Theoretical Computer Science I:
Logic
— Winter 2016/2017 —
DHBW Mannheim

Prof. Dr. Karl Stroetmann

October 4, 2016

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

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 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 the tools of 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*

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

   while first-order logic also investigates the quantifiers

   $$\text{“}\forall\text{” and “}\exists\text{”},$$

   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.

5. The last chapter deals with program verification.

   The correctness of programs can be verified mathematically. We will discuss two methods in this lecture: *Computational induction* can be used to verify the correctness of recursive procedures, while *symbolic execution* can be used to verify the correctness of iterative procedures. (An iterative procedure uses loops instead of recursion.)

Remark: I am currently translating these lecture notes from German to English. Hence, I expect several typos to be present. If you find any mistakes, typos or otherwise, I would like you to report these mistakes to me via the following address:

karl.stroetmann@dhbw-mannheim.de

Alternatively, you can report these mistakes via discord. Reporting these mistakes to me vocally after the lecture is no use as I tend to forget them before I have a chance to correct them.

‘

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 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 sections 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 2.1 on page 8. 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 >~

=>
```

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] 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============================v2.5.0=-

Welcome to the setlX interpreter!

Open Source Software from http://setlX.randoom.org/
(c) 2011-2016 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.

```
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:

```
Hello, World!
~< Result: om >~

=>
```

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 value of $36 \cdot 37/2$, can be achieved vie the following commmand:

```
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 9 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:
```

to the screen. After entering a natural number n and hitting the `enter` key, the program will compute the set $\{1, \cdots, n\}$ 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 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: Ein einfaches Programm zur Berechnung der Summe $\sum_{i=1}^{n} i$.

Let us discuss the program shown in Figure 2.2 on page 9 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 comments that extends to the end of the line. In order to have multi-line comments, we can use the strings "/*" and "*/". Every text starting with the string "/*" and ending with the string "*/" is ignored.

   Note that multi-line comments can not be nested.

2. The second line is an assignment. The function *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

   $$\{\ a\ \texttt{..}\ b\ \}$$

   evaluates to the set

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

   The operator "+/" computes the sum of all elements of the set

   $$\{i \in \mathbb{N} \mid 1 \leq i \wedge i \leq n\}.$$

   Of course, this is exactly 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 10 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("Zahl eingeben: ");
10  total := sum(n);
11  print("Sum 0 + 1 + 2 + ... + ", n, " = ", total);
```

Figure 2.3: Ein rekursives Programm zur Berechnung der Summe $\sum_{i=0}^{n} i$.

1. The first seven lines define the procedure `sum`. In SetlX, the definition of a procedure is started with the key word "`procedure`". The keyword is followed by the list of arguments. These arguments are separated by the character "`,`" and are enclosed in parenthesis. As in the programming language C, the body of the procedure is enclosed with 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 $body_1$ are executed. Here, $body_1$ is a list of statements.

   (c) Otherwise, the statements in $body_2$ are executed.

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

   (a) In SetlX, we have to enclose $body_1$ and $body_2$ 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 "`;`".

2. After defining the procedure `sum`, line 9 reads a number that is assigned to the variable `n`.

3. Next, line 10 calls the procedure `sum` for the given value of `n`. This value is then assigned to the variable `total`.

4. 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:

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

These equations become evident if we substitute the definition

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

in these equations, since we have:

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

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 11 shows the file `simple.stlx`. We discuss it line by line.

```
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          2^a
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: Berechnung von ∪, ∩, \ und Potenz-Menge

1. The first two lines show that sets can be defined as explicit enumerations of their elements.
2. 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 "∪" corresponds to "+", "∩" corresponds to "*", while "\" corresponds to "-".
3. LIne 13 computes the power set of the set a.
4. Line 16 checks whether a is a subset of b.
5. Line 18 checks whether the number 1 is an element of the set a.
6. Line 20 computes the cartesian product of a and b. The set operator "×" 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 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 to 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 or equal 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

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

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:

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

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 that have both factors less than or equal to 10. (A product of the form $a \cdot b$ is called *trivial* if and only if either of the factors a or b is equal to 1.) A mathematical notation for the set p is as follows:

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

Iterators can be quite useful. For example, consider the program `primes-difference.stlx` that is shown in Figure 2.5 on page 13. This program computes the set of all prime numbers less than n. The underlying idea is that a number is prime iff[2] it can not be written as a non-trivial product. Hence, if we take the set of all numbers less than or equal to n 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: Programm zur Berechnung der Primzahlen bis n.

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

$$\{\mathit{expr} : x_1 \texttt{ in } S_1,\ \cdots,\ x_n \texttt{ in } S_n\}\ .$$

Here, *expr* is a term that makes use of the variables $x_1, \cdots, x_n$. Furthermore, $S_1, \cdots, S_n$ are expressions that return sets (or lists) when they are evaluated. Here, an expression of the form "x_i `in` S_i" is called an *iterator* 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

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

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:

$$M := \{\mathit{expr} : x_1 \texttt{ in } S_1,\ \cdots,\ x_n \texttt{ in } S_n \mid \mathit{cond}\}.$$

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

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

i.e. M is defined as the set of all those values that we get when we substitute those values x_i form 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` contains the set of all prime numbers that are less than 100. The idea is 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

[2] Henceforth, the word "iff" is used as an abbreviation for "if and only if".

number x we can use the operator % in SetlX: The expression p % x computes the rest that is left over after p is divided by x. Hence,

```
{ x : x in {1..p} | 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 14 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 "⟨" and "⟩" 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 $x \in \mathit{dom}(R)$, 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 14 shows this more concretely. Furthermore, the program shows that for a binary relation R, in SetlX we write $\mathit{dom}(R)$ as domain(R) and $\mathit{rng}(R)$ 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.

```
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.

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

The relation Q that is computed in line 1 of Figure 2.7 represents the function $x \mapsto x^2$ on the set $\{1, \cdots, 10\}$. 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 15 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 follows:

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

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 sets as arithmetic progressions, iterations, or via selection. The program primes-tuple.stlx in Figure 2.9 on page 15 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 14 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

Das Programm sort.stlx in Abbildung 2.10 auf Seite 16 zeigt ein einfaches Verfahren um eine Liste von natürlichen Zahlen zu sortieren. Der Ausdruck

```
max(s)
```

berechnet das größte Element der Liste s. Damit läuft die Variable n in dem Iterator

```
n in [1 .. max(s)]
```

dann von 1 bis zur größten in s auftretenden Zahl. Aufgrund der Bedingung "`n in s`" wird die Zahl n genau dann in die resultierende Liste eingefügt, wenn n ein Element der Liste s ist. Da der Iterator

"`n in [1 .. max(s)]`"

die Zahlen der Reihe nach aufzählt, ist das Ergebnis eine sortierte Liste, die genau die Zahlen enthält, die Elemente von s sind.

Offensichtlich ist der in der Prozedur `sort()` implementierte Algorithmus nicht sehr effizient. Wir werden später noch effizientere Algorithmen diskutieren.

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

Figure 2.10: Sortieren einer Menge.

Analog zu der Funktion `max()` gibt es noch die Funktion `min()` die das Minimum einer Menge oder Liste berechnet.

Weiterhin können die Operatoren "`+/`" und "`*/`" auf Mengen angewendet werden. Der Operator "`+/`" berechnet die Summe aller Elemente einer Menge, während der Operator "`*/`" das Produkt der Elemente berechnet. Ist die zu Grunde liegende Menge oder Liste leer, so geben diese Operatoren "`+/`" als Ergebnis "`om`" zurück. Die Operatoren "`+/`" und "`*/`" können auch als binäre Operatoren verwendet werden. Ein Ausdruck der Form

```
x +/ s
```

gibt als Ergebnis den Wert x zurück, wenn die Menge s leer ist. Falls `s` nicht leer ist, liefert der obige Ausdruck die Summe der Elemente der Menge `s`. Ein Ausdruck der Form "`x */ s`" funktioniert analog: Falls `s` leer ist, wird `x` zurück gegeben, sonst ist das Ergebnis das Produkt der Elemente aus `s`.

Als nächstes besprechen wir die Funktion "`from`", mit dem wir ein (nicht näher spezifiziertes) Element aus einer Menge auswählen können. Die Syntax ist:

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

Hierbei ist s eine Menge und x eine Variable. Wird diese Anweisung ausgeführt, so wird ein nicht näher spezifiziertes Element aus der Menge s entfernt. Dieses Element wird darüber hinaus der Variablen x zugewiesen. Falls s leer ist, so erhält x den undefinierten Wert "`om`" und s bleibt unverändert. Das Programm from.stlx in Abbildung 2.11 auf Seite 17 nutzt diese Anweisung um eine Menge elementweise auszugeben. Jedes Element wird dabei in einer eigenen Zeile ausgedruckt.

Neben der Funktion "`from`" gibt es noch die Funktion "`arb`", die ein beliebiges Element aus einer Menge auswählt, die Menge selbst aber unverändert lässt. Nach den Befehlen

```
s := { 1, 2 };
x := arb(s);
print("x = ", x);
print("s = ", s);
```

```
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: Menge elementweise ausdrucken.

erhalten wir die folgende Ausgabe:

```
x = 13
s = {2, 3, 5, 7, 13}
```

Weiterhin steht für Listen der Operator "+" zur Verfügung, mit dem zwei Listen aneinander gehängt werden können. Außerdem gibt es noch den unären Operator "#", der für Mengen und Listen die Anzahl der Elemente berechnet. Schließlich kann man Elemente von Listen mit der Schreibweise

x := t[n];

indizieren. In diesem Fall muss t eine Liste sein, die mindestens die Länge n hat. Die obige Anweisung weist der Variablen x dann den Wert des n-ten Elementes der Liste t zu. Die obige Zuweisung lässt sich auch umdrehen: Mit

t[n] := x;

wird die Liste t so abgeändert, dass das n-te Element danach den Wert x hat. Im Gegensatz zu der Sprache C werden in SetlX Listen mit 1 beginnend indiziert, falls wir die beiden Befehle

```
l := [ 1, 2, 3 ];
x := l[1];
```

ausführen, hat x also anschließend den Wert 1. Wollen wir auf das letzte Element einer Liste l zugreifen, so können wir dafür den Ausdruck

```
l[#l]
```

benutzen, denn #l gibt die Anzahl der Elemente der Liste l an. Alternativ können wir aber auch den Ausdruck

```
l[-1]
```

benutzen um auf das letzte Element von l zuzugreifen. Analog greift der Ausdruck l[-2] auf das vorletzte Element der Liste l zu.

Das Programm simple-tuple.stlx in Abbildung 2.12 auf Seite 18 demonstriert die eben vorgestellten Operatoren. Zusätzlich sehen wir in Zeile 19, dass SetlX simultane Zuweisungen unterstützt. Das Programm produziert die folgende Ausgabe:

```
[1, 2, 3] + [2, 3, 4, 5, 6] = [1, 2, 3, 2, 3, 4, 5, 6]
# {5, 6, 7} = 3
# [1, 2, 3] = 3
[2, 3, 4, 5, 6][3] = 4
b = [2, 3, 4, 5, 6, om, om, om, om, 42]
d = [3, 4, 5]
x = 2, y = 1
b[-1] = 42
```

```
a := [ 1, 2, 3 ];
b := [ 2, 3, 4, 5, 6 ];
c := { 5, 6, 7 };
// Aneinanderh\"{a}ngen von Tupeln mit +
print(a, " + ", b, " = ", a + b);
// Berechnung der Anzahl der Elemente einer Menge
print("# ", c, " = ", # c);
// Berechnung der L\"{a}nge eines Tupels
print("# ", a, " = ", # a);
// Auswahl des dritten Elements von b
print(b, "[3] = ", b[3] );
// \"{U}berschreiben des 10. Elements von b
b[10] := 42;
print("b = ", b);
// Auswahl einer Teilliste
d := b[2..4];
print( "d = ", d);
x := 1;  y := 2;
[ x, y ] := [ y, x ];
print("x = ", x, ", y = ", y);
print("b[-1] = ", b[-1]);
```

Figure 2.12: Weitere Operatoren auf Tupeln und Mengen.

2.5.1 Anwendung: *Sortieren durch Auswahl*

```
minSort := procedure(l) {
    if (l == []) {
       return [];
    }
    m := min(l);
    return [x : x in l | x == m] + minSort([ x : x in l | x != m ]);
};

l := [ 13, 5, 13, 7, 2, 4 ];
print( "sort( ", l, " ) = ", minSort(l) );
```

Figure 2.13: Implementierung des Algorithmus *Sortieren durch Auswahl*.

Als praktische Anwendung zeigen wir eine Implementierung des Algorithmus *Sortieren durch Auswahl*. Dieser Algorithmus, dessen Aufgabe es ist, eine gegebene Liste L zu sortieren, kann wie folgt beschrieben werden:

1. Falls L leer ist, so ist auch $\mathit{sort}(L)$ leer:

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

2. Sonst berechnen wir das Minimum m von L:

   $$m = \min(L).$$

   Dann entfernen wir m aus der Liste L und sortieren die Restliste rekursiv:

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

Abbildung 2.13 auf Seite 18 zeigt das Programm `min-sort.stlx`, das diese Idee in SetlX umsetzt.

2.6 Kontroll-Strukturen

Die Sprache SetlX stellt alle Kontroll-Strukturen zur Verfügung, die in modernen Sprachen üblich sind. Wir haben "`if`"-Abfragen bereits mehrfach gesehen. In der allgemeinsten Form hat eine Fallunterscheidung die in Abbildung 2.14 auf Seite 19 gezeigte Struktur.

```
1     if (test_0) {
2         body_0
3     } else if (test_1) {
4         body_1
5         ⋮
6     } else if (test_n) {
7         body_n
8     } else {
9         body_{n+1}
10    }
```

Figure 2.14: Struktur der allgemeinen Fallunterscheidung.

Hierbei steht $test_i$ für einen Test, der "`true`" oder "`false`" liefert. Liefert der Test "`true`", so wird der zugehörigen Anweisungen in $body_i$ ausgeführt, andernfalls wird der nächste Test $test_{i+1}$ versucht. Schlagen alle Tests fehl, so wird $body_{n+1}$ ausgeführt.

Die Tests selber können dabei die binären Operatoren

"==", "`!=`", ">", "<", ">=", "<=", "`in`",

verwenden. Dabei steht "==" für den Vergleich auf Gleichheit, "`!=`" für den Vergleich auf Ungleichheit. Für Zahlen führen die Operatoren ">", "<", ">=" und "<=" dieselben Größenvergleiche durch wie in der Sprache C. Für Mengen überprüfen diese Operatoren analog, ob die entsprechenden Teilmengen-Beziehung erfüllt ist. Der Operator "`in`" überprüft, ob das erste Argument ein Element der als zweites Argument gegeben Menge ist: Der Test

x `in` S

hat genau dann den Wert `true`, wenn $x \in S$ gilt. Aus den einfachen Tests, die mit Hilfe der oben vorgestellten Operatoren definiert werden können, können mit Hilfe der Junktoren "&&" (logisches *und*), "`||`" (logisches *oder*) und "`!`" (logisches *nicht*) komplexere Tests aufgebaut werden. Dabei bindet der Operator "`||`" am schwächsten und der Operator "`!`" bindet am stärksten. Ein Ausdruck der Form

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

wird also als

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

geklammert. Damit haben die logischen Operatoren dieselbe Präzedenz wie in der Sprache C.

SetlX unterstützt die Verwendung von Quantoren. Die Syntax für die Verwendung des Allquantors ist

`forall (` x `in` s `|` *cond* `)`

Hier ist s eine Menge und *cond* ist ein Ausdruck, der von der Variablen x abhängt und einen Wahrheitswert als Ergebnis zurück liefert. Die Auswertung des oben angegebenen Allquantors liefert genau dann `true` wenn die Auswertung von *cond* für alle Elemente der Menge s den Wert `true` ergibt. Abbildung 2.15 auf Seite 20 zeigt das Programm `primes-forall.stlx`, welches die Primzahlen mit Hilfe eines Allquantors berechnet. Die Bedingung

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

trifft auf genau die Zahlen p zu, für die gilt, dass alle Teiler Elemente der Menge $\{1, p\}$ sind. Dies sind genau die Primzahlen.

```
1  isPrime := procedure(p) {
2      return 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  print([ p : p in [2..n] | isPrime(p) ]);
```

Figure 2.15: Berechnung der Primzahlen mit Hilfe eines Allquantors

Neben dem Allquantor gibt es noch den Existenzquantor. Die Syntax ist:

```
exists (x in s | cond)
```

Hierbei ist wieder eine s eine Menge und *cond* ist ein Ausdruck, der zu `true` oder `false` ausgewertet werden kann. Falls es wenigstens ein x gibt, so dass die Auswertung von *cond* true ergibt, liefert die Auswertung des Existenzquantor ebenfalls den Wert `true`.

Bemerkung: Liefert die Auswertung eines Ausdrucks der Form

```
exists (x in s | cond)
```

den Wert `true`, so wird **zusätzlich** der Variablen x ein Wert zugewiesen, für den die Bedingung *cond* erfüllt ist. Falls die Auswertung des Existenzquantors den Wert `false` ergibt, ändert sich der Wert von x nicht.

`Switch`-Blöcke

Als Alternative zur Fallunterscheidung mit Hilfe von `if-then-else`-Konstrukten gibt es noch den `switch`-Block. Ein solcher Block hat die in Abbildung 2.16 auf Seite 20 gezeigte Struktur. Bei der Abarbeitung werden der Reihe nach die Tests $test_1, \cdots, test_n$ ausgewertet. Für den ersten Test $test_i$, dessen Auswertung den Wert `true` ergibt, wird der zugehörige Block $body_i$ ausgeführt. Nur dann, wenn alle Tests $test_1, \cdots, test_n$ scheitern, wird der Block $body_{n+1}$ hinter dem Schlüsselwort `default` ausgeführt. Den selben Effekt könnte man natürlich auch mit einer `if-else if-`$\cdots$`-else if-else`-Konstruktion erreichen, nur ist die Verwendung eines `switch`-Blocks oft übersichtlicher.

```
1  switch {
2      case test_1 : body_1
3      :
4      case test_n : body_n
5      default: body_{n+1}
6  }
```

Figure 2.16: Struktur eines `switch`-Blocks

Abbildung 2.17 zeigt das Programm `switch.stlx`, bei dem es darum geht, in Abhängigkeit von der letzten Ziffer einer Zahl eine Meldung auszugeben. Bei der Behandlung der Aussagenlogik werden wir noch realistischere Anwendungs-Beispiele für den `switch`-Block kennenlernen.

```
print("Zahl eingeben:");
n := read();
m := n % 10;
switch {
    case m == 0 : print("letzte Ziffer ist 0");
    case m == 1 : print("letzte Ziffer ist 1");
    case m == 2 : print("letzte Ziffer ist 2");
    case m == 3 : print("letzte Ziffer ist 3");
    case m == 4 : print("letzte Ziffer ist 4");
    case m == 5 : print("letzte Ziffer ist 5");
    case m == 6 : print("letzte Ziffer ist 6");
    case m == 7 : print("letzte Ziffer ist 7");
    case m == 8 : print("letzte Ziffer ist 8");
    case m == 9 : print("letzte Ziffer ist 9");
    default     : print("impossible");
}
```

Figure 2.17: Anwendung eines `switch`-Blocks

In der Sprache C gibt es eine analoge Konstruktion. In C ist es so, dass nach einem Block, der nicht durch einen `break`-Befehl abgeschlossen wird, auch alle folgenden Blocks ausgeführt werden. Dies ist in SetlX anders: Dort wird immer genau ein Block ausgeführt.

Neben dem `switch`-Block gibt es noch dem `match`-Block, den wir allerdings erst später diskutieren werden.

2.6.1 Schleifen

Es gibt in SetlX Kopf-gesteuerte Schleifen (`while`-Schleifen) und Schleifen, die über die Elemente einer Menge oder einer Liste iterieren (`for`-Schleifen). Wir diskutieren diese Schleifenformen jetzt im Detail.

`while`-Schleifen

Die allgemeine Syntax der `while`-Schleife ist in Abbildung 2.18 auf Seite 21 gezeigt. Hierbei ist *test* ein Ausdruck, der zu Beginn ausgewertet wird und der "`true`" oder "`false`" ergeben muss. Ergibt die Auswertung "`false`", so ist die Auswertung der `while`-Schleife bereits beendet. Ergibt die Auswertung allerdings "`true`", so wird anschließend *body* ausgewertet. Danach beginnt die Auswertung der Schleife dann wieder von vorne, d.h. es wird wieder *test* ausgewertet und danach wird abhängig von dem Ergebnis dieser wieder *body* ausgewertet. Das ganze passiert so lange, bis irgendwann einmal die Auswertung von *test* den Wert "`false`" ergibt. Die von SetlX unterstützten `while`-Schleifen funktionieren genauso wie in der Sprache C.

```
while (test) {
    body
}
```

Figure 2.18: Struktur der `while`-Schleife

Abbildung 2.19 auf Seite 22 zeigt das Programm `primes-while.stlx`, das die Primzahlen mit Hilfe einer `while`-Schleife berechnet. Hier ist die Idee, dass eine Zahl genau dann Primzahl ist, wenn es keine kleinere Primzahl gibt, die diese Zahl teilt.

```
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 := primes + { p };
8       }
9       p := p + 1;
10  }
```

Figure 2.19: Iterative Berechnung der Primzahlen.

`for`-Schleifen

Die allgemeine Syntax der `for`-Schleife ist in Abbildung 2.20 auf Seite 22 gezeigt. Hierbei ist s eine Menge, eine Liste, oder ein String und x ist der Name einer Variablen. Diese Variable wird nacheinander mit allen Werten aus der Menge (oder Liste) s belegt und anschließend wird mit dem jeweiligen Wert von x der Schleifenrumpf *body* ausgeführt. Falls s ein String ist, dann iteriert die Schleife über die Buchstaben aus s.

```
for (x in s) {
    body
}
```

Figure 2.20: Struktur der `for`-Schleife.

Abbildung 2.21 auf Seite 22 zeigt das Programm `primes-for.stlx`, das die Primzahlen mit Hilfe einer `for`-Schleife berechnet. Der dabei verwendete Algorithmus ist als das *Sieb des Eratosthenes* bekannt. Das funktioniert wie folgt: Sollen alle Primzahlen kleiner oder gleich n berechnet werden, so wird zunächst ein Tupel der Länge n gebildet, dessen i-tes Element den Wert i hat. Das passiert in Zeile 2. Anschließend werden alle Zahlen, die Vielfache von 2, 3, 4, $\cdots$ sind, aus der Menge der Primzahlen entfernt, indem die Zahl, die an dem entsprechenden Index in der Liste `primes` steht, auf 0 gesetzt wird. Dazu sind zwei Schleifen erforderlich: Die äußere `for`-Schleife iteriert i über alle Werte von 2 bis n. Die innere `while`-Schleife iteriert dann für gegebenes i über alle Werte $i \cdot j$, für die das Produkt $i \cdot j \leq n$ ist. Schließlich werden in der letzten Zeile alle die Indizes i ausgedruckt, für die `primes`$[i]$ nicht auf 0 gesetzt worden ist, denn das sind genau die Primzahlen.

```
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 := j + 1;
8       }
9   }
10  print({ i : i in [2 .. n] | primes[i] > 0 });
```

Figure 2.21: Berechnung der Primzahlen nach Eratosthenes.

Der Algorithmus aus Abbildung 2.21 kann durch die folgende Beobachtungen noch verbessert werden:

1. Es reicht aus, wenn j mit i initialisiert wird, denn alle kleineren Vielfachen von i wurden bereits vorher auf 0 gesetzt.

2. Falls in der äußeren Schleife die Zahl i keine Primzahl ist, so bringt es nichts mehr, die innere `while`-Schleife in den Zeilen 9 bis 11 zu durchlaufen, denn alle Indizes, für die dort `primes[i*j]` auf 0 gesetzt wird, sind schon bei dem vorherigen Durchlauf der äußeren Schleife, bei der `primes[i]` auf 0 gesetzt wurde, zu 0 gesetzt worden.

   Abbildung 2.22 auf Seite 23 zeigt das Programm `primes-eratosthenes.stlx`, das diese Ideen umsetzt. Um den Durchlauf der inneren while Schleife in dem Fall, dass `primes[i] = 0` ist, zu überspringen, haben wir den Befehl "`continue`" benutzt. Der Aufruf von "`continue`" bricht die Abarbeitung des Schleifen-Rumpfs für den aktuellen Wert von `i` ab, weist der Variablen `i` den nächsten Wert aus `[1..n]` zu und fährt dann mit der Abarbeitung der Schleife in Zeile 4 fort. Der Befehl "`continue`" verhält sich also genauso, wie der Befehl "`continue`" in der Sprache *C*.

```
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 := j + 1;
11      }
12  }
13  print({ i : i in [2 .. n] | primes[i] > 0 });
```

Figure 2.22: Effizientere Berechnung der Primzahlen nach Eratosthenes.

2.6.2 Fixpunkt-Algorithmen

Angenommen, wir wollen in der Menge $\mathbb{R}$ der reellen Zahlen die Gleichung

$$x = \cos(x)$$

lösen. Ein naives Verfahren, das hier zum Ziel führt, basiert auf der Beobachtung, dass die Folge $(x_n)_n$, die durch

$$x_0 := 0 \text{ und } x_{n+1} := \cos(x_n) \text{ für alle } n \in \mathbb{N}$$

definiert ist, gegen eine Lösung der obigen Gleichung konvergiert. Damit führt das in Abbildung 2.23 auf Seite 24 angegebene Programm `solve.stlx` zum Ziel.

Bei dieser Implementierung wird die Schleife in dem Moment abgebrochen, wenn die Werte von `x` und `old_x` nahe genug beieinander liegen. Dieser Test kann aber am Anfang der Schleife noch gar nicht durchgeführt werden, weil da die Variable `old_x` noch gar keinen Wert hat. Daher können wir diesen Test erst ausführen, wenn der nächste Wert von `x` berechnet worden ist, denn erst dann haben wir zwei Werte, die wir vergleichen können. Ist der Test erfolgreich, so brechen wir die Schleife mit Hilfe des Kommandos "`break`" ab. In der Sprache *C* hat das Kommando "`break`" dieselbe Funktion.

Abbildung 2.24 zeigt das Programm `fixpoint.stlx`. Hier ist eine generische Funktion `solve` implementiert, die zwei Argumente verarbeitet:

1. Das Argument `f` ist die Funktion, für die ein Fixpunkt berechnet werden soll.

2. Das Argument `x0` gibt den Startwert der Fixpunkt-Iteration an.

```
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 2.23: Lösung der Gleichung $x = cos(x)$ durch Iteration.

```
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-15) {
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 2.24: Eine generische Implementierung des Fixpunkt-Algorithmus

.

In Zeile 11 haben wir die Funktion `solve` aufgerufen, um die Gleichung $x = \cos(x)$ zu lösen. In Zeile 12 lösen wir die Gleichung

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

die zu der quadratischen Gleichung $x^2 + x = 1$ äquivalent ist. Beachten Sie, dass wir die Funktion $x \mapsto \frac{1}{1+x}$ in SetlX durch den Ausdruck

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

definieren können. Hier habe ich die Fließkomma-Zahl 1.0 verwendet, da SetlX sonst mit rationalen Zahlen rechnen würde, was wesentlich aufwendiger ist.

Ganz nebenbei zeigt das obige Beispiel auch, dass Sie in SetlX nicht nur mit rationalen Zahlen, sondern auch mit reellen Zahlen rechnen können. Eine Zahlen-Konstante, die den Punkt "`.`" enthält, wird automatisch als reelle Zahl erkannt und auch so abgespeichert. In SetlX stehen unter anderem die folgenden reellen Funktionen zur Verfügung:

1. Der Ausdruck `sin`(x) berechnet den Sinus von x. Außerdem stehen die trigonometrischen Funktionen `cos`(x) und `tan`(x) zur Verfügung. Die Umkehr-Funktionen der trigonometrischen Funktionen sind `asin`(x), `acos`(x) und `atan`(x).

   Der Sinus Hyperbolicus von x wird durch `sinh`(x) berechnet. Entsprechend berechnet `cosh`(x) den Kosinus Hyperbolicus und `tanh`(x) den Tangens Hyperbolicus.

2. Der Ausdruck `exp`(x) berechnet die Potenz zur Basis e, es gilt

$$\texttt{exp}(x) = e^x.$$

3. Der Ausdruck $\texttt{log}(x)$ berechnet den natürlichen Logarithmus von x. Der Logarithmus zur Basis 10 von x wird durch $\texttt{log10}(x)$ berechnet.

4. Der Ausdruck $\texttt{abs}(x)$ berechnet den Absolut-Betrag von x, während $\texttt{signum}(x)$ das Vorzeichen von x berechnet.

5. Der Ausdruck $\texttt{sqrt}(x)$ berechnet die Quadrat-Wurzel von x, es gilt

   $$\texttt{sqrt}(x) = \sqrt{x}.$$

6. Der Ausdruck $\texttt{cbrt}(x)$ berechnet die dritte Wurzel von x, es gilt

   $$\texttt{cbrt}(x) = \sqrt[3]{x}.$$

7. Der Ausdruck $\texttt{ceil}(x)$ berechnet die kleinste ganze Zahl, die größer oder gleich x ist, es gilt

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

   Die Funktion `ceil` rundet ihr Argument also immer auf.

8. Der Ausdruck $\texttt{floor}(x)$ berechnet die größte ganze Zahl, die kleiner oder gleich x ist, es gilt

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

   Die Funktion `floor` rundet ihr Argument also immer ab.

9. Der Ausdruck $\texttt{round}(x)$ rundet x zu einer ganzen Zahl.

Darüber hinaus unterstützt SetlX die Verwendung von rationalen Zahlen. Die Eingabe von

```
1/2 + 1/3;
```

liefert das Ergebnis `5/6`. Im Unterschied zu der Sprache C kann es bei rationalen Zahlen keinen Überlauf geben. Damit können wir die rationalen Zahlen benutzen, um mit beliebig hoher Genauigkeit zu rechnen. Beispielsweise kann die Euler'sche Zahl `e` mit Hilfe der Formel

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

berechnet werden. In SetlX können wir `e` mit Hilfe des Kommandos

```
nDecimalPlaces(+/ { 1/n! : n in {0..50} }, 50);
```

auf 50 Stellen Genauigkeit berechnen. Wir erhalten den Wert

`1.71828182845904523536028747135266249775724709369995`.

2.6.3 Verschiedenes

Der Interpreter bietet die Möglichkeit, komplexe Programme zu laden. Der Befehl

```
load(file);
```

lädt das Programm, dass sich in der Datei *file* befindet und führt die in dem Programm vorhandenen Befehle aus. Führen wir beispielsweise den Befehl

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

im Interpreter aus und enthält die Datei "`primes-forall.stlx`" das in Abbildung 2.15 auf Seite 20 gezeigte Programm, so können wir anschließend mit den in dieser Datei definierten Variablen arbeiten. Beispielsweise liefert der Befehl

```
print(isPrime);
```

die Ausgabe:

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

Zeichenketten, auch bekannt als *Strings*, werden in SetlX in doppelte Hochkommata gesetzt. Der Operator "+" kann dazu benutzt werden, zwei Strings aneinander zu hängen, der Ausdruck

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

liefert also das Ergebnis

```
"abcuvw".
```

Zusätzlich kann eine natürliche Zahl n mit einem String s über den Multiplikations-Operator "*" verknüpft werden. Der Ausdruck

```
n * s;
```

liefert als Ergebnis die n-malige Verkettung von s. Beispielsweise ist das Ergebnis von

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

der String "abcabcabc".

2.7 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*. The value is an element of the set

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

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

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

Hence, 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

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

At the start of a game of Texas Hold'em, every player receives two cards. These two cards are known as *preflop* or *hole*. Next, there is a bidding phase where players can bet on their cards. After this bidding phase, the dealer puts three cards 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 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 2.25 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*",

```
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 | #{ k1, k2, k3 } == 3 }
7  trips  := { f : f in flops | [ "3", "d" ] in f || [ "3", "h" ] in f };
8  print(1.0 * #trips / #flops);
```

Figure 2.25: Berechnung von Wahrscheinlichkeiten im Poker

(c) "Q" is short for "*Queen*",

(d) "K" is short for "*King*", and

(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 $[v, s]$. 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 dealt to the 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

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

   However, these inequalities are satisfied if and only if the set $\{\texttt{k1}, \texttt{k2}, \texttt{k3}\}$ 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 7 computes the subset of flops that contain at least one more card with the value 3. As the 3 of clubs and the 3 of spades have already been dealt to the player, the only cards with value 3 that are left are the 3 of diamonds and the 3 of hearts.

8. Finally, the probability for obtaining another card with value 3 in the flop is computed as the ratio of the number of flops containing a card with value 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 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 and their complexities.

2.8 Fallstudie: Berechnung von Pfaden

Wir wollen dieses Kapitel mit einer praktisch relevanten Anwendung der Sprache SETLX abschließen. Dazu betrachten wir das Problem, Pfade in einem *Graphen* zu bestimmen. Abstrakt gesehen beinhaltet ein Graph die Information, zwischen welchen Punkten es direkte Verbindungen gibt. Zur Vereinfachung wollen wir zunächst annehmen, dass die einzelnen Punkte durch Zahlen identifiziert werden. Dann können wir eine direkte Verbindung zwischen zwei Punkten durch ein Paar von Zahlen darstellen. Den Graphen selber stellen wir als eine Menge solcher Paaren dar. Wir betrachten ein Beispiel. Sei R wie folgt definiert:

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

In diesem Graphen haben wir die Punkte 1, 2, 3, 4 und 5. Ein Darstellung dieses Graphen finden Sie in Abbildung 2.26. Beachten Sie, dass die Verbindungen in diesem Graphen *Einbahn-Straßen* sind: Wir haben zwar eine Verbindung von 1 nach 2, aber keine Verbindung von 2 nach 1.

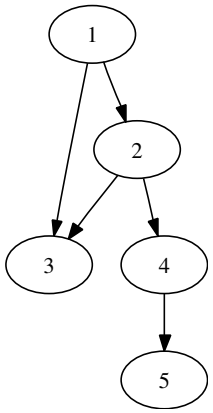

Figure 2.26: Ein einfacher Graph.

In dem Graphen sind nur die unmittelbaren Verbindungen zwischen zwei Punkten verzeichnet. Es gibt aber unter Umständen auch noch andere Verbindungen. Beispielsweise gibt es eine unmittelbare Verbindung von 1 nach 3. Es gibt darüber hinaus noch einen Pfad von 1 nach 3, der über den Punkt 2 geht. Unser Ziel in diesem Abschnitt ist es einen Algorithmus zu entwickeln, der überprüft, ob zwischen zwei Punkten eine Verbindung existiert und gegebenenfalls berechnet. Dazu entwickeln wir zunächst einen Algorithmus, der nur überprüft, ob es eine Verbindung zwischen zwei Punkten gibt und erweitern diesen Algorithmus dann später so, dass diese Verbindung auch berechnet wird.

2.8.1 Berechnung des transitiven Abschlusses einer Relation

Als erstes bemerken wir, dass ein Graph R nichts anderes ist als eine binäre Relation. Um feststellen zu können, ob es zwischen zwei Punkten eine Verbindung gibt, müssen wir den transitiven Abschluss R^+ der Relation R bilden. Wir haben bereits in der Mathematik-Vorlesung gezeigt, dass R^+ wie folgt berechnet werden kann:

$$R^+ = \bigcup_{i=1}^{\infty} R^i = R^1 \cup R^2 \cup R^3 \cup \cdots$$

Auf den ersten Blick betrachtet sieht diese Formel so aus, als ob wir unendlich lange rechnen müssten. Aber versuchen wir einmal, diese Formel anschaulich zu verstehen. Zunächst steht da R. Das sind die Verbindungen, die unmittelbar gegeben sind. Als nächstes steht dort R^2 und das ist $R \circ R$. Es gilt aber

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

In R^2 sind also alle die Pfade enthalten, die aus zwei direkten Verbindungen zusammengesetzt sind. Allgemein lässt sich durch Induktion sehen, dass R^n alle die Pfade enthält, die aus n direkten Verbindungen zusammengesetzt sind. Nun ist die Zahl der Punkte, die wir haben, endlich. Sagen wir mal, dass es k Punkte sind. Dann macht es keinen Sinn solche Pfade zu betrachten, die aus mehr als $k-1$ direkten Verbindungen zusammengesetzt sind, denn wir wollen ja nicht im Kreis herum laufen. Damit kann dann aber die Formel zur Berechnung des transitiven Abschlusses vereinfacht werden:

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

Diese Formel könnten wir tatsächlich so benutzen. Es ist aber noch effizienter, einen Fixpunkt-Algorithmus zu verwenden. Dazu zeigen wir zunächst, dass der transitive Abschluss R^+ die folgende Fixpunkt-Gleichung erfüllt:

$$R^+ = R \cup R \circ R^+. \tag{2.1}$$

Wir erinnern hier daran, dass wir vereinbart haben, dass der Operator $\circ$ stärker bindet als der Operator $\cup$, so dass der Ausdruck $R \cup R \circ R^+$ als $R \cup (R \circ R^+)$ zu lesen ist. Die Fixpunkt-Gleichung 2.1 lässt sich algebraisch beweisen. Es gilt

$$\begin{array}{lll}
 & R \cup R \circ R^+ & \\
= & R \cup R \circ \bigcup_{i=1}^{\infty} R^i & \\
= & R \cup R \circ \left(R^1 \cup R^2 \cup R^3 \cup \cdots\right) & \\
= & R \cup \left(R \circ R^1 \cup R \circ R^2 \cup R \circ R^3 \cup \cdots\right) & \text{Distributiv-Gesetz} \\
= & R \cup \left(R^2 \cup R^3 \cup R^4 \cup \cdots\right) & \text{Potenz-Gesetz} \\
= & R^1 \cup \left(R^2 \cup R^3 \cup R^4 \cup \cdots\right) & \\
= & \bigcup_{i=1}^{\infty} R^i & \\
= & R^+ &
\end{array}$$

Die Gleichung 2.1 kann benutzt werden um den transitiven Abschluss iterativ zu berechnen. Wir definieren eine Folge $(T_n)_{n\in\mathbb{N}}$ durch Induktion folgt:

I.A.	$n = 1$:	$T_1 := R$
I.S.	$n \mapsto n+1$:	$T_{n+1} := R \cup R \circ T_n$.

Die Relationen T_n lassen sich auf die Relation R zurückführen:

1. $T_1 = R$.
2. $T_2 = R \cup R \circ T_1 = R \cup R \circ R = R^1 \cup R^2$.
3. $\begin{array}{rcl} T_3 & = & R \cup R \circ T_2 \\ & = & R \cup R \circ (R^1 \cup R^2) \\ & = & R^1 \cup R^2 \cup R^3. \end{array}$

Allgemein können wir durch vollständige Induktion über $n \in \mathbb{N}$ beweisen, dass

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

gilt. Der Induktions-Anfang folgt unmittelbar aus der Definition von T_1. Um den Induktions-Schritt durchzuführen, betrachten wir

$$\begin{aligned}
T_{n+1} &= R \cup R \circ T_n && \text{gilt nach Definition} \\
&= R \cup R \circ \left(\bigcup_{i=1}^{n} R^i \right) && \text{gilt nach Induktions-Voraussetzung} \\
&= R \cup R^2 \cup \cdots \cup R^{n+1} && \text{Distributiv-Gesetz} \\
&= R^1 \cup \cdots \cup R^{n+1} \\
&= \bigcup_{i=1}^{n+1} R^i && \square
\end{aligned}$$

Die Folge $(T_n)_{n\in\mathbb{N}}$ hat eine weitere nützliche Eigenschaft: Sie ist *monoton steigend*. Allgemein nennen wir eine Folge von Mengen $(X_n)_{n\in\mathbb{N}}$ *monoton steigend*, wenn

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

gilt, wenn also die Mengen X_n mit wachsendem Index n immer größer werden. Die Monotonie der Folge $(T_n)_{n\in\mathbb{N}}$ folgt aus der gerade bewiesenen Eigenschaft $T_n = \bigcup_{i=1}^{n} R^i$, denn es gilt

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

und die letzte Formel ist offenbar wahr. Ist nun die Relation R endlich, so ist natürlich auch R^+ eine endliche Menge. Da die Folge T_n aber in dieser Menge liegt, denn es gilt ja

$$T_n = \bigcup_{i=1}^{n} R^i \subseteq \bigcup_{i=1}^{\infty} R^i = R^+ \quad \text{für alle } n \in \mathbb{N},$$

können die Mengen T_n nicht beliebig groß werden. Aufgrund der Monotonie der Folge $(T_n)_{n\in\mathbb{N}}$ muss es daher einen Index k geben, ab dem die Mengen T_n alle gleich sind:

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

Berücksichtigen wir die Gleichung $T_n = \bigcup_{i=1}^{n} R^i$, so haben wir

$$T_n = \bigcup_{i=1}^{n} R^i = \bigcup_{i=1}^{k} R^i = T_k \quad \text{für alle } n \geq k.$$

Daraus folgt dann aber, dass

$$T_n = \bigcup_{i=1}^{n} R^i = \bigcup_{i=1}^{\infty} R^i = R^+ \quad \text{für alle } n \geq k$$

gilt. Der Algorithmus zur Berechnung von R^+ sieht nun so aus, dass wir die Iteration

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

solange durchführen bis $T_{n+1} = T_n$ gilt, denn dann gilt auch $T_n = R^+$.

Das Programm `transitive-closure.stlx` in Abbildung 2.27 auf Seite 31 zeigt eine Implementierung dieses Gedankens. Lassen wir dieses Programm laufen, so erhalten wir als Ausgabe:

```
R = {[2, 3], [4, 5], [1, 3], [2, 4], [1, 2]}
R+ = {[1, 5], [2, 3], [4, 5], [1, 4], [1, 3], [2, 4], [1, 2], [2, 5]}
```

Der transitive Abschluss R^+ der Relation R lässt sich jetzt anschaulich interpretieren: Er enthält alle Paare $\langle x, y\rangle$, für die es einen *Pfad* von x nach y gibt. Ein Pfad von x nach y ist dabei eine Liste der Form

$$\big[x_1, x_2, \cdots, x_n\big],$$

```
transClosure := procedure(r) {
    t := r;
    while (true) {
        oldT := t;
        t    := r + product(r, t);
        if (t == oldT) {
            return t;
        }
    }
};
product := procedure(r1, r2) {
    return { [x,z] : [x,y] in r1, [y,z] in r2 };
};
r := { [1,2], [2,3], [1,3], [2,4], [4,5] };
print( "r = ", r );
print( "computing transitive closure of r" );
t := transClosure(r);
print( "r+ = ", t );
```

Figure 2.27: Berechnung des transitiven Abschlusses.

für die $x = x_1$ und $y = x_n$ gilt und für die außerdem

$$\langle x_i, x_{i+1} \rangle \in R \quad \text{für alle } i = 1, \cdots, n-1 \text{ gilt.}$$

Die Funktion *product*(r_1, r_2) berechnet das relationale Produkt $r_1 \circ r_2$ nach der Formel

$$r_1 \circ r_2 = \{\langle x, z \rangle \mid \exists y : \langle x, y \rangle \in r_1 \wedge \langle y, z \rangle \in r_2\}.$$

Die Implementierung dieser Prozedur zeigt die allgemeine Form der Mengen-Definition durch Iteratoren in SetlX. Allgemein können wir eine Menge durch den Ausdruck

$$\{\ \textit{expr} : [x_1^{(1)}, \cdots, x_{n(1)}^{(1)}] \texttt{ in } s_1, \cdots, [x_1^{(k)}, \cdots, x_{n(k)}^{(k)}] \texttt{ in } s_k \mid \textit{cond}\ \}$$

definieren. Dabei muss s_i für alle $i = 1, \cdots, k$ eine Menge von Listen der Länge $n(i)$ sein. Bei der Auswertung dieses Ausdrucks werden für die Variablen $x_1^{(i)}, \cdots, x_{n(i)}^{(i)}$ die Werte eingesetzt, die die entsprechenden Komponenten der Listen haben, die in der Menge s_i auftreten. Beispielsweise würde die Auswertung von

```
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 };
```

für `m` die Menge

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

berechnen.

2.8.2 Berechnung der Pfade

Als nächstes wollen wir das Programm zur Berechnung des transitiven Abschlusses so erweitern, dass wir nicht nur feststellen können, dass es einen Pfad zwischen zwei Punkten gibt, sondern dass wir diesen auch berechnen können. Die Idee ist, dass wir statt des relationalen Produkts, das für zwei Relationen definiert ist, ein sogenanntes *Pfad-Produkt*, das auf Mengen von Pfaden definiert ist, berechnen. Vorab führen wir für Pfade, die wir ja durch Listen repräsentieren, drei Begriffe ein.

1. Die Funktion *first*(p) liefert den ersten Punkt der Liste p:

   $$\mathit{first}\big([x_1, \cdots, x_m]\big) = x_1.$$

2. Die Funktion *last*(p) liefert den letzten Punkt der Liste p:

   $$\mathit{last}\big([x_1, \cdots, x_m]\big) = x_m.$$

3. Sind $p = [x_1, \cdots, x_m]$ und $q = [y_1, \cdots, y_n]$ zwei Pfade mit $\mathit{first}(q) = \mathit{last}(p)$, dann definieren wir die Summe von p und q als

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

Sind nun P_1 und P_2 Mengen von Pfaden, so definieren wir das *Pfad-Produkt* von P_1 und P_2 als

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

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

Figure 2.28: Berechnung aller Verbindungen.

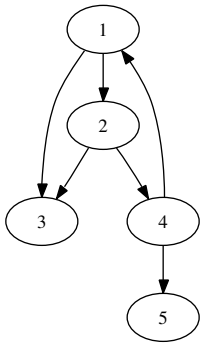

Figure 2.29: Ein zyklischer Graph.

Damit können wir das Programm in Abbildung 2.27 so abändern, dass alle möglichen Verbindungen zwischen zwei Punkten berechnet werden. Abbildung 2.28 zeigt das resultierende Programm `path.stlx`. Leider funktioniert das Programm dann nicht mehr, wenn der Graph Zyklen enthält. Abbildung 2.29 zeigt einen Graphen, der einen Zyklus enthält. In diesem Graphen gibt es unendlich viele Pfade, die von dem Punkt 1 zu dem Punkt 2 führen:

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

Offenbar sind die Pfade unwichtig, die einen Punkt mehrfach enthalten und die daher zyklisch sind. Solche Pfade sollten wir bei der Berechnung des Pfad-Produktes eliminieren.

```
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 2.30: Berechnung aller Verbindungen in zyklischen Graphen

Abbildung 2.30 zeigt einen Ausschnitt des geänderten Programms `path-cyclic.stlx`, das auch für zyklische Graphen funktioniert.

1. In Zeile 2 berücksichtigen wir nur die Pfade $x \oplus y$, die nicht zyklisch sind.
2. In Zeile 5 überprüfen wir, ob die Konkatenation $l_1 \oplus l_2$ zyklisch ist. Die Kombination von l_1 und l_2 ist genau dann zyklisch, wenn die Listen l_1 und l_2 mehr als ein gemeinsames Element enthalten. Die Listen l_1 und l_2 enthalten mindestens ein gemeinsames Element, denn wir verknüpfen diese beiden Listen ja nur dann, wenn das letzte Element der Liste l_1 mit dem ersten Element der Liste l_2 übereinstimmt. Wenn es nun noch einen weiteren Punkt geben würde, der sowohl in l_1 als auch in l_2 auftreten würde, dann wäre der Pfad $l_1 \oplus l_2$ zyklisch.

In den meisten Fällen sind wird gar nicht daran interessiert, alle möglichen Verbindungen zwischen allen Punkten zu berechnen, das wäre nämlich viel zu aufwendig, sondern wir wollen nur zwischen zwei gegebenen Punkten eine Verbindung finden. Abbildung 2.31 zeigt die Implementierung einer Prozedur `findPath`(x, y, r), die überprüft, ob es in dem Graphen r eine Verbindung von x nach y gibt und die diese Verbindung berechnet. Das vollständige Programm finden Sie in der Datei `find-path.stlx`. Wir diskutieren nun die Implementierung der Prozedur `findPath`.

1. In Zeile 2 initialisieren wir p so, dass zunächst nur der Pfad der Länge 0, der mit dem Punkt x startet, in p liegt.
2. In Zeile 5 versuchen wir, die Pfade, die wir bereits berechnet haben, mit Hilfe der Relation r zu verlängern. Dabei erhalten wir gegebenenfalls neue Pfade.
3. In Zeile 6 selektieren wir aus allen diesen Pfaden die Pfade aus, die zum Ziel y führen.
4. Wenn wir dann in Zeile 7 feststellen, dass wir einen solchen Pfad berechnet haben, geben wir einen dieser Pfade in Zeile 8 zurück.
5. Falls es nicht gelingt einen solchen Pfad zu berechnen und wir keine neuen Pfade mehr finden können, verlassen wir die Prozedur in Zeile 11 mit dem Befehl `return`. Da wir bei diesem `return`-Befehl keinen Wert zurückgeben, ist der Rückgabewert der Prozedur in diesem Fall automatisch Ω.

```
findPath := procedure(x, y, r) {
    p := { [x] };
    while (true) {
        oldP  := p;
        p     := p + pathProduct(p, r);
        found := { l : l in p | l[-1] == y };
        if (found != {}) {
            return arb(found);
        }
        if (p == oldP) {
            return;
        }
    }
};
```

Figure 2.31: Berechnung aller Verbindungen zwischen zwei Punkten

2.8.3 The Wolf, the Goat, and the Cabbage

Next, we present an application of the theory developed so far. We show how to solve a problem from agricultural economy. 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, he has to cross a river. The boat that he 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 farmer leaves the goat alone with the cabbage, the goat will eat the cabbage. Is it possible for the agricultural economist to develop a schedule that allows him 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 nodes in a graph and this graph will be represented as a binary relation. To this end we define the set

$$\texttt{all} := \{\texttt{"farmer"}, \texttt{"wolf"}, \texttt{"goat"}, \texttt{"cabbage"}\}.$$

Every node will be represented as a subset `s` of the set `all`. The idea is that `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 in 2 ** all | !problem(s) && !problem(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 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" in s && "cabbage" in s || "wolf" in s && "goat" in 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 descibes 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 is 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 2.32 on page 36 shows the program wolf-goat-cabbage.stlx that combines the statements shown so far. The solution computed by this program is shown in Figure 2.33.

2.9 Terme und Matching

Neben den bisher vorgestellten Datenstrukturen gibt es noch eine weitere wichtige Datenstruktur, die sogenannten *Terme*, die insbesondere nützlich ist, wenn wir Programme schreiben wollen, die Formeln manipulieren. Wollen wir beispielsweise ein Programm schreiben, dass als Eingabe einen String wie

`“x * sin(x)”`

einliest, diesen String als eine Funktion in der Variablen “x” interpretiert und dann die Ableitung dieser Funktion nach der Variablen “x” berechnet, so sprechen wir von *symbolischer Programmierung*. Wollen wir einen Ausdruck wie `“x * sin(x)”` darstellen, so eignen sich *Terme* am besten dazu. Im nächsten Unterabschnitt werden wir zunächst

```
problem := procedure(s) {
    return !("farmer" in s) &&
           ("goat" in s && "cabbage" in s || "wolf" in s && "goat" in s);
};

all := { "farmer", "wolf", "goat", "cabbage" };
p   := { s : s in 2 ** all | !problem(s) && !problem(all - s) };
r1  := { [s, s - b]: s in p, b in 2 ** s
                   | s - b in p && "farmer" in b && #b <= 2
       };
r2  := { [y, x] : [x, y] in r1 };
r   := r1 + r2;

start := all;
goal  := {};

path  := findPath(start, goal, r);
```

Figure 2.32: Solving the wolf-goat-cabbage problem.

```
{"cabbage", "farmer", "goat", "wolf"}                                  {}
                          >>>> {"farmer", "goat"} >>>>
{"cabbage", "wolf"}                                  {"farmer", "goat"}
                          <<<< {"farmer"} <<<<
{"cabbage", "farmer", "wolf"}                                      {"goat"}
                          >>>> {"farmer", "wolf"} >>>>
{"cabbage"}                                      {"farmer", "goat", "wolf"}
                          <<<< {"farmer", "goat"} <<<<
{"cabbage", "farmer", "goat"}                                      {"wolf"}
                          >>>> {"cabbage", "farmer"} >>>>
{"goat"}                                  {"cabbage", "farmer", "wolf"}
                          <<<< {"farmer"} <<<<
{"farmer", "goat"}                                  {"cabbage", "wolf"}
                          >>>> {"farmer", "goat"} >>>>
{}                                  {"cabbage", "farmer", "goat", "wolf"}
```

Figure 2.33: A schedule for the agricultural economist.

Terme zusammen mit den in SetlX vordefinierten Funktionen vorstellen, die zur Verarbeitung von Termen benutzt werden können. Anschließend stellen wir das sogenannte *Matching* vor, mit dessen Hilfe sich Terme besonders leicht manipulieren lassen.

2.9.1 Konstruktion und Manipulation von Termen

Terme werden mit Hilfe sogenannter *Funktions-Zeichen* gebildet. Es ist wichtig, dass Sie Funktions-Zeichen nicht mit Funktionen oder Variablen verwechseln. In SetlX beginnen Funktionen-Zeichen im Gegensatz zu einem Variablen-Namen daher mit einem großen Buchstaben. Auf den Großbuchstaben können dann beliebig viele Buchstaben, Ziffern und der Unterstrich "_" folgen. Zusätzlich gibt es noch Funktionszeichen, die mit dem Zeichen "^" beginnen. Solche Funktions-Zeichen werden intern von SetlX verwendet um Operator-Symbole wie "+" oder "*" darzustellen. Die folgenden Strings können beispielsweise als Funktions-Zeichen vewendet werden:

`F`, `FabcXYZ`, `^sum`, `Hugo_`.

Damit sind wir nun in der Lage, Terme zu definieren. Ist F ein Funktions-Zeichen und sind t_1, t_2, $\cdots$, beliebige SetlX-Werte, so ist der Ausdruck

$$F(t_1, t_2, \cdots, t_n)$$

ein Term. Beachten Sie, dass Terme ganz ähnlich aussehen wie die Aufrufe von Funktionen. Terme und Aufrufe von Funktionen unterscheiden sich nur dadurch, dass bei einem Term links vor der ersten öffnenden Klammer ein Funktions-Zeichen steht, während bei einem Funktions-Aufruf dort statt dessen eine Variable steht, der eine Funktions-Definition zugewiesen worden ist.

Beispiele:

1. `Adresse("Rotebühlplatz 41", 70178, "Stuttgart")`
   ist ein Term, der eine Adresse repräsentiert.
2. `Product(Variable("x"), Sin(Variable("x")))`
   ist ein Term, der einen arithmetischen Ausdruck repräsentiert, den Sie mathematisch als $x \cdot \sin(x)$ schreiben würden. ◇

An dieser Stelle fragen Sie sich vielleicht, wie Terme ausgewertet werden. Die Antwort ist: Gar nicht! Terme werden nur dazu benutzt, Daten darzustellen. Terme sind also bereits Werte genauso wie auch Zahlen, Strings, Mengen oder Listen als Werte aufgefasst werden. Genausowenig wie Sie die Zahl 42 auswerten müssen, müssen Sie einen Term auswerten.

Nehmen wir einmal an, dass es in SetlX keine Listen geben würde. Dann könnten wir Listen als Terme darstellen. Zunächst würden wir ein Funktions-Zeichen benötigen, mit dem wir die leere Liste darstellen könnten. Wir wählen dazu das Funktions-Zeichen `Nil`. Damit haben wir dann also die Entsprechung

$$\texttt{Nil}() \mathrel{\widehat{=}} [].$$

Beachten Sie hier, dass die Klammern hinter dem Funktions-Zeichen `Nil` nicht weggelassen werden dürfen! Um nun eine Liste darzustellen, deren erstes Element x ist und deren restliche Elemente durch die Restliste r gegeben sind, verwenden wir das Funktions-Zeichen `Cons`. Dann haben wir die Entsprechung

$$\texttt{Cons(x, r)} \mathrel{\widehat{=}} [x] + r.$$

Konkret können wir nun die Liste `[1,2,3]` durch den Term

```
Cons(1, Cons(2, Cons(3, Nil())))
```

darstellen. In der Sprache *Prolog*, werden Listen intern in ähnlicher Form als Terme dargestellt.

Es gibt zwei vordefinierte Funktionen in SetlX, mit denen wir auf die Komponenten eines Terms zugreifen können und es gibt eine weitere Funktion, mit deren Hilfe wir Terme konstruieren können.

1. Die Funktion `fct` berechnet das Funktions-Zeichen eines Terms. Falls t ein Term der Form $F(s_1, \cdots, s_n)$ ist, so ist das Ergebnis des Funktions-Aufrufs

   $$\texttt{fct}(F(s_1, \cdots, s_n))$$

   das Funktions-Zeichen F dieses Terms. Beispielsweise liefert der Ausdruck

   ```
   fct(Cons(1, Cons(2, Cons(3, Nil()))))
   ```

   als Ergebnis das Funktions-Zeichen `"Cons"`.
2. Die Funktion `args` berechnet die Argumente eines Terms. Falls t ein Term der Form $F(s_1, \cdots, s_n)$ ist, dann liefert der Ausdruck

   $$\texttt{args}(F(s_1, \cdots, s_n))$$

   als Ergebnis die Liste $[s_1, \cdots, s_n]$ der Argumente des Terms t . Beispielsweise liefert der Ausdruck

```
args(Cons(1, Cons(2, Cons(3, Nil()))))
```

das Ergebnis

```
[1, Cons(2, Cons(3, Nil()))].
```

3. Ist ein Funktions-Zeichen f und eine Liste l von Argumenten gegeben, so erzeugt die Funktion `makeTerm` durch den Aufruf

   `makeTerm`(f, l)

   einen Term t mit dem Funktions-Zeichen f und der Argument-Liste l, für t gilt also

   $\texttt{fct}(t) = f \quad$ und $\quad \texttt{args}(t) = l.$

   Beispielsweise liefert der Aufruf

   ```
   makeTerm("Cons", [ 1, Nil() ])
   ```

   als Ergebnis den Term

   ```
   Cons(1,Nil()).
   ```

   Diesen Term hätten wir natürlich auch unmittelbar hinschreiben können.

```
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  };
```

Figure 2.34: Einfügen eines Elements am Ende einer Liste.

In Abbildung 2.34 auf Seite 38 sehen Sie die Implementierung einer Funktion append, deren Aufgabe es ist, ein Element x am Ende einer Liste l einzufügen, wobei vorausgesetzt ist, dass die Liste l als Term mit Hilfe der Funktions-Zeichen "`Cons`" und "`Nil`" dargestellt wird.

1. Zunächst wird in Zeile 2 überprüft, ob die Liste l leer ist. Die Liste l ist genau dann leer, wenn $l =$ `Nil()` gilt. Daher können wir einfach das Funktions-Zeichen des Terms l testen um herauszufinden, ob l die leere Liste repräsentiert.

2. Falls l nicht leer ist, muss l die Form

   $l =$ `Cons(head, tail)`

   haben. Dann ist *head* das erste Element der Liste l und *tail* bezeichnet die Liste der restlichen Elemente. In diesem Fall müssen wir x rekursiv in die Liste *tail* einfügen. Als Ergebnis wird in Zeile 6 dann eine neue Liste erzeugt, deren erstes Element *head* ist, während die Liste der restlichen Elemente durch den rekursiven Aufruf von *append* berechnet wird.

In manchen Fällen ist es sehr unbequem, dass Funktions-Zeichen immer mit einem großen Buchstaben beginnen müssen. Deswegen gibt es in SetlX einen Escape-Mechanismus, der es erlaubt, auch Funktionszeichen zu verwenden, die mit einem kleinen Buchstaben beginnen: Falls wir einem Funktionszeichen den Operator "@" voranstellen, dann darf das Funktionszeichen auch mit einem kleinen Buchstaben beginnen. Wollen wir beispielsweise Terme benutzen um algebraische Ausdrücke darzustellen, die trigonometrische Funktionen enthalten, so können wir einen Ausdruck der Form $\sin(x)$ durch den Term

```
@sin("x")
```

darstellen.

2.9.2 Matching

Der Umgang mit Termen wäre sehr mühsam, wenn wir die Terme jedesmal mit Hilfe der Funktionen `fct` und `args` auseinander nehmen müssten. Abbildung 2.35 zeigt eine weitere Implementierung der Funktion append, bei der wir die Kontroll-Struktur `match` an Stelle der Funktionen "`fct`" and "`args`" verwendet haben. In Zeile 3 wird überprüft, ob die Liste l leer ist. Die wahre Stärke des Matchings sehen wir allerdings ist in Zeile 4, denn dort wird nicht nur überprüft, ob die Liste l die Form

`Cons(head,tail)`

hat, sondern gleichzeitig werden die Variablen *head* and *tail* so gesetzt, dass anschließend die Gleichung

$l =$ `Cons(head,tail)`

erfüllt ist.

```
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 2.35: Implementierung von append mit Hilfe von *Matching*.

Im Allgemeinen ist ein `match`-Block so ähnlich aufgebaut wie ein `switch`-Block und hat die in Abbildung 2.36 gezeigte Struktur. Hier bezeichnet e einen Ausdruck, dessen Auswertung einen Term ergibt. Die Ausdrücke $t_1, \cdots, t_n$ sind sogenannte *Muster*, die freie Variablen enthalten. Bei der Auswertung eines `match`-Blocks versucht SetlX die in dem Muster t_i auftretenden Variablen so zu setzen, dass das Muster zu dem Ergebnis der Auswertung von e gleich ist. Gelingt dies, so wird die mit $body_i$ bezeichnete Gruppe von Befehlen ausgeführt. Andernfalls versucht SetlX das nächste Muster t_{i+1} mit e zur Deckung zu bringen. Falls keines der Muster $t_1, \cdots, t_n$ mit e zur Deckung zu bringen ist, wird ersatzweise $body_{n+1}$ ausgeführt.

```
1    match (e) {
2        case t_1 : body_1
3        ⋮
4        case t_n : body_n
5        default: body_{n+1}
6    }
```

Figure 2.36: Struktur eines `match`-Blocks

Wir zeigen zum Abschluss dieses Abschnitts ein komplexeres Beispiel. Die in Abbildung 2.37 auf Seite 40 gezeigte Funktion `diff` wird mit zwei Argumenten aufgerufen:

1. Das erste Argument t ist ein Term, der einen arithmetischen Ausdruck repräsentiert.
2. Das zweite Argument x ist ein String, der als Variable interpretiert wird.

Die Aufgabe der Funktion `diff` besteht darin, den durch t gegebenen Ausdruck nach der in x angegebenen Variablen zu differenzieren. Wollen wir beispielsweise die Funktion

$x \mapsto x^x$

nach x ableiten, so können wir die Funktion `diff` wie folgt aufrufen.

```
1  diff := procedure(t, x) {
2      match (t) {
3          case t1 + t2 :
4              return diff(t1, x) + diff(t2, x);
5          case t1 - t2 :
6              return diff(t1, x) - diff(t2, x);
7          case t1 * t2 :
8              return diff(t1, x) * t2 + t1 * diff(t2, x);
9          case t1 / t2 :
10             return ( diff(t1, x) * t2 - t1 * diff(t2, x) ) / t2 * t2;
11         case f ** g :
12             return diff( @exp(g * @ln(f)), x);
13         case ln(a) :
14             return diff(a, x) / a;
15         case exp(a) :
16             return diff(a, x) * @exp(a);
17         case ^variable(x) : // x is defined above as second argument
18             return 1;
19         case ^variable(y) : // y not yet defined, matches any other variable
20             return 0;
21         case n | isNumber(n):
22             return 0;
23     }
24 };
```

Figure 2.37: A function to perform symbolic differentiation.

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

Hier wandelt die Funktion `parse` den String "`x ** x`" in einen Term um. Die genaue Struktur dieses Terms diskutieren wir weiter unten. Wir betrachten zunächst den `match`-Befehl in Abbildung 2.37. In Zeile 3 hat der zu differenzierende Ausdruck die Form `t1 + t2`. Um einen solchen Ausdruck nach einer Variablen x zu differenzieren, müssen wir sowohl `t1` als auch `t2` nach x differenzieren. Die dabei erhaltenen Ergebnisse sind dann zu addieren. Etwas interessanter ist Zeile 8, welche die Produkt-Regel der Differenzial-Rechnung umsetzt. Die Produkt-Regel lautet:

$$\frac{d}{dx}(t_1 \cdot t_2) = \frac{d\,t_1}{dx} \cdot t_2 + t_1 \cdot \frac{d\,t_2}{dx}.$$

Bemerken Sie, dass in Zeile 7 das Muster

```
t1 * t2
```

zum einen dazu dient, zu erkennen, dass der zu differenzierende Ausdruck ein Produkt ist, zum anderen aber auch die beiden Faktoren des Produkts extrahiert und an die Variablen t_1 und t_2 bindet. In den Zeilen 12 und 16 haben wir den Funktions-Zeichen "`exp`" und "`ln`" den Operator "`@`" vorangestellen müssen, denn sonst würden die Strings "`exp`" und "`ln`" nicht als Funktions-Zeichen sondern als Variablen aufgefasst werden.

Die Regel zur Berechnung der Ableitung eines Ausdrucks der Form f^g beruht auf der Gleichung

$$f^g = \exp\bigl(\ln\bigl(f^g\bigr)\bigr) = \exp\bigl(g \cdot \ln(f)\bigr),$$

die in Zeile 12 umgesetzt wird.

Um einen Ausdruck der Form $\ln(f)$ abzuleiten, müssen wir die Kettenregel anwenden. Da $\frac{d}{dx}\ln(x) = \frac{1}{x}$ ist, haben wir insgesamt

$$\frac{d}{dx}\ln(f) = \frac{1}{f} \cdot \frac{d\,f}{dx}.$$

Diese Gleichung wurde in Zeile 14 verwendet. In analoger Weise wird dann in Zeile 16 mit Hilfe der Kettenregel ein Ausdruck der Form $\exp(f)$ abgeleitet.

Um das Beispiel in Abbildung 2.37 besser zu verstehen müssen wir wissen, wie die Funktion `parse` einen String in einen Term umwandelt. Die Funktion `parse` muss sowohl Operator-Symbole als auch Variablen verarbeiten. Eine Variable der Form `"x"` wird in den Term

```
^variable("x")
```

umgewandelt. Dies erklärt die Zeilen 19 und 21 von Abbildung 2.37.

Wir können die interne Darstellung eines Terms mit Hilfe der Funktion "`canonical`" ausgeben. Beispielsweise liefert der Ausdruck

```
canonical(parse("x ** x"))
```

das Ergebnis

```
^power(^variable("x"), ^variable("x")).
```

Dies zeigt, dass der Exponentiations-Operator "`**`" in SetlX intern durch das Funktions-Zeichen "`^power`" dargestellt wird. Die interne Darstellung des Operators "`+`" ist "`^sum`", "`-`" wird durch das Funktions-Zeichen "`^difference`" dargestellt, "`*`" wird durch das Funktions-Zeichen "`^product`" dargestellt und der Operator "`/`" wird durch das Funktions-Zeichen "`^quotient`" dargestellt.

Terme sind in dem folgenden Sinne *viral*: Falls ein Argument eines der Operatoren "`+`", "`-`", "`*`", "`/`", "`\`" und "`%`" ein Term ist, so erzeugt der Operator als Ergebnis automatisch einen Term. Beispielsweise liefert der Ausdruck

```
parse("x") + 2
```

den Term

```
^sum(^variable("x"), 2).
```

Zeile 21 zeigt, dass an ein Muster in einem `case` eine Bedingung angeschlossen werden kann: Das Muster

```
case n:
```

passt zunächst auf jeden Term. Allerdings wollen wir in Zeile 21 nur Zahlen matchen. Daher haben wir an dieses Muster mit Hilfe des Operators "`|`" noch die Bedingung `isNumber(n)` angehängt, mit der wir sicherstellen, dass n tatsächlich eine Zahl ist.

2.9.3 Ausblick

Wir konnten in diesem einführenden Kapitel nur einen Teil der Sprache SetlX behandeln. Einige weitere Features der Sprache SetlX werden wir noch in den folgenden Kapiteln diskutieren. Zusätzlich finden Sie weitere Informationen in dem Tutorial, das im Netz unter der Adresse

https://github.com/karlstroetmann/setlX/blob/master/tutorial.pdf

abgelegt ist.

Bemerkung: Die meisten der in diesem Abschnitt vorgestellten Algorithmen sind nicht effizient. Sie dienen nur dazu, die Begriffsbildungen aus der Mengenlehre konkret werden zu lassen. Die Entwicklung effizienter Algorithmen ist Gegenstand des zweiten Semesters.

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` zurück gegeben.

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 k finden, so dass zwischen k^2 und $(k+1)^2$ 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 dass uns die Frage, ob die Berechnung von $f(a)$ terminiert, beantworten kann. Die Idee wäre im wesentlichen, dass gilt:

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

Falls der Aufruf $f(a)$ nicht terminiert, sollte statt dessen `stops`$(f, a) = 0$ gelten. Wenn wir eine solche Funktion `stops` hätten, dann könnten wir

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

```
findCounterExample := procedure(n) {
    legendre := procedure(n) {
        k := n * n + 1;
        while (k < (n + 1) ** 2) {
            if (isPrime(k)) {
                print("$n$**2 < $k$ < $n+1$**2");
                return true;
            }
            k += 1;
        }
        return false;
    };
    while (true) {
       if (legendre(n)) {
           n := n + 1;
       } else {
           print("Legendre was wrong, no prime between $n**2$ and $(n+1)**2$!");
           return;
       }
    }
};
```

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

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

$$\texttt{stops(findCounterExample, 1)} = 1$$

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)` statt dessen 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 stimmt.

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` zumindest sehr schwierig wird.

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

$$\texttt{procedure(x) \{ } \cdots \texttt{ \}}$$

hat und sich als SetlX-Funktion parsen lässt. Die Menge der Test-Funktionen bezeichnen wir mit TF. □

Beispiele:

1. $s_1 =$ "`procedure(x) { return 0; }`"
   s_1 ist eine (sehr einfache) Test-Funktion.

2. $s_2 =$ "`procedure(x) { while (true) { x := x + 1; } }`"

   s_2 ist ebenfalls eine Test-Funktion. Offenbar liefert diese Test-Funktion nie ein Ergebnis, aber für die Frage, ob s_2 eine Test-Funktion ist oder nicht, ist dies irrelevant.

3. $s_3 =$ "`procedure(x) { return ++x; }`"

   s_3 ist keine Test-Funktion, denn da SetlX den Präfix-Operator "++" nicht unterstützt, lässt sich der String s_3 nicht fehlerfrei parsen.

4. $s_4 =$ "`procedure(x, y) { return x + y; }`"

   s_4 ist keine Test-Funktion, denn ein String ist nur dann eine Test-Funktion, wenn die Funktion mit genau einen Parameter aufgerufen wird.

Um das Halte-Problem übersichtlicher formulieren zu können, führen wir noch drei zusätzliche Notationen ein.

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

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

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

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

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

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

und sagen, dass der Aufruf $t(a_1, \cdots, a_k)$ *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. `procedure(x) { return 0; }("emil")` $\rightsquigarrow 0$
2. `procedure(x) { return 0; }("emil")` $\downarrow$
3. `procedure(x) { while (true) { x := x + 1; } }("hugo")` $\uparrow$

Das *Halte-Problem* für SetlX-Funktionen ist die Frage, ob es eine `SetlX`-Funktion

`stops := procedure`$(t,\ a)\ \{\cdots\}$

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

1. $t \notin TF \quad \Leftrightarrow \quad$ `stops`$(t, a) \rightsquigarrow 2$.

   Der Aufruf `stops(`t`,` a`)` liefert genau dann den Wert 2 zurück, wenn t keine Test-Funktion ist.

2. $t \in TF \wedge t(a)\downarrow \quad \Leftrightarrow \quad$ `stops`$(t, a) \rightsquigarrow 1$.

   Der Aufruf `stops(`t`,` a`)` liefert genau dann den Wert 1 zurück, wenn t eine Test-Funktion ist und der Aufruf $t(a)$ terminiert.

3. $t \in TF \wedge t(a)\uparrow \quad \Leftrightarrow \quad$ `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:

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

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:

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

Diese beiden Fälle analysieren wir nun im Detail:

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

   Nach der Spezifikation von `stops` bedeutet dies

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

   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. $\lightning$

   Damit ist der erste Fall ausgeschlossen.

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

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

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

   Schauen wir nun, was wirklich beim Aufruf `turing(turing)` passiert: In Zeile 2 erhält 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 ausgeführt. Dieser Block besteht aber nur aus einer Endlos-Schleife, aus der wir nie wieder zurück kommen. Das steht im Widerspruch zu dem, was die Funktion `stops` behauptet hat. $\lightning$

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 löst, falsch sein, denn diese Annahme ist die Ursache für die Widersprüche, die wir erhalten haben. Insgesamt haben wir daher gezeigt, dass es keine SetlX-Funktion geben kann, die das Halte-Problem löst. □

Bemerkung: Der Nachweis, dass das Halte-Problem unlösbar ist, wurde 1936 von Alan Turing (1912 – 1954) [Tur36] erbracht. Turing hat das Problem damals natürlich nicht für die Sprache SetlX gelöst, sondern für 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 können wir uns fragen, ob es vielleicht eine andere Programmier-Sprache gibt, in der wir das Halte-Problem dann vielleicht doch lösen könnten. Wenn es in dieses Programmier-Sprache Prozeduren, `if`-Verzweigungen und `while`-Schleifen gibt, und wenn wir dort Programm-Texte als Argumente von Funktionen übergeben können, dann ist leicht zu sehen, dass der obige Beweis der Unlösbarkeit des Halte-Problems sich durch geeignete syntaktische Modifikationen auch auf die andere Programmier-Sprache übertragen lässt.

3.2 Unlösbarkeit des Äquivalenz-Problems

Es gibt noch eine ganze Reihe anderer Funktionen, die nicht berechenbar sind. In der Regel werden wir den Nachweis, dass eine bestimmt Funktion nicht berechenbar ist, indirekt führen und annehmen, dass die gesuchte Funktion doch berechenbar ist. Unter dieser Annahme konstruieren wir dann eine Funktion, die das Halte-Problem löst, was im Widerspruch zu der Unlösbarkeit 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 benötigen wir aber noch eine Definition.

Definition 4 ($\simeq$) Es seien t_1 und t_2 zwei SetlX-Funktionen und a_1, $\cdots$, a_k seien Argumente, mit denen wir diese Funktionen füttern können. Wir definieren

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

g.d.w. einer der beiden folgen Fälle auftritt:

1. $t_1(a_1, \cdots, a_k)\uparrow \quad \wedge \quad t_2(a_1, \cdots, a_k)\uparrow$,
   beide Funktionen divergieren also für die gegebenen Argumente.
2. $\exists r : \Big(t_1(a_1, \cdots, a_k) \rightsquigarrow r \;\wedge\; t_2(a_1, \cdots, a_k) \rightsquigarrow r\Big)$,
   die Funktionen liefern also für die gegebenen Argumente das gleiche Ergebnis.

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

Wir kommen jetzt zum *Äquivalenz-Problem*. Die Funktion `equal`, die die Form

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

hat, möge folgender Spezifikation genügen:

1. $p_1 \notin TF \;\vee\; p_2 \notin TF \quad \Leftrightarrow \quad \texttt{equal}(p_1, p_2, a) \rightsquigarrow 2.$
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:

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

3. Ansonsten gilt

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

Wir sagen, dass eine Funktion, die der eben angegebenen Spezifikation genügt, das *Äquivalenz-Problem* löst.

Theorem 5 (Rice, 1953) *Das Äquivalenz-Problem ist unlösbar.*

Beweis: Wir führen den Beweis indirekt und nehmen an, dass es doch eine Implementierung der Funktion `equal` gibt, die das Äquivalenz-Problem löst. Wir betrachten die in Abbildung 3.3 angegeben 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 für 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 zurück geben. Damit liefert die Funktion `stops` aber für eine Test-Funktion p und ein Argument a genau dann 1, wenn der Aufruf $p(a)$ terminiert und würde folglich das Halte-Problem lösen. Das kann nicht sein, also kann es keine Funktion `equal` geben, die das Äquivalenz-Problem löst. □

Die Unlösbarkeit des Äquivalenz-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* $\cdots$*, 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* $\cdots$*, 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,* $\cdots$ *dann*", und "*genau dann, wenn*" die folgenden Abkürzungen ein:

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

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

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

schreiben. Das *Beweis-Prinzip*, das wir oben verwendet haben, ist dabei wie folgt: Aus den Aussagen

1. SonneScheint
2. EsRegnet
3. $\texttt{SonneScheint} \wedge \texttt{EsRegnet} \rightarrow \texttt{Regenbogen}$

folgt logisch die Aussage

Regenbogen.

Um Beweis-Prinzipien übersichtlicher angeben zu können, führen wir die folgende Notation ein:

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

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 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. ◇

Lösung: Es wird die folgende Schluss-Regel verwendet:

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

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

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. Eine 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 $\vdash$ steht für die Menge aller Schluss-Regeln und symbolisiert damit den *Herleitungs-Begriff*, den wir auch als *Kalkül* bezeichnen. Wir werden in einem späteren Abschnitt den Kalkül formal definieren. Parallel zu dem Herleitungsbegriff, der seiner Natur nach syntaktisch ist, gibt es auch einen *semantischen*, also inhaltlichen *Folgerungs-Begriff*. Wir schreiben

$$M \models r,$$

wenn die Aussage r logisch aus den Aussagen M folgt. 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 Herleitungsbegriff zu finden, der die folgenden beiden Bedingungen erfüllt:

1. Der Herleitungsbegriff sollte **korrekt** sein, es sollte also nicht möglich sein, Unsinn zu beweisen. Es sollte also gelten

   Aus $M \vdash r$ 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 Herleitungsbegriff sollte **vollständig** sein, d.h. wenn eine Aussage r aus einer Menge von anderen Aussagen M logisch folgt, dann sollte sie auch aus M beweisbar sein:

   Aus $M \models r$ 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 bis zu einer Milliarden 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.

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

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 sheduling*).

   International tätige Fluggesellschaften müssen bei der Einteilung ihrer Crews einerseits gesetzlich vorgesehene Ruhezeiten einhalten, wollen aber ihr Personal möglichst effizient einsetzen. Das führt zu Problemen, 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 lösen. 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. Aussagenlogische Formeln sind dann Wörter, die aus dem Alphabet

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

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

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

   Hier steht $\top$ für die Formel, die immer wahr ist, während $\bot$ für die Formel steht, die immer falsch ist. Die Formel $\top$ trägt auch den Namen *Verum*, für $\bot$ 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 $\neg f \in \mathcal{F}$.

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

[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.

$(f_1 \vee f_2) \in \mathcal{F}$ (*gelesen*: f_1 oder f_2),
$(f_1 \wedge f_2) \in \mathcal{F}$ (*gelesen*: f_1 und f_2),
$(f_1 \rightarrow f_2) \in \mathcal{F}$ (*gelesen*: wenn f_1, dann f_2),
$(f_1 \leftrightarrow f_2) \in \mathcal{F}$ (*gelesen*: f_1 genau dann, wenn 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*.

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

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

   Beachten Sie, dass die Junktoren $\wedge$ und $\vee$ gleich stark binden. Dies ist anders als in der Sprache SetlX, denn dort bindet der Operator "&&" stärker als der Operator "||".

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 $\cdots$, 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 $\cdots$, dann*" als "$\rightarrow$" geschrieben.
2. Bei den Beweisen, die wir über aussagenlogische Formeln führen, schreiben wir für diesen Junktor statt dessen "$\Rightarrow$".

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 statt dessen "$\Leftrightarrow$". ◇

4.3.2 Semantik der aussagenlogischen Formeln

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

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

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

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

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

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

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

Eine aussagenlogische Interpretation $\mathcal{I}$ interpretiert die Aussage-Variablen. Um nicht nur Variablen sondern auch aussagenlogische 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 $\ominus$, $\oslash$, $\odot$, $\oplus$ und $\ominus$ mit deren Hilfe wir die aussagenlogischen Junktoren interpretieren können:

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

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

$$\ominus = \{\langle \texttt{true}, \texttt{false}\rangle, \langle \texttt{false}, \texttt{true}\rangle\}.$$

Wir könnten auch die Funktionen $\oslash$, $\odot$, $\oplus$ und $\ominus$ als Relationen definieren, es ist aber anschaulicher, wenn wir die Werte dieser Funktionen durch eine Tabelle festlegen. Diese Tabelle ist oben auf Seite 53 abgebildet.

p	q	$\ominus(p)$	$\odot(p,q)$	$\oslash(p,q)$	$\oplus(p,q)$	$\ominus(p,q)$
true	true	false	true	true	true	true
true	false	false	true	false	false	false
false	true	true	true	false	true	false
false	false	true	false	false	true	true

Table 4.1: Interpretation 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 $\widehat{\mathcal{I}}(f)$ bezeichnen. Wir setzen:

1. $\widehat{\mathcal{I}}(\bot) := \texttt{false}$.
2. $\widehat{\mathcal{I}}(\top) := \texttt{true}$.
3. $\widehat{\mathcal{I}}(p) := \mathcal{I}(p)$ für alle $p \in \mathcal{P}$.
4. $\widehat{\mathcal{I}}(\neg f) := \ominus\big(\widehat{\mathcal{I}}(f)\big)$ für alle $f \in \mathcal{F}$.

5. $\widehat{\mathcal{I}}(f \wedge g) := \circledwedge\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ für alle $f, g \in \mathcal{F}$.
6. $\widehat{\mathcal{I}}(f \vee g) := \circledvee\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ für alle $f, g \in \mathcal{F}$.
7. $\widehat{\mathcal{I}}(f \to g) := \ominus\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ für alle $f, g \in \mathcal{F}$.
8. $\widehat{\mathcal{I}}(f \leftrightarrow g) := \circleddash\big(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g)\big)$ für alle $f, g \in \mathcal{F}$.

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

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

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

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

$$\begin{array}{lcl}
\mathcal{I}\Big((p \to q) \to (\neg p \to q) \to q\Big) & = & \ominus\Big(\mathcal{I}\big((p \to q)\big), \mathcal{I}\big((\neg p \to q) \to q\big)\Big) \\
& = & \ominus\Big(\ominus\big(\mathcal{I}(p), \mathcal{I}(q)\big), \mathcal{I}\big((\neg p \to q) \to q\big)\Big) \\
& = & \ominus\Big(\ominus\big(\texttt{true}, \texttt{false}\big), \mathcal{I}\big((\neg p \to q) \to q\big)\Big) \\
& = & \ominus\Big(\texttt{false}, \mathcal{I}\big((\neg p \to q) \to q\big)\Big) \\
& = & \texttt{true}
\end{array}$$

Beachten Sie, dass wir bei der Berechnung gerade soviele Teile der Formel ausgewertet haben, wie notwendig waren um den Wert der Formel zu bestimmen. Trotzdem ist die eben durchgeführte Rechnung für die Praxis zu umständlich. Stattdessen wird der Wert einer Formel direkt mit Hilfe der Tabelle 4.1 auf Seite 53 berechnet. Wir zeigen exemplarisch, wie wir den Wahrheits-Wert der Formel

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

für beliebige Belegungen $\mathcal{I}$ über diese Tabelle berechnen können. Um nun die Wahrheitswerte dieser Formel unter einer gegebenen Belegung der Aussage-Variablen bestimmen zu können, bauen wir eine Tabelle auf, die für jede in der Formel auftretende Teilformel eine Spalte enthält. Tabelle 4.2 auf Seite 54 zeigt die entstehende Tabelle.

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

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

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

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

Damit erhalten wir folgende Rechnung:

1. $\mathcal{I}(\neg p) = \ominus(\mathcal{I}(p)) = \ominus(\texttt{true}) = \texttt{false}$

2. $\mathcal{I}(p \to q) = \ominus\,(\mathcal{I}(p), \mathcal{I}(q)) = \ominus\,(\texttt{true}, \texttt{false}) = \texttt{false}$
3. $\mathcal{I}(\neg p \to q) = \ominus\big(\mathcal{I}(\neg p), \mathcal{I}(q)\big) = \ominus(\texttt{false}, \texttt{false}) = \texttt{true}$
4. $\mathcal{I}\big((\neg p \to q) \to q\big) = \ominus\big(\mathcal{I}(\neg p \to q), \mathcal{I}(q)\big) = \ominus(\texttt{true}, \texttt{false}) = \texttt{false}$
5. $\mathcal{I}\big((p \to q) \to (\neg p \to q) \to q\big) = \ominus\big(\mathcal{I}(p \to q), \mathcal{I}((\neg p \to q) \to q)\big) = \ominus\,(\texttt{false}, \texttt{false}) = \texttt{true}$

Für komplexe Formeln ist die Auswertung von Hand viel zu mühsam und fehleranfällig um praktikabel zu sein. Wir zeigen deshalb 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

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

berechnen wollen, so müssen wir die Details der Teilformeln f und g nicht kennen, es reicht, wenn wir die Werte $\mathcal{I}(f)$ und $\mathcal{I}(g)$ kennen. Das ist problematisch, denn in der Umgangssprache hat der Junktor "*wenn* $\cdots$*, dann*" auch eine *kausale* Bedeutung. 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. Dass ist problematisch, weil wir diesen Satz in der Umgangssprache als sinnlos erkennen. Insofern ist die extensionale Interpretation des sprachlichen Junktors "*wenn* $\cdots$*, dann*" nur eine Approximation der umgangssprachlichen Interpretation, die sich für die Mathematik und die Informatik aber als ausreichend erwiesen hat.

Es gibt durchaus andere Logiken, in denen die Interpretation des Operators "$\to$" von der hier gegebenen Definition abweicht. Diese Logiken sind allerdings wesentlich komplizierter als die Form der Logik, die wir hier betrachten.

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. Jedesmal, wenn wir ein Programm zur Berechnung irgendwelcher Wert 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

$$\mathit{rep} : \mathcal{F} \to \text{SetlX}$$

definieren, die einer aussagenlogischen Formel f einen Term $\mathit{rep}(f)$ 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`.

   $\mathit{rep}(\top) := \texttt{true}$

2. $\bot$ wird repräsentiert durch den Wahrheitswert `false`.

   $\mathit{rep}(\bot) := \texttt{false}$

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

   `^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 `parse` in eine aussagenlogische Formel umwandeln wollen, dann werden Variablen automatisch in dieser Form dargestellt. Damit haben wir also

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

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

   $\mathit{rep}(\neg f) := \texttt{!}\,\mathit{rep}(f)$.

5. Sind f_1 und f_2 aussagenlogische Formel, so repräsentieren wir $f_1 \vee f_2$ mit Hilfe des Operators "`||`":

   $\mathit{rep}(f \vee g) := \mathit{rep}(f)$ `||` $\mathit{rep}(g)$.

6. Sind f_1 und f_2 aussagenlogische Formel, so repräsentieren wir $f_1 \wedge f_2$ mit Hilfe des Operators "`&&`":

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

7. Sind f_1 und f_2 aussagenlogische Formel, so repräsentieren wir $f_1 \rightarrow f_2$ mit Hilfe des Operators "`=>`":

   $\mathit{rep}(f \rightarrow g) := \mathit{rep}(f)$ `=>` $\mathit{rep}(g)$.

8. Sind f_1 und f_2 aussagenlogische Formel, so repräsentieren wir $f_1 \leftrightarrow f_2$ mit Hilfe des Operators "`<==>`":

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

Bei der Wahl der Repräsentation, mit der wir eine Formel in SetlX repäsentieren, 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 vordefinierte Funktion `parse` benutzen um einen String in eine Formel umzuwandeln. Beispielweise liefert der Aufruf

```
f := parse("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 57 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   evaluate := procedure(f, i) {
2       match (f) {
3           case true:          return true;
4           case false:         return false;
5           case ^variable(p): return i[p];
6           case !g:            return !evaluate(g, i);
7           case g && h:        return  evaluate(g, i) && evaluate(h, 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          default:            abort("syntax error in evaluate($f$, $i$)");
12      }
13  };
```

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 repäsentiert 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 repäsentiert f die Formel $\bot$. Also ist das Ergebnis der Auswertung unabhängig von der aussagenlogischen Interpretation i immer `false`.
3. In Zeile 5 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 6 betrachten wir den Fall, dass f die Form `!`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 7 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 8 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 9 betrachten wir den Fall, dass f die Form g_1 => 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 10 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 11 hinweisen.

4.3.5 Eine Anwendung

Wir betrachten eine spielerische Anwendung der Aussagenlogik. Inspektor Watson wird zu einem Juweliergeschäft gerufen, in das eingebrochen worden ist. In der unmittelbaren Umgebung werden drei Verdächtige 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

$$\mathcal{I} = \big\{\langle a, \texttt{false}\rangle, \langle b, \texttt{false}\rangle, \langle c, \texttt{true}\rangle\big\},$$

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. In den Zeilen 7 – 17 definieren wir die Formeln f_1, $\cdots$, f_6 . Wir müssen hier die Formeln in die SetlX-Repräsentation bringen. Diese Arbeit wird uns durch die Benutzung der Funktion `parse` leicht gemacht.

```
1  // This procedure turns a subset m of the set of all variables
2  // v into a propositional valuation i, such that i[x] is true
3  // iff x is an element of the set m.
4  createValuation := procedure(m, v) {
5      return { [ x, x in m ] : x in v };
6  };
7  // Austin, Brian, or Colin is guilty.
8  f1 := parse("a || b || c");
9  // If Austin is guilty, he has exactly one accomplice.
10 f2 := parse("a => b || c");    // at least one accomplice
11 f3 := parse("a => !(b && c)"); // at most  one accomplice
12 // If Brian is innocent, then Colin is innocent, too.
13 f4 := parse("!b => !c");
14 // If exactly two are guilty, then Colin is one of them.
15 f5 := parse("!(a && b && !c)");
16 // If Colin is innocent, then Austin is guilty.
17 f6 := parse("!c => a");
18 fs := { f1, f2, f3, f4, f5, f6 };
19 v  := { "a", "b", "c" };
20 all := 2 ** v;
21 print("all = ", all);
22 // b is the set of all propositional valuations.
23 b  := { createValuation(m, v) : m in all };
24 s  := { i : i in b | forall (f in fs | evaluate(f, i)) };
25 print("Set of all valuations satisfying all facts: ", s);
26 if (#s == 1) {
27     i := arb(s);
28     offenders := { x : x in v | i[x] };
29     print("Set of offenders: ", offenders);
30 }
```

Figure 4.2: Programm zur Aufkärung des Einbruchs.

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.

3. 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

   $$\mathcal{I} = \{\langle \text{"a"}, \texttt{true}\rangle, \langle \text{"b"}, \texttt{false}\rangle, \langle \text{"c"}, \texttt{true}\rangle\}.$$

   Das allgemeine Prinzip ist offenbar, dass für eine aussagenlogische Variable x das Paar $\langle x, \texttt{true}\rangle$ genau dann in der Belegung $\mathcal{I}$ enthalten ist, wenn $x \in m$ ist, andernfalls ist das Paar $\langle x, \texttt{false}\rangle$ in $\mathcal{I}$. Damit könnten wir die Menge aller Belegungen, die genau die Elemente aus m wahrmachen, wie folgt schreiben:

   ```
   { [ x, true ] : x in m } + { [ x, false ] : x in all | !(x 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.

4. In Zeile 23 sammeln wir in der Menge b alle möglichen Belegungen auf.
5. In Zeile 24 berechnen wir die Menge s aller der Belegungen i, für die alle Formeln aus der Menge *fs* wahr werden.
6. 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 anschließemd 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

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

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

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

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

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

$$\models f.$$ ◇

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

Beispiele:

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

Wir können die Tatsache, dass es sich bei diesen Formeln um Tautologien handelt, durch eine Tabelle nachweisen, die analog zu der auf Seite 54 gezeigten Tabelle 4.2 aufgebaut ist. Dieses Verfahren ist zwar konzeptuell sehr einfach, allerdings zu ineffizient, wenn die Anzahl der aussagenlogischen Variablen groß ist. Ziel dieses Kapitels ist daher die Entwicklung eines effizienteren Verfahren.

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

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

$$\models f \leftrightarrow g$$

gilt. ◇

Beispiele: Es gelten die folgenden Äquivalenzen:

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

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.

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 Wahrheitstafeln heraus. Solche Wahrheitstafeln 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 jedesmal der Wert `true` ergibt. Dazu müssen wir zunächst einen Weg finden, alle möglichen Belegungen einer Formel zu berechnen. Wir haben früher schon gesehen, dass Belegungen $\mathcal{I}$ zu Teilmengen M der Menge der aussagenlogischen Variablen $\mathcal{P}$ korrespondieren, denn für jedes $M \subseteq \mathcal{P}$ können wir eine aussagenlogische Belegung $\mathcal{I}(M)$ wie folgt definieren:

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

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

$$\mathcal{I} = \big\{\langle p, \texttt{true}\rangle \mid p \in M\big\} \cup \big\{\langle p, \texttt{false}\rangle \mid p \notin M\big\}.$$

Dies lässt sich noch zu

$$\mathcal{I} = \big\{\langle p, p \in M\rangle \mid p \in \mathcal{P}\big\}$$

vereinfachen. Mit dieser Idee können wir nun eine Prozedur implementieren, die für eine gegebene aussagenlogische Formel f testet, of 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 in a | !evaluate(f, i) });
9       }
10  };
11  collectVars := procedure(f) {
12      match (f) {
13          case true:         return {};
14          case false:        return {};
15          case ^variable(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 62 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 57 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, das 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

   $$\mathcal{I} : \mathcal{P} \to \mathbb{B} \quad \text{mit} \quad \mathcal{P} = \texttt{collectVars}(f)$$

   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

$$2^{\mathcal{P}} = \big\{\{\}, \{p\}, \{q\}, \{p, q\}\big\}.$$

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 $\mathcal{I}(m_i)$:

$\mathcal{I}(m_1) := \big\{\langle x, x \in \{\}\rangle \mid x \in \{p, q\}\big\} = \big\{\langle p, p \in \{\}\rangle, \langle q, q \in \{\}\rangle\big\} = \big\{\langle p, \texttt{false}\rangle, \langle q, \texttt{false}\rangle\big\}.$

$\mathcal{I}(m_2) := \big\{\langle x, x \in \{p\}\rangle \mid x \in \{p, q\}\big\} = \big\{\langle p, p \in \{p\}\rangle, \langle q, q \in \{p\}\rangle\big\} = \big\{\langle p, \texttt{true}\rangle, \langle q, \texttt{false}\rangle\big\}.$

$\mathcal{I}(m_3) := \big\{\langle x, x \in \{q\}\rangle \mid x \in \{p, q\}\big\} = \big\{\langle p, p \in \{q\}\rangle, \langle q, q \in \{q\}\rangle\big\} = \big\{\langle p, \texttt{false}\rangle, \langle q, \texttt{true}\rangle\big\}.$

$\mathcal{I}(m_4) := \big\{\langle x, x \in \{p, q\}\rangle \mid x \in \{p, q\}\big\} = \big\{\langle p, p \in \{p, q\}\rangle, \langle q, q \in \{p, q\}\rangle\big\} = \big\{\langle p, \texttt{true}\rangle, \langle q, \texttt{true}\rangle\big\}.$

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}{lcl}
 & (p \to q) \to (\neg p \to q) \to q & \text{(Elimination von } \to\text{)} \\
\leftrightarrow & (\neg p \vee q) \to (\neg p \to q) \to q & \text{(Elimination von } \to\text{)} \\
\leftrightarrow & (\neg p \vee q) \to (\neg\neg p \vee q) \to q & \text{(Elimination der Doppelnegation)} \\
\leftrightarrow & (\neg p \vee q) \to (p \vee q) \to q & \text{(Elimination von } \to\text{)} \\
\leftrightarrow & \neg(\neg p \vee q) \vee ((p \vee q) \to q) & \text{(DeMorgan)} \\
\leftrightarrow & (\neg\neg p \wedge \neg q) \vee ((p \vee q) \to q) & \text{(Elimination der Doppelnegation)} \\
\leftrightarrow & (p \wedge \neg q) \vee ((p \vee q) \to q) & \text{(Elimination von } \to\text{)} \\
\leftrightarrow & (p \wedge \neg q) \vee (\neg(p \vee q) \vee q) & \text{(DeMorgan)} \\
\leftrightarrow & (p \wedge \neg q) \vee ((\neg p \wedge \neg q) \vee q) & \text{(Distributivität)} \\
\leftrightarrow & (p \wedge \neg q) \vee ((\neg p \vee q) \wedge (\neg q \vee q)) & \text{(Tertium-non-Datur)} \\
\leftrightarrow & (p \wedge \neg q) \vee ((\neg p \vee q) \wedge \top) & \text{(Neutrales Element)} \\
\leftrightarrow & (p \wedge \neg q) \vee (\neg p \vee q) & \text{(Distributivität)} \\
\leftrightarrow & (p \vee (\neg p \vee q)) \wedge (\neg q \vee (\neg p \vee q)) & \text{(Assoziativität)} \\
\leftrightarrow & ((p \vee \neg p) \vee q) \wedge (\neg q \vee (\neg p \vee q)) & \text{(Tertium-non-Datur)} \\
\leftrightarrow & (\top \vee q) \wedge (\neg q \vee (\neg p \vee q)) & \text{(Neutrales Element)} \\
\leftrightarrow & \top \wedge (\neg q \vee (\neg p \vee q)) & \text{(Neutrales Element)} \\
\leftrightarrow & \neg q \vee (\neg p \vee q) & \text{(Assoziativität)} \\
\leftrightarrow & (\neg q \vee \neg p) \vee q & \text{(Kommutativität)} \\
\leftrightarrow & (\neg p \vee \neg q) \vee q & \text{(Assoziativität)} \\
\leftrightarrow & \neg p \vee (\neg q \vee q) & \text{(Tertium-non-Datur)} \\
\leftrightarrow & \neg p \vee \top & \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* $f = \bot$.
2. $f = p$, *wobei p eine aussagenlogische Variable ist.*
   In diesem Fall sprechen wir von einem positiven *Literal.*
3. $f = \neg p$, *wobei p eine aussagenlogische Variable ist.*
   In diesem Fall sprechen wir von einem negativen *Literal.*

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

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

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

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

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

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

$$k = l_1 \vee \cdots \vee l_r$$

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

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

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

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

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

in Mengen-Schreibweise, so erhalten wir

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

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

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

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

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

1. $\top \in k$.
2. *Es existiert $p \in \mathcal{P}$ mit $p \in k$ und $\neg p \in k$.*

   In diesem Fall bezeichnen wir p und $\neg p$ als komplementäre Literale*.* ◇

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

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

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

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

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

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

$$f = k_1 \wedge \cdots \wedge k_n,$$

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

Aus der Definition der KNF folgt sofort:

Korollar 14 *Ist $f = k_1 \wedge \cdots \wedge k_n$ in konjunktiver Normalform, so gilt*

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

Damit können wir für eine Formel $f = k_1 \wedge \cdots \wedge k_n$ in konjunktiver Normalform leicht entscheiden, ob f eine Tautologie ist, denn f ist genau dann eine Tautologie, wenn alle Klauseln k_i trivial sind.

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

$$f = k_1 \wedge \cdots \wedge k_n$$

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

$$f = \{k_1, \cdots, k_n\}.$$

Wir geben ein Beispiel: Sind p, q und r Aussage-Variablen, so ist die Formel

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

in konjunktiver Normalform. In Mengen-Schreibweise wird daraus

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

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

1. Eliminiere alle Vorkommen des Junktors "$\leftrightarrow$" mit Hilfe der Äquivalenz
   $$(f \leftrightarrow g) \leftrightarrow (f \rightarrow g) \wedge (g \rightarrow f)$$
2. Eliminiere alle Vorkommen des Junktors "$\rightarrow$" mit Hilfe der Äquivalenz
   $$(f \rightarrow g) \leftrightarrow \neg f \vee g$$
3. Schiebe die Negationszeichen soweit es geht nach innen. Verwende dazu die folgenden Äquivalenzen:
   (a) $\neg\bot \leftrightarrow \top$
   (b) $\neg\top \leftrightarrow \bot$
   (c) $\neg\neg f \leftrightarrow f$
   (d) $\neg(f \wedge g) \leftrightarrow \neg f \vee \neg g$
   (e) $\neg(f \vee g) \leftrightarrow \neg f \wedge \neg g$

   In dem Ergebnis, das wir nach diesem Schritt erhalten, stehen die Negationszeichen nur noch unmittelbar vor den aussagenlogischen Variablen. Formeln mit dieser Eigenschaft bezeichnen wir auch als Formeln in *Negations-Normalform*.
4. Stehen in der Formel jetzt "$\vee$"-Junktoren über "$\wedge$"-Junktoren, so können wir durch *Ausmultiplizieren*, sprich Verwendung der Distributiv-Gesetze
   $$f \vee (g \wedge h) \leftrightarrow (f \vee g) \wedge (f \vee h) \quad \text{und} \quad (f \wedge g) \vee h \leftrightarrow (f \vee h) \wedge (g \vee h)$$
   diese Junktoren 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

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

1. Da die Formel den Junktor "$\leftrightarrow$" nicht enthält, ist im ersten Schritt nichts zu tun.
2. Die Elimination von "$\to$" 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
$$\big\{\{p, \neg q\}\big\}.$$
Beachten Sie, dass sich die Formel durch die Überführung in Mengen-Schreibweise noch einmal deutlich vereinfacht hat.

Damit ist die Formel in KNF überführt.

4.4.3 Berechnung der konjunktiven Normalform in 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

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

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

1. Wenn f eine Aussage-Variable p ist, so ist nichts zu tun:
$$\texttt{elimGdw}(p) = p \quad \text{für alle } p \in \mathcal{P}.$$
2. Hat f die Form $f = \neg g$, so eliminieren wir den Junktor "$\leftrightarrow$" aus der Formel g:
$$\texttt{elimGdw}(\neg g) = \neg\texttt{elimGdw}(g).$$
3. Im Falle $f = g_1 \wedge g_2$ eliminieren wir den Junktor "$\leftrightarrow$" aus den Formeln g_1 und g_2:
$$\texttt{elimGdw}(g_1 \wedge g_2) = \texttt{elimGdw}(g_1) \wedge \texttt{elimGdw}(g_2).$$

4. Im Falle $f = g_1 \vee g_2$ eliminieren wir den Junktor "$\leftrightarrow$" aus den Formeln g_1 und g_2:

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

5. Im Falle $f = g_1 \rightarrow g_2$ eliminieren wir den Junktor "$\leftrightarrow$" aus den Formeln g_1 und g_2:

$$\texttt{elimGdw}(g_1 \rightarrow g_2) = \texttt{elimGdw}(g_1) \rightarrow \texttt{elimGdw}(g_2).$$

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

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

   Das führt auf die Gleichung:

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

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

Abbildung 4.4 auf Seite 68 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 68 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 Junktor "$\leftrightarrow$" bereits vorher eliminiert wurde. Dadurch entfällt ein Fall.

```
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 $\rightarrow$.

Als nächstes zeigen wir die Routinen zur Berechnung der Negations-Normalform. Abbildung 4.6 auf Seite 70 zeigt die Implementierung der Funktionen nnf und neg, die sich wechselseitig aufrufen. Dabei berechnet neg(f)

die Negations-Normalform von $\neg f$, während $\mathsf{nnf}(f)$ die Negations-Normalform von f berechnet, es gilt also

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

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. $\mathsf{nnf}(\neg f) = \mathtt{neg}(f)$,
2. $\mathsf{nnf}(f_1 \wedge f_2) = \mathtt{nnf}(f_1) \wedge \mathtt{nnf}(f_2)$,
3. $\mathsf{nnf}(f_1 \vee f_2) = \mathtt{nnf}(f_1) \vee \mathtt{nnf}(f_2)$.

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

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

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 Literale und eine Menge von Klauseln interpretieren wir als Konjunktion der Klauseln. Abbildung 4.7 auf Seite 71 zeigt die Implementierung der Funktion knf.

1. Falls die Formel f, die wir in KNF transformieren wollen, die Form

   $$f = \neg g$$

   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 $\big\{\{f\}\big\}$ 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

   $$\mathtt{knf}(f_1) = \{h_1, \cdots, h_m\} \quad \text{und} \quad \mathtt{knf}(f_2) = \{k_1, \cdots, k_n\}.$$

   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

   $$\mathtt{knf}(f_1 \wedge f_2) = \mathtt{knf}(f_1) \cup \mathtt{knf}(f_2).$$

   Das liefert Zeile 4 der Implementierung.

```
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.

3. Falls $f = f_1 \vee f_2$ ist, transformieren wir zunächst f_1 und f_2 in KNF. Dabei erhalten wir

   $$\texttt{knf}(f_1) = \{h_1, \cdots, h_m\} \quad \text{und} \quad \texttt{knf}(f_2) = \{k_1, \cdots, k_n\}.$$

   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:

   $$\begin{array}{lll} & f_1 \vee f_2 & \\ \leftrightarrow & (h_1 \wedge \cdots \wedge h_m) \vee (k_1 \wedge \cdots \wedge k_n) & \\ \leftrightarrow & (h_1 \vee k_1) \quad \wedge \quad \cdots \quad \wedge \quad (h_m \vee k_1) \quad \wedge & \\ & \quad \vdots \qquad\qquad\qquad\qquad\qquad\quad \vdots & \\ & (h_1 \vee k_n) \quad \wedge \quad \cdots \quad \wedge \quad (h_m \vee k_n) & \\ \leftrightarrow & \big\{h_i \vee k_j : i \in \{1, \cdots, m\}, j \in \{1, \cdots, n\}\big\} & \end{array}$$

   Berücksichtigen wir noch, dass Klauseln in der Mengen-Schreibweise als Mengen von Literalen aufgefasst werden, die implizit disjunktiv verknüpft werden, so können wir für $h_i \vee k_j$ auch $h_i \cup k_j$ schreiben. Insgesamt erhalten wir damit

   $$\texttt{knf}(f_1 \vee f_2) = \big\{h \cup k \mid h \in \texttt{knf}(f_1) \ \wedge \ k \in \texttt{knf}(f_2)\big\}.$$

   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 $\big\{\{f\}\big\}$. Dieses Ergebnis geben wir in Zeile 6 zurück.

Zum Abschluss zeigen wir in Abbildung 4.8 auf Seite 71 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. Im letzten Schritt entfernt die Funktion `simplify` alle Klauseln, die trivial sind. Eine Klausel k ist dann trivial, wenn es eine aussagenlogische Variablen p gibt, so dass sowohl p als auch $\neg p$ in k der Menge k auftritt. Daher berechnet die Funktion `isTrivial` für eine Klausel c zunächst die Menge

```
{ p in c | fct(p) == "^variable" }
```

```
knf := procedure(f) {
    match (f) {
        case !g     : return { { !g } };
        case g && h : return knf(g) + knf(h);
        case g || h : return { k1 + k2 : k1 in knf(g), k2 in knf(h) };
        default     : return { { f } }; // f is a variable
    }
};
```

Figure 4.7: Berechnung der konjunktiven Normalform.

aller Variablen, die in der Klausel c auftreten. Anschließend wird die Menge

```
{ args(l)[1] : l in c | fct(l) == "^not" }
```

berechnet. Diese Menge enthält alle die aussagenlogischen Variablen p, für die $\neg p$ ein Element der Klausel c ist, denn die Formel $\neg p$ wird intern in SetlX durch den Term

```
^not(p)
```

dargestellt. Falls also das Literal l die Form $l = \neg p$ hat, so hat der Term l das Funktions-Zeichen `^not`. Die Anwendung der Funktion `fct` auf l liefert uns genau dieses Funktions-Zeichen. Der Ausdruck `args`(l) berechnet die Liste der Argumente des Funktions-Zeichens `^not` und diese Liste enthält als einziges Element gerade die aussagenlogische Variable p, die wir daher durch den Ausdruck

```
args(l)[1]
```

aus dem Literal l extrahieren können. Falls nun die beiden oben berechneten Mengen ein gemeinsames Element haben, so ist die Klausel c trivial. Dies wird dadurch geprüft, dass der Schnitt der beiden Mengen berechnet wird. Das vollständige Programm zur Berechnung der konjunktiven Normalform finden Sie als die Datei `knf.stlx` unter GitHub.

```
normalize := procedure(f) {
    n1 := elimGdw(f);
    n2 := elimFolgt(n1);
    n3 := nnf(n2);
    n4 := knf(n3);
    return simplify(n4);
};
simplify := procedure(k) {
    return { c : c in k | !isTrivial(c) };
};
isTrivial := procedure(c) {
    return { p : p in c | fct(p) == "^variable" } *
           { args(l)[1] : l in c | fct(l) == "^not" } != {};
};
```

Figure 4.8: Normalisierung einer Formel

4.5 Der Herleitungs-Begriff

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

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

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

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

äquivalent ist. Diese Formel ist nun genau dann eine Tautologie, wenn jede der Klauseln k_1, $\cdots$, 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 eine weiteres Verfahren vor, mit dessen Hilfe wir entscheiden können, ob eine Formel aus einer gegebenen Menge von Formeln folgt. Die Idee bei diesem Verfahren ist es, die Formel f mit Hilfe von *Schluss-Regeln* aus den gegebenen Formeln $f_1, \cdots, f_n$ herzuleiten. Das Konzept einer Schluss-Regel wird in der nun folgenden Definition festgelegt.

Definition 15 (Schluss-Regel) *Eine* Schluss-Regel *ist eine Paar $\langle\{f_1, \cdots, f_n\}, k\rangle$. Dabei ist $\{f_1, \cdots, f_n\}$ eine Menge von Formeln und k ist eine einzelne Formel. Die Formeln f_1, $\cdots$, f_n bezeichnen wir als* Prämissen*, die Formel k heißt die* Konklusion *der Schluss-Regel. Ist das Paar $\langle\{f_1, \cdots, f_n\}, k\rangle$ eine Schluss-Regel, so schreiben wir dies als:*

$$\frac{f_1 \quad \cdots \quad f_n}{k}.$$

Wir lesen diese Schluss-Regel wie folgt: "Aus f_1, $\cdots$, f_n kann auf k geschlossen werden." ◇

Beispiele für Schluss-Regeln:

Modus Ponens	*Modus Ponendo Tollens*	*Modus Tollendo Tollens*
$\dfrac{p \quad p \rightarrow q}{q}$	$\dfrac{\neg q \quad p \rightarrow q}{\neg p}$	$\dfrac{\neg p \quad p \rightarrow q}{\neg q}$

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*

$$\frac{f_1 \quad \cdots \quad f_n}{k}$$

ist genau dann korrekt, *wenn* $\models f_1 \wedge \cdots \wedge f_n \rightarrow k$ *gilt.* ◇

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:

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

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

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

Wenn wir jetzt noch berücksichtigen, dass die Formel $\neg p \vee q$ äquivalent 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:

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

□

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

$$M \vdash f.$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $\{k_1, \cdots, k_n\}$ eine Menge von Klauseln und k eine einzelne Klausel, so haben wir:*

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

Beweis: Der Beweis verläuft durch eine Induktion nach der Definition der Relation $\vdash$.

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

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

   zeigen, was offensichtlich ist.

2. Fall: Es gilt $\{k_1, \cdots, k_n\} \vdash k$ weil es eine aussagenlogische Variable p und Klauseln g und h gibt, so dass

   $$\{k_1, \cdots, k_n\} \vdash g \cup \{p\} \quad \text{und} \quad \{k_1, \cdots, k_n\} \vdash h \cup \{\neg p\}$$

   gilt und daraus haben wir mit der Schnitt-Regel auf

   $$\{k_1, \cdots, k_n\} \vdash g \cup h$$

geschlossen, wobei $k = g \cup h$ gilt.

Nach Induktions-Voraussetzung haben wir dann

$$\models k_1 \wedge \cdots \wedge k_n \rightarrow g \vee p \quad \text{und} \quad \models k_1 \wedge \cdots \wedge k_n \rightarrow h \vee \neg p.$$

Wegen

$$\models (g \vee p) \wedge (h \vee \neg p) \rightarrow g \vee h \quad \text{und} \quad k = g \cup h$$

folgt daraus die Behauptung. □

Die Umkehrung dieses Satzes gilt leider nur in abgeschwächter Form und zwar dann, wenn k die leere Klausel ist, also im Fall $k = \bot$.

Satz 21 (Widerlegungs-Vollständigkeit) *Ist $M = \{k_1, \cdots, k_n\}$ eine Menge von Klauseln, so haben wir:*

Wenn $\models k_1 \wedge \cdots \wedge k_n \rightarrow \bot$ 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*

$$M \models \bot$$

wenn es keine aussagenlogische Interpretation $\mathcal{I}$ gibt, die alle Formel aus M erfüllt. Bezeichnen wir die Menge der aussagenlogischen Interpretationen mit $\textsc{Ali}$, so schreibt sich das formal als

$$M \models \bot \quad \textit{g.d.w.} \quad \forall \mathcal{I} \in \textsc{Ali} : \exists g \in M : \mathcal{I}(g) = \texttt{false}.$$ ◇

Bemerkung: Ist $M = \{k_1, \cdots, k_n\}$ eine Menge von Klauseln, so können Sie sich leicht überlegen, dass M genau dann nicht erfüllbar ist, wenn

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

gilt. ◇

Wir führen den Bewies 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.

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 $c_2 \cup \{\overline{l}\}$ Klauseln aus S sind, dann ist auch die Klausel $c_1 \cup c_2$ ein Element der Klauselmenge S.

Wir erläutern nun die Implementierung der Funktion `saturate`.

```
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;
31          }
32          clauses += derived;
33      }
34  };
```

Figure 4.10: Die 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 eines `return`-Befehls 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 vorher hatten, 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 23 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      vars    := collectVars(clauses);
37      clauses := saturate(clauses);
38      if ({} in clauses) {
39          return false;
40      }
41      literals := {};  // refuted literal
42      for (p in vars) {
43          if (exists(c in clauses | p in c && c <= literals + {p})) {
44              literals += { !p };
45          } else {
46              literals += { p };
47          }
48      }
49      result := { complement(l) : l in literals };
50      return result;
51  };
```

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 $\{\texttt{true}, \texttt{false}\}$ 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 `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 `false` 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 `result` zurück, bei der wir jedes Literal l aus der Menge `literals` durch sein Komplement $\overline{l}$ ersetzen.

   Im engeren Sinne ist die Menge `result` keine aussagenlogische Belegung, aber wir können aus dieser Menge leicht eine aussagenlogische Belegung $\mathcal{I}$ erzeugen, indem wir $\mathcal{I}$ wie folgt definieren:

   $$\mathcal{I}(p) := \begin{cases} \texttt{true} & \text{falls } p \in \texttt{result}; \\ \texttt{false} & \text{sonst.} \end{cases}$$

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 $\neg p$ zu der Menge `literals` hinzufügen, wenn die Belegung $\mathcal{I}$ die Variable p auf `true` abbilden muss um die Klauseln zu erfüllen.

   Die Bedingung dafür ist wie folgt: Angenommen, wir haben bereits Werte für die Variablen $p_1, \cdots, p_n$ in der Menge `literals` gefunden. Die Werte dieser Variablen seien durch die Literale $l_1, \cdots, l_n$ in der Menge

`literals` wie folgt festgelegt: Wenn $l_i = \neg p_i$ ist, dann gilt $\mathcal{I}(p_i) = \texttt{true}$ und falls $l_i = p_i$ gilt, so haben wir $\mathcal{I}(p_i) = \texttt{false}$. Nehmen wir nun weiter an, dass eine Klausel c in der Menge `clauses` existiert, so dass

$$c \subseteq \{l_1, \cdots, l_n, p\} \quad \text{und} \quad p \in c$$

gilt. Wenn $\mathcal{I}(c) = \texttt{true}$ gelten soll, dann muss $\mathcal{I}(p) = \texttt{true}$ gelten, denn nach Konstruktion von $\mathcal{I}$ gilt

$$\mathcal{I}(l_i) = \texttt{false} \quad \text{für alle } i \in \{1, \cdots, n\}$$

und damit ist p das einzige Literal in der Klausel c, das wir mit Hilfe der Belegung $\mathcal{I}$ überhaupt noch wahr machen können. In diesem Fall fügen wir also das Literal $\neg p$ in die Menge `literals` ein.

Falls wir keine solche Klausel c finden, dann setzen wir $\texttt{I}(p) := \texttt{false}$ und fügen 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 wie die aussagenlogischen Variablen, die in der Menge `clauses` auftreten, in der selben Reihenfolge durch, in der diese Variablen in der `for`-Schleife in Zeile 42 betrachtet werden. Wir bezeichnen diese Variablen als

$$p_1, p_2, p_3, \cdots, p_k$$

und zeigen durch Induktion nach n, dass nach n Durchläufen der Schleife für jede Klausel $d \in \texttt{clauses}$, in der nur die Variablen p_1, $\cdots$, p_n vorkommen,

$$\mathcal{I}(d) = \texttt{true}$$

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, $\cdots$, p_n, p_{n+1} vorkommen. Die Klausel ist dann eine Teilmenge einer Menge der Form

$$\{l_1, \cdots, l_n, l_{n+1}\}, \quad \text{wobei } l_i \in \{p_i, \neg p_i\} \text{ für alle } i \in \{1, \cdots, n+1\} \text{ gilt.}$$

Nun gibt es mehrere Möglichkeiten, die wir getrennt untersuchen.

(a) Es gibt ein $i \in \{1, \cdots, n\}$, so dass $l_i \in d$ und $\mathcal{I}(l_i) = \texttt{true}$ ist.
Da eine Klausel als Disjunktion ihrer Literale aufgefasst wird, gilt dann auch $\mathcal{I}(d) = \texttt{true}$ unabhängig davon, ob $\mathcal{I}(p_{n+1})$ den Wert `true` oder `false` hat.

(b) Für alle $i \in \{1, \cdots, n\}$ mit $l_i \in d$ gilt $\mathcal{I}(l_i) = \texttt{false}$ und es gilt $l_{n+1} = p_{n+1}$.
Dann gilt für die Klausel d gerade die Bedingung

$$d \subseteq \texttt{literals} \cup \{p_{n+1}\}$$

und daher wird in Zeile 44 der Funktion `findValuation` das Literal $\neg p_{n+1}$ zu der Menge `literals` hinzugefügt. Nach Definition der Belegung $\mathcal{I}$, die von der Funktion `findValuation` zurück gegeben wird, heißt dies gerade, dass

$$\mathcal{I}(p_{n+1}) = \texttt{true}$$

ist und dann gilt natürlich auch $\mathcal{I}(d) = \texttt{true}$.

(c) Für alle $i \in \{1, \cdots, n\}$ mit $l_i \in d$ gilt $\mathcal{I}(l_i) = \texttt{false}$ und es gilt $l_{n+1} = \neg p_{n+1}$.
An dieser Stelle ist eine weitere Fall-Unterscheidung notwendig.

i. Es gibt eine Klausel c in der Menge `clauses`, so dass

$$c \subseteq \texttt{literals} \cup \{p_{n+1}\}$$

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 $\neg p_{n+1}$ zur Menge `literals` hinzugefügt und damit wäre zunächst $\mathcal{I}(p_{n+1}) = \texttt{true}$ und damit $\mathcal{I}(\neg p_{n+1}) = \texttt{false}$, woraus insgesamt $\mathcal{I}(d) = \texttt{false}$ 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

$$c' \subseteq \texttt{literals} \quad \text{und} \quad d' \subseteq \texttt{literals}$$

gelten würde. Daraus würde sowohl

$$\mathcal{I}(c') = \texttt{false} \quad \text{als auch} \quad \mathcal{I}(d') = \texttt{false}$$

folgen und das würde auch

$$\mathcal{I}(c' \cup d') = \texttt{false} \tag{$*$}$$

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 $p_1, \cdots, p_n$ enthält, gilt nach Induktions-Voraussetzung

$$\mathcal{I}(c' \cup d') = \texttt{true}.$$

Dies steht aber im Widerspruch zu $(*)$. Dieser Widerspruch zeigt, dass es keine Klausel $c \in \texttt{clauses}$ mit

$$c \subseteq \texttt{literals} \cup \{p_{n+1}\} \quad \text{und} \quad p_{n+1} \in c$$

geben kann und damit tritt der hier untersuchte Fall gar nicht auf.

ii. Es gibt <u>keine</u> Klausel c in der Menge `clauses`, so dass

$$c \subseteq \texttt{literals} \cup \{p_{n+1}\} \quad \text{und} \quad p_{n+1} \in c$$

gilt. In diesem Fall wird das Literal p_{n+1} zur Menge `literals` hinzugefügt und damit gilt zunächst $\mathcal{I}(p_{n+1}) = \texttt{false}$ und folglich $\mathcal{I}(\neg p_{n+1}) = \texttt{true}$, woraus schließlich $\mathcal{I}(d) = \texttt{true}$ folgt.

Wir sehen, dass der erste Fall der vorherigen Fall-Unterscheidung nicht auftritt und dass im zweiten Fall $\mathcal{I}(d) = \texttt{true}$ gilt, womit wir insgesamt $\mathcal{I}(d) = \texttt{true}$ gezeigt haben. Damit ist der Induktions-Schritt abgeschlossen.

Da jede Klausel $c \in \texttt{clauses}$ nur eine endliche Anzahl von Variablen enthält, haben wir insgesamt gezeigt, dass für alle diese Klauseln $\mathcal{I}(c) = \texttt{true}$ gilt. □

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

$$M \models \bot$$

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 $d \in \texttt{clauses}$

$$\mathcal{I}(d) = \texttt{true}$$

gilt. Die Menge M ist aber eine Teilmenge der Menge `clauses` und damit sehen wir, dass die Menge M erfüllbar ist. Dies steht im Widerspruch zu $M \models \bot$ 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. □

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

$$\texttt{eval}(k, \mathcal{I}) = \texttt{true} \quad \text{für alle } k \in K$$

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 [DLL62] zurück. Verfeinerungen dieses Verfahrens werden beispielsweise eingesetzt, um die Korrektheit digitaler elektronischer Schaltungen nachzuweisen.

Um das Verfahren zu motivieren überlegen wir zunächst, bei welcher Form der Klausel-Menge K unmittelbar klar ist, ob es eine Belegung gibt, die K löst und wie diese Belegung aussieht. Betrachten wir dazu ein Beispiel:

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

Die Klausel-Menge K_1 entspricht der aussagenlogischen Formel

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

Daher ist K_1 lösbar und die Belegung

$$\mathcal{I} = \{\ \langle p, \texttt{true}\rangle,\ \langle q, \texttt{false}\rangle,\ \langle r, \texttt{true}\rangle,\ \langle s, \texttt{false}\rangle,\ \langle t, \texttt{false}\rangle\ \}$$

ist eine Lösung. Betrachten wir eine weiteres Beispiel:

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

Diese Klausel-Menge entspricht der Formel

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

Offensichtlich ist K_2 unlösbar. Als letztes Beispiel betrachten wir

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

Diese Klausel-Menge kodiert die Formel

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

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*

$$k = \{p\} \quad \textit{oder} \quad k = \{\neg p\}$$

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*
   $$\forall k \in K : \mathit{card}(k) = 1 \quad \textit{und} \quad \forall p \in \mathcal{P} : \neg\big(\{p\} \in K \wedge \{\neg p\} \in K\big).$$
   In diesem Fall ist die aussagenlogische Belegung
   $$\mathcal{I} = \big\{\langle p, \texttt{true}\rangle \mid \{p\} \in K\big\} \cup \big\{\langle p, \texttt{false}\rangle \mid \{\neg p\} \in K\big\}$$
   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 Möglickeiten 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 Klausel eine Unit-Klausel ist. Solche Schnitte bezeichnen wir als *Unit-Schnitte*. Um alle mit einer gegebenen Unit-Klausel $\{l\}$ möglichen Schnitte durchführen zu können, definieren wir eine Funktion

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

so, dass für eine Klausel-Menge K und ein Literal l die Funktion $\mathit{unitCut}(K, l)$ die Klausel-Menge K soweit wie möglich mit Unit-Schnitten mit der Klausel $\{l\}$ vereinfacht:

$$\mathit{unitCut}(K, l) = \Big\{k \backslash \big\{\overline{l}\big\} \;\Big|\; k \in K\Big\}.$$

Beachten Sie, dass die Menge $\mathit{unitCut}(K, l)$ 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

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

Offenbar impliziert die Klausel $\{p\}$ die Klausel $\{p, q, \neg r\}$, denn immer wenn $\{p\}$ erfüllt ist, ist automatisch auch $\{q, p, \neg r\}$ erfüllt. Das liegt daran, dass

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

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

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

die eine gegebene Klauselmenge K, welche die Unit-Klausel $\{l\}$ enthält, mittels Subsumption dadurch vereinfacht, dass alle durch $\{l\}$ subsumierten Klauseln aus K gelöscht werden. Die Unit-Klausel $\{l\}$ selbst behalten wir natürlich. Daher definieren wir:

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

In der obigen Definition muss $\{l\}$ in das Ergebnis eingefügt werden, weil die Menge $\big\{k \in K \mid l \notin k\big\}$ 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 $K \cup \big\{\{p\}\big\}$ oder $K \cup \big\{\{\neg p\}\big\}$ erfüllbar ist.*

Beweis: Ist K erfüllbar durch eine Belegung $\mathcal{I}$, so gibt es für $\mathcal{I}(p)$ zwei Möglichkeiten: Falls $\mathcal{I}(p) = \texttt{true}$ ist, ist damit auch die Menge $K \cup \big\{\{p\}\big\}$ erfüllbar, andernfalls ist $K \cup \big\{\{\neg p\}\big\}$ erfüllbar.

Da K sowohl eine Teilmenge von $K \cup \big\{\{p\}\big\}$ als auch von $K \cup \big\{\{\neg p\}\big\}$ ist, ist klar, dass K erfüllbar ist, wenn eine dieser Mengen erfüllbar sind. □

Wir können nun eine Menge K von Klauseln dadurch vereinfachen, dass wir eine aussagenlogische Variable p wählen, die in K vorkommt. Anschließend bilden wir die Mengen

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

und untersuchen rekursiv ob K_1 erfüllbar ist. Falls wir eine Lösung für K_1 finden, ist dies auch eine Lösung für die ursprüngliche Klausel-Menge K und wir haben unser Ziel erreicht. Andernfalls untersuchen wir rekursiv ob K_2 erfüllbar ist. Falls wir nun eine Lösung finden, ist dies auch eine Lösung von K und wenn wir für K_2 keine Lösung finden, dann hat auch K keine Lösung. 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:

$$\mathcal{I}(k) = \texttt{true} \quad \text{für alle } k \in K.$$

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
   $$K \cup \big\{\{p\}\big\}$$
   zu lösen. Falls diese gelingt, haben wir eine Lösung von K.
   (b) Andernfalls versuchen wir die Klausel-Menge
   $$K \cup \big\{\{\neg p\}\big\}$$
   zu lösen. Wenn auch dies fehlschlägt, ist K unlösbar, andernfalls haben wir eine Lösung von K.

Für die Implementierung ist es zweckmäßig, die beiden oben definierten Funktionen *unitCut*() und *subsume*() zusammen zu fassen. Wir definieren eine Funktion

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

wie folgt:

$$\mathit{reduce}(K, l) = \big\{\, k \backslash \{\bar{l}\} \mid k \in K \wedge \bar{l} \in k \,\big\} \cup \big\{\, k \in K \mid \bar{l} \notin k \wedge l \notin k \big\} \cup \big\{\{l\}\big\}.$$

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 $\mathit{reduce}(K, l)$ 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:

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

Wir zeigen nun mit dem Verfahren von Davis und Putnam, dass K nicht lösbar ist. Da die Menge K keine Unit-Klauseln enthält, ist im ersten Schritt nichts zu tun. Da K nicht trivial ist, sind wir noch nicht fertig. Also gehen wir jetzt zu Schritt 3 und wählen eine aussagenlogische Variable, die in K auftritt. An dieser Stelle ist es sinnvoll eine Variable zu wählen, die in möglichst vielen Klauseln von K auftritt. Wir wählen daher die aussagenlogische Variable p.

1. Zunächst bilden wir die Menge
   $$K_0 := K \cup \big\{\{p\}\big\}$$
   und versuchen, diese Menge zu lösen. Dazu bilden wir
   $$K_1 := \mathit{reduce}\big(K_0, p\big) = \Big\{\{r, \neg t\},\ \{r, s\},\ \{\neg r, q\},\ \{\neg q, s, \neg r\},\ \{\neg r, \neg s\},\ \{\neg s\},\ \{p\}\Big\}.$$
   Die Klausel-Menge K_1 enthält die Unit-Klausel $\{\neg s\}$, so dass wir als nächstes mit dieser Klausel reduzieren können:
   $$K_2 := \mathit{reduce}\big(K_1,\ \neg s\big) = \Big\{\{r, \neg t\},\ \{r\},\ \{\neg r, q\},\ \{\neg q, \neg r\},\ \{\neg s\},\ \{p\}\Big\}.$$

Hier haben wir nun die neue Unit-Klausel $\{r\}$, mit der wir als nächstes reduzieren:

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

Da K_3 die Unit-Klausel $\{q\}$ enthält, reduzieren wir jetzt mit q:

$$K_4 := \mathit{reduce}\big(K_2,\ q\big) = \Big\{\{r\},\ \{q\},\ \{\},\ \{\neg s\},\ \{p\}\Big\}.$$

Die Klausel-Menge K_4 enthält die leere Klausel und ist damit unlösbar.

2. Also bilden wir jetzt die Menge

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

   und versuchen, diese Menge zu lösen. Dazu bilden wir

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

   Die Menge K_6 enthält die Unit-Klausel $\{\neg s\}$. Wir bilden daher

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

   Die Menge K_7 enthält die neue Unit-Klausel $\{q\}$, mit der wir als nächstes reduzieren:

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

   Da K_8 die leere Klausel enthält, ist K_8 und damit auch die ursprünglich gegebene Menge K unlösbar.

Bei diesem Beispiel hatten wir Glück, denn wir mussten nur eine einzige Fallunterscheidung durchführen. Bei komplexeren Beispielen ist es häufig so, dass wir 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 86 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 $\big\{\{\}\big\}$ zurück, denn die leere Klausel repräsentiert die unerfüllbare Formel $\bot$.

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 $\big\{\{\}\big\}$ 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

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

   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

   $$\texttt{clauses} \cup \big\{\{\neg\texttt{l}\}\big\}$$

   lösbar ist. Ist diese Menge lösbar, so ist diese Lösung auch eine Lösung der Menge `clauses` und wir geben diese Lösung zurück. Ist die Menge unlösbar, dann muss auch die Menge `clauses` unlösbar sein.

```
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      r := davisPutnam(clauses + { {l} }, literals + { l, notL });
12      if (r != { {} }) {
13          return r;
14      }
15      return davisPutnam(clauses + { {notL} }, literals + { l, 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 87 gezeigt.

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, die 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 $\{\texttt{l}\}$ möglich sind und entfernt darüber hinaus alle Klauseln, die durch die Unit-Klausel $\{\texttt{l}\}$ subsumiert werden.

```
1   saturate := procedure(s) {
2       units := { k in s | #k == 1 };
3       used := {};
4       while (units != {}) {
5           unit  := arb(units);
6           used  := used + { unit };
7           l     := arb(unit);
8           s     := reduce(s, l);
9           units := { k in s | #k == 1 } - used;
10      }
11      return s;
12  };
```

Figure 4.13: Die Prozedur `saturate`.

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*(K, l), die eine Klausel-Menge K mit Hilfe des Literals l reduziert, als

$$\mathit{reduce}(K,l) = \big\{\, k \backslash \{\bar{l}\} \mid k \in K \wedge \bar{l} \in k \,\big\} \cup \big\{\, k \in K \mid \bar{l} \notin k \wedge l \notin k \big\} \cup \big\{\{l\}\big\}$$

definiert. Die Implementierung setzt diese Definition unmittelbar um.

```
1   reduce := procedure(s, l) {
2       notL := negateLiteral(l);
3       return   { k - { notL } : k in s | notL in k }
4              + { k in s | !(notL in k) && !(l in k) }
5              + { {l} };
6   };
```

Figure 4.14: Die Prozedur `reduce`.

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 88 gezeigt wird.

1. Die Prozedur `selectLiteral` wählt ein beliebiges Literal aus einer gegeben 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 `arb()` ein Literal ausgewählt.

2. Die Prozedur `negateLiteral` bildet die Negation $\bar{l}$ eines gegebenen Literals l.

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:

```
selectLiteral := procedure(s, forbidden) {
    return arb(+/ s - forbidden);
};
negateLiteral := procedure(l) {
    match (l) {
        case !p : return p;
        case  p : return !p;
    }
};
```

Figure 4.15: Die Prozeduren `select` und `negateLiteral`.

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 89 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.

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

$$\texttt{Var}(i, j) \quad \text{mit } i, j \in \{1, \cdots, 8\}$$

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 90 zeigt die Zuordnung der Variablen zu den Feldern.

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

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

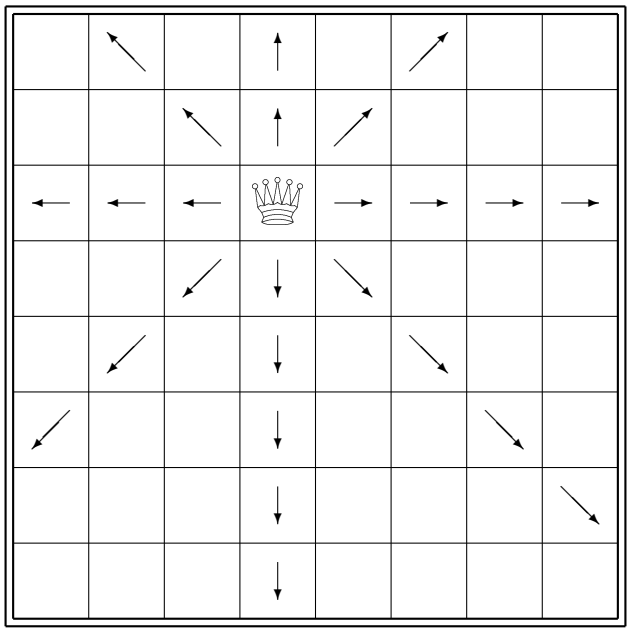

Figure 4.16: Das 8-Damen-Problem.

gegeben, so brauchen wir eine Formel die aussagt, dass ***höchstens*** eine der Variablen aus V den Wert `true` hat. Das ist aber gleichbedeutend damit, dass für jedes Paar $x_i, x_j \in V$ mit $x_i \neq x_j$ die folgende Formel gilt:

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

Diese Formel drückt aus, dass die Variablen x_i und x_j nicht gleichzeitig den Wert `true` annehmen. Nach den DeMorgan'schen Gesetzen gilt

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

und die Klausel auf der rechten Seite dieser Äquivalenz schreibt sich in Mengen-Schreibweise als

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

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

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

geschrieben werden. Wir setzen diese Überlegungen in eine SetlX-Prozedur um. Die in Abbildung 4.18 gezeigte Prozedur `atMostOne()` bekommt als Eingabe eine Menge V von aussagenlogischen Variablen. Der Aufruf `atMostOne`(V) berechnet eine Menge von Klauseln. Diese Klauseln sind genau dann wahr, wenn höchstens eine der Variablen aus V den Wert `true` hat.

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

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

auf und rufen mit dieser Menge die Hilfs-Prozedur `atMostOne()` auf, die das Ergebnis als Menge von Klauseln liefert.

Als nächstes berechnen wir eine Formel die aussagt, dass mindestens eine Dame in einer gegebenen Spalte steht.

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.

```
1  atMostOne := procedure(v) {
2      return { { !p, !q } : p in v, q in v | p != q };
3  };
```

Figure 4.18: Die Prozedur `atMostOne`.

Für die erste Spalte hätte diese Formel im Falle eine 8×8-Bretts die Form

$$\mathtt{Var}(1,1) \vee \mathtt{Var}(2,1) \vee \mathtt{Var}(3,1) \vee \mathtt{Var}(4,1) \vee \mathtt{Var}(5,1) \vee \mathtt{Var}(6,1) \vee \mathtt{Var}(7,1) \vee \mathtt{Var}(8,1)$$

und wenn allgemein eine Spalte c mit $c \in \{1, \cdots, 8\}$ gegeben ist, lautet die Formel

$$\mathtt{Var}(1,c) \vee \mathtt{Var}(2,c) \vee \mathtt{Var}(3,c) \vee \mathtt{Var}(4,c) \vee \mathtt{Var}(5,c) \vee \mathtt{Var}(6,c) \vee \mathtt{Var}(7,c) \vee \mathtt{Var}(8,c).$$

```
atMostOneInRow := procedure(row, n) {
    return atMostOne({ Var(row, j) : j in [1 .. n] });
};
```

Figure 4.19: Die Prozedur `atMostOneInRow`.

Schreiben wir diese Formel in der Mengenschreibweise als Menge von Klauseln, so erhalten wir

$$\{\{\mathtt{Var}(1,c), \mathtt{Var}(2,c), \mathtt{Var}(3,c), \mathtt{Var}(4,c), \mathtt{Var}(5,c), \mathtt{Var}(6,c), \mathtt{Var}(7,c), \mathtt{Var}(8,c)\}\}.$$

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.

```
oneInColumn := procedure(column, n) {
    return { { Var(row, column) : row in { 1 .. n } } };
};
```

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 Reihe mindestens eine Dame steht. Solche Formeln sind aber unnötig, denn wenn wir wissen, dass in jeder Spalte mindestens eine Dame steht, so wissen wir bereits, dass auf dem Brett mindestens 8 Damen stehen. Wenn wir nun zusätzlich wissen, dass in jeder Zeile höchstens eine Dame steht, so ist automatisch klar, dass in jeder Zeile genau eine Dame stehen muss, denn sonst kommen wir insgesamt nicht auf 8 Damen. Weiter folgt aus der Tatsache, dass in jeder Spalte eine Dame steht und daraus, dass es insgesamt nicht mehr als 8 Damen sind, dass in jeder Spalte höchstens eine Dame stehen kann.

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

$$\mathtt{Var}(8,1),\ \mathtt{Var}(7,2),\ \mathtt{Var}(6,3),\ \mathtt{Var}(5,4),\ \mathtt{Var}(4,5),\ \mathtt{Var}(3,6),\ \mathtt{Var}(2,7),\ \mathtt{Var}(1,8).$$

Die Indizes i und j der Variablen $\mathtt{Var}(i,j)$ 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 $\{3, \cdots, 15\}$ annimmt. Diesen Wert k geben wir nun als Argument bei der Prozedur `atMostOneInUpperDiagonal` mit. Diese Prozedur ist in Abbildung 4.21 gezeigt.

```
atMostOneInUpperDiagonal := procedure(k, n) {
    s := { Var(r, c) : c in [1..n], r in [1..n] | r + c == k };
    return atMostOne(s);
};
```

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

$$\texttt{Var}(1,1),\ \texttt{Var}(2,2),\ \texttt{Var}(3,3),\ \texttt{Var}(4,4),\ \texttt{Var}(5,5),\ \texttt{Var}(6,6),\ \texttt{Var}(7,7),\ \texttt{Var}(8,8)$$

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 $\{-6, \cdots, 6\}$ 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

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

rechnet für ein Schach-Brett der Größe n eine Menge von Klauseln aus, die genau dann erfüllt sind, wenn auf dem Schach-Brett

1. in jeder 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 ausdruckt. 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`.

```
solve := procedure(n) {
    clauses  := allClauses(n);
    solution := davisPutnam(clauses, {});
    if (solution != { {} }) {
        printBoard(solution, n);
    } else {
        print("The problem is not solvable for " + n + " queens!");
        print("Try to increase the number of queens.");
    }
};
```

Figure 4.24: Die Prozedur `solve`.

```
printBoard := procedure(i, n) {
    if (i == { {} }) {
        return;
    }
    print( "        " + ((8*n+1) * "-") );
    for (row in [1..n]) {
        line := "        |";
        for (col in [1..n]) {
            line += "       |";
        }
        print(line);
        line := "        |";
        for (col in [1..n]) {
            if ({ Var(row, col) } in i) {
                line += "   Q   |";
            } else {
                line += "       |";
            }
        }
        print(line);
        line := "        |";
        for (col in [1..n]) {
            line += "       |";
        }
        print(line);
        print( "        " + ((8*n+1) * "-") );
    }
};
```

Figure 4.25: Die Prozedur `printBoard`().

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

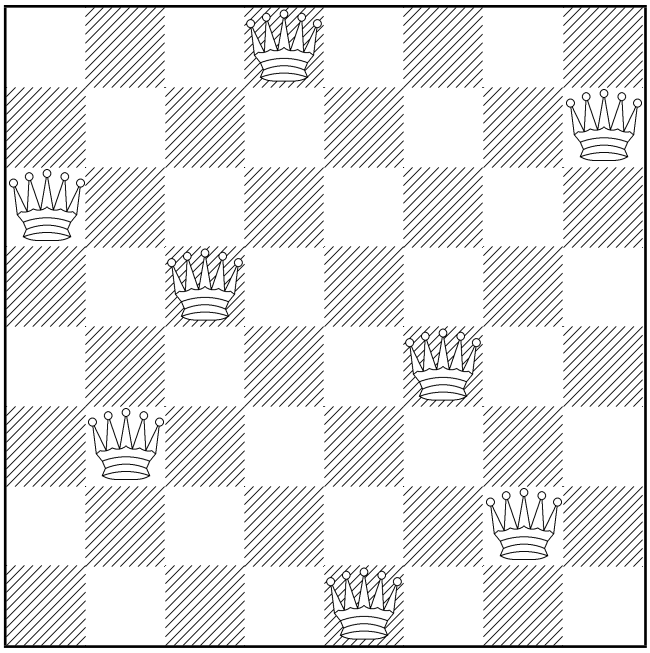

Figure 4.26: Eine Lösung des 8-Damen-Problems.

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 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.

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:
   $$\mathit{vater}(x), \quad \mathit{mutter}(\mathit{isaac}), \quad x + 7, \quad \cdots.$$
3. Verschiedene Objekte werden durch *Prädikats-Zeichen* in Relation gesetzt:
   $$\mathit{istBruder}\big(\mathit{albert}, \mathit{vater}(\mathit{bruno})\big), \quad x + 7 < x \cdot 7, \quad n \in \mathbb{N}, \quad \cdots.$$
   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*

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

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:*

   $$\mathit{arity} : \mathcal{F} \cup \mathcal{P} \to \mathbb{N}_0.$$

   Wir sagen, dass das Funktions- oder Prädikats-Zeichen f ein n-stelliges Zeichen ist, falls $\mathit{arity}(f) = n$ gilt.
5. *Da wir in der Lage sein müssen, Variablen, Funktions- und Prädikats-Zeichen unterscheiden zu können, vereinbaren wir, dass die Mengen $\mathcal{V}$, $\mathcal{F}$ und $\mathcal{P}$ paarweise disjunkt sein müssen:*

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

Als Bezeichner für Objekte verwenden wir Ausdrücke, die aus Variablen und Funktions-Zeichen aufgebaut sind. Solche Ausdrücke nennen wir *Terme*. Formal werden diese wie folgt definiert.

Definition 27 (Terme, $\mathcal{T}_\Sigma$) *Ist $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle$ eine Signatur, so definieren wir die Menge der Σ-Terme $\mathcal{T}_\Sigma$ induktiv:*

1. *Für jede Variable $x \in \mathcal{V}$ gilt $x \in \mathcal{T}_\Sigma$.*
2. *Ist $f \in \mathcal{F}$ ein n-stelliges Funktions-Zeichen und sind $t_1, \cdots, t_n \in \mathcal{T}_\Sigma$, so gilt auch*

   $$f(t_1, \cdots, t_n) \in \mathcal{T}_\Sigma.$$

   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. ◇

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. $\mathit{arity} := \big\{\langle 0, 0\rangle, \langle 1, 0\rangle, \langle +, 2\rangle, \langle -, 2\rangle, \langle *, 2\rangle, \langle =, 2\rangle, \langle \leq, 2\rangle\big\}$,

   gibt die Stelligkeit der Funktions- und Prädikats-Zeichen an und
5. $\Sigma_{\text{arith}} := \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle$ 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. ◇

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, $\mathcal{A}_\Sigma$) *Gegeben sei eine Signatur $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity}\rangle$. Die Menge der atomaren Σ-Formeln $\mathcal{A}_\Sigma$ wird wie folgt definiert: Ist $p \in \mathcal{P}$ ein n-stelliges Prädikats-Zeichen und sind n Σ-Terme t_1, $\cdots$, t_n gegeben, so ist $p(t_1, \cdots, t_n)$ eine* atomaren Σ-Formel *:*

$$p(t_1, \cdots, t_n) \in \mathcal{A}_\Sigma.$$

Falls p ein 0-stelliges Prädikats-Zeichen ist, dann schreiben wir auch p anstelle von $p()$. In diesem Fall nennen wir p eine Aussage-Variable *.* ◇

Beispiel: Setzen wir das letzte Beispiel fort, so können wir sehen, dass

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

eine atomare Σ_{arith}-Formel ist. Beachten Sie, dass wir bisher noch nichts über den Wahrheitswert von solchen Formeln ausgesagt haben. Die Frage, wann eine Formel als wahr oder falsch gelten soll, wird erst im nächsten Abschnitt untersucht. ◇

Bei der Definition der prädikatenlogischen Formeln ist es notwendig, zwischen sogenannten *gebundenen* und *freien* Variablen zu unterscheiden. Wir führen diese Begriffe zunächst informal mit Hilfe eines Beispiels aus der Analysis ein. Wir betrachten die folgende 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 "$\int \cdot \, \mathrm{d}t$" in der Analysis Variablen bindet. Die oben gemachten Ausführungen zeigen, dass es zwei verschiedene Arten von Variable gibt: *freie Variablen* und *gebundene Variablen*. Um diese Begriffe präzisieren zu können, definieren wir zunächst für einen Σ-Term t die Menge der in t enthaltenen Variablen.

Definition 29 (*Var*(t)) *Ist t ein Σ-Term, mit $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle$, so definieren wir die Menge Var(t) der Variablen, die in t auftreten, durch Induktion nach dem Aufbau des Terms:*

1. *Var$(x) := \{x\}$ für alle $x \in \mathcal{V}$,*
2. *Var$\big(f(t_1, \cdots, t_n)\big) :=$ Var$(t_1) \cup \cdots \cup$ Var(t_n).* ◇

Definition 30 (Σ-Formel, $\mathbb{F}_\Sigma$, gebundene und freie Variablen, *BV*(F), *FV*(F))
Es sei $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, arity \rangle$ eine Signatur. Die Menge der Σ-Formeln bezeichnen wir mit $\mathbb{F}_\Sigma$. Wir definieren diese Menge induktiv. Gleichzeitig definieren wir für jede Formel $F \in \mathbb{F}_\Sigma$ die Menge BV(F) der in F gebunden auftretenden Variablen und die Menge FV(F) der in F frei auftretenden Variablen.

1. *Es gilt $\bot \in \mathbb{F}_\Sigma$ und $\top \in \mathbb{F}_\Sigma$ und wir definieren*
   $$FV(\bot) := FV(\top) := BV(\bot) := BV(\top) := \{\}.$$
2. *Ist $F = p(t_1, \cdots, t_n)$ eine atomare Σ-Formel, so gilt $F \in \mathbb{F}_\Sigma$. Weiter definieren wir:*
   (a) *FV$\big(p(t_1, \cdots, t_n)\big) :=$ Var$(t_1) \cup \cdots \cup$ Var(t_n).*
   (b) *BV$\big(p(t_1, \cdots, t_n)\big) := \{\}$.*
3. *Ist $F \in \mathbb{F}_\Sigma$, so gilt $\neg F \in \mathbb{F}_\Sigma$. Weiter definieren wir:*
   (a) *FV$\big(\neg F\big) :=$ FV(F).*
   (b) *BV$\big(\neg F\big) :=$ BV(F).*
4. *Sind $F, G \in \mathbb{F}_\Sigma$ und gilt außerdem*
   $$\big(FV(F) \cup FV(G)\big) \cap \big(BV(F) \cup BV(G)\big) = \{\},$$
   so gilt auch
   (a) *$(F \wedge G) \in \mathbb{F}_\Sigma$,*
   (b) *$(F \vee G) \in \mathbb{F}_\Sigma$,*
   (c) *$(F \to G) \in \mathbb{F}_\Sigma$,*
   (d) *$(F \leftrightarrow G) \in \mathbb{F}_\Sigma$.*

   Weiter definieren wir für alle Junktoren $\odot \in \{\wedge, \vee, \to, \leftrightarrow\}$:
   (a) *FV$\big(F \odot G\big) :=$ FV$(F) \cup$ FV(G).*
   (b) *BV$\big(F \odot G\big) :=$ BV$(F) \cup$ BV(G).*
5. *Sei $x \in \mathcal{V}$ und $F \in \mathbb{F}_\Sigma$ mit $x \notin$ BV(F). Dann gilt:*
   (a) *$(\forall x\colon F) \in \mathbb{F}_\Sigma$.*
   (b) *$(\exists x\colon F) \in \mathbb{F}_\Sigma$.*

   Weiter definieren wir
   (a) *FV$\big((\forall x\colon F)\big) :=$ FV$\big((\exists x\colon F)\big) :=$ FV$(F) \backslash \{x\}$.*
   (b) *BV$\big((\forall x\colon F)\big) :=$ BV$\big((\exists x\colon F)\big) :=$ BV$(F) \cup \{x\}$.*

Ist die Signatur Σ aus dem Zusammenhang klar oder aber unwichtig, so schreiben wir auch $\mathbb{F}$ statt $\mathbb{F}_\Sigma$ und sprechen dann einfach von Formeln statt von Σ-Formeln. ◇

Bei der oben gegebenen Definition haben wir darauf geachtet, dass eine Variable nicht gleichzeitig frei und gebunden in einer Formel auftreten kann, denn durch eine leichte Induktion nach dem Aufbau der Formeln lässt sich zeigen, dass für alle $F \in \mathbb{F}_\Sigma$ folgendes gilt:

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

Beispiel: Setzen wir das oben begonnene Beispiel fort, so sehen wir, dass

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

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

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

Darüber hinaus legen wir fest, dass Quantoren stärker binden als die aussagenlogischen Junktoren. Damit können wir

$$\forall x\colon p(x) \wedge G \quad \text{statt} \quad \big(\forall x\colon p(x)\big) \wedge G$$

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

$$\mathtt{n}_1 = \mathtt{n}_2 \quad \text{anstelle von} \quad =(\mathtt{n}_1, \mathtt{n}_2).$$

Die Formel $(\exists x\colon \leq(+(y,x),y))$ wird dann lesbarer als

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

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

$$\big(\forall x \in M : F\big) \overset{\text{def}}{\Longleftrightarrow} \forall x : \big(x \in M \rightarrow F\big), \quad \text{und} \quad \big(\exists x \in M : F\big) \overset{\texttt{def}}{\Longleftrightarrow} \exists x : \big(x \in M \wedge F\big).$$

definiert sind.

5.2 Semantik der Prädikatenlogik

Als nächstes legen wir die Bedeutung der Formeln fest. Dazu definieren wir den Begriff einer 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*

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

gegeben. Eine Σ-Struktur *$\mathcal{S}$ ist ein Paar $\langle \mathcal{U}, \mathcal{J} \rangle$, so dass folgendes gilt:*

1. *$\mathcal{U}$ ist eine nicht-leere Menge. Diese Menge nennen wir auch das* Universum *der Σ-Struktur. Dieses Universum enthält die Werte, die sich später bei der Auswertung der Terme ergeben werden.*
2. *$\mathcal{J}$ ist die* Interpretation *der Funktions– und Prädikats-Zeichen. Formal definieren wir $\mathcal{J}$ als eine Abbildung mit folgenden Eigenschaften:*
   (a) *Jedem Funktions-Zeichen $f \in \mathcal{F}$ mit $\mathit{arity}(f) = m$ wird eine m-stellige Funktion*
   $$f^{\mathcal{J}}\colon \mathcal{U} \times \cdots \times \mathcal{U} \rightarrow \mathcal{U}$$
   zugeordnet, die m-Tupel des Universums $\mathcal{U}$ in das Universum $\mathcal{U}$ abbildet.

(b) Jedem Prädikats-Zeichen $p \in \mathcal{P}$ mit arity$(p) = n$ wird eine n-stellige Funktion

$$p^{\mathcal{J}} : \mathcal{U} \times \cdots \times \mathcal{U} \to \mathbb{B}$$

zugeordnet, die jedem n-Tupel des Universums $\mathcal{U}$ einen Wahrheitswert aus der Menge $\mathbb{B} = \{\mathtt{true}, \mathtt{false}\}$ zuordnet.

(c) Ist das Zeichen "=" ein Element der Menge der Prädikats-Zeichen $\mathcal{P}$, so gilt

$$=^{\mathcal{J}} (u, v) = \mathtt{true} \quad \textit{g.d.w.} \quad u = v,$$

das Gleichheits-Zeichen wird also durch die identische Relation $id_{\mathcal{U}}$ interpretiert. ◇

Beispiel: Wir geben ein Beispiel für eine Σ_{arith}-Struktur $\mathcal{S}_{\text{arith}} = \langle \mathcal{U}_{\text{arith}}, \mathcal{J}_{\text{arith}} \rangle$, indem wir definieren:

1. $\mathcal{U}_{\text{arith}} = \mathbb{N}$.
2. Die Abbildung $\mathcal{J}_{\text{arith}}$ legen wir dadurch fest, dass die Funktions-Zeichen 0, 1, $+$, $-$, $*$ durch die entsprechend benannten Funktionen auf der Menge $\mathbb{N}$ der natürlichen Zahlen zu interpretieren sind.

   Ebenso sollen die Prädikats-Zeichen $=$ und $\leq$ durch die Gleichheits-Relation und die Kleiner-Gleich-Relation interpretiert werden. ◇

Beispiel: Wir geben ein weiteres Beispiel. Die Signatur Σ_G der Gruppen-Theorie sei definiert als

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

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

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. $*^{\mathcal{J}} := \Big\{\big\langle\langle a, a\rangle, a\big\rangle, \big\langle\langle a, b\rangle, b\big\rangle, \big\langle\langle b, a\rangle, b\big\rangle, \big\langle\langle b, b\rangle, a\big\rangle\Big\}$
3. $=^{\mathcal{J}}$ ist die Identität:

   $=^{\mathcal{J}} := \Big\{\big\langle\langle a, a\rangle, \mathtt{true}\big\rangle, \big\langle\langle a, b\rangle, \mathtt{false}\big\rangle, \big\langle\langle b, a\rangle, \mathtt{false}\big\rangle, \big\langle\langle b, b\rangle, \mathtt{true}\big\rangle\Big\}$

   Beachten Sie, dass wir bei der Interpretation des Gleichheits-Zeichens keinen Spielraum haben! ◇

Falls wir Terme auswerten wollen, die Variablen enthalten, so müssen wir für diese Variablen irgendwelche Werte aus dem Universum einsetzen. Welche Werte wir einsetzen, kann durch eine *Variablen-Belegung* festgelegt werden. Diesen Begriff definieren wir nun.

Definition 32 (Variablen-Belegung) *Es sei eine Signatur*

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

gegeben. Weiter sei $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ eine Σ-Struktur. Dann bezeichnen wir eine Abbildung

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

als eine $\mathcal{S}$-Variablen-Belegung.

Ist $\mathcal{I}$ eine $\mathcal{S}$-Variablen-Belegung, $x \in \mathcal{V}$ und $c \in \mathcal{U}$, so bezeichnet $\mathcal{I}[x/c]$ die Variablen-Belegung, die der Variablen x den Wert c zuordnet und die ansonsten mit $\mathcal{I}$ übereinstimmt:

$$\mathcal{I}[x/c](y) := \begin{cases} c & \textit{falls } y = x; \\ \mathcal{I}(y) & \textit{sonst.} \end{cases}$$ ◇

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 $f(t_1, \cdots, t_n)$ definieren wir*
$$\mathcal{S}\big(\mathcal{I}, f(t_1, \cdots, t_n)\big) := f^{\mathcal{J}}\big(\mathcal{S}(\mathcal{I}, t_1), \cdots, \mathcal{S}(\mathcal{I}, t_n)\big).$$ ◇

Beispiel: Mit der oben definieren Σ_{arith}-Struktur $\mathcal{S}_{\text{arith}}$ definieren wir eine $\mathcal{S}_{\text{arith}}$-Variablen-Belegung $\mathcal{I}$ durch

$$\mathcal{I} := \big\{\langle x, 0\rangle, \langle y, 7\rangle, \langle z, 42\rangle\big\},$$

es gilt also

$$\mathcal{I}(\mathtt{x}) := 0, \quad \mathcal{I}(\mathtt{y}) := 7, \quad \text{und} \quad \mathcal{I}(\mathtt{z}) := 42.$$

Dann gilt offenbar

$$\mathcal{S}\big(\mathcal{I}, \mathtt{x} + \mathtt{y}\big) = 7.$$

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 $p(t_1, \cdots, t_n)$ den Wert $\mathcal{S}\big(\mathcal{I}, p(t_1, \cdots, t_n)\big)$ wie folgt:*

$$\mathcal{S}\big(\mathcal{I}, p(t_1, \cdots, t_n)\big) := p^{\mathcal{J}}\big(\mathcal{S}(\mathcal{I}, t_1), \cdots, \mathcal{S}(\mathcal{I}, t_n)\big).$$ ◇

Beispiel: In Fortführung des obigen Beispiels gilt:

$$\mathcal{S}\big(\mathcal{I}, z \leq x + y\big) = \mathtt{false}.$$ ◇

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

Die Semantik dieser Funktionen hatten wir durch die Tabelle in Abbildung 4.1 auf Seite 53 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. $\mathcal{S}(\mathcal{I}, \top) := \mathtt{true}$ *und* $\mathcal{S}(\mathcal{I}, \bot) := \mathtt{false}$.
2. $\mathcal{S}(\mathcal{I}, \neg F) := \ominus\big(\mathcal{S}(\mathcal{I}, F)\big)$.
3. $\mathcal{S}(\mathcal{I}, F \wedge G) := \otimes\big(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G)\big)$.
4. $\mathcal{S}(\mathcal{I}, F \vee G) := \oslash\big(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G)\big)$.
5. $\mathcal{S}(\mathcal{I}, F \to G) := \oplus\big(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G)\big)$.
6. $\mathcal{S}(\mathcal{I}, F \leftrightarrow G) := \odot\big(\mathcal{S}(\mathcal{I}, F), \mathcal{S}(\mathcal{I}, G)\big)$.
7. $\mathcal{S}\big(\mathcal{I}, \forall x\colon F\big) := \begin{cases} \mathtt{true} & \text{falls } \mathcal{S}(\mathcal{I}[x/c], F) = \mathtt{true} \quad \text{für alle } c \in \mathcal{U} \text{ gilt;} \\ \mathtt{false} & \text{sonst.} \end{cases}$

8. $\mathcal{S}\big(\mathcal{I}, \exists x\colon F\big) := \begin{cases} \texttt{true} & \textit{falls } \mathcal{S}(\mathcal{I}[x/c], F) = \texttt{true} \quad \textit{für ein } c \in \mathcal{U} \textit{ gilt;} \\ \texttt{false} & \textit{sonst.} \end{cases}$ ◇

Beispiel: In Fortführung des obigen Beispiels gilt

$$\mathcal{S}\big(\mathcal{I}, \forall \mathtt{x} : x * 0 < 1\big) = \texttt{true}.$$ ◇

Definition 36 (Allgemeingültig) *Ist F eine Σ-Formel, so dass für jede Σ-Struktur $\mathcal{S}$ und für jede $\mathcal{S}$-Variablen-Belegung $\mathcal{I}$*

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

gilt, so bezeichnen wir F als allgemeingültig *. In diesem Fall schreiben wir*

$$\models F.$$ ◇

Ist F eine Formel für die $FV(F) = \{\}$ ist, dann hängt der Wert $\mathcal{S}(\mathcal{I}, F)$ offenbar gar nicht von der Interpretation $\mathcal{I}$ ab. Solche Formeln bezeichnen wir auch als *geschlossene* Formeln. In diesem Fall schreiben wir kürzer $\mathcal{S}(F)$ an Stelle von $\mathcal{S}(\mathcal{I}, F)$. Gilt dann zusätzlich $\mathcal{S}(F) = \texttt{true}$, so sagen wir auch dass $\mathcal{S}$ ein *Modell* von F ist. Wir schreiben dann

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

Die Definition der Begriffe "*erfüllbar*" und "*äquivalent*" lassen sich nun aus der Aussagenlogik übertragen. Um unnötigen Ballast in den Definitionen zu vermeiden, nehmen wir im Folgenden immer eine feste Signatur Σ als gegeben an. Dadurch können wir in den folgenden Definitionen von Termen, Formeln, Strukturen, etc. sprechen und meinen damit Σ-Terme, Σ-Formeln und Σ-Strukturen.

Definition 37 (äquivalent) *Zwei Formeln F und G heißen* äquivalent *g.d.w. gilt*

$$\models F \leftrightarrow G.$$ ◇

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*

$$\forall m \in M : \mathcal{S}(\mathcal{I}, m) = \texttt{true}$$

gilt. Andernfalls heißt M unerfüllbar *oder auch* widersprüchlich *. Wir schreiben dafür auch*

$$M \models \bot$$ ◇

Unser Ziel ist es, ein Verfahren anzugeben, mit dem wir in der Lage sind zu überprüfen, ob eine Menge M von Formeln *widersprüchlich* ist, ob also $M \models \bot$ gilt. Es zeigt sich, dass dies im Allgemeinen nicht möglich ist, die Frage, ob $M \models \bot$ gilt, ist unentscheidbar. Ein Beweis dieser Tatsache geht allerdings über den Rahmen dieser Vorlesung heraus. Dem gegenüber ist es möglich, ähnlich wie in der Aussagenlogik einen *Kalkül* $\vdash$ anzugeben, so dass gilt

$$M \vdash \bot \quad \text{g.d.w.} \quad M \models \bot.$$

Ein solcher Kalkül kann dann zur Implementierung eines *Semi-Entscheidungs-Verfahrens* benutzt werden: Um zu überprüfen, ob $M \models \bot$ gilt, versuchen wir, aus der Menge M die Formel $\bot$ herzuleiten. Falls wir dabei systematisch vorgehen, indem wir alle möglichen Beweise durchprobieren, so werden wir, falls tatsächlich $M \models \bot$ gilt, auch irgendwann einen Beweis finden, der $M \vdash \bot$ zeigt. Wenn allerdings der Fall

$$M \not\models \bot$$

vorliegt, so werden wir dies im allgemeinen nicht feststellen können, denn die Menge aller Beweise ist unendlich 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 *perfekt*, 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 103 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.

In den nächsten Abschnitten gehen wir daran, den oben erwähnten Kalkül $\vdash$ zu definieren. Es zeigt sich, dass die Arbeit wesentlich einfacher wird, wenn wir uns auf bestimmte Formeln, sogenannte *Klauseln*, beschränken. Wir zeigen daher zunächst im nächsten 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.

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

$$\Sigma_G = \big\langle \{x, y, z\},\ \{1, *\},\ \{=\},\ \{\langle 1, 0\rangle, \langle *, 2\rangle, \langle =, 2\rangle\}\big\rangle$$

gegeben worden. Hierbei ist also "1" ein 0-stelliges Funktions-Zeichen, "$*$" ist eine 2-stelliges 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.

```
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, y ] : x in u, y in u | x == y };
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

2. Das in Zeile 3 definierte Universum u besteht aus den beiden Strings "a" und "b".

3. In Zeile 4 definieren wir eine Funktion product als binäre Relation. Für die so definierte Funktion gilt

   $$\texttt{product}(\langle\texttt{"a"},\texttt{"a"}\rangle) = \texttt{"a"}, \quad \texttt{product}(\langle\texttt{"a"},\texttt{"b"}\rangle) = \texttt{"b"},$$
   $$\texttt{product}(\langle\texttt{"b"},\texttt{"a"}\rangle) = \texttt{"b"}, \quad \texttt{product}(\langle\texttt{"b"},\texttt{"b"}\rangle) = \texttt{"a"}.$$

   Diese Funktion verwenden wir später als die Interpretation $*^{\mathcal{J}}$ des Funktions-Zeichens "$*$".

4. Ebenso haben wir in Zeile 5 die Interpretation $=^{\mathcal{J}}$ des Prädikats-Zeichens "$=$" als die binäre Relation equals 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 werden, stellen wir den Operator "*" durch das Funktions-Zeichen "^product" dar und das Prädikats-Zeichen "=" wird durch das Zeichen "^equals" dargestellt, denn dies sind die Namen, die von SetlX intern benutzt werden. Das neutrale Element "1" stellen wir durch das Funktions-Zeichen "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 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 $\mathcal{S}$ und eine Variablen-Belegung $\mathcal{I}$ übergeben werden. Die Formel wird dabei als Term dargestellt, ganz ähnlich, wie wir das bei der Implementierung der Aussagenlogik schon praktiziert haben. Beispielsweise können wir die Formel

$$\forall x : \forall y : x * y = y * x$$

durch den Term

```
^forall(^variable("x"), ^forall(^variable("y"),
     ^equals(^product(^variable("x"), ^variable("y")),
            ^product(^variable("y"), ^variable("x"))
            )))
```

darstellen und dass ist im Wesentlichen auch die Struktur, die erzeugt wird, wenn wir den String

"forall (x in u | exists (y in u | x * y == E()))"

```
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 in _ | g) :
12              return forall (c in u | evalFormula(g, s, modify(i, x, c)));
13         case exists (x in _ | 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

mit Hilfe der in SetlX vordefinierten Funktion `parse` in einen Term umwandeln.

Bemerkung: An dieser Stelle wundern Sie sich vermutlich, warum wir oben "`x in _`" und "`y in _`" schreiben, denn wir wollen die Variablen `x` und `y` eigentlich ja gar nicht einschränken. Der Grund ist, dass die Syntax von SetlX nur solche Quantoren erlaubt, in denen die Variablen auf eine Menge eingeschränkt sind. Daher sind wir gezwungen, bei der Verwendung von Quantoren die Variablen syntaktisch einzuschränken. Wir schreiben deswegen

`forall (x in _ | `g`)` bzw. `exists (x in _ | `g`)`

an Stelle von

`forall (x | `g`)` bzw. `exists (x | `g`)`.

Da wir x hier nicht wirklich einschränken wollen, schreiben wir "`x in _`" und benutzen "`_`" als sogenannte *anonyme* Variable. ◇

Die Auswertung einer prädikatenlogischen Formel ist nun analog zu der in Abbildung 4.1 auf Seite 57 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 $\forall y \in u : h$, so wird die Formel f durch den Term

$$f = \texttt{^forall}(y, u, h)$$

dargestellt. Das Muster

```
forall (x in _ | g)
```

bindet daher `x` an die tatsächlich auftretende Variable y und `g` an die Teilformel h. Die Auswertung von $\forall x : g$ geschieht nach der Formel

$$\mathcal{S}(\mathcal{I}, \forall x : g) := \begin{cases} \texttt{true} & \text{falls } \mathcal{S}(\mathcal{I}[x/c], g) = \texttt{true} \quad \text{für alle } c \in \mathcal{U} \text{ gilt;} \\ \texttt{false} & \text{sonst.} \end{cases}$$

Um die Auswertung implementieren zu können, verwenden wir eine Prozedur *modify*(), welche die Variablen-Belegung i an der Stelle x zu c abändert, es gilt also

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

Die Implementierung dieser Prozedur wird später in Abbildung 5.4 gezeigt. Bei der Auswertung eines All-Quantors können wir ausnutzen, dass die Sprache SetlX selber den Quantor `forall` 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.

```
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  };
9  evalTerm := procedure(t, s, i) {
10     if (fct(t) == "^variable") {
11         varName := args(t)[1];
12         return i[varName];
13     }
14     j       := s[2];
15     f       := fct(t); // function symbol
16     fJ      := j[f];
17     argList := args(t);
18     argsVal := evalTermList(argList, s, i);
19     if (#argsVal > 0) {
20         result := fJ[argsVal];
21     } else {
22         result := fJ;   // t is a constant
23     }
24     return result;
25 };
26 evalTermList := procedure(tl, s, i) {
27     return [ evalTerm(t, s, i) : t in tl ];
28 };
29 modify := procedure(i, v, c) {
30     x := args(v)[1];  // v = ^variable(x)
31     i[x] := c;
32     return i;
33 };
```

Figure 5.4: Auswertung von Termen und atomaren Formeln.

Abbildung 5.4 zeigt die Auswertung atomarer Formeln und prädikatenlogischer Terme. Um eine atomare Formel der Form

$$a = P(t_1, \cdots, t_n)$$

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 $t_1, \cdots, t_n$ aus und überprüfen dann, ob das Ergebnis dieser Auswertung tatsächlich ein Element der Menge pJ ist.

Die Prozedur `evalTerm()` arbeitet wie folgt: Das erste Argument t der Prozedur `evalTerm`$(t, \mathcal{S}, \mathcal{I})$ 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 12 einfach den Wert zurück, der in der Variablen-Belegung $\mathcal{I}$ für

diese Variable eingetragen ist. Die Variablen-Belegung wird dabei durch eine zweistellige Relation dargestellt, die wir als Funktion benutzen.

2. Falls der auszuwertende Term t die Form

   $$t = F(t_1, \cdots, t_n)$$

   hat, werden in Zeile 18 zunächst rekursiv die Subterme t_1, $\cdots$, t_n ausgewertet. Anschließend wird die Interpretation $F^{\mathcal{J}}$ des Funktions-Zeichens F herangezogen, um die Funktion $F^{\mathcal{J}}$ für die gegebenen Argumente auszuwerten, wobei in Zeile 20 der Fall betrachtet wird, dass tatsächlich Argumente vorhanden sind, während in Zeile 22 der Fall behandelt wird, dass es sich bei dem Funktions-Zeichen F um eine Konstante handelt, deren Wert dann unmittelbar durch $F^{\mathcal{J}}$ gegeben ist.

Die Implementierung der Prozedur `evalTermList()` wendet die Funktion `evalTerm()` auf alle Terme der gegebenen Liste an. 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  g1 := parse("forall (x in u | x * E() == x)");
2  g2 := parse("forall (x in u | exists (y in u | x * y == E()))");
3  g3 := parse("forall (x in u | forall (y in u | forall (z in u | (x*y)*z == x*(y*z) )))");
4  gt := { g1, g2, g3 };
5
6  print("checking group theory in the structure ", s);
7  for (f in gt) {
8      print( "checking ", f, ": ", evalFormula(f, s, i) );
9  }
```

Figure 5.5: Axiome der Gruppen-Theorie

Wir zeigen nun, wie sich die in Abbildung 5.3 gezeigte Funktion `evalFormula`$(f, \mathcal{S}, \mathcal{I})$ benutzen lässt um zu überprüfen, ob die in Abbildung 5.2 gezeigte Struktur die Axiome der *Gruppen-Theorie* erfüllt. Die Axiome der Gruppen-Theorie sind wie folgt:

1. Die Konstante 1 ist das rechts-neutrale Element der Multiplikation:

   $$\forall x \colon x * 1 = x.$$

2. Für jedes Element x gibt es ein rechts-inverses Element y, dass mit dem Element x multipliziert die 1 ergibt:

   $$\forall x \colon \exists y \colon x * y = 1.$$

3. Es gilt das Assoziativ-Gesetz:

   $$\forall x \colon \forall y \colon \forall z \colon (x * y) * z = x * (y * z).$$

Diese Axiome sind in den Zeilen 1 bis 3 der Abbildung 5.5 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.

Bemerkung: Mit dem oben vorgestellten Programm können wir überprüfen, ob eine prädikatenlogische Formel in einer vorgegebenen endlichen Struktur erfüllt ist. Wir können damit allerdings nicht überprüfen, ob eine Formel allgemeingültig ist, denn einerseits können wir das Programm nicht anwenden, wenn die Strukturen ein unendliches Universum haben, andererseits ist selbst die Zahl der verschiedenen endlichen Stukturen, die wir ausprobieren müssten, unendlich groß.

Aufgabe 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 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. ◇

5.3 Normalformen für prädikatenlogische Formeln

In diesem Abschnitt werden wir verschiedene Möglichkeiten zur Umformung prädikatenlogischer Formeln kennenlernen. Zunächst geben wir einige Äquivalenzen an, mit deren Hilfe Quantoren manipuliert werden können.

Satz 39 *Es gelten die folgenden Äquivalenzen:*

1. $\models \neg\big(\forall x\colon f\big) \leftrightarrow \big(\exists x\colon \neg f\big)$
2. $\models \neg\big(\exists x\colon f\big) \leftrightarrow \big(\forall x\colon \neg f\big)$
3. $\models \big(\forall x\colon f\big) \wedge \big(\forall x\colon g\big) \leftrightarrow \big(\forall x\colon f \wedge g\big)$
4. $\models \big(\exists x\colon f\big) \vee \big(\exists x\colon g\big) \leftrightarrow \big(\exists x\colon f \vee g\big)$
5. $\models \big(\forall x\colon \forall y\colon f\big) \leftrightarrow \big(\forall y\colon \forall x\colon f\big)$
6. $\models \big(\exists x\colon \exists y\colon f\big) \leftrightarrow \big(\exists y\colon \exists x\colon f\big)$
7. *Falls x eine Variable ist, für die $x \notin FV(f)$ ist, so haben wir*

   $$\models \big(\forall x\colon f\big) \leftrightarrow f \quad \textit{und} \quad \models \big(\exists x\colon f\big) \leftrightarrow f.$$

8. *Falls x eine Variable ist, für die $x \notin FV(g) \cup BV(g)$ gilt, so haben wir die folgenden Äquivalenzen:*
   - *(a)* $\models \big(\forall x\colon f\big) \vee g \leftrightarrow \forall x\colon (f \vee g)$
   - *(b)* $\models g \vee \big(\forall x\colon f\big) \leftrightarrow \forall x\colon (g \vee f)$
   - *(c)* $\models \big(\exists x\colon f\big) \wedge g \leftrightarrow \exists x\colon (f \wedge g)$
   - *(d)* $\models g \wedge \big(\exists x\colon f\big) \leftrightarrow \exists x\colon (g \wedge f)$

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

$$\big(\forall u : \exists v : p(u,v)\big)[u/z] = \forall z : \exists v : p(z,v)$$

Damit können wir eine letzte Äquivalenz angeben: Ist f eine prädikatenlogische Formel, ist $x \in BV(F)$ und ist y eine Variable, die in f nicht auftritt, so gilt

$$\models f \leftrightarrow f[x/y].$$

Mit Hilfe der oben stehenden Äquivalenzen können wir eine Formel so umformen, dass die Quantoren nur noch außen stehen. Eine solche Formel ist dann in *pränexer Normalform*. Wir führen das Verfahren an einem Beispiel vor: Wir zeigen, dass die Formel

$$\big(\forall x\colon p(x)\big) \rightarrow \big(\exists x\colon p(x)\big)$$

allgemeingültig ist:

$$\begin{array}{ll} & \big(\forall x\colon p(x)\big) \rightarrow \big(\exists x\colon p(x)\big) \\ \Leftrightarrow & \neg\big(\forall x\colon p(x)\big) \vee \big(\exists x\colon p(x)\big) \\ \Leftrightarrow & \big(\exists x\colon \neg p(x)\big) \vee \big(\exists x\colon p(x)\big) \\ \Leftrightarrow & \exists x\colon \big(\neg p(x) \vee p(x)\big) \\ \Leftrightarrow & \exists x\colon \top \\ \Leftrightarrow & \top \end{array}$$

In diesem Fall haben wir Glück gehabt, dass es uns gelungen ist, die Formel als Tautologie zu erkennen. Im Allgemeinen reichen die obigen Umformungen aber nicht aus, um prädikatenlogische Tautologien erkennen zu können. Um Formeln noch stärker normalisieren zu können, führen wir einen weiteren Äquivalenz-Begriff ein. Diesen Begriff wollen wir vorher durch ein Beispiel motivieren. Wir betrachten die beiden Formeln

$$f_1 = \forall x\colon \exists y\colon p(x,y) \quad \text{und} \quad f_2 = \forall x\colon p\big(x, s(x)\big).$$

Die beiden Formeln f_1 und f_2 sind nicht äquivalent, denn sie entstammen noch nicht einmal der gleichen Signatur: In der Formel f_2 wird das Funktions-Zeichen s verwendet, das in der Formel f_1 überhaupt nicht auftritt. Auch wenn die beiden Formeln f_1 und f_2 nicht äquivalent sind, so besteht zwischen ihnen doch die folgende Beziehung: Ist S_1 eine prädikatenlogische Struktur, in der die Formel f_1 gilt:

$$\mathcal{S}_1 \models f_1,$$

dann können wir diese Struktur zu einer Struktur S_2 erweitern, in der die Formel f_2 gilt:

$$\mathcal{S}_2 \models f_2.$$

Dazu muss lediglich die Interpretation des Funktions-Zeichens s so gewählt werden, dass für jedes x tatsächlich $p\big(x, s(x)\big)$ gilt. Dies ist möglich, denn die Formel f_1 sagt ja aus, dass wir 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 $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity}\rangle$ eine Signatur. Ferner sei f eine geschlossene Σ-Formel der Form*

$$f = \forall x_1, \cdots, x_n\colon \exists y\colon g(x_1, \cdots, x_n, y).$$

Dann wählen wir ein neues n-stelliges Funktions-Zeichen s, d.h. wir nehmen ein Zeichen s, dass in der Menge $\mathcal{F}$ nicht auftritt und erweitern die Signatur Σ zu der Signatur

$$\Sigma' := \Big\langle \mathcal{V}, \mathcal{F} \cup \{s\}, \mathcal{P}, \mathit{arity} \cup \big\{\langle s, n\rangle\big\}\Big\rangle,$$

in der wir s als neues n-stelliges Funktions-Zeichen deklarieren. Anschließend definieren wir die Σ'-Formel f' wie folgt:

$$f' := \mathit{Skolem}(f) := \forall x_1\colon \cdots \forall x_n\colon g\big(x_1, \cdots, x_n, s(x_1, \cdots, x_n)\big)$$

Wir lassen also den Existenz-Quantor $\exists y$ weg und ersetzen jedes Auftreten der Variable y durch den Term $s(x_1, \cdots, x_n)$. Wir sagen, dass die Formel f' aus der Formel f durch einen Skolemisierungs-Schritt *hervorgegangen ist.* $\diamond$

In welchem Sinne sind eine Formel f und eine Formel f', die aus f durch einen Skolemisierungs-Schritt hervorgegangen sind, äquivalent? Zur Beantwortung dieser Frage dient die folgende Definition.

Definition 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*

$$f \approx_e g.$$ $\diamond$

Satz 42 *Falls die Formel f' aus der Formel f durch einen Skolemisierungs-Schritt 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 pränexe Normalform. Anschließend können wir die Existenz-Quantoren der Reihe nach durch Skolemisierungs-Schritte 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 pränexe Normalform gebracht und anschließend skolemisiert, so hat das Ergebnis die Gestalt

$$\forall x_1, \cdots, x_n : g$$

und in der Formel g treten keine Quantoren mehr auf. Die Formel g wird auch als die *Matrix* der obigen Formel bezeichnet. Wir können nun g mit Hilfe der uns aus dem letzten Kapitel bekannten aussagenlogischen Äquivalenzen in konjunktive Normalform bringen. Wir haben dann eine Formel der Gestalt

$$\forall x_1, \cdots, x_n : (k_1 \wedge \cdots \wedge k_m).$$

Dabei sind die k_i Disjunktionen von *Literalen*. In der Prädikatenlogik ist ein *Literal* entweder eine atomare Formel oder die Negation einer atomaren Formel. Wenden wir hier die Äquivalenz $(\forall x\colon f_1 \wedge f_2) \leftrightarrow (\forall x\colon f_1) \wedge (\forall x\colon f_2)$ an, so können wir die All-Quantoren auf die einzelnen k_i verteilen und die resultierende Formel hat die Gestalt

$$\big(\forall x_1, \cdots, x_n : k_1\big) \wedge \cdots \wedge \big(\forall x_1, \cdots, x_n : k_m\big).$$

Ist eine Formel F in der obigen Gestalt, so sagen wir, dass F in *prädikatenlogischer Klausel-Normalform* ist und eine Formel der Gestalt

$$\forall x_1, \cdots, x_n : k,$$

bei der k eine Disjunktion prädikatenlogischer Literale ist, bezeichnen wir als *prädikatenlogische Klausel*. Ist M eine Menge von Formeln deren Erfüllbarkeit wir untersuchen wollen, so können wir nach dem bisher gezeigten M immer in eine Menge prädikatenlogischer Klauseln umformen. Da dann nur noch All-Quantoren vorkommen, können wir hier die Notation noch vereinfachen indem wir vereinbaren, dass alle Formeln implizit allquantifiziert sind, wir lassen also die All-Quantoren weg.

Wozu sind nun die Umformungen in Skolem-Normalform gut? Es geht darum, dass wir ein Verfahren entwickeln wollen, mit dem es möglich ist für eine prädikatenlogische Formel f zu zeigen, dass f allgemeingültig ist, dass also

$$\models f$$

gilt. Wir wissen, dass

$$\models f \quad \text{g.d.w.} \quad \{\neg f\} \models \bot$$

gilt, denn die Formel f ist genau dann allgemeingültig, wenn es keine Struktur gibt, in der die Formel $\neg f$ erfüllbar ist. Wir bilden daher zunächst $\neg f$ und formen $\neg f$ in prädikatenlogische Klausel-Normalform um. Wir erhalten Klauseln $k_1, \cdots, k_n$, so dass

$$\neg f \approx_e k_1 \wedge \cdots \wedge k_n$$

gilt. Anschließend versuchen wir, aus den Klauseln $k_1, \cdots, k_n$ eine Widerspruch herzuleiten:

$$\{k_1, \cdots, k_n\} \vdash \bot$$

Wenn dies gelingt wissen wir, dass die Menge $\{k_1, \cdots, k_n\}$ unerfüllbar ist. Dann ist auch $\neg f$ unerfüllbar und damit ist f allgemeingültig. Damit wir aus den Klauseln $k_1, \cdots, k_n$ einen Widerspruch herleiten können, brauchen wir natürlich noch einen Kalkül, der mit prädikatenlogischen Klauseln arbeitet. Einen solchen Kalkül werden wir am Ende dieses Kapitel vorstellen.

Um das Verfahren näher zu erläutern demonstrieren wir es an einem Beispiel. Wir wollen untersuchen, ob

$$\models \big(\exists x\colon \forall y\colon p(x,y)\big) \rightarrow \big(\forall y\colon \exists x\colon p(x,y)\big)$$

gilt. Wir wissen, dass dies äquivalent dazu ist, dass

$$\Big\{\neg\Big(\big(\exists x\colon \forall y\colon p(x,y)\big) \rightarrow \big(\forall y\colon \exists x\colon p(x,y)\big)\Big)\Big\} \models \bot$$

gilt. Wir bringen zunächst die negierte Formel in pränexe Normalform.

$$\begin{array}{ll} & \neg\Big(\big(\exists x\colon \forall y\colon p(x,y)\big) \rightarrow \big(\forall y\colon \exists x\colon p(x,y)\big)\Big) \\ \leftrightarrow & \neg\Big(\neg\big(\exists x\colon \forall y\colon p(x,y)\big) \vee \big(\forall y\colon \exists x\colon p(x,y)\big)\Big) \\ \leftrightarrow & \big(\exists x\colon \forall y\colon p(x,y)\big) \wedge \neg\big(\forall y\colon \exists x\colon p(x,y)\big) \\ \leftrightarrow & \big(\exists x\colon \forall y\colon p(x,y)\big) \wedge \big(\exists y\colon \neg\exists x\colon p(x,y)\big) \\ \leftrightarrow & \big(\exists x\colon \forall y\colon p(x,y)\big) \wedge \big(\exists y\colon \forall x\colon \neg p(x,y)\big) \end{array}$$

Um an dieser Stelle weitermachen zu können, ist es nötig, die Variablen in dem zweiten Glied der Konjunktion umzubenennen. Wir ersetzen x durch u und y durch v und erhalten

$$\begin{array}{ll} & \big(\exists x\colon \forall y\colon p(x,y)\big) \wedge \big(\exists y\colon \forall x\colon \neg p(x,y)\big) \\ \leftrightarrow & \big(\exists x\colon \forall y\colon p(x,y)\big) \wedge \big(\exists v\colon \forall u\colon \neg p(u,v)\big) \\ \leftrightarrow & \exists v\colon \Big(\big(\exists x\colon \forall y\colon p(x,y)\big) \wedge \big(\forall u\colon \neg p(u,v)\big)\Big) \\ \leftrightarrow & \exists v\colon \exists x\colon \Big(\big(\forall y\colon p(x,y)\big) \wedge \big(\forall u\colon \neg p(u,v)\big)\Big) \\ \leftrightarrow & \exists v\colon \exists x\colon \forall y\colon \Big(p(x,y) \wedge \big(\forall u\colon \neg p(u,v)\big)\Big) \\ \leftrightarrow & \exists v\colon \exists x\colon \forall y\colon \forall u\colon \Big(p(x,y) \wedge \neg p(u,v)\Big) \end{array}$$

An dieser Stelle müssen wir skolemisieren um die Existenz-Quantoren los zu werden. Wir führen dazu zwei neue Funktions-Zeichen s_1 und s_2 ein. Dabei gilt $\texttt{arity}(s_1) = 0$ und $\texttt{arity}(s_2) = 0$, denn vor den Existenz-Quantoren stehen keine All-Quantoren.

$$\begin{array}{ll} & \exists v\colon \exists x\colon \forall y\colon \forall u\colon \Big(p(x,y) \wedge \neg p(u,v)\Big) \\ \approx_e & \exists x\colon \forall y\colon \forall u\colon \Big(p(x,y) \wedge \neg p(u,s_1)\Big) \\ \approx_e & \forall y\colon \forall u\colon \Big(p(s_2,y) \wedge \neg p(u,s_1)\Big) \end{array}$$

Da jetzt nur noch All-Quantoren auftreten, können wir diese auch noch weglassen, da wir ja vereinbart haben, dass alle freien Variablen implizit allquantifiziert sind. Damit können wir nun die prädikatenlogische Klausel-Normalform angeben, diese ist

$$M := \Big\{\big\{p(s_2,y)\big\}, \big\{\neg p(u,s_1)\big\}\Big\}.$$

Wir zeigen, dass die Menge M widersprüchlich ist. Dazu betrachten wir zunächst die Klausel $\big\{p(s_2,y)\big\}$ und setzen in dieser Klausel für y die Konstante s_1 ein. Damit erhalten wir die Klausel

$$\big\{p(s_2,s_1)\big\}. \tag{1}$$

Das Ersetzung von y durch s_1 begründen wir damit, dass die obige Klausel ja implizit allquantifiziert ist und wenn etwas für alle y gilt, dann sicher auch für $y = s_1$.

Als nächstes betrachten wir die Klausel $\{\neg p(u, s_1)\}$. Hier setzen wir für die Variablen u die Konstante s_2 ein und erhalten dann die Klausel

$$\{\neg p(s_2, s_1)\} \tag{2}$$

Nun wenden wir auf die Klauseln (1) und (2) die Schnitt-Regel an und finden

$$\{p(s_2, s_1)\}, \quad \{\neg p(s_2, s_1)\} \quad \vdash \quad \{\}.$$

Damit haben wir einen Widerspruch hergeleitet und gezeigt, dass die Menge M unerfüllbar ist. Damit ist dann auch

$$\Big\{\neg\Big(\big(\exists x\colon \forall y\colon p(x,y)\big) \to \big(\forall y\colon \exists x\colon p(x,y)\big)\Big)\Big\}$$

unerfüllbar und folglich gilt

$$\models \big(\exists x\colon \forall y\colon p(x,y)\big) \to \big(\forall y\colon \exists x\colon p(x,y)\big).$$

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*

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

gegeben. Eine Σ-Substitution ist eine endliche Menge von Paaren der Form

$$\sigma = \big\{\langle x_1, t_1\rangle, \cdots, \langle x_n, t_n\rangle\big\}.$$

Dabei gilt:

1. $x_i \in \mathcal{V}$, *die x_i sind also Variablen.*
2. $t_i \in \mathcal{T}_\Sigma$, *die t_i sind also Terme.*
3. *Für $i \neq j$ ist $x_i \neq x_j$, die Variablen sind also paarweise verschieden.*

Ist $\sigma = \big\{\langle x_1, t_1\rangle, \cdots, \langle x_n, t_n\rangle\big\}$ eine Σ-Substitution, so schreiben wir

$$\sigma = \big[x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\big].$$

Außerdem definieren wir den Domain *einer Substitution als*

$$\mathit{dom}(\sigma) = \{x_1, \cdots, x_n\}.$$

Die Menge aller Substitutionen bezeichnen wir mit Subst. ◇

Substitutionen werden für uns dadurch interessant, dass wir sie auf Terme *anwenden* können. Ist t ein Term und σ eine Substitution, so ist $t\sigma$ der Term, der aus t dadurch entsteht, dass jedes Vorkommen einer Variablen x_i durch den zugehörigen Term t_i ersetzt wird. Die formale Definition folgt.

Definition 44 (Anwendung einer Substitution)
Es sei t ein Term und es sei $\sigma = \big[x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\big]$ eine Substitution. Wir definieren die Anwendung *von σ auf t (Schreibweise $t\sigma$) durch Induktion über den Aufbau von t:*

1. *Falls t eine Variable ist, gibt es zwei Fälle:*
   (a) $t = x_i$ *für ein $i \in \{1, \cdots, n\}$. Dann definieren wir* $\quad x_i\sigma := t_i$.
   (b) $t = y$ *mit $y \in \mathcal{V}$, aber $y \notin \{x_1, \cdots, x_n\}$. Dann definieren wir* $\quad y\sigma := y$.

2. *Andernfalls muss t die Form $t = f(s_1, \cdots, s_m)$ haben. Dann können wir $t\sigma$ durch*

$$f(s_1, \cdots, s_m)\sigma := f(s_1\sigma, \cdots, s_m\sigma).$$

definieren, denn nach Induktions-Voraussetzung sind die Ausdrücke $s_i\sigma$ bereits definiert. ◇

Genau wie wir Substitutionen auf Terme anwenden können, können wir eine Substitution auch auf prädikatenlogische Klauseln anwenden. Dabei werden Prädikats-Zeichen und Junktoren wie Funktions-Zeichen behandelt. Wir ersparen uns eine formale Definition und geben statt dessen zunächst einige Beispiele. Wir definieren eine Substitution σ durch

$$\sigma := \big[x_1 \mapsto c,\ x_2 \mapsto f(d)\big].$$

In den folgenden drei Beispielen demonstrieren wir zunächst, wie eine Substitution auf einen Term angewendet werden kann. Im vierten Beispiel wenden wir die Substitution dann auf eine Formel an:

1. $x_3\sigma = x_3$,
2. $f(x_2)\sigma = f\big(f(d)\big)$,
3. $h(x_1, g(x_2))\sigma = h\big(c, g(f(d))\big)$.
4. $\big\{p(x_2), q(d, h(x_3, x_1))\big\}\sigma = \big\{p(f(d)),\ q(d, h(x_3, c))\big\}$.

Als nächstes zeigen wir, wie Substitutionen miteinander verknüpft werden können.

Definition 45 (Komposition von Substitutionen) *Es seien*

$$\sigma = \big[x_1 \mapsto s_1, \cdots, x_m \mapsto s_m\big] \quad \textit{und} \quad \tau = \big[y_1 \mapsto t_1, \cdots, y_n \mapsto t_n\big]$$

zwei Substitutionen mit $dom(\sigma) \cap dom(\tau) = \{\}$. Dann definieren wir die Komposition *$\sigma\tau$ von σ und τ als*

$$\sigma\tau := \big[x_1 \mapsto s_1\tau, \cdots, x_m \mapsto s_m\tau,\ y_1 \mapsto t_1, \cdots, y_n \mapsto t_n\big]$$ ◇

Beispiel: Wir führen das obige Beispiel fort und setzen

$$\sigma := \big[x_1 \mapsto c,\ x_2 \mapsto f(x_3)\big] \quad \text{und} \quad \tau := \big[x_3 \mapsto h(c, c),\ x_4 \mapsto d\big].$$

Dann gilt:

$$\sigma\tau = \big[x_1 \mapsto c,\ x_2 \mapsto f(h(c, c)),\ x_3 \mapsto h(c, c),\ x_4 \mapsto d\big].$$ □

Die Definition der Komposition von Substitutionen ist mit dem Ziel gewählt worden, dass der folgende Satz gilt.

Satz 46 *Ist t ein Term und sind σ und τ Substitutionen mit $dom(\sigma) \cap dom(\tau) = \{\}$, so gilt*

$$(t\sigma)\tau = t(\sigma\tau).$$ □

Der Satz kann durch Induktion über den Aufbau des Termes t bewiesen werden.

Definition 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.* ◇

Was syntaktische Gleichungen angeht 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 $E = \{s_1 \doteq t_1, \cdots, s_n \doteq t_n\}$ eine syntaktisches Gleichungs-System und ist σ eine Substitution, so definieren wir

$$E\sigma := \{s_1\sigma \doteq t_1\sigma, \cdots, s_n\sigma \doteq t_n\sigma\}.$$

Beispiel: Wir verdeutlichen die bisher eingeführten Begriffe anhand eines Beispiels. Wir betrachten die Gleichung

$$p(x_1, f(x_4)) \doteq p(x_2, x_3)$$

und definieren die Substitution

$$\sigma := \big[x_1 \mapsto x_2,\ x_3 \mapsto f(x_4)\big].$$

Die Substitution σ löst die obige syntaktische Gleichung, denn es gilt

$$p(x_1, f(x_4))\sigma = p(x_2, f(x_4)) \quad \text{und}$$
$$p(x_2, x_3)\sigma \quad = p(x_2, f(x_4)).$$ $\diamond$

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

$$f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n)$$

ist sicher dann nicht durch eine Substitution lösbar, wenn f und g verschiedene Funktions-Zeichen sind, denn für jede Substitution σ gilt ja

$$f(s_1, \cdots, s_m)\sigma = f(s_1\sigma, \cdots, s_m\sigma) \quad \text{und} \quad g(t_1, \cdots, t_n)\sigma = g(t_1\sigma, \cdots, t_n\sigma).$$

Falls $f \neq g$ ist, haben die Terme $f(s_1, \cdots, s_m)\sigma$ und $g(t_1, \cdots, t_n)\sigma$ verschieden Funktions-Zeichen und können daher syntaktisch nicht identisch werden.

Die andere Form einer syntaktischen Gleichung, die garantiert unlösbar ist, ist

$$x \doteq f(t_1, \cdots, t_n) \quad \text{falls } x \in \mathit{Var}\big(f(t_1, \cdots, t_n)\big).$$

Das diese syntaktische Gleichung unlösbar ist liegt daran, dass die rechte Seite immer mindestens ein Funktions-Zeichen mehr enthält als die linke.

Mit diesen Vorbemerkungen können wir nun ein Verfahren angeben, mit dessen Hilfe es möglich ist, Mengen von syntaktischen Gleichungen zu lösen, oder festzustellen, dass es keine Lösung gibt. Das Verfahren operiert auf Paaren der Form $\langle F, \tau\rangle$. Dabei ist F ein syntaktisches Gleichungs-System und τ ist eine Substitution. Wir starten das Verfahren mit dem Paar $\langle E, []\rangle$. Hierbei ist E das zu lösende Gleichungs-System und $[]$ ist die leere Substitution. Das Verfahren arbeitet indem die im Folgenden dargestellten Reduktions-Regeln solange angewendet werden, bis entweder feststeht, dass die Menge der Gleichungen keine Lösung hat, oder aber ein Paar der Form $\langle\{\}, \sigma\rangle$ erreicht wird. In diesem Fall ist σ ein Unifikator der Menge E, mit der wir gestartet sind. Es folgen die Reduktions-Regeln:

1. Falls $y \in \mathcal{V}$ eine Variable ist, die nicht in dem Term t auftritt, so können wir die folgende Reduktion durchführen:

$$\Big\langle E \cup \big\{y \doteq t\big\}, \sigma \Big\rangle \quad \rightsquigarrow \quad \Big\langle E\big[y \mapsto t\big], \sigma\big[y \mapsto t\big] \Big\rangle$$

   Diese Reduktions-Regel ist folgendermaßen zu lesen: Enthält die zu untersuchende Menge von syntaktischen Gleichungen eine Gleichung der Form $y \doteq t$, wobei die Variable y nicht in t auftritt, dann können wir diese Gleichung aus der gegebenen Menge von Gleichungen entfernen. Gleichzeitig wird die Substitution σ in die Substitution $\sigma\big[y \mapsto t\big]$ transformiert und auf die restlichen syntaktischen Gleichungen wird die Substitution $[y \mapsto t]$ angewendet.

2. Wenn die Variable y in dem Term t auftritt, falls also $y \in \mathit{var}(t)$ ist und wenn außerdem $t \neq y$ ist, dann hat das Gleichungs-System $E \cup \{y \doteq t\}$ keine Lösung, wir schreiben

$$\Big\langle E \cup \{y \doteq t\}, \sigma \Big\rangle \rightsquigarrow \Omega.$$

3. Falls $y \in \mathcal{V}$ eine Variable ist und t keine Variable ist, so haben wir folgende Reduktions-Regel:

$$\Big\langle E \cup \{t \doteq y\}, \sigma \Big\rangle \quad \rightsquigarrow \quad \Big\langle E \cup \{y \doteq t\}, \sigma \Big\rangle.$$

   Diese Regel wird benötigt, um anschließend eine der ersten beiden Regeln anwenden zu können.

4. Triviale syntaktische Gleichungen von Variablen können wir einfach weglassen:

$$\Big\langle E \cup \{x \doteq x\}, \sigma \Big\rangle \quad \rightsquigarrow \quad \Big\langle E, \sigma \Big\rangle.$$

5. Ist f ein n-stelliges Funktions-Zeichen, so gilt

$$\Big\langle E \cup \{f(s_1, \cdots, s_n) \doteq f(t_1, \cdots, t_n)\}, \sigma \Big\rangle \rightsquigarrow \Big\langle E \cup \{s_1 \doteq t_1, \cdots, s_n \doteq t_n\}, \sigma \Big\rangle.$$

   Eine syntaktische Gleichung der Form $f(s_1, \cdots, s_n) \doteq f(t_1, \cdots, t_n)$ wird also ersetzt durch die n syntaktische Gleichungen $s_1 \doteq t_1$, $\cdots$, $s_n \doteq t_n$.

   Diese Regel ist im übrigen der Grund dafür, dass wir mit Mengen von syntaktischen Gleichungen arbeiten müssen, denn auch wenn wir mit nur einer syntaktischen Gleichung starten, kann durch die Anwendung dieser Regel die Zahl der syntaktischen Gleichungen erhöht werden.

   Ein Spezialfall dieser Regel ist

$$\Big\langle E \cup \{c \doteq c\}, \sigma \Big\rangle \rightsquigarrow \Big\langle E, \sigma \Big\rangle.$$

   Hier steht c für eine Konstante, also ein 0-stelliges Funktions-Zeichen. Triviale Gleichungen über Konstanten können also einfach weggelassen werden.

6. Das Gleichungs-System $E \cup \{f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n)\}$ hat keine Lösung, falls die Funktions-Zeichen f und g verschieden sind, wir schreiben

$$\Big\langle E \cup \{f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n)\}, \sigma \Big\rangle \rightsquigarrow \Omega \qquad \text{falls } f \neq g.$$

Haben wir ein nicht-leeres Gleichungs-System E gegeben und starten mit dem Paar $\langle E, [] \rangle$, so lässt sich immer eine der obigen Regeln anwenden. Diese geht solange bis einer der folgenden Fälle eintritt:

1. Die 2. oder 6. Regel ist anwendbar. Dann ist das Ergebnis Ω und das Gleichungs-System E hat keine Lösung.
2. Das Paar $\langle E, [] \rangle$ wird reduziert zu einem Paar $\langle \{\}, \sigma \rangle$. Dann ist σ ein Unifikator von E. In diesem Falls schreiben wir $\sigma = \mathit{mgu}(E)$. Falls $E = \{s \doteq t\}$ ist, schreiben wir auch $\sigma = \mathit{mgu}(s, t)$. Die Abkürzung *mgu* steht hier für "*most general unifier*".

Beispiel: Wir wenden das oben dargestellte Verfahren an, um die syntaktische Gleichung

$$p(x_1, f(x_4)) \doteq p(x_2, x_3)$$

zu lösen. Wir haben die folgenden Reduktions-Schritte:

$$
\begin{array}{rl}
 & \big\langle \big\{ p(x_1, f(x_4)) \doteq p(x_2, x_3) \big\}, [] \big\rangle \\
\leadsto & \big\langle \big\{ x_1 \doteq x_2, f(x_4) \doteq x_3 \big\}, [] \big\rangle \\
\leadsto & \big\langle \big\{ f(x_4) \doteq x_3 \big\}, \big[x_1 \mapsto x_2\big] \big\rangle \\
\leadsto & \big\langle \big\{ x_3 \doteq f(x_4) \big\}, \big[x_1 \mapsto x_2\big] \big\rangle \\
\leadsto & \big\langle \big\{ \big\}, \big[x_1 \mapsto x_2,\ x_3 \mapsto f(x_4)\big] \big\rangle
\end{array}
$$

In diesem Fall ist das Verfahren also erfolgreich und wir erhalten die Substitution

$$\big[x_1 \mapsto x_2,\ x_3 \mapsto f(x_4)\big]$$

als Lösung der oben gegebenen syntaktischen Gleichung. ◇

Beispiel: Wir geben ein weiteres Beispiel und betrachten das Gleichungs-System

$$E = \big\{ p(h(x_1, c)) \doteq p(x_2),\ q(x_2, d) \doteq q(h(d, c), x_4) \big\}$$

Wir haben folgende Reduktions-Schritte:

$$
\begin{array}{rl}
 & \big\langle \big\{ p(h(x_1, c)) \doteq p(x_2),\ q(x_2, d) \doteq q(h(d, c), x_4) \big\}, [] \big\rangle \\
\leadsto & \big\langle \big\{ p(h(x_1, c)) \doteq p(x_2),\ x_2 \doteq h(d, c),\ d \doteq x_4 \big\}, [] \big\rangle \\
\leadsto & \big\langle \big\{ p(h(x_1, c)) \doteq p(x_2),\ x_2 \doteq h(d, c),\ x_4 \doteq d \big\}, [] \big\rangle \\
\leadsto & \big\langle \big\{ p(h(x_1, c)) \doteq p(x_2),\ x_2 \doteq h(d, c) \big\}, \big[x_4 \mapsto d\big] \big\rangle \\
\leadsto & \big\langle \big\{ p(h(x_1, c)) \doteq p(h(d, c)) \big\}, \big[x_4 \mapsto d,\ x_2 \mapsto h(d, c)\big] \big\rangle \\
\leadsto & \big\langle \big\{ h(x_1, c) \doteq h(d, c) \big\}, \big[x_4 \mapsto d,\ x_2 \mapsto h(d, c)\big] \big\rangle \\
\leadsto & \big\langle \big\{ x_1 \doteq d,\ c \doteq c \big\}, \big[x_4 \mapsto d,\ x_2 \mapsto h(d, c)\big] \big\rangle \\
\leadsto & \big\langle \big\{ x_1 \doteq d, \big\}, \big[x_4 \mapsto d,\ x_2 \mapsto h(d, c)\big] \big\rangle \\
\leadsto & \big\langle \big\{ \big\}, \big[x_4 \mapsto d,\ x_2 \mapsto h(d, c),\ x_1 \mapsto d\big] \big\rangle
\end{array}
$$

Damit haben wir die Substitution $\big[x_4 \mapsto d,\ x_2 \mapsto h(d, c),\ x_1 \mapsto d\big]$ als Lösung des anfangs gegebenen syntaktischen Gleichungs-Systems gefunden. ◇

5.5 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 leichter, einen vollstängen Kalkül für die Prädikatenlogik einzuführen. Zwar gibt es auch für den Fall, dass die Signatur Σ das Gleichheits-Zeichen enthält, einen vollständigen Kalkül. Dieser ist allerdings deutlich aufwendiger als der Kalkül, den wir jetzt einführen, 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. $p(s_1, \cdots, s_n)$ *und* $p(t_1, \cdots, t_n)$ *sind atomare Formeln,*
3. *die syntaktische Gleichung* $p(s_1, \cdots, s_n) \doteq p(t_1, \cdots, t_n)$ *ist lösbar mit*

$$\mu = \mathit{mgu}\big(p(s_1, \cdots, s_n), p(t_1, \cdots, t_n)\big).$$

Dann ist

$$\frac{k_1 \cup \{p(s_1, \cdots, s_n)\} \qquad \{\neg p(t_1, \cdots, t_n)\} \cup k_2}{k_1\mu \cup k_2\mu}$$

eine Anwendung der Resolutions-Regel. ◇

Die Resolutions-Regel ist eine Kombination aus der *Substitutions-Regel* und der Schnitt-Regel. Die Substitutions-Regel hat die Form

$$\frac{k}{k\sigma}.$$

Hierbei ist k eine prädikatenlogische Klausel und 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

$$M = \Big\{\big\{p(x)\big\}, \big\{\neg p(f(x))\big\}\Big\}$$

ist widersprüchlich. Wir können die Resolutions-Regel aber nicht unmittelbar anwenden, denn die syntaktische Gleichung

$$p(x) \doteq p(f(x))$$

ist unlösbar. Das liegt daran, dass **zufällig** in beiden Klauseln dieselbe Variable verwendet wird. Wenn wir die Variable x in der zweiten Klausel jedoch zu y umbenennen, erhalten wir die Klausel-Menge

$$\Big\{\big\{p(x)\big\}, \big\{\neg p(f(y))\big\}\Big\}.$$

Hier können wir die Resolutions-Regel anwenden, denn die syntaktische Gleichung

$$p(x) \doteq p(f(y))$$

hat die Lösung $[x \mapsto f(y)]$. Dann erhalten wir

$$\big\{p(x)\big\}, \quad \big\{\neg p(f(y))\big\} \quad \vdash \quad \{\}.$$

und haben damit die Inkonsistenz der Klausel-Menge M nachgewiesen.

Die Resolutions-Regel alleine ist nicht ausreichend, um aus einer Klausel-Menge M, die inkonsistent ist, in jedem Fall die leere Klausel ableiten zu können: Wir brauchen noch eine zweite Regel. Um das einzusehen, betrachten wir die Klausel-Menge

$$M = \Big\{\big\{p(f(x), y), p(u, g(v))\big\}, \big\{\neg p(f(x), y), \neg p(u, g(v))\big\}\Big\}$$

Wir werden gleich zeigen, dass die Menge M widersprüchlich ist. Man kann nachweisen, dass mit der Resolutions-Regel alleine ein solcher Nachweis nicht gelingt. Ein einfacher, aber für die Vorlesung zu aufwendiger Nachweis dieser Behauptung kann geführt werden, indem wir ausgehend von der Menge M alle möglichen Resolutions-Schritte durchführen. Dabei würden wir dann sehen, dass die leere Klausel nie berechnet wird. 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. $p(s_1, \cdots, s_n)$ *und* $p(t_1, \cdots, t_n)$ *sind atomare Formeln,*
3. *die syntaktische Gleichung* $p(s_1, \cdots, s_n) \doteq p(t_1, \cdots, t_n)$ *ist lösbar,*
4. $\mu = \mathit{mgu}\big(p(s_1, \cdots, s_n), p(t_1, \cdots, t_n)\big)$.

Dann sind

$$\frac{k \cup \{p(s_1, \cdots, s_n),\ p(t_1, \cdots, t_n)\}}{k\mu \cup \{p(s_1, \cdots, s_n)\mu\}} \quad \textit{und} \quad \frac{k \cup \{\neg p(s_1, \cdots, s_n),\ \neg p(t_1, \cdots, t_n)\}}{k\mu \cup \{\neg p(s_1, \cdots, s_n)\mu\}}$$

Anwendungen der Faktorisierungs-Regel. ◇

Wir zeigen, wie sich mit Resolutions- und Faktorisierungs-Regel die Widersprüchlichkeit der Menge M beweisen lässt.

1. Zunächst wenden wir die Faktorisierungs-Regel auf die erste Klausel an. Dazu berechnen wir den Unifikator

   $$\mu = \mathit{mgu}\big(p(f(x), y), p(u, g(v))\big) = [y \mapsto g(v), u \mapsto f(x)].$$

   Damit können wir die Faktorisierungs-Regel anwenden:

   $$\big\{p(f(x), y), p(u, g(v))\big\} \quad \vdash \quad \big\{p(f(x), g(v))\big\}.$$

2. Jetzt wenden wir die Faktorisierungs-Regel auf die zweite Klausel an. Dazu berechnen wir den Unifikator

   $$\mu = \mathit{mgu}\big(\neg p(f(x), y), \neg p(u, g(v))\big) = [y \mapsto g(v), u \mapsto f(x)].$$

   Damit können wir die Faktorisierungs-Regel anwenden:

   $$\big\{\neg p(f(x), y), \neg p(u, g(v))\big\} \quad \vdash \quad \big\{\neg p(f(x), g(v))\big\}.$$

3. Wir schließen den Beweis mit einer Anwendung der Resolutions-Regel ab. Der dabei verwendete Unifikator ist die leere Substitution, es gilt also $\mu = []$.

   $$\big\{p(f(x), g(v))\big\}, \quad \big\{\neg p(f(x), g(v))\big\} \quad \vdash \quad \{\}.$$

Ist M eine Menge von prädikatenlogischen Klauseln und ist k eine prädikatenlogische Klausel, die durch Anwendung der Resolutions-Regel und der Faktorisierungs-Regel aus M hergeleitet werden kann, so schreiben wir

$$M \vdash k.$$

Dies wird als M *leitet* k *her* gelesen.

Definition 51 (Allabschluss) *Ist k eine prädikatenlogische Klausel und ist $\{x_1, \cdots, x_n\}$ die Menge aller Variablen, die in k auftreten, so definieren wir den* Allabschluss *$\forall(k)$ der Klausel k als*

$$\forall(k) := \forall x_1 \colon \cdots \forall x_n \colon k.$$ ◇

Die für uns wesentlichen Eigenschaften des Beweis-Begriffs $M \vdash k$ werden in den folgenden beiden Sätzen zusammengefasst.

Satz 52 (Korrektheits-Satz)
Ist $M = \{k_1, \cdots, k_n\}$ eine Menge von Klauseln und gilt $M \vdash k$, so folgt

$$\models \forall(k_1) \wedge \cdots \wedge \forall(k_n) \rightarrow \forall(k).$$

Falls also eine Klausel k aus einer Menge M hergeleitet werden kann, so ist k tatsächlich eine Folgerung aus M. □

Die Umkehrung des obigen Korrektheits-Satzes gilt nur für die leere Klausel.

Satz 53 (Widerlegungs-Vollständigkeit)
Ist $M = \{k_1, \cdots, k_n\}$ eine Menge von Klauseln und gilt $\models \forall(k_1) \wedge \cdots \wedge \forall(k_n) \rightarrow \bot$, so folgt

$$M \vdash \{\}.$$ □

Damit haben wir nun ein Verfahren in der Hand, um für eine gegebene prädikatenlogischer Formel f die Frage, ob $\models f$ gilt, untersuchen zu können.

1. Wir berechnen zunächst die Skolem-Normalform von $\neg f$ und erhalten dabei so etwas wie
   $$\neg f \approx_e \forall x_1, \cdots, x_m : g.$$
2. Anschließend bringen wir die Matrix g in konjunktive Normalform:
   $$g \leftrightarrow k_1 \wedge \cdots \wedge k_n.$$
   Daher haben wir nun
   $$\neg f \approx_e k_1 \wedge \cdots \wedge k_n$$
   und es gilt:
   $$\models f \quad \text{g.d.w.} \quad \{\neg f\} \models \bot \quad \text{g.d.w.} \quad \{k_1, \cdots, k_n\} \models \bot.$$
3. Nach dem Korrektheits-Satz und dem Satz über die Widerlegungs-Vollständigkeit gilt
   $$\{k_1, \cdots, k_n\} \models \bot \quad \text{g.d.w.} \quad \{k_1, \cdots, k_n\} \vdash \bot.$$
   Wir versuchen also, nun die Widersprüchlichkeit der Menge $M = \{k_1, \cdots, k_n\}$ zu zeigen, indem wir aus M die leere Klausel ableiten. Wenn diese gelingt, haben wir damit die Allgemeingültigkeit der ursprünglich gegebenen Formel f gezeigt.

Beispiel: Zum Abschluss demonstrieren wir das skizzierte Verfahren an einem Beispiel. Wir gehen von folgenden Axiomen aus:

1. Jeder Drache ist glücklich, wenn alle seine Kinder fliegen können.
2. Rote Drachen können fliegen.
3. Die Kinder eines roten Drachens sind immer rot.

Wie werden zeigen, dass aus diesen Axiomen folgt, dass alle roten Drachen glücklich sind. Als erstes formalisieren wir die Axiome und die Behauptung in der Prädikatenlogik. Wir wählen die folgende Signatur

$$\Sigma_{\mathit{Drache}} := \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity} \rangle \quad \text{mit}$$

1. $\mathcal{V} := \{x, y, z\}$.
2. $\mathcal{F} = \{\}$.
3. $\mathcal{P} := \{\mathit{rot}, \mathit{fliegt}, \mathit{glücklich}, \mathit{kind}\}$.
4. $\mathit{arity} := \big\{\langle \mathit{rot}, 1\rangle, \langle \mathit{fliegt}, 1\rangle, \langle \mathit{glücklich}, 1\rangle, \langle \mathit{kind}, 2\rangle\big\}$

Das Prädikat $\mathit{kind}(x, y)$ soll genau dann wahr sein, wenn x ein Kind von y ist. Formalisieren wir die Axiome und die Behauptung, so erhalten wir die folgenden Formeln $f_1, \cdots, f_4$:

1. $f_1 := \forall x : \Big(\forall y : \big(\mathit{kind}(y, x) \rightarrow \mathit{fliegt}(y)\big) \rightarrow \mathit{glücklich}(x)\Big)$
2. $f_2 := \forall x : \big(\mathit{rot}(x) \rightarrow \mathit{fliegt}(x)\big)$
3. $f_3 := \forall x : \big(\mathit{rot}(x) \rightarrow \forall y : \big(\mathit{kind}(y, x) \rightarrow \mathit{rot}(y)\big)\big)$
4. $f_4 := \forall x : \big(\mathit{rot}(x) \rightarrow \mathit{glücklich}(x)\big)$

Wir wollen zeigen, dass die Formel

$$f := f_1 \wedge f_2 \wedge f_3 \rightarrow f_4$$

allgemeingültig ist. Wir betrachten also die Formel $\neg f$ und stellen fest

$$\neg f \leftrightarrow f_1 \wedge f_2 \wedge f_3 \wedge \neg f_4.$$

Als nächstes müssen wir diese Formel in eine Menge von Klauseln umformen. Da es sich hier um eine Konjunktion mehrerer Formeln handelt, können wir die einzelnen Formeln f_1, f_2, f_3 und $\neg f_4$ getrennt in Klauseln umwandeln.

1. Die Formel f_1 kann wie folgt umgeformt werden:

$$\begin{aligned}
f_1 &= \forall x : \Big(\forall y : \big(\mathit{kind}(y,x) \rightarrow \mathit{fliegt}(y)\big) \rightarrow \mathit{glücklich}(x)\Big) \\
&\leftrightarrow \forall x : \Big(\neg\forall y : \big(\mathit{kind}(y,x) \rightarrow \mathit{fliegt}(y)\big) \vee \mathit{glücklich}(x)\Big) \\
&\leftrightarrow \forall x : \Big(\neg\forall y : \big(\neg\mathit{kind}(y,x) \vee \mathit{fliegt}(y)\big) \vee \mathit{glücklich}(x)\Big) \\
&\leftrightarrow \forall x : \Big(\exists y : \neg\big(\neg\mathit{kind}(y,x) \vee \mathit{fliegt}(y)\big) \vee \mathit{glücklich}(x)\Big) \\
&\leftrightarrow \forall x : \Big(\exists y : \big(\mathit{kind}(y,x) \wedge \neg\mathit{fliegt}(y)\big) \vee \mathit{glücklich}(x)\Big) \\
&\leftrightarrow \forall x : \exists y : \Big(\big(\mathit{kind}(y,x) \wedge \neg\mathit{fliegt}(y)\big) \vee \mathit{glücklich}(x)\Big) \\
&\approx_e \forall x : \Big(\big(\mathit{kind}(s(x),x) \wedge \neg\mathit{fliegt}(s(x))\big) \vee \mathit{glücklich}(x)\Big)
\end{aligned}$$

   Im letzten Schritt haben wir dabei die Skolem-Funktion s mit $\mathit{arity}(s) = 1$ eingeführt. Anschaulich berechnet diese Funktion für jeden Drachen x, der nicht glücklich ist, ein Kind $s(x)$, das nicht fliegen kann. Wenn wir in der Matrix dieser Formel das "$\vee$" noch ausmultiplizieren, so erhalten wir die beiden Klauseln

$$k_1 := \big\{\mathit{kind}(s(x),x), \mathit{glücklich}(x)\big\},$$
$$k_2 := \big\{\neg\mathit{fliegt}(s(x)), \mathit{glücklich}(x)\big\}.$$

2. Analog finden wir für f_2:

$$\begin{aligned}
f_2 &= \forall x : \big(\mathit{rot}(x) \rightarrow \mathit{fliegt}(x)\big) \\
&\leftrightarrow \forall x : \big(\neg\mathit{rot}(x) \vee \mathit{fliegt}(x)\big)
\end{aligned}$$

   Damit ist f_2 zu folgender Klauseln äquivalent:

$$k_3 := \big\{\neg\mathit{rot}(x), \mathit{fliegt}(x)\big\}.$$

3. Für f_3 sehen wir:

$$\begin{aligned}
f_3 &= \forall x : \Big(\mathit{rot}(x) \rightarrow \forall y : \big(\mathit{kind}(y,x) \rightarrow \mathit{rot}(y)\big)\Big) \\
&\leftrightarrow \forall x : \Big(\neg\mathit{rot}(x) \vee \forall y : \big(\neg\mathit{kind}(y,x) \vee \mathit{rot}(y)\big)\Big) \\
&\leftrightarrow \forall x : \forall y : \big(\neg\mathit{rot}(x) \vee \neg\mathit{kind}(y,x) \vee \mathit{rot}(y)\big)
\end{aligned}$$

   Das liefert die folgende Klausel:

$$k_4 := \big\{\neg\mathit{rot}(x), \neg\mathit{kind}(y,x), \mathit{rot}(y)\big\}.$$

4. Umformung der Negation von f_4 liefert:

$$\begin{aligned}
\neg f_4 &= \neg\forall x : \big(\mathit{rot}(x) \rightarrow \mathit{glücklich}(x)\big) \\
&\leftrightarrow \neg\forall x : \big(\neg\mathit{rot}(x) \vee \mathit{glücklich}(x)\big) \\
&\leftrightarrow \exists x : \neg\big(\neg\mathit{rot}(x) \vee \mathit{glücklich}(x)\big) \\
&\leftrightarrow \exists x : \big(\mathit{rot}(x) \wedge \neg\mathit{glücklich}(x)\big) \\
&\approx_e \mathit{rot}(d) \wedge \neg\mathit{glücklich}(d)
\end{aligned}$$

Die hier eingeführte Skolem-Konstante d steht für einen unglücklichen roten Drachen. Das führt zu den Klauseln

$$k_5 = \{\mathit{rot}(d)\},$$
$$k_6 = \{\neg\mathit{glücklich}(d)\}.$$

Wir müssen also untersuchen, ob die Menge M, die aus den folgenden Klauseln besteht, widersprüchlich ist:

1. $k_1 = \{\mathit{kind}(s(x), x),\ \mathit{glücklich}(x)\}$
2. $k_2 = \{\neg\mathit{fliegt}(s(x)),\ \mathit{glücklich}(x)\}$
3. $k_3 = \{\neg\mathit{rot}(x),\ \mathit{fliegt}(x)\}$
4. $k_4 = \{\neg\mathit{rot}(x),\ \neg\mathit{kind}(y, x),\ \mathit{rot}(y)\}$
5. $k_5 = \{\mathit{rot}(d)\}$
6. $k_6 = \{\neg\mathit{glücklich}(d)\}$

Sei also $M := \{k_1, k_2, k_3, k_4, k_5, k_6\}$. Wir zeigen, dass $M \vdash \bot$ gilt:

1. Es gilt
   $$\mathit{mgu}\big(\mathit{rot}(d), \mathit{rot}(x)\big) = [x \mapsto d].$$
   Daher können wir die Resolutions-Regel auf die Klauseln k_5 und k_4 wie folgt anwenden:
   $$\{\mathit{rot}(d)\},\ \{\neg\mathit{rot}(x), \neg\mathit{kind}(y, x), \mathit{rot}(y)\} \ \vdash\ \{\neg\mathit{kind}(y, d), \mathit{rot}(y)\}.$$
2. Wir wenden nun auf die resultierende Klausel und auf die Klausel k_1 die Resolutions-Regel an. Dazu berechnen wir zunächst
   $$\mathit{mgu}\big(\mathit{kind}(y, d), \mathit{kind}(s(x), x)\big) = [y \mapsto s(d), x \mapsto d].$$
   Dann haben wir
   $$\{\neg\mathit{kind}(y, d), \mathit{rot}(y)\},\ \{\mathit{kind}(s(x), x), \mathit{glücklich}(x)\} \ \vdash\ \{\mathit{glücklich}(d), \mathit{rot}(s(d))\}.$$
3. Jetzt wenden wir auf die eben abgeleitete Klausel und die Klausel k_6 die Resolutions-Regel an. Wir haben:
   $$\mathit{mgu}\big(\mathit{glücklich}(d), \mathit{glücklich}(d)\big) = []$$
   Also erhalten wir
   $$\{\mathit{glücklich}(d), \mathit{rot}(s(d))\},\ \{\neg\mathit{glücklich}(d)\} \ \vdash\ \{\mathit{rot}(s(d))\}.$$
4. Auf die Klausel $\{\mathit{rot}(s(d))\}$ und die Klausel k_3 wenden wir die Resolutions-Regel an. Zunächst haben wir
   $$\mathit{mgu}\big(\mathit{rot}(s(d)), \neg\mathit{rot}(x)\big) = [x \mapsto s(d)]$$
   Also liefert die Anwendung der Resolutions-Regel:
   $$\{\mathit{rot}(s(d))\},\ \{\neg\mathit{rot}(x), \mathit{fliegt}(x)\} \ \vdash\ \{\mathit{fliegt}(s(d))\}$$
5. Um die so erhaltenen Klausel $\{\mathit{fliegt}(s(d))\}$ mit der Klausel k_3 resolvieren zu können, berechnen wir
   $$\mathit{mgu}\big(\mathit{fliegt}(s(d)), \mathit{fliegt}(s(x))\big) = [x \mapsto d]$$
   Dann liefert die Resolutions-Regel
   $$\{\mathit{fliegt}(s(d))\},\ \{\neg\mathit{fliegt}(s(x)), \mathit{glücklich}(x)\} \ \vdash\ \{\mathit{glücklich}(d)\}.$$
6. Auf das Ergebnis $\{\mathit{glücklich}(d)\}$ und die Klausel k_6 können wir nun die Resolutions-Regel anwenden:
   $$\{\mathit{glücklich}(d)\},\ \{\neg\mathit{glücklich}(d)\} \ \vdash\ \{\}.$$

Da wir im letzten Schritt die leere Klausel erhalten haben, ist insgesamt $M \vdash \bot$ nachgewiesen worden und damit haben wir gezeigt, dass alle kommunistischen Drachen glücklich sind. ◇

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

$$\forall x : \big(x \in R \leftrightarrow \neg x \in x\big).$$

Zeigen Sie mit Hilfe des in diesem Abschnitt definierten Kalküls, dass diese Formel widersprüchlich ist.

Aufgabe 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 Barbiere 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 5.6 auf Seite 123 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 5.6 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 5.6: 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 5.7 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.

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.

```
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 5.7: Gilt das Kommutativ-Gesetz in allen Gruppen?

Abbildung 5.8 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 5.8 gezeigte Text in einer Datei mit dem Namen "*group.in*" gespeichert, so können wir *Mace4* durch das Kommando

mace4 -f group.in

starten. *Mace4* sucht für alle positiven natürlichen Zahlen $n = 1, 2, 3, \cdots$, ob es eine Struktur $\mathcal{S} = \langle \mathcal{U}, \mathcal{J} \rangle$ mit $\mathit{card}(U) = n$ gibt, in der die angegebenen Formeln gelten. Bei $n = 6$ wird *Mace4* fündig und berechnet tatsächlich eine Gruppe mit 6 Elementen, in der das Kommutativ-Gesetz verletzt ist.

```
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 5.8: Gibt es eine Gruppe, in der das Kommutativ-Gesetz nicht gilt?

Abbildung 5.9 zeigt einen Teil der von *Mace4* produzierten Ausgabe. Die Elemente der Gruppe sind die Zahlen $0, \cdots, 5$, die Konstante a ist das Element 0, b ist das Element 1, e ist das Element 2. Weiter sehen wir, dass das Inverse von 0 wieder 0 ist, das Inverse von 1 ist 1 das Inverse von 2 ist 2, das Inverse von 3 ist 4, das Inverse von 4 ist 3 und das Inverse von 5 ist 5. Die Multiplikation wird durch die folgende Gruppen-Tafel realisiert:

$\circ$	0	1	2	3	4	5
0	2	3	0	1	5	4
1	4	2	1	5	0	3
2	0	1	2	3	4	5
3	5	0	3	4	2	1
4	1	5	4	2	3	0
5	3	4	5	0	1	2

Diese Gruppen-Tafel zeigt, dass

$$a \circ b = 0 \circ 1 = 3, \quad \text{aber} \quad b \circ a = 1 \circ 0 = 4$$

gilt, mithin ist das Kommutativ-Gesetz tatsächlich verletzt.

Bemerkung: Der Theorem-Beweiser *Prover9* ist ein Nachfolger des Theorem-Beweisers *Otter*. Mit Hilfe von *Otter*

```
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 5.9: Ausgabe von *Mace4*.

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. Nichtdestoweniger 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.

[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.