mathcal{P} in die Menge der Wahrheitswerte \mathbb{B} . Ist eine Formel f gegeben, so ist klar, dass bei der Interpretation \mathcal{I} nur die Aussage-Variablen p eine Rolle spielen, die auch in der Formel f auftreten. Wir können daher die Interpretation \mathcal{I} durch eine funktionale Relation darstellen, also durch eine Menge von Paaren der Form $[p, b]$, für die p eine Aussage-Variable ist und für die zusätzlich $b \in \mathbb{B}$ gilt:

$$\mathcal{I} \subseteq \mathcal{P} \times \mathbb{B}.$$

Damit können wir jetzt eine einfache Funktion schreiben, die den Wahrheitswert einer aussagenlogischen Formel f unter einer gegebenen aussagenlogischen Interpretation \mathcal{I} berechnet. Die Funktion `evaluate.stlx` ist in Abbildung 4.1 auf Seite 62 gezeigt. Die Funktion `evaluate` erwartet zwei Argumente:

1. Das erste Argument f ist eine aussagenlogische Formel, die so durch einen Term dargestellt wird, wie wir das weiter oben beschrieben haben.
2. Das zweite Argument I ist eine aussagenlogische Interpretation, die als funktionale Relation dargestellt wird. Für eine aussagenlogische Variable mit dem Namen p können wir den Wert, der dieser Variablen durch I zugeordnet wird, mittels des Ausdrucks $I[p]$ berechnen.

```

1  loadLibrary("termUtilities");
2  evaluate := procedure(f, I) {
3      match (f) {
4          case true:           return true;
5          case false:          return false;
6          case p | isVariable(p): return I[varName(p)];
7          case !g:              return !evaluate(g, I);
8          case g && h:           return evaluate(g, I) && evaluate(h, I);
9          case g || h:           return evaluate(g, I) || evaluate(h, I);
10         case g => h:            return evaluate(g, I) => evaluate(h, I);
11         case g <==> h:          return evaluate(g, I) == evaluate(h, I);
12         default:               abort("syntax error in evaluate($f$, $I$)");
13     }
14 };

```

Figure 4.1: Auswertung einer aussagenlogischen Formel.

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

1. Falls die Formel f den Wert `true` hat, so repräsentiert f die Formel \top . Also ist das Ergebnis der Auswertung unabhängig von der aussagenlogischen Interpretation I immer `true`.

2. Falls die Formel f den Wert **false** hat, so repräsentiert f die Formel \perp . Also ist das Ergebnis der Auswertung unabhängig von der aussagenlogischen Interpretation I immer **false**.
3. In Zeile 6 betrachten wir den Fall, dass das Argument f eine aussagenlogische Variable repräsentiert.
In diesem Fall müssen wir die Belegung I , die ja eine Funktion von den aussagenlogischen Variablen in die Wahrheitswerte ist, auf die Variable f anwenden. Da wir die Belegung als eine funktionale Relation dargestellt haben, können wir diese Relation durch den Ausdruck $I[p]$ sehr einfach für die Variable p auswerten.
4. In Zeile 7 betrachten wir den Fall, dass f die Form $\neg g$ hat und folglich die Formel $\neg g$ repräsentiert. In diesem Fall werten wir erst g unter der Belegung I aus und negieren dann das Ergebnis.
5. In Zeile 8 betrachten wir den Fall, dass f die Form $g_1 \ \&\& \ g_2$ hat und folglich die Formel $g_1 \wedge g_2$ repräsentiert. In diesem Fall werten wir zunächst g_1 und g_2 unter der Belegung I aus und verknüpfen das Ergebnis mit dem Operator “&&”.
6. In Zeile 9 betrachten wir den Fall, dass f die Form $g_1 \ || \ g_2$ hat und folglich die Formel $g_1 \vee g_2$ repräsentiert. In diesem Fall werten wir zunächst g_1 und g_2 unter der Belegung I aus und verknüpfen das Ergebnis mit dem Operator “||”.
7. In Zeile 10 betrachten wir den Fall, dass f die Form $g_1 \Rightarrow g_2$ hat und folglich die Formel $g_1 \rightarrow g_2$ repräsentiert. In diesem Fall werten wir zunächst g_1 und g_2 unter der Belegung I aus und benutzen dann den Operator “=>” der Sprache SETLX.
8. In Zeile 11 führen wir die Auswertung einer Formel $g_1 \ <==> \ g_2$ auf die Gleichheit zurück: Die Formel $f \leftrightarrow g$ ist genau dann wahr, wenn f und g den selben Wahrheitswert haben.
9. Wenn keiner der vorhergehenden Fälle greift, liegt ein Syntax-Fehler vor, auf den wir in Zeile 12 hinweisen.

4.3.5 Eine Anwendung

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

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

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

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

Diese Aussage zerlegen wir zunächst in zwei Teilaussagen:

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

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

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

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

3. Wenn Bruno unschuldig ist, dann ist auch Claus unschuldig:

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

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

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

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

5. Wenn Claus unschuldig ist, ist Anton schuldig.

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

Wir haben nun eine Menge $F = \{f_1, f_2, f_3, f_4, f_5, f_6\}$ von Formeln. Wir fragen uns nun, für welche Belegungen \mathcal{I} alle Formeln aus der Menge F wahr werden. Wenn es genau eine Belegungen gibt, für die dies der Fall ist, dann liefert uns die Belegung den oder die Täter. Eine Belegung entspricht dabei 1-zu-1 der Menge der Täter. Hätten wir beispielsweise

$$\mathcal{I} = \{\langle a, \text{false} \rangle, \langle b, \text{false} \rangle, \langle c, \text{true} \rangle\},$$

so wäre Claus der alleinige Täter. Diese Belegung löst unser Problem allerdings nicht, denn Sie widerspricht der dritten Aussage: Da Bruno unschuldig wäre, wäre dann auch Claus unschuldig. Da es zu zeitraubend ist, alle Belegungen von Hand auszuprobieren, schreiben wir besser ein Programm, das die notwendigen Berechnungen für uns durchführt. Abbildung 4.2 zeigt das Programm `watson.stlx`. Wir diskutieren diese Programm nun Zeile für Zeile.

```

1  createValuation := procedure(M, V) {
2      return { [ x, x in M ] : x in V };
3  };
4  // Austin, Brian, or Colin is guilty.
5  f1 := parse("a || b || c");
6  // If Austin is guilty, he has exactly one accomplice.
7  f2 := parse("a => b || c");    // at least one accomplice
8  f3 := parse("a => !(b && c)"); // at most one accomplice
9  // If Brian is innocent, then Colin is innocent, too.
10 f4 := parse("!b => !c");
11 // If exactly two are guilty, then Colin is one of them.
12 f5 := parse("! (a && b && !c)");
13 // If Colin is innocent, then Austin is guilty.
14 f6 := parse("!c => a");
15 fs := { f1, f2, f3, f4, f5, f6 };
16 v  := { "a", "b", "c" };
17 All := 2 ** v;
18 print("All = ", All);
19 // b is the set of all propositional valuations.
20 B  := { createValuation(m, v) : m in All };
21 s  := { I : I in B | forall (f in fs | evaluate(f, I)) };
22 print("Set of all valuations satisfying all facts: ", s);
23 if (#s == 1) {
24     I := arb(s);
25     offenders := { x : x in v | I[x] };
26     print("Set of offenders: ", offenders);
27 }

```

Figure 4.2: Programm zur Aufklärung des Einbruchs.

1. In den Zeilen 7 – 17 definieren wir die Formeln f_1, \dots, f_6 . Wir müssen hier die Formeln in die SETLX-Repräsentation bringen. Diese Arbeit wird uns durch die Benutzung der Funktion `parse` leicht gemacht.
2. Als nächstes müssen wir uns überlegen, wie wir alle Belegungen aufzählen können. Wir hatten oben schon beobachtet, dass die Belegungen 1-zu-1 zu den möglichen Mengen der Täter korrespondieren. Die Mengen der möglichen Täter sind aber alle Teilmengen der Menge

$$\{\text{"a"}, \text{"b"}, \text{"c"}\}.$$

Wir berechnen daher in Zeile 20 zunächst die Menge aller dieser Teilmengen.

- Wir brauchen jetzt eine Möglichkeit, eine Teilmenge in eine Belegung umzuformen. In den Zeilen 3 – 5 haben wir eine Prozedur implementiert, die genau dies leistet. Um zu verstehen, wie diese Funktion arbeitet, betrachten wir ein Beispiel und nehmen an, dass wir aus der Menge

$$m = \{ \text{"a"}, \text{"c"} \}$$

eine Belegung \mathcal{I} erstellen sollen. Wir erhalten dann

$$\mathcal{I} = \{ \langle \text{"a"}, \text{true} \rangle, \langle \text{"b"}, \text{false} \rangle, \langle \text{"c"}, \text{true} \rangle \}.$$

Das allgemeine Prinzip ist offenbar, dass für eine aussagenlogische Variable x das Paar $\langle x, \text{true} \rangle$ genau dann in der Belegung \mathcal{I} enthalten ist, wenn $x \in m$ ist, andernfalls ist das Paar $\langle x, \text{false} \rangle$ in \mathcal{I} . Damit könnten wir die Menge aller Belegungen, die genau die Elemente aus m wahrmachen, wie folgt schreiben:

$$\{ [x, \text{true}] : x \text{ in } m \} + \{ [x, \text{false}] : x \text{ in all} \mid \neg(x \text{ in } m) \}$$

Es geht aber einfacher, denn wir können beide Fälle zusammenfassen, indem wir fordern, dass das Paar $\langle x, x \in m \rangle$ ein Element der Belegung \mathcal{I} ist. Genau das steht in Zeile 5.

- In Zeile 23 sammeln wir in der Menge b alle möglichen Belegungen auf.
- In Zeile 24 berechnen wir die Menge s aller der Belegungen I , für die alle Formeln aus der Menge fs wahr werden.
- Falls es genau eine Belegung gibt, die alle Formeln wahr macht, dann haben wir das Problem lösen können. In diesem Fall extrahieren wir in Zeile 26 diese Belegungen aus der Menge s und geben anschließend die Menge der Täter aus.

Lassen wir das Programm laufen, so erhalten wir als Ausgabe

Set of offenders: {"b", "c"}

Damit liefern unsere ursprünglichen Formeln ausreichende Information um die Täter zu überführen: Bruno und Claus sind schuldig.

4.4 Tautologien

Die Tabelle in Abbildung 4.2 zeigt, dass die Formel

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

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

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

$$\mathcal{I}(f) = \text{true} \quad \text{für jede aussagenlogische Interpretation } \mathcal{I},$$

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

$$\models f.$$

◇

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

Beispiele:

- $\models p \vee \neg p$
- $\models p \rightarrow p$
- $\models p \wedge q \rightarrow p$

4. $\models p \rightarrow p \vee q$
5. $\models (p \rightarrow \perp) \leftrightarrow \neg p$
6. $\models p \wedge q \leftrightarrow q \wedge p$

Wir können die Tatsache, dass es sich bei diesen Formeln um Tautologien handelt, durch eine Tabelle nachweisen, die analog zu der auf Seite 59 gezeigten Tabelle 4.2 aufgebaut ist. Dieses Verfahren ist zwar konzeptuell sehr einfach, allerdings zu ineffizient, wenn die Anzahl der aussagenlogischen Variablen groß ist. Ziel dieses Kapitels ist daher die Entwicklung eines effizienteren Verfahrens.

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

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

$$\models f \leftrightarrow g$$

gilt.

◇

Beispiele: Es gelten die folgenden Äquivalenzen:

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

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

p	q	$\neg p$	$\neg q$	$p \wedge q$	$\neg(p \wedge q)$	$\neg p \vee \neg q$
true	true	false	false	true	false	false
true	false	false	true	false	true	true
false	true	true	false	false	true	true
false	false	true	true	false	true	true

Table 4.3: Nachweis der ersten DeMorgan'schen Regel.

Werte stehen. Daher sind die Formeln, die zu diesen Spalten gehören, äquivalent.

4.4.1 Testen der Allgemeingültigkeit in SETLX

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

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

Um die aussagenlogische Belegung \mathcal{I} in SETLX darstellen zu können, fassen wir die Belegung \mathcal{I} als links-totale und rechts-eindeutige Relation $\mathcal{I} \subseteq \mathcal{P} \times \mathbb{B}$ auf. Dann haben wir

$$\mathcal{I} = \{ \langle p, \text{true} \rangle \mid p \in M \} \cup \{ \langle p, \text{false} \rangle \mid p \notin M \}.$$

Dies lässt sich noch zu

$$\mathcal{I} = \{ \langle p, p \in M \rangle \mid p \in \mathcal{P} \}$$

vereinfachen. Mit dieser Idee können wir nun eine Prozedur implementieren, die für eine gegebene aussagenlogische Formel f testet, ob f eine Tautologie ist.

```

1  tautology := procedure(f) {
2    P := collectVars(f);
3    // A is the set of all propositional valuations.
4    A := { { [x, x in M] : x in P } : M in 2 ** P };
5    if (forall (I in A | evaluate(f, I))) {
6      return {};
7    } else {
8      return arb({ I : I in A | !evaluate(f, I) });
9    }
10 };
11 collectVars := procedure(f) {
12   match (f) {
13     case true:      return {};
14     case false:     return {};
15     case p | isVariable(p): return { p };
16     case !g:        return collectVars(g);
17     case g && h:     return collectVars(g) + collectVars(h);
18     case g || h:     return collectVars(g) + collectVars(h);
19     case g => h:     return collectVars(g) + collectVars(h);
20     case g <==> h:  return collectVars(g) + collectVars(h);
21     default:        abort("syntax error in collectVars($f$)");
22   }
23 };

```

Figure 4.3: Überprüfung der Allgemeingültigkeit einer aussagenlogischen Formel.

Die in Abbildung 4.3 auf Seite 67 gezeigte Funktion **tautology** testet, ob die als Argument übergebene aussagenlogische Formel f allgemeingültig ist. Die Prozedur verwendet die Funktion **evaluate** aus dem in Abbildung 4.1 auf Seite 62 gezeigten Programm. Wir diskutieren die Definition der Funktion *tautology* nun Zeile für Zeile:

1. In Zeile 2 sammeln wir alle aussagenlogischen Variablen auf, die in der zu überprüfenden Formel auftreten. Die dazu benötigte Prozedur `collectVars` ist in den Zeilen 11 – 23 gezeigt. Diese Prozedur ist durch Induktion über den Aufbau einer Formel definiert und liefert als Ergebnis die Menge aller Aussage-Variablen, die in der aussagenlogischen Formel f auftreten.

Es ist klar, dass bei der Berechnung von $\mathcal{I}(f)$ für eine Formel f und eine aussagenlogische Interpretation \mathcal{I} nur die Werte von $\mathcal{I}(p)$ eine Rolle spielen, für die die Variable p in f auftritt. Zur Analyse von f können wir uns also auf aussagenlogische Interpretationen der Form

$$\mathcal{I} : \mathcal{P} \rightarrow \mathbb{B} \quad \text{mit} \quad \mathcal{P} = \text{collectVars}(f)$$

beschränken.

2. In Zeile 4 berechnen wir die Menge aller aussagenlogischen Interpretationen über der Menge \mathcal{P} der aussagenlogischen Variablen. Wir berechnen für eine Menge m von aussagenlogischen Variablen die Interpretation $\mathcal{I}(m)$ wie oben diskutiert mit Hilfe der Formel

$$\mathcal{I}(m) := \{\langle x, x \in m \rangle \mid x \in \mathcal{P}\}.$$

Betrachten wir zur Verdeutlichung als Beispiel die Formel

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

Die Menge \mathcal{P} der aussagenlogischen Variablen, die in dieser Formel auftreten, ist

$$\mathcal{P} = \{p, q\}.$$

Die Potenz-Menge der Menge \mathcal{P} ist

$$2^{\mathcal{P}} = \{\{\}, \{p\}, \{q\}, \{p, q\}\}.$$

Wir bezeichnen die vier Elemente dieser Menge mit m_1, m_2, m_3, m_4 :

$$m_1 := \{\}, m_2 := \{p\}, m_3 := \{q\}, m_4 := \{p, q\}.$$

Aus jeder dieser Mengen m_i gewinnen wir nun eine aussagenlogische Interpretation $\mathcal{I}(m_i)$:

$$\mathcal{I}(m_1) := \{\langle x, x \in \{\} \rangle \mid x \in \{p, q\}\} = \{\langle p, p \in \{\} \rangle, \langle q, q \in \{\} \rangle\} = \{\langle p, \text{false} \rangle, \langle q, \text{false} \rangle\}.$$

$$\mathcal{I}(m_2) := \{\langle x, x \in \{p\} \rangle \mid x \in \{p, q\}\} = \{\langle p, p \in \{p\} \rangle, \langle q, q \in \{p\} \rangle\} = \{\langle p, \text{true} \rangle, \langle q, \text{false} \rangle\}.$$

$$\mathcal{I}(m_3) := \{\langle x, x \in \{q\} \rangle \mid x \in \{p, q\}\} = \{\langle p, p \in \{q\} \rangle, \langle q, q \in \{q\} \rangle\} = \{\langle p, \text{false} \rangle, \langle q, \text{true} \rangle\}.$$

$$\mathcal{I}(m_4) := \{\langle x, x \in \{p, q\} \rangle \mid x \in \{p, q\}\} = \{\langle p, p \in \{p, q\} \rangle, \langle q, q \in \{p, q\} \rangle\} = \{\langle p, \text{true} \rangle, \langle q, \text{true} \rangle\}.$$

damit haben wir alle möglichen Interpretationen der Variablen p und q .

3. In Zeile 5 testen wir, ob die Formel f für alle möglichen Interpretationen I aus der Menge a aller Interpretationen wahr ist. Ist dies der Fall, so geben wir die leere Menge als Ergebnis zurück.

Falls es allerdings eine Belegungen I in der Menge a gibt, für die die Auswertung von f den Wert *false* liefert, so bilden wir in Zeile 8 die Menge aller solcher Belegungen und wählen mit Hilfe der Funktion *arb* eine beliebige Belegungen aus dieser Menge aus, die wir dann als Gegenbeispiel zurück geben.

4.4.2 Nachweis der Allgemeingültigkeit durch Äquivalenz-Umformungen

Wollen wir nachweisen, dass eine Formel eine Tautologie ist, können wir uns prinzipiell immer einer Wahrheits-Tafel bedienen. Aber diese Methode hat einen Haken: Kommen in der Formel n verschiedene Aussage-Variablen vor, so hat die Tabelle 2^n Zeilen. Beispielsweise hat die Tabelle zum Nachweis eines der Distributiv-Gesetze bereits 8 Zeilen, da hier 3 verschiedene Variablen auftreten. Eine andere Möglichkeit nachzuweisen, dass eine Formel eine Tautologie ist, ergibt sich dadurch, dass wir die Formel mit Hilfe der oben aufgeführten Äquivalenzen *vereinfachen*. Wenn es gelingt, eine Formel F unter Verwendung dieser Äquivalenzen zu \top zu vereinfachen, dann ist gezeigt, dass F eine Tautologie ist. Wir demonstrieren das Verfahren zunächst an einem Beispiel. Mit Hilfe einer Wahrheits-Tafel hatten wir schon gezeigt, dass die Formel

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

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

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

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

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

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

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

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

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

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

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

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

$$k = l_1 \vee \dots \vee l_r$$

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

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

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

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

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

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

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

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

$$l_1 \vee \dots \vee l_r \vee \perp \leftrightarrow l_1 \vee \dots \vee l_r$$

in Mengen-Schreibweise, so erhalten wir

$$\{l_1, \dots, l_r, \perp\} \leftrightarrow \{l_1, \dots, l_r\}.$$

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

$$\{\perp\} \leftrightarrow \{\}.$$

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

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

1. $\top \in k$.
2. Es existiert $p \in \mathcal{P}$ mit $p \in k$ und $\neg p \in k$.

In diesem Fall bezeichnen wir p und $\neg p$ als **komplementäre Literale**. \diamond

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

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

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

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

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

1. $\mathcal{I}(p_i) = \text{true}$ für alle $i = 1, \dots, m$ und

2. $\mathcal{I}(q_j) = \text{false}$ für alle $j = 1, \dots, n$,

Mit dieser Interpretation würde offenbar $\mathcal{I}(k) = \text{false}$ gelten und damit könnte k keine Tautologie sein. Also ist die Annahme, dass k nicht trivial ist, falsch. \square

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

$$f = k_1 \wedge \dots \wedge k_n,$$

wobei die k_i für alle $i = 1, \dots, n$ Klauseln sind. \diamond

Aus der Definition der KNF folgt sofort:

Korollar 14 Ist $f = k_1 \wedge \dots \wedge k_n$ in konjunktiver Normalform, so gilt

$$\models f \quad \text{genau dann, wenn} \quad \models k_i \quad \text{für alle } i = 1, \dots, n. \quad \square$$

Damit können wir für eine Formel $f = k_1 \wedge \dots \wedge k_n$ in konjunktiver Normalform leicht entscheiden, ob f eine Tautologie ist, denn f ist genau dann eine Tautologie, wenn alle Klauseln k_i trivial sind.

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

$$f = k_1 \wedge \dots \wedge k_n$$

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

$$f = \{k_1, \dots, k_n\}.$$

Wir geben ein Beispiel: Sind p, q und r Aussage-Variablen, so ist die Formel

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

in konjunktiver Normalform. In Mengen-Schreibweise wird daraus

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

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

1. Eliminiere alle Vorkommen des Junktors “ \leftrightarrow ” mit Hilfe der Äquivalenz

$$(f \leftrightarrow g) \leftrightarrow (f \rightarrow g) \wedge (g \rightarrow f)$$

2. Eliminiere alle Vorkommen des Junktors “ \rightarrow ” mit Hilfe der Äquivalenz

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

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

$$(a) \quad \neg \perp \leftrightarrow \top$$

$$(b) \quad \neg \top \leftrightarrow \perp$$

$$(c) \quad \neg \neg f \leftrightarrow f$$

$$(d) \quad \neg(f \wedge g) \leftrightarrow \neg f \vee \neg g$$

$$(e) \quad \neg(f \vee g) \leftrightarrow \neg f \wedge \neg g$$

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

4. Stehen in der Formel jetzt “ \vee ”-Junktoren über “ \wedge ”-Junktoren, so können wir durch *Ausmultiplizieren*, sprich Verwendung der Äquivalenz

$$(f_1 \wedge \cdots \wedge f_m) \vee (g_1 \wedge \cdots \wedge g_n) \\ \leftrightarrow ((f_1 \vee g_1) \wedge \cdots \wedge (f_1 \vee g_n)) \wedge \cdots \wedge ((f_m \vee g_1) \wedge \cdots \wedge (f_m \vee g_n))$$

den Junktor “ \vee ” nach innen schieben.

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

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

Wir demonstrieren das Verfahren am Beispiel der Formel

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

1. Da die Formel den Junktor “ \leftrightarrow ” nicht enthält, ist im ersten Schritt nichts zu tun.
2. Die Elimination von “ \rightarrow ” liefert

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

3. Die Umrechnung auf Negations-Normalform liefert

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

4. Durch “Ausmultiplizieren” erhalten wir

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

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

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

Da die Reihenfolge der Elemente einer Menge aber unwichtig ist und außerdem eine Menge jedes Element nur einmal enthält, stellen wir fest, dass diese beiden Klauseln gleich sind. Fassen wir jetzt die Klauseln noch in einer Menge zusammen, so erhalten wir

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

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

Damit ist die Formel in KNF überführt.

4.4.3 Berechnung der konjunktiven Normalform in SETLX

Wir geben nun eine Reihe von Prozeduren an, mit deren Hilfe sich eine gegebene Formel f in konjunktive Normalform überführen lässt. Wir beginnen mit einer Prozedur

$$\text{elimGdw} : \mathcal{F} \rightarrow \mathcal{F}$$

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

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

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

2. Hat f die Form $f = \neg g$, so eliminieren wir den Junktoren “ \neg ” aus der Formel g :

$$\text{elimGdw}(\neg g) = \neg \text{elimGdw}(g).$$

3. Im Falle $f = g_1 \wedge g_2$ eliminieren wir den Junktoren “ \wedge ” aus den Formeln g_1 und g_2 :

$$\text{elimGdw}(g_1 \wedge g_2) = \text{elimGdw}(g_1) \wedge \text{elimGdw}(g_2).$$

4. Im Falle $f = g_1 \vee g_2$ eliminieren wir den Junktoren “ \vee ” aus den Formeln g_1 und g_2 :

$$\text{elimGdw}(g_1 \vee g_2) = \text{elimGdw}(g_1) \vee \text{elimGdw}(g_2).$$

5. Im Falle $f = g_1 \rightarrow g_2$ eliminieren wir den Junktoren “ \rightarrow ” aus den Formeln g_1 und g_2 :

$$\text{elimGdw}(g_1 \rightarrow g_2) = \text{elimGdw}(g_1) \rightarrow \text{elimGdw}(g_2).$$

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

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

Das führt auf die Gleichung:

$$\text{elimGdw}(g_1 \leftrightarrow g_2) = \text{elimGdw}((g_1 \rightarrow g_2) \wedge (g_2 \rightarrow g_1)).$$

Der Aufruf von `elimGdw` auf der rechten Seite der Gleichung ist notwendig, denn der Junktoren “ \leftrightarrow ” kann ja noch in g_1 und g_2 auftreten.

Abbildung 4.4 auf Seite 73 zeigt die Implementierung der Prozedur `elimGdw`.

```

1  elimGdw := procedure(f) {
2      match (f) {
3          case !g      : return !elimGdw(g);
4          case g && h   : return elimGdw(g) && elimGdw(h);
5          case g || h   : return elimGdw(g) || elimGdw(h);
6          case g => h   : return elimGdw(g) => elimGdw(h);
7          case g <==> h : return elimGdw((g => h) && (h => g));
8          default      : return f;
9      }
10 };

```

Figure 4.4: Elimination von \leftrightarrow .

Als nächstes betrachten wir die Prozedur zur Elimination des Junktors “ \rightarrow ”. Abbildung 4.5 auf Seite 74 zeigt die Implementierung der Funktion `elimFolgt`. Die der Implementierung zu Grunde liegende Idee ist dieselbe wie bei der Elimination des Junktors “ \leftrightarrow ”. Der einzige Unterschied besteht darin, dass wir jetzt die Äquivalenz

$$(g_1 \rightarrow g_2) \leftrightarrow (\neg g_1 \vee g_2)$$

benutzen. Außerdem können wir schon voraussetzen, dass der Junktoren “ \leftrightarrow ” bereits vorher eliminiert wurde. Dadurch entfällt ein Fall.

Als nächstes zeigen wir die Routinen zur Berechnung der Negations-Normalform. Abbildung 4.6 auf Seite 75 zeigt die Implementierung der Funktionen `nnf` und `neg`, die sich wechselseitig aufrufen. Dabei berechnet `neg(f)` die Negations-Normalform von $\neg f$, während `nnf(f)` die Negations-Normalform von f berechnet, es gilt also

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

```

1  elimFolgt := procedure(f) {
2      match (f) {
3          case !g      : return !elimFolgt(g);
4          case g && h : return  elimFolgt(g) && elimFolgt(h);
5          case g || h : return  elimFolgt(g) || elimFolgt(h);
6          case g => h : return !elimFolgt(g) || elimFolgt(h);
7          default      : return f;
8      }
9  };

```

Figure 4.5: Elimination von \rightarrow .

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

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

zur Anwendung. Wir beschreiben die Umformung in Negations-Normalform durch die folgenden Gleichungen:

1. $\text{nnf}(\neg f) = \text{neg}(f)$,
2. $\text{nnf}(f_1 \wedge f_2) = \text{nnf}(f_1) \wedge \text{nnf}(f_2)$,
3. $\text{nnf}(f_1 \vee f_2) = \text{nnf}(f_1) \vee \text{nnf}(f_2)$.

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

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

Als letztes stellen wir die Prozeduren vor, mit denen die Formeln, die bereits in Negations-Normalform sind, ausmultipliziert und dadurch in konjunktive Normalform gebracht werden. Gleichzeitig werden die zu normalisierenden Formeln dabei in die Mengen-Schreibweise transformiert, d.h. die Formeln werden als Mengen von Mengen von Literalen dargestellt. Dabei interpretieren wir eine Menge von Literalen als Disjunktion der Literalen und eine Menge von Klauseln interpretieren wir als Konjunktion der Klauseln. Abbildung 4.7 auf Seite 76 zeigt die Implementierung der Funktion **knf**.

1. Falls die Formel f , die wir in KNF transformieren wollen, die Form

$$f = \neg g$$

```

1  nnf := procedure(f) {
2      match (f) {
3          case !g      : return neg(g);
4          case g && h   : return nnf(g) && nnf(h);
5          case g || h   : return nnf(g) || nnf(h);
6          default      : return f;
7      }
8  };
9  neg := procedure(f) {
10     match (f) {
11         case !g      : return nnf(g);
12         case g && h   : return neg(g) || neg(h);
13         case g || h   : return neg(g) && neg(h);
14         default      : return !f;
15     }
16 };

```

Figure 4.6: Berechnung der Negations-Normalform.

hat, so muss g eine Aussage-Variable sein, denn f ist ja bereits in Negations-Normalform. Damit können wir f in eine Klausel transformieren, indem wir $\{\neg g\}$, also $\{f\}$ schreiben. Da eine KNF eine Menge von Klauseln ist, ist dann $\{\{f\}\}$ das Ergebnis, das wir in Zeile 3 zurück geben.

2. Falls $f = f_1 \wedge f_2$ ist, transformieren wir zunächst f_1 und f_2 in KNF. Dabei erhalten wir

$$\text{knf}(f_1) = \{h_1, \dots, h_m\} \quad \text{und} \quad \text{knf}(f_2) = \{k_1, \dots, k_n\}.$$

Dabei sind die h_i und die k_j Klauseln. Um nun die KNF von $f_1 \wedge f_2$ zu bilden, reicht es aus, die Vereinigung dieser beiden Mengen zu bilden, wir haben also

$$\text{knf}(f_1 \wedge f_2) = \text{knf}(f_1) \cup \text{knf}(f_2).$$

Das liefert Zeile 4 der Implementierung.

3. Falls $f = f_1 \vee f_2$ ist, transformieren wir zunächst f_1 und f_2 in KNF. Dabei erhalten wir

$$\text{knf}(f_1) = \{h_1, \dots, h_m\} \quad \text{und} \quad \text{knf}(f_2) = \{k_1, \dots, k_n\}.$$

Dabei sind die h_i und die k_j Klauseln. Um nun die KNF von $f_1 \vee f_2$ zu bilden, rechnen wir wie folgt:

$$\begin{aligned}
& f_1 \vee f_2 \\
& \Leftrightarrow (h_1 \wedge \cdots \wedge h_m) \vee (k_1 \wedge \cdots \wedge k_n) \\
& \Leftrightarrow (h_1 \vee k_1) \quad \wedge \quad \cdots \quad \wedge \quad (h_m \vee k_1) \quad \wedge \\
& \qquad \qquad \qquad \vdots \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \vdots \\
& (h_1 \vee k_n) \quad \wedge \quad \cdots \quad \wedge \quad (h_m \vee k_n) \\
& \Leftrightarrow \{h_i \vee k_j : i \in \{1, \dots, m\}, j \in \{1, \dots, n\}\}
\end{aligned}$$

Berücksichtigen wir noch, dass Klauseln in der Mengen-Schreibweise als Mengen von Literalen aufgefasst werden, die implizit disjunktiv verknüpft werden, so können wir für $h_i \vee k_j$ auch $h_i \cup k_j$ schreiben. Insgesamt erhalten wir damit

$$\text{knf}(f_1 \vee f_2) = \{h \cup k \mid h \in \text{knf}(f_1) \wedge k \in \text{knf}(f_2)\}.$$

Das liefert die Zeile 5 der Implementierung der Prozedur knf.

4. Falls die Formel f , die wir in KNF transformieren wollen, eine Aussage-Variable ist, so transformieren wir

f zunächst in eine Klausel. Das liefert $\{f\}$. Da eine KNF eine Menge von Klauseln ist, ist die KNF dann $\{\{f\}\}$. Dieses Ergebnis geben wir in Zeile 6 zurück.

```

1  knf := procedure(f) {
2      match (f) {
3          case !g      : return { { !g } };
4          case g && h : return knf(g) + knf(h);
5          case g || h : return { k1 + k2 : k1 in knf(g), k2 in knf(h) };
6          default      : return { { f } }; // f is a variable
7      }
8  };

```

Figure 4.7: Berechnung der konjunktiven Normalform.

Zum Abschluss zeigen wir in Abbildung 4.8 auf Seite 76 wie die einzelnen Funktionen zusammenspielen. Die Funktion **normalize** eliminiert zunächst die Junktoren “ \leftrightarrow ” und “ \rightarrow ” und bringt die Formel in Negations-Normalform. Die Negations-Normalform wird nun mit Hilfe der Funktion **knf** in konjunktive Normalform gebracht, wobei gleichzeitig die Formel in Mengen-Schreibweise überführt wird. Schließlich entfernt die Funktion **simplify** alle Klauseln aus der Menge **n4**, die trivial sind. Die Funktion **isTrivial** überprüft, ob eine Klausel C , die in Mengen-Schreibweise vorliegt, sowohl eine Variable p als auch die Negation $\neg p$ dieser Variablen enthält, denn dann ist diese Klausel trivial. Das vollständige Programm zur Berechnung der konjunktiven Normalform finden Sie als die Datei **knf.stlx** unter GitHub.

```

1  normalize := procedure(f) {
2      n1 := elimGdw(f);
3      n2 := elimFolgt(n1);
4      n3 := nnf(n2);
5      n4 := knf(n3);
6      return simplify(n4);
7  };
8  simplify := procedure(F) {
9      return { C : C in F | !isTrivial(C) };
10 };
11 isTrivial := procedure(C) {
12     return { p : p in C | !p in C } != {};
13 };

```

Figure 4.8: Normalisierung einer Formel

4.5 Der Herleitungs-Begriff

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

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

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

$$f_1 \wedge \dots \wedge f_n \rightarrow g$$

äquivalent ist. Diese Formel ist nun genau dann eine Tautologie, wenn jede der Klauseln k_1, \dots, k_n trivial ist.

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

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

und erhalten nach mühsamer Rechnung

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

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

Wir stellen daher nun ein weiteres Verfahren vor, mit dessen Hilfe wir entscheiden können, ob eine Formel aus einer gegebenen Menge von Formeln folgt. Die Idee bei diesem Verfahren ist es, die Formel f mit Hilfe von [Schluss-Regeln](#) aus den gegebenen Formeln f_1, \dots, f_n herzuleiten. Das Konzept einer Schluss-Regel wird in der nun folgenden Definition festgelegt.

Definition 15 (Schluss-Regel) Eine [Schluss-Regel](#) ist eine Paar $\langle \{f_1, \dots, f_n\}, k \rangle$. Dabei ist $\{f_1, \dots, f_n\}$ eine Menge von Formeln und k ist eine einzelne Formel. Die Formeln f_1, \dots, f_n bezeichnen wir als [Prämissen](#), die Formel k heißt die [Konklusion](#) der Schluss-Regel. Ist das Paar $\langle \{f_1, \dots, f_n\}, k \rangle$ eine Schluss-Regel, so schreiben wir dies als:

$$\frac{f_1 \quad \dots \quad f_n}{k}.$$

Wir lesen diese Schluss-Regel wie folgt: “Aus f_1, \dots, f_n kann auf k geschlossen werden.” \diamond

Beispiele für Schluss-Regeln:

Modus Ponens	Modus Ponendo Tollens	Modus Tollendo Tollens
$\frac{p \quad p \rightarrow q}{q}$	$\frac{\neg q \quad p \rightarrow q}{\neg p}$	$\frac{\neg p \quad p \rightarrow q}{\neg q}$

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

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

$$\frac{f_1 \quad \dots \quad f_n}{k}$$

ist genau dann [korrekt](#), wenn $\models f_1 \wedge \dots \wedge f_n \rightarrow k$ gilt. \diamond

Mit dieser Definition sehen wir, dass die oben als “[Modus Ponens](#)” und “[Modus Ponendo Tollens](#)” bezeichneten Schluss-Regeln korrekt sind, während die als “[Modus Tollendo Tollens](#)” bezeichnete Schluss-Regel nicht korrekt ist.

Im Folgenden gehen wir davon aus, dass alle Formeln Klauseln sind. Einerseits ist dies keine echte Einschränkung, denn wir können ja jede Formel in eine äquivalente Menge von Klauseln umrechnen. Andererseits haben viele in der Praxis auftretende aussagenlogische Probleme die Gestalt von Klauseln. Daher stellen wir jetzt eine Schluss-Regel vor, in der sowohl die Prämissen als auch die Konklusion Klauseln sind.

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

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

◇

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

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

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

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

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

Satz 18 Die Schnitt-Regel ist korrekt.

Beweis: Wir müssen zeigen, dass

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

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

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

□

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

$$M \vdash f.$$

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

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

$$\text{Falls } f \in M \text{ ist, dann gilt } M \vdash f.$$

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

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

◇

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

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

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

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

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

Dieses Beispiel zeigt, dass die Klausel $k_1 \cup k_2$ durchaus auch weniger Elemente enthalten kann als die Summe $\#k_1 + \#k_2$. Dieser Fall tritt genau dann ein, wenn es Literale gibt, die sowohl in k_1 als auch in k_2 vorkommen.

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

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

$$\text{knf}(p \rightarrow q) = \{\{\neg p, q\}\}, \quad \text{knf}(q \rightarrow r) = \{\{\neg q, r\}\}, \quad \text{knf}(p \rightarrow r) = \{\{\neg p, r\}\}.$$

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

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

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

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

4.5.1 Eigenschaften des Herleitungs-Begriffs

Die Relation \vdash hat zwei wichtige Eigenschaften:

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

$$\text{Wenn } \{k_1, \dots, k_n\} \vdash k \text{ gilt, dann gilt auch } \models k_1 \wedge \dots \wedge k_n \rightarrow k.$$

Beweis: Der Beweis verläuft durch eine Induktion nach der Definition der Relation \vdash .

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

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

zeigen, was offensichtlich ist.

2. Fall: Es gilt $\{k_1, \dots, k_n\} \vdash k$, weil es eine aussagenlogische Variable p und Klauseln g und h gibt, so dass

$$\{k_1, \dots, k_n\} \vdash g \cup \{p\} \quad \text{und} \quad \{k_1, \dots, k_n\} \vdash h \cup \{\neg p\}$$

gilt und daraus haben wir mit der Schnitt-Regel auf

$$\{k_1, \dots, k_n\} \vdash g \cup h$$

geschlossen, wobei $k = g \cup h$ gilt. Wir müssen nun zeigen, dass

$$\models k_1 \wedge \dots \wedge k_n \rightarrow g \vee h$$

gilt. Es sei also \mathcal{I} eine aussagenlogische Interpretation, so dass

$$\mathcal{I}(k_1 \wedge \dots \wedge k_n) = \text{true}$$

ist. Dann müssen wir zeigen, dass

$$\mathcal{I}(g) = \text{true} \quad \text{oder} \quad \mathcal{I}(h) = \text{true}$$

ist. Nach Induktions-Voraussetzung wissen wir

$$\models k_1 \wedge \cdots \wedge k_n \rightarrow g \vee p \quad \text{und} \quad \models k_1 \wedge \cdots \wedge k_n \rightarrow h \vee \neg p.$$

Wegen $\mathcal{I}(k_1 \wedge \cdots \wedge k_n) = \mathbf{true}$ folgt dann

$$\mathcal{I}(g \vee p) = \mathbf{true} \quad \text{und} \quad \mathcal{I}(h \vee \neg p) = \mathbf{true}.$$

Nun gibt es zwei Fälle:

(a) Fall: $\mathcal{I}(p) = \mathbf{true}$.

Dann ist $\mathcal{I}(\neg p) = \mathbf{false}$ und daher folgt aus der Tatsache, dass $\mathcal{I}(h \vee \neg p) = \mathbf{true}$ ist, dass

$$\mathcal{I}(h) = \mathbf{true}$$

sein muss. Daraus folgt aber sofort

$$\mathcal{I}(g \vee h) = \mathbf{true}. \quad \checkmark$$

(b) Fall: $\mathcal{I}(p) = \mathbf{false}$.

Nun folgt aus $\mathcal{I}(g \vee p) = \mathbf{true}$, dass

$$\mathcal{I}(g) = \mathbf{true}$$

gelten muss. Also gilt auch in diesem Fall

$$\mathcal{I}(g \vee h) = \mathbf{true}. \quad \checkmark$$

□

Die Umkehrung dieses Satzes gilt leider nur in abgeschwächter Form und zwar dann, wenn k die leere Klausel ist, also im Fall $k = \perp$.

Satz 21 (Widerlegungs-Vollständigkeit) Ist $M = \{k_1, \dots, k_n\}$ eine Menge von Klauseln, so haben wir:

Wenn $\models k_1 \wedge \cdots \wedge k_n \rightarrow \perp$ gilt, dann gilt auch $M \vdash \{\}$.

4.5.2 Beweis der Widerlegungs-Vollständigkeit

Der Beweis der Widerlegungs-Vollständigkeit der Aussagenlogik benötigt den Begriff der *Erfüllbarkeit*, den wir jetzt formal einführen.

Definition 22 (Erfüllbarkeit) Es sei M eine Menge von aussagenlogischen Formeln. Falls es eine aussagenlogische Interpretation \mathcal{I} gibt, die alle Formeln aus M erfüllt, nennen wir M *erfüllbar*.

Wir sagen, dass M *unerfüllbar* ist und schreiben

$$M \models \perp$$

wenn es keine aussagenlogische Interpretation \mathcal{I} gibt, die alle Formel aus M erfüllt. Bezeichnen wir die Menge der aussagenlogischen Interpretationen mit *ALI*, so schreibt sich das formal als

$$M \models \perp \quad \text{g.d.w.} \quad \forall \mathcal{I} \in \text{ALI} : \exists g \in M : \mathcal{I}(g) = \mathbf{false}.$$

◇

Bemerkung: Ist $M = \{k_1, \dots, k_n\}$ eine Menge von Klauseln, so können Sie sich leicht überlegen, dass M genau dann nicht erfüllbar ist, wenn

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

gilt.

◇

Wir führen den Beweis der Widerlegungs-Vollständigkeit mit Hilfe eines Programms, das in den Abbildungen 4.9, 4.10 und 4.11 auf den folgenden Seiten gezeigt ist. Die Grundidee bei diesem Programm besteht darin, dass wir versuchen, aus einer gegebenen Menge M von Klauseln alle Klauseln herzuleiten, die mit der Schnittregel aus M herleitbar sind. Wenn wir dabei auch die leere Klausel herleiten, dann ist M aufgrund der Korrektheit der Schnitt-Regel offenbar unerfüllbar. Falls es uns aber nicht gelingt, die leere Klausel aus M abzuleiten,

dann konstruieren wir aus der Menge aller Klauseln, die wir aus M hergeleitet haben, eine aussagenlogische Interpretation \mathcal{I} , die alle Klauseln aus M erfüllt.

Wir diskutieren zunächst die Hilfsprozeduren, die in Abbildung 4.9 gezeigt sind.

1. Die Funktion **complement** erhält als Argument ein Literal l und berechnet das *Komplement* \overline{l} dieses Literals. Falls l die Form $\neg p$ mit einer aussagenlogischen Variablen p hat, so gilt $\overline{\neg p} = p$. Falls das Literal l eine aussagenlogische Variable p ist, haben wir $\overline{p} = \neg p$.
2. Die Funktion **extractVar** extrahiert die aussagenlogische Variable aus einem Literal l . Die Implementierung verläuft analog zur Implementierung der Funktion **complement** über eine Fallunterscheidung, bei der wir berücksichtigen, dass l entweder die Form $\neg p$ oder die Form p hat, wobei p eine aussagenlogische Variable ist.
3. Die Funktion **collectVars** erhält als Argument eine Menge M von Klauseln, wobei die einzelnen Klauseln $C \in M$ als Mengen von Literalen dargestellt werden. Aufgabe der Funktion **collectVars** ist es, die Menge aller aussagenlogischen Variablen zu berechnen, die in einer der Klauseln C aus M vorkommen. Bei der Implementierung iterieren wir zunächst über die Klauseln C der Menge M und dann für jede Klausel C über die in C vorkommenden Literale l , wobei die Literale mit Hilfe der Funktion **extractVar** in aussagenlogische Variablen umgewandelt werden.
4. Die Funktion **cutRule** erhält als Argumente zwei Klauseln C_1 und C_2 und berechnet *alle* die Klauseln, die mit Hilfe der Schnittregel aus C_1 und C_2 gefolgert werden können. Beispielsweise können wir aus den beiden Klauseln

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

mit der Schnitt-Regel sowohl die Klausel

$$\{q, \neg q\} \quad \text{als auch die Klausel} \quad \{p, \neg p\}$$

herleiten.

```

1  complement := procedure(l) {
2      match (l) {
3          case !p : return p;
4          case p : return !p;
5      }
6  };
7  extractVar := procedure(l) {
8      match (l) {
9          case !p : return p;
10         case p : return p;
11     }
12 };
13 collectVars := procedure(M) {
14     return { extractVar(l) : C in M, l in C };
15 };
16 cutRule := procedure(C1, C2) {
17     return { (C1 - {l}) + (C2 - {complement(l)})
18             : l in C1
19             | complement(l) in C2
20         };
21 };

```

Figure 4.9: Verschiedene Hilfsprozeduren, die in Abbildung 4.10 genutzt werden.

```

22  saturate := procedure(Clauses) {
23      while (true) {
24          Derived := {} +/ { cutRule(C1, C2) : C1 in Clauses, C2 in Clauses };
25          if ({} in Derived) {
26              return { {} }; // clauses are inconsistent
27          }
28          Derived -= Clauses;
29          if (Derived == {}) {
30              return Clauses; // no new clauses found
31          }
32          Clauses += Derived;
33      }
34  };

```

Figure 4.10: Die Funktion `saturate`.

Abbildung 4.10 zeigt die Funktion `saturate`. Diese Funktion erhält als Eingabe eine Menge `Clauses` von aussagenlogischen Klauseln, die als Mengen von Literalen dargestellt werden. Aufgabe der Funktion ist es, alle Klauseln herzuleiten, die mit Hilfe der Schnittregel auf direktem oder indirekten Wege aus der Menge `Clauses` hergeleitet werden können. Genauer sagen wir, dass die Menge S der Klauseln, die von der Funktion `saturate` zurück gegeben wird, unter Anwendung der Schnitt-Regel *saturiert* ist, was formal wie folgt definiert ist:

1. Falls S die leere Klausel $\{\}$ enthält, dann ist S saturiert.
2. Andernfalls muss `Clauses` eine Teilmenge von S sein und es muss zusätzlich Folgendes gelten: Falls $C_1 \cup \{l\}$ und $C_2 \cup \{\bar{l}\}$ Klauseln aus S sind, dann ist auch die Klausel $C_1 \cup C_2$ ein Element der Klauselmenge S :

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

Wir erläutern nun die Implementierung der Funktion `saturate`.

1. Die `while`-Schleife, die in Zeile 23 beginnt, hat die Aufgabe, die Schnitt-Regel solange wie möglich anzuwenden um mit Hilfe der Schnitt-Regel neue Klauseln aus den gegebenen Klauseln herzuleiten. Da die Bedingung dieser Schleife den Wert `true` hat, kann diese Schleife nur durch die Ausführung des `return`-Befehls in Zeile 31 abgebrochen werden.
2. In Zeile 24 wird die Menge `Derived` dadurch berechnet, dass alle Klauseln berechnet werden, die mit Hilfe der Schnitt-Regel aus zwei der Klauseln in der Menge `Clauses` gefolgert werden können.
3. Falls die Menge `Derived` die leere Klausel enthält, dann ist die Menge `Clauses` widersprüchlich und die Funktion `saturate` gibt als Ergebnis die Menge $\{\{\}\}$ zurück.
4. Andernfalls ziehen wir in Zeile 28 von der Menge `Derived` zunächst die Klauseln ab, die wir schon in der Menge `Clauses` vorhanden waren, denn es geht uns darum festzustellen, ob wir im letzten Schritt tatsächlich neue Klauseln gefunden haben, oder ob alle Klauseln, die wir im letzten Schritt in Zeile 24 hergeleitet haben, eigentlich schon vorher bekannt waren.
5. Falls wir nun in Zeile 29 feststellen, dass wir keine neuen Klauseln hergeleitet haben, dann ist die Menge `Clauses` saturiert und wir geben diese Menge in Zeile 30 zurück.
6. Andernfalls fügen wir in Zeile 32 die Klauseln, die wir neu gefunden haben, zu der Menge `Clauses` hinzu und setzen die `while`-Schleife fort.

An dieser Stelle müssen wir uns überlegen, dass die `while`-Schleife tatsächlich irgendwann abbricht. Das hat zwei Gründe:

1. In jeder Iteration der Schleife wird die Anzahl der Elemente der Menge **Clauses** mindestens um Eins erhöht, denn wir wissen ja, dass die Menge **Derived**, die wir zu **Clauses** hinzufügen, einerseits nicht leer ist und andererseits auch nur solche Klauseln enthält, die nicht bereits in **Clauses** auftreten.
2. Die Menge **Clauses**, mit der wir ursprünglich starten, enthält eine bestimmte Anzahl n von aussagenlogischen Variablen. Bei der Anwendung der Schnitt-Regel werden aber keine neue Variablen erzeugt. Daher bleibt die Anzahl der aussagenlogischen Variablen, die in **Clauses** auftreten, immer gleich. Damit ist natürlich auch die Anzahl der Literale, die in **Clauses** auftreten, beschränkt: Wenn es nur n aussagenlogische Variablen gibt, dann kann es auch höchstens $2 \cdot n$ Literale geben. Jede Klausel aus **Clauses** ist aber eine Teilmenge der Menge aller Literale. Da eine Menge mit k Elementen insgesamt 2^k Teilmengen hat, gibt es höchstens $2^{2 \cdot n}$ verschiedene Klauseln, die in **Clauses** auftreten können.

Aus den beiden oben angegebenen Gründen können wir schließen, dass die **while**-Schleife in Zeile 23 nach spätestens $2^{2 \cdot n}$ Iterationen abgebrochen wird.

```

35  findValuation := procedure(Clauses) {
36      Variables := collectVars(Clauses);
37      Clauses   := saturate(Clauses);
38      if ({ } in Clauses) {
39          return false;
40      }
41      Literals  := { };
42      for (p in Variables) {
43          if (exists(C in Clauses |
44              p in C && C <= {complement(l) : l in Literals} + {p}))
45              ) {
46              Literals += { p };
47          } else {
48              Literals += { !p };
49          }
50      }
51      return Literals;
52  };

```

Figure 4.11: Die Funktion `findValuation`.

Als nächstes diskutieren wir die Implementierung der Funktion `findValuation`, die in Abbildung 4.11 gezeigt ist. Diese Funktion erhält als Eingabe eine Menge **Clauses** von Klauseln. Falls diese Menge widersprüchlich ist, soll die Funktion das Ergebnis **false** zurück geben. Andernfalls soll eine aussagenlogische Belegung \mathcal{I} berechnet werden, unter der alle Klauseln aus der Menge **Clauses** erfüllt sind. Im Detail arbeitet die Funktion `findValuation` wie folgt.

1. Zunächst berechnen wir in Zeile 36 die Menge aller aussagenlogischen Variablen, die in der Menge **Clauses** auftreten. Wir benötigen diese Menge, denn wir müssen diese Variablen ja auf die Menge $\{\mathbf{true}, \mathbf{false}\}$ abbilden.
2. In Zeile 37 saturieren wir die Menge **Clauses** und berechnen alle Klauseln, die aus der ursprünglich gegebenen Menge von Klauseln mit Hilfe der Schnitt-Regel hergeleitet werden können. Hier können zwei Fälle auftreten:
 - (a) Falls die leere Klausel hergeleitet werden kann, dann ist die ursprünglich gegebene Menge von Klauseln widersprüchlich und wir geben als Ergebnis an Stelle einer Belegung den Wert **false** zurück, denn eine widersprüchliche Menge von Klauseln ist sicher nicht erfüllbar.

- (b) Andernfalls berechnen wir nun eine aussagenlogische Belegung, unter der alle Klauseln aus der Menge **Clauses** wahr werden. Zu diesem Zweck berechnen wir zunächst eine Menge von Literalen **Literals**. Die Idee ist dabei, dass wir die Variable **p** genau dann in die Menge **Literals** aufnehmen, wenn die gesuchte Belegung \mathcal{I} die Variable **p** zu **true** auswertet. Andernfalls nehmen wir an Stelle von **p** das Literal $\neg p$ in der Menge **Literals** auf. Als Ergebnis geben wir daher in Zeile 50 die Menge **Literals** zurück. Die gesuchte aussagenlogische Belegung \mathcal{I} kann dann gemäß der Formel

$$\mathcal{I}(p) = \begin{cases} \text{true} & \text{falls } p \in \text{Literals} \\ \text{false} & \text{falls } !p \in \text{Literals} \end{cases}$$

berechnet werden.

3. Die Berechnung der Menge **Literals** erfolgt nun über eine **for**-Schleife. Dabei ist die Idee, dass wir für eine aussagenlogische Variable **p** genau dann das Literal **p** zu der Menge **Literals** hinzufügen, wenn die Belegung \mathcal{I} die Variable **p** auf **true** abbilden muss um die Klauseln zu erfüllen. Andernfalls fügen wir stattdessen das Literal **!p** zu dieser Menge hinzu.

Die Bedingung dafür ist wie folgt: Angenommen, wir haben bereits Werte für die Variablen p_1, \dots, p_n in der Menge **Literals** gefunden. Die Werte dieser Variablen seien durch die Literale l_1, \dots, l_n in der Menge **Literals** wie folgt festgelegt: Wenn $l_i = p_i$ ist, dann gilt $\mathcal{I}(p_i) = \text{true}$ und falls $l_i = \neg p_i$ gilt, so haben wir $\mathcal{I}(p_i) = \text{false}$. Nehmen wir nun weiter an, dass eine Klausel C in der Menge **Clauses** existiert, so dass

$$C \subseteq \{\overline{l_1}, \dots, \overline{l_n}, p\} \quad \text{und} \quad p \in C$$

gilt. Wenn $\mathcal{I}(C) = \text{true}$ gelten soll, dann muss $\mathcal{I}(p) = \text{true}$ gelten, denn nach Konstruktion von \mathcal{I} gilt

$$\mathcal{I}(\overline{l_i}) = \text{false} \quad \text{für alle } i \in \{1, \dots, n\}$$

und damit ist **p** das einzige Literal in der Klausel C , das wir mit Hilfe der Belegung \mathcal{I} überhaupt noch wahr machen können. In diesem Fall fügen wir also das Literal **p** in die Menge **Literals** ein.

Der entscheidende Punkt ist nun der Nachweis, dass die Funktion **findValuation** in dem Falle, dass in Zeile 39 nicht der Wert **false** zurück gegeben wird, eine aussagenlogische Belegung \mathcal{I} berechnet, bei der alle Klauseln aus der Menge **Clauses** den Wert **true** erhalten. Um diesen Nachweis zu erbringen, nummerieren wir die aussagenlogischen Variablen, die in der Menge **Clauses** auftreten, in derselben Reihenfolge durch, in der diese Variablen in der **for**-Schleife in Zeile 42 betrachtet werden. Wir bezeichnen diese Variablen als

$$p_1, p_2, p_3, \dots, p_k$$

und zeigen durch Induktion nach n , dass nach n Durchläufen der Schleife für jede Klausel $D \in \text{Clauses}$, in der nur die Variablen p_1, \dots, p_n vorkommen,

$$\mathcal{I}(D) = \text{true}$$

gilt.

I.A.: $n = 0$.

Die einzige Klausel, in der überhaupt keine Variablen vorkommen, ist die leere Klausel. Da wir aber vorausgesetzt haben, dass **Clauses** die leere Klausel nicht enthält, ist die zu zeigende Behauptung trivialerweise wahr.

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

Wir setzen nun voraus, dass die Behauptung vor dem $(n+1)$ -ten Durchlauf der **for**-Schleife gilt und haben zu zeigen, dass die Behauptung dann auch nach diesem Durchlauf erfüllt ist. Sei dazu D eine Klausel, in der nur die Variablen p_1, \dots, p_n, p_{n+1} vorkommen. Die Klausel ist dann eine Teilmenge einer Menge der Form

$$\{l_1, \dots, l_n, l_{n+1}\}, \quad \text{wobei } l_i \in \{p_i, \neg p_i\} \text{ für alle } i \in \{1, \dots, n+1\} \text{ gilt.}$$

Nun gibt es mehrere Möglichkeiten, die wir getrennt untersuchen.

- (a) Es gibt ein $i \in \{1, \dots, n\}$, so dass $l_i \in D$ und $\mathcal{I}(l_i) = \mathbf{true}$ ist.

Da eine Klausel als Disjunktion ihrer Literale aufgefasst wird, gilt dann auch $\mathcal{I}(D) = \mathbf{true}$ unabhängig davon, ob $\mathcal{I}(p_{n+1})$ den Wert **true** oder **false** hat.

- (b) Für alle $i \in \{1, \dots, n\}$ mit $l_i \in D$ gilt $\mathcal{I}(l_i) = \mathbf{false}$ und es gilt $l_{n+1} = p_{n+1}$.

Dann gilt für die Klausel D gerade die Bedingung

$$C \subseteq \{\overline{l_1}, \dots, \overline{l_n}\} \cup \{p_{n+1}\}$$

und daher wird in Zeile 44 der Funktion `findValuation` das Literal p_{n+1} zu der Menge **Literals** hinzugefügt. Nach Definition der Belegung \mathcal{I} , die von der Funktion `findValuation` zurück gegeben wird, heißt dies gerade, dass

$$\mathcal{I}(p_{n+1}) = \mathbf{true}$$

ist und dann gilt natürlich auch $\mathcal{I}(D) = \mathbf{true}$.

- (c) Für alle $i \in \{1, \dots, n\}$ mit $l_i \in d$ gilt $\mathcal{I}(l_i) = \mathbf{false}$ und es gilt $l_{n+1} = \neg p_{n+1}$.

An dieser Stelle ist eine weitere Fall-Unterscheidung notwendig.

- i. Es gibt eine weitere Klausel C in der Menge **Clauses**, so dass

$$C \subseteq \{\overline{l_1}, \dots, \overline{l_n}\} \cup \{p_{n+1}\}$$

gilt. Hier sieht es zunächst so aus, als ob wir ein Problem hätten, denn in diesem Fall würde um die Klausel C wahr zu machen das Literal p_{n+1} zur Menge **Literals** hinzugefügt und damit wäre zunächst $\mathcal{I}(p_{n+1}) = \mathbf{true}$ und damit $\mathcal{I}(\neg p_{n+1}) = \mathbf{false}$, woraus insgesamt $\mathcal{I}(D) = \mathbf{false}$ folgern würde. In diesem Fall würden sich die Klauseln C und D in der Form

$$C = C' \cup \{p_{n+1}\}, \quad D = D' \cup \{\neg p_{n+1}\}$$

schreiben lassen, wobei

$$C' \subseteq \{\overline{l} \mid l \in \mathbf{Literals}\} \quad \text{und} \quad D' \subseteq \{\overline{l} \mid l \in \mathbf{Literals}\}$$

gelten würde. Daraus würde sowohl

$$\mathcal{I}(C') = \mathbf{false} \quad \text{als auch} \quad \mathcal{I}(D') = \mathbf{false}$$

folgen und das würde auch

$$\mathcal{I}(C' \cup D') = \mathbf{false} \tag{*}$$

implizieren. Die entscheidende Beobachtung ist nun, dass die Klausel $C' \cup D'$ mit Hilfe der Schnitt-Regel aus den beiden Klauseln

$$C = C' \cup \{p_{n+1}\}, \quad D = D' \cup \{\neg p_{n+1}\},$$

gefolgert werden kann. Das heißt dann aber, dass die Klausel $C' \cup D'$ ein Element der Menge **Clauses** sein muss, denn die Menge **Clauses** ist ja saturiert! Da die Klausel $C' \cup D'$ außerdem nur die aussagenlogischen Variablen p_1, \dots, p_n enthält, gilt nach Induktions-Voraussetzung

$$\mathcal{I}(C' \cup D') = \mathbf{true}.$$

Dies steht aber im Widerspruch zu (*). Dieser Widerspruch zeigt, dass es keine Klausel $C \in \mathbf{Clauses}$ mit

$$C \subseteq \{\overline{l} \mid l \in \mathbf{Literals}\} \cup \{p_{n+1}\} \quad \text{und} \quad p_{n+1} \in C$$

geben kann und damit tritt der hier untersuchte Fall gar nicht auf.

- ii. Es gibt keine Klausel C in der Menge **Clauses**, so dass

$$C \subseteq \{\overline{l} \mid l \in \mathbf{Literals}\} \cup \{p_{n+1}\} \quad \text{und} \quad p_{n+1} \in C$$

gilt. In diesem Fall wird das Literal $\neg p_{n+1}$ zur Menge **Literals** hinzugefügt und damit gilt zunächst $\mathcal{I}(p_{n+1}) = \mathbf{false}$ und folglich $\mathcal{I}(\neg p_{n+1}) = \mathbf{true}$, woraus schließlich $\mathcal{I}(D) = \mathbf{true}$ folgt.

Wir sehen, dass der erste Fall der vorherigen Fall-Unterscheidung nicht auftritt und dass im zweiten Fall $\mathcal{I}(D) = \mathbf{true}$ gilt, womit wir insgesamt $\mathcal{I}(D) = \mathbf{true}$ gezeigt haben. Damit ist der Induktionsschritt abgeschlossen.

Da jede Klausel $C \in \mathbf{Clauses}$ nur eine endliche Anzahl von Variablen enthält, haben wir insgesamt gezeigt, dass für alle diese Klauseln $\mathcal{I}(C) = \mathbf{true}$ gilt. \square

Beweis der Widerlegungs-Vollständigkeit der Schnitt-Regel: Wir haben nun alles Material zusammen um zeigen zu können, dass die Schnitt-Regel widerlegungs-vollständig ist. Wir nehmen also an, dass M eine endliche Menge von Klauseln ist, die nicht erfüllbar ist, was wir als

$$M \models \perp$$

schreiben. Wir rufen die Funktion `findValuation` mit dieser Menge M als Argument auf. Jetzt gibt es zwei Möglichkeiten:

1. Fall: Die Funktion `findValuation` liefert als Ergebnis `false`. Nach Konstruktion der Funktionen `findValuation` `saturate` tritt dieser Fall nur ein, wenn sich die leere Klausel $\{\}$ aus den Klauseln der Menge M mit Hilfe der Schnitt-Regel herleiten lässt. Dann haben wir also

$$M \vdash \{\},$$

was zu zeigen war.

2. Fall: Die Funktion `findValuation` liefert als Ergebnis eine aussagenlogische Belegung \mathcal{I} . Bei der Diskussion der Funktion `findValuation` haben wir gezeigt, dass für alle Klauseln $D \in \mathbf{Clauses}$

$$\mathcal{I}(D) = \mathbf{true}$$

gilt. Die Menge M ist aber eine Teilmenge der Menge $\mathbf{Clauses}$ und damit sehen wir, dass die Menge M erfüllbar ist. Dies steht im Widerspruch zu $M \models \perp$ und folglich kann der zweite Fall gar nicht auftreten.

Folglich liefert die Funktion `findValuation` für eine unerfüllbare Menge von Klauseln immer das Ergebnis `false`, was impliziert, dass $M \vdash \{\}$ gilt. \square

4.6 Das Verfahren von Davis und Putnam

In der Praxis stellt sich oft die Aufgabe, für eine gegebene Menge von Klauseln K eine Belegung \mathcal{I} der Variablen zu berechnen, so dass

$$\mathbf{eval}(k, \mathcal{I}) = \mathbf{true} \quad \text{für alle } k \in K$$

gilt. In diesem Fall sagen wir auch, dass die Belegung \mathcal{I} eine *Lösung* der Klausel-Menge K ist. Im letzten Abschnitt haben wir bereits die Prozedur `findValuation` kennengelernt, mit der wir eine solche Belegung berechnen könnten. Bedauerlicherweise ist diese Prozedur für eine praktische Anwendung nicht effizient genug. Wir werden daher in diesem Abschnitt ein Verfahren vorstellen, mit dem die Berechnung einer Lösung einer aussagenlogischen Klausel-Menge auch in der Praxis möglich ist. Dieses Verfahren geht auf Davis und Putnam [DP60, DLL62] zurück. Verfeinerungen dieses Verfahrens werden beispielsweise eingesetzt, um die Korrektheit digitaler elektronischer Schaltungen nachzuweisen.

Um das Verfahren zu motivieren überlegen wir zunächst, bei welcher Form der Klausel-Menge K unmittelbar klar ist, ob es eine Belegung gibt, die K löst und wie diese Belegung aussieht. Betrachten wir dazu ein Beispiel:

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

Die Klausel-Menge K_1 entspricht der aussagenlogischen Formel

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

Daher ist K_1 lösbar und die Belegung

$$\mathcal{I} = \{ \langle p, \text{true} \rangle, \langle q, \text{false} \rangle, \langle r, \text{true} \rangle, \langle s, \text{false} \rangle, \langle t, \text{false} \rangle \}$$

ist eine Lösung. Betrachten wir ein weiteres Beispiel:

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

Diese Klausel-Menge entspricht der Formel

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

Offensichtlich ist K_2 unlösbar. Als letztes Beispiel betrachten wir

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

Diese Klausel-Menge kodiert die Formel

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

Offenbar ist K_3 ebenfalls unlösbar, denn eine Lösung \mathcal{I} müsste p gleichzeitig wahr und falsch machen. Wir nehmen die an den letzten drei Beispielen gemachten Beobachtungen zum Anlass für zwei Definitionen.

Definition 23 (Unit-Klausel) Eine Klausel k heißt **Unit-Klausel**, wenn k nur aus einem Literal besteht. Es gilt dann entweder

$$k = \{p\} \quad \text{oder} \quad k = \{ \neg p \}$$

für eine geeignete Aussage-Variable p . ◇

Definition 24 (Triviale Klausel-Mengen) Eine Klausel-Menge K heißt **trivial** wenn einer der beiden folgenden Fälle vorliegt.

1. K enthält die leere Klausel: $\{ \} \in K$.

In diesem Fall ist K offensichtlich unlösbar.

2. K enthält nur Unit-Klauseln mit **verschiedenen** Aussage-Variablen. Bezeichnen wir die Menge der aussagenlogischen Variablen mit \mathcal{P} , so schreibt sich diese Bedingung als

$$\forall k \in K : \text{card}(k) = 1 \quad \text{und} \quad \forall p \in \mathcal{P} : \neg(\{p\} \in K \wedge \{ \neg p \} \in K).$$

In diesem Fall ist die aussagenlogische Belegung

$$\mathcal{I} = \{ \langle p, \text{true} \rangle \mid \{p\} \in K \} \cup \{ \langle p, \text{false} \rangle \mid \{ \neg p \} \in K \}$$

eine Lösung von K . ◇

Wie können wir nun eine Menge von Klauseln so vereinfachen, dass die Menge schließlich nur noch aus Unit-Klauseln besteht? Es gibt drei Möglichkeiten, Klauselmengen zu vereinfachen:

1. Schnitt-Regel,
2. Subsumption und
3. Fallunterscheidung.

Wir betrachten diese Möglichkeiten jetzt der Reihe nach.

4.6.1 Vereinfachung mit der Schnitt-Regel

Eine typische Anwendung der Schnitt-Regel hat die Form:

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

Die hierbei erzeugte Klausel $k_1 \cup k_2$ wird in der Regel mehr Literale enthalten als die Prämissen $k_1 \cup \{p\}$ und

$\{\neg p\} \cup k_2$. Enthält die Klausel $k_1 \cup \{p\}$ insgesamt $m + 1$ Literale und enthält die Klausel $\{\neg p\} \cup k_2$ insgesamt $n + 1$ Literale, so kann die Konklusion $k_1 \cup k_2$ bis zu $m + n$ Literale enthalten. Natürlich können es auch weniger Literale sein, und zwar dann, wenn es Literale gibt, die sowohl in k_1 als auch in k_2 auftreten. Im allgemeinen ist $m + n$ größer als $m + 1$ und als $n + 1$. Die Klauseln wachsen nur dann sicher nicht, wenn entweder $n = 0$ oder $m = 0$ ist. Dieser Fall liegt vor, wenn einer der beiden Klauseln nur aus einem Literal besteht und folglich eine **Unit-Klausel** ist. Da es unser Ziel ist, die Klausel-Mengen zu vereinfachen, lassen wir nur solche Anwendungen der Schnitt-Regel zu, bei denen eine der Klauseln eine Unit-Klausel ist. Solche Schnitte bezeichnen wir als **Unit-Schnitte**. Um alle mit einer gegebenen Unit-Klausel $\{l\}$ möglichen Schnitte durchführen zu können, definieren wir eine Funktion

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

so, dass für eine Klausel-Menge K und ein Literal l die Funktion $\text{unitCut}(K, l)$ die Klausel-Menge K soweit wie möglich mit Unit-Schnitten mit der Klausel $\{l\}$ vereinfacht:

$$\text{unitCut}(K, l) = \left\{ k \setminus \{\overline{l}\} \mid k \in K \right\}.$$

Beachten Sie, dass die Menge $\text{unitCut}(K, l)$ genauso viele Klauseln enthält wie die Menge K . Allerdings sind die Klauseln aus der Menge K , die das Literal \overline{l} enthalten, verkürzt worden.

4.6.2 Vereinfachung durch Subsumption

Das Prinzip der Subsumption demonstrieren wir zunächst an einem Beispiel. Wir betrachten

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

Offenbar impliziert die Klausel $\{p\}$ die Klausel $\{p, q, \neg r\}$, denn immer wenn $\{p\}$ erfüllt ist, ist automatisch auch $\{q, p, \neg r\}$ erfüllt. Das liegt daran, dass

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

gilt. Allgemein sagen wir, dass eine Klausel k von einer Unit-Klausel u **subsumiert** wird, wenn

$$u \subseteq k$$

gilt. Ist K eine Klausel-Menge mit $k \in K$ und $u \in K$ und wird k durch u subsumiert, so können wir K durch Unit-Subsumption zu $K - \{k\}$ vereinfachen, wir können also die Klausel k aus K löschen. Allgemein definieren wir eine Funktion

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

die eine gegebene Klauselmengemenge K , welche die Unit-Klausel $\{l\}$ enthält, mittels Subsumption dadurch vereinfacht, dass alle durch $\{l\}$ subsumierten Klauseln aus K gelöscht werden. Die Unit-Klausel $\{l\}$ selbst behalten wir natürlich. Daher definieren wir:

$$\text{subsume}(K, l) := (K \setminus \{k \in K \mid l \in k\}) \cup \{\{l\}\} = \{k \in K \mid l \notin k\} \cup \{\{l\}\}.$$

In der obigen Definition muss $\{l\}$ in das Ergebnis eingefügt werden, weil die Menge $\{k \in K \mid l \notin k\}$ die Unit-Klausel $\{l\}$ nicht enthält.

4.6.3 Vereinfachung durch Fallunterscheidung

Ein Kalkül, der nur mit Unit-Schnitten und Subsumption arbeitet, ist nicht widerlegungs-vollständig. Wir brauchen daher eine weitere Möglichkeit, Klausel-Mengen zu vereinfachen. Eine solche Möglichkeit bietet das Prinzip der **Fallunterscheidung**. Dieses Prinzip basiert auf dem folgenden Satz.

Satz 25 *Ist K eine Menge von Klauseln und ist p eine aussagenlogische Variable, so ist K genau dann erfüllbar, wenn $K \cup \{\{p\}\}$ oder $K \cup \{\{\neg p\}\}$ erfüllbar ist.*

Beweis: Ist K erfüllbar durch eine Belegung \mathcal{I} , so gibt es für $\mathcal{I}(p)$ zwei Möglichkeiten: Falls $\mathcal{I}(p) = \mathbf{true}$ ist, ist damit auch die Menge $K \cup \{\{p\}\}$ erfüllbar, andernfalls ist $K \cup \{\{\neg p\}\}$ erfüllbar.

Da K sowohl eine Teilmenge von $K \cup \{\{p\}\}$ als auch von $K \cup \{\{\neg p\}\}$ ist, ist klar, dass K erfüllbar ist, wenn eine dieser Mengen erfüllbar sind. \square

Wir können nun eine Menge K von Klauseln dadurch vereinfachen, dass wir eine aussagenlogische Variable p wählen, die in K vorkommt. Anschließend bilden wir die Mengen

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

und untersuchen rekursiv ob K_1 erfüllbar ist. Falls wir eine Lösung für K_1 finden, ist dies auch eine Lösung für die ursprüngliche Klausel-Menge K und wir haben unser Ziel erreicht. Andernfalls untersuchen wir rekursiv ob K_2 erfüllbar ist. Falls wir nun eine Lösung finden, ist dies auch eine Lösung von K und wenn wir weder für K_1 noch für K_2 eine Lösung finden, dann kann auch K keine Lösung haben, denn jede Lösung \mathcal{I} von K muss die Variable P entweder wahr oder falsch machen. Die rekursive Untersuchung von K_1 bzw. K_2 ist leichter als die Untersuchung von K , weil wir ja in K_1 und K_2 mit den Unit-Klausel $\{p\}$ bzw. $\{\neg p\}$ zunächst Unit-Subsumptionen und anschließend Unit-Schnitte durchführen können.

4.6.4 Der Algorithmus

Wir können jetzt den Algorithmus von Davis und Putnam skizzieren. Gegeben sei eine Menge K von Klauseln. Gesucht ist dann eine Lösung von K . Wir suchen also eine Belegung \mathcal{I} , so dass gilt:

$$\mathcal{I}(k) = \text{true} \quad \text{für alle } k \in K.$$

Das Verfahren von Davis und Putnam besteht nun aus den folgenden Schritten.

1. Führe alle Unit-Schnitte und Unit-Subsumptionen aus, die mit Klauseln aus K möglich sind.
2. Falls K nun trivial ist, sind wir fertig.
3. Andernfalls wählen wir eine aussagenlogische Variable p , die in K auftritt.

- (a) Jetzt versuchen wir rekursiv die Klausel-Menge

$$K \cup \{\{p\}\}$$

zu lösen. Falls diese gelingt, haben wir eine Lösung von K .

- (b) Andernfalls versuchen wir die Klausel-Menge

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

zu lösen. Wenn auch dies fehlschlägt, ist K unlösbar, andernfalls haben wir eine Lösung von K .

Für die Implementierung ist es zweckmäßig, die beiden oben definierten Funktionen *unitCut()* und *subsume()* zusammen zu fassen. Wir definieren eine Funktion

$$\text{reduce} : 2^{\mathcal{K}} \times \mathcal{L} \rightarrow 2^{\mathcal{K}}$$

wie folgt:

$$\text{reduce}(K, l) = \{k \setminus \{\bar{l}\} \mid k \in K \wedge \bar{l} \in k\} \cup \{k \in K \mid \bar{l} \notin k \wedge l \notin k\} \cup \{\{l\}\}.$$

Die Menge enthält also einerseits die Ergebnisse von Schnitten mit der Unit-Klausel $\{l\}$ und andererseits nur noch die Klauseln k , die mit l nichts zu tun haben weil weder $l \in k$ noch $\bar{l} \in k$ gilt. Außerdem fügen wir auch noch die Unit-Klausel $\{l\}$ hinzu. Dadurch erreichen wir, dass die beiden Mengen K und $\text{reduce}(K, l)$ logisch äquivalent sind, wenn wir diese Mengen als Formeln in konjunktiver Normalform interpretieren.

4.6.5 Ein Beispiel

Zur Veranschaulichung demonstrieren wir das Verfahren von Davis und Putnam an einem Beispiel. Die Menge K sei wie folgt definiert:

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

Wir zeigen nun mit dem Verfahren von Davis und Putnam, dass K nicht lösbar ist. Da die Menge K keine Unit-Klauseln enthält, ist im ersten Schritt nichts zu tun. Da K nicht trivial ist, sind wir noch nicht fertig. Also gehen wir jetzt zu Schritt 3 und wählen eine aussagenlogische Variable, die in K auftritt. An dieser Stelle ist es sinnvoll eine Variable zu wählen, die in möglichst vielen Klauseln von K auftritt. Wir wählen daher die aussagenlogische Variable p .

1. Zunächst bilden wir die Menge

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

und versuchen, diese Menge zu lösen. Dazu bilden wir

$$K_1 := \text{reduce}(K_0, p) = \left\{ \{r, \neg t\}, \{r, s\}, \{\neg r, q\}, \{\neg q, s, \neg r\}, \{\neg r, \neg s\}, \{\neg s\}, \{p\} \right\}.$$

Die Klausel-Menge K_1 enthält die Unit-Klausel $\{\neg s\}$, so dass wir als nächstes mit dieser Klausel reduzieren können:

$$K_2 := \text{reduce}(K_1, \neg s) = \left\{ \{r, \neg t\}, \{r\}, \{\neg r, q\}, \{\neg q, \neg r\}, \{\neg s\}, \{p\} \right\}.$$

Hier haben wir nun die neue Unit-Klausel $\{r\}$, mit der wir als nächstes reduzieren:

$$K_3 := \text{reduce}(K_2, r) = \left\{ \{r\}, \{q\}, \{\neg q\}, \{\neg s\}, \{p\} \right\}$$

Da K_3 die Unit-Klausel $\{q\}$ enthält, reduzieren wir jetzt mit q :

$$K_4 := \text{reduce}(K_3, q) = \left\{ \{r\}, \{q\}, \{\}, \{\neg s\}, \{p\} \right\}.$$

Die Klausel-Menge K_4 enthält die leere Klausel und ist damit unlösbar.

2. Also bilden wir jetzt die Menge

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

und versuchen, diese Menge zu lösen. Dazu bilden wir

$$K_6 = \text{reduce}(K_5, \neg p) = \left\{ \{q, s\}, \{r, s\}, \{\neg s\}, \{\neg q, s\}, \{\neg r, \neg s\}, \{\neg p\} \right\}.$$

Die Menge K_6 enthält die Unit-Klausel $\{\neg s\}$. Wir bilden daher

$$K_7 = \text{reduce}(K_6, \neg s) = \left\{ \{q\}, \{r\}, \{\neg s\}, \{\neg q\}, \{\neg p\} \right\}.$$

Die Menge K_7 enthält die neue Unit-Klausel $\{q\}$, mit der wir als nächstes reduzieren:

$$K_8 = \text{reduce}(K_7, q) = \left\{ \{q\}, \{r\}, \{\neg s\}, \{\}, \{\neg p\} \right\}.$$

Da K_8 die leere Klausel enthält, ist K_8 und damit auch die ursprünglich gegebene Menge K unlösbar.

Bei diesem Beispiel hatten wir Glück, denn wir mussten nur eine einzige Fallunterscheidung durchführen. Bei komplexeren Beispielen ist es häufig so, dass wir mehrere Fallunterscheidungen durchführen müssen.

4.6.6 Implementierung des Algorithmus von Davis und Putnam

Wir zeigen jetzt die Implementierung der Prozedur `davisPutnam`, mit der die Frage, ob eine Menge von Klauseln erfüllbar ist, beantwortet werden kann. Die Implementierung ist in Abbildung 4.12 auf Seite 91 gezeigt. Die Prozedur erhält zwei Argumente: Die Mengen **Clauses** und **Literals**. Hier ist **Clauses** eine Menge von Klauseln und **Literals** ist eine Menge von Literalen. Falls die Vereinigung dieser beiden Mengen erfüllbar ist, so liefert der Aufruf

`davisPutnam(Clauses, Literals)`

eine Menge von Unit-Klauseln R , so dass jede Belegung \mathcal{I} , die alle Unit-Klauseln aus R erfüllt, auch alle Klauseln aus der Menge **clauses** erfüllt. Falls die Menge **clauses** nicht erfüllbar ist, liefert der Aufruf

davisPutnam(Clauses, Literals)

als Ergebnis die Menge $\{\{\}\}$ zurück, denn die leere Klausel repräsentiert die unerfüllbare Formel \perp .

Sie fragen sich vielleicht, wozu wir in der Prozedur **davisPutnam** die Menge **Literals** brauchen. Der Grund ist, dass wir uns bei den rekursiven Aufrufen merken müssen, welche Literale wir schon benutzt haben. Diese Literale sammeln wir in der Menge **literals**.

Die in Abbildung 4.12 gezeigte Implementierung funktioniert wie folgt:

1. In Zeile 2 reduzieren wir mit Hilfe der Methode **saturate** solange wie möglich die gegebene Klausel-Menge **Clauses** mit Hilfe von Unit-Schnitten und entfernen alle Klauseln, die durch Unit-Klauseln subsumiert werden.
2. Anschließend testen wir in Zeile 3, ob die so vereinfachte Klausel-Menge die leere Klausel enthält und geben in diesem Fall als Ergebnis die Menge $\{\{\}\}$ zurück.
3. Dann testen wir in Zeile 6, ob bereits alle Klauseln C aus der Menge **Clauses** Unit-Klauseln sind. Wenn dies so ist, dann ist die Menge **Clauses** trivial und wir geben diese Menge als Ergebnis zurück.
4. Andernfalls wählen wir in Zeile 9 ein Literal l aus der Menge **Clauses**, dass wir noch nicht benutzt haben. Wir untersuchen dann in Zeile 10 rekursiv, ob die Menge

$\text{Clauses} \cup \{\{l\}\}$

lösbar ist. Dabei gibt es zwei Fälle:

- (a) Falls diese Menge lösbar ist, geben wir die Lösung dieser Menge als Ergebnis zurück.
- (b) Sonst prüfen wir rekursiv, ob die Menge

$\text{Clauses} \cup \{\{\bar{l}\}\}$

lösbar ist. Ist diese Menge lösbar, so ist diese Lösung auch eine Lösung der Menge **Clauses** und wir geben diese Lösung zurück. Ist die Menge unlösbar, dann muss auch die Menge **Clauses** unlösbar sein.

```

1  davisPutnam := procedure(Clauses, Literals) {
2      Clauses := saturate(Clauses);
3      if ({ } in Clauses) {
4          return { { } };
5      }
6      if (forall (C in Clauses | #C == 1)) {
7          return Clauses;
8      }
9      l := selectLiteral(Clauses, Literals);
10     notL := negateLiteral(l);
11     S := davisPutnam(Clauses + { {l} }, Literals + { l });
12     if (S != { { } }) {
13         return S; // solution found
14     }
15     return davisPutnam(Clauses + { {notL} }, Literals + { notL });
16 };

```

Figure 4.12: Die Prozedur **davisPutnam**.

Wir diskutieren nun die Hilfsprozeduren, die bei der Implementierung der Prozedur **davisPutnam** verwendet wurden. Als erstes besprechen wir die Funktion **saturate**. Diese Prozedur erhält eine Menge S von Klauseln als Eingabe und führt alle möglichen Unit-Schnitte und Unit-Subsumptionen durch. Die Prozedur **saturate** ist in Abbildung 4.13 auf Seite 92 gezeigt.

```

1  saturate := procedure(S) {
2      Units := { C : C in S | #C == 1 };
3      Used  := {};
4      while (Units != {}) {
5          Unit := arb(Units);
6          Used += { Unit };
7          l    := arb(Unit);
8          S    := reduce(S, l);
9          Units := { C : C in S | #C == 1 && !(C in Used) };
10     }
11     return S;
12 };

```

Figure 4.13: Die Prozedur **saturate**.

Die Implementierung von **saturate** funktioniert wie folgt:

1. Zunächst berechnen wir in Zeile 2 die Menge **Units** aller Unit-Klauseln.
2. Dann initialisieren wir in Zeile 3 die Menge **Used** als die leere Menge. In dieser Menge merken wir uns, welche Unit-Klauseln wir schon für Unit-Schnitte und Subsumptionen benutzt haben.
3. Solange die Menge **Units** der Unit-Klauseln nicht leer ist, wählen wir in Zeile 5 mit Hilfe der Funktion **arb** eine beliebige Unit-Klausel **Unit** aus der Menge **Units** aus.
4. In Zeile 6 fügen wir die Klausel **Unit** zu der Menge **Used** der benutzten Klausel hinzu.
5. In Zeile 7 extrahieren mit der Funktion **arb** das Literal **l** der Klausel **Unit**. Die Funktion **arb** liefert ein beliebiges Element der Menge zurück, das dieser Funktion als Argument übergeben wird. Enthält diese Menge nur ein Element, so wird also dieses Element zurück gegeben.
6. In Zeile 8 wird die eigentliche Arbeit durch einen Aufruf der Prozedur **reduce** geleistet. Diese Funktion berechnet alle Unit-Schnitte, die mit der Unit-Klausel **{l}** möglich sind und entfernt darüber hinaus alle Klauseln, die durch die Unit-Klausel **{l}** subsumiert werden.
7. Wenn die Unit-Schnitte mit der Unit-Klausel **{l}** berechnet werden, können neue Unit-Klauseln entstehen, die wir in Zeile 9 aufsammeln. Wir sammeln dort aber nur die Unit-Klauseln auf, die wir noch nicht benutzt haben.
8. Die Schleife in den Zeilen 4 – 10 wird nun solange durchlaufen, wie wir Unit-Klauseln finden, die wir noch nicht benutzt haben.
9. Am Ende geben wir die verbliebende Klauselmenge als Ergebnis zurück.

Die dabei verwendete Prozedur **reduce()** ist in Abbildung 4.14 gezeigt. Im vorigen Abschnitt hatten wir die Funktion $reduce(S, l)$, die eine Klausel-Menge S mit Hilfe des Literals l reduziert, als

$$reduce(S, l) = \left\{ C \setminus \{\bar{l}\} \mid C \in S \wedge \bar{l} \in C \right\} \cup \left\{ C \in S \mid \bar{l} \notin C \wedge l \notin C \right\} \cup \left\{ \{l\} \right\}$$

definiert. Die Implementierung setzt diese Definition unmittelbar um.

Die Implementierung des Algorithmus von Davis und Putnam benutzt außer den bisher diskutierten Prozeduren noch zwei weitere Hilfsprozeduren, deren Implementierung in Abbildung 4.15 auf Seite 93 gezeigt wird.

```

1  reduce := procedure(S, l) {
2      notL := negateLiteral(l);
3      return { C - { notL } : C in S | notL in C }
4              + { C : C in S | !(notL in C) && !(l in C) }
5              + { {l} };
6  };

```

Figure 4.14: Die Prozedur `reduce`.

1. Die Prozedur `selectLiteral` wählt ein beliebiges Literal aus einer gegebenen Menge s von Klauseln aus, das außerdem nicht in der Menge `forbidden` von Literalen vorkommen darf, die bereits benutzt worden sind. Dazu werden alle Klauseln, die ja Mengen von Literalen sind, vereinigt. Von dieser Menge wird dann die Menge der bereits benutzten Literalen abgezogen und aus der resultierenden Menge wird mit Hilfe der Funktion `rnd()` zufällig ein Literal ausgewählt. An Stelle der Funktion `rnd()` hätten wir hier auch die Funktion `arb()` benutzen können. Allerdings habe ich experimentell herausgefunden, dass der Davis-Putnam Algorithmus im Allgemeinen deutlich effizienter wird, wenn wir das Literal mit Hilfe der Funktion `rnd()` auswählen.
2. Die Prozedur `negateLiteral` bildet das Komplement \bar{l} eines gegebenen Literals l .

```

1  selectLiteral := procedure(S, Used) {
2      return rnd({} +/ S - Used); // used rnd instead of arb for efficiency
3  };
4  negateLiteral := procedure(l) {
5      match (l) {
6          case !p : return p;
7          case p : return !p;
8      }
9  };

```

Figure 4.15: Die Prozeduren `select` und `negateLiteral`.

Die oben dargestellte Version des Verfahrens von Davis und Putnam lässt sich in vielerlei Hinsicht verbessern. Aus Zeitgründen können wir auf solche Verbesserungen leider nicht weiter eingehen. Der interessierte Leser sei hier auf die Arbeit [MMZ⁺01] verwiesen:

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

4.7 Das 8-Damen-Problem

In diesem Abschnitt zeigen wir, wie bestimmte kombinatorische Problem in aussagenlogische Probleme umformuliert werden können. Diese können dann anschließend mit dem Algorithmus von Davis und Putnam bzw. mit Verbesserungen dieses Algorithmus gelöst werden. Als konkretes Beispiel betrachten wir das 8-Damen-Problem. Dabei geht es darum, 8 Damen so auf einem Schach-Brett aufzustellen, dass keine Dame eine andere Dame schlagen kann. Beim Schach-Spiel kann eine Dame dann eine andere Figur schlagen, wenn diese Figur entweder

- in derselben Zeile,
- in derselben Spalte oder

- in derselben Diagonale

wie die Dame steht. Abbildung 4.16 auf Seite 94 zeigt ein Schachbrett, in dem sich in der dritten Zeile in der vierten Spalte eine Dame befindet. Diese Dame kann auf alle die Felder ziehen, die mit Pfeilen markierte sind, und kann damit Figuren, die sich auf diesen Feldern befinden, schlagen.

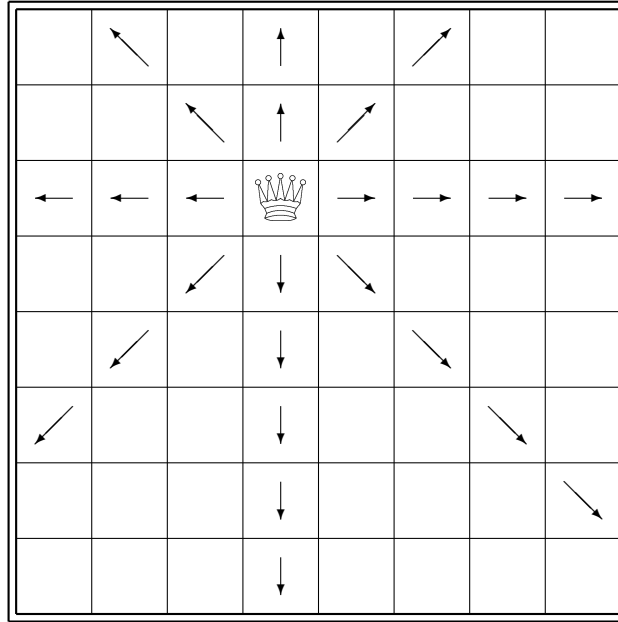


Figure 4.16: Das 8-Damen-Problem.

Als erstes überlegen wir uns, wie wir ein Schach-Brett mit den darauf positionierten Damen aussagenlogisch repräsentieren können. Eine Möglichkeit besteht darin, für jedes Feld eine aussagenlogische Variable einzuführen. Diese Variable drückt aus, dass auf dem entsprechenden Feld eine Dame steht. Wir ordnen diesen Variablen wie folgt Namen zu: Die Variable, die das j -te Feld in der i -ten Zeile bezeichnet, stellen wir durch den Term

$$@Var(i, j) \quad \text{mit } i, j \in \{1, \dots, 8\}$$

dar. Wir nummerieren die Zeilen dabei von oben beginnend von 1 bis 8 durch, während die Spalten von links nach rechts numeriert werden. Abbildung 4.17 auf Seite 95 zeigt die Zuordnung der Variablen zu den Feldern. Zur Vereinfachung habe ich dort statt $@Var(i, j)$ nur $Var(i, j)$ geschrieben.

Als nächstes überlegen wir uns, wie wir die einzelnen Bedingungen des 8-Damen-Problems als aussagenlogische Formeln kodieren können. Letztlich lassen sich alle Aussagen der Form

- “in einer Zeile steht höchstens eine Dame”,
- “in einer Spalte steht höchstens eine Dame”, oder
- “in einer Diagonale steht höchstens eine Dame”

auf dasselbe Grundmuster zurückführen: Ist eine Menge von aussagenlogischen Variablen

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

gegeben, so brauchen wir eine Formel die aussagt, dass **höchstens** eine der Variablen aus V den Wert **true** hat. Das ist aber gleichbedeutend damit, dass für jedes Paar $x_i, x_j \in V$ mit $x_i \neq x_j$ die folgende Formel gilt:

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

Diese Formel drückt aus, dass die Variablen x_i und x_j nicht gleichzeitig den Wert **true** annehmen. Nach den DeMorgan’schen Gesetzen gilt

Var(1,1)	Var(1,2)	Var(1,3)	Var(1,4)	Var(1,5)	Var(1,6)	Var(1,7)	Var(1,8)
Var(2,1)	Var(2,2)	Var(2,3)	Var(2,4)	Var(2,5)	Var(2,6)	Var(2,7)	Var(2,8)
Var(3,1)	Var(3,2)	Var(3,3)	Var(3,4)	Var(3,5)	Var(3,6)	Var(3,7)	Var(3,8)
Var(4,1)	Var(4,2)	Var(4,3)	Var(4,4)	Var(4,5)	Var(4,6)	Var(4,7)	Var(4,8)
Var(5,1)	Var(5,2)	Var(5,3)	Var(5,4)	Var(5,5)	Var(5,6)	Var(5,7)	Var(5,8)
Var(6,1)	Var(6,2)	Var(6,3)	Var(6,4)	Var(6,5)	Var(6,6)	Var(6,7)	Var(6,8)
Var(7,1)	Var(7,2)	Var(7,3)	Var(7,4)	Var(7,5)	Var(7,6)	Var(7,7)	Var(7,8)
Var(8,1)	Var(8,2)	Var(8,3)	Var(8,4)	Var(8,5)	Var(8,6)	Var(8,7)	Var(8,8)

Figure 4.17: Zuordnung der Variablen.

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

und die Klausel auf der rechten Seite dieser Äquivalenz schreibt sich in Mengen-Schreibweise als

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

Die Formel, die für eine Variablen-Menge V ausdrückt, dass keine zwei verschiedenen Variablen gleichzeitig gesetzt sind, kann daher als Klausel-Menge in der Form

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

geschrieben werden. Wir setzen diese Überlegungen in eine SETLX-Prozedur um. Die in Abbildung 4.18 gezeigte Prozedur `atMostOne()` bekommt als Eingabe eine Menge S von aussagenlogischen Variablen. Der Aufruf `atMostOne(S)` berechnet eine Menge von Klauseln. Diese Klauseln sind genau dann wahr, wenn höchstens eine der Variablen aus S den Wert `true` hat.

```

1  atMostOne := procedure(S) {
2      return { { !p, !q } : p in S, q in S | p != q };
3  };

```

Figure 4.18: Die Prozedur `atMostOne`.

Mit Hilfe der Prozedur `atMostOne` können wir nun die Prozedur `atMostOneInRow` implementieren. Der Aufruf

`atMostOneInRow(row, n)`

berechnet für eine gegebene Zeile `row` bei einer Brettgröße von `n` eine Formel, die ausdrückt, dass in der Zeile `row` höchstens eine Dame steht. Abbildung 4.19 zeigt die Prozedur `atMostOneInRow()`: Wir sammeln alle Variablen der durch `row` spezifizierten Zeile in der Menge

$$\{\text{Var}(\text{row}, j) \mid j \in \{1, \dots, n\}\}$$

auf und rufen mit dieser Menge die Funktion `atMostOne()` auf, die das Ergebnis als Menge von Klauseln liefert.

```

1  atMostOneInRow := procedure(row, n) {
2      return atMostOne({ @Var(row, j) : j in [1 .. n] });
3  };

```

Figure 4.19: Die Prozedur `atMostOneInRow`.

Als nächstes berechnen wir eine Formel die aussagt, dass **mindestens** eine Dame in einer gegebenen Spalte steht. Für die erste Spalte hätte diese Formel im Falle eine 8×8 -Bretts die Form

$$\text{Var}(1, 1) \vee \text{Var}(2, 1) \vee \text{Var}(3, 1) \vee \text{Var}(4, 1) \vee \text{Var}(5, 1) \vee \text{Var}(6, 1) \vee \text{Var}(7, 1) \vee \text{Var}(8, 1)$$

und wenn allgemein eine Spalte `c` mit $c \in \{1, \dots, 8\}$ gegeben ist, lautet die Formel

$$\text{Var}(1, c) \vee \text{Var}(2, c) \vee \text{Var}(3, c) \vee \text{Var}(4, c) \vee \text{Var}(5, c) \vee \text{Var}(6, c) \vee \text{Var}(7, c) \vee \text{Var}(8, c).$$

Schreiben wir diese Formel in der Mengenschreibweise als Menge von Klauseln, so erhalten wir

$$\{\{\text{Var}(1, c), \text{Var}(2, c), \text{Var}(3, c), \text{Var}(4, c), \text{Var}(5, c), \text{Var}(6, c), \text{Var}(7, c), \text{Var}(8, c)\}\}.$$

Abbildung 4.20 zeigt eine SETLX-Prozedur, die für eine gegebene Spalte `column` und eine gegebene Brettgröße `n` die entsprechende Klausel-Menge berechnet. Der Schritt, von einer einzelnen Klausel zu einer Menge von Klauseln überzugehen ist notwendig, da unsere Implementierung des Algorithmus von Davis und Putnam ja mit einer Menge von Klauseln arbeitet.

```

1  oneInColumn := procedure(column, n) {
2      return { { @Var(row, column) : row in { 1 .. n } } };
3  };

```

Figure 4.20: Die Prozedur `oneInColumn`.

An dieser Stelle erwarten Sie vielleicht, dass wir noch Formeln angeben die ausdrücken, dass in einer gegebenen Spalte höchstens eine Dame steht und dass in jeder Zeile mindestens eine Dame steht. Solche Formeln sind aber unnötig, denn wenn wir wissen, dass in jeder Spalte mindestens eine Dame steht, so wissen wir bereits, dass auf dem Brett mindestens 8 Damen stehen. Wenn wir nun zusätzlich wissen, dass in jeder Zeile höchstens eine Dame steht, so ist automatisch klar, dass höchstens 8 Damen auf dem Brett stehen. Damit stehen also insgesamt genau 8 Damen auf dem Brett. Dann kann aber in jeder Spalte nur höchstens eine Dame stehen und

genauso muss in jeder Zeile mindestens eine Dame stehen, denn sonst würden wir nicht auf 8 Damen kommen.

Als nächstes überlegen wir uns, wie wir die Variablen, die auf derselben Diagonale stehen, charakterisieren können. Es gibt grundsätzlich zwei verschiedene Arten von Diagonalen: absteigende Diagonalen und aufsteigende Diagonalen. Wir betrachten zunächst die aufsteigenden Diagonalen. Die längste aufsteigende Diagonale, wir sagen dazu auch *Hauptdiagonale*, besteht im Fall eines 8×8 -Bretts aus den Variablen

$$\text{Var}(8, 1), \text{Var}(7, 2), \text{Var}(6, 3), \text{Var}(5, 4), \text{Var}(4, 5), \text{Var}(3, 6), \text{Var}(2, 7), \text{Var}(1, 8).$$

Die Indizes i und j der Variablen $\text{Var}(i, j)$ erfüllen offenbar die Gleichung

$$i + j = 9.$$

Allgemein erfüllen die Indizes der Variablen einer aufsteigenden Diagonale die Gleichung

$$i + j = k,$$

wobei k einen Wert aus der Menge $\{3, \dots, 15\}$ annimmt. Diesen Wert k geben wir nun als Argument bei der Prozedur `atMostOneInUpperDiagonal` mit. Diese Prozedur ist in Abbildung 4.21 gezeigt.

```

1  atMostOneInUpperDiagonal := procedure(k, n) {
2      s := { @Var(r, c) : c in [1..n], r in [1..n] | r + c == k };
3      return atMostOne(s);
4  };

```

Figure 4.21: Die Prozedur `atMostOneInUpperDiagonal`.

Um zu sehen, wie die Variablen einer fallenden Diagonale charakterisiert werden können, betrachten wir die fallende Hauptdiagonale, die aus den Variablen

$$\text{Var}(1, 1), \text{Var}(2, 2), \text{Var}(3, 3), \text{Var}(4, 4), \text{Var}(5, 5), \text{Var}(6, 6), \text{Var}(7, 7), \text{Var}(8, 8)$$

besteht. Die Indizes i und j dieser Variablen erfüllen offenbar die Gleichung

$$i - j = 0.$$

Allgemein erfüllen die Indizes der Variablen einer absteigenden Diagonale die Gleichung

$$i - j = k,$$

wobei k einen Wert aus der Menge $\{-6, \dots, 6\}$ annimmt. Diesen Wert k geben wir nun als Argument bei der Prozedur `atMostOneInLowerDiagonal` mit. Diese Prozedur ist in Abbildung 4.22 gezeigt.

```

1  atMostOneInLowerDiagonal := procedure(k, n) {
2      s := { @Var(r, c) : c in [1..n], r in [1..n] | r - c == k };
3      return atMostOne(s);
4  };

```

Figure 4.22: Die Prozedur `atMostOneInLowerDiagonal`.

Jetzt sind wir in der Lage, unsere Ergebnisse zusammen zu fassen: Wir können eine Menge von Klauseln konstruieren, die das 8-Damen-Problem vollständig beschreibt. Abbildung 4.23 zeigt die Implementierung der Prozedur `allClauses`. Der Aufruf

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

rechnet für ein Schach-Brett der Größe n eine Menge von Klauseln aus, die genau dann erfüllt sind, wenn auf dem Schach-Brett

1. in jeder Zeile höchstens eine Dame steht (Zeile 2),

2. in jeder absteigenden Diagonale höchstens eine Dame steht (Zeile 3),
3. in jeder aufsteigenden Diagonale höchstens eine Dame steht (Zeile 4) und
4. in jeder Spalte mindestens eine Dame steht (Zeile 5).

Die Ausdrücke in den einzelnen Zeilen liefern Mengen, deren Elemente Klausel-Mengen sind. Was wir als Ergebnis brauchen ist aber eine Klausel-Menge und keine Menge von Klausel-Mengen. Daher bilden wir mit dem Operator “+” die Vereinigung dieser Mengen.

```

1  allClauses := procedure(n) {
2      return  +/ { atMostOneInRow(row, n)           : row in {1..n}           }
3              + +/ { atMostOneInLowerDiagonal(k, n) : k in {-(n-2) .. n-2} }
4              + +/ { atMostOneInUpperDiagonal(k, n) : k in {3 .. 2*n - 1} }
5              + +/ { oneInColumn(column, n)         : column in {1 .. n}    };
6  };

```

Figure 4.23: Die Prozedur `allClauses`.

Als letztes zeigen wir in Abbildung 4.24 die Prozedur `solve`, mit der wir das 8-Damen-Problem lösen können. Hierbei ist `printBoard()` eine Prozedur, welche die Lösung in lesbarere Form als Schachbrett ausdrückt. Das funktioniert allerdings nur, wenn ein Font verwendet wird, bei dem alle Zeichen die selbe Breite haben. Diese Prozedur ist der Vollständigkeit halber in Abbildung 4.25 gezeigt, wir wollen die Implementierung aber nicht weiter diskutieren. Das vollständige Programm finden Sie auf meiner Webseite unter dem Namen `queens.stlx`.

```

1  solve := procedure(n) {
2      Clauses := allClauses(n);
3      Solution := davisPutnam(Clauses, {});
4      if (Solution != { {} }) {
5          // print(solution);
6          printBoard(Solution, n);
7      } else {
8          print("The problem is not solvable for $n$ queens!");
9          print("Try to increase the number of queens.");
10     }
11 };

```

Figure 4.24: Die Prozedur `solve`.

Die durch den Aufruf `davisPutnam(Clauses, {})` berechnete Menge `solution` enthält für jede der Variablen `Var(i, j)` entweder die Unit-Klausel `{@Var(i, j)}` (falls auf diesem Feld eine Dame steht) oder aber die Unit-Klausel `{!@Var(i, j)}` (falls das Feld leer bleibt). Eine graphische Darstellung des durch die berechnete Belegung dargestellten Schach-Bretts sehen Sie in Abbildung 4.26.

Das 8-Damen-Problem ist natürlich nur eine spielerische Anwendung der Aussagen-Logik. Trotzdem zeigt es die Leistungsfähigkeit des Algorithmus von Davis und Putnam sehr gut, denn die Menge der Klauseln, die von der Prozedur `allClauses` berechnet wird, füllt unformatiert fünf Bildschirm-Seiten, falls diese eine Breite von 80 Zeichen haben. In dieser Klausel-Menge kommen 64 verschiedene Variablen vor. Der Algorithmus von Davis und Putnam benötigt zur Berechnung einer Belegung, die diese Klauseln erfüllt, auf meinem iMac weniger als fünf Sekunden.

In der Praxis gibt es viele Probleme, die sich in ganz ähnlicher Weise auf die Lösung einer Menge von Klauseln zurückführen lassen. Dazu gehört zum Beispiel das Problem, einen Stundenplan zu erstellen, der

```

1  printBoard := procedure(i, n) {
2      if (i == { {} }) {
3          return;
4      }
5      print( "          " + ((8*n+1) * "-") );
6      for (row in [1..n]) {
7          line := "          |";
8          for (col in [1..n]) {
9              line += "          |";
10         }
11         print(line);
12         line := "          |";
13         for (col in [1..n]) {
14             if ({ @Var(row, col) } in i) {
15                 line += "      Q      |";
16             } else {
17                 line += "          |";
18             }
19         }
20         print(line);
21         line := "          |";
22         for (col in [1..n]) {
23             line += "          |";
24         }
25         print(line);
26         print( "          " + ((8*n+1) * "-") );
27     }
28 };

```

Figure 4.25: Die Prozedur `printBoard()`.

gewissen Nebenbedingungen genügt. Verallgemeinerungen des Stundenplan-Problems werden in der Literatur als *Scheduling-Problemen* bezeichnet. Die effiziente Lösung solcher Probleme ist Gegenstand der aktuellen Forschung.

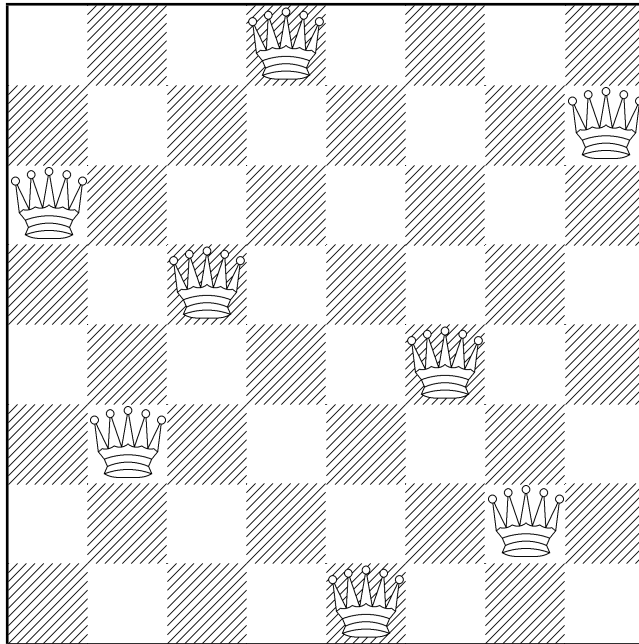


Figure 4.26: Eine Lösung des 8-Damen-Problems.

Chapter 5

Prädikatenlogik

In der Aussagenlogik haben wir die Verknüpfung von elementaren Aussagen mit Junktoren untersucht. Die Prädikatenlogik untersucht zusätzlich auch die Struktur der Aussagen. Dazu werden in der Prädikatenlogik die folgenden zusätzlichen Begriffe eingeführt:

1. Als Bezeichnungen für Objekte werden *Terme* verwendet.
2. Diese Terme werden aus *Variablen* und *Funktions-Zeichen* zusammengesetzt:

$$\text{vater}(x), \quad \text{mutter}(\text{isaac}), \quad x + 7, \quad \dots$$

3. Verschiedene Objekte werden durch *Prädikats-Zeichen* in Relation gesetzt:

$$\text{istBruder}(\text{albert}, \text{vater}(\text{bruno})), \quad x + 7 < x \cdot 7, \quad n \in \mathbb{N}, \quad \dots$$

Die dabei entstehenden Formeln werden als *atomare* Formeln bezeichnet.

4. Atomare Formeln lassen sich durch aussagenlogische Junktoren verknüpfen:

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

5. Schließlich werden *Quantoren* eingeführt, um zwischen *existentiell* und *universell* quantifizierten Variablen unterscheiden zu können:

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

Wir werden im nächsten Abschnitt die Syntax der prädikatenlogischen Formeln festlegen und uns dann im darauf folgenden Abschnitt mit der Semantik dieser Formeln beschäftigen.

5.1 Syntax der Prädikatenlogik

Zunächst definieren wir den Begriff der *Signatur*. Inhaltlich ist das nichts anderes als eine strukturierte Zusammenfassung von Variablen, Funktions- und Prädikats-Zeichen zusammen mit einer Spezifikation der Stelligkeit dieser Zeichen.

Definition 26 (Signatur) Eine *Signatur* ist ein 4-Tupel

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

für das Folgendes gilt:

1. \mathcal{V} ist die Menge der Variablen.
2. \mathcal{F} ist die Menge der Funktions-Zeichen.

3. \mathcal{P} ist die Menge der Prädikats-Zeichen.

4. arity ist eine Funktion, die jedem Funktions- und jedem Prädikats-Zeichen seine **Stelligkeit** zuordnet:

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

Wir sagen, dass das Funktions- oder Prädikats-Zeichen f ein n -stelliges Zeichen ist, falls $\text{arity}(f) = n$ gilt.

5. Da wir in der Lage sein müssen, Variablen, Funktions- und Prädikats-Zeichen unterscheiden zu können, vereinbaren wir, dass die Mengen \mathcal{V} , \mathcal{F} und \mathcal{P} paarweise disjunkt sein müssen:

$$\mathcal{V} \cap \mathcal{F} = \{\}, \quad \mathcal{V} \cap \mathcal{P} = \{\}, \quad \text{und} \quad \mathcal{F} \cap \mathcal{P} = \{\}. \quad \diamond$$

Als Bezeichner für Objekte verwenden wir Ausdrücke, die aus Variablen und Funktions-Zeichen aufgebaut sind. Solche Ausdrücke nennen wir **Terme**. Formal werden diese wie folgt definiert.

Definition 27 (Terme, \mathcal{T}_Σ) Ist $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$ eine Signatur, so definieren wir die Menge der **Σ -Terme** \mathcal{T}_Σ induktiv:

1. Für jede Variable $x \in \mathcal{V}$ gilt $x \in \mathcal{T}_\Sigma$.

2. Ist $f \in \mathcal{F}$ ein n -stelliges Funktions-Zeichen und sind $t_1, \dots, t_n \in \mathcal{T}_\Sigma$, so gilt auch

$$f(t_1, \dots, t_n) \in \mathcal{T}_\Sigma.$$

Falls $c \in \mathcal{F}$ ein 0-stelliges Funktions-Zeichen ist, lassen wir auch die Schreibweise c anstelle von $c()$ zu. In diesem Fall nennen wir c eine **Konstante**. \diamond

Beispiel: Es sei

1. $\mathcal{V} := \{x, y, z\}$ die Menge der Variablen,
2. $\mathcal{F} := \{0, 1, +, -, *\}$ die Menge der Funktions-Zeichen,
3. $\mathcal{P} := \{=, \leq\}$ die Menge der Prädikats-Zeichen,
4. $\text{arity} := \{\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle +, 2 \rangle, \langle -, 2 \rangle, \langle *, 2 \rangle, \langle =, 2 \rangle, \langle \leq, 2 \rangle\}$,
gibt die Stelligkeit der Funktions- und Prädikats-Zeichen an und
5. $\Sigma_{\text{arith}} := \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$ sei eine Signatur.

Dann können wir wie folgt Σ_{arith} -Terme konstruieren:

1. $x, y, z \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
denn alle Variablen sind auch Σ_{arith} -Terme.
2. $0, 1 \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
denn 0 und 1 sind 0-stellige Funktions-Zeichen.
3. $+(0, x) \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
denn es gilt $0 \in \mathcal{T}_{\Sigma_{\text{arith}}}$, $x \in \mathcal{T}_{\Sigma_{\text{arith}}}$ und $+$ ist ein 2-stelliges Funktions-Zeichen.
4. $*+(0, x), 1 \in \mathcal{T}_{\Sigma_{\text{arith}}}$,
denn $+(0, x) \in \mathcal{T}_{\Sigma_{\text{arith}}}$, $1 \in \mathcal{T}_{\Sigma_{\text{arith}}}$ und $*$ ist ein 2-stelliges Funktions-Zeichen.

In der Praxis werden wir für bestimmte zweistellige Funktionen eine Infix-Schreibweise verwenden. Diese ist dann als Abkürzung für die oben definierte Darstellung zu verstehen. \diamond

Als nächstes definieren wir den Begriff der *atomaren Formeln*. Darunter verstehen wir solche Formeln, die man nicht in kleinere Formeln zerlegen kann, atomare Formeln enthalten also weder Junktoren noch Quantoren.

Definition 28 (Atomare Formeln, \mathcal{A}_Σ) Gegeben sei eine Signatur $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$. Die Menge der atomaren Σ -Formeln \mathcal{A}_Σ wird wie folgt definiert: Ist $p \in \mathcal{P}$ ein n -stelliges Prädikats-Zeichen und sind n Σ -Terme t_1, \dots, t_n gegeben, so ist $p(t_1, \dots, t_n)$ eine *atomare Σ -Formel*:

$$p(t_1, \dots, t_n) \in \mathcal{A}_\Sigma.$$

Falls p ein 0-stelliges Prädikats-Zeichen ist, dann schreiben wir auch p anstelle von $p()$. In diesem Fall nennen wir p eine *Aussage-Variable*. \diamond

Beispiel: Setzen wir das letzte Beispiel fort, so können wir sehen, dass

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

eine atomare Σ_{arith} -Formel ist. Beachten Sie, dass wir bisher noch nichts über den Wahrheitswert von solchen Formeln ausgesagt haben. Die Frage, wann eine Formel als wahr oder falsch gelten soll, wird erst im nächsten Abschnitt untersucht. \diamond

Bei der Definition der prädikatenlogischen Formeln ist es notwendig, zwischen sogenannten *gebundenen* und *freien* Variablen zu unterscheiden. Wir führen diese Begriffe zunächst informal mit Hilfe eines Beispiels aus der Analysis ein. Wir betrachten die folgende Identität:

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

In dieser Gleichung treten die Variablen x und y *frei* auf, während die Variable t durch das Integral *gebunden* wird. Damit meinen wir folgendes: Wir können in dieser Gleichung für x und y beliebige Werte einsetzen, ohne dass sich an der Gültigkeit der Formel etwas ändert. Setzen wir zum Beispiel für x den Wert 2 ein, so erhalten wir

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

und diese Identität ist ebenfalls gültig. Demgegenüber macht es keinen Sinn, wenn wir für die gebundene Variable t eine Zahl einsetzen würden. Die linke Seite der entstehenden Gleichung wäre einfach undefiniert. Wir können für t höchstens eine andere Variable einsetzen. Ersetzen wir die Variable t beispielsweise durch u , so erhalten wir

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

und das ist dieselbe Aussage wie oben. Das funktioniert allerdings nicht mit jeder Variablen. Setzen wir für t die Variable y ein, so erhalten wir

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

Diese Aussage ist aber falsch! Das Problem liegt darin, dass bei der Ersetzung von t durch y die vorher freie Variable y gebunden wurde.

Ein ähnliches Problem erhalten wir, wenn wir für y beliebige Terme einsetzen. Solange diese Terme die Variable t nicht enthalten, geht alles gut. Setzen wir beispielsweise für y den Term x^2 ein, so erhalten wir

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

und diese Formel ist gültig. Setzen wir allerdings für y den Term t^2 ein, so erhalten wir

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

und diese Formel ist nicht mehr gültig.

In der Prädikatenlogik binden die Quantoren “ \forall ” (*für alle*) und “ \exists ” (*es gibt*) Variablen in ähnlicher Weise, wie der Integral-Operator “ $\int \cdot dt$ ” in der Analysis Variablen bindet. Die oben gemachten Ausführungen zeigen, dass es zwei verschiedene Arten von Variable gibt: *freie Variablen* und *gebundene Variablen*. Um diese Begriffe präzisieren zu können, definieren wir zunächst für einen Σ -Term t die Menge der in t enthaltenen Variablen.

Definition 29 ($\text{Var}(t)$) Ist t ein Σ -Term, mit $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$, so definieren wir die Menge $\text{Var}(t)$ der Variablen, die in t auftreten, durch Induktion nach dem Aufbau des Terms:

1. $\text{Var}(x) := \{x\}$ für alle $x \in \mathcal{V}$,
2. $\text{Var}(f(t_1, \dots, t_n)) := \text{Var}(t_1) \cup \dots \cup \text{Var}(t_n)$. \diamond

Definition 30 (Σ -Formel, \mathbb{F}_Σ , gebundene und freie Variablen, $BV(F)$, $FV(F)$)

Es sei $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$ eine Signatur. Die Menge der Σ -Formeln bezeichnen wir mit \mathbb{F}_Σ . Wir definieren diese Menge induktiv. Gleichzeitig definieren wir für jede Formel $F \in \mathbb{F}_\Sigma$ die Menge $BV(F)$ der in F gebunden auftretenden Variablen und die Menge $FV(F)$ der in F frei auftretenden Variablen.

1. Es gilt $\perp \in \mathbb{F}_\Sigma$ und $\top \in \mathbb{F}_\Sigma$ und wir definieren

$$FV(\perp) := FV(\top) := BV(\perp) := BV(\top) := \{\}.$$
2. Ist $F = p(t_1, \dots, t_n)$ eine atomare Σ -Formel, so gilt $F \in \mathbb{F}_\Sigma$. Weiter definieren wir:

- (a) $FV(p(t_1, \dots, t_n)) := \text{Var}(t_1) \cup \dots \cup \text{Var}(t_n)$.
- (b) $BV(p(t_1, \dots, t_n)) := \{\}$.

3. Ist $F \in \mathbb{F}_\Sigma$, so gilt $\neg F \in \mathbb{F}_\Sigma$. Weiter definieren wir:

- (a) $FV(\neg F) := FV(F)$.
- (b) $BV(\neg F) := BV(F)$.

4. Sind $F, G \in \mathbb{F}_\Sigma$ und gilt außerdem

$$(FV(F) \cup FV(G)) \cap (BV(F) \cup BV(G)) = \{\},$$

so gilt auch

- (a) $(F \wedge G) \in \mathbb{F}_\Sigma$,
- (b) $(F \vee G) \in \mathbb{F}_\Sigma$,
- (c) $(F \rightarrow G) \in \mathbb{F}_\Sigma$,
- (d) $(F \leftrightarrow G) \in \mathbb{F}_\Sigma$.

Weiter definieren wir für alle Junktoren $\odot \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$:

- (a) $FV(F \odot G) := FV(F) \cup FV(G)$.
- (b) $BV(F \odot G) := BV(F) \cup BV(G)$.

5. Sei $x \in \mathcal{V}$ und $F \in \mathbb{F}_\Sigma$ mit $x \notin BV(F)$. Dann gilt:

- (a) $(\forall x: F) \in \mathbb{F}_\Sigma$.
- (b) $(\exists x: F) \in \mathbb{F}_\Sigma$.

Weiter definieren wir

- (a) $FV((\forall x: F)) := FV((\exists x: F)) := FV(F) \setminus \{x\}$.

$$(b) \text{ } BV((\forall x: F)) := BV((\exists x: F)) := BV(F) \cup \{x\}.$$

Ist die Signatur Σ aus dem Zusammenhang klar oder aber unwichtig, so schreiben wir auch \mathbb{F} statt \mathbb{F}_Σ und sprechen dann einfach von Formeln statt von Σ -Formeln. \diamond

Bei der oben gegebenen Definition haben wir darauf geachtet, dass eine Variable nicht gleichzeitig frei und gebunden in einer Formel auftreten kann, denn durch eine leichte Induktion nach dem Aufbau der Formeln lässt sich zeigen, dass für alle $F \in \mathbb{F}_\Sigma$ folgendes gilt:

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

Beispiel: Setzen wir das oben begonnene Beispiel fort, so sehen wir, dass

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

eine Formel aus $\mathbb{F}_{\Sigma_{\text{arith}}}$ ist. Die Menge der gebundenen Variablen ist $\{x\}$, die Menge der freien Variablen ist $\{y\}$. \diamond

Wenn wir Formeln immer in der oben definierten Präfix-Notation anschreiben würden, dann würde die Lesbarkeit unverhältnismäßig leiden. Zur Abkürzung vereinbaren wir, dass in der Prädikatenlogik dieselben Regeln zur Klammer-Ersparnis gelten sollen, die wir schon in der Aussagenlogik verwendet haben. Zusätzlich werden gleiche Quantoren zusammengefasst: Beispielsweise schreiben wir

$$\forall x, y: p(x, y) \quad \text{statt} \quad \forall x: (\forall y: p(x, y)).$$

Darüber hinaus legen wir fest, dass Quantoren stärker binden als die aussagenlogischen Junktoren. Damit können wir

$$\forall x: p(x) \wedge G \quad \text{statt} \quad (\forall x: p(x)) \wedge G$$

schreiben. Außerdem vereinbaren wir, dass wir zweistellige Prädikats- und Funktions-Zeichen auch in Infix-Notation angeben dürfen. Um eine eindeutige Lesbarkeit zu erhalten, müssen wir dann gegebenenfalls Klammern setzen. Wir schreiben beispielsweise

$$\mathbf{n}_1 = \mathbf{n}_2 \quad \text{anstelle von} \quad =(\mathbf{n}_1, \mathbf{n}_2).$$

Die Formel $(\exists x: \leq(+ (y, x), y))$ wird dann lesbarer als

$$\exists x: y + x \leq y$$

geschrieben. Außerdem finden Sie in der Literatur häufig Ausdrücke der Form $\forall x \in M : F$ oder $\exists x \in M : F$. Hierbei handelt es sich um Abkürzungen, die durch

$$(\forall x \in M : F) \stackrel{\text{def}}{\iff} \forall x : (x \in M \rightarrow F), \quad \text{und} \quad (\exists x \in M : F) \stackrel{\text{def}}{\iff} \exists x : (x \in M \wedge F).$$

definiert sind.

5.2 Semantik der Prädikatenlogik

Als nächstes legen wir die Bedeutung der Formeln fest. Dazu definieren wir den Begriff einer Σ -Struktur. Eine solche Struktur legt fest, wie die Funktions- und Prädikats-Zeichen der Signatur Σ zu interpretieren sind.

Definition 31 (Struktur) Es sei eine Signatur

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

gegeben. Eine Σ -Struktur \mathcal{S} ist ein Paar $\langle \mathcal{U}, \mathcal{J} \rangle$, so dass folgendes gilt:

1. \mathcal{U} ist eine nicht-leere Menge. Diese Menge nennen wir auch das **Universum** der Σ -Struktur. Dieses Universum enthält die Werte, die sich später bei der Auswertung der Terme ergeben werden.
2. \mathcal{J} ist die **Interpretation** der Funktions- und Prädikats-Zeichen. Formal definieren wir \mathcal{J} als eine Abbildung mit folgenden Eigenschaften:

(a) Jedem Funktions-Zeichen $f \in \mathcal{F}$ mit $\text{arity}(f) = m$ wird eine m -stellige Funktion

$$f^{\mathcal{J}} : \mathcal{U} \times \cdots \times \mathcal{U} \rightarrow \mathcal{U}$$

zugeordnet, die m -Tupel des Universums \mathcal{U} in das Universum \mathcal{U} abbildet.

(b) Jedem Prädikats-Zeichen $p \in \mathcal{P}$ mit $\text{arity}(p) = n$ wird eine Teilmenge

$$p^{\mathcal{J}} \subseteq \mathcal{U} \times \cdots \times \mathcal{U}$$

zugeordnet. Die Idee ist, dass eine atomare Formel der Form $p(t_1, \dots, t_n)$ genau dann als wahr interpretiert wird, wenn die Interpretation des Tupels $\langle t_1, \dots, t_n \rangle$ in der Menge $p^{\mathcal{J}}$ liegt.

(c) Ist das Zeichen “=” ein Element der Menge der Prädikats-Zeichen \mathcal{P} , so gilt

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

Eine Formel der Art $s = t$ wird als genau dann als wahr interpretiert, wenn die Interpretation des Terms s den selben Wert ergibt wie die Interpretation des Terms t . \diamond

Beispiel: Die Signatur Σ_G der Gruppen-Theorie sei definiert als

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

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

Dann können wir eine Σ_G Struktur $\mathcal{Z} = \langle \{a, b\}, \mathcal{J} \rangle$ definieren, indem wir die Interpretation \mathcal{J} wie folgt festlegen:

1. $1^{\mathcal{J}} := a$,
2. $*^{\mathcal{J}} := \{ \langle \langle a, a \rangle, a \rangle, \langle \langle a, b \rangle, b \rangle, \langle \langle b, a \rangle, b \rangle, \langle \langle b, b \rangle, a \rangle \}$,
3. $=^{\mathcal{J}} := \{ \langle a, a \rangle, \langle b, b \rangle \}$.

Beachten Sie, dass wir bei der Interpretation des Gleichheits-Zeichens keinen Spielraum haben! \diamond

Falls wir Terme auswerten wollen, die Variablen enthalten, so müssen wir für diese Variablen irgendwelche Werte aus dem Universum einsetzen. Welche Werte wir einsetzen, kann durch eine **Variablen-Belegung** festgelegt werden. Diesen Begriff definieren wir nun.

Definition 32 (Variablen-Belegung) Es sei eine Signatur

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

gegeben. Weiter sei $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ eine Σ -Struktur. Dann bezeichnen wir eine Abbildung

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

als eine **S-Variablen-Belegung**.

Ist \mathcal{I} eine \mathcal{S} -Variablen-Belegung, $x \in \mathcal{V}$ und $c \in \mathcal{U}$, so bezeichnet $\mathcal{I}[x/c]$ die Variablen-Belegung, die der Variablen x den Wert c zuordnet und die ansonsten mit \mathcal{I} übereinstimmt:

$$\mathcal{I}[x/c](y) := \begin{cases} c & \text{falls } y = x; \\ \mathcal{I}(y) & \text{sonst.} \end{cases} \quad \diamond$$

Definition 33 (Semantik der Terme) Ist $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ eine Σ -Struktur und \mathcal{I} eine \mathcal{S} -Variablen-Belegung, so definieren wir für jeden Term t den Wert $\mathcal{S}(\mathcal{I}, t)$ durch Induktion über den Aufbau von t :

1. Für Variablen $x \in \mathcal{V}$ definieren wir:

$$\mathcal{S}(\mathcal{I}, x) := \mathcal{I}(x).$$

2. Für Σ -Terme der Form $f(t_1, \dots, t_n)$ definieren wir

$$\mathcal{S}(\mathcal{I}, f(t_1, \dots, t_n)) := f^{\mathcal{J}}(\mathcal{S}(\mathcal{I}, t_1), \dots, \mathcal{S}(\mathcal{I}, t_n)).$$

◇

Beispiel: Mit der oben definierten Σ_G -Struktur \mathcal{Z} definieren wir eine \mathcal{Z} -Variablen-Belegung \mathcal{I} durch

$$\mathcal{I} := \{ \langle x, a \rangle, \langle y, b \rangle, \langle z, a \rangle \},$$

es gilt also

$$\mathcal{I}(x) := a, \quad \mathcal{I}(y) := b, \quad \text{und} \quad \mathcal{I}(z) := a.$$

Dann gilt

$$\mathcal{Z}(\mathcal{I}, x * y) = b.$$

◇

Definition 34 (Semantik der atomaren Σ -Formeln) Ist \mathcal{S} eine Σ -Struktur und \mathcal{I} eine \mathcal{S} -Variablen-Belegung, so definieren wir für jede atomare Σ -Formel $p(t_1, \dots, t_n)$ den Wert $\mathcal{S}(\mathcal{I}, p(t_1, \dots, t_n))$ wie folgt:

$$\mathcal{S}(\mathcal{I}, p(t_1, \dots, t_n)) := \left(\langle \mathcal{S}(\mathcal{I}, t_1), \dots, \mathcal{S}(\mathcal{I}, t_n) \rangle \in p^{\mathcal{J}} \right).$$

◇

Beispiel: In Fortführung des obigen Beispiels gilt:

$$\mathcal{Z}(\mathcal{I}, x * y = y * x) = \text{true}.$$

◇

Um die Semantik beliebiger Σ -Formeln definieren zu können, nehmen wir an, dass wir, genau wie in der Aussagenlogik, die folgenden Funktionen zur Verfügung haben:

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

Die Semantik dieser Funktionen hatten wir durch die Tabelle in Abbildung 4.1 auf Seite 58 gegeben.

Definition 35 (Semantik der Σ -Formeln) Ist \mathcal{S} eine Σ -Struktur und \mathcal{I} eine \mathcal{S} -Variablen-Belegung, so definieren wir für jede Σ -Formel F den Wert $\mathcal{S}(\mathcal{I}, F)$ durch Induktion über den Aufbau von F :

1. $\mathcal{S}(\mathcal{I}, \top) := \text{true}$ und $\mathcal{S}(\mathcal{I}, \perp) := \text{false}$.
2. $\mathcal{S}(\mathcal{I}, \neg F) := \ominus(\mathcal{S}(\mathcal{I}, F))$.
3. $\mathcal{S}(\mathcal{I}, F \wedge G) := \oslash(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G))$.
4. $\mathcal{S}(\mathcal{I}, F \vee G) := \odot(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G))$.
5. $\mathcal{S}(\mathcal{I}, F \rightarrow G) := \oplus(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G))$.
6. $\mathcal{S}(\mathcal{I}, F \leftrightarrow G) := \ominus(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G))$.
7. $\mathcal{S}(\mathcal{I}, \forall x: F) := \begin{cases} \text{true} & \text{falls } \mathcal{S}(\mathcal{I}[x/c], F) = \text{true} \text{ für alle } c \in \mathcal{U} \text{ gilt;} \\ \text{false} & \text{sonst.} \end{cases}$

$$8. \mathcal{S}(\mathcal{I}, \exists x: F) := \begin{cases} \text{true} & \text{falls } \mathcal{S}(\mathcal{I}[x/c], F) = \text{true} \text{ für ein } c \in \mathcal{U} \text{ gilt;} \\ \text{false} & \text{sonst.} \end{cases} \quad \diamond$$

Beispiel: In Fortführung des obigen Beispiels gilt

$$\mathcal{S}(\mathcal{I}, \forall x: 1 * x = x) = \text{true}. \quad \diamond$$

Definition 36 (Allgemeingültig) Ist F eine Σ -Formel, so dass für jede Σ -Struktur \mathcal{S} und für jede \mathcal{S} -Variablen-Belegung \mathcal{I}

$$\mathcal{S}(\mathcal{I}, F) = \text{true}$$

gilt, so bezeichnen wir F als **allgemeingültig**. In diesem Fall schreiben wir

$$\models F. \quad \diamond$$

Ist F eine Formel für die $FV(F) = \{\}$ ist, dann hängt der Wert $\mathcal{S}(\mathcal{I}, F)$ offenbar gar nicht von der Interpretation \mathcal{I} ab. Solche Formeln bezeichnen wir auch als **geschlossene** Formeln. In diesem Fall schreiben wir kürzer $\mathcal{S}(F)$ an Stelle von $\mathcal{S}(\mathcal{I}, F)$. Gilt dann zusätzlich $\mathcal{S}(F) = \text{true}$, so sagen wir auch dass \mathcal{S} ein **Modell** von F ist. Wir schreiben dann

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

Die Definition der Begriffe “**erfüllbar**” und “**äquivalent**” lassen sich nun aus der Aussagenlogik übertragen. Um unnötigen Ballast in den Definitionen zu vermeiden, nehmen wir im Folgenden immer eine feste Signatur Σ als gegeben an. Dadurch können wir in den folgenden Definitionen von Termen, Formeln, Strukturen, etc. sprechen und meinen damit Σ -Terme, Σ -Formeln und Σ -Strukturen.

Definition 37 (äquivalent) Zwei Formeln F und G heißen **äquivalent** g.d.w. gilt

$$\models F \leftrightarrow G. \quad \diamond$$

Alle aussagenlogischen Äquivalenzen sind auch prädikatenlogische Äquivalenzen.

Definition 38 (Erfüllbar) Eine Menge $M \subseteq \mathbb{F}_\Sigma$ ist genau dann **erfüllbar**, wenn es eine Struktur \mathcal{S} und eine Variablen-Belegung \mathcal{I} gibt, so dass

$$\forall m \in M : \mathcal{S}(\mathcal{I}, m) = \text{true}$$

gilt. Andernfalls heißt M **unerfüllbar** oder auch **widersprüchlich**. Wir schreiben dafür auch

$$M \models \perp \quad \diamond$$

Unser Ziel ist es, ein Verfahren anzugeben, mit dem wir in der Lage sind zu überprüfen, ob eine Menge M von Formeln **widersprüchlich** ist, ob also $M \models \perp$ gilt. Es zeigt sich, dass dies im Allgemeinen nicht möglich ist, die Frage, ob $M \models \perp$ gilt, ist unentscheidbar. Ein Beweis dieser Tatsache geht allerdings über den Rahmen dieser Vorlesung hinaus. Dem gegenüber ist es möglich, ähnlich wie in der Aussagenlogik einen **Kalkül** \vdash anzugeben, so dass gilt

$$M \vdash \perp \quad \text{g.d.w.} \quad M \models \perp.$$

Ein solcher Kalkül kann dann zur Implementierung eines **Semi-Entscheidungs-Verfahrens** benutzt werden: Um zu überprüfen, ob $M \models \perp$ gilt, versuchen wir, aus der Menge M die Formel \perp herzuleiten. Falls wir dabei systematisch vorgehen, indem wir alle möglichen Beweise durchprobieren, so werden wir, falls tatsächlich $M \models \perp$ gilt, auch irgendwann einen Beweis finden, der $M \vdash \perp$ zeigt. Wenn allerdings der Fall

$$M \not\models \perp$$

vorliegt, so werden wir dies im allgemeinen nicht feststellen können, denn die Menge aller Beweise ist unendlich groß und wir können nie alle Beweise ausprobieren. Wir können lediglich sicherstellen, dass wir jeden Beweis irgendwann versuchen. Wenn es aber keinen Beweis gibt, so können wir das nie sicher sagen, denn zu jedem festen Zeitpunkt haben wir ja immer nur einen Teil der in Frage kommenden Schlüsse ausprobiert.

Die Situation ist ähnlich der, wie bei der Überprüfung bestimmter zahlentheoretischer Fragen. Wir betrachten dazu ein konkretes Beispiel: Eine Zahl n heißt ist eine **perfekte Zahl**, wenn die Summe aller echten Teiler

von n wieder die Zahl n ergibt. Beispielsweise ist die Zahl 6 perfekt, denn die Menge der echten Teiler von 6 ist $\{1, 2, 3\}$ und es gilt

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

Bisher sind alle bekannten perfekten Zahlen durch 2 teilbar. Die Frage, ob es auch ungerade Zahlen gibt, die perfekt sind, ist ein offenes mathematisches Problem. Um dieses Problem zu lösen könnten wir eine Programm schreiben, dass der Reihe nach für alle ungerade Zahlen überprüft, ob die Zahl perfekt ist. Abbildung 5.1 auf Seite 109 zeigt ein solches Programm. Wenn es eine ungerade perfekte Zahl gibt, dann wird dieses Programm diese Zahl auch irgendwann finden. Wenn es aber keine ungerade perfekte Zahl gibt, dann wird das Programm bis zum St. Nimmerleinstag rechnen und wir werden nie mit Sicherheit wissen, dass es keine ungeraden perfekten Zahlen gibt.

```

1  perfect := procedure(n) {
2      return +/ { x : x in {1 .. n-1} | n % x == 0 } == n;
3  };
4  findPerfect := procedure() {
5      n := 1;
6      while (true) {
7          if (perfect(n)) {
8              if (n % 2 == 0) {
9                  print(n);
10             } else {
11                 print("Heureka: Odd perfect number $n$ found!");
12             }
13         }
14         n := n + 1;
15     }
16 };
17 findPerfect();

```

Figure 5.1: Suche nach einer ungeraden perfekten Zahl.

5.2.1 Implementierung prädikatenlogischer Strukturen in SETLX

Der im letzten Abschnitt präsentierte Begriff einer prädikatenlogischen Struktur erscheint zunächst sehr abstrakt. Wir wollen in diesem Abschnitt zeigen, dass sich dieser Begriff in einfacher Weise in SETLX implementieren lässt. Dadurch gelingt es, diesen Begriff zu veranschaulichen. Als konkretes Beispiel wollen wir Strukturen zu Gruppen-Theorie betrachten. Die Signatur Σ_G der Gruppen-Theorie war im letzten Abschnitt durch die Definition

$$\Sigma_G = \langle \{x, y, z\}, \{1, *\}, \{=\}, \{\langle 1, 0 \rangle, \langle *, 2 \rangle, \langle =, 2 \rangle\} \rangle$$

gegeben worden. Hierbei ist also “1” ein 0-stelliges Funktions-Zeichen, “*” ist eine 2-stellige Funktions-Zeichen und “=” ist ein 2-stelliges Prädikats-Zeichen. Wir hatten bereits eine Struktur \mathcal{S} angegeben, deren Universum aus der Menge $\{a, b\}$ besteht. In SETLX können wir diese Struktur durch den in Abbildung 5.2 gezeigten Code implementieren.

1. Zur Abkürzung haben wir in den Zeile 1 und 2 die Variablen a und b als die Strings “a” und “b” definiert. Dadurch können wir weiter unten die Interpretation des Funktions-Zeichens “*” kürzer angeben.
2. Das in Zeile 3 definierte Universum U besteht aus den beiden Strings “a” und “b”.

```

1  a := "a";
2  b := "b";
3  U := { a, b }; // the universe
4  product := { [[a, a], a], [[a, b], b], [[b, a], b], [[b, b], a] };
5  equals := { [ x, x ] : x in U };
6  J := { [ "E", a ], [ "@@product", product ], [ "@@equals", equals ] };
7  S := [ U, J ];
8  I := { [ "x", a ], [ "y", b ], [ "z", a ] };

```

Figure 5.2: Implementierung einer Struktur zur Gruppen-Theorie

3. In Zeile 4 definieren wir eine Funktion **product** als binäre Relation. Für die so definierte Funktion gilt

$$\begin{aligned} \text{product}(\text{"a"}, \text{"a"}) &= \text{"a"}, & \text{product}(\text{"a"}, \text{"b"}) &= \text{"b"}, \\ \text{product}(\text{"b"}, \text{"a"}) &= \text{"b"}, & \text{product}(\text{"b"}, \text{"b"}) &= \text{"a"}. \end{aligned}$$

Diese Funktion verwenden wir später als die Interpretation $*^{\mathcal{J}}$ des Funktions-Zeichens “*”.

4. In Zeile 5 haben wir die Interpretation $=^{\mathcal{J}}$ des Prädikats-Zeichens “=” als Menge aller Paare der Form $[x, x]$ dargestellt.
5. In Zeile 6 fassen wir die einzelnen Interpretationen zu der Relation J zusammen, so dass für ein Funktions-Zeichen f die Interpretation $f^{\mathcal{J}}$ durch den Wert $J(f)$ gegeben ist.

Da wir später den in SETLX eingebauten Parser verwenden wollen, müssen wir die prädikatenlogischen Formeln als Terme darstellen. Dazu stellen wir den Operator “*” durch den Funktor “@@product” dar und für das Prädikats-Zeichen “=” verwenden wir den Funktor “@@equals”, denn das sind die Funktoren, die von dem in SETLX integrierten Parser intern benutzt werden. Das neutrale Element “1” stellen wir durch den Funktor “@E” dar, so dass später der Ausdruck “1” durch den Term “@E()” repräsentiert wird.

6. Die Interpretation J wird dann in Zeile 7 mit dem Universum U zu der Struktur S zusammengefasst.
7. Schließlich zeigt Zeile 8, dass eine Variablen-Belegung ebenfalls als Relation dargestellt werden kann. Die erste Komponente der Paare, aus denen diese Relation besteht, sind die Variablen. Die zweite Komponente ist dann jeweils ein Wert aus dem Universum.

Als nächstes überlegen wir uns, wie wir prädikatenlogische Formeln in einer solchen Struktur auswerten können. Abbildung 5.3 zeigt die Implementierung der Prozedur $evalFormula(f, S, I)$, der als Argumente eine prädikatenlogische Formel f , eine Struktur S und eine Variablen-Belegung I übergeben werden. Die Formel wird dabei als Term dargestellt.

Die Auswertung einer prädikatenlogischen Formel ist nun analog zu der in Abbildung 4.1 auf Seite 62 gezeigten Auswertung aussagenlogischer Formeln. Neu ist nur die Behandlung der Quantoren. In den Zeilen 11 und 12 behandeln wir die Auswertung allquantifizierter Formeln. Ist f eine Formel der Form $\forall y : h$, so wird die Formel f durch den Term

$$f = @Forall(y, h)$$

dargestellt. Das Muster

$$@Forall(x, g)$$

bindet daher x an die tatsächlich auftretende Variable y und g an die Teilformel h . Die Auswertung von $\forall x: h$ geschieht nach der Formel

$$S(I, \forall x: h) := \begin{cases} \text{true} & \text{falls } S(I[x/c], h) = \text{true} \text{ für alle } c \in \mathcal{U} \text{ gilt;} \\ \text{false} & \text{sonst.} \end{cases}$$

```

1  evalFormula := procedure(f, S, I) {
2      U := S[1];
3      match (f) {
4          case true      : return true;
5          case false     : return false;
6          case !g        : return !evalFormula(g, S, I);
7          case g && h     : return  evalFormula(g, S, I) && evalFormula(h, S, I);
8          case g || h     : return  evalFormula(g, S, I) || evalFormula(h, S, I);
9          case g => h     : return  evalFormula(g, S, I) => evalFormula(h, S, I);
10         case g <=> h    : return  evalFormula(g, S, I) == evalFormula(h, S, I);
11         case @Forall(x, g) :
12             return forall (c in U | evalFormula(g, S, modify(I, x, c)));
13         case @Exists(x, g) :
14             return exists (c in U | evalFormula(g, S, modify(I, x, c)));
15         default : return evalAtomic(f, S, I); // atomic formula
16     }
17 };

```

Figure 5.3: Auswertung prädikatenlogischer Formeln

Um die Auswertung implementieren zu können, verwenden wir eine Prozedur *modify()*, welche die Variablen-Belegung *I* an der Stelle *x* zu *c* abändert, es gilt also

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

Die Implementierung dieser Prozedur ist in Abbildung 5.4 auf Seite 111 gezeigt. Bei der Auswertung eines All-Quantors können wir ausnutzen, dass die Sprache SETLX die beiden Quantoren “ \forall ” und “ \exists ” durch die Operatoren **forall** bzw. **exists** unterstützt. Wir können also direkt testen, ob die Formel für alle möglichen Werte *c*, die wir für die Variable *x* einsetzen können, richtig ist. Die Auswertung eines Existenz-Quantors ist analog zur Auswertung eines All-Quantors.

Bei der Implementierung der in Zeile 31 gezeigten Prozedur *modify(I, x, c)*, die als Ergebnis die Variablen-Belegung $\mathcal{I}[x/c]$ berechnet, nutzen wir aus, dass wir bei einer Funktion, die als binäre Relation gespeichert ist, den Wert, der in dieser Relation für ein Argument *x* eingetragen ist, durch eine Zuweisung der Form $\mathcal{I}(x) := c$ abändern können.

```

1  modify := procedure(I, v, c) {
2      x := args(v)[1];
3      I[x] := c;
4      return I;
5  };

```

Figure 5.4: Die Implementierung der Funktion *modify*.

Abbildung 5.5 auf Seite 112 zeigt die Auswertung atomarer Formeln. Um eine atomare Formel der Form

$$a = p(t_1, \dots, t_n)$$

auszuwerten, verschaffen wir uns in Zeile 4 zunächst die dem Prädikats-Zeichen *p* in der Struktur *S* zugeordnete Menge *pJ*. Anschließend werten wir die Argumente t_1, \dots, t_n aus. Dabei erhalten wir die Liste $[t_1, \dots, t_n]$ der Argumente des Prädikats-Zeichens *p* durch Anwendung der Funktion **args** in Zeile 5. Die Auswertung des Terms t_i in der Struktur *S* unter der Variablen-Belegung *I* hatten wir im letzten Abschnitt mit

$$\mathcal{S}(\mathcal{I}, t_i)$$

bezeichnet. Wir erhalten daher in Zeile 6 als Auswertung der Liste $[t_1, \dots, t_n]$ die Liste

$$[\mathcal{S}(\mathcal{I}, t_1), \dots, \mathcal{S}(\mathcal{I}, t_n)].$$

Dies ist eine Liste von Werten aus dem Universum \mathcal{U} der Struktur \mathcal{S} . Wir prüfen nun in Zeile 7, ob diese Liste tatsächlich ein Element der Menge pJ ist.

```

1  evalAtomic := procedure(a, S, I) {
2      J := S[2];
3      p := fct(a); // predicate symbol
4      pJ := J[p];
5      argList := args(a);
6      argsVal := evalTermList(argList, S, I);
7      return argsVal in pJ;
8  };

```

Figure 5.5: Auswertung von atomaren Formeln.

Die in Abbildung 5.6 auf Seite 113 gezeigte Prozedur `evalTerm()` arbeitet wie folgt: Das erste Argument t der Prozedur `evalTerm(t, S, I)` ist der auszuwertende Term. Das zweite Argument \mathcal{S} ist eine prädikatenlogische Struktur und das dritte Argument \mathcal{I} ist eine Variablen-Belegung.

1. Falls t eine Variable ist, so geben wir in Zeile 4 einfach den Wert zurück, der in der Variablen-Belegung \mathcal{I} für diese Variable eingetragen ist. Die Variablen-Belegung \mathcal{I} wird dabei durch eine zweistellige Relation dargestellt, die wir als Funktion benutzen.

Variablen werden von dem in SETLX vorgegebenen Parser als Terme der Form

`@@@variable(v)`

dargestellt. Hierbei ist v dann ein String, der als der Name der Variablen interpretiert wird. Konkret liefert beispielsweise der Aufruf

`parse("x")`

als Ergebnis den Term

`@@@variable("x").`

2. Falls der auszuwertende Term t die Form

$$t = f(t_1, \dots, t_n)$$

hat, werden in Zeile 10 zunächst rekursiv die Subterme t_1, \dots, t_n ausgewertet. Anschließend wird die Interpretation `fJ` des Funktions-Zeichens f herangezogen, um die Funktion f für die gegebenen Argumente auszuwerten, wobei in Zeile 12 der Fall betrachtet wird, dass tatsächlich Argumente vorhanden sind, während in Zeile 14 der Fall behandelt wird, dass es sich bei dem Funktions-Zeichen f um eine Konstante handelt, deren Wert dann unmittelbar durch die Interpretation `fJ` gegeben ist.

Die Implementierung der Prozedur `evalTermList()` wendet die Funktion `evalTerm()` auf alle Terme der gegebenen Liste an.

Wir zeigen nun, wie sich die in Abbildung 5.3 gezeigte Funktion `evalFormula(f, S, I)` benutzen lässt um zu überprüfen, ob die in Abbildung 5.2 gezeigte Struktur die Axiome einer **kommutativen Gruppe** *Gruppen-Theorie* erfüllt. Diese Axiome sind wie folgt:

1. Die Konstante 1 ist das rechts-neutrale Element der Multiplikation:

$$\forall x: x * 1 = x.$$

```

1  evalTerm := procedure(t, S, I) {
2      if (fct(t) == "@@variable") {
3          varName := args(t)[1];
4          return I[varName];
5      }
6      J      := S[2];
7      f      := fct(t); // function symbol
8      fJ     := J[f];
9      argList := args(t);
10     argsVal := evalTermList(argList, S, I);
11     if (#argsVal > 0) {
12         return fJ[argsVal];
13     } else {
14         return fJ;    // t is a constant
15     }
16 };
17 evalTermList := procedure(tl, S, I) {
18     return [ evalTerm(t, S, I) : t in tl ];
19 };

```

Figure 5.6: Auswertung von Termen

2. Für jedes Element x gibt es ein rechts-inverses Element y , dass mit dem Element x multipliziert die 1 ergibt:

$$\forall x: \exists y: x * y = 1.$$

3. Es gilt das Assoziativ-Gesetz:

$$\forall x: \forall y: \forall z: (x * y) * z = x * (y * z).$$

4. Es gilt das Kommutativ-Gesetz:

$$\forall x: \forall y: x * y = y * x.$$

Diese Axiome sind in den Zeilen 1 bis 3 der Abbildung 5.7 wiedergegeben, wobei wir die “1” durch das Funktions-Zeichen “E” dargestellt haben. Die Schleife in den Zeilen 7 bis 9 überprüft schließlich, ob die Formeln in der oben definierten Struktur erfüllt sind.

```

1  g1 := parse("@Forall(x, x * @E() == x)");
2  g2 := parse("@Forall(x, @Exists(y, x * y == @E()))");
3  g3 := parse("@Forall(x, @Forall(y, @Forall(z, (x * y) * z == x * (y * z) )))");
4  g4 := parse("@Forall(x, @Forall(y, x * y == y * x))");
5  GT := { g1, g2, g3, g4 };

```

Figure 5.7: Axiome der Gruppen-Theorie

Bemerkung: Das oben vorgestellte Programm finden sie auf GitHub unter der Adresse:

<https://github.com/karlstroetmann/Logik/blob/master/SetlX/fol-evaluate.stlx>

Mit diesem Programm können wir überprüfen, ob eine prädikatenlogische Formel in einer vorgegebenen endlichen Struktur erfüllt ist. Wir können damit allerdings nicht überprüfen, ob eine Formel allgemeingültig ist, denn einerseits können wir das Programm nicht anwenden, wenn die Strukturen ein unendliches Universum haben,

andererseits ist selbst die Zahl der verschiedenen endlichen Stukturen, die wir ausprobieren müssten, unendlich groß. \diamond

Aufgabe 2:

1. Zeigen Sie, dass die Formel

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

nicht allgemeingültig ist, indem Sie in SETLX eine geeignete prädikatenlogische Struktur \mathcal{S} implementieren, in der diese Formel falsch ist.

2. Entwickeln Sie ein SETLX-Programm, dass die obige Formel in allen Strukturen ausprobiert, in denen das Universum aus einer vorgegebenen Zahl n verschiedener Elemente besteht und testen Sie Ihr Programm für $n = 2$.
3. Überlegen Sie, wie viele verschiedene Strukturen mit n Elementen es für die obige Formel gibt.
4. Geben Sie eine erfüllbare prädikatenlogische Formel F an, die in einer prädikatenlogischen Struktur $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ immer falsch ist, wenn das Universum \mathcal{U} endlich ist. \diamond

5.3 Normalformen für prädikatenlogische Formeln

Im nächsten Abschnitt gehen wir daran, den früher erwähnten Kalkül \vdash für die Prädikatenlogik zu definieren. Es zeigt sich, dass die Arbeit wesentlich einfacher wird, wenn wir uns auf bestimmte Formeln, sogenannte *Klauseln*, beschränken. Wir zeigen daher zunächst in diesem Abschnitt, dass jede Formel-Menge M so in eine Menge von Klauseln K transformiert werden kann, dass M genau dann erfüllbar ist, wenn K erfüllbar ist. Daher ist die Beschränkung auf Klauseln keine echte Einschränkung. Zunächst geben wir einige Äquivalenzen an, mit deren Hilfe Quantoren manipuliert werden können.

Satz 39 *Es gelten die folgenden Äquivalenzen:*

1. $\models \neg(\forall x: f) \leftrightarrow (\exists x: \neg f)$
2. $\models \neg(\exists x: f) \leftrightarrow (\forall x: \neg f)$
3. $\models (\forall x: f) \wedge (\forall x: g) \leftrightarrow (\forall x: f \wedge g)$
4. $\models (\exists x: f) \vee (\exists x: g) \leftrightarrow (\exists x: f \vee g)$
5. $\models (\forall x: \forall y: f) \leftrightarrow (\forall y: \forall x: f)$
6. $\models (\exists x: \exists y: f) \leftrightarrow (\exists y: \exists x: f)$
7. Falls x eine Variable ist, für die $x \notin FV(f)$ ist, so haben wir

$$\models (\forall x: f) \leftrightarrow f \quad \text{und} \quad \models (\exists x: f) \leftrightarrow f.$$

8. Falls x eine Variable ist, für die $x \notin FV(g) \cup BV(g)$ gilt, so haben wir die folgenden Äquivalenzen:

- (a) $\models (\forall x: f) \vee g \leftrightarrow \forall x: (f \vee g)$
- (b) $\models g \vee (\forall x: f) \leftrightarrow \forall x: (g \vee f)$
- (c) $\models (\exists x: f) \wedge g \leftrightarrow \exists x: (f \wedge g)$
- (d) $\models g \wedge (\exists x: f) \leftrightarrow \exists x: (g \wedge f)$

Um die Äquivalenzen der letzten Gruppe anwenden zu können, ist es notwendig, gebundene Variablen umzubenennen. Ist f eine prädikatenlogische Formel und sind x und y zwei Variablen, so bezeichnet $f[x/y]$ die Formel, die aus f dadurch entsteht, dass jedes Auftreten der Variablen x in f durch y ersetzt wird. Beispielsweise gilt

$$(\forall u : \exists v : p(u, v))[u/z] = \forall z : \exists v : p(z, v)$$

Damit können wir eine letzte Äquivalenz angeben: Ist f eine prädikatenlogische Formel, ist $x \in BV(F)$ und ist y eine Variable, die in f nicht auftritt, so gilt

$$\models f \leftrightarrow f[x/y].$$

Mit Hilfe der oben stehenden Äquivalenzen können wir eine Formel so umformen, dass die Quantoren nur noch außen stehen. Eine solche Formel ist dann in *pränexer Normalform*. Wir führen das Verfahren an einem Beispiel vor: Wir zeigen, dass die Formel

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

allgemeingültig ist:

$$\begin{aligned} & (\forall x: p(x)) \rightarrow (\exists x: p(x)) \\ \Leftrightarrow & \neg(\forall x: p(x)) \vee (\exists x: p(x)) \\ \Leftrightarrow & (\exists x: \neg p(x)) \vee (\exists x: p(x)) \\ \Leftrightarrow & \exists x: (\neg p(x) \vee p(x)) \\ \Leftrightarrow & \exists x: \top \\ \Leftrightarrow & \top \end{aligned}$$

In diesem Fall haben wir Glück gehabt, dass es uns gelungen ist, die Formel als Tautologie zu erkennen. Im Allgemeinen reichen die obigen Umformungen aber nicht aus, um prädikatenlogische Tautologien erkennen zu können. Um Formeln noch stärker normalisieren zu können, führen wir einen weiteren Äquivalenz-Begriff ein. Diesen Begriff wollen wir vorher durch ein Beispiel motivieren. Wir betrachten die beiden Formeln

$$f_1 = \forall x: \exists y: p(x, y) \quad \text{und} \quad f_2 = \forall x: p(x, s(x)).$$

Die beiden Formeln f_1 und f_2 sind nicht äquivalent, denn sie entstammen noch nicht einmal der gleichen Signatur: In der Formel f_2 wird das Funktions-Zeichen s verwendet, das in der Formel f_1 überhaupt nicht auftritt. Auch wenn die beiden Formeln f_1 und f_2 nicht äquivalent sind, so besteht zwischen ihnen doch die folgende Beziehung: Ist S_1 eine prädikatenlogische Struktur, in der die Formel f_1 gilt:

$$S_1 \models f_1,$$

dann können wir diese Struktur zu einer Struktur S_2 erweitern, in der die Formel f_2 gilt:

$$S_2 \models f_2.$$

Dazu muss lediglich die Interpretation des Funktions-Zeichens s so gewählt werden, dass für jedes x tatsächlich $p(x, s(x))$ gilt. Dies ist möglich, denn die Formel f_1 sagt ja aus, dass wir tatsächlich zu jedem x einen Wert y finden, für den $p(x, y)$ gilt. Die Funktion s muss also lediglich zu jedem x dieses y zurück geben.

Definition 40 (Skolemisierung) Es sei $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$ eine Signatur. Ferner sei f eine geschlossene Σ -Formel der Form

$$f = \forall x_1, \dots, x_n: \exists y: g(x_1, \dots, x_n, y).$$

Dann wählen wir ein neues n -stelliges Funktions-Zeichen s , d.h. wir nehmen ein Zeichen s , dass in der Signatur Σ nicht auftritt und erweitern die Signatur Σ zu der Signatur

$$\Sigma' := \langle \mathcal{V}, \mathcal{F} \cup \{s\}, \mathcal{P}, \text{arity} \cup \{\langle s, n \rangle\} \rangle,$$

in der wir s als neues n -stelliges Funktions-Zeichen deklarieren. Anschließend definieren wir die Σ' -Formel f' wie folgt:

$$f' := \text{Skolem}(f) := \forall x_1: \dots \forall x_n: g(x_1, \dots, x_n, s(x_1, \dots, x_n))$$

Wir lassen also den Existenz-Quantor $\exists y$ weg und ersetzen jedes Auftreten der Variable y durch den Term $s(x_1, \dots, x_n)$. Wir sagen, dass die Formel f' aus der Formel f durch einen **Skolemisierungsschritt** hervorgegangen ist. \diamond

Beispiel: Es f die folgende Formel aus der Gruppen-Theorie:

$$f := \forall x : \exists y : y * x = 1.$$

Dann gilt

$$\text{Skolem}(f) = \forall x : s(x) * x = 1. \quad \diamond$$

In welchem Sinne sind eine Formel f und eine Formel f' , die aus f durch einen Skolemisierungsschritt hervorgegangen sind, äquivalent? Zur Beantwortung dieser Frage dient die folgende Definition.

Definition 41 (Erfüllbarkeits-Äquivalenz)

Zwei geschlossene Formeln f und g heißen **erfüllbarkeits-äquivalent** falls f und g entweder beide erfüllbar oder beide unerfüllbar sind. Wenn f und g erfüllbarkeits-äquivalent sind, so schreiben wir

$$f \approx_e g. \quad \diamond$$

Satz 42 Falls die Formel f' aus der Formel f durch einen Skolemisierungsschritt hervorgegangen ist, so sind f und f' erfüllbarkeits-äquivalent.

Wir können nun ein einfaches Verfahren angeben, um Existenz-Quantoren aus einer Formel zu eliminieren. Dieses Verfahren besteht aus zwei Schritten: Zunächst bringen wir die Formel in pränex Normalform. Anschließend können wir die Existenz-Quantoren der Reihe nach durch Skolemisierungsschritte eliminieren. Nach dem eben gezeigten Satz ist die resultierende Formel zu der ursprünglichen Formel erfüllbarkeits-äquivalent. Dieses Verfahren der Eliminierung von Existenz-Quantoren durch die Einführung neuer Funktions-Zeichen wird als **Skolemisierung** bezeichnet. Haben wir eine Formel F in pränex Normalform gebracht und anschließend skolemisiert, so hat das Ergebnis die Gestalt

$$\forall x_1, \dots, x_n : g$$

und in der Formel g treten keine Quantoren mehr auf. Die Formel g wird auch als die **Matrix** der obigen Formel bezeichnet. Wir können nun g mit Hilfe der uns aus dem letzten Kapitel bekannten aussagenlogischen Äquivalenzen in konjunktive Normalform bringen. Wir haben dann eine Formel der Gestalt

$$\forall x_1, \dots, x_n : (k_1 \wedge \dots \wedge k_m).$$

Dabei sind die k_i Disjunktionen von **Literalen**. In der Prädikatenlogik ist ein **Literal** entweder eine atomare Formel oder die Negation einer atomaren Formel. Wenden wir hier die Äquivalenz

$$(\forall x : (f_1 \wedge f_2)) \leftrightarrow (\forall x : f_1) \wedge (\forall x : f_2)$$

an, so können wir die All-Quantoren auf die einzelnen k_i verteilen und die resultierende Formel hat die Gestalt

$$(\forall x_1, \dots, x_n : k_1) \wedge \dots \wedge (\forall x_1, \dots, x_n : k_m).$$

Ist eine Formel F in der obigen Gestalt, so sagen wir, dass F in **prädikatenlogischer Klausel-Normalform** ist und eine Formel der Gestalt

$$\forall x_1, \dots, x_n : k,$$

bei der k eine Disjunktion prädikatenlogischer Literale ist, bezeichnen wir als **prädikatenlogische Klausel**. Ist M eine Menge von Formeln deren Erfüllbarkeit wir untersuchen wollen, so können wir nach dem bisher gezeigten M immer in eine Menge prädikatenlogischer Klauseln umformen. Da dann nur noch All-Quantoren vorkommen, können wir hier die Notation noch vereinfachen indem wir vereinbaren, dass alle Formeln implizit allquantifiziert sind, wir lassen also die All-Quantoren weg.

Wozu sind nun die Umformungen in Skolem-Normalform gut? Es geht darum, dass wir ein Verfahren entwickeln wollen, mit dem es möglich ist für eine prädikatenlogische Formel f zu zeigen, dass f allgemeingültig ist, dass also

$$\models f$$

gilt. Wir wissen, dass

$$\models f \quad \text{g.d.w.} \quad \{\neg f\} \models \perp$$

gilt, denn die Formel f ist genau dann allgemeingültig, wenn es keine Struktur gibt, in der die Formel $\neg f$ erfüllbar ist. Wir bilden daher zunächst $\neg f$ und formen $\neg f$ in prädikatenlogische Klausel-Normalform um. Wir erhalten Klauseln k_1, \dots, k_n , so dass

$$\neg f \approx_e k_1 \wedge \dots \wedge k_n$$

gilt. Anschließend versuchen wir, aus den Klauseln k_1, \dots, k_n einen Widerspruch herzuleiten:

$$\{k_1, \dots, k_n\} \vdash \perp$$

Wenn dies gelingt, dann wissen wir, dass die Menge $\{k_1, \dots, k_n\}$ unerfüllbar ist. Damit ist auch $\neg f$ unerfüllbar und also ist f allgemeingültig. Damit wir aus den Klauseln k_1, \dots, k_n einen Widerspruch herleiten können, brauchen wir natürlich noch einen Kalkül, der mit prädikatenlogischen Klauseln arbeitet. Einen solchen Kalkül werden wir am Ende dieses Kapitels vorstellen.

Um das Verfahren näher zu erläutern demonstrieren wir es an einem Beispiel. Wir wollen untersuchen, ob

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

gilt. Wir wissen, dass dies äquivalent dazu ist, dass

$$\left\{ \neg \left((\exists x: \forall y: p(x, y)) \rightarrow (\forall y: \exists x: p(x, y)) \right) \right\} \models \perp$$

gilt. Wir bringen zunächst die negierte Formel in pränex Normalform.

$$\begin{aligned} & \neg \left((\exists x: \forall y: p(x, y)) \rightarrow (\forall y: \exists x: p(x, y)) \right) \\ \leftrightarrow & \neg \left(\neg (\exists x: \forall y: p(x, y)) \vee (\forall y: \exists x: p(x, y)) \right) \\ \leftrightarrow & (\exists x: \forall y: p(x, y)) \wedge \neg (\forall y: \exists x: p(x, y)) \\ \leftrightarrow & (\exists x: \forall y: p(x, y)) \wedge (\exists y: \neg \exists x: p(x, y)) \\ \leftrightarrow & (\exists x: \forall y: p(x, y)) \wedge (\exists y: \forall x: \neg p(x, y)) \end{aligned}$$

Um an dieser Stelle weitermachen zu können, ist es nötig, die Variablen in dem zweiten Glied der Konjunktion umzubenennen. Wir ersetzen x durch u und y durch v und erhalten

$$\begin{aligned} & (\exists x: \forall y: p(x, y)) \wedge (\exists y: \forall x: \neg p(x, y)) \\ \leftrightarrow & (\exists x: \forall y: p(x, y)) \wedge (\exists v: \forall u: \neg p(u, v)) \\ \leftrightarrow & \exists v: \left((\exists x: \forall y: p(x, y)) \wedge (\forall u: \neg p(u, v)) \right) \\ \leftrightarrow & \exists v: \exists x: \left((\forall y: p(x, y)) \wedge (\forall u: \neg p(u, v)) \right) \\ \leftrightarrow & \exists v: \exists x: \forall y: \left(p(x, y) \wedge (\forall u: \neg p(u, v)) \right) \\ \leftrightarrow & \exists v: \exists x: \forall y: \forall u: \left(p(x, y) \wedge \neg p(u, v) \right) \end{aligned}$$

An dieser Stelle müssen wir skolemisieren um die Existenz-Quantoren los zu werden. Wir führen dazu zwei neue Funktions-Zeichen s_1 und s_2 ein. Dabei gilt $\text{arity}(s_1) = 0$ und $\text{arity}(s_2) = 0$, denn vor den Existenz-Quantoren stehen keine All-Quantoren.

$$\begin{aligned} & \exists v: \exists x: \forall y: \forall u: \left(p(x, y) \wedge \neg p(u, v) \right) \\ \approx_e & \exists x: \forall y: \forall u: \left(p(x, y) \wedge \neg p(u, s_1) \right) \\ \approx_e & \forall y: \forall u: \left(p(s_2, y) \wedge \neg p(u, s_1) \right) \end{aligned}$$

Da jetzt nur noch All-Quantoren auftreten, können wir diese auch noch weglassen, da wir ja vereinbart haben, dass alle freien Variablen implizit allquantifiziert sind. Damit können wir nun die prädikatenlogische Klausel-Normalform angeben, diese ist

$$M := \left\{ \{p(s_2, y)\}, \{\neg p(u, s_1)\} \right\}.$$

Wir zeigen, dass die Menge M widersprüchlich ist. Dazu betrachten wir zunächst die Klausel $\{p(s_2, y)\}$ und setzen in dieser Klausel für y die Konstante s_1 ein. Damit erhalten wir die Klausel

$$\{p(s_2, s_1)\}. \quad (1)$$

Das Ersetzen von y durch s_1 begründen wir damit, dass die obige Klausel ja implizit allquantifiziert ist und wenn etwas für alle y gilt, dann sicher auch für $y = s_1$.

Als nächstes betrachten wir die Klausel $\{\neg p(u, s_1)\}$. Hier setzen wir für die Variablen u die Konstante s_2 ein und erhalten dann die Klausel

$$\{\neg p(s_2, s_1)\} \quad (2)$$

Nun wenden wir auf die Klauseln (1) und (2) die Schnitt-Regel an und finden

$$\{p(s_2, s_1)\}, \quad \{\neg p(s_2, s_1)\} \quad \vdash \quad \{\}.$$

Damit haben wir einen Widerspruch hergeleitet und gezeigt, dass die Menge M unerfüllbar ist. Damit ist dann auch

$$\left\{ \neg \left((\exists x: \forall y: p(x, y)) \rightarrow (\forall y: \exists x: p(x, y)) \right) \right\}$$

unerfüllbar und folglich gilt

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

5.4 Unifikation

In dem Beispiel im letzten Abschnitt haben wir die Terme s_1 und s_2 geraten, die wir für die Variablen y und u in den Klauseln $\{p(s_2, y)\}$ und $\{\neg p(u, s_1)\}$ eingesetzt haben. Wir haben diese Terme mit dem Ziel gewählt, später die Schnitt-Regel anwenden zu können. In diesem Abschnitt zeigen wir nun ein Verfahren, mit dessen Hilfe wir die benötigten Terme ausrechnen können. Dazu benötigen wir zunächst den Begriff einer *Substitution*.

Definition 43 (Substitution) *Es sei eine Signatur*

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

*gegeben. Eine **Σ-Substitution** ist eine endliche Menge von Paaren der Form*

$$\sigma = \{ \langle x_1, t_1 \rangle, \dots, \langle x_n, t_n \rangle \}.$$

Dabei gilt:

1. $x_i \in \mathcal{V}$, die x_i sind also Variablen.
2. $t_i \in \mathcal{T}_\Sigma$, die t_i sind also Terme.
3. Für $i \neq j$ ist $x_i \neq x_j$, die Variablen sind also paarweise verschieden.

Ist $\sigma = \{ \langle x_1, t_1 \rangle, \dots, \langle x_n, t_n \rangle \}$ eine Σ -Substitution, so schreiben wir

$$\sigma = [x_1 \mapsto t_1, \dots, x_n \mapsto t_n].$$

Außerdem definieren wir den Domain einer Substitution als

$$\text{dom}(\sigma) = \{x_1, \dots, x_n\}.$$

Die Menge aller Substitutionen bezeichnen wir mit Subst. ◇

Substitutionen werden für uns dadurch interessant, dass wir sie auf Terme *anwenden* können. Ist t ein Term und σ eine Substitution, so ist $t\sigma$ der Term, der aus t dadurch entsteht, dass jedes Vorkommen einer Variablen x_i durch den zugehörigen Term t_i ersetzt wird. Die formale Definition folgt.

Definition 44 (Anwendung einer Substitution)

Es sei t ein Term und es sei $\sigma = [x_1 \mapsto t_1, \dots, x_n \mapsto t_n]$ eine Substitution. Wir definieren die Anwendung von σ auf t (Schreibweise $t\sigma$) durch Induktion über den Aufbau von t :

1. Falls t eine Variable ist, gibt es zwei Fälle:

- (a) $t = x_i$ für ein $i \in \{1, \dots, n\}$. Dann definieren wir $x_i\sigma := t_i$.
- (b) $t = y$ mit $y \in \mathcal{V}$, aber $y \notin \{x_1, \dots, x_n\}$. Dann definieren wir $y\sigma := y$.

2. Andernfalls muss t die Form $t = f(s_1, \dots, s_m)$ haben. Dann können wir $t\sigma$ durch

$$f(s_1, \dots, s_m)\sigma := f(s_1\sigma, \dots, s_m\sigma).$$

definieren, denn nach Induktions-Voraussetzung sind die Ausdrücke $s_i\sigma$ bereits definiert. \diamond

Genau wie wir Substitutionen auf Terme anwenden können, können wir eine Substitution auch auf prädikatenlogische Klauseln anwenden. Dabei werden Prädikats-Zeichen und Junktoren wie Funktions-Zeichen behandelt. Wir ersparen uns eine formale Definition und geben stattdessen zunächst einige Beispiele. Wir definieren eine Substitution σ durch

$$\sigma := [x_1 \mapsto c, x_2 \mapsto f(d)].$$

In den folgenden drei Beispielen demonstrieren wir zunächst, wie eine Substitution auf einen Term angewendet werden kann. Im vierten Beispiel wenden wir die Substitution dann auf eine Formel an:

- 1. $x_3\sigma = x_3$,
- 2. $f(x_2)\sigma = f(f(d))$,
- 3. $h(x_1, g(x_2))\sigma = h(c, g(f(d)))$.
- 4. $\{p(x_2), q(d, h(x_3, x_1))\}\sigma = \{p(f(d)), q(d, h(x_3, c))\}$.

Als nächstes zeigen wir, wie Substitutionen miteinander verknüpft werden können.

Definition 45 (Komposition von Substitutionen) Es seien

$$\sigma = [x_1 \mapsto s_1, \dots, x_m \mapsto s_m] \quad \text{und} \quad \tau = [y_1 \mapsto t_1, \dots, y_n \mapsto t_n]$$

zwei Substitutionen mit $\text{dom}(\sigma) \cap \text{dom}(\tau) = \{\}$. Dann definieren wir die **Komposition** $\sigma\tau$ von σ und τ als

$$\sigma\tau := [x_1 \mapsto s_1\tau, \dots, x_m \mapsto s_m\tau, y_1 \mapsto t_1, \dots, y_n \mapsto t_n] \quad \diamond$$

Beispiel: Wir führen das obige Beispiel fort und setzen

$$\sigma := [x_1 \mapsto c, x_2 \mapsto f(x_3)] \quad \text{und} \quad \tau := [x_3 \mapsto h(c, c), x_4 \mapsto d].$$

Dann gilt:

$$\sigma\tau = [x_1 \mapsto c, x_2 \mapsto f(h(c, c)), x_3 \mapsto h(c, c), x_4 \mapsto d]. \quad \square$$

Die Definition der Komposition von Substitutionen ist mit dem Ziel gewählt worden, dass der folgende Satz gilt.

Satz 46 Ist t ein Term und sind σ und τ Substitutionen mit $\text{dom}(\sigma) \cap \text{dom}(\tau) = \{\}$, so gilt

$$(t\sigma)\tau = t(\sigma\tau). \quad \square$$

Der Satz kann durch Induktion über den Aufbau des Termes t bewiesen werden.

Definition 47 (Syntaktische Gleichung) Unter einer **syntaktischen Gleichung** verstehen wir in diesem Abschnitt ein Konstrukt der Form $s \doteq t$, wobei einer der beiden folgenden Fälle vorliegen muss:

- 1. s und t sind Terme oder

2. s und t sind atomare Formeln.

Weiter definieren wir ein **syntaktisches Gleichungs-System** als eine Menge von syntaktischen Gleichungen. \diamond

Was syntaktische Gleichungen angeht, so machen wir keinen Unterschied zwischen Funktions-Zeichen und Prädikats-Zeichen. Dieser Ansatz ist deswegen berechtigt, weil wir Prädikats-Zeichen ja auch als spezielle Funktions-Zeichen auffassen können, nämlich als Funktions-Zeichen, die einen Wahrheitswert aus der Menge \mathbb{B} berechnen.

Definition 48 (Unifikator) Eine Substitution σ **löst** eine syntaktische Gleichung $s \doteq t$ genau dann, wenn $s\sigma = t\sigma$ ist, wenn also durch die Anwendung von σ auf s und t tatsächlich identische Objekte entstehen. Ist E ein syntaktisches Gleichungs-System, so sagen wir, dass σ ein **Unifikator** von E ist wenn σ jede syntaktische Gleichung in E löst. \diamond

Ist $E = \{s_1 \doteq t_1, \dots, s_n \doteq t_n\}$ eine syntaktisches Gleichungs-System und ist σ eine Substitution, so definieren wir

$$E\sigma := \{s_1\sigma \doteq t_1\sigma, \dots, s_n\sigma \doteq t_n\sigma\}.$$

Beispiel: Wir verdeutlichen die bisher eingeführten Begriffe anhand eines Beispiels. Wir betrachten die Gleichung

$$p(x_1, f(x_4)) \doteq p(x_2, x_3)$$

und definieren die Substitution

$$\sigma := [x_1 \mapsto x_2, x_3 \mapsto f(x_4)].$$

Die Substitution σ löst die obige syntaktische Gleichung, denn es gilt

$$\begin{aligned} p(x_1, f(x_4))\sigma &= p(x_2, f(x_4)) \quad \text{und} \\ p(x_2, x_3)\sigma &= p(x_2, f(x_4)). \end{aligned}$$

\diamond

Als nächstes entwickeln wir ein Verfahren, mit dessen Hilfe wir von einer vorgegebenen Menge E von syntaktischen Gleichungen entscheiden können, ob es einen Unifikator σ für E gibt. Wir überlegen uns zunächst, in welchen Fällen wir eine syntaktischen Gleichung $s \doteq t$ garantiert nicht lösen können. Da gibt es zwei Möglichkeiten: Eine syntaktische Gleichung

$$f(s_1, \dots, s_m) \doteq g(t_1, \dots, t_n)$$

ist sicher dann nicht durch eine Substitution lösbar, wenn f und g verschiedene Funktions-Zeichen sind, denn für jede Substitution σ gilt ja

$$f(s_1, \dots, s_m)\sigma = f(s_1\sigma, \dots, s_m\sigma) \quad \text{und} \quad g(t_1, \dots, t_n)\sigma = g(t_1\sigma, \dots, t_n\sigma).$$

Falls $f \neq g$ ist, haben die Terme $f(s_1, \dots, s_m)\sigma$ und $g(t_1, \dots, t_n)\sigma$ verschieden Funktions-Zeichen und können daher syntaktisch nicht identisch werden.

Die andere Form einer syntaktischen Gleichung, die garantiert unlösbar ist, ist

$$x \doteq f(t_1, \dots, t_n) \quad \text{falls } x \in \text{Var}(f(t_1, \dots, t_n)).$$

Das diese syntaktische Gleichung unlösbar ist liegt daran, dass die rechte Seite immer mindestens ein Funktions-Zeichen mehr enthält als die linke.

Mit diesen Vorbemerkungen können wir nun ein Verfahren angeben, mit dessen Hilfe es möglich ist, Mengen von syntaktischen Gleichungen zu lösen, oder festzustellen, dass es keine Lösung gibt. Das Verfahren operiert auf Paaren der Form $\langle F, \tau \rangle$. Dabei ist F ein syntaktisches Gleichungs-System und τ ist eine Substitution. Wir starten das Verfahren mit dem Paar $\langle E, [] \rangle$. Hierbei ist E das zu lösende Gleichungs-System und $[]$ ist die leere Substitution. Das Verfahren arbeitet, indem die im Folgenden dargestellten Reduktions-Regeln solange angewendet werden, bis entweder feststeht, dass die Menge der Gleichungen keine Lösung hat, oder aber ein Paar der Form $\langle \{\}, \sigma \rangle$ erreicht wird. In diesem Fall ist σ ein Unifikator der Menge E , mit der wir gestartet sind. Es folgen die Reduktions-Regeln:

1. Falls $y \in \mathcal{V}$ eine Variable ist, die **nicht** in dem Term t auftritt, so können wir die folgende Reduktion durchführen:

$$\langle E \cup \{y \doteq t\}, \sigma \rangle \rightsquigarrow \langle E[y \mapsto t], \sigma[y \mapsto t] \rangle$$

Diese Reduktions-Regel ist folgendermaßen zu lesen: Enthält die zu untersuchende Menge von syntaktischen Gleichungen eine Gleichung der Form $y \doteq t$, wobei die Variable y nicht in t auftritt, dann können wir diese Gleichung aus der gegebenen Menge von Gleichungen entfernen. Gleichzeitig wird die Substitution σ in die Substitution $\sigma[y \mapsto t]$ transformiert und auf die restlichen syntaktischen Gleichungen wird die Substitution $[y \mapsto t]$ angewendet.

2. Wenn die Variable y in dem Term t auftritt, falls also $y \in \text{var}(t)$ ist und wenn außerdem $t \neq y$ ist, dann hat das Gleichungs-System $E \cup \{y \doteq t\}$ **keine** Lösung, wir schreiben

$$\langle E \cup \{y \doteq t\}, \sigma \rangle \rightsquigarrow \Omega.$$

3. Falls $y \in \mathcal{V}$ eine Variable ist und t keine Variable ist, so haben wir folgende Reduktions-Regel:

$$\langle E \cup \{t \doteq y\}, \sigma \rangle \rightsquigarrow \langle E \cup \{y \doteq t\}, \sigma \rangle.$$

Diese Regel wird benötigt, um anschließend eine der ersten beiden Regeln anwenden zu können.

4. Triviale syntaktische Gleichungen von Variablen können wir einfach weglassen:

$$\langle E \cup \{x \doteq x\}, \sigma \rangle \rightsquigarrow \langle E, \sigma \rangle.$$

5. Ist f ein n -stelliges Funktions-Zeichen, so gilt

$$\langle E \cup \{f(s_1, \dots, s_n) \doteq f(t_1, \dots, t_n)\}, \sigma \rangle \rightsquigarrow \langle E \cup \{s_1 \doteq t_1, \dots, s_n \doteq t_n\}, \sigma \rangle.$$

Eine syntaktische Gleichung der Form $f(s_1, \dots, s_n) \doteq f(t_1, \dots, t_n)$ wird also ersetzt durch die n syntaktischen Gleichungen $s_1 \doteq t_1, \dots, s_n \doteq t_n$.

Diese Regel ist im übrigen der Grund dafür, dass wir mit Mengen von syntaktischen Gleichungen arbeiten müssen, denn auch wenn wir mit nur einer syntaktischen Gleichung starten, kann durch die Anwendung dieser Regel die Zahl der syntaktischen Gleichungen erhöht werden.

Ein Spezialfall dieser Regel ist

$$\langle E \cup \{c \doteq c\}, \sigma \rangle \rightsquigarrow \langle E, \sigma \rangle.$$

Hier steht c für eine Konstante, also ein 0-stelliges Funktions-Zeichen. Triviale Gleichungen über Konstanten können also einfach weggelassen werden.

6. Das Gleichungs-System $E \cup \{f(s_1, \dots, s_m) \doteq g(t_1, \dots, t_n)\}$ hat **keine** Lösung, falls die Funktions-Zeichen f und g verschieden sind, wir schreiben

$$\langle E \cup \{f(s_1, \dots, s_m) \doteq g(t_1, \dots, t_n)\}, \sigma \rangle \rightsquigarrow \Omega \quad \text{falls } f \neq g.$$

Haben wir ein nicht-leeres Gleichungs-System E gegeben und starten mit dem Paar $\langle E, [] \rangle$, so lässt sich immer eine der obigen Regeln anwenden. Diese geht solange bis einer der folgenden Fälle eintritt:

1. Die 2. oder die 6. Regel ist anwendbar. Dann hat das Gleichungs-System E **keine** Lösung und als Ergebnis der Unifikation wird Ω zurück gegeben.

2. Das Paar $\langle E, [] \rangle$ wird reduziert zu einem Paar $\langle \{\}, \sigma \rangle$. Dann ist σ ein Unifikator von E . In diesem Fall schreiben wir $\sigma = \text{mgu}(E)$. Falls $E = \{s \doteq t\}$ ist, schreiben wir auch $\sigma = \text{mgu}(s, t)$. Die Abkürzung mgu steht hier für “*most general unifier*”.

Beispiel: Wir wenden das oben dargestellte Verfahren an, um die syntaktische Gleichung

$$p(x_1, f(x_4)) \doteq p(x_2, x_3)$$

zu lösen. Wir haben die folgenden Reduktions-Schritte:

$$\begin{aligned} & \langle \{p(x_1, f(x_4)) \doteq p(x_2, x_3)\}, [] \rangle \\ \rightsquigarrow & \langle \{x_1 \doteq x_2, f(x_4) \doteq x_3\}, [] \rangle \\ \rightsquigarrow & \langle \{f(x_4) \doteq x_3\}, [x_1 \mapsto x_2] \rangle \\ \rightsquigarrow & \langle \{x_3 \doteq f(x_4)\}, [x_1 \mapsto x_2] \rangle \\ \rightsquigarrow & \langle \{\}, [x_1 \mapsto x_2, x_3 \mapsto f(x_4)] \rangle \end{aligned}$$

In diesem Fall ist das Verfahren also erfolgreich und wir erhalten die Substitution

$$[x_1 \mapsto x_2, x_3 \mapsto f(x_4)]$$

als Lösung der oben gegebenen syntaktischen Gleichung. \diamond

Beispiel: Wir geben ein weiteres Beispiel und betrachten das Gleichungs-System

$$E = \{p(h(x_1, c)) \doteq p(x_2), q(x_2, d) \doteq q(h(d, c), x_4)\}$$

Wir haben folgende Reduktions-Schritte:

$$\begin{aligned} & \langle \{p(h(x_1, c)) \doteq p(x_2), q(x_2, d) \doteq q(h(d, c), x_4)\}, [] \rangle \\ \rightsquigarrow & \langle \{p(h(x_1, c)) \doteq p(x_2), x_2 \doteq h(d, c), d \doteq x_4\}, [] \rangle \\ \rightsquigarrow & \langle \{p(h(x_1, c)) \doteq p(x_2), x_2 \doteq h(d, c), x_4 \doteq d\}, [] \rangle \\ \rightsquigarrow & \langle \{p(h(x_1, c)) \doteq p(x_2), x_2 \doteq h(d, c)\}, [x_4 \mapsto d] \rangle \\ \rightsquigarrow & \langle \{p(h(x_1, c)) \doteq p(h(d, c))\}, [x_4 \mapsto d, x_2 \mapsto h(d, c)] \rangle \\ \rightsquigarrow & \langle \{h(x_1, c) \doteq h(d, c)\}, [x_4 \mapsto d, x_2 \mapsto h(d, c)] \rangle \\ \rightsquigarrow & \langle \{x_1 \doteq d, c \doteq c\}, [x_4 \mapsto d, x_2 \mapsto h(d, c)] \rangle \\ \rightsquigarrow & \langle \{x_1 \doteq d\}, [x_4 \mapsto d, x_2 \mapsto h(d, c)] \rangle \\ \rightsquigarrow & \langle \{\}, [x_4 \mapsto d, x_2 \mapsto h(d, c), x_1 \mapsto d] \rangle \end{aligned}$$

Damit haben wir die Substitution $[x_4 \mapsto d, x_2 \mapsto h(d, c), x_1 \mapsto d]$ als Lösung des anfangs gegebenen syntaktischen Gleichungs-Systems gefunden. \diamond

5.5 Ein Kalkül für die Prädikatenlogik ohne Gleichheit

In diesem Abschnitt setzen wir voraus, dass unsere Signatur Σ das Gleichheits-Zeichen nicht verwendet, denn durch diese Einschränkung wird es wesentlich einfacher, einen vollständigen Kalkül für die Prädikatenlogik einzuführen. Zwar gibt es auch für den Fall, dass die Signatur Σ das Gleichheits-Zeichen enthält, einen vollständigen Kalkül. Dieser ist allerdings deutlich aufwendiger als der Kalkül, den wir gleich einführen werden, denn der Kalkül für die Prädikatenlogik ohne das Gleichheits-Zeichen besteht nur aus zwei Schluss-Regeln, die wir jetzt definieren.

Definition 49 (Resolution) *Es gelte:*

1. k_1 und k_2 sind prädikatenlogische Klauseln,
2. $p(s_1, \dots, s_n)$ und $p(t_1, \dots, t_n)$ sind atomare Formeln,
3. die syntaktische Gleichung $p(s_1, \dots, s_n) \doteq p(t_1, \dots, t_n)$ ist lösbar mit

$$\mu = \text{mgu}(p(s_1, \dots, s_n), p(t_1, \dots, t_n)).$$

Dann ist

$$\frac{k_1 \cup \{p(s_1, \dots, s_n)\} \quad \{\neg p(t_1, \dots, t_n)\} \cup k_2}{k_1\mu \cup k_2\mu}$$

eine Anwendung der **Resolutions-Regel**. ◇

Die Resolutions-Regel ist eine Kombination aus der **Substitutions-Regel** und der Schnitt-Regel. Die Substitutions-Regel hat die Form

$$\frac{k}{k\sigma}.$$

Hierbei ist k eine prädikatenlogische Klausel und σ ist eine Substitution. Unter Umständen kann es sein, dass wir bei der Anwendung der Resolutions-Regel die Variablen in einer der beiden Klauseln erst umbenennen müssen bevor wir die Regel anwenden können. Betrachten wir dazu ein Beispiel. Die Klausel-Menge

$$M = \left\{ \{p(x)\}, \{\neg p(f(x))\} \right\}$$

ist widersprüchlich. Wir können die Resolutions-Regel aber nicht unmittelbar anwenden, denn die syntaktische Gleichung

$$p(x) \doteq p(f(x))$$

ist unlösbar. Das liegt daran, dass **zufällig** in beiden Klauseln dieselbe Variable verwendet wird. Wenn wir die Variable x in der zweiten Klausel jedoch zu y umbenennen, erhalten wir die Klausel-Menge

$$\left\{ \{p(x)\}, \{\neg p(f(y))\} \right\}.$$

Hier können wir die Resolutions-Regel anwenden, denn die syntaktische Gleichung

$$p(x) \doteq p(f(y))$$

hat die Lösung $[x \mapsto f(y)]$. Dann erhalten wir

$$\{p(x)\}, \quad \{\neg p(f(y))\} \quad \vdash \quad \{\}.$$

und haben damit die Inkonsistenz der Klausel-Menge M nachgewiesen.

Die Resolutions-Regel alleine ist nicht ausreichend, um aus einer Klausel-Menge M , die inkonsistent ist, in jedem Fall die leere Klausel ableiten zu können: Wir brauchen noch eine zweite Regel. Um das einzusehen, betrachten wir die Klausel-Menge

$$M = \left\{ \{p(f(x), y), p(u, g(v))\}, \{\neg p(f(x), y), \neg p(u, g(v))\} \right\}$$

Wir werden gleich zeigen, dass die Menge M widersprüchlich ist. Man kann nachweisen, dass mit der Resolutions-Regel alleine ein solcher Nachweis nicht gelingt. Ein einfacher, aber für die Vorlesung zu aufwendiger Nachweis dieser Behauptung kann geführt werden, indem wir ausgehend von der Menge M alle möglichen Resolutions-Schritte durchführen. Dabei würden wir dann sehen, dass die leere Klausel nie berechnet werden kann. Wir stellen daher jetzt die **Faktorisierungs-Regel** vor, mir der wir später zeigen werden, dass M widersprüchlich ist.

Definition 50 (Faktorisierung) *Es gelte*

1. k ist eine prädikatenlogische Klausel,
2. $p(s_1, \dots, s_n)$ und $p(t_1, \dots, t_n)$ sind atomare Formeln,
3. die syntaktische Gleichung $p(s_1, \dots, s_n) \doteq p(t_1, \dots, t_n)$ ist lösbar,
4. $\mu = \text{mgu}(p(s_1, \dots, s_n), p(t_1, \dots, t_n))$.

Dann sind

$$\frac{k \cup \{p(s_1, \dots, s_n), p(t_1, \dots, t_n)\}}{k\mu \cup \{p(s_1, \dots, s_n)\mu\}} \quad \text{und} \quad \frac{k \cup \{\neg p(s_1, \dots, s_n), \neg p(t_1, \dots, t_n)\}}{k\mu \cup \{\neg p(s_1, \dots, s_n)\mu\}}$$

Anwendungen der Faktorisierungs-Regel. ◇

Wir zeigen, wie sich mit Resolutions- und Faktorisierungs-Regel die Widersprüchlichkeit der Menge M beweisen lässt.

1. Zunächst wenden wir die Faktorisierungs-Regel auf die erste Klausel an. Dazu berechnen wir den Unifikator

$$\mu = \text{mgu}(p(f(x), y), p(u, g(v))) = [y \mapsto g(v), u \mapsto f(x)].$$

Damit können wir die Faktorisierungs-Regel anwenden:

$$\{p(f(x), y), p(u, g(v))\} \vdash \{p(f(x), g(v))\}.$$

2. Jetzt wenden wir die Faktorisierungs-Regel auf die zweite Klausel an. Dazu berechnen wir den Unifikator

$$\mu = \text{mgu}(\neg p(f(x), y), \neg p(u, g(v))) = [y \mapsto g(v), u \mapsto f(x)].$$

Damit können wir die Faktorisierungs-Regel anwenden:

$$\{\neg p(f(x), y), \neg p(u, g(v))\} \vdash \{\neg p(f(x), g(v))\}.$$

3. Wir schließen den Beweis mit einer Anwendung der Resolutions-Regel ab. Der dabei verwendete Unifikator ist die leere Substitution, es gilt also $\mu = []$.

$$\{p(f(x), g(v))\}, \{\neg p(f(x), g(v))\} \vdash \{\}.$$

Ist M eine Menge von prädikatenlogischen Klauseln und ist k eine prädikatenlogische Klausel, die durch Anwendung der Resolutions-Regel und der Faktorisierungs-Regel aus M hergeleitet werden kann, so schreiben wir

$$M \vdash k.$$

Dies wird als *M leitet k her* gelesen.

Definition 51 (Allabschluss) *Ist k eine prädikatenlogische Klausel und ist $\{x_1, \dots, x_n\}$ die Menge aller Variablen, die in k auftreten, so definieren wir den Allabschluss $\forall(k)$ der Klausel k als*

$$\forall(k) := \forall x_1 \dots \forall x_n : k. \quad \diamond$$

Die für uns wesentlichen Eigenschaften des Beweis-Begriffs $M \vdash k$ werden in den folgenden beiden Sätzen zusammengefasst.

Satz 52 (Korrektheits-Satz)

Ist $M = \{k_1, \dots, k_n\}$ eine Menge von Klauseln und gilt $M \vdash k$, so folgt

$$\models \forall(k_1) \wedge \dots \wedge \forall(k_n) \rightarrow \forall(k).$$

Falls also eine Klausel k aus einer Menge M hergeleitet werden kann, so ist k tatsächlich eine Folgerung aus M . □

Die Umkehrung des obigen Korrektheits-Satzes gilt nur für die leere Klausel.

Satz 53 (Widerlegungs-Vollständigkeit)

Ist $M = \{k_1, \dots, k_n\}$ eine Menge von Klauseln und gilt $\models \forall(k_1) \wedge \dots \wedge \forall(k_n) \rightarrow \perp$, so folgt

$$M \vdash \{\}.$$

□

Damit haben wir nun ein Verfahren in der Hand, um für eine gegebene prädikatenlogischer Formel f die Frage, ob $\models f$ gilt, untersuchen zu können.

1. Wir berechnen zunächst die Skolem-Normalform von $\neg f$ und erhalten dabei so etwas wie

$$\neg f \approx_e \forall x_1, \dots, x_m: g.$$

2. Anschließend bringen wir die Matrix g in konjunktive Normalform:

$$g \leftrightarrow k_1 \wedge \dots \wedge k_n.$$

Daher haben wir nun

$$\neg f \approx_e k_1 \wedge \dots \wedge k_n$$

und es gilt:

$$\models f \quad \text{g.d.w.} \quad \{\neg f\} \models \perp \quad \text{g.d.w.} \quad \{k_1, \dots, k_n\} \models \perp.$$

3. Nach dem Korrektheits-Satz und dem Satz über die Widerlegungs-Vollständigkeit gilt

$$\{k_1, \dots, k_n\} \models \perp \quad \text{g.d.w.} \quad \{k_1, \dots, k_n\} \vdash \perp.$$

Wir versuchen also, nun die Widersprüchlichkeit der Menge $M = \{k_1, \dots, k_n\}$ zu zeigen, indem wir aus M die leere Klausel ableiten. Wenn diese gelingt, haben wir damit die Allgemeingültigkeit der ursprünglich gegebenen Formel f gezeigt.

Beispiel: Zum Abschluss demonstrieren wir das skizzierte Verfahren an einem Beispiel. Wir gehen von folgenden Axiomen aus:

1. Jeder Drache ist glücklich, wenn alle seine Kinder fliegen können.
2. Rote Drachen können fliegen.
3. Die Kinder eines roten Drachens sind immer rot.

Wie werden zeigen, dass aus diesen Axiomen folgt, dass alle roten Drachen glücklich sind. Als erstes formalisieren wir die Axiome und die Behauptung in der Prädikatenlogik. Wir wählen die Signatur

$$\Sigma_{\text{Drache}} := \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \text{arity} \rangle$$

wobei die Mengen \mathcal{V} , \mathcal{F} , \mathcal{P} und arity wie folgt definiert sind:

1. $\mathcal{V} := \{x, y, z\}$.
2. $\mathcal{F} = \{\}$.
3. $\mathcal{P} := \{\text{rot}, \text{fliegt}, \text{glücklich}, \text{kind}\}$.
4. $\text{arity} := \{\langle \text{rot}, 1 \rangle, \langle \text{fliegt}, 1 \rangle, \langle \text{glücklich}, 1 \rangle, \langle \text{kind}, 2 \rangle\}$

Das Prädikat $\text{kind}(x, y)$ soll genau dann wahr sein, wenn x ein Kind von y ist. Formalisieren wir die Axiome und die Behauptung, so erhalten wir die folgenden Formeln f_1, \dots, f_4 :

1. $f_1 := \forall x : \left(\forall y : (kind(y, x) \rightarrow fliegt(y)) \rightarrow glücklich(x) \right)$
2. $f_2 := \forall x : (rot(x) \rightarrow fliegt(x))$
3. $f_3 := \forall x : (rot(x) \rightarrow \forall y : (kind(y, x) \rightarrow rot(y)))$
4. $f_4 := \forall x : (rot(x) \rightarrow glücklich(x))$

Wir wollen zeigen, dass die Formel

$$f := f_1 \wedge f_2 \wedge f_3 \rightarrow f_4$$

allgemeingültig ist. Wir betrachten also die Formel $\neg f$ und stellen fest

$$\neg f \leftrightarrow f_1 \wedge f_2 \wedge f_3 \wedge \neg f_4.$$

Als nächstes müssen wir diese Formel in eine Menge von Klauseln umformen. Da es sich hier um eine Konjunktion mehrerer Formeln handelt, können wir die einzelnen Formeln f_1 , f_2 , f_3 und $\neg f_4$ getrennt in Klauseln umwandeln.

1. Die Formel f_1 kann wie folgt umgeformt werden:

$$\begin{aligned}
 f_1 &= \forall x : \left(\forall y : (kind(y, x) \rightarrow fliegt(y)) \rightarrow glücklich(x) \right) \\
 &\leftrightarrow \forall x : \left(\neg \forall y : (kind(y, x) \rightarrow fliegt(y)) \vee glücklich(x) \right) \\
 &\leftrightarrow \forall x : \left(\neg \forall y : (\neg kind(y, x) \vee fliegt(y)) \vee glücklich(x) \right) \\
 &\leftrightarrow \forall x : \left(\exists y : \neg (\neg kind(y, x) \vee fliegt(y)) \vee glücklich(x) \right) \\
 &\leftrightarrow \forall x : \left(\exists y : (kind(y, x) \wedge \neg fliegt(y)) \vee glücklich(x) \right) \\
 &\leftrightarrow \forall x : \exists y : \left((kind(y, x) \wedge \neg fliegt(y)) \vee glücklich(x) \right) \\
 &\approx_e \forall x : \left((kind(s(x), x) \wedge \neg fliegt(s(x))) \vee glücklich(x) \right)
 \end{aligned}$$

Im letzten Schritt haben wir dabei die Skolem-Funktion s mit $arity(s) = 1$ eingeführt. Anschaulich berechnet diese Funktion für jeden Drachen x , der nicht glücklich ist, ein Kind $s(x)$, das nicht fliegen kann. Wenn wir in der Matrix dieser Formel das “ \vee ” noch ausmultiplizieren, so erhalten wir die beiden Klauseln

$$\begin{aligned}
 k_1 &:= \{kind(s(x), x), glücklich(x)\}, \\
 k_2 &:= \{\neg fliegt(s(x)), glücklich(x)\}.
 \end{aligned}$$

2. Analog finden wir für f_2 :

$$\begin{aligned}
 f_2 &= \forall x : (rot(x) \rightarrow fliegt(x)) \\
 &\leftrightarrow \forall x : (\neg rot(x) \vee fliegt(x))
 \end{aligned}$$

Damit ist f_2 zu folgender Klauseln äquivalent:

$$k_3 := \{\neg rot(x), fliegt(x)\}.$$

3. Für f_3 sehen wir:

$$\begin{aligned}
 f_3 &= \forall x : \left(rot(x) \rightarrow \forall y : (kind(y, x) \rightarrow rot(y)) \right) \\
 &\leftrightarrow \forall x : \left(\neg rot(x) \vee \forall y : (\neg kind(y, x) \vee rot(y)) \right) \\
 &\leftrightarrow \forall x : \forall y : (\neg rot(x) \vee \neg kind(y, x) \vee rot(y))
 \end{aligned}$$

Das liefert die folgende Klausel:

$$k_4 := \{\neg rot(x), \neg kind(y, x), rot(y)\}.$$

4. Umformung der Negation von f_4 liefert:

$$\begin{aligned}
 \neg f_4 &= \neg \forall x : (\text{rot}(x) \rightarrow \text{glücklich}(x)) \\
 &\leftrightarrow \neg \forall x : (\neg \text{rot}(x) \vee \text{glücklich}(x)) \\
 &\leftrightarrow \exists x : \neg(\neg \text{rot}(x) \vee \text{glücklich}(x)) \\
 &\leftrightarrow \exists x : (\text{rot}(x) \wedge \neg \text{glücklich}(x)) \\
 &\approx_e \text{rot}(d) \wedge \neg \text{glücklich}(d)
 \end{aligned}$$

Die hier eingeführte Skolem-Konstante d steht für einen unglücklichen roten Drachen. Das führt zu den Klauseln

$$\begin{aligned}
 k_5 &= \{\text{rot}(d)\}, \\
 k_6 &= \{\neg \text{glücklich}(d)\}.
 \end{aligned}$$

Wir müssen also untersuchen, ob die Menge M , die aus den folgenden Klauseln besteht, widersprüchlich ist:

1. $k_1 = \{\text{kind}(s(x), x), \text{glücklich}(x)\}$
2. $k_2 = \{\neg \text{fliegt}(s(x)), \text{glücklich}(x)\}$
3. $k_3 = \{\neg \text{rot}(x), \text{fliegt}(x)\}$
4. $k_4 = \{\neg \text{rot}(x), \neg \text{kind}(y, x), \text{rot}(y)\}$
5. $k_5 = \{\text{rot}(d)\}$
6. $k_6 = \{\neg \text{glücklich}(d)\}$

Sei also $M := \{k_1, k_2, k_3, k_4, k_5, k_6\}$. Wir zeigen, dass $M \vdash \perp$ gilt:

1. Es gilt

$$\text{mgu}(\text{rot}(d), \text{rot}(x)) = [x \mapsto d].$$

Daher können wir die Resolutions-Regel auf die Klauseln k_5 und k_4 wie folgt anwenden:

$$\{\text{rot}(d)\}, \{\neg \text{rot}(x), \neg \text{kind}(y, x), \text{rot}(y)\} \vdash \{\neg \text{kind}(y, d), \text{rot}(y)\}.$$

2. Wir wenden nun auf die resultierende Klausel und auf die Klausel k_1 die Resolutions-Regel an. Dazu berechnen wir zunächst

$$\text{mgu}(\text{kind}(y, d), \text{kind}(s(x), x)) = [y \mapsto s(d), x \mapsto d].$$

Dann haben wir

$$\{\neg \text{kind}(y, d), \text{rot}(y)\}, \{\text{kind}(s(x), x), \text{glücklich}(x)\} \vdash \{\text{glücklich}(d), \text{rot}(s(d))\}.$$

3. Jetzt wenden wir auf die eben abgeleitete Klausel und die Klausel k_6 die Resolutions-Regel an. Wir haben:

$$\text{mgu}(\text{glücklich}(d), \text{glücklich}(d)) = []$$

Also erhalten wir

$$\{\text{glücklich}(d), \text{rot}(s(d))\}, \{\neg \text{glücklich}(d)\} \vdash \{\text{rot}(s(d))\}.$$

4. Auf die Klausel $\{\text{rot}(s(d))\}$ und die Klausel k_3 wenden wir die Resolutions-Regel an. Zunächst haben wir

$$\text{mgu}(\text{rot}(s(d)), \neg \text{rot}(x)) = [x \mapsto s(d)]$$

Also liefert die Anwendung der Resolutions-Regel:

$$\{\text{rot}(s(d))\}, \{\neg \text{rot}(x), \text{fliegt}(x)\} \vdash \{\text{fliegt}(s(d))\}$$

5. Um die so erhaltenen Klausel $\{fliegt(s(d))\}$ mit der Klausel k_3 resolvieren zu können, berechnen wir

$$\text{mgu}(fliegt(s(d)), fliegt(s(x))) = [x \mapsto d]$$

Dann liefert die Resolutions-Regel

$$\{fliegt(s(d))\}, \{\neg fliegt(s(x)), glücklich(x)\} \vdash \{glücklich(d)\}.$$

6. Auf das Ergebnis $\{glücklich(d)\}$ und die Klausel k_6 können wir nun die Resolutions-Regel anwenden:

$$\{glücklich(d)\}, \{\neg glücklich(d)\} \vdash \{\}.$$

Da wir im letzten Schritt die leere Klausel erhalten haben, ist insgesamt $M \vdash \perp$ nachgewiesen worden und damit haben wir gezeigt, dass alle kommunistischen Drachen glücklich sind. \diamond

Aufgabe 3: Die von Bertrant Russell definierte *Russell-Menge* R ist definiert als die Menge aller der Mengen, die sich nicht selbst enthalten. Damit gilt also

$$\forall x : (x \in R \leftrightarrow \neg x \in x).$$

Zeigen Sie mit Hilfe des in diesem Abschnitt definierten Kalküls, dass diese Formel widersprüchlich ist.

Aufgabe 4: Gegeben seien folgende Axiome:

1. Jeder Barbier rasiert alle Personen, die sich nicht selbst rasieren.
2. Kein Barbier rasiert jemanden, der sich selbst rasiert.

Zeigen Sie, dass aus diesen Axiomen logisch die folgende Aussage folgt:

Alle Barbieri sind blond.

5.6 *Prover9* und *Mace4*

Der im letzten Abschnitt beschriebene Kalkül lässt sich automatisieren und bildet die Grundlage moderner automatischer Beweiser. Gleichzeitig lässt sich auch die Suche nach Gegenbeispielen automatisieren. Wir stellen in diesem Abschnitt zwei Systeme vor, die diesen Zwecken dienen.

1. *Prover9* dient dazu, automatisch prädikatenlogische Formeln zu beweisen.
2. *Mace4* untersucht, ob eine gegebene Menge prädikatenlogischer Formeln in einer endlichen Struktur erfüllbar ist. Gegebenenfalls wird diese Struktur berechnet.

Die beiden Programme *Prover9* und *Mace4* wurden von William McCune [McC10] entwickelt, stehen unter der **GPL** (*Gnu General Public Licence*) und können unter der Adresse

<http://www.cs.unm.edu/~mccune/prover9/download/>

im Quelltext heruntergeladen werden. Wir diskutieren zunächst *Prover9* und schauen uns anschließend *Mace4* an.

5.6.1 Der automatische Beweiser *Prover9*

Prover9 ist ein Programm, das als Eingabe zwei Mengen von Formeln bekommt. Die erste Menge von Formeln wird als Menge von *Axiomen* interpretiert, die zweite Menge von Formeln sind die zu beweisenden *Theoreme*, die aus den Axiomen gefolgert werden sollen. Wollen wir beispielsweise zeigen, dass in der Gruppen-Theorie aus der Existenz eines links-inversen Elements auch die Existenz eines rechts-inversen Elements folgt und dass außerdem das links-neutrale Element auch rechts-neutral ist, so können wir zunächst die Gruppen-Theorie wie folgt axiomatisieren:

1. $\forall x : e \cdot x = x,$
2. $\forall x : \exists y : y \cdot x = e,$
3. $\forall x : \forall y : \forall z : (x \cdot y) \cdot z = x \cdot (y \cdot z).$

Wir müssen nun zeigen, dass aus diesen Axiomen die beiden Formeln

$$\forall x : x \cdot e = x \quad \text{und} \quad \forall x : \exists y : y \cdot x = e$$

logisch folgen. Wir können diese Formeln wie in Abbildung 5.8 auf Seite 129 gezeigt für *Prover9* darstellen. Der Anfang der Axiome wird in dieser Datei durch “**formulas(sos)**” eingeleitet und durch das Schlüsselwort “**end_of_list**” beendet. Zu beachten ist, dass sowohl die Schlüsselwörter als auch die einzelnen Formel jeweils durch einen Punkt “.” beendet werden. Die Axiome in den Zeilen 2, 3, und 4 drücken aus, dass

1. **e** ein links-neutrales Element ist,
2. zu jedem Element x ein links-inverses Element y existiert und
3. das Assoziativ-Gesetz gilt.

Aus diesen Axiomen folgt, dass das **e** auch ein rechts-neutrales Element ist und dass außerdem zu jedem Element x ein rechts-neutrales Element y existiert. Diese beiden Formeln sind die zu beweisenden *Ziele* und werden in der Datei durch “**formulas(goal)**” markiert. Trägt die in Abbildung 5.8 gezeigte Datei den Namen “**group2.in**”, so können wir das Programm *Prover9* mit dem Befehl

```
prover9 -f group2.in
```

starten und erhalten als Ergebnis die Information, dass die beiden in Zeile 8 und 9 gezeigten Formeln tatsächlich aus den vorher angegebenen Axiomen folgen. Ist eine Formel nicht beweisbar, so gibt es zwei Möglichkeiten: In bestimmten Fällen kann *Prover9* tatsächlich erkennen, dass ein Beweis unmöglich ist. In diesem Fall bricht das Programm die Suche nach einem Beweis mit einer entsprechenden Meldung ab. Wenn die Dinge ungünstig liegen, ist es auf Grund der Unentscheidbarkeit der Prädikatenlogik nicht möglich zu erkennen, dass die Suche nach einem Beweis scheitern muss. In einem solchen Fall läuft das Programm solange weiter, bis kein freier Speicher mehr zur Verfügung steht und bricht dann mit einer Fehlermeldung ab.

```

1  formulas(sos).
2  all x (e * x = x).                % left neutral
3  all x exists y (y * x = e).       % left inverse
4  all x all y all z ((x * y) * z = x * (y * z)). % associativity
5  end_of_list.
6
7  formulas(goals).
8  all x (x * e = x).                % right neutral
9  all x exists y (x * y = e).       % right inverse
10 end_of_list.

```

Figure 5.8: Textuelle Darstellung der Axiome der Gruppentheorie.

Prover9 versucht, einen indirekten Beweis zu führen. Zunächst werden die Axiome in prädikatenlogische Klauseln überführt. Dann wird jedes zu beweisenden Theorem negiert und die negierte Formel wird ebenfalls in Klauseln überführt. Anschließend versucht *Prover9* aus der Menge aller Axiome zusammen mit den Klauseln, die sich aus der Negation eines der zu beweisenden Theoreme ergeben, die leere Klausel herzuleiten. Gelingt dies, so ist bewiesen, dass das jeweilige Theorem tatsächlich aus den Axiomen folgt. Abbildung 5.9 zeigt eine Eingabe-Datei für *Prover9*, bei der versucht wird, das Kommutativ-Gesetz aus den Axiomen der Gruppentheorie zu folgern. Der Beweis-Versuch mit *Prover9* schlägt allerdings fehl. In diesem Fall wird die Beweissuche nicht endlos fortgesetzt. Dies liegt daran, dass es *Prover9* gelingt, in endlicher Zeit alle aus den gegebenen Voraussetzungen folgenden Formeln abzuleiten. Leider ist ein solcher Fall eher die Ausnahme als die Regel.

```

1  formulas(sos).
2  all x (e * x = x).                % left neutral
3  all x exists y (y * x = e).        % left inverse
4  all x all y all z ((x * y) * z = x * (y * z)). % associativity
5  end_of_list.
6
7  formulas(goals).
8  all x all y (x * y = y * x).        % * is commutative
9  end_of_list.

```

Figure 5.9: Gilt das Kommutativ-Gesetz in allen Gruppen?

5.6.2 *Mace4*

Dauert ein Beweisversuch mit *Prover9* endlos, so ist zunächst nicht klar, ob das zu beweisende Theorem gilt. Um sicher zu sein, dass eine Formel nicht aus einer gegebenen Menge von Axiomen folgt, reicht es aus, eine Struktur zu konstruieren, in der alle Axiome erfüllt sind, in der das zu beweisende Theorem aber falsch ist. Das Programm *Mace4* dient genau dazu, solche Strukturen zu finden. Das funktioniert natürlich nur, solange die Strukturen endlich sind. Abbildung 5.10 zeigt eine Eingabe-Datei, mit deren Hilfe wir die Frage, ob es endliche nicht-kommutative Gruppen gibt, unter Verwendung von *Mace4* beantworten können. In den Zeilen 2, 3 und 4 stehen die Axiome der Gruppen-Theorie. Die Formel in Zeile 5 postuliert, dass für die beiden Elemente a und b das Kommutativ-Gesetz nicht gilt, dass also $a \cdot b \neq b \cdot a$ ist. Ist der in Abbildung 5.10 gezeigte Text in einer Datei mit dem Namen “*group.in*” gespeichert, so können wir *Mace4* durch das Kommando

mace4 -f group.in

starten. *Mace4* sucht für alle positiven natürlichen Zahlen $n = 1, 2, 3, \dots$, ob es eine Struktur $\mathcal{S} = \langle \mathcal{U}, \mathcal{T} \rangle$ mit $\text{card}(\mathcal{U}) = n$ gibt, in der die angegebenen Formeln gelten. Bei $n = 6$ wird *Mace4* fündig und berechnet tatsächlich eine Gruppe mit 6 Elementen, in der das Kommutativ-Gesetz verletzt ist.

```

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 5.10: Gibt es eine Gruppe, in der das Kommutativ-Gesetz nicht gilt?

Abbildung 5.11 zeigt einen Teil der von *Mace4* produzierten Ausgabe. Die Elemente der Gruppe sind die Zahlen $0, \dots, 5$, die Konstante a ist das Element 0, b ist das Element 1, e ist das Element 2. Weiter sehen wir, dass das Inverse von 0 wieder 0 ist, das Inverse von 1 ist 1 das Inverse von 2 ist 2, das Inverse von 3 ist 4, das Inverse von 4 ist 3 und das Inverse von 5 ist 5. Die Multiplikation wird durch die folgende Gruppen-Tafel realisiert:

\circ	0	1	2	3	4	5
0	2	3	0	1	5	4
1	4	2	1	5	0	3
2	0	1	2	3	4	5
3	5	0	3	4	2	1
4	1	5	4	2	3	0
5	3	4	5	0	1	2

Diese Gruppen-Tafel zeigt, dass

$$a \circ b = 0 \circ 1 = 3, \quad \text{aber} \quad b \circ a = 1 \circ 0 = 4$$

gilt, mithin ist das Kommutativ-Gesetz tatsächlich verletzt.

```

1  ===== DOMAIN SIZE 6 =====
2
3  === Mace4 starting on domain size 6. ===
4
5  ===== MODEL =====
6
7  interpretation( 6, [number=1, seconds=0], [
8
9      function(a, [ 0 ]),
10
11     function(b, [ 1 ]),
12
13     function(e, [ 2 ]),
14
15     function(f1(_), [ 0, 1, 2, 4, 3, 5 ]),
16
17     function(*(_,_), [
18         2, 3, 0, 1, 5, 4,
19         4, 2, 1, 5, 0, 3,
20         0, 1, 2, 3, 4, 5,
21         5, 0, 3, 4, 2, 1,
22         1, 5, 4, 2, 3, 0,
23         3, 4, 5, 0, 1, 2 ])
24 ]).
25
26  ===== end of model =====

```

Figure 5.11: Ausgabe von *Mace4*.

Bemerkung: Der Theorem-Beweiser *Prover9* ist ein Nachfolger des Theorem-Beweisers *Otter*. Mit Hilfe von *Otter* ist es McCune 1996 gelungen, die Robbin'sche Vermutung zu beweisen [McC97]. Dieser Beweis war damals sogar der *New York Times* eine Schlagzeile wert, nachzulesen unter

<http://www.nytimes.com/library/cyber/week/1210math.html>.

Dies zeigt, dass automatische Theorem-Beweiser durchaus nützliche Werkzeuge sein können. Nichtsdestoweniger ist die Prädikatenlogik unentscheidbar und bisher sind nur wenige offene mathematische Probleme mit Hilfe von automatischen Beweisern gelöst worden. Das wird sich vermutlich auch in der näheren Zukunft nicht ändern.

◇

Bibliography

- [DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
- [DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.
- [McC97] William McCune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19:263–276, December 1997.
- [McC10] William McCune. Prover9 and Mace4. <http://www.cs.unm.edu/~mccune/prover9/>, 2005–2010.
- [MMZ⁺01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC’01)*, 2001.
- [Ric53] Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 83, 1953.
- [SDSD86] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming With Sets: An Introduction to SETL*. Springer-Verlag, 1986.
- [Tur36] Alan M. Turing. On computable numbers, with an application to the “Entscheidungsproblem”. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.