Skript

An Introduction to Coq

Wintersemester 2019/2020

Tanja Almeroth
tanja.almeroth@hs-rm.de

Hochschule RheinMain
Fachbereich Design Informatik Medien

Wenn Leute nicht glauben, dass Mathematik
einfach ist, dann nur deshalb, weil sie nicht begreifen,
wie kompliziert das Leben ist.

John von Neumann

https://www.youtube.com/watch?v=2E7fBHIUGBs

Contents

1. Introduction 1
- 1.1. Preface 1
- 1.2. Overview 1
- 1.3. Logic 1
- 1.4. Proof Assistants 2
- 1.5. Trivia 2
- 1.6. Functional Programming 2
- 1.7. System Requirements 2
- 1.8. Languages 3

2. Basics: Functional Programming in Coq 5
- 2.1. Introduction 5
- 2.2. Data and Functions 5
- 2.3. Boolean 6
- 2.4. Some Notation 7
- 2.5. Types 7
- 2.6. New Types from Old 7
- 2.7. Tuples 8
- 2.8. Modules 8
- 2.9. Numbers 8
- 2.10. Multi Argument-Function by Recursion 11
- 2.11. Introducing Notations 11
- 2.12. Proof by Simplification 12
- 2.13. Proof-Techniques 13
- 2.14. Proof by Case Analysis 15

3. Importing 18
- 3.1. Building Coq Libraries 18
- 3.2. Potential Troubles 19

4. Induction 20
- 4.1. Proof by Induction 20
- 4.2. Proofs Within Proofs 21
- 4.3. Formal vs. Informal Proofs 22

5. Lists - Working with structured Data 23
- 5.1. Pairs of Numbers 23
- 5.2. Introduction to RT-Proofs 24
- 5.3. Implementation In Coq 24
- 5.4. Axioms of Kolmogorov 39
- 5.5. Bell's Inequality 40
- 5.6. PROSA and the Discretization of Time 41

6. Coq as Code Generator 42

A. Additional Materials 45
- A.1. Coq and Predicate Logic 45

B. Grundlagen und Schreibweisen **46**
B.1. Mengen . . . 46
B.1.1. Die Elementbeziehung und die Enthaltenseinsrelation . . . 46
B.1.2. Definition spezieller Mengen . . . 46
B.1.3. Operationen auf Mengen . . . 47
B.1.4. Gesetze für Mengenoperationen . . . 48
B.1.5. Tupel (Vektoren) und das Kreuzprodukt . . . 48
B.1.6. Die Anzahl von Elementen in Mengen . . . 49
B.2. Relationen und Funktionen . . . 49
B.2.1. Eigenschaften von Relationen . . . 49
B.2.2. Eigenschaften von Funktionen . . . 50
B.2.3. Hüllenoperatoren . . . 51
B.2.4. Permutationen . . . 52
B.3. Summen und Produkte . . . 53
B.3.1. Summen . . . 53
B.3.2. Produkte . . . 53
B.4. Logarithmieren, Potenzieren und Radizieren . . . 54
B.5. Gebräuchliche griechische Buchstaben . . . 55

C. Grundlagen und Schreibweisen **56**
C.1. Mengen . . . 56
C.1.1. Die Elementbeziehung und die Enthaltenseinsrelation . . . 56
C.1.2. Definition spezieller Mengen . . . 56
C.1.3. Operationen auf Mengen . . . 57
C.1.4. Gesetze für Mengenoperationen . . . 58
C.1.5. Tupel (Vektoren) und das Kreuzprodukt . . . 58
C.1.6. Die Anzahl von Elementen in Mengen . . . 59
C.2. Relationen und Funktionen . . . 59
C.2.1. Eigenschaften von Relationen . . . 59
C.2.2. Eigenschaften von Funktionen . . . 60
C.2.3. Hüllenoperatoren . . . 61
C.2.4. Permutationen . . . 62
C.3. Summen und Produkte . . . 63
C.3.1. Summen . . . 63
C.3.2. Produkte . . . 63
C.4. Logarithmieren, Potenzieren und Radizieren . . . 64
C.5. Gebräuchliche griechische Buchstaben . . . 65

D. Einige (wenige) Grundlagen der elementaren Logik **67**

E. Einige formale Grundlagen von Beweistechniken **69**
E.1. Direkte Beweise . . . 70
E.1.1. Die Kontraposition . . . 71
E.2. Der Ringschluss . . . 71
E.3. Widerspruchsbeweise . . . 72
E.4. Der Schubfachschluss . . . 73
E.5. Gegenbeispiele . . . 73
E.6. Induktionsbeweise und das Induktionsprinzip . . . 73
E.6.1. Die vollständige Induktion . . . 74
E.6.2. Induktive Definitionen . . . 75
E.6.3. Die strukturelle Induktion . . . 76

1. Introduction

This is a summary of the electronic text-book [PdAC+19] with comments and a little rewritten sections by the author and based on the peers and supervisor's feedback. Other references are marked. This summary aims to be give a clear instruction of the usage and applicability of Coq.

1.1. Preface

Within this introduction the mathematical underpinning of reliable software is given. The building blocks are
!!!TODO!!!

- basics concepts of logic (see sec. **??**
- computer assisted theorem proving (see sec. **??**
- Coq-proof assistant (see sec. **??**
- functional programming (see sec. **??**
- operational semantics (see sec. **??**
- logics for reasoning about programs (see sec. **??**
- static type systems (see sec. **??**

1.2. Overview

There is a lot of motivation for reliable software. First of all the scale, complexity and number of involved people in modern systems is increasing. Therefore building correct software is extremely difficult. Information processing is waved into every aspect of society, leading to amplified costs of bugs and insecurities upto multiple levels. An on the hand rule says the later an error is located the more expensive it is.
Computer scientists and software engineers have responded to improve reliability with a lot of design threats and to improve reliability and mathematical technices for reasoning. Within this work it should be contributed to validates these properties. They are:

1. basic tools from logic for making and justifying precise claims about programs
2. use of proof assistants to construct rigorous logical arguments
3. functional programmings as method of programming and simplifying reasoning about programs as a bridge between programming and logic

1.3. Logic

> "As a matter of fact, logic has turned out to be significantly more effective in computer science then it has been in mathematics." [PdAC+19]

Volumes have been written about the central role of logic in computer science. It's fundamental tool *inductive proof* is going to be explored very deeply within this work.

1.4. Proof Assistants

In computer science proof assistants are an important tool for helping construct formal proofs of logical propositions. There are two categories of these tools.
First of all there are automated theorem proofers. These are able of a ”push-button-operation” which returns true, false” or “ran out of time” given a proposition. Example applications are SAT-solver, SMT-solvers or model checkers. Second there are proof assistants, which are hybrid tools that automate the more routine-like aspects of a proof, while depending on human guidance. Examples are Isabelle, Agda, Twelf, ACL2, PVS or Coq.
The Coq proof assistant has been developed since 1983 and gathered a large community in research and industry. It provides a rich environment for interactive development of machine-checked code for formal reasoning.
It’s kernel is a simple proof checker, ensuring that correct dilution of sets are ever performed. Moreover, there are high-level facilities for proof development. Coq has been applied as critical enabler across computer science and mathematics a platform for modeling programming languages and as an environment for developing *formally certified software and hardware.*

1.5. Trivia

Some French computer scientist have a tradition of naming their software as animal species. *Coq* is the French word for roster, which is the national symbol of France. Coq sounds like the initial of the Calculus of Constructions. One of Coq’s early developers is called Terry Coquand.

1.6. Functional Programming

There are two meanings of the term. It is either refereed to programming idioms (something like a pattern) or something else as in this work.

Functional programming refers to a way of programming, which is free of side effects. By side effects phenomena as I/O or redirecting pointers are meant. For example, let’s imagine iterative sorting. A sorting-function might take a list of numbers and rearrange the pointers to these numbers. In functional programming a new list is returned which contains the same number arranged in an order.
Advantages of functional programming are that we are having a new data structure leaving the old one intact. Therefore there is no reason to worry about the structure being shared, whether one part of the program might break an invariant that another part of the program relies on.
The industry is interested in functional programming due to it’s simple behavior in the presence of accuracy. Furthermore, functional programming is more easy to parallelize then the counter parts. For example the map-reduce idiom, which relies at the heart of massively distributed query processor like Hadoop is functional programming.

> “ [...] When we come to look more closely, we find that these two sides of Coq are actually aspects of the very same underlying machinery - i.e. proofs are programs.”

1.7. System Requirements

Coq runs on Windows, Linux and macOS. A current installation can be found on the Coq-homepage [Coqa]. The listings in this work from [PdAC+19] have been tested using Coq 8.8.1. The following choices of integrated development environments (IDEs) are available.

Coq in the command line It is not recommended to use Coq in the command line mode. Because the interactive mode is preferred, when using Coq as a proof assistant.

Proof General Proof general is an Emacs based IDE. It is recommended to users who are familiar with the Emacs-editor. Proof general supports multipel proof assistants. The corresponding proof general mode, the socalled `Coq-mode`, will be invoted automatically when a proof script a `v.`-file is opened [PROa].

CoqIDE CoqIDE is a simple stand-alone IDE. It labels itself as a user-friendly replacement to `coqtop` [COQb]. It should be available with any Coq-installation. It shall be warned that CoqIDE should be run with the asynchronous and error reliance model disabled.

Coquille Coquille is a vim plug-in used by the author. It can be found at https://github.com/Werner2005/coquille. It provides syntax check by color highlighting and and interactive evaluation. The author worked with coquille to explorer Coq.
It provides a similar workflow as CoqIDE. Coquille labels itself as a user friendly replacement of `coqtop`. The running buffer is the one where navigation takes place. To that coquille provides forwards and backwards navigation.

In order to launch coquille open a Coq script (a `.v`-file) by gvim or vim from the console and run (`:CoqLaunch`). Coqille's main screen provides an evaluation of the Coq-code untill the position of the curser (press `F4`), a stepwise forward (press `F2`) and backwards (press `F3`) evaluation. Furthermore, the plug-in provides syntax highlighting (see Figure **??**). While running a buffer it is deposited grey. A green deposition indicates a correct evaluation. And a just proven subgoal is displayed in the goal window. Failing commands are deposited red and the message window reports errors.

Encoding Coq proof scripts (`.v`-files) are encoded by ASCII. But support for unicode is provided, too [Gal].

1.8. Languages

Coq uses three different languages. First, there is a vernacular language. It is a top-level interaction. It's keywords start with a capital letter e.g. `Theorem`, `Proof` and `Qed`. Second, there is the tactics language (e.g. `intros` and `exact`). Finally there is an unnamed language of Coq-terms. It consists of a lot of operators (e.g. `for all A:Prop, A → A`). Technically, it is part of the tactics language, but it is useful to think of it as it's own thing [?]. https://coq.inria.fr/refman/language/gallina-specification-language.html

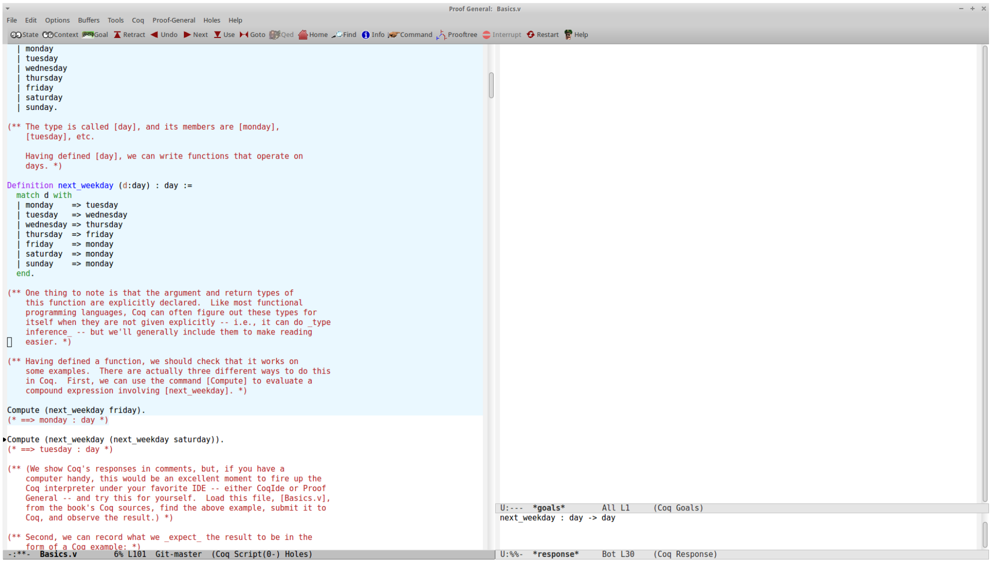

Figure 1: Coq interface launched in gvim using coquille.
Left window: The open Coq-script `Basics.v`. Upper right: The *goal window*. Lower right: The *message window* [COQb].

2. Basics: Functional Programming in Coq

In this chapter we are going to introduce the most essential elements of Coq's functional programming language called *Gallina*. Moreover, *tactics* which can be applied to prove properties of Coq programs are introduced (subsection 2.13 and 2.14).

2.1. Introduction

As mentioned in section 1 the Coq-listing in this work are from the book [PdAC+19]. In this work the definitions from Coq-standart libraries are given to recapulate common data structures, which are deliverd with common Coq-distributions. The listings are given in such an order as they would be in an executable Coq-script.

2.2. Data and Functions

Enumerated Type The built-in features set in Coq is extremely small. In particular, Coq is a powerful mechanism for defining data types from scratch. Current Coq distributions come preloaded with an extensive standard library containing Boolean, numbers, data structures like lists and hash tables.

In this course we are going to explicitly recapitulate the used definitions. It is started by a rather simple example of a self-defined type.

Example 1: *We are defining a type called* `day`*.*

```
Inductive day: Type :=
  | monday
  | tuesday
  | wednesday
  | thursday
  | friday
  | saturday
  | sunday.
```

Listing 1: `day`

The type members are called `monday`*,* `tuesday` *,* `wednesday` *… and* `sunday`*.*
We are defining a function operating on `day`*:*

```
Definition next_weekday(d:day) day:=
  match d with
    | monday ⇒tuesday
    | tuesday ⇒wednesday
    | wednesday ⇒thursday
    | thursday ⇒friday
    | friday ⇒monday
    | saturday ⇒monday
    | sunday ⇒monday
  end.
```

Listing 2: `next_weekday`

Coq is able of *type interference*, whenever the type is not defined explicitly. But for readability, we are including types in the following. Note that the keyword for function is `Definition` and the type if called `Inductive`. We are calling the data-type `day` an inductively defined data-type.

For testing the above definition and function we have three possibilities in Coq:

1. Compute a compound expression including the function:
   `compute (next_weekday friday).` Coq output: `= monday : day` or
   `compute (next_weekday (next_weekday saturday)).` Coq output: `= tuesday : day`

2. Record an expected behaviour as a Coq-`Example` and verify the assertion:
   The second weekday after Saturday is Tuesday.

```
Example. test_next_weekday: (next_weekday
    (next_weeekday(saturday)) = tuesday
Proof. simpl. reflexivity. Qed.
```

Listing 3: `text_next_weekday`

   The details of the implementation are not important right now. We are going to come back to them later.

3. Moreover, Coq can be asked to `extract` a program from our `Definition` in a programming language which is more conventional, being equipped with a high performance compiler. In particular this is one of the main uses of Coq. It provides a method to transfer proved-correct algorithms in Gallina to efficient machine code (see section 6). (Assuming the correctness of the corresponding high performance compiler e.g. the Ocaml, Haskell or Scheme compiler.)

2.3. Boolean

Of course, Coq provides an default implementation of Boolean. (See the Coq.Init.Datatypes in the Coq-Library documentation). In the following we are going to be consistent with the Coq library documentation according to the standard library. We are going to introduce coinciding self-defined data types whenever possible.
Boolean can be defined as follows:

```
Inductive bool: Type :=
  | true
  | false.
```

Listing 4: `bool`

A function with multiple input arguments is implemented by

```
Definition orb (b1: bool) (b2: bool) : bool :=
match b1 with
  | true →true
  | false →b2
end.
```

Listing 5: `orb`

The interested reader might compare this to the set denoted by $\mathbb{B}$ in example 39 given in Appendix C.1.2. The functions `negb` and `andb`, corresponding to the Boolean functions *n*egation and *a*nd, are implemented in a similar manner.
And a new symbolic notations is implemented as follows:

```
Notation "x && y" := (andb x y).
```

```
        Notation "x||y" := (orb x y).
```

Listing 6: introducing a new notation

2.4. Some Notation

As in the Coq doc documentation tool the following notation convention is introduced:

1. In `.v-files` comments are annotated by *(* some comment *)*.
2. Within these comments Coq-code is denoted by `[example]`.
3. And we write `Admitted.` at the end of an incomplete proof.

2.5. Types

Every expression in Coq has a type, which can be revealed by `check`.

1. `Check true.` gives the type of the expression `boolean`.
2. `Check negb.` returns `bool → bool` . (Read as "bool arrow bool"). It is the function's input data type and the output data type, given input data of that type.
3. `andb` returns `bool→ bool→ bool`. This function produces an output of type `bool` given two inputs of type `bool`.

2.6. New Types from Old

Note that so far the data-types we have seen were enumerated types. Let's define a data type `rgb` and a data type `primary`, whose constructor takes this type as an argument.

```
        Inductive rgb: Type:=
          | red      (* These are the
               *)
          | green    (* expressions in
               *)
          | blue.    (* the set [rgb].
               *)
```

Listing 7: `rgb`

The constructors of the type `rgb` are `red`, `green` and `blue`.

```
        Inductive color: Type :=
          | black   (* An expression
              in the set color *)
          | white   (* An expression
              in the set color *)
          | primary (p:rgb). (* If
              [p] is in the set
              [rgb], then the
              constructor [primary]
              applied to the argument
              [p] is an expression in
              the set color.*)
```

Listing 8: `color`

The constructors of the type `color` are `black`, `white` and `primary`.

Note that expressions formed as in Listing `rgb` and `color` (see Listing 7 and 8) are the only ones belonging to the sets `rgb` and `color`. A function on this struct can by defined by patternmatching just as in the previus example (Listing 5).

```
        Definition monochrome (c : color) : bool :=
             match c with
              | black ⇒true
              | white ⇒true
```

```
            | primary q ⇒false
            end.
```

Listing 9: `monochrome`

2.7. Tuples

A single constructor with multiple parameters can be used to create a type `tuple`.

Example 2 (A nibble: half a byte):

```
Inductive bit: Type := (* a bit *)
 |B0
 |B1.

Inductive nibble: Type := (* a nibble *)
 |bits( b0 b1 b2 b3: bit).

Check(bits B1 B0 B1 B0) (* ⇒bits B1 B0 B1 B0 : nibble *)
```

Listing 10: `bit` and `nibble`

Hence, a tuple of four bits is a nibble. Assume we would like to test a nibbel in order to see if all it's bits are 0. The nibble is unwrapped by pattern-matching:

```
Definition all_zero(nb: nibble): bool :=
  match nb with
    |(bits B0 B0 B0 B0) →true
    |(bits _ _ _ _ ) →false (* This wildcard pattern was included to avoid introducing variable names, which are not going to be used. *)
  end.
```

Listing 11: `all_zero`

2.8. Modules

Coq provides a *module system* to aid organizing large developments (like Packages in Java (see [Ull17, Section 3.6.5]). In this course most of it is not going to be needed. Namespaces use the .-notation as in C++ Data structures (see [Cpp]).

```
Module X
...
... foo... (* a definition declared as foo *)
...
End X

X.foo (* Referes to foo, which is definded in the module X. *)
```

Listing 12: `Module`

2.9. Numbers

Note that on the one hand the types day, bool, bits and tuples have a finite set of values, while on the other hand the set of natural numbers $\mathbb{N}$ is an infinite set.
Hence, we have to construct $\mathbb{N}$ using a data type with a finite number of constructors. Recall, that many representations of $\mathbb{N}$ exist (e.g. hexadecimal with base 16, octa with base 8, binary with base 2). The most familiar might be the decimal representation.

However, each representation of ℕ can be useful under different circumstances. The binary representation is valuable in computer hardware, because it presents a simple circuitry. Here simple proofs are aimed using the unary (base 1) representation.

```
Module NatPlayground

Inductive nat: Type :=
  | O
  | S (n:nat).
```

Listing 13: `nat`

The two constructors of the set `nat` are `S` and `O`.

One might picture this representation by strokes in the beergarden. (Actually the definition in Listing 13 is an inductive definition as delinated in the Appendix E.6.2.)

- The expression `O` is in the set `nat`.
- The expression `O` represents the natural number zero.
- If `n` is an expression belonging to the set `nat`, then `Sn` represents a natural number $n \in \mathbb{N} \setminus \{0\}$.

There is a way of converting this representation refereed to as `nat` into it's decimal representation. This is elaborated in example 3.

Example 3 (Converting Unary to Decimal):

unary	decimal
O	: 0
S O	: 1
S (S O)	: 2
S(S (S O))	: 3

Moreover, it is easy to see the following:

- The expression `O` belongs to the set `nat`.
- If `n` is in the set `nat`, it follows `S n` is an expression belonging to the set `nat`.
- And expressions belongs the set `nat` if and only if they are formed by either of those methods (i.e. `O` or `S n`).

Note that the same rules apply to the definitions of `day`, `bool` and `color`. These conditions are a precise force of the `Inductive` declaration. Expressions build from other data constructors like `true`, `false`, `andb(S(false( O ( O S ))))` do not belong to this set. But on the other hand, `O` and `S` are arbitrary symbols chosen in the defined representation. Actually the *interpretation* of these marks is given by their usage in computing. To realize this functions, which "pattern match" a representation of natural numbers, are written.

We are following the idea: If n has the form `S m` for some `m` in the set `nat`, then return `m`.

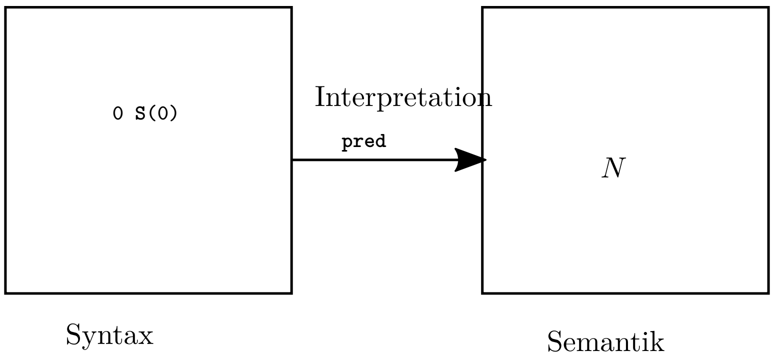

Figure 2: An Illustration of Syntax and Semantik at the Example of the Function `pred`.

Example 4 (Predecessor Function): Note that although the predecessor of the number zero is not defined, it was declared in Listing 14 for simplicity.

```
Definition pred (n : nat) : nat :=
  match n with
    | O ⇒ O
    | S m ⇒ m
  end.

End NatPlayground

Check pred.
```

Listing 14: `pred`

```
nat →nat
```

Listing 15: Coq-output

Because natural numbers are such a pervasive form of data, Coq always prints out a natural number as decimal by default. (It uses a tiny “build-in magic” for paring and printing.)

```
Definition minustwo( n: nat) : nat :=
  match n with
    | O ⇒ O
    | S O ⇒ O
    | S (S m) = > m
  end.

Compute(minustwo(4))
```

Listing 16: `minustwo`

```
= 2: nat
```

Listing 17: Coq-output

Note that, there is a fundamental difference. The functions `pred` and `minustwo` apply computational rules. There are no computational rules about `S`. I.e., by the definition of `pred`, `pred 2` can be simplified to `1`. In the case of `S` no such behaviour exists. The definition of `S` has no behaviour at all.

In order to define more functions over numbers pattern matching used in the definitions of `pred` and `minustwo` is not sufficient. We are introducing recursion. Recursion in Coq is notated by the keyword `Fixpoint`. Note that the previous definitions obviously did not use recursion, because the definitions did not call themseves.

```
Fixpoint evenb (n : nat) : bool :=
  match n with
  | 0 ⇒true
  | S 0 ⇒false
  | S (S n') ⇒evenb n'
  end.

Definition oddb (n:nat): bool := negb (even bn). (* a simple definition of odd
    *)

Example test_oddb1: oddb 1 = true
      Proof. simpl. reflexivity. Qed.
```

Listing 18: `evenb` and `oddb`

Note that in this proof `simpl`. actually has no effect on the goal. All the work is done by `reflexivity`. We are going to come to back to that later.

2.10. Multi Argument-Function by Recursion

```
Module NatPlayground2

Fixpoint plus (n : nat) (m : nat): nat :=
  match n with
    | 0 ⇒m
    | S n' ⇒S (plus n' m)
  end.

 Compute (plus 3 2).
```

Listing 19: `plus`

A notation convention for calling functions with two expressions and matching two expressions with multiple arguments of the same type exist in Coq, too. See Listing 20.

```
Fixpoint minus (n m: nat): nat :=
 match n, m with
   | 0 , _ ⇒0
   | S _ , 0 ⇒n
   | S n', S m' ⇒minus n' m'
  end.

 Example test_mult1: (minus 3 2) = 1.


 End NatPlayground2

 Fixpoint exp (base power : nat) : nat :=
   match power with
     | 0 ⇒S 0
     | S p ⇒mult base (exp base p)
   end.
```

Listing 20: `minus` and `exp`

2.11. Introducing Notations

For the Coq-parser and Coq-prettifyer's sake we are introducing a new notation:

notation	function	functionality	uses
x - y	minus	subtract two natural numbers	see Listing 20
x * y	mult	multiply natural numbers	nested `match`es
x =? y	eqb	test natural numbers for equality yielding a Boolean	nested `match`es
x <=? y	leb	test if a first argument is less or equal yielding a Boolean	nested `match`es

Table 1: common operators on natural numbers, full implementation can be found in [PdAC+19, section: Basics, Functional Programming in Coq: Numbers]

```
Notation " x + y ":= (plus x y)
                    (at level 50, left associativity) (* This line is not of
                        interest for our purpose. *)
                    : nat_scope. (* This line is not of our interest. *)

Check(( 0 + 1 ) + 1).
```

Listing 21: operator overloading of `+`

Listing 21 demonstrates the result of introducing a new notation using Coq and operator overloading by the example of `+`.
And in Table 1 some more functions for our purpose are listed. The interested reader is referred to the literature for an exact definition (see Table 1).
Note that we when we said that Coq comes with almost nothing built-in, we really mean it. Even equality testing is a user-defined operation.

2.12. Proof by Simplification

Till now we have defined a few data types and functions. Example: We have seen all claims were shown by the same proofs. `simple` and `reflexivity`. `simpl` simplified equations and `reflexivity` checked weather both sides contain identical values. A rule of thumb might say that, `simple` can be used to show more simple properties. For example consider the following observation: "0+n reduces to n, no matter, what n is." The mathematical precisely formulated statement can be proven by the definition of zero directly.

```
Theorem. plus_0_n: forall n: nat, 0+n = n.
  Proof. intros n. simpl. reflexivity. Qed.
```

Listing 22: `plus_0_n`

Remark 5: *`Reflexivity` does not only check whether both sides of an equation contain identical values. On top of this it simplifies, which makes it more powerful. We have seen `simpl` added in Listing 22 so we can see the intermediate state. Therefore, the proof in Listing 22 can be simplified by omitting `simpl`. (See Listing 23.)*

```
Theorem plus_0_n': forall n: nat, 0 + n = n.
  Proof: intros n. reflexivity. Qed.
```

Listing 23: `plus_0_n'`

By looking at the Coq output we can see that `reflexivity` somehow tries “unfolding” defined terms.
Note that `example` and `theorem` (and `Lemma`, `Fact`, `Remark` and a few other), mean pretty much the same in Coq, while in mathematics they do not.

2.13. Proof-Techniques

Intros, Simplification and Reflexivity Whenever a `Theorem` starts with `forall n`, we might start a proof by the phrase: Assume `n` is some natural number. By `intros n` we can tell this to Coq.
A *tactic* is a command used between `Proof` and `Qed`, which guides the process of checking some claim for example the keyword `intros`, `simpl` and `reflexivity` are tactics.

Remark 6: *Notation: The suffix `_l` is pronounced “on the left” (see Listing 24).*

```
Theorem mult_0_l: forall n: nat, 0 * n = 0.
  Proof. intros n. reflexivity. Qed.
```

Listing 24: `mult_0_l`

In order to understand the tactics `reflexivity` carefully analyses the Coq-output of Listing 24. Let’s recall what reflexivity in terms of an equivalence relation means:

Definition 7 (Equivalence Relation): *Let $\mathcal{M} \neq \emptyset$ be a binary set. Assume $x, y \in \mathcal{M}$ and $\sim \subset A \times A$ an arbitrary relation. Then $\sim$ is said to be an equivalence relation if and only if for all $a, b, c \in \mathcal{M}$*

1. *$a \sim a$ (reflexivity)*
2. *$a \sim b$ if and only if $b \sim a$ (symmetry)*
3. *if $a \sim b$ and $b \sim c$ then $a \sim c$ (transitivity)*

A more detailed approach of understanding the tactics `reflexivity` is elaborated in section 6. However, due to the Coq tactics index it is recommended to use reflexivity, if the goal is to prove that something is equifalent to the identety relation.

Proof by Rewriting Consider the following example theorem:

```
Theorem plus_id_example: forall n m: nat,
  n = m →
  n+n = m+m.
```

Listing 25: `plus_id_examples`

We would like to show this theorem instead by making a claim about all numbers by looking at the special case when `n=m`. $\rightarrow$ corresponds to the mathematical implication operator $\implies$. Note that the arrow $\rightarrow$ tells Coq to rewrite the object of focus from left to right and $\leftarrow$ can be used to rewrite from right to left.

In order to proof theorem `plus_id_example` the theorem’s hypothesis is rewritten as shown in Listing 26.

```
1 Theorem plus_id_example: forall n m: nat,
2   n = m →
3   n + n = m + m.
4   Proof.
5     intros n m.
6     intros H.
7     rewrite →H.
8     reflexivity.
9   Qed.
```

Listing 26: `plus_id_example` and it's proof

For clearity of the semantics this Coq-code's (Listing 26) translation to a mathematical proof is listed in table 2. N ote that the syntax and semantics of the Coq-code in

line no.	a mathematical translation	comment
1 2 3	Theorem (`plus_id_example`): $\forall n, m \in \mathbb{N}$: $n = m$ $\implies n + n = m + m.$	Declares the statement with the name `plus_id_example` in Coq as `subgoal`.
4	Proof:	Moves the last proven `subgoal` into Coq's focus.
5	Assume $n, m \in \mathbb{N}$.	Moves the universally quantified variables `n` and `m` into the focus.
6	$H(n, m) := (n = m)$.	Moves the hypothesis `H` of the `subgoal` into the focus of Coq.
7	$H(n, m) \implies$ $\implies (n + n = m + m \Leftrightarrow m + m = m + m)$.	Substitutes by H from left to right in the `subgoal`. Simplifies if possible.
8	trivial.	We obtained a trivial statement.
9	#.	Quod era demonstrandum.

Table 2: `plus_id_example`'s mathematical translation

Listing 26 can be easily understood by a reader who is familiar with the notation of predicate logic as in D. The function of the tactics `reflexivity` is also similar to the principle of reflexivity in predicate logic (see: Need a predicate logic reference!). Therefore a one-to-one translation of this Listing into predicate logic can be found in appendix A A.1.

`rewrite` can also be used with a previously proven theorem instead of a hypothesis of context. If the statement of the previously proven theorem involves prequantified variables, Coq reuses them.

```
Theorem mult_0_plus: forall n m: nat,
  (0 + n) * m = n * m.
Proof.
  intros n m.              (* Let n,m ∈ ℕ *)
  reflexivity →plus_0_n.   (* rewrite 0+n, by n *)
  reflexivity.             (* m = m *)
Qed.
```

Listing 27: `mult_0_plus`

Admitted and Abort The command `Admitted` tells Coq to skip a proof of a theorem and to accept it as given. It can be used in long proofs to subsidiary state longer `Lemmas`. If a proof was started it can be interrupted by the command `Abort`. Note, if a proof was forgotten in the following shenanigans is able to be shown.

2.14. Proof by Case Analysis

Consider the following example. It clearly demonstrates that, it is not possible to prove everything using the simplification tactic.

Destruct

```
Theorem plus_1_neq_0_firsttry : forall n: nat,
  (n + 1) =? 0 = false.
  Proof.
    intros n.
    simpl. (* Does nothing! *)
  Abort.
```

Listing 28: `plus_1_neq_0_firsttry`

One might imagine a natural number `n` as string build up of it's constructors `S` and `O` at the end. Evaluating the expression `(n+1)=? 0` suffices to either evaluate the operation `eqb` or `plus`. By recalling (or looking up) the declarations of those operators (see Listing 19 and table 1) it is noted that both scopes begin by a `match`. Hence a pattern matching of an unknown string that is the first argument of the operator is executed. But due to the generality of this string no matching can be proceeded. It follows `simpl` carries out nothing. `Abort` interrupts Coq. In general unknown hypothetical values like arbitrary numbers, Boolean or lists might not be able to be evaluated.
Note that if we assume `n=O` it can be calculated that `n+1` is not equal to zero. Hence `n+1?=0` equals to `false` by type interference. On the other hand if `n=Sm` for some arbitrary `m` $\in$ `nat` we can not calculate this expression. Although it is not know, which number `n+1` is, it can be calculate that it begins with an `S`. It follows `n+1 =? 0` evaluates to `false`.
If such a procedure is carried out by Coq, two subgoals have to be generated including variables named like in the called `intros` pattern.
Therefore the tactics `destruct` is applied.

```
intros n.
destruct n as[ |n'] eqn: E.
```

Listing 29: `destruct`

1. The expression `[ |n']` between `as` and `eqn` is called the *intro pattern*. It is a list of lists of names separated by `|`.
2. It can be refereed to any data type in the intro pattern.
3. In the above case the first entry of the list is empty, because the `O`-constructor is *nullary* and the constructor of `S` is *unary*, therefore it is written `n'` as the second member of our list.

4. `E` is the variable for the equation in the following.

5. In general every subgoal, which follows the destruct, is going to be marked by -.

The bullets are not necessarily to list. Coq asks to show every listed subgoal following a `destruct` in sequence. But due to readability and clearance, guarantee of correctness and convenience while debugging, sub goals should be listed.
Moreover, there are no hard or fast rules in proof formatting (i.e., indents and linebreaks). Bullets in the beginning of the line foster readability and limiting the number of characters per line to 80 aims readability.
The tactic `destruct` can be applied to any inductively defined data type (e.g. `bool`).

Example 8:
This a first example of the usage of bullets within the `destruct` *scope:*

```
Theorem newb_involutive: forall b: bool,
  neg b ( neg b ) = b.

Proof.
  intros b. destruct b eq n: E.
    - reflexivity.
    - reflexivity.
Qed.
```

Listing 30: `newb_involutive`

Note that in the Listing 30 a name specification is not required, because there is no `as` clause in the `destruct`. Since no variable has to be bounded by the use of the tactics, Listing empty lists as `[]` or `[|]` would be bad style and lead to a confusing choice by Coq.
`destruct` is able to be invoked inside subgoals to generate more proof obligations. The nested subgoals can either be grouped by an addition sign `+`, the so called asterix (`*`), or a curly parenthesis (`{}`). In particular the curly parenthesis is applicable if more then three levels of subgoals are required.

Example 9:

```
destruct a eqn: EqnA.
  + reflexivity.
  + destruct b eqn: EqnB.
    - reflexivity.
    - destruct c eqn: EqnC.
        * simpl.
        (* ... *)

 destruct d eqn: EqnD.
  { reflectivity.
    destruct e eqn: EqnE. }
     { reflexivity.
       destruct f eqn: EqnF. }
         { simpl. }
         (* ... *)
```

Listing 31: syntax of nested `destruct` expressions

To use a case analysis right after introducing variables it is written:

```
intros y. destruct y as [  |y] eqn:E.
```

Listing 32: `intros` and `destruct`

This is able to be shortened to the expression in Listing 33.

```
intros[ |y].
```

Listing 33: shortform `intros` and `destruct`

Note that the equation recording assumption is lost in trade of shortage. Moreover, if there is no necessity to name arguments, they can be replaced by `[]` (see Listing 34).

```
Theorem andb_commutative:
  forall b, c andb b c = andb c b.
  Proof.
    intros [] [].
      - reflexivity.
      - reflexivity.
      - reflexivity.
      - reflexivity.
  Qed.
```

Listing 34: `andb_commutative`

3. Importing

In order to use the definitions from the previous chapter we are going to compile the file `Basics.v` and import the compiles version in the current file `Induction.v`. (In case the book [PdAC+19] is used and downloaded as an archive, the following steps can be skipped.)

3.1. Building Coq Libraries

First of all, a Coq project, called *_CoqProject*, is created. It is going to map the current directory "`.`" to the directory `lf` wherein the source files are kept. Using the CoqIDE, Proof General or executing Coq via the command line the compiling and building process differs. The Proof General using reader is refereed to the literature [PdAC+19, Section, Induction, Proof by Induction].

Create a file called `_coqProject` within the working directory which contains the file `Basics.v`. Within this file add the line

```
-Q .LF
```

Listing 35: `naming a library`

which declares the library's name `LF.Basics`. To make the executable `Basics.vo`-file out of the `Basics.v` multiple options exist.
Using the CoqIDE, the file `Basics.v` should be opened, and the button in the compile menu "*Compile Buffer*" is clicked. If using the command line the file `Basics.vo` can be build or compiled using the Coq make-file utility. By the way, the *Coq-makefile utility* is installed together with Coq. Type the following console-command:

```
coq_makefile -f _CoqProject *.v -o Makefile
```

Listing 36: `coq-makefile`

In addition run this command, whenever files to the working directory `lf` were added or removed. To compile `Basics.v`, call either on of the following commands

```
make (* builds the complete directory *)
make Basics.vo (* builds Basics.vo *)
```

Listing 37: `make`

Note that, `make` compiles and calculates dependencies automatically. The Coq-compiler, which is called *coqc* can be called by:

```
coqc -Q.LF Basics.v
```

Listing 38: `coqc`

But remember always to prefer `make` over compiling, because of the dependencies.

To include the compiled chapter `Basics.vo` into the source code of the chapter `Induction.v` it is written

```
From LF Require Export Basics.
```

Listing 39: `Require Export`

into the first line of the `Induction.v`-file.

3.2. Potential Troubles

If an error arises which complains about missing identifiers in the file the ”load path” for Coq might not be set up correctly. Checking the loaded path by the command

```
Print LoadPath.
```

Listing 40: checking the loaded path

might be helpful. The error message

```
  Compiled library Foo makes inconsistent assumptions over library Bar
```

Listing 41: possible version error

might be generated, because incompatible versions of Coq are installed on a machine. In particular, CoqIDE and Proof General can use different versions of Coq for compiling. To resolve these issues build the files again.

Receiving the error message

```
Unable to locate library Basics.
```

Listing 42: possible compiling error

might be caused if two libraries are dependent and `Bar` was modified and compiled singly. To overcome this issue, build `Basics.vo` again. In case to many files are affected build everything again.

```
Make clean, make
```

Listing 43: rebuilding libraries

Using the CoqIDE and running `coqc` in the command line might cause inconsistency. A workaround of this issue is always using the *make* button from the CoqIDE and never calling the compiler directly.

4. Induction

In this section the tactics `induction`, which is applicable analogously to the mathematical principle of induction, is introduced.

4.1. Proof by Induction

In the proof in listing 23 it was shown, that `0` is the neutral element with respect to `+` from the left of the group of natural numbers `nat`. It should be shown that `0` also is the natural element for `+` in `nat` from the right. This writes as:

```
Theorem plus_n_0_first: forall n: nat,
  n = n + 0.
```

Listing 44: `plus_n_0_first`

(If the reader is not familiar with groups it is not sufficient to understand the algebraic nature of a group. Thus, it can be refereed to [Rei] for an introduction to groups.) But the tactics which were introduced till now, are not sufficiently powerful to proof this theorem. And applying the tactics `simpl` yields no results, too (see listing 46).

```
Theorem plus_n_0_firsttry : forall n:nat,
  n = n + 0.
```

Listing 45: `plus_n_0_firsttry`

Applying reflexivity can not proof the theorem. In fact, simplifying the expression `n + 0` leads to nowhere. Because looking at the definition of `plus`, it becomes obvious. If `n` is an unknown number, the `match` can not be applied.

```
Proof.
  intros n.
  simpl. (* Does nothing. *)
Abort.
```

Listing 46: `Proof` of `plus_n_0_firsttry`

Furthermore, proofing by the `destruct` tactic is going to fail, because in the case `n = Sn'` the expression `S n' = S n' + 0` can not be simplified by the same reason as above (see listing 47).

```
Theorem plus_1_new_0: forall n : nat,
  (n+1) =? 0 = false.

Proof.
  intros n. destruct n as [| n'] eqn:E.
  - (* n = 0 *)
    reflexivity.
  - (* n = S n' *)
    simpl.       (* Does nothing. *)
Abort.
```

Listing 47: `plus_1_new_0`

Recall the mathematical principle of induction. (E.g. see Appendix E.6.) Due to apply induction in Coq the steps are the same and the syntax is similar to the `destruct` tactics.

```
Theorem minus_diag: forall n:nat,
  minus n n = 0.

Proof.
  intros n. induction n as [| n' IHn'].
  - (* case: n = 0 *)
    simpl. reflexivity.
  - (* case: n = S n' *)
    simpl. rewrite →IHn'. reflexivity.
  Qed.
```

Listing 48: `minus_diag`

The `as`-clause has two parts separated by `|`.

- In the above statement of `induction` the first subgoal is to show the induction basis for `n=0`.
- And the second subgoal is the induction step for `n = Sn'`, since `nat` is defined inductivly (see listing 13). The assumption `n'+0 = n'` is added to Coq's context named as `IHn'` as induction hypothesis. It must be shown `Sn' = Sn' + 0`.
  Applying `simpl` yields `Sn' = S(n'+0)`, which follows from the induction hypothesis. Hence, applying `reflexivity` finishes this proof.

4.2. Proofs Within Proofs

In Coq and *formal mathematics* large proofs are often broken into a sequence of theorems and later proofs might refer to more early stated proofs. Sometimes a proof requires miscellaneous trivial or to little general facts. Therefore the facts sometimes do not need a name.
To state intermediate goals like a little sub theorem during proofs the tactics `assert` is used.

Example 10: *We can show the previous theorem* `mult_0_plus`*. This is an example using* `assert`*.*

```
1 Theorem mult_0_plus_prime: forall n m: nat,
2   (0 + n) * m = n * m.
3 Proof.
4   intros n m.
5   assert (H: 0 + n = n).
6     reflexivity.
7     rewrite →H.
8   reflexivity.
9 Qed.
```

Listing 49: `mult_0_plus'`

In listing 49 the tactics `assert` in line 5 introduces two subgoals. The assertion itself is listed with `H` the introduction as prefix. The proof of the assertion is bounded by curly parenthesis.
It provides reducibility and interactive use of Coq it is more easy to see when the proof of the first subgoal is finished.
The second subgoal is the same as the subgoal in the proof before the subgoal exact of in the context the assumption `H` was added. But the previously proven fact is able to be used to make progress. Let's look at another example.

It should be shown that we have for all natural numbers `n, m` ∈ `nat`: `(n+m)+ (p+q)= (m+n)+ (p+q)`. But the tactics is not going to be applied, in the most useful way. If `rewrite` is applied it is going to act on the wrong plus sign (i.e. the summand `n+m` is switched with the `p+q`).

```
Theorem plus_rearrange_firsttry : forall n m p q: nat,
  (n + m) + (p + q) = (m + n) + (p + q).
Proof.
  intros n m p q.
  (* We just need to swap (n + m) for (m + n)... seems
     like plus_comm should do the trick! *)
  rewrite ->plus_comm.
  (* Doesn't work ... Coq rewrites the wrong plus! *)
Abort.
```

Listing 50: `plus_rearrange_firstttry`

This habit can be worked around by rewriting the wanted variables as in listing 51 using the tactics `assert`.

```
Theorem plus_rearrange : forall n m p q: nat,
  (n + m) + (p + q) = (m + n) + (p + q).
Proof.
  intros n m p q.
  assert (H: n + m = m + n).
  { rewrite ->plus_comm. reflexivity. }
  rewrite ->H.
  reflexivity.
Qed.
```

Listing 51: `plus_rearrange`

4.3. Formal vs. Informal Proofs

> 'Informal proofs are algorithms; Formal proofs are code.'

The question about a mathematical proof has challenged mathematics and philosophy for millennia. There is a discourse about formal and informal proofs in mathematics. An informal proof is a proof that is a proof which is written in some natural language (e.g. English). It guides a human reader. One might state that a proof of a mathematical proposition P is written or spoken text, which the reader or hearer that P is true. Because it is worked with Coq in this scope, it is heavily worked with formal proofs. That is, it is preconditioned that the proof can be methodically derived from a set of formal logical rules and that the proof can be methodically derived from a certain set of formal logic rules. Moreover, a proof is a list which guides the program in checking. But due to readability for the human reader, a proof is structured by comments and bullets. Note that Coq has been designed in such a way that its induction hypothesis generates the same subgoal in the same order a mathematician would write an inductive proof.

5. Lists - Working with structured Data

5.1. Pairs of Numbers

Using an inductive type definition each constructor can take any number of arguments. Therefore, let's make a pair of numbers.

```
Inductive natprod: Type:=
 | pair (n1 n2: nat).

check(pair 3,5)
```

Listing 52: `natprod`

By the decalaration in this listing (52) there is only one way to construct a pair of numbers; Applying the constructor `pair` to two argumens of type `nat`. Some simple function on a pair might be given by:

```
Definition fst( p: natprod): nat :=
  match p with
    | pair xy ⇒x (* return 1st component)
  end.

 Definition snd ( p: natprod): nat :=
   match p with
     | pair xy ⇒y (* return 2nd component*)
   end.
```

Listing 53: `fst`

And an abbericvation such that the mathematical standart notation can be used is implemented by:

```
Notation "(x,y)" = (pair xy).
```

Listing 54: `pair`-naotation

This notation shall be used in pattern matches and expressions as:

```
Definiton fst'(p:natprod):nat:=
match p with
  |(x,y) ⇒x
end.

Definition snd' (p:natprod):nat :=
match p with
  |(x,y) ⇒y
end.
```

Listing 55: `fst'` and `snd'`

5.2. Introduction to RT-Proofs

This section contains listings from the working directory found at git@gitlab.cs.hs-rm.de:almeroth/prosa_working_dir.git. They show the most important and modified parts of the Coq-code form the RT-Proofs clone found in this repository. The clone is from the artifact evaluation form this work [CSB16] an can be found here [PROb]. Note that the style definitions for Coq-code from [Gia] were used in the listings of this work to pretify this LaTeX-document. Formal Proofs for real-time systems (RT-Proofs) is a project running between multiple research faculties. The PROSA library is where this development takes place. The output of the RT-Proofs project is published here https://prosa.mpi-sws.org/.

The Repository This repository is sought to be ment for self-teaching somebody to work with the proof assistant Coq and the PROSA -Frame-Work.

```
**Eine Einführung in Coq **



[IntroductionToCoq.pdf](https://gitlab.cs.hs-rm.de/almeroth/softwarefoundations/blob/master/IntroductionT
    sind die ".pdf"-Seiten einer Zusammenfassung über [Coq](https://coq.inria.fr/).
Coq ist eine Software zur semi-automatischen formalen Beweisführung.

Diese Einführung verwendet das Skript 'mbasics' von [Steffen
    Reith](https://www.hs-rm.de/de/hochschule/personen/reith-steffen/) über eine
    Einführung in Mathematische Grundlagen für Studierende der Informatik.



Zum clonen des Skriptes sei auf Git-Submodule verwiesen.

    [git-Online-Dokumentation](https://git-scm.com/book/en/v2/Git-Tools-Submodules)


Die [Latex2_{\epsilon}](https://www.latex-project.org/)-Dateien können wie folgt
    gebaut werden.

[IntroductionToCoq.tex](https://gitlab.cs.hs-rm.de/almeroth/softwarefoundations/blob/master/IntroductionT
     ist die Latex-Master-Datei und das Dokument kann gebaut werden mittels den
    folgenden Komandos:

    xelatexIntroductionToCoq.tex bibtex IntroductionToCoq.tex
    xelatexIntroductionToCoq.tex bibtex IntroductionToCoq.tex

```

Listing 56: README.md, Source: [CSB16], modified.

5.3. Implementation In Coq

These are listings from the working directory found at git@gitlab.cs.hs-rm.de:almeroth/prosa_working_dir.git/basic.

The Schedule

```
Require Import rt.util.all.
Require Import rt.model.basic.job rt.model.basic.arrival_sequence
    rt.model.basic.schedule
               rt.model.basic.platform rt.model.basic.priority.
From mathcomp Require Import ssreflect ssrbool ssrfun eqtype ssrnat fintype bigop
    seq path.

```

```
Module ConcreteScheduler.

  Import Job ArrivalSequence Schedule Platform Priority.

  Section Implementation.

    Context {Job: eqType}.
    Variable job_cost: Job → time.

    (* Let num_cpus denote the number of processors, ...*)
    Variable num_cpus: nat.

    (* ... and let arr_seq be any arrival sequence.*)
    Variable arr_seq: arrival_sequence Job.

    (* Assume a JLDP policy is given. *)
    Variable higher_eq_priority: JLDP_policy arr_seq.

    (* Consider the list of pending jobs at time t. *)
    Definition jobs_pending_at (sched: schedule num_cpus arr_seq) (t: time) :=
      [seq j ← jobs_arrived_up_to arr_seq t | pending job_cost sched j t].

    (* Next, we sort this list by priority. *)
    Definition sorted_pending_jobs (sched: schedule num_cpus arr_seq) (t: time) :=
      sort (higher_eq_priority t) (jobs_pending_at sched t).

    (* Starting from the empty schedule as a base, ... *)
    Definition empty_schedule : schedule num_cpus arr_seq :=
      fun t cpu ⇒ None.

    (* ..., we redefine the mapping of jobs to processors at any time t as follows.
       The i-th job in the sorted list is assigned to the i-th cpu, or to None
       if the list is short. *)
    Definition update_schedule (prev_sched: schedule num_cpus arr_seq)
                               (t_next: time) : schedule num_cpus arr_seq :=
      fun cpu t ⇒
        if t == t_next then
          nth_or_none (sorted_pending_jobs prev_sched t) cpu
        else prev_sched cpu t.

    (* The schedule is iteratively constructed by applying assign_jobs at every time
        t, ... *)
    Fixpoint schedule_prefix (t_max: time) : schedule num_cpus arr_seq :=
      if t_max is t_prev.+1 then
        (* At time t_prev + 1, schedule jobs that have not completed by time t_prev.
            *)
        update_schedule (schedule_prefix t_prev) t_prev.+1
      else
        (* At time 0, schedule any jobs that arrive. *)
        update_schedule empty_schedule 0.

    Definition scheduler (cpu: processor num_cpus) (t: time) := (schedule_prefix t)
        cpu t.

  End Implementation.

  Section Proofs.

    Context {Job: eqType}.
    Variable job_cost: Job → time.

    (* Assume a positive number of processors. *)
    Variable num_cpus: nat.
    Hypothesis H_at_least_one_cpu: num_cpus > 0.

    (* Let arr_seq be any arrival sequence of jobs where ...*)
    Variable arr_seq: arrival_sequence Job.
    (* ...jobs have positive cost and...*)
    Hypothesis H_job_cost_positive:
      forall (j: JobIn arr_seq), job_cost_positive job_cost j.
    (* ... at any time, there are no duplicates of the same job. *)
    Hypothesis H_arrival_sequence_is_a_set :
```

```
  arrival_sequence_is_a_set arr_seq.

(* Consider any JLDP policy higher_eq_priority that is transitive and total. *)
Variable higher_eq_priority: JLDP_policy arr_seq.
Hypothesis H_priority_transitive: forall t, transitive (higher_eq_priority t).
Hypothesis H_priority_total: forall t, total (higher_eq_priority t).

(* Let sched denote our concrete scheduler implementation. *)
Let sched := scheduler job_cost num_cpus arr_seq higher_eq_priority.

(* Next, we provide some helper lemmas about the scheduler construction. *)
Section HelperLemmas.

  (* First, we show that the scheduler preserves its prefixes. *)
  Lemma scheduler_same_prefix :
    forall t t_max cpu,
      t <= t_max ->
      schedule_prefix job_cost num_cpus arr_seq higher_eq_priority t_max cpu t =
      scheduler job_cost num_cpus arr_seq higher_eq_priority cpu t.
  Proof.
    intros t t_max cpu LEt.
    induction t_max.
    {
      by rewrite leqn0 in LEt; move: LEt => /eqP EQ; subst.
    }
    {
      rewrite leq_eqVlt in LEt.
      move: LEt => /orP [/eqP EQ | LESS]; first by subst.
      {
        feed IHt_max; first by done.
        unfold schedule_prefix, update_schedule at 1.
        assert (FALSE: t ==t_max.+1 = false).
        {
          by apply negbTE; rewrite neq_ltn LESS orTb.
        } rewrite FALSE.
        by rewrite -IHt_max.
      }
    }
  Qed.

  (* With respect to the sorted list of pending jobs, ...*)
  Let sorted_jobs (t: time) :=
    sorted_pending_jobs job_cost num_cpus arr_seq higher_eq_priority sched t.

  (* ..., we show that a job is mapped to a processor based on that list, ... *)
  Lemma scheduler_nth_or_none_mapping :
    forall t cpu x,
      sched cpu t = x ->
      nth_or_none (sorted_jobs t) cpu = x.
  Proof.
    intros t cpu x SCHED.
    unfold sched, scheduler, schedule_prefix in *.
    destruct t.
    {
      unfold update_schedule in SCHED; rewrite eq_refl in SCHED.
      rewrite -SCHED; f_equal.
      unfold sorted_jobs, sorted_pending_jobs; f_equal.
      unfold jobs_pending_at; apply eq_filter; red; intro j'.
      unfold pending; f_equal; f_equal.
      unfold completed, service.
      by rewrite big_geq // big_geq //.
    }
    {
      unfold update_schedule at 1 in SCHED; rewrite eq_refl in SCHED.
      rewrite -SCHED; f_equal.
      unfold sorted_jobs, sorted_pending_jobs; f_equal.
      unfold jobs_pending_at; apply eq_filter; red; intro j'.
      unfold pending; f_equal; f_equal.
      unfold completed, service; f_equal.
      apply eq_big_nat; move => t0 /andP [_ LT].
      unfold service_at; apply eq_bigl; red; intros cpu'.
      fold (schedule_prefix job_cost num_cpus arr_seq higher_eq_priority).
```

```
      by rewrite /scheduled_on 2?scheduler_same_prefix ?leqnn //.
    }
  Qed.

  (* ..., a scheduled job is mapped to a cpu corresponding to its position, ... *)
  Lemma scheduler_nth_or_none_scheduled :
    forall j t,
      scheduled sched j t ->
      exists (cpu: processor num_cpus),
        nth_or_none (sorted_jobs t) cpu = Some j.
  Proof.
    intros j t SCHED.
    move: SCHED => /existsP [cpu /eqP SCHED]; exists cpu.
    by apply scheduler_nth_or_none_mapping.
  Qed.

  (* ..., and that a backlogged job has a position larger than or equal to the number
     of processors. *)
  Lemma scheduler_nth_or_none_backlogged :
    forall j t,
      backlogged job_cost sched j t ->
      exists i,
        nth_or_none (sorted_jobs t) i = Some j /\ i >= num_cpus.
  Proof.
    intros j t BACK.
    move: BACK => /andP [PENDING /negP NOTCOMP].
    assert (IN: j \in sorted_jobs t).
    {
      rewrite mem_sort mem_filter PENDING andTb.
      move: PENDING => /andP [ARRIVED _].
      by rewrite JobIn_has_arrived.
    }
    apply nth_or_none_mem_exists in IN; des.
    exists n; split; first by done.
    rewrite leqNgt; apply/negP; red; intro LT.
    apply NOTCOMP; clear NOTCOMP PENDING.
    apply/existsP; exists (Ordinal LT); apply/eqP.
    unfold sorted_jobs in *; clear sorted_jobs.
    unfold sched, scheduler, schedule_prefix in *; clear sched.
    destruct t.
    {
      unfold update_schedule; rewrite eq_refl.
      rewrite -IN; f_equal.
      fold (schedule_prefix job_cost num_cpus arr_seq higher_eq_priority).
      unfold sorted_pending_jobs; f_equal.
      apply eq_filter; red; intros x.
      unfold pending; f_equal; f_equal.
      unfold completed; f_equal.
      by unfold service; rewrite 2?big_geq //.
    }
    {
      unfold update_schedule at 1; rewrite eq_refl.
      rewrite -IN; f_equal.
      unfold sorted_pending_jobs; f_equal.
      apply eq_filter; red; intros x.
      unfold pending; f_equal; f_equal.
      unfold completed; f_equal.
      unfold service; apply eq_big_nat; move => i /andP [_ LTi].
      unfold service_at; apply eq_bigl; red; intro cpu.
      unfold scheduled_on; f_equal.
      fold (schedule_prefix job_cost num_cpus arr_seq higher_eq_priority).
      by rewrite scheduler_same_prefix.
    }
  Qed.

End HelperLemmas.

(* Now, we prove the important properties about the implementation. *)

(* Jobs do not execute before they arrive, ...*)
```

```
Theorem scheduler_jobs_must_arrive_to_execute:
  jobs_must_arrive_to_execute sched.
Proof.
  unfold jobs_must_arrive_to_execute.
  intros j t SCHED.
  move: SCHED => /existsP [cpu /eqP SCHED].
  unfold sched, scheduler, schedule_prefix in SCHED.
  destruct t.
  {
    rewrite /update_schedule eq_refl in SCHED.
    apply (nth_or_none_mem _ cpu j) in SCHED.
    rewrite mem_sort mem_filter in SCHED.
    move: SCHED => /andP [_ ARR].
    by apply JobIn_has_arrived in ARR.
  }
  {
    unfold update_schedule at 1 in SCHED; rewrite eq_refl /= in SCHED.
    apply (nth_or_none_mem _ cpu j) in SCHED.
    rewrite mem_sort mem_filter in SCHED.
    move: SCHED => /andP [_ ARR].
    by apply JobIn_has_arrived in ARR.
  }
Qed.

(* ..., jobs are sequential, ... *)
Theorem scheduler_sequential_jobs: sequential_jobs sched.
Proof.
  unfold sequential_jobs, sched, scheduler, schedule_prefix.
  intros j t cpu1 cpu2 SCHED1 SCHED2.
  destruct t; rewrite /update_schedule eq_refl in SCHED1 SCHED2;
  have UNIQ := nth_or_none_uniq _ cpu1 cpu2 j _ SCHED1 SCHED2; (apply ord_inj,
      UNIQ);
  rewrite sort_uniq filter_uniq //;
  by apply JobIn_uniq.
Qed.

(* ... and jobs do not execute after completion. *)
Theorem scheduler_completed_jobs_dont_execute:
  completed_jobs_dont_execute job_cost sched.
Proof.
  rename H_job_cost_positive into GT0.
  unfold completed_jobs_dont_execute, service.
  intros j t.
  induction t; first by rewrite big_geq.
  {
    rewrite big_nat_recr // /=.
    rewrite leq_eqVlt in IHt; move: IHt => /orP [/eqP EQ | LESS]; last first.
    {
      destruct (job_cost j); first by rewrite ltn0 in LESS.
      rewrite -addn1; rewrite ltnS in LESS.
      apply leq_add; first by done.
      by apply service_at_most_one, scheduler_sequential_jobs.
    }
    rewrite EQ -{2}[job_cost j]addn0; apply leq_add; first by done.
    destruct t.
    {
      rewrite big_geq // in EQ.
      specialize (GT0 j); unfold job_cost_positive in *.
      by rewrite -EQ ltn0 in GT0.
    }
    {
      unfold service_at; rewrite big_mkcond.
      apply leq_trans with (n := \sum_(cpu < num_cpus) 0);
        last by rewrite big_const_ord iter_addn mul0n addn0.
      apply leq_sum; intros cpu _; desf.
      move: Heq => /eqP SCHED.
      unfold scheduler, schedule_prefix in SCHED.
      unfold sched, scheduler, schedule_prefix, update_schedule at 1 in SCHED.
      rewrite eq_refl in SCHED.
      apply (nth_or_none_mem _ cpu j) in SCHED.
      rewrite mem_sort mem_filter in SCHED.
```

```
     fold (update_schedule job_cost num_cpus arr_seq higher_eq_priority) in SCHED.
     move: SCHED => /andP [/andP [_ /negP NOTCOMP] _].
     exfalso; apply NOTCOMP; clear NOTCOMP.
     unfold completed; apply/eqP.
     unfold service; rewrite -EQ.
     rewrite big_nat_cond [\sum_(_ <= _ < _ | true)_]big_nat_cond.
     apply eq_bigr; move => i /andP [/andP [_ LT] _].
     apply eq_bigl; red; ins.
     unfold scheduled_on; f_equal.
     fold (schedule_prefix job_cost num_cpus arr_seq higher_eq_priority).
     by rewrite scheduler_same_prefix.
    }
   }
  Qed.

  (* In addition, the scheduler is work conserving ... *)
  Theorem scheduler_work_conserving:
   work_conserving job_cost sched.
  Proof.
   unfold work_conserving; intros j t BACK cpu.
   set jobs := sorted_pending_jobs job_cost num_cpus arr_seq higher_eq_priority sched t.
   destruct (sched cpu t) eqn:SCHED; first by exists j0; apply/eqP.
   apply scheduler_nth_or_none_backlogged in BACK.
   destruct BACK as [cpu_out [NTH GE]].
   exfalso; rewrite leqNgt in GE; move: GE => /negP GE; apply GE.
   apply leq_ltn_trans with (n := cpu); last by done.
   apply scheduler_nth_or_none_mapping in SCHED.
   apply nth_or_none_size_none in SCHED.
   apply leq_trans with (n := size jobs); last by done.
   by apply nth_or_none_size_some in NTH; apply ltnW.
  Qed.

  (* ... and enforces the JLDP policy. *)
  Theorem scheduler_enforces_policy :
   enforces_JLDP_policy job_cost sched higher_eq_priority.
  Proof.
   unfold enforces_JLDP_policy; intros j j_hp t BACK SCHED.
   set jobs := sorted_pending_jobs job_cost num_cpus arr_seq higher_eq_priority sched t.
   apply scheduler_nth_or_none_backlogged in BACK.
   destruct BACK as [cpu_out [SOME GE]].
   apply scheduler_nth_or_none_scheduled in SCHED.
   destruct SCHED as [cpu SCHED].
   have EQ1 := nth_or_none_nth jobs cpu j_hp j SCHED.
   have EQ2 := nth_or_none_nth jobs cpu_out j j SOME.
   rewrite -EQ1 -{2}EQ2.
   apply sorted_lt_idx_implies_rel; [by done | by apply sort_sorted | ].
   - by apply leq_trans with (n := num_cpus).
   - by apply nth_or_none_size_some in SOME.
  Qed.

 End Proofs.

End ConcreteScheduler.
```

Listing 57: `schedule.v`

```
Require Import rt.util.all.
Require Import rt.model.basic.job rt.model.basic.arrival_sequence rt.model.basic.schedule
           rt.model.basic.platform rt.model.basic.priority.
From mathcomp Require Import ssreflect ssrbool ssrfun eqtype ssrnat fintype bigop seq path.

Module ConcreteScheduler.

 Import Job ArrivalSequence Schedule Platform Priority.
```

```

Section Implementation.

  Context {Job: eqType}.
  Variable job_cost: Job →time.

  (* Let num_cpus denote the number of processors, ...*)
  Variable num_cpus: nat.

  (* ... and let arr_seq be any arrival sequence.*)
  Variable arr_seq: arrival_sequence Job.

  (* Assume a JLDP policy is given. *)
  Variable higher_eq_priority: JLDP_policy arr_seq.

  (* Consider the list of pending jobs at time t. *)
  Definition jobs_pending_at (sched: schedule num_cpus arr_seq) (t: time) :=
    [seq j ←jobs_arrived_up_to arr_seq t | pending job_cost sched j t].

  (* Next, we sort this list by priority. *)
  Definition sorted_pending_jobs (sched: schedule num_cpus arr_seq) (t: time) :=
    sort (higher_eq_priority t) (jobs_pending_at sched t).

  (* Starting from the empty schedule as a base, ... *)
  Definition empty_schedule : schedule num_cpus arr_seq :=
    fun t cpu ⇒None.

  (* ..., we redefine the mapping of jobs to processors at any time t as follows.
     The i-th job in the sorted list is assigned to the i-th cpu, or to None
     if the list is short. *)
  Definition update_schedule (prev_sched: schedule num_cpus arr_seq)
                             (t_next: time) : schedule num_cpus arr_seq :=
    fun cpu t ⇒
      if t ==t_next then
        nth_or_none (sorted_pending_jobs prev_sched t) cpu
      else prev_sched cpu t.

  (* The schedule is iteratively constructed by applying assign_jobs at every time
      t, ... *)
  Fixpoint schedule_prefix (t_max: time) : schedule num_cpus arr_seq :=
    if t_max is t_prev.+1 then
      (* At time t_prev + 1, schedule jobs that have not completed by time t_prev.
          *)
      update_schedule (schedule_prefix t_prev) t_prev.+1
    else
      (* At time 0, schedule any jobs that arrive. *)
      update_schedule empty_schedule 0.

  Definition scheduler (cpu: processor num_cpus) (t: time) := (schedule_prefix t)
      cpu t.

End Implementation.

Section Proofs.

  Context {Job: eqType}.
  Variable job_cost: Job →time.

  (* Assume a positive number of processors. *)
  Variable num_cpus: nat.
  Hypothesis H_at_least_one_cpu: num_cpus > 0.

  (* Let arr_seq be any arrival sequence of jobs where ...*)
  Variable arr_seq: arrival_sequence Job.
  (* ...jobs have positive cost and...*)
  Hypothesis H_job_cost_positive:
    forall (j: JobIn arr_seq), job_cost_positive job_cost j.
  (* ... at any time, there are no duplicates of the same job. *)
  Hypothesis H_arrival_sequence_is_a_set :
    arrival_sequence_is_a_set arr_seq.

  (* Consider any JLDP policy higher_eq_priority that is transitive and total. *)
```

```
  Variable higher_eq_priority: JLDP_policy arr_seq.
  Hypothesis H_priority_transitive: forall t, transitive (higher_eq_priority t).
  Hypothesis H_priority_total: forall t, total (higher_eq_priority t).

  (* Let sched denote our concrete scheduler implementation. *)
  Let sched := scheduler job_cost num_cpus arr_seq higher_eq_priority.

  (* Next, we provide some helper lemmas about the scheduler construction. *)
  Section HelperLemmas.

    (* First, we show that the scheduler preserves its prefixes. *)
    Lemma scheduler_same_prefix :
      forall t t_max cpu,
        t <= t_max →
        schedule_prefix job_cost num_cpus arr_seq higher_eq_priority t_max cpu t =
        scheduler job_cost num_cpus arr_seq higher_eq_priority cpu t.
    Proof.
      intros t t_max cpu LEt.
      induction t_max.
      {
        by rewrite leqn0 in LEt; move: LEt ⇒/eqP EQ; subst.
      }
      {
        rewrite leq_eqVlt in LEt.
        move: LEt ⇒/orP [/eqP EQ | LESS]; first by subst.
        {
          feed IHt_max; first by done.
          unfold schedule_prefix, update_schedule at 1.
          assert (FALSE: t ==t_max.+1 = false).
          {
            by apply negbTE; rewrite neq_ltn LESS orTb.
          } rewrite FALSE.
          by rewrite -IHt_max.
        }
      }
    Qed.

    (* With respect to the sorted list of pending jobs, ...*)
    Let sorted_jobs (t: time) :=
      sorted_pending_jobs job_cost num_cpus arr_seq higher_eq_priority sched t.

    (* ..., we show that a job is mapped to a processor based on that list, ... *)
    Lemma scheduler_nth_or_none_mapping :
      forall t cpu x,
        sched cpu t = x →
        nth_or_none (sorted_jobs t) cpu = x.
    Proof.
      intros t cpu x SCHED.
      unfold sched, scheduler, schedule_prefix in *.
      destruct t.
      {
        unfold update_schedule in SCHED; rewrite eq_refl in SCHED.
        rewrite -SCHED; f_equal.
        unfold sorted_jobs, sorted_pending_jobs; f_equal.
        unfold jobs_pending_at; apply eq_filter; red; intro j'.
        unfold pending; f_equal; f_equal.
        unfold completed, service.
        by rewrite big_geq // big_geq //.
      }
      {
        unfold update_schedule at 1 in SCHED; rewrite eq_refl in SCHED.
        rewrite -SCHED; f_equal.
        unfold sorted_jobs, sorted_pending_jobs; f_equal.
        unfold jobs_pending_at; apply eq_filter; red; intro j'.
        unfold pending; f_equal; f_equal.
        unfold completed, service; f_equal.
        apply eq_big_nat; move ⇒t0 /andP [_ LT].
        unfold service_at; apply eq_bigl; red; intros cpu'.
        fold (schedule_prefix job_cost num_cpus arr_seq higher_eq_priority).
        by rewrite /scheduled_on 2?scheduler_same_prefix ?leqnn //.
      }
    Qed.
```

```

    (* ..., a scheduled job is mapped to a cpu corresponding to its position, ... *)
    Lemma scheduler_nth_or_none_scheduled :
      forall j t,
        scheduled sched j t →
        exists (cpu: processor num_cpus),
          nth_or_none (sorted_jobs t) cpu = Some j.
    Proof.
      intros j t SCHED.
      move: SCHED ⇒/existsP [cpu /eqP SCHED]; exists cpu.
      by apply scheduler_nth_or_none_mapping.
    Qed.

    (* ..., and that a backlogged job has a position larger than or equal to the number
       of processors. *)
    Lemma scheduler_nth_or_none_backlogged :
      forall j t,
        backlogged job_cost sched j t →
        exists i,
          nth_or_none (sorted_jobs t) i = Some j ∧ i >= num_cpus.
    Proof.
      intros j t BACK.
      move: BACK ⇒/andP [PENDING /negP NOTCOMP].
      assert (IN: j ∈ sorted_jobs t).
      {
        rewrite mem_sort mem_filter PENDING andTb.
        move: PENDING ⇒/andP [ARRIVED _].
        by rewrite JobIn_has_arrived.
      }
      apply nth_or_none_mem_exists in IN; des.
      exists n; split; first by done.
      rewrite leqNgt; apply/negP; red; intro LT.
      apply NOTCOMP; clear NOTCOMP PENDING.
      apply/existsP; exists (Ordinal LT); apply/eqP.
      unfold sorted_jobs in *; clear sorted_jobs.
      unfold sched, scheduler, schedule_prefix in *; clear sched.
      destruct t.
      {
        unfold update_schedule; rewrite eq_refl.
        rewrite -IN; f_equal.
        fold (schedule_prefix job_cost num_cpus arr_seq higher_eq_priority).
        unfold sorted_pending_jobs; f_equal.
        apply eq_filter; red; intros x.
        unfold pending; f_equal; f_equal.
        unfold completed; f_equal.
        by unfold service; rewrite 2?big_geq //.
      }
      {
        unfold update_schedule at 1; rewrite eq_refl.
        rewrite -IN; f_equal.
        unfold sorted_pending_jobs; f_equal.
        apply eq_filter; red; intros x.
        unfold pending; f_equal; f_equal.
        unfold completed; f_equal.
        unfold service; apply eq_big_nat; move ⇒i /andP [_ LTi].
        unfold service_at; apply eq_bigl; red; intro cpu.
        unfold scheduled_on; f_equal.
        fold (schedule_prefix job_cost num_cpus arr_seq higher_eq_priority).
        by rewrite scheduler_same_prefix.
      }
    Qed.

  End HelperLemmas.

  (* Now, we prove the important properties about the implementation. *)

  (* Jobs do not execute before they arrive, ...*)
  Theorem scheduler_jobs_must_arrive_to_execute:
    jobs_must_arrive_to_execute sched.
  Proof.
```

```
  unfold jobs_must_arrive_to_execute.
  intros j t SCHED.
  move: SCHED ⇒/existsP [cpu /eqP SCHED].
  unfold sched, scheduler, schedule_prefix in SCHED.
  destruct t.
  {
    rewrite /update_schedule eq_refl in SCHED.
    apply (nth_or_none_mem _ cpu j) in SCHED.
    rewrite mem_sort mem_filter in SCHED.
    move: SCHED ⇒/andP [_ ARR].
    by apply JobIn_has_arrived in ARR.
  }
  {
    unfold update_schedule at 1 in SCHED; rewrite eq_refl /= in SCHED.
    apply (nth_or_none_mem _ cpu j) in SCHED.
    rewrite mem_sort mem_filter in SCHED.
    move: SCHED ⇒/andP [_ ARR].
    by apply JobIn_has_arrived in ARR.
  }
Qed.

(* ..., jobs are sequential, ... *)
Theorem scheduler_sequential_jobs: sequential_jobs sched.
Proof.
  unfold sequential_jobs, sched, scheduler, schedule_prefix.
  intros j t cpu1 cpu2 SCHED1 SCHED2.
  destruct t; rewrite /update_schedule eq_refl in SCHED1 SCHED2;
  have UNIQ := nth_or_none_uniq _ cpu1 cpu2 j _ SCHED1 SCHED2; (apply ord_inj, UNIQ);
  rewrite sort_uniq filter_uniq //;
  by apply JobIn_uniq.
Qed.

(* ... and jobs do not execute after completion. *)
Theorem scheduler_completed_jobs_dont_execute:
  completed_jobs_dont_execute job_cost sched.
Proof.
  rename H_job_cost_positive into GT0.
  unfold completed_jobs_dont_execute, service.
  intros j t.
  induction t; first by rewrite big_geq.
  {
    rewrite big_nat_recr // /=.
    rewrite leq_eqVlt in IHt; move: IHt ⇒/orP [/eqP EQ | LESS]; last first.
    {
      destruct (job_cost j); first by rewrite ltn0 in LESS.
      rewrite -addn1; rewrite ltnS in LESS.
      apply leq_add; first by done.
      by apply service_at_most_one, scheduler_sequential_jobs.
    }
    rewrite EQ -{2}[job_cost j]addn0; apply leq_add; first by done.
    destruct t.
    {
      rewrite big_geq // in EQ.
      specialize (GT0 j); unfold job_cost_positive in *.
      by rewrite -EQ ltn0 in GT0.
    }
    {
      unfold service_at; rewrite big_mkcond.
      apply leq_trans with (n := \sum_(cpu < num_cpus) 0);
        last by rewrite big_const_ord iter_addn mul0n addn0.
      apply leq_sum; intros cpu _; desf.
      move: Heq ⇒/eqP SCHED.
      unfold scheduler, schedule_prefix in SCHED.
      unfold sched, scheduler, schedule_prefix, update_schedule at 1 in SCHED.
      rewrite eq_refl in SCHED.
      apply (nth_or_none_mem _ cpu j) in SCHED.
      rewrite mem_sort mem_filter in SCHED.
      fold (update_schedule job_cost num_cpus arr_seq higher_eq_priority) in SCHED.
      move: SCHED ⇒/andP [/andP [_ /negP NOTCOMP] _].
      exfalso; apply NOTCOMP; clear NOTCOMP.
```

```
          unfold completed; apply/eqP.
          unfold service; rewrite -EQ.
          rewrite big_nat_cond [\sum_(_ <= _ < _ | true)_]big_nat_cond.
          apply eq_bigr; move => i /andP [/andP [_ LT] _].
          apply eq_bigl; red; ins.
          unfold scheduled_on; f_equal.
          fold (schedule_prefix job_cost num_cpus arr_seq higher_eq_priority).
          by rewrite scheduler_same_prefix.
        }
      }
    Qed.

    (* In addition, the scheduler is work conserving ... *)
    Theorem scheduler_work_conserving:
      work_conserving job_cost sched.
    Proof.
      unfold work_conserving; intros j t BACK cpu.
      set jobs := sorted_pending_jobs job_cost num_cpus arr_seq higher_eq_priority
          sched t.
      destruct (sched cpu t) eqn:SCHED; first by exists j0; apply/eqP.
      apply scheduler_nth_or_none_backlogged in BACK.
      destruct BACK as [cpu_out [NTH GE]].
      exfalso; rewrite leqNgt in GE; move: GE => /negP GE; apply GE.
      apply leq_ltn_trans with (n := cpu); last by done.
      apply scheduler_nth_or_none_mapping in SCHED.
      apply nth_or_none_size_none in SCHED.
      apply leq_trans with (n := size jobs); last by done.
      by apply nth_or_none_size_some in NTH; apply ltnW.
    Qed.

    (* ... and enforces the JLDP policy. *)
    Theorem scheduler_enforces_policy :
      enforces_JLDP_policy job_cost sched higher_eq_priority.
    Proof.
      unfold enforces_JLDP_policy; intros j j_hp t BACK SCHED.
      set jobs := sorted_pending_jobs job_cost num_cpus arr_seq higher_eq_priority
          sched t.
      apply scheduler_nth_or_none_backlogged in BACK.
      destruct BACK as [cpu_out [SOME GE]].
      apply scheduler_nth_or_none_scheduled in SCHED.
      destruct SCHED as [cpu SCHED].
      have EQ1 := nth_or_none_nth jobs cpu j_hp j SCHED.
      have EQ2 := nth_or_none_nth jobs cpu_out j j SOME.
      rewrite -EQ1 -{2}EQ2.
      apply sorted_lt_idx_implies_rel; [by done | by apply sort_sorted | |].
      - by apply leq_trans with (n := num_cpus).
      - by apply nth_or_none_size_some in SOME.
    Qed.

  End Proofs.

End ConcreteScheduler.
```

Listing 58: `schedule.v`, Source: [CSB16]

The Job

```
Require Import rt.model.basic.time rt.util.all.
Require Import rt.implementation.basic.task.
From mathcomp Require Import ssreflect ssrbool ssrnat eqtype seq.

Module ConcreteJob.

  Import Time.
  Import ConcreteTask.

  Section Defs.

    (* Definition of a concrete task. *)
```

```
    Record concrete_job :=
    {
      job_id: nat;
      job_cost: time;
      job_deadline: time;
      job_task: concrete_task_eqType
    }.

    (* To make it compatible with ssreflect, we define a decidable
       equality for concrete jobs. *)
    Definition job_eqdef (j1 j2: concrete_job) :=
      (job_id j1 == job_id j2) &&
      (job_cost j1 == job_cost j2) &&
      (job_deadline j1 == job_deadline j2) &&
      (job_task j1 == job_task j2).

    (* Next, we prove that job_eqdef is indeed an equality, ... *)
    Lemma eqn_job : Equality.axiom job_eqdef.
    Proof.
      unfold Equality.axiom; intros x y.
      destruct (job_eqdef x y) eqn:EQ.
      {
        apply ReflectT; unfold job_eqdef in *.
        move: EQ => /andP [/andP [/andP [/eqP ID /eqP COST] /eqP DL] /eqP TASK].
        by destruct x, y; simpl in *; subst.
      }
      {
        apply ReflectF.
        unfold job_eqdef, not in *; intro BUG.
        apply negbT in EQ; rewrite negb_and in EQ.
        destruct x, y.
        rewrite negb_and in EQ.
        move: EQ => /orP [EQ | /eqP TASK]; last by apply TASK; inversion BUG.
        move: EQ => /orP [EQ | /eqP DL].
        rewrite negb_and in EQ.
        move: EQ => /orP [/eqP ID | /eqP COST].
        by apply ID; inversion BUG.
        by apply COST; inversion BUG.
        by apply DL; inversion BUG.
      }
    Qed.

    (* ..., which allows instantiating the canonical structure. *)
    Canonical concrete_job_eqMixin := EqMixin eqn_job.
    Canonical concrete_job_eqType := Eval hnf in EqType concrete_job
        concrete_job_eqMixin.

  End Defs.

End ConcreteJob.
```

Listing 59: `job.v`, Source: [CSB16]

The Task

```
Require Import rt.model.basic.time rt.util.all.
Require Import rt.model.basic.task.
From mathcomp Require Import ssreflect ssrbool ssrnat eqtype seq.

Module ConcreteTask.

  Import Time SporadicTaskset.

  Section Defs.

    (* Definition of a concrete task. *)
    Record concrete_task :=
    {
      task_id: nat; (* for uniqueness *)
```

```
      task_cost: time;
      task_period: time;
      task_deadline: time
    }.

    (* To make it compatible with ssreflect, we define a decidable
       equality for concrete tasks. *)
    Definition task_eqdef (t1 t2: concrete_task) :=
      (task_id t1 ==task_id t2) &&
      (task_cost t1 ==task_cost t2) &&
      (task_period t1 ==task_period t2) &&
      (task_deadline t1 ==task_deadline t2).

    (* Next, we prove that task_eqdef is indeed an equality, ... *)
    Lemma eqn_task : Equality.axiom task_eqdef.
    Proof.
      unfold Equality.axiom; intros x y.
      destruct (task_eqdef x y) eqn:EQ.
      {
        apply ReflectT.
        unfold task_eqdef in *.
        move: EQ => /andP [/andP [/andP [/eqP ID /eqP COST] /eqP PERIOD] /eqP DL].
        by destruct x, y; simpl in *; subst.
      }
      {
        apply ReflectF.
        unfold task_eqdef, not in *; intro BUG.
        apply negbT in EQ; rewrite negb_and in EQ.
        destruct x, y.
        rewrite negb_and in EQ.
        move: EQ => /orP [EQ | /eqP DL]; last by apply DL; inversion BUG.
        move: EQ => /orP [EQ | /eqP PERIOD].
        rewrite negb_and in EQ.
        move: EQ => /orP [/eqP ID | /eqP COST].
        by apply ID; inversion BUG.
        by apply COST; inversion BUG.
        by apply PERIOD; inversion BUG.
      }
    Qed.

    (* ..., which allows instantiating the canonical structure. *)
    Canonical concrete_task_eqMixin := EqMixin eqn_task.
    Canonical concrete_task_eqType := Eval hnf in EqType concrete_task
        concrete_task_eqMixin.

  End Defs.

  Section ConcreteTaskset.

    Definition concrete_taskset :=
      taskset_of concrete_task_eqType.

  End ConcreteTaskset.

End ConcreteTask.
```

Listing 60: `task.v`, Source: [CSB16]

Bertogna's EDF example

```
Require Import rt.util.all.
Require Import rt.model.basic.job rt.model.basic.task
               rt.model.basic.schedule rt.model.basic.schedulability
               rt.model.basic.priority rt.model.basic.platform.
Require Import rt.analysis.basic.workload_bound
               rt.analysis.basic.interference_bound_edf
               rt.analysis.basic.bertogna_edf_comp.
Require Import rt.implementation.basic.job
               rt.implementation.basic.task
```

```
               rt.implementation.basic.schedule
               rt.implementation.basic.arrival_sequence.
From mathcomp Require Import ssreflect ssrbool ssrnat eqtype seq bigop div.

Module ResponseTimeAnalysisEDF.

  Import Job Schedule SporadicTaskset Priority Schedulability Platform
      InterferenceBoundEDF WorkloadBound ResponseTimeIterationEDF.
  Import ConcreteJob ConcreteTask ConcreteArrivalSequence ConcreteScheduler.

  Section ExampleRTA.

    Let tsk1 := {| task_id := 1; task_cost := 2; task_period := 5; task_deadline :=
        3|}.
    Let tsk2 := {| task_id := 2; task_cost := 4; task_period := 6; task_deadline :=
        5|}.
    Let tsk3 := {| task_id := 3; task_cost := 3; task_period := 12; task_deadline :=
        11|}.

    (* Let ts be a task set containing these three tasks. *)
    Let ts := [:: tsk1; tsk2; tsk3].

    Section FactsAboutTaskset.

      Fact ts_is_a_set: uniq ts.
      Proof.
        by compute.
      Qed.

      Fact ts_has_valid_parameters:
        valid_sporadic_taskset task_cost task_period task_deadline ts.
      Proof.
        intros tsk IN.
        repeat (move: IN ⇒/orP [/eqP EQ | IN]; subst; compute); by done.
      Qed.

      Fact ts_has_constrained_deadlines:
        forall tsk,
          tsk ∈ ts →
          task_deadline tsk <= task_period tsk.
      Proof.
        intros tsk IN.
        repeat (move: IN ⇒/orP [/eqP EQ | IN]; subst; compute); by done.
      Qed.

    End FactsAboutTaskset.

    (* Assume there are two processors. *)
    Let num_cpus := 2.

    (* Recall the EDF RTA schedulability test. *)
    Let schedulability_test :=
      edf_schedulable task_cost task_period task_deadline num_cpus.

    Fact schedulability_test_succeeds :
      schedulability_test ts = true.
    Proof.
      unfold schedulability_test, edf_schedulable, edf_claimed_bounds; desf.
      apply negbT in Heq; move: Heq ⇒/negP ALL.
      exfalso; apply ALL; clear ALL.
      assert (STEPS: \sum_(tsk ←ts) (task_deadline tsk - task_cost tsk) + 1 = 11).
      {
        by rewrite unlock; compute.
      } rewrite STEPS; clear STEPS.

      Ltac f :=
        unfold edf_rta_iteration; simpl;
        unfold edf_response_time_bound, div_floor, total_interference_bound_edf,
            interference_bound_edf, interference_bound_generic, W; simpl;
        repeat rewrite addnE;
        repeat rewrite big_cons; repeat rewrite big_nil;
        repeat rewrite addnE; simpl;
```

```
        unfold num_cpus, divn; simpl.

      rewrite [edf_rta_iteration]lock; simpl.

      unfold locked at 11; destruct master_key; f.
      unfold locked at 10; destruct master_key; f.
      unfold locked at 9; destruct master_key; f.
      unfold locked at 8; destruct master_key; f.
      unfold locked at 7; destruct master_key; f.
      unfold locked at 6; destruct master_key; f.
      unfold locked at 5; destruct master_key; f.
      unfold locked at 4; destruct master_key; f.
      unfold locked at 3; destruct master_key; f.
      unfold locked at 2; destruct master_key; f.
      by unfold locked at 1; destruct master_key; f.
    Qed.

    (* Let arr_seq be the periodic arrival sequence from ts. *)
    Let arr_seq := periodic_arrival_sequence ts.

    (* Let sched be the work-conserving EDF scheduler. *)
    Let sched := scheduler job_cost num_cpus arr_seq (EDF job_deadline).

    (* Recall the definition of deadline miss. *)
    Let no_deadline_missed_by :=
      task_misses_no_deadline job_cost job_deadline job_task sched.

    (* Next, we prove that ts is schedulable with the result of the test. *)
    Corollary ts_is_schedulable:
      forall tsk,
        tsk ∈ ts →
        no_deadline_missed_by tsk.
    Proof.
      intros tsk IN.
      have VALID := periodic_arrivals_valid_job_parameters ts
          ts_has_valid_parameters.
      have EDFVALID := @edf_valid_policy _ arr_seq job_deadline.
      unfold valid_JLDP_policy, valid_sporadic_job, valid_realtime_job in *; des.
      apply taskset_schedulable_by_edf_rta with (task_cost := task_cost)
          (task_period := task_period) (task_deadline := task_deadline) (ts0 := ts).
      - by apply ts_is_a_set.
      - by apply ts_has_valid_parameters.
      - by apply ts_has_constrained_deadlines.
      - by apply periodic_arrivals_all_jobs_from_taskset.
      - by apply periodic_arrivals_valid_job_parameters, ts_has_valid_parameters.
      - by apply periodic_arrivals_are_sporadic.
      - by compute.
      - by apply scheduler_jobs_must_arrive_to_execute.
      - apply scheduler_completed_jobs_dont_execute; intro j'.
        -- by specialize (VALID j'); des.
        -- by apply periodic_arrivals_is_a_set.
      - by apply scheduler_sequential_jobs, periodic_arrivals_is_a_set.
      - by apply scheduler_work_conserving.
      - by apply scheduler_enforces_policy; ins.
      - by apply schedulability_test_succeeds.
      - by apply IN.
    Qed.

  End ExampleRTA.

End ResponseTimeAnalysisEDF.
```

Listing 61: `bertogna_edf_example.v`, Source: [CSB16]

Kaiser's EDF TODO.

5.4. Axioms of Kolmogorov

If I might have time.

- set theory and the scheduler
- Die Borellsche $\sigma - Algebra$
- $\mathbb{R}$ and $\mathbb{N}$
- note that $\#\mathcal{M}$ in the mbasics scrip is defined for finite sets only
- $|\mathbb{R}| = |\mathbb{N}|$?
- C if f is a function falls es gibt höchstens ein $b \in B$ mit $(a, b) \in f$
  $A^n = ?$
- Es ist messbar.
- Borel-Cantelli lemma, abhängigkeits Definition$\mapsto$ Zeit-Diskretisierung
- Bsp: endlich viele Würfe einer Münze, Lebesgue, Faires Modell
  Zufallsvariable ist messbar abhängig d.h.

  $$\mu(A) = \mathbb{R}(\lambda^{-1}(A)) = \mathbb{P}(\{X|X(\omega)\} \in A\}) = \mathbb{P}(\{X \in A \geq 1|X_n = 1\} \quad (1)$$

  where μ denotes the probability measure of A
  λ denotes the Lebesgue measure
  $\mathbb{P}$ the probability
  A an event in the probability space

- Die Verteilung von $Y = X - 1$ mit $X(\omega) = min\{\omega\}$.

  $$\mu(A) = p \sum_{k \in A} (1-p)^{k-1} \quad (2)$$

  Jede Verteilfunktion mit den eigenschaften von satz 3.5 (mein wahrscheinlichkeitstheore & statistik Skript) ist eine Zufallsvariable.
  Was muss gelten?
  Das suksezzive Gesetzt vom Logarithmus. Gleichung (2) beschreibt die Wahrscheinlichkeitsverteilung eines Random Walks .

5.5. Bell's Inequality

5.6. PROSA and the Discretization of Time

6. Coq as Code Generator

Add how does the proof tactic `reflexivity` actually work (see 2.13)
Steffen's interest
The solutuin is here: [Ext]
Can a code generator be build e.g. Ada VHDh, Scala? See sec. 2.2 bullet 3.

⋆ ⋆ ⋆ End ⋆ ⋆ ⋆

A. Additional Materials

A.1. Coq and Predicate Logic

The semantics of a Coq-`theorem` might be interpreted as a logical formula. The Coq-`Proof.` might be the justification of this formula. For readability in the following premises denoted by capital letters (P and N) are introduced which do not appear in Coq.

line no.	predicate logical translation
1-3	`plus_id_example` means '$\forall\ n\ m\ [n\ , m \in \mathbb{N}] : n = m \rightarrow n + n = m + m$.'
4	Proof: $P\ n\ m := (n + n = m + m)$
5	$N\ n\ m\ := (n,\ m \in \mathbb{N})$
6 7	$H\ n\ m := (n = m)$ $\forall n\ m\ [N\ n\ m] : H\ n\ m \rightarrow (P\ n\ m \leftrightarrow (m + m = m + m))$
8	(* by reflexivity and simplification it is concluded *) $\forall n\ m\ [N\ n\ m] : H\ n\ m \rightarrow P\ n\ m.$
9	$\#.$

Table 3: listing 26 `plus_id_example`'s translation into predicate logic.

TODO: Reference for the logical notation and basics.

B. Grundlagen und Schreibweisen

B.1. Mengen

Es ist sehr schwer den fundamentalen Begriff der Menge mathematisch exakt zu definieren. Aus diesem Grund soll uns hier die von Cantor im Jahr 1895 gegebene Erklärung genügen, da sie für unsere Zwecke völlig ausreichend ist:

Definition 11 (Georg Cantor ([Can95])): *Unter einer* ‚Menge' *verstehen wir jede Zusammenfassung M von bestimmten wohlunterschiedenen Objecten m unsrer Anschauung oder unseres Denkens (welche die* ‚Elemente' *von M genannt werden) zu einem Ganzen*[1]*.*

Für die Formulierung "'genau dann wenn"' verwenden wir im Folgenden die Abkürzung gdw. um Schreibarbeit zu sparen.

B.1.1. Die Elementbeziehung und die Enthaltenseinsrelation

Sehr oft werden einfache große lateinische Buchstaben wie N, M, A, B oder C als Symbole für Mengen verwendet und kleine Buchstaben für die Elemente einer Menge. Mengen von Mengen notiert man gerne mit kalligraphischen Buchstaben wie $\mathcal{A}$, $\mathcal{B}$ oder $\mathcal{M}$.

Definition 12: *Sei M eine beliebige Menge, dann ist*

- *$a \in M$ gdw. a ist ein Element der Menge M,*
- *$a \notin M$ gdw. a ist kein Element der Menge M,*
- *$M \subseteq N$ gdw. aus $a \in M$ folgt $a \in N$ (M ist* Teilmenge *von N),*
- *$M \not\subseteq N$ gdw. es gilt nicht $M \subseteq N$. Gleichwertig: es gibt ein $a \in M$ mit $a \notin N$ (M ist keine Teilmenge von N) und*
- *$M \subset N$ gdw. es gilt $M \subseteq N$ und $M \neq N$ (M ist echte Teilmenge von N).*

Statt $a \in M$ schreibt man auch $M \ni a$, was in einigen Fällen zu einer deutlichen Vereinfachung der Notation führt.

B.1.2. Definition spezieller Mengen

Spezielle Mengen können auf verschiedene Art und Weise definiert werden, wie z.B.

- durch Angabe von Elementen: So ist $\{a_1, \ldots, a_n\}$ die Menge, die aus den Elementen $a_1, \ldots, a_n$ besteht, oder
- durch eine Eigenschaft E: Dabei ist $\{a \mid E(a)\}$ die Menge aller Elemente a, die die Eigenschaft[2] E besitzen.

Alternativ zu der Schreibweise $\{a \mid E(a)\}$ wird auch oft $\{a\colon E(a)\}$ verwendet.

[1] Diese Zitat entspricht der originalen Schreibweise von Cantor.
[2] Die Eigenschaft E kann man dann auch als *Prädikat* bezeichnen.

Example 13: *Mengen, die durch die Angabe von Elementen definiert sind:*

- $\mathbb{B} =_{\text{def}} \{0, 1\}$
- $\mathbb{N} =_{\text{def}} \{0, 1, 2, 3, 4, 5, 6, 7, 8, \dots\}$ *(Menge der* natürlichen Zahlen*)*
- $\mathbb{Z} =_{\text{def}} \{\dots, -4, -3, -2, -1, 0, 1, 2, 3, 4, \dots\}$ *(Menge der* ganzen Zahlen*)*
- $2\mathbb{Z} =_{\text{def}} \{0, \pm 2, \pm 4, \pm 6, \pm 8, \dots\}$ *(Menge der geraden ganzen Zahlen)*
- $\mathbb{P} =_{\text{def}} \{2, 3, 5, 7, 11, 13, 17, 19, \dots\}$ *(Menge der* Primzahlen*)*

Example 14: *Mengen, die durch eine Eigenschaft E definiert sind:*

- $\{n \mid n \in \mathbb{N} \textit{ und } n \textit{ ist durch } 3 \textit{ teilbar}\}$
- $\{n \mid n \in \mathbb{N} \textit{ und } n \textit{ ist Primzahl und } n \leq 40\}$
- $\emptyset =_{\text{def}} \{a \mid a \neq a\}$ *(die leere Menge)*

Aus Definition 38 ergibt sich, dass die leere Menge (Schreibweise: $\emptyset$) Teilmenge jeder Menge ist. Dabei ist zu beachten, dass $\{\emptyset\} \neq \emptyset$ gilt, denn $\{\emptyset\}$ enthält *ein* Element (die leere Menge) und $\emptyset$ enthält *kein* Element.

B.1.3. Operationen auf Mengen

Definition 15: *Seien A und B beliebige Mengen, dann ist*

- $A \cap B =_{\text{def}} \{a \mid a \in A \textit{ und } a \in B\}$ *(*Schnitt *von A und B),*
- $A \cup B =_{\text{def}} \{a \mid a \in A \textit{ oder } a \in B\}$ *(*Vereinigung *von A und B),*
- $A \setminus B =_{\text{def}} \{a \mid a \in A \textit{ und } a \notin B\}$ *(*Differenz *von A und B),*
- $\overline{A} =_{\text{def}} M \setminus A$ *(*Komplement *von A bezüglich einer festen Grundmenge M) und*
- $\mathcal{P}(A) =_{\text{def}} \{B \mid B \subseteq A\}$ *(*Potenzmenge *von A).*

Zwei Mengen A und B mit $A \cap B = \emptyset$ nennt man disjunkt*.*

Example 16: *Sei $A = \{2, 3, 5, 7\}$ und $B = \{1, 2, 4, 6\}$, dann ist $A \cap B = \{2\}$, $A \cup B = \{1, 2, 3, 4, 5, 6, 7\}$ und $A \setminus B = \{3, 5, 7\}$. Wählen wir als Grundmenge die natürlichen Zahlen, also $M = \mathbb{N}$, dann ist $\overline{A} = \{n \in \mathbb{N} \mid n \neq 2 \textit{ und } n \neq 3 \textit{ und } n \neq 5 \textit{ und } n \neq 7\} = \{1, 4, 6, 8, 9, 10, 11, \dots\}$.*
Als Potenzmenge der Menge A ergibt sich die folgende Menge von Mengen von natürlichen Zahlen $\mathcal{P}(A) = \{\emptyset, \{2\}, \{3\}, \{5\}, \{7\}, \{2, 3\}, \{2, 5\}, \{2, 7\}, \{3, 5\}, \{3, 7\}, \{5, 7\}, \{2, 3, 5\}, \{2, 3, 7\}, \{2, 5, 7\}, \{3, 5, 7\}, \{2, 3, 5, 7\}\}$.
Offensichtlich ist die Menge $\{0, 2, 4, 6, 8, \dots\}$ der geraden natürlichen Zahlen und die Menge $\{1, 3, 5, 7, 9, \dots\}$ der ungeraden natürlichen Zahlen disjunkt.

B.1.4. Gesetze für Mengenoperationen

Für die klassischen Mengenoperationen gelten die folgenden Beziehungen:

$A \cap B$	$=$	$B \cap A$	Kommutativgesetz für den Schnitt
$A \cup B$	$=$	$B \cup A$	Kommutativgesetz für die Vereinigung
$A \cap (B \cap C)$	$=$	$(A \cap B) \cap C$	Assoziativgesetz für den Schnitt
$A \cup (B \cup C)$	$=$	$(A \cup B) \cup C$	Assoziativgesetz für die Vereinigung
$A \cap (B \cup C)$	$=$	$(A \cap B) \cup (A \cap C)$	Distributivgesetz
$A \cup (B \cap C)$	$=$	$(A \cup B) \cap (A \cup C)$	Distributivgesetz
$A \cap A$	$=$	A	Duplizitätsgesetz für den Schnitt
$A \cup A$	$=$	A	Duplizitätsgesetz für die Vereinigung
$A \cap (A \cup B)$	$=$	A	Absorptionsgesetz
$A \cup (A \cap B)$	$=$	A	Absorptionsgesetz
$\overline{A \cap B}$	$=$	$(\overline{A} \cup \overline{B})$	de-Morgansche Regel
$\overline{A \cup B}$	$=$	$(\overline{A} \cap \overline{B})$	de-Morgansche Regel
$\overline{\overline{A}}$	$=$	A	Gesetz des doppelten Komplements

Die "'de-Morganschen Regeln"' wurden nach dem englischen Mathematiker Augustus De Morgan[3] benannt.
Als Abkürzung schreibt man statt $X_1 \cup X_2 \cup \cdots \cup X_n$ (bzw. $X_1 \cap X_2 \cap \cdots \cap X_n$) einfach $\bigcup_{i=1}^{n} X_i$ (bzw. $\bigcap_{i=1}^{n} X_i$). Möchte man alle Mengen X_i mit $i \in \mathbb{N}$ schneiden (bzw. vereinigen), so schreibt man kurz $\bigcap_{i \in \mathbb{N}} X_i$ (bzw. $\bigcup_{i \in \mathbb{N}} X_i$).
Oft benötigt man eine Verknüpfung von zwei Mengen, eine solche Verknüpfung wird allgemein wie folgt definiert:

Definition 17 ("'Verknüpfung von Mengen"'): *Seien A und B zwei Mengen und "'$\odot$"' eine beliebige Verknüpfung zwischen den Elementen dieser Mengen, dann definieren wir*

$$A \odot B =_{\text{def}} \{a \odot b \mid a \in A \text{ und } b \in B\}.$$

Example 18: *Die Menge $3\mathbb{Z} = \{0, \pm 3, \pm 6, \pm 9, \dots\}$ enthält alle Vielfachen[4] von 3, damit ist $3\mathbb{Z} + \{1\} = \{1, 4, -2, 7, -5, 10, -8, \dots\}$. Die Menge $3\mathbb{Z} + \{1\}$ schreibt man kurz oft auch als $3\mathbb{Z} + 1$, wenn klar ist, was mit dieser Abkürzung gemeint ist.*

B.1.5. Tupel (Vektoren) und das Kreuzprodukt

Seien $A, A_1, \dots, A_n$ im folgenden Mengen, dann bezeichnet

- $(a_1, \dots, a_n) =_{\text{def}}$ die Elemente $a_1, \dots, a_n$ in genau dieser festgelegten *Reihenfolge* und z.B. $(3, 2) \neq (2, 3)$. Wir sprechen von einem n-Tupel.
- $A_1 \times A_2 \times \cdots \times A_n =_{\text{def}} \{(a_1, \dots, a_n) \mid a_1 \in A_1, a_2 \in A_2, \dots, a_n \in A_n\}$ (Kreuzprodukt der Mengen $A_1, A_2, \dots, A_n$),
- $A^n =_{\text{def}} \underbrace{A \times A \times \cdots \times A}_{n\text{-mal}}$ (n-faches Kreuzprodukt der Menge A) und
- speziell gilt $A^1 = \{(a) \mid a \in A\}$.

[3] ⋆1806 in Madurai, Tamil Nadu, Indien - †1871 in London, England
[4] Eigentlich müsste man statt $3\mathbb{Z}$ die Notation $\{3\}\mathbb{Z}$ verwenden. Dies ist allerdings unüblich.

Wir nennen 2-Tupel auch *Paare*, 3-Tupel auch *Tripel*, 4-Tupel auch *Quadrupel* und 5-Tupel *Quintupel*. Bei n-Tupeln ist, im Gegensatz zu Mengen, eine Reihenfolge vorgegeben, d.h. es gilt z.B. immer $\{a, b\} = \{b, a\}$, aber im Allgemeinen $(a, b) \neq (b, a)$.

Example 19: *Sei $A = \{1, 2, 3\}$ und $B = \{a, b, c\}$, dann bezeichnet das Kreuzprodukt von A und B die Menge von Paaren $A \times B = \{(1, a), (1, b), (1, c), (2, a), (2, b), (2, c), (3, a), (3, b), (3, c)\}$.*

B.1.6. Die Anzahl von Elementen in Mengen

Sei A eine Menge, die endlich viele Elemente[5] enthält, dann ist

$$\#A =_{\text{def}} \text{Anzahl der Elemente in der Menge } A.$$

Beispielsweise ist $\#\{4, 7, 9\} = 3$. Mit dieser Definition gilt

- $\#(A^n) = (\#A)^n$,
- $\#\mathcal{P}(A) = 2^{\#A}$,
- $\#A + \#B = \#(A \cup B) + \#(A \cap B)$ und
- $\#A = \#(A \setminus B) + \#(A \cap B)$.

B.2. Relationen und Funktionen

B.2.1. Eigenschaften von Relationen

Seien $A_1, \dots, A_n$ beliebige Mengen, dann ist R eine *n-stellige Relation* gdw. $R \subseteq A_1 \times A_2 \times \cdots \times A_n$. Eine zweistellige Relation nennt man auch *binäre Relation*. Oft werden auch Relationen $R \subseteq A^n$ betrachtet, diese bezeichnet man dann als n-stellige Relation über der Menge A.

Definition 20: *Sei R eine zweistellige Relation über A, dann ist R*

- reflexiv *gdw. $(a, a) \in R$ für alle $a \in A$,*
- symmetrisch *gdw. aus $(a, b) \in R$ folgt $(b, a) \in R$,*
- antisymmetrisch *gdw. aus $(a, b) \in R$ und $(b, a) \in R$ folgt $a = b$,*
- transitiv *gdw. aus $(a, b) \in R$ und $(b, c) \in R$ folgt $(a, c) \in R$ und*
- linear *gdw. es gilt immer $(a, b) \in R$ oder $(b, a) \in R$.*

Definition 21: *Sei R eine zweistellige Relation, dann*

- *heißt R* Halbordnung *gdw. R ist reflexiv, antisymmetrisch und transitiv,*
- Ordnung *gdw. R ist eine lineare Halbordnung und*
- Äquivalenzrelation *gdw. R reflexiv, transitiv und symmetrisch ist.*

Example 22: *Die Teilmengenrelation "'$\subseteq$"' auf allen Teilmengen von $\mathbb{Z}$ ist eine Halbordnung, aber keine Ordnung.*

[5] Solche Mengen werden als *endliche Mengen* bezeichnet.

Example 23: *Wir schreiben* $a \equiv b \bmod n$*, falls es eine ganze Zahl* q *gibt, für die* $a - b = qn$ *gilt. Für* $n \geq 2$ *ist die Relation* $R_n(a,b) =_{\text{def}} \{(a,b) \mid a \equiv b \bmod n\} \subseteq \mathbb{Z}^2$ *eine Äquivalenzrelation.*

B.2.2. Eigenschaften von Funktionen

Seien A und B beliebige Mengen. f ist eine *Funktion* von A nach B (Schreibweise: $f\colon A \to B$) gdw. $f \subseteq A \times B$ und für jedes $a \in A$ gibt es *höchstens* ein $b \in B$ mit $(a,b) \in f$. Ist also $(a,b) \in f$, so schreibt man $f(a) = b$. Ebenfalls gebräuchlich ist die Notation $a \mapsto b$.

Remark 24: *Unsere Definition von Funktion umfasst auch mehrstellige Funktionen. Seien* C *und* B *Mengen und* $A = C^n$ *das* n*-fache Kreuzprodukt von* C*. Die Funktion* $f\colon A \to B$ *ist dann eine* n*-stellige Funktion, denn sie bildet* n*-Tupel aus* C^n *auf Elemente aus* B *ab.*

Definition 25: *Sei* f *eine* n*-stellige Funktion. Möchte man die Funktion* f *benutzen, aber keine Namen für die Argumente vergeben, so schreibt man auch*

$$f(\underbrace{\cdot, \cdot, \dots, \cdot}_{n\text{-mal}})$$

Ist also der Namen des Arguments einer einstelligen Funktion $g(x)$ *für eine Betrachtung unwichtig, so kann man* $g(\cdot)$ *schreiben, um anzudeuten, dass* g *einstellig ist, ohne dies weiter zu erwähnen.*

Definition 26: *Sei nun* $R \subseteq A_1 \times A_2 \times \cdots \times A_n$ *eine* n*-stellige Relation, dann definieren wir eine Funktion* $P_R^n\colon A_1 \times A_2 \times \cdots \times A_n \to \{0,1\}$ *wie folgt:*

$$P_R^n(x_1, \dots, x_n) =_{\text{def}} \begin{cases} 1, & \textit{falls } (x_1, \dots, x_n) \in R \\ 0, & \textit{sonst} \end{cases}$$

Eine solche n*-stellige Funktion, die "'anzeigt"', ob ein Element aus* $A_1 \times A_2 \times \cdots \times A_n$ *entweder zu* R *gehört oder nicht, nennt man (*n*-stelliges)* Prädikat*.*

Example 27: *Sei* $\mathbb{P} =_{\text{def}} \{n \in \mathbb{N} \mid n \textit{ ist Primzahl}\}$*, dann ist* $\mathbb{P}$ *eine* 1*-stellige Relation über den natürlichen Zahlen. Das Prädikat* $P_{\mathbb{P}}^1(n)$ *liefert für eine natürliche Zahl* n *genau dann* 1*, wenn* n *eine Primzahl ist.*

Ist für ein Prädikat P_R^n sowohl die Relation R als auch die Stelligkeit n aus dem Kontext klar, dann schreibt man auch kurz P oder verwendet das Relationensymbol R als Notation für das Prädikat P_R^n.

Nun legen wir zwei spezielle Funktionen fest, die oft sehr hilfreich sind:

Definition 28: *Sei* $\alpha \in \mathbb{R}$ *eine beliebige reelle Zahl, dann gilt*

- $\lceil \alpha \rceil =_{\text{def}}$ *die kleinste ganze Zahl, die größer oder gleich* α *ist (*$\triangleq$ *"'Aufrunden"')*
- $\lfloor \alpha \rfloor =_{\text{def}}$ *die größte ganze Zahl, die kleiner oder gleich* α *ist (*$\triangleq$ *"'Abrunden"')*

Definition 29: *Für eine beliebige Funktion* f *legen wir fest:*

- *Der* Definitionsbereich *von* f *ist* $D_f =_{\text{def}} \{a \mid \textit{es gibt ein } b \textit{ mit } f(a) = b\}$*.*
- *Der* Wertebereich *von* f *ist* $W_f =_{\text{def}} \{b \mid \textit{es gibt ein } a \textit{ mit } f(a) = b\}$*.*

- *Die Funktion $f\colon A \to B$ ist* total *gdw.* $D_f = A$.
- *Die Funktion $f\colon A \to B$ heißt* surjektiv *gdw.* $W_f = B$.
- *Die Funktion f heißt* injektiv *(oder eineindeutig*[6]*) gdw. immer wenn* $f(a_1) = f(a_2)$ *gilt auch* $a_1 = a_2$.
- *Die Funktion f heißt* bijektiv *gdw. f ist injektiv und surjektiv.*

Mit Hilfe der Kontraposition (siehe Abschnitt E.1.1) kann man für die Injektivität alternativ auch zeigen, dass immer wenn $a_1 \neq a_2$, dann muss auch $f(a_1) \neq f(a_2)$ gelten.

Example 30: *Sei die Funktion $f\colon \mathbb{N} \to \mathbb{Z}$ durch $f(n) = (-1)^n \lceil \frac{n}{2} \rceil$ gegeben. Die Funktion f ist surjektiv, denn $f(0) = 0, f(1) = -1, f(2) = 1, f(3) = -2, f(4) = 2, \ldots$, d.h. die ungeraden natürlichen Zahlen werden auf die negativen ganzen Zahlen abgebildet, die geraden Zahlen aus $\mathbb{N}$ werden auf die positiven ganzen Zahlen abgebildet und deshalb ist $W_f = \mathbb{Z}$.*
Weiterhin ist f auch injektiv, denn aus[7] $(-1)^{a_1} \lceil \frac{a_1}{2} \rceil = (-1)^{a_2} \lceil \frac{a_2}{2} \rceil$ *folgt, dass entweder a_1 und a_2 gerade oder a_1 und a_2 ungerade, denn sonst würden auf der linken und rechten Seite der Gleichung unterschiedliche Vorzeichen auftreten. Ist a_1 gerade und a_2 gerade, dann gilt $\lceil \frac{a_1}{2} \rceil = \lceil \frac{a_2}{2} \rceil$ und auch $a_1 = a_2$. Sind a_1 und a_2 ungerade, dann gilt $-\lceil \frac{a_1}{2} \rceil = -\lceil \frac{a_2}{2} \rceil$, woraus auch folgt, dass $a_1 = a_2$. Damit ist die Funktion f bijektiv. Weiterhin ist f auch total, d.h. $D_f = \mathbb{N}$.*

Definition 31: *Unter einem n-stelligen* Operator *f (auf der Menge Y) versteht man in der Mathematik eine Funktion der Form $f\colon Y^n \to Y$. Einfache Beispiele für zweistellige Operatoren sind der Additions- oder Multiplikationsoperator.*

B.2.3. Hüllenoperatoren

Definition 32: *Sei X eine Menge. Ein einstelliger Operator $\Psi\colon \mathcal{P}(X) \to \mathcal{P}(X)$ heißt* Hüllenoperator*, wenn er die folgenden drei Eigenschaften erfüllt:*

Einbettung: *für alle $A \in \mathcal{P}(X)$ gilt $A \subseteq \Psi(A)$*

Monotonie: *für alle $A, B \in \mathcal{P}(X)$ mit $A \subseteq B$ folgt $\Psi(A) \subseteq \Psi(B)$*

Abgeschlossenheit: *für alle $A \in \mathcal{P}(X)$ gilt $\Psi(\Psi(A)) = \Psi(A)$*

Aufgrund der Monotonieeigenschaft eines Hüllenoperators kann man bei der Abgeschlossenheit die Eigenschaft $\Psi(\Psi(A)) = \Psi(A)$ auch durch $\Psi(\Psi(A)) \subseteq \Psi(A)$ ersetzen. In der Informatik spielen Hüllenoperatoren eine große Rolle. Gute Beispiele hierfür sind z.B. die *transitive Hülle* (vgl. Computergraphik), die Kleene-Hülle (vgl. Formale Sprachen) oder der Abschluss einer Komplexitätsklasse unter Schnitt oder Vereinigung.

---

[6] Achtung: Dieser Begriff wird manchmal unterschiedlich, je nach Autor, in den Bedeutungen "‘bijektiv"’ oder "‘injektiv"’ verwendet.

[7] Für die Definition der Funktion $\lceil \cdot \rceil$ siehe Definition 54.

B.2.4. Permutationen

Sei S eine beliebige endliche Menge, dann heißt eine bijektive Funktion π der Form $\pi\colon S \to S$ *Permutation.* Das bedeutet, dass die Funktion π Elemente aus S wieder auf Elemente aus S abbildet, wobei für jedes $b \in S$ ein $a \in S$ mit $f(a) = b$ existiert (Surjektivität) und falls $f(a_1) = f(a_2)$ gilt, dann ist $a_1 = a_2$ (Injektivität).

Remark 33: *Man kann den Permutationsbegriff auch auf unendliche Mengen erweitern, aber besonders häufig werden in der Informatik* Permutationen von endlichen Mengen *benötigt. Aus diesem Grund sollen hier nur endliche Mengen S betrachtet werden.*

Sei nun $S = \{1, \dots, n\}$ (eine endliche Menge) und $\pi\colon \{1, \dots, n\} \to \{1, \dots, n\}$ eine Permutation. Permutationen dieser Art kann man sehr anschaulich mit Hilfe einer Matrix aufschreiben:

$$\pi = \begin{pmatrix} 1 & 2 & \dots & n \\ \pi(1) & \pi(2) & \dots & \pi(n) \end{pmatrix}$$

Durch diese Notation wird klar, dass das Element 1 der Menge S durch das Element $\pi(1)$ ersetzt wird, das Element 2 wird mit $\pi(2)$ vertauscht und allgemein das Element i durch $\pi(i)$ für $1 \leq i \leq n$. In der zweiten Zeile dieser Matrixnotation findet sich also *jedes* (Surjektivität) Element der Menge S genau *einmal* (Injektivität).

Example 34: *Sei $S = \{1, \dots, 3\}$ eine Menge mit drei Elementen. Dann gibt es, wie man ausprobieren kann, genau 6 Permutationen von S:*

$$\begin{array}{llllll} \pi_1 & = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix} & \pi_2 & = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix} & \pi_3 & = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix} \\ \pi_4 & = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} & \pi_5 & = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} & \pi_6 & = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} \end{array}$$

Theorem 35: *Sei S eine endliche Menge mit $n = |S|$, dann gibt es genau $n!$ (Fakultät) verschiedene Permutationen von S.*

Beweis: Jede Permutation π der Menge S von n Elementen kann als Matrix der Form

$$\pi = \begin{pmatrix} 1 & 2 & \dots & n \\ \pi(1) & \pi(2) & \dots & \pi(n) \end{pmatrix}$$

aufgeschrieben werden. Damit ergibt sich die Anzahl der Permutationen von S durch die Anzahl der verschiedenen zweiten Zeilen solcher Matrizen. In jeder solchen Zeile muss jedes der n Elemente von S genau einmal vorkommen, da π eine bijektive Abbildung ist, d.h. wir haben für die erste Position der zweiten Zeile der Matrixdarstellung genau n verschiedene Möglichkeiten, für die zweite Position noch $n-1$ und für die dritte noch $n-2$. Für die n-te Position bleibt nur noch 1 mögliches Element aus S übrig[8]. Zusammengenommen haben wir also $n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \ldots \cdot 2 \cdot 1 = n!$ verschiedene mögliche Permutationen der Menge S. #

[8]Dies kann man sich auch als die Anzahl der verschiedenen Möglichkeiten vorstellen, die bestehen, wenn man aus einer Urne mit n numerierten Kugeln alle Kugeln *ohne* Zurücklegen nacheinander zieht.

B.3. Summen und Produkte

B.3.1. Summen

Zur abkürzenden Schreibweise verwendet man für Summen das Summenzeichen $\sum$. Dabei ist

$$\sum_{i=1}^{n} a_i =_{\text{def}} a_1 + a_2 + \cdots + a_n.$$

Mit Hilfe dieser Definition ergeben sich auf elementare Weise die folgenden Rechenregeln:

- Sei $a_i = a$ für $1 \leq i \leq n$, dann gilt $\sum\limits_{i=1}^{n} a_i = n \cdot a$ (Summe gleicher Summanden).
- $\sum\limits_{i=1}^{n} a_i = \sum\limits_{i=1}^{m} a_i + \sum\limits_{i=m+1}^{n} a_i$, wenn $1 < m < n$ (Aufspalten einer Summe).
- $\sum\limits_{i=1}^{n} (a_i + b_i + c_i + \ldots) = \sum\limits_{i=1}^{n} a_i + \sum\limits_{i=1}^{n} b_i + \sum\limits_{i=1}^{n} c_i + \ldots$ (Addition von Summen).
- $\sum\limits_{i=1}^{n} a_i = \sum\limits_{i=l}^{n+l-1} a_{i-l+1}$ und $\sum\limits_{i=l}^{n} a_i = \sum\limits_{i=1}^{n-l+1} a_{i+l-1}$ (Umnumerierung von Summen).
- $\sum\limits_{i=1}^{n} \sum\limits_{j=1}^{m} a_{i,j} = \sum\limits_{j=1}^{m} \sum\limits_{i=1}^{n} a_{i,j}$ (Vertauschen der Summationsfolge).

Manchmal verwendet man keine Laufindizes an ein Summenzeichen, sondern man beschreibt (durch ein Prädikat) welche Zahlen aufsummiert werden sollen. So kann man eine Funktion definieren, die die Summe aller Teiler einer natürlichen Zahl liefert:

$$\sigma(n) =_{\text{def}} \sum_{\substack{t \leq n \\ t \text{ teilt } n}} t$$

B.3.2. Produkte

Zur abkürzenden Schreibweise verwendet man für Produkte das Produktzeichen $\prod$. Dabei ist

$$\prod_{i=1}^{n} a_i =_{\text{def}} a_1 \cdot a_2 \cdot \ldots \cdot a_n.$$

Mit Hilfe dieser Definition ergeben sich auf elementare Weise die folgenden Rechenregeln:

- Sei $a_i = a$ für $1 \leq i \leq n$, dann gilt $\prod\limits_{i=1}^{n} a_i = a^n$ (Produkt gleicher Faktoren).
- $\prod\limits_{i=1}^{n} (ca_i) = c^n \prod\limits_{i=1}^{n} a_i$ (Vorziehen von konstanten Faktoren)
- $\prod\limits_{i=1}^{n} a_i = \prod\limits_{i=1}^{m} a_i \cdot \prod\limits_{i=m+1}^{n} a_i$, wenn $1 < m < n$ (Aufspalten in Teilprodukte).
- $\prod\limits_{i=1}^{n} (a_i \cdot b_i \cdot c_i \cdot \ldots) = \prod\limits_{i=1}^{n} a_i \cdot \prod\limits_{i=1}^{n} b_i \cdot \prod\limits_{i=1}^{n} c_i \cdot \ldots$ (Das Produkt von Produkten).

- $\prod_{i=1}^{n} a_i = \prod_{i=l}^{n+l-1} a_{i-l+1}$ und $\prod_{i=l}^{n} a_i = \prod_{i=1}^{n-l+1} a_{i+l-1}$ (Umnumerierung von Produkten).
- $\prod_{i=1}^{n} \prod_{j=1}^{m} a_{i,j} = \prod_{j=1}^{m} \prod_{i=1}^{n} a_{i,j}$ (Vertauschen der Reihenfolge bei Doppelprodukten).

Ähnlich wie bei Summen kann man bei Produkten auch ohne Laufindex arbeiten. So ist z.B. die *Eulersche ϕ-Funktion* wie folgt definiert:

$$\phi(n) =_{\text{def}} n \prod_{p|n} (1 - \frac{1}{p})$$

In das Produkt gehen also alle Primteiler p von n mit dem Faktor $1 - 1/p$ ein.
Oft werden Summen- oder Produktsymbole verwendet bei denen der Startindex größer als der Stopindex ist. Solche Summen bzw. Produkte sind "‘leer"’, d.h. es wird nichts summiert bzw. multipliziert. Sind dagegen Start- und Endindex gleich, so tritt nur genau ein Wert auf, d.h. das Summen- bzw. Produktsymbol hat in diesem Fall keine Auswirkung. Es ergeben sich also die folgenden Rechenregeln:

- Seien $n, m \in \mathbb{Z}$ und $n < m$, dann
$$\sum_{i=m}^{n} a_i = 0 \text{ und } \prod_{i=m}^{n} a_i = 1$$
- Sei $n \in \mathbb{Z}$, dann
$$\sum_{i=n}^{n} a_i = a_n = \prod_{i=n}^{n} a_i$$

Example 36: *Die folgende Identität wird Euler zugeschrieben. Erstaunlicherweise kann man damit eine Summe von Kehrwerten von Potenzen mit einem Produkt von Primzahlen in Verbindung bringen.*

$$\sum_{n=1}^{\infty} \frac{1}{n^s} = \prod_{p} \frac{1}{1 - \frac{1}{p^s}}$$

Diese Identiät kann man Die Summe auf der linken Seite ist auch als Riemannsche Zetafuntion $\zeta(s)$ *bekannt. Erstaunlicherweise gilt dann der folgende Zusammenhang*

$$\zeta(2) = \sum_{n=1}^{\infty} \frac{1}{n^2} = \prod_{p} \frac{1}{1 - \frac{1}{p^2}} = \frac{\pi^2}{6},$$

denn durch diese Gleichung wird die Menge der Primzahlen in Beziehung zur Kreiszahl π gebracht.

B.4. Logarithmieren, Potenzieren und Radizieren

Die Schreibweise a^b ist eine Abkürzung für

$$a^b =_{\text{def}} \underbrace{a \cdot a \cdot \ldots \cdot a}_{b-\text{mal}}$$

und wird als *Potenzierung* bezeichnet. Dabei ist a die *Basis*, b der *Exponent* und a^b die b-te *Potenz* von a. Seien nun $r, s, t \in \mathbb{R}$ und $r, t \geq 0$ durch die folgende Gleichung verbunden:

$$r^s = t.$$

Dann lässt sich diese Gleichung wie folgt umstellen und es gelten die folgenden Rechenregeln:

Logarithmieren	Potenzieren	Radizieren
$\mathbf{s} = \log_{\mathbf{r}} \mathbf{t}$	$\mathbf{t} = \mathbf{r}^{\mathbf{s}}$	$\mathbf{r} = \sqrt[\mathbf{s}]{\mathbf{t}}$
i) $\log_r(\frac{u}{v}) = \log_r u - \log_r v$	i) $r^u \cdot r^v = r^{u+v}$	i) $\sqrt[s]{u} \cdot \sqrt[s]{v} = \sqrt[s]{u \cdot v}$
ii) $\log_r(u \cdot v) = \log_r u + \log_r v$	ii) $\frac{r^u}{r^v} = r^{u-v}$	ii) $\frac{\sqrt[s]{u}}{\sqrt[s]{v}} = \sqrt[s]{\left(\frac{u}{v}\right)}$
iii) $\log_r(t^u) = u \cdot \log_r t$	iii) $u^s \cdot v^s = (u \cdot v)^s$	iii) $\sqrt[u]{\sqrt[v]{t}} = \sqrt[u \cdot v]{t}$
iv) $\log_r(\sqrt[u]{t}) = \frac{1}{u} \cdot \log_r t$	iv) $\frac{u^s}{v^s} = \left(\frac{u}{v}\right)^s$	
v) $\frac{\log_r t}{\log_r u} = \log_u t$ (Basiswechsel)	v) $(r^u)^v = r^{u \cdot v}$	

Zusätzlich gilt: Wenn $r > 1$, dann ist $s_1 < s_2$ gdw. $r^{s_1} < r^{s_2}$ (Monotonie).
Da $\sqrt[s]{t} = t^{\left(\frac{1}{s}\right)}$ gilt, können die Gesetze für das Radizieren leicht aus den Potenzierungsgesetzen abgeleitet werden. Weiterhin legen wir spezielle Schreibweisen für die Logarithmen zur Basis 10, e (Eulersche Zahl) und 2 fest: $\lg t =_{\text{def}} \log_{10} t$, $\ln t =_{\text{def}} \log_e t$ und $\operatorname{lb} t =_{\text{def}} \log_2 t$.

B.5. Gebräuchliche griechische Buchstaben

In der Informatik, Mathematik und Physik ist es üblich, griechische Buchstaben zu verwenden. Ein Grund hierfür ist, dass es so möglich wird mit einer größeren Anzahl von Unbekannten arbeiten zu können, ohne unübersichtliche und oft unhandliche Indizes benutzen zu müssen.

Kleinbuchstaben:

Symbol	Bezeichnung	Symbol	Bezeichnung	Symbol	Bezeichnung
α	Alpha	β	Beta	γ	Gamma
δ	Delta	ϕ	Phi	φ	Phi
ξ	Xi	ζ	Zeta	ϵ	Epsilon
θ	Theta	λ	Lambda	π	Pi
σ	Sigma	η	Eta	μ	Mu

Großbuchstaben:

Symbol	Bezeichnung	Symbol	Bezeichnung	Symbol	Bezeichnung
Γ	Gamma	Δ	Delta	Φ	Phi
Ξ	Xi	Θ	Theta	Λ	Lambda
Π	Pi	Σ	Sigma	Ψ	Psi
Ω	Omega				

C. Grundlagen und Schreibweisen

C.1. Mengen

Es ist sehr schwer den fundamentalen Begriff der Menge mathematisch exakt zu definieren. Aus diesem Grund soll uns hier die von Cantor im Jahr 1895 gegebene Erklärung genügen, da sie für unsere Zwecke völlig ausreichend ist:

Definition 37 (Georg Cantor ([Can95])): *Unter einer* ‚Menge' *verstehen wir jede Zusammenfassung M von bestimmten wohlunterschiedenen Objecten m unsrer Anschauung oder unseres Denkens (welche die* ‚Elemente' *von M genannt werden) zu einem Ganzen*[9]*.*

Für die Formulierung "'genau dann wenn"' verwenden wir im Folgenden die Abkürzung gdw. um Schreibarbeit zu sparen.

C.1.1. Die Elementbeziehung und die Enthaltenseinsrelation

Sehr oft werden einfache große lateinische Buchstaben wie N, M, A, B oder C als Symbole für Mengen verwendet und kleine Buchstaben für die Elemente einer Menge. Mengen von Mengen notiert man gerne mit kalligraphischen Buchstaben wie $\mathcal{A}$, $\mathcal{B}$ oder $\mathcal{M}$.

Definition 38: *Sei M eine beliebige Menge, dann ist*

- *$a \in M$ gdw. a ist ein Element der Menge M,*
- *$a \notin M$ gdw. a ist kein Element der Menge M,*
- *$M \subseteq N$ gdw. aus $a \in M$ folgt $a \in N$ (M ist* Teilmenge *von N),*
- *$M \not\subseteq N$ gdw. es gilt nicht $M \subseteq N$. Gleichwertig: es gibt ein $a \in M$ mit $a \notin N$ (M ist keine Teilmenge von N) und*
- *$M \subset N$ gdw. es gilt $M \subseteq N$ und $M \neq N$ (M ist echte Teilmenge von N).*

Statt $a \in M$ schreibt man auch $M \ni a$, was in einigen Fällen zu einer deutlichen Vereinfachung der Notation führt.

C.1.2. Definition spezieller Mengen

Spezielle Mengen können auf verschiedene Art und Weise definiert werden, wie z.B.

- durch Angabe von Elementen: So ist $\{a_1, \ldots, a_n\}$ die Menge, die aus den Elementen $a_1, \ldots, a_n$ besteht, oder
- durch eine Eigenschaft E: Dabei ist $\{a \mid E(a)\}$ die Menge aller Elemente a, die die Eigenschaft[10] E besitzen.

Alternativ zu der Schreibweise $\{a \mid E(a)\}$ wird auch oft $\{a\colon E(a)\}$ verwendet.

[9] Diese Zitat entspricht der originalen Schreibweise von Cantor.
[10] Die Eigenschaft E kann man dann auch als *Prädikat* bezeichnen.

Example 39: *Mengen, die durch die Angabe von Elementen definiert sind:*

- $\mathbb{B} =_{\text{def}} \{0, 1\}$
- $\mathbb{N} =_{\text{def}} \{0, 1, 2, 3, 4, 5, 6, 7, 8, \dots\}$ *(Menge der* natürlichen Zahlen*)*
- $\mathbb{Z} =_{\text{def}} \{\dots, -4, -3, -2, -1, 0, 1, 2, 3, 4, \dots\}$ *(Menge der* ganzen Zahlen*)*
- $2\mathbb{Z} =_{\text{def}} \{0, \pm 2, \pm 4, \pm 6, \pm 8, \dots\}$ *(Menge der geraden ganzen Zahlen)*
- $\mathbb{P} =_{\text{def}} \{2, 3, 5, 7, 11, 13, 17, 19, \dots\}$ *(Menge der* Primzahlen*)*

Example 40: *Mengen, die durch eine Eigenschaft E definiert sind:*

- $\{n \mid n \in \mathbb{N} \textit{ und } n \textit{ ist durch } 3 \textit{ teilbar}\}$
- $\{n \mid n \in \mathbb{N} \textit{ und } n \textit{ ist Primzahl und } n \leq 40\}$
- $\emptyset =_{\text{def}} \{a \mid a \neq a\}$ *(die leere Menge)*

Aus Definition 38 ergibt sich, dass die leere Menge (Schreibweise: $\emptyset$) Teilmenge jeder Menge ist. Dabei ist zu beachten, dass $\{\emptyset\} \neq \emptyset$ gilt, denn $\{\emptyset\}$ enthält *ein* Element (die leere Menge) und $\emptyset$ enthält *kein* Element.

C.1.3. Operationen auf Mengen

Definition 41: *Seien A und B beliebige Mengen, dann ist*

- $A \cap B =_{\text{def}} \{a \mid a \in A \textit{ und } a \in B\}$ *(*Schnitt *von A und B),*
- $A \cup B =_{\text{def}} \{a \mid a \in A \textit{ oder } a \in B\}$ *(*Vereinigung *von A und B),*
- $A \setminus B =_{\text{def}} \{a \mid a \in A \textit{ und } a \notin B\}$ *(*Differenz *von A und B),*
- $\overline{A} =_{\text{def}} M \setminus A$ *(*Komplement *von A bezüglich einer festen Grundmenge M) und*
- $\mathcal{P}(A) =_{\text{def}} \{B \mid B \subseteq A\}$ *(*Potenzmenge *von A).*

Zwei Mengen A und B mit $A \cap B = \emptyset$ nennt man disjunkt*.*

Example 42: *Sei $A = \{2, 3, 5, 7\}$ und $B = \{1, 2, 4, 6\}$, dann ist $A \cap B = \{2\}$, $A \cup B = \{1, 2, 3, 4, 5, 6, 7\}$ und $A \setminus B = \{3, 5, 7\}$. Wählen wir als Grundmenge die natürlichen Zahlen, also $M = \mathbb{N}$, dann ist $\overline{A} = \{n \in \mathbb{N} \mid n \neq 2 \textit{ und } n \neq 3 \textit{ und } n \neq 5 \textit{ und } n \neq 7\} = \{1, 4, 6, 8, 9, 10, 11, \dots\}$.*
Als Potenzmenge der Menge A ergibt sich die folgende Menge von Mengen von natürlichen Zahlen $\mathcal{P}(A) = \{\emptyset, \{2\}, \{3\}, \{5\}, \{7\}, \{2, 3\}, \{2, 5\}, \{2, 7\}, \{3, 5\}, \{3, 7\}, \{5, 7\}, \{2, 3, 5\}, \{2, 3, 7\}, \{2, 5, 7\}, \{3, 5, 7\}, \{2, 3, 5, 7\}\}$.
Offensichtlich ist die Menge $\{0, 2, 4, 6, 8, \dots\}$ der geraden natürlichen Zahlen und die Menge $\{1, 3, 5, 7, 9, \dots\}$ der ungeraden natürlichen Zahlen disjunkt.

C.1.4. Gesetze für Mengenoperationen

Für die klassischen Mengenoperationen gelten die folgenden Beziehungen:

$$\begin{array}{rcll}
A \cap B & = & B \cap A & \text{Kommutativgesetz für den Schnitt} \\
A \cup B & = & B \cup A & \text{Kommutativgesetz für die Vereinigung} \\
A \cap (B \cap C) & = & (A \cap B) \cap C & \text{Assoziativgesetz für den Schnitt} \\
A \cup (B \cup C) & = & (A \cup B) \cup C & \text{Assoziativgesetz für die Vereinigung} \\
A \cap (B \cup C) & = & (A \cap B) \cup (A \cap C) & \text{Distributivgesetz} \\
A \cup (B \cap C) & = & (A \cup B) \cap (A \cup C) & \text{Distributivgesetz} \\
A \cap A & = & A & \text{Duplizitätsgesetz für den Schnitt} \\
A \cup A & = & A & \text{Duplizitätsgesetz für die Vereinigung} \\
A \cap (A \cup B) & = & A & \text{Absorptionsgesetz} \\
A \cup (A \cap B) & = & A & \text{Absorptionsgesetz} \\
\overline{A \cap B} & = & (\overline{A} \cup \overline{B}) & \text{de-Morgansche Regel} \\
\overline{A \cup B} & = & (\overline{A} \cap \overline{B}) & \text{de-Morgansche Regel} \\
\overline{\overline{A}} & = & A & \text{Gesetz des doppelten Komplements}
\end{array}$$

Die "'de-Morganschen Regeln"' wurden nach dem englischen Mathematiker Augustus De Morgan[11] benannt.
Als Abkürzung schreibt man statt $X_1 \cup X_2 \cup \cdots \cup X_n$ (bzw. $X_1 \cap X_2 \cap \cdots \cap X_n$) einfach $\bigcup_{i=1}^{n} X_i$ (bzw. $\bigcap_{i=1}^{n} X_i$). Möchte man alle Mengen X_i mit $i \in \mathbb{N}$ schneiden (bzw. vereinigen), so schreibt man kurz $\bigcap_{i\in\mathbb{N}} X_i$ (bzw. $\bigcup_{i\in\mathbb{N}} X_i$).
Oft benötigt man eine Verknüpfung von zwei Mengen, eine solche Verknüpfung wird allgemein wie folgt definiert:

Definition 43 ("'Verknüpfung von Mengen"'): *Seien A und B zwei Mengen und "'$\odot$"' eine beliebige Verknüpfung zwischen den Elementen dieser Mengen, dann definieren wir*

$$A \odot B =_{\text{def}} \{a \odot b \mid a \in A \text{ und } b \in B\}.$$

Example 44: *Die Menge $3\mathbb{Z} = \{0, \pm 3, \pm 6, \pm 9, \dots\}$ enthält alle Vielfachen[12] von 3, damit ist $3\mathbb{Z} + \{1\} = \{1, 4, -2, 7, -5, 10, -8, \dots\}$. Die Menge $3\mathbb{Z} + \{1\}$ schreibt man kurz oft auch als $3\mathbb{Z} + 1$, wenn klar ist, was mit dieser Abkürzung gemeint ist.*

C.1.5. Tupel (Vektoren) und das Kreuzprodukt

Seien $A, A_1, \dots, A_n$ im folgenden Mengen, dann bezeichnet

- $(a_1, \dots, a_n) =_{\text{def}}$ die Elemente $a_1, \dots, a_n$ in genau dieser festgelegten *Reihenfolge* und z.B. $(3, 2) \neq (2, 3)$. Wir sprechen von einem n-Tupel.
- $A_1 \times A_2 \times \cdots \times A_n =_{\text{def}} \{(a_1, \dots, a_n) \mid a_1 \in A_1, a_2 \in A_2, \dots, a_n \in A_n\}$ (Kreuzprodukt der Mengen $A_1, A_2, \dots, A_n$),
- $A^n =_{\text{def}} \underbrace{A \times A \times \cdots \times A}_{n\text{-mal}}$ (n-faches Kreuzprodukt der Menge A) und
- speziell gilt $A^1 = \{(a) \mid a \in A\}$.

[11] ⋆1806 in Madurai, Tamil Nadu, Indien - †1871 in London, England
[12] Eigentlich müsste man statt $3\mathbb{Z}$ die Notation $\{3\}\mathbb{Z}$ verwenden. Dies ist allerdings unüblich.

Wir nennen 2-Tupel auch *Paare*, 3-Tupel auch *Tripel*, 4-Tupel auch *Quadrupel* und 5-Tupel *Quintupel*. Bei n-Tupeln ist, im Gegensatz zu Mengen, eine Reihenfolge vorgegeben, d.h. es gilt z.B. immer $\{a, b\} = \{b, a\}$, aber im Allgemeinen $(a, b) \neq (b, a)$.

Example 45: *Sei $A = \{1, 2, 3\}$ und $B = \{a, b, c\}$, dann bezeichnet das Kreuzprodukt von A und B die Menge von Paaren $A \times B = \{(1, a), (1, b), (1, c), (2, a), (2, b), (2, c), (3, a), (3, b), (3, c)\}$.*

C.1.6. Die Anzahl von Elementen in Mengen

Sei A eine Menge, die endlich viele Elemente[13] enthält, dann ist

$$\#A =_{\text{def}} \text{Anzahl der Elemente in der Menge } A.$$

Beispielsweise ist $\#\{4, 7, 9\} = 3$. Mit dieser Definition gilt

- $\#(A^n) = (\#A)^n$,
- $\#\mathcal{P}(A) = 2^{\#A}$,
- $\#A + \#B = \#(A \cup B) + \#(A \cap B)$ und
- $\#A = \#(A \setminus B) + \#(A \cap B)$.

C.2. Relationen und Funktionen

C.2.1. Eigenschaften von Relationen

Seien $A_1, \ldots, A_n$ beliebige Mengen, dann ist R eine *n-stellige Relation* gdw. $R \subseteq A_1 \times A_2 \times \cdots \times A_n$. Eine zweistellige Relation nennt man auch *binäre Relation*. Oft werden auch Relationen $R \subseteq A^n$ betrachtet, diese bezeichnet man dann als n-stellige Relation über der Menge A.

Definition 46: *Sei R eine zweistellige Relation über A, dann ist R*

- reflexiv *gdw. $(a, a) \in R$ für alle $a \in A$,*
- symmetrisch *gdw. aus $(a, b) \in R$ folgt $(b, a) \in R$,*
- antisymmetrisch *gdw. aus $(a, b) \in R$ und $(b, a) \in R$ folgt $a = b$,*
- transitiv *gdw. aus $(a, b) \in R$ und $(b, c) \in R$ folgt $(a, c) \in R$ und*
- linear *gdw. es gilt immer $(a, b) \in R$ oder $(b, a) \in R$.*

Definition 47: *Sei R eine zweistellige Relation, dann*

- *heißt R* Halbordnung *gdw. R ist reflexiv, antisymmetrisch und transitiv,*
- Ordnung *gdw. R ist eine lineare Halbordnung und*
- Äquivalenzrelation *gdw. R reflexiv, transitiv und symmetrisch ist.*

Example 48: *Die Teilmengenrelation "'$\subseteq$"' auf allen Teilmengen von $\mathbb{Z}$ ist eine Halbordnung, aber keine Ordnung.*

[13]Solche Mengen werden als *endliche Mengen* bezeichnet.

Example 49: *Wir schreiben* $a \equiv b \bmod n$*, falls es eine ganze Zahl* q *gibt, für die* $a - b = qn$ *gilt. Für* $n \geq 2$ *ist die Relation* $R_n(a,b) =_{\text{def}} \{(a,b) \mid a \equiv b \bmod n\} \subseteq \mathbb{Z}^2$ *eine Äquivalenzrelation.*

C.2.2. Eigenschaften von Funktionen

Seien A und B beliebige Mengen. f ist eine *Funktion* von A nach B (Schreibweise: $f\colon A \to B$) gdw. $f \subseteq A \times B$ und für jedes $a \in A$ gibt es *höchstens* ein $b \in B$ mit $(a,b) \in f$. Ist also $(a,b) \in f$, so schreibt man $f(a) = b$. Ebenfalls gebrächlich ist die Notation $a \mapsto b$.

Remark 50: *Unsere Definition von Funktion umfasst auch mehrstellige Funktionen. Seien* C *und* B *Mengen und* $A = C^n$ *das* n*-fache Kreuzprodukt von* C*. Die Funktion* $f\colon A \to B$ *ist dann eine* n*-stellige Funktion, denn sie bildet* n*-Tupel aus* C^n *auf Elemente aus* B *ab.*

Definition 51: *Sei* f *eine* n*-stellige Funktion. Möchte man die Funktion* f *benutzen, aber keine Namen für die Argumente vergeben, so schreibt man auch*

$$f(\underbrace{\cdot,\cdot,\ldots,\cdot}_{n\text{-mal}})$$

Ist also der Namen des Arguments einer einstelligen Funktion $g(x)$ *für eine Betrachtung unwichtig, so kann man* $g(\cdot)$ *schreiben, um anzudeuten, dass* g *einstellig ist, ohne dies weiter zu erwähnen.*

Definition 52: *Sei nun* $R \subseteq A_1 \times A_2 \times \cdots \times A_n$ *eine* n*-stellige Relation, dann definieren wir eine Funktion* $P_R^n\colon A_1 \times A_2 \times \cdots \times A_n \to \{0,1\}$ *wie folgt:*

$$P_R^n(x_1,\ldots,x_n) =_{\text{def}} \begin{cases} 1, & \text{falls } (x_1,\ldots,x_n) \in R \\ 0, & \text{sonst} \end{cases}$$

Eine solche n*-stellige Funktion, die "'anzeigt"', ob ein Element aus* $A_1 \times A_2 \times \cdots \times A_n$ *entweder zu* R *gehört oder nicht, nennt man (*n*-stelliges)* Prädikat*.*

Example 53: *Sei* $\mathbb{P} =_{\text{def}} \{n \in \mathbb{N} \mid n \text{ ist Primzahl}\}$*, dann ist* $\mathbb{P}$ *eine* 1*-stellige Relation über den natürlichen Zahlen. Das Prädikat* $P_{\mathbb{P}}^1(n)$ *liefert für eine natürliche Zahl* n *genau dann* 1*, wenn* n *eine Primzahl ist.*

Ist für ein Prädikat P_R^n sowohl die Relation R als auch die Stelligkeit n aus dem Kontext klar, dann schreibt man auch kurz P oder verwendet das Relationensymbol R als Notation für das Prädikat P_R^n.

Nun legen wir zwei spezielle Funktionen fest, die oft sehr hilfreich sind:

Definition 54: *Sei* $\alpha \in \mathbb{R}$ *eine beliebige reelle Zahl, dann gilt*

- $\lceil \alpha \rceil =_{\text{def}}$ *die kleinste ganze Zahl, die größer oder gleich* α *ist (*$\triangleq$ *"'Aufrunden"')*
- $\lfloor \alpha \rfloor =_{\text{def}}$ *die größte ganze Zahl, die kleiner oder gleich* α *ist (*$\triangleq$ *"'Abrunden"')*

Definition 55: *Für eine beliebige Funktion* f *legen wir fest:*

- *Der* Definitionsbereich *von* f *ist* $D_f =_{\text{def}} \{a \mid \text{es gibt ein } b \text{ mit } f(a) = b\}$*.*
- *Der* Wertebereich *von* f *ist* $W_f =_{\text{def}} \{b \mid \text{es gibt ein } a \text{ mit } f(a) = b\}$*.*

- *Die Funktion* $f: A \to B$ *ist* total *gdw.* $D_f = A$.
- *Die Funktion* $f: A \to B$ *heißt* surjektiv *gdw.* $W_f = B$.
- *Die Funktion* f *heißt* injektiv *(oder eineindeutig*[14]*) gdw. immer wenn* $f(a_1) = f(a_2)$ *gilt auch* $a_1 = a_2$.
- *Die Funktion* f *heißt* bijektiv *gdw.* f *ist injektiv und surjektiv.*

Mit Hilfe der Kontraposition (siehe Abschnitt E.1.1) kann man für die Injektivität alternativ auch zeigen, dass immer wenn $a_1 \neq a_2$, dann muss auch $f(a_1) \neq f(a_2)$ gelten.

Example 56: *Sei die Funktion* $f: \mathbb{N} \to \mathbb{Z}$ *durch* $f(n) = (-1)^n \lceil \frac{n}{2} \rceil$ *gegeben. Die Funktion* f *ist surjektiv, denn* $f(0) = 0, f(1) = -1, f(2) = 1, f(3) = -2, f(4) = 2, \ldots$, *d.h. die ungeraden natürlichen Zahlen werden auf die negativen ganzen Zahlen abgebildet, die geraden Zahlen aus* $\mathbb{N}$ *werden auf die positiven ganzen Zahlen abgebildet und deshalb ist* $W_f = \mathbb{Z}$.
Weiterhin ist f *auch injektiv, denn aus*[15] $(-1)^{a_1} \lceil \frac{a_1}{2} \rceil = (-1)^{a_2} \lceil \frac{a_2}{2} \rceil$ *folgt, dass entweder* a_1 *und* a_2 *gerade oder* a_1 *und* a_2 *ungerade, denn sonst würden auf der linken und rechten Seite der Gleichung unterschiedliche Vorzeichen auftreten. Ist* a_1 *gerade und* a_2 *gerade, dann gilt* $\lceil \frac{a_1}{2} \rceil = \lceil \frac{a_2}{2} \rceil$ *und auch* $a_1 = a_2$. *Sind* a_1 *und* a_2 *ungerade, dann gilt* $-\lceil \frac{a_1}{2} \rceil = -\lceil \frac{a_2}{2} \rceil$, *woraus auch folgt, dass* $a_1 = a_2$. *Damit ist die Funktion* f *bijektiv. Weiterhin ist* f *auch total, d.h.* $D_f = \mathbb{N}$.

Definition 57: *Unter einem* n*-stelligen* Operator f *(auf der Menge* Y*) versteht man in der Mathematik eine Funktion der Form* $f: Y^n \to Y$. *Einfache Beispiele für zweistellige Operatoren sind der Additions- oder Multiplikationsoperator.*

C.2.3. Hüllenoperatoren

Definition 58: *Sei* X *eine Menge. Ein einstelliger Operator* $\Psi: \mathcal{P}(X) \to \mathcal{P}(X)$ *heißt* Hüllenoperator, *wenn er die folgenden drei Eigenschaften erfüllt:*

Einbettung: *für alle* $A \in \mathcal{P}(X)$ *gilt* $A \subseteq \Psi(A)$

Monotonie: *für alle* $A, B \in \mathcal{P}(X)$ *mit* $A \subseteq B$ *folgt* $\Psi(A) \subseteq \Psi(B)$

Abgeschlossenheit: *für alle* $A \in \mathcal{P}(X)$ *gilt* $\Psi(\Psi(A)) = \Psi(A)$

Aufgrund der Monotonieeigenschaft eines Hüllenoperators kann man bei der Abgeschlossenheit die Eigenschaft $\Psi(\Psi(A)) = \Psi(A)$ auch durch $\Psi(\Psi(A)) \subseteq \Psi(A)$ ersetzen. In der Informatik spielen Hüllenoperatoren eine große Rolle. Gute Beispiele hierfür sind z.B. die *transitive Hülle* (vgl. Computergraphik), die Kleene-Hülle (vgl. Formale Sprachen) oder der Abschluss einer Komplexitätsklasse unter Schnitt oder Vereinigung.

[14] Achtung: Dieser Begriff wird manchmal unterschiedlich, je nach Autor, in den Bedeutungen "'bijektiv"' oder "'injektiv"' verwendet.
[15] Für die Definition der Funktion $\lceil \cdot \rceil$ siehe Definition 54.

C.2.4. Permutationen

Sei S eine beliebige endliche Menge, dann heißt eine bijektive Funktion π der Form $\pi\colon S \to S$ *Permutation.* Das bedeutet, dass die Funktion π Elemente aus S wieder auf Elemente aus S abbildet, wobei für jedes $b \in S$ ein $a \in S$ mit $f(a) = b$ existiert (Surjektivität) und falls $f(a_1) = f(a_2)$ gilt, dann ist $a_1 = a_2$ (Injektivität).

Remark 59: *Man kann den Permutationsbegriff auch auf unendliche Mengen erweitern, aber besonders häufig werden in der Informatik* Permutationen von endlichen Mengen *benötigt. Aus diesem Grund sollen hier nur endliche Mengen S betrachtet werden.*

Sei nun $S = \{1, \ldots, n\}$ (eine endliche Menge) und $\pi\colon \{1, \ldots, n\} \to \{1, \ldots, n\}$ eine Permutation. Permutationen dieser Art kann man sehr anschaulich mit Hilfe einer Matrix aufschreiben:

$$\pi = \begin{pmatrix} 1 & 2 & \ldots & n \\ \pi(1) & \pi(2) & \ldots & \pi(n) \end{pmatrix}$$

Durch diese Notation wird klar, dass das Element 1 der Menge S durch das Element $\pi(1)$ ersetzt wird, das Element 2 wird mit $\pi(2)$ vertauscht und allgemein das Element i durch $\pi(i)$ für $1 \leq i \leq n$. In der zweiten Zeile dieser Matrixnotation findet sich also *jedes* (Surjektivität) Element der Menge S genau *einmal* (Injektivität).

Example 60: *Sei $S = \{1, \ldots, 3\}$ eine Menge mit drei Elementen. Dann gibt es, wie man ausprobieren kann, genau* 6 *Permutationen von S:*

$$\begin{array}{llllll} \pi_1 & = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix} & \pi_2 & = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix} & \pi_3 & = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix} \\ \pi_4 & = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} & \pi_5 & = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} & \pi_6 & = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} \end{array}$$

Theorem 61: *Sei S eine endliche Menge mit $n = |S|$, dann gibt es genau $n!$ (Fakultät) verschiedene Permutationen von S.*

Beweis: Jede Permutation π der Menge S von n Elementen kann als Matrix der Form

$$\pi = \begin{pmatrix} 1 & 2 & \ldots & n \\ \pi(1) & \pi(2) & \ldots & \pi(n) \end{pmatrix}$$

aufgeschrieben werden. Damit ergibt sich die Anzahl der Permutationen von S durch die Anzahl der verschiedenen zweiten Zeilen solcher Matrizen. In jeder solchen Zeile muss jedes der n Elemente von S genau einmal vorkommen, da π eine bijektive Abbildung ist, d.h. wir haben für die erste Position der zweiten Zeile der Matrixdarstellung genau n verschiedene Möglichkeiten, für die zweite Position noch $n-1$ und für die dritte noch $n-2$. Für die n-te Position bleibt nur noch 1 mögliches Element aus S übrig[16]. Zusammengenommen haben wir also $n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \ldots \cdot 2 \cdot 1 = n!$ verschiedene mögliche Permutationen der Menge S. #

[16]Dies kann man sich auch als die Anzahl der verschiedenen Möglichkeiten vorstellen, die bestehen, wenn man aus einer Urne mit n numerierten Kugeln alle Kugeln *ohne* Zurücklegen nacheinander zieht.

C.3. Summen und Produkte

C.3.1. Summen

Zur abkürzenden Schreibweise verwendet man für Summen das Summenzeichen $\sum$. Dabei ist

$$\sum_{i=1}^{n} a_i =_{\text{def}} a_1 + a_2 + \cdots + a_n.$$

Mit Hilfe dieser Definition ergeben sich auf elementare Weise die folgenden Rechenregeln:

- Sei $a_i = a$ für $1 \leq i \leq n$, dann gilt $\sum\limits_{i=1}^{n} a_i = n \cdot a$ (Summe gleicher Summanden).
- $\sum\limits_{i=1}^{n} a_i = \sum\limits_{i=1}^{m} a_i + \sum\limits_{i=m+1}^{n} a_i$, wenn $1 < m < n$ (Aufspalten einer Summe).
- $\sum\limits_{i=1}^{n} (a_i + b_i + c_i + \ldots) = \sum\limits_{i=1}^{n} a_i + \sum\limits_{i=1}^{n} b_i + \sum\limits_{i=1}^{n} c_i + \ldots$ (Addition von Summen).
- $\sum\limits_{i=1}^{n} a_i = \sum\limits_{i=l}^{n+l-1} a_{i-l+1}$ und $\sum\limits_{i=l}^{n} a_i = \sum\limits_{i=1}^{n-l+1} a_{i+l-1}$ (Umnumerierung von Summen).
- $\sum\limits_{i=1}^{n} \sum\limits_{j=1}^{m} a_{i,j} = \sum\limits_{j=1}^{m} \sum\limits_{i=1}^{n} a_{i,j}$ (Vertauschen der Summationsfolge).

Manchmal verwendet man keine Laufindizes an ein Summenzeichen, sondern man beschreibt (durch ein Prädikat) welche Zahlen aufsummiert werden sollen. So kann man eine Funktion definieren, die die Summe aller Teiler einer natürlichen Zahl liefert:

$$\sigma(n) =_{\text{def}} \sum_{\substack{t \leq n \\ t \text{ teilt } n}} t$$

C.3.2. Produkte

Zur abkürzenden Schreibweise verwendet man für Produkte das Produktzeichen $\prod$. Dabei ist

$$\prod_{i=1}^{n} a_i =_{\text{def}} a_1 \cdot a_2 \cdot \ldots \cdot a_n.$$

Mit Hilfe dieser Definition ergeben sich auf elementare Weise die folgenden Rechenregeln:

- Sei $a_i = a$ für $1 \leq i \leq n$, dann gilt $\prod\limits_{i=1}^{n} a_i = a^n$ (Produkt gleicher Faktoren).
- $\prod\limits_{i=1}^{n} (ca_i) = c^n \prod\limits_{i=1}^{n} a_i$ (Vorziehen von konstanten Faktoren)
- $\prod\limits_{i=1}^{n} a_i = \prod\limits_{i=1}^{m} a_i \cdot \prod\limits_{i=m+1}^{n} a_i$, wenn $1 < m < n$ (Aufspalten in Teilprodukte).
- $\prod\limits_{i=1}^{n} (a_i \cdot b_i \cdot c_i \cdot \ldots) = \prod\limits_{i=1}^{n} a_i \cdot \prod\limits_{i=1}^{n} b_i \cdot \prod\limits_{i=1}^{n} c_i \cdot \ldots$ (Das Produkt von Produkten).

- $\prod_{i=1}^{n} a_i = \prod_{i=l}^{n+l-1} a_{i-l+1}$ und $\prod_{i=l}^{n} a_i = \prod_{i=1}^{n-l+1} a_{i+l-1}$ (Umnumerierung von Produkten).
- $\prod_{i=1}^{n} \prod_{j=1}^{m} a_{i,j} = \prod_{j=1}^{m} \prod_{i=1}^{n} a_{i,j}$ (Vertauschen der Reihenfolge bei Doppelprodukten).

Ähnlich wie bei Summen kann man bei Produkten auch ohne Laufindex arbeiten. So ist z.B. die *Eulersche ϕ-Funktion* wie folgt definiert:

$$\phi(n) =_{\text{def}} n \prod_{p|n} (1 - \frac{1}{p})$$

In das Produkt gehen also alle Primteiler p von n mit dem Faktor $1 - 1/p$ ein.
Oft werden Summen- oder Produktsymbole verwendet bei denen der Startindex größer als der Stopindex ist. Solche Summen bzw. Produkte sind "‘leer"’, d.h. es wird nichts summiert bzw. multipliziert. Sind dagegen Start- und Endindex gleich, so tritt nur genau ein Wert auf, d.h. das Summen- bzw. Produktsymbol hat in diesem Fall keine Auswirkung. Es ergeben sich also die folgenden Rechenregeln:

- Seien $n, m \in \mathbb{Z}$ und $n < m$, dann

$$\sum_{i=m}^{n} a_i = 0 \text{ und } \prod_{i=m}^{n} a_i = 1$$

- Sei $n \in \mathbb{Z}$, dann

$$\sum_{i=n}^{n} a_i = a_n = \prod_{i=n}^{n} a_i$$

Example 62: *Die folgende Identität wird Euler zugeschrieben. Erstaunlicherweise kann man damit eine Summe von Kehrwerten von Potenzen mit einem Produkt von Primzahlen in Verbindung bringen.*

$$\sum_{n=1}^{\infty} \frac{1}{n^s} = \prod_{p} \frac{1}{1 - \frac{1}{p^s}}$$

Diese Identiät kann man Die Summe auf der linken Seite ist auch als Riemannsche Zetafuntion $\zeta(s)$ *bekannt. Erstaunlicherweise gilt dann der folgende Zusammenhang*

$$\zeta(2) = \sum_{n=1}^{\infty} \frac{1}{n^2} = \prod_{p} \frac{1}{1 - \frac{1}{p^2}} = \frac{\pi^2}{6},$$

denn durch diese Gleichung wird die Menge der Primzahlen in Beziehung zur Kreiszahl π gebracht.

C.4. Logarithmieren, Potenzieren und Radizieren

Die Schreibweise a^b ist eine Abkürzung für

$$a^b =_{\text{def}} \underbrace{a \cdot a \cdot \ldots \cdot a}_{b-\text{mal}}$$

und wird als *Potenzierung* bezeichnet. Dabei ist a die *Basis*, b der *Exponent* und a^b die b-te *Potenz* von a. Seien nun $r, s, t \in \mathbb{R}$ und $r, t \geq 0$ durch die folgende Gleichung verbunden:

$$r^s = t.$$

Dann lässt sich diese Gleichung wie folgt umstellen und es gelten die folgenden Rechenregeln:

Logarithmieren	Potenzieren	Radizieren
$\mathbf{s} = \log_{\mathbf{r}} \mathbf{t}$	$\mathbf{t} = \mathbf{r}^{\mathbf{s}}$	$\mathbf{r} = \sqrt[\mathbf{s}]{\mathbf{t}}$
i) $\log_r(\frac{u}{v}) = \log_r u - \log_r v$	i) $r^u \cdot r^v = r^{u+v}$	i) $\sqrt[s]{u} \cdot \sqrt[s]{v} = \sqrt[s]{u \cdot v}$
ii) $\log_r(u \cdot v) = \log_r u + \log_r v$	ii) $\frac{r^u}{r^v} = r^{u-v}$	ii) $\frac{\sqrt[s]{u}}{\sqrt[s]{v}} = \sqrt[s]{\left(\frac{u}{v}\right)}$
iii) $\log_r(t^u) = u \cdot \log_r t$	iii) $u^s \cdot v^s = (u \cdot v)^s$	iii) $\sqrt[u]{\sqrt[v]{t}} = \sqrt[u \cdot v]{t}$
iv) $\log_r(\sqrt[u]{t}) = \frac{1}{u} \cdot \log_r t$	iv) $\frac{u^s}{v^s} = \left(\frac{u}{v}\right)^s$	
v) $\frac{\log_r t}{\log_r u} = \log_u t$ (Basiswechsel)	v) $(r^u)^v = r^{u \cdot v}$	

Zusätzlich gilt: Wenn $r > 1$, dann ist $s_1 < s_2$ gdw. $r^{s_1} < r^{s_2}$ (Monotonie).
Da $\sqrt[s]{t} = t^{\left(\frac{1}{s}\right)}$ gilt, können die Gesetze für das Radizieren leicht aus den Potenziergesetzen abgeleitet werden. Weiterhin legen wir spezielle Schreibweisen für die Logarithmen zur Basis 10, e (Eulersche Zahl) und 2 fest: $\lg t =_{\text{def}} \log_{10} t$, $\ln t =_{\text{def}} \log_e t$ und $\operatorname{lb} t =_{\text{def}} \log_2 t$.

C.5. Gebräuchliche griechische Buchstaben

In der Informatik, Mathematik und Physik ist es üblich, griechische Buchstaben zu verwenden. Ein Grund hierfür ist, dass es so möglich wird mit einer größeren Anzahl von Unbekannten arbeiten zu können, ohne unübersichtliche und oft unhandliche Indizes benutzen zu müssen.

Kleinbuchstaben:

Symbol	Bezeichnung	Symbol	Bezeichnung	Symbol	Bezeichnung
α	Alpha	β	Beta	γ	Gamma
δ	Delta	ϕ	Phi	φ	Phi
ξ	Xi	ζ	Zeta	ϵ	Epsilon
θ	Theta	λ	Lambda	π	Pi
σ	Sigma	η	Eta	μ	Mu

Großbuchstaben:

Symbol	Bezeichnung	Symbol	Bezeichnung	Symbol	Bezeichnung
Γ	Gamma	Δ	Delta	Φ	Phi
Ξ	Xi	Θ	Theta	Λ	Lambda
Π	Pi	Σ	Sigma	Ψ	Psi
Ω	Omega				

D. Einige (wenige) Grundlagen der elementaren Logik

Aussagen sind entweder *wahr* ($\triangleq 1$) oder *falsch* ($\triangleq 0$). So sind die Aussagen

"'Wiesbaden liegt am Mittelmeer"' und "'$1 = 7$"'

sicherlich falsch, wogegen die Aussagen

"'Wiesbaden liegt in Hessen"' und "'$11 = 11$"'

sicherlich wahr sind. Aussagen werden meist durch *Aussagenvariablen* formalisiert, die nur die Werte 0 oder 1 annehmen können. Oft verwendet man auch eine oder mehrere Unbekannte, um eine Aussage zu parametrisieren. So könnte "'$P(x)$"' etwa für "'Wiesbaden liegt im Bundesland x"' stehen, d.h. "'$P(\text{Hessen})$"' wäre wahr, wogegen "'$P(\text{Bayern})$"' eine falsche Aussage ist. Solche Konstrukte mit Parameter nennt man auch *Prädikat* oder *Aussageformen*.
Um die Verknüpfung von Aussagen auch formal aufschreiben zu können, werden die folgenden logischen Operatoren verwendet

Symbol	umgangssprachlicher Name	Name in der Logik
$\wedge$	und	Konjunktion
$\vee$	oder	Disjunktion / Alternative
$\neg$	nicht	Negation
$\rightarrow$	folgt	Implikation
$\leftrightarrow$	genau dann wenn (*gdw.*)	Äquivalenz

Zusätzlich werden noch die Quantoren $\exists$ ("'es existiert"') und $\forall$ ("'für alle"') verwendet, die z.B. wie folgt gebraucht werden können

$\forall x\colon P(x)$ bedeutet "'Für alle x gilt die Aussage $P(x)$"'.

$\exists x\colon P(x)$ bedeutet "'Es existiert ein x, für das die Aussage $P(x)$ gilt"'.

Üblicherweise läßt man sogar den Doppelpunkt weg und schreibt statt $\forall x\colon P(x)$ vereinfachend $\forall x P(x)$. Die Aussageform $P(x)$ wird auch als Prädikat (siehe Definition 52 auf Seite 60) bezeichnet, weshalb man von *Prädikatenlogik* spricht.

Example 63: *Die Aussage "'Jede gerade natürliche Zahl kann als Produkt von 2 und einer anderen natürlichen Zahl geschrieben werden"' lässt sich dann wie folgt schreiben*

$$\forall n \in \mathbb{N}\colon ((n \textit{ ist gerade}) \rightarrow (\exists m \in \mathbb{N}\colon n = 2 \cdot m))$$

Die folgende logische Formel wird wahr gdw. n eine ungerade natürliche Zahl ist.

$$\exists m \in \mathbb{N}\colon (n = 2 \cdot m + 1)$$

Für die logischen Konnektoren sind die folgenden Wahrheitswertetafeln festgelegt:

p	$\neg p$
0	1
1	0

und

p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Jetzt kann man Aussagen auch etwas komplexer verknüpfen:

Example 64: *Nun wird der $\wedge$-Operator verwendet werden. Dazu soll die Aussage "'Für alle natürlichen Zahlen n und m gilt, wenn n kleiner gleich m und m kleiner gleich n gilt, dann ist m gleich n"'*

$$\forall n, m \in \mathbb{N} \left(((n \leq m) \wedge (m \leq n)) \rightarrow (n = m)\right)$$

Oft benutzt man noch den negierten Quantor $\nexists$ ("'es existiert kein"').

Cubum autem in duos cubos, aut quadrato-quadratum in duos quadrato-quadratos, et generaliter nullam in infinitum ultra quadratum potestatem in duos eiusdem nominis fas est dividere cuius rei demonstrationem mirabilem sane detexi. Hanc marginis exiguitas non caperet.

Example 65 ("'Großer Satz von Fermat"'): *Die Richtigkeit dieser Aussage konnte erst* 1994 *nach mehr als* 350 *Jahren von Andrew Wiles und Richard Taylor gezeigt werden:*

$$\forall n \in \mathbb{N} \nexists a, b, c \in \mathbb{N} \left(((n > 2) \wedge (a \cdot b \cdot c \neq 0)) \rightarrow a^n + b^n = c^n\right)$$

Für den Fall $n = 2$ hat die Gleichung $a^n + b^n = c^n$ unendlich viele ganzzahlige Lösungen (die so genannten Pythagoräischen Zahlentripel) wie z.B. $3^2 + 4^2 = 5^2$. Diese sind seit mehr als 3500 *Jahren bekannt und haben z.B. geholfen die Cheops-Pyramide zu bauen.*

E. Einige formale Grundlagen von Beweistechniken

Praktisch arbeitende Informatiker glauben oft völlig ohne (formale) Beweistechniken auskommen zu können. Dabei meinen sie sogar, dass formale Beweise keinerlei Berechtigung in der Praxis der Informatik haben und bezeichnen solches Wissen als "‘in der Praxis irrelevantes Zeug, das nur von und für seltsame Wissenschaftler erfunden wurde"’. Studenten in den ersten Semestern unterstellen sogar oft, dass mathematische Grundlagen und Beweistechniken nur als "‘Filter"’ dienen, um die Anzahl der Studenten zu reduzieren. Oft stellen sich beide Gruppen auf den Standpunkt, dass die Korrektheit von Programmen und Algorithmen durch "‘Lassen wir es doch mal laufen und probieren es aus!"’ ($\triangleq$ Testen) belegt werden könne[17]. Diese Einstellung zeigt sich oft auch darin, dass Programme mit Hilfe einer IDE schnell "‘testweise"’ übersetzt werden, in der Hoffnung oder (wesentlich schlimmer) in der Überzeugung, dass jedes übersetzbare Programm ($\triangleq$ syntaktisch korrekt) automatisch auch semantisch korrekt ist. Das diese Einstellung völlig falsch ist zeigen Myriaden in der Praxis auftretende Softwarefehler eindrücklich.

Theoretiker, die sich mit den Grundlagen der Informatik beschäftigen, vertreten oft den Standpunkt, dass die Korrektheit *jedes* Programms rigoros *bewiesen* werden muss. Wahrscheinlich ist die Position zwischen diesen beiden Extremen richtig, denn zum einen ist der formale Beweis von (großen) Programmen oft nicht praktikabel oder aufgrund der enormen Komplexität nicht möglich und zum anderen kann das Testen mit einer (relativ kleinen) Menge von Eingaben sicherlich nicht belegen, dass ein Programm vollständig den Spezifikationen entspricht. Im praktischen Einsatz ist es dann oft mit Eingaben konfrontiert, die zu einer fehlerhaften Reaktion führen oder es sogar abstürzen[18] lassen. Bei einfacher Anwendersoftware sind solche Fehler ärgerlich, aber oft zu verschmerzen. Bei sicherheitskritischer Software (z.B. bei der Regelung von Atomkraftwerken, Airbags und Bremssystemen in Autos, in der Medizintechnik, bei Finanztransaktionssystemen oder bei der Steuerung von Raumsonden) gefährden solche Fehler menschliches Leben oder führen zu extrem hohen finanziellen Verlusten und müssen deswegen unbedingt vermieden werden.

Für den Praktiker bringen Kenntnisse über formale Beweise aber noch andere Vorteile. Viele Beweise beschreiben direkt den zur Lösung benötigten Algorithmus, d.h. eigentlich wird die Richtigkeit einer Aussage durch die (implizite) Angabe eines Algorithmus gezeigt. Aber es gibt noch einen anderen Vorteil. Ist der umzusetzende Algorithmus komplex (z.B. aufgrund einer komplizierten Schleifenstruktur oder einer verschachtelten Rekursion), so ist es unwahrscheinlich, eine korrekte Implementation an den Kunden liefern zu können, ohne die Hintergründe ($\triangleq$ Beweis) verstanden zu haben. All dies zeigt, dass auch ein praktischer Informatiker Einblicke in Beweistechniken haben sollte. Interessanterweise zeigt die persönliche Erfahrung im praktischen Umfeld auch, dass solches (theoretisches) Wissen über die Hintergründe oft zu klarer strukturierten und effizienteren Programmen führt.

Aus diesen Gründen sollen in den folgenden Abschnitten einige grundlegende Beweistechniken mit Hilfe von Beispielen (unvollständig) kurz vorgestellt werden.

[17] Diese Bemerkung ist natürlich etwas polemisch, da *richtiges* Testen natürlich die Qualität einer Software verbessert. Weiterhin werden solche Tests systematisch durchgeführt und haben somit eine ganz andere Qualität als "‘herumprobieren"’.

[18] Dies wird eindrucksvoll durch viele Softwarepakete und verbreitete Betriebssysteme im PC-Umfeld belegt.

E.1. Direkte Beweise

Um einen *direkten Beweis* zu führen, müssen wir, beginnend von einer initialen Aussage ($\triangleq$ Hypothese), durch Angabe einer Folge von (richtigen) Zwischenschritten zu der zu beweisenden Aussage ($\triangleq$ Folgerung) gelangen. Jeder Zwischenschritt ist dabei entweder unmittelbar klar oder muss wieder durch einen weiteren (kleinen) Beweis belegt werden. Dabei müssen nicht alle Schritte völlig formal beschrieben werden, sondern es kommt darauf an, dass sich dem Leser die eigentliche Strategie erschließt.

Theorem 66: *Sei $n \in \mathbb{N}$. Falls $n \geq 4$, dann ist $2^n \geq n^2$.*

Wir müssen also, in Abhängigkeit des Parameters n, die Richtigkeit dieser Aussage belegen. Einfaches Ausprobieren ergibt, dass $2^4 = 16 \geq 16 = 4^2$ und $2^5 = 32 \geq 25 = 5^2$, d.h. intuitiv scheint die Aussage richtig zu sein. Wir wollen die Richtigkeit der Aussage nun durch eine Reihe von (kleinen) Schritten belegen:
Beweis:
Wir haben schon gesehen, dass die Aussage für $n = 4$ und $n = 5$ richtig ist. Erhöhen wir n auf $n + 1$, so verdoppelt sich der Wert der linken Seite der Ungleichung von 2^n auf $2 \cdot 2^n = 2^{n+1}$. Für die rechte Seite ergibt sich ein Verhältnis von $(\frac{n+1}{n})^2$. Je größer n wird, desto kleiner wird der Wert $\frac{n+1}{n}$, d.h. der maximale Wert ist bei $n = 4$ mit 1.25 erreicht. Wir wissen $1.25^2 = 1.5625$, d.h. immer wenn wir n um eins erhöhen, verdoppelt sich der Wert der linken Seite, wogegen sich der Wert der rechten Seite um maximal das 1.5625-fache erhöht. Damit muss die linke Seite der Ungleichung immer größer als die rechte Seite sein. #

Dieser Beweis war nur wenig formal, aber sehr ausführlich und wurde am Ende durch das Symbol "‘#"’ markiert. Im Laufe der Zeit hat es sich eingebürgert, das Ende eines Beweises mit einem besonderen Marker abzuschließen. Besonders bekannt ist hier "‘qed"’, eine Abkürzung für die lateinische Floskel "‘quod erat demonstrandum"’, die mit "‘was zu beweisen war"’ übersetzt werden kann. In neuerer Zeit werden statt "‘qed"’ mit der gleichen Bedeutung meist die Symbole "‘$\Box$"’ oder "‘#"’ verwendet.
Nun stellt sich die Frage: "‘Wie formal und ausführlich muss ein Beweis sein?"’ Diese Frage kann so einfach nicht beantwortet werden, denn das hängt u.a. davon ab, welche Lesergruppe durch den Beweis von der Richtigkeit einer Aussage überzeugt werden soll und wer den Beweis schreibt. Ein Beweis für ein Übungsblatt sollte auch auf Kleinigkeiten Rücksicht nehmen, wogegen ein solcher Stil für eine wissenschaftliche Zeitschrift vielleicht nicht angebracht wäre, da die potentielle Leserschaft über ganz andere Erfahrungen und viel mehr Hintergrundwissen verfügt.
Nun noch eine Bemerkung zum Thema "‘Formalismus"’. Die menschliche Sprache ist unpräzise, mehrdeutig und Aussagen können oft auf verschiedene Weise interpretiert werden, wie das tägliche Zusammenleben der Menschen und die "‘Juristerei"’ eindrucksvoll demonstriert. Diese Defizite sollen Formalismen[19] ausgleichen, d.h. die Antwort muss lauten: "‘So viele Formalismen wie notwendig und so wenige wie möglich!"’. Durch Übung und Praxis lernt man die Balance zwischen diesen Anforderungen zu halten und es zeigt sich bald, dass "‘Geübte"’ die formale Beschreibung sogar wesentlich leichter verstehen.

Oft kann man andere, schon bekannte, Aussagen dazu verwenden, die Richtigkeit einer neuen (evtl. kompliziert wirkenden) Aussage zu belegen.

[19] In diesem Zusammenhang sind Programmiersprachen auch Formalismen, die eine präzise Beschreibung von Algorithmen erzwingen und die durch einen Compiler verarbeitet werden können.

Theorem 67: *Sei $n \in \mathbb{N}$ die Summe von 4 Quadratzahlen, die größer als 0 sind, dann muss $2^n \geq n^2$ sein.*

Beweis: Die Menge der Quadratzahlen ist $\mathbb{S} = \{0, 1, 4, 9, 16, 25, 36, \dots\}$, d.h. 1 ist die kleinste Quadratzahl, die größer als 0 ist. Damit muss unsere Summe von 4 Quadratzahlen größer oder gleich als 4 sein. Die Aussage folgt direkt aus Satz 66. #

E.1.1. Die Kontraposition

Mit Hilfe von direkten Beweisen haben wir Zusammenhänge der Form "‘Wenn Aussage H richtig ist, dann folgt daraus die Aussage C"’ untersucht. Manchmal ist es schwierig, einen Beweis für eine solchen Zusammenhang zu finden. Völlig gleichwertig ist die Behauptung "‘Wenn die Aussage C falsch ist, dann ist die Aussage H falsch"’ und oft ist eine solche Aussage leichter zu zeigen.
Die *Kontraposition* von Satz 66 ist also die folgende Aussage: "‘Wenn nicht $2^n \geq n^2$, dann gilt nicht $n \geq 4$."’. Das entspricht der Aussage: "‘Wenn $2^n < n^2$, dann gilt $n < 4$."’, was offensichtlich zu der ursprünglichen Aussage von Satz 66 gleichwertig ist. Diese Technik ist oft besonders hilfreich, wenn man die Richtigkeit einer Aussage zeigen soll, die aus zwei Teilaussagen zusammengesetzt und die durch ein "‘genau dann wenn"’[20] verknüpft sind. In diesem Fall sind zwei Teilbeweise zu führen, denn zum einen muss gezeigt werden, dass aus der ersten Aussage die zweite folgt und umgekehrt muss gezeigt werden, dass aus der zweiten Aussage die erste folgt.

Theorem 68: *Eine natürliche Zahl n ist durch drei teilbar genau dann, wenn die Quersumme ihrer Dezimaldarstellung durch drei teilbar ist.*

Beweis: Für die Dezimaldarstellung von n gilt

$$n = \sum_{i=0}^{k} a_i \cdot 10^i, \text{wobei } a_i \in \{0, 1, \dots, 9\} \text{ ("‘Ziffern"’) und } 0 \leq i \leq k.$$

Mit $\mathrm{QS}(n)$ wird die Quersumme von n bezeichnet, d.h. $\mathrm{QS}(n) = \sum_{i=0}^{k} a_i$. Mit Hilfe einer einfachen vollständigen Induktion kann man zeigen, dass für jedes $i \geq 0$ ein $b \in \mathbb{N}$ existiert, sodass $10^i = 9b + 1$. Damit gilt $n = \sum_{i=0}^{k} a_i \cdot 10^i = \sum_{i=0}^{k} a_i(9b_i + 1) = \mathrm{QS}(n) + 9\sum_{i=0}^{k} a_i b_i$, d.h. es existiert ein $c \in \mathbb{N}$, so dass $n = \mathrm{QS}(n) + 9c$.

"‘$\Rightarrow$"’: Wenn n durch 3 teilbar ist, dann muss auch $\mathrm{QS}(n) + 9c$ durch 3 teilbar sein. Da $9c$ sicherlich durch 3 teilbar ist, muss auch $\mathrm{QS}(n) = n - 9c$ durch 3 teilbar sein.

"‘$\Leftarrow$"’: Dieser Fall soll durch Kontraposition gezeigt werden. Sei nun n nicht durch 3 teilbar, dann darf $\mathrm{QS}(n)$ nicht durch 3 teilbar sein, denn sonst wäre $n = 9c + \mathrm{QS}(n)$ durch 3 teilbar. #

E.2. Der Ringschluss

Oft findet man mehrere Aussagen, die zueinander äquivalent sind. Ein Beispiel dafür ist Satz 69. Um die Äquivalenz dieser Aussagen zu beweisen, müssten jeweils zwei "‘genau dann wenn"’ Beziehungen untersucht werden, d.h. es werden vier Teilbeweise notwendig. Dies kann mit Hilfe eines so genannten *Ringschlusses* abgekürzt werden, denn es reicht zu zeigen, dass aus der ersten Aussage die zweite folgt, aus der zweiten Aussage die dritte und dass schließlich aus der dritten Aussage wieder die erste folgt.

[20] Oft wird "‘genau dann wenn"’ durch *gdw.* abgekürzt.

Im Beweis zu Satz 69 haben wir deshalb nur drei anstatt vier Teilbeweise zu führen, was zu einer Arbeitsersparnis führt. Diese Arbeitsersparnis wird um so größer, je mehr äquivalente Aussagen zu untersuchen sind. Dabei ist die Reihenfolge der Teilbeweise nicht wichtig, solange die einzelnen Teile zusammen einen Ring bilden.

Theorem 69: *Seien A und B zwei beliebige Mengen, dann sind die folgenden drei Aussagen äquivalent:*

i) $A \subseteq B$

ii) $A \cup B = B$

iii) $A \cap B = A$

Beweis: Im folgenden soll ein Ringschluss verwendet werden, um die Äquivalenz der drei Aussagen zu zeigen:
"'i) $\Rightarrow$ ii)"': Da nach Voraussetzung $A \subseteq B$ ist, gilt für jedes Element $a \in A$ auch $a \in B$, d.h. in der Vereinigung $A \cup B$ sind alle Elemente nur aus B, was $A \cup B = B$ zeigt.

"'ii) $\Rightarrow$ iii)"': Wenn $A \cup B = B$ gilt, dann ergibt sich durch Einsetzen und mit den Regeln aus Abschnitt C.1.4 (Absorptionsgesetz) direkt $A \cap B = A \cap (A \cup B) = A$.

"'iii) $\Rightarrow$ i)"': Sei nun $A \cap B = A$, dann gibt es kein Element $a \in A$ für das $a \notin B$ gilt. Dies ist aber gleichwertig zu der Aussage $A \subseteq B$.
Damit hat sich ein Ring von Aussagen "'i) $\Rightarrow$ ii)"', "'ii) $\Rightarrow$ iii)"' und "'iii) $\Rightarrow$ i)"' gebildet, was die Äquivalenz aller Aussagen zeigt. #

E.3. Widerspruchsbeweise

Obwohl die Technik der Widerspruchsbeweise auf den ersten Blick sehr kompliziert erscheint, ist sie meist einfach anzuwenden, extrem mächtig und liefert oft sehr kurze Beweise. Angenommen wir sollen die Richtigkeit einer Aussage "'aus der Hypothese H folgt C"' zeigen. Dazu beweisen wir, dass sich ein Widerspruch ergibt, wenn wir, von H und der Annahme, dass C falsch ist, ausgehen. Also war die Annahme falsch, und die Aussage C muss richtig sein.
Anschaulicher wird diese Beweistechnik durch folgendes Beispiel: Nehmen wir einmal an, dass Alice eine bürgerliche Frau ist und deshalb auch keine Krone trägt. Es ist klar, dass jede Königin eine Krone trägt. Wir sollen nun beweisen, dass Alice keine Königin ist. Dazu nehmen wir an, dass Alice eine Königin ist, d.h. Alice trägt eine Krone. Dies ist ein Widerspruch! Also war unsere Annahme falsch, und wir haben gezeigt, dass Alice keine Königin sein kann.

Der Beweis zu folgendem Satz verwendet diese Technik:

Theorem 70: *Sei S eine endliche Untermenge einer unendlichen Menge U. Sei T das Komplement von S bzgl. U, dann ist T eine unendliche Menge.*

Beweis: Hier ist unsere Hypothese "'S endlich, U unendlich und T Komplement von S bzgl. U"' und unsere Folgerung ist "'T ist unendlich"'. Wir nehmen also an, dass T eine endliche Menge ist. Da T das Komplement von S ist, gilt $S \cap T = \emptyset$, also ist $\#(S) + \#(T) = \#(S \cap T) + \#(S \cup T) = \#(S \cup T) = n$, wobei n eine Zahl aus $\mathbb{N}$ ist (siehe Abschnitt C.1.6). Damit ist $S \cup T = U$ eine endliche Menge. Dies ist ein Widerspruch zu unserer Hypothese! Also war die Annahme "'T ist endlich"' falsch. #

E.4. Der Schubfachschluss

Der *Schubfachschluss* ist auch als *Dirichlets Taubenschlagprinzip* bekannt. Werden $n > k$ Tauben auf k Boxen verteilt, so gibt es mindestens eine Box in der sich wenigstens zwei Tauben aufhalten. Allgemeiner formuliert sagt das Taubenschlagprinzip, dass wenn n Objekte auf k Behälter aufgeteilt werden, dann gibt es mindestens eine Box die mindestens $\lceil \frac{n}{k} \rceil$ Objekte enthält.

Example 71: *Auf einer Party unterhalten sich* 8 *Personen (*$\triangleq$ *Objekte), dann gibt es mindestens einen Wochentag (*$\triangleq$ *Box) an dem* $\lceil \frac{8}{7} \rceil = 2$ *Personen aus dieser Gruppe Geburtstag haben.*

E.5. Gegenbeispiele

Im wirklichen Leben wissen wir nicht, ob eine Aussage richtig oder falsch ist. Oft sind wir dann mit einer Aussage konfrontiert, die auf den ersten Blick richtig ist und sollen dazu ein Programm entwickeln. Wir müssen also entscheiden, ob diese Aussage wirklich richtig ist, denn sonst ist evtl. alle Arbeit umsonst und hat hohen Aufwand verursacht. In solchen Fällen kann man versuchen, ein einziges Beispiel dafür zu finden, dass die Aussage falsch ist, um so unnötige Arbeit zu sparen.

Wir zeigen, dass die folgenden Vermutungen falsch sind:

Conjecture 72: *Wenn* $p \in \mathbb{N}$ *eine Primzahl ist, dann ist* p *ungerade.*

Gegenbeispiel: Die natürliche Zahl 2 ist eine Primzahl und 2 ist gerade. #

Conjecture 73: *Es gibt keine Zahlen* $a, b \in \mathbb{N}$*, sodass* $a \bmod b = b \bmod a$*.*

Gegenbeispiel: Für $a = b = 2$ gilt $a \bmod b = b \bmod a = 0$. #

E.6. Induktionsbeweise und das Induktionsprinzip

Sei nun eine Menge von natürlichen Zahlen X mit den folgenden zwei Eigenschaften gegeben:

(IA) $0 \in X$
(IS) Ist eine beliebige natürliche Zahl n ein Element von X, so ist auch die Zahl $n + 1$ ein Element von X.

Man kann sich nun leicht überlegen, dass dann X alle natürlichen Zahlen enthält, d.h. es gilt $X = \mathbb{N}$. Mit Hilfe dieser Beobachtung konstruieren wir eine der nützlichsten Beweismethoden in der Informatik bzw. Mathematik: Das *Induktionsprinzip* bzw. die Methode des *Induktionsbeweises*. Die Idee funktioniert mit folgender Beobachtung:

> Angenommen man kann nachweisen, dass 0 die Eigenschaft E hat[21] (kurz: $E(0)$) und weiterhin, dass wenn n die Eigenschaft E hat, dann gilt auch $E(n + 1)$. Ist dies der Fall, so muss jede natürliche Zahl die Eigenschaft E haben.

Im Folgenden wollen wir nachweisen, dass für jedes $n \in \mathbb{N}$ eine bestimmte Eigenschaft E gilt. Wir schreiben also abkürzend $E(n)$ für die Aussage "‘n besitzt die Eigenschaft E"’, d.h. der Schreibweise $E(0)$ drücken wir also aus, dass die erste natürliche Zahl 0 die Eigenschaft E besitzt, dann erhalten wir die folgende Vorgehensweise:

[21] Mit E wird also ein Prädikat oder Aussagenform bezeichnet (siehe Abschnitt C.1.2)

Induktionsprinzip: Es gelten
(IA) $E(0)$
(IS) Für $n \geq 0$ gilt, wenn $E(n)$ korrekt ist, dann ist auch $E(n+1)$ richtig.

Sind diese beiden Aussagen erfüllt, so hat jede natürliche Zahl die Eigenschaft E. Dabei ist **IA** die Abkürzung für *Induktionsanfang* und **IS** ist die Kurzform von *Induktionsschritt*. Die Voraussetzung ($\triangleq$ Hypothese) $E(n)$ ist korrekt für n und wird im Induktionsschritt als *Induktionsvoraussetzung* benutzt (kurz: **IV**). Hat man also den Induktionsanfang und den Induktionsschritt gezeigt, dann ist es anschaulich, dass jede natürliche Zahl die Eigenschaft E haben muss.
Es gibt verschiedene Versionen von Induktionsbeweisen. Die bekannteste Version ist die vollständige Induktion, bei der Aussagen über natürliche Zahlen gezeigt werden.

E.6.1. Die vollständige Induktion

Wie in Piratenfilmen üblich, seien Kanonenkugeln in einer Pyramide mit quadratischer Grundfläche gestapelt. Wir stellen uns die Frage, wieviele Kugeln (in Abhängigkeit von der Höhe) in einer solchen Pyramide gestapelt sind.

Theorem 74: *Mit einer quadratische Pyramide aus Kanonenkugeln der Höhe $n \geq 1$ als Munition, können wir $\frac{n(n+1)(2n+1)}{6}$ Schüsse abgeben.*

Beweis: Einfacher formuliert: wir sollen zeigen, dass $\sum\limits_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.
(IA) Eine Pyramide der Höhe $n = 1$ enthält $\frac{1\cdot 2\cdot 3}{6} = 1$ Kugel, d.h. wir haben die Eigenschaft für $n = 1$ verifiziert.
(IV) Für $k \leq n$ gilt $\sum\limits_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$.
(IS) Wir müssen nun zeigen, dass $\sum\limits_{i=1}^{n+1} i^2 = \frac{(n+1)((n+1)+1)(2(n+1)+1)}{6}$ gilt und dabei muss die Induktionsvoraussetzung $\sum\limits_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$ benutzt werden. Es ergeben sich die folgenden Schritte:

$$
\begin{aligned}
\sum_{i=1}^{n+1} i^2 &= \sum_{i=1}^{n} i^2 + (n+1)^2 \\
&\overset{\textbf{(IV)}}{=} \frac{n(n+1)(2n+1)}{6} + (n^2+2n+1) \\
&= \frac{2n^3+3n^2+n}{6} + (n^2+2n+1) \\
&= \frac{2n^3+9n^2+13n+6}{6} \\
&= \frac{(n+1)(2n^2+7n+6)}{6} \quad (\star) \\
&= \frac{(n+1)(n+2)(2n+3)}{6} \quad (\star\star) \\
&= \frac{(n+1)((n+1)+1)(2(n+1)+1)}{6}
\end{aligned}
$$

Die Zeile $\star$ (bzw. $\star\star$) ergibt sich, indem man $2n^3 + 9n^2 + 13n + 6$ durch $n + 1$ teilt (bzw. $2n^2 + 7n + 6$ durch $n + 2$). #

Das Induktionsprinzip kann man auch variieren. Dazu geht man davon aus, dass die Eigenschaft E für alle Zahlen $k \leq n$ erfüllt ist (Induktionsvoraussetzung). Obwohl dies auf den ersten Blick wesentlich komplizierter erscheint, vereinfacht dieser Ansatz oft den Beweis:

Verallgemeinertes Induktionsprinzip: Wenn die zwei Aussagen (also Induktionsanfang und Induktionsschritt)
(IA) $E(0)$

(IS) Wenn für alle $0 \leq k \leq n$ die Eigenschaft $E(k)$ gilt, dann ist auch $E(n+1)$ richtig,

gelten, dann haben alle natürliche Zahlen die Eigenschaft E. Damit ist das verallgemeinerte Induktionsprinzip eine Verallgemeinerung des weiter oben vorgestellten Induktionsprinzips, wie das folgende Beispiel veranschaulicht:

Theorem 75: *Jede natürliche Zahl $n \geq 2$ läßt sich als Produkt von Primzahlen schreiben.*

Beweis: Das verallgemeinerte Induktionsprinzip wird wie folgt verwendet:
(IA) Offensichtlich ist 2 das Produkt von einer Primzahl.
(IV) Jede natürliche Zahl m mit $2 \leq m \leq n$ kann als Produkt von Primzahlen geschrieben werden.
(IS) Nun wird eine Fallunterscheidung durchgeführt:

i) Sei $n+1$ wieder eine Primzahl, dann ist nichts zu zeigen, da $n+1$ direkt ein Produkt von Primzahlen ist.

ii) Sei $n+1$ keine Primzahl, dann existieren mindestens zwei Zahlen p und q mit $2 \leq p, q < n+1$ und $p \cdot q = n+1$. Nach Induktionsvoraussetzung sind dann p und q wieder als Produkt von Primzahlen darstellbar. Etwa $p = p_1 \cdot p_2 \cdot \ldots \cdot p_s$ und $q = q_1 \cdot q_2 \cdot \ldots \cdot q_t$. Damit ist aber $n+1 = p \cdot q = p_1 \cdot p_2 \cdot \ldots \cdot p_s \cdot q_1 \cdot q_2 \cdot \ldots \cdot q_t$ ein Produkt von Primzahlen. #

Induktionsbeweise treten z.B. bei der Analyse von Programmen immer wieder auf und spielen deshalb für die Informatik eine ganz besonders wichtige Rolle.

E.6.2. Induktive Definitionen

Das Induktionsprinzip kann man aber auch dazu verwenden, (Daten-)Strukturen formal zu spezifizieren. Dies macht diese Technik für Anwendungen in der Informatik besonders interessant. Dazu werden in einem ersten Schritt ($\triangleq$ Induktionsanfang) die "'atomaren"' Objekte definiert und dann in einem zweiten Schritt die zusammengesetzten Objekte ($\triangleq$ Induktionsschritt). Diese Technik ist als *induktive Definition* bekannt.

Example 76: *Die Menge der* binären Bäume *ist wie folgt definiert:*

(IA) *Ein einzelner Knoten w ist ein* Baum *und w ist die* Wurzel *dieses Baums.*
(IS) *Seien $T_1, T_2, \ldots, T_n$ Bäume mit den Wurzeln $k_1, \ldots, k_n$ und w ein einzelner neuer Knoten. Verbinden wir den Knoten w mit allen Wurzeln $k_1, \ldots, k_n$, dann entsteht ein neuer Baum mit der Wurzel w. Nichts sonst ist ein Baum.*

Example 77: *Die Menge der* arithmetischen Ausdrücke *ist wie folgt definiert:*

(IA) *Jeder Buchstabe und jede Zahl ist ein arithmetischer Ausdruck.*
(IS) *Seien E und F Ausdrücke, so sind auch $E+F$, $E*F$ und $[E]$ Ausdrücke. Nichts sonst ist ein Ausdruck.*

D.h. x, $x+y$, $[2*x+z]$ sind arithmetische Ausdrücke, aber beispielsweise sind $x+$, yy, $][x+y$ sowie $x+*z$ keine arithmetischen Ausdrücke im Sinn dieser Definition.

Example 78: *Die Menge der* aussagenlogischen Formeln *ist wie folgt definiert:*

(IA) *Jede aussagenlogische Variable $x_1, x_2, x_3, \ldots$ ist eine aussagenlogische Formel.*

(IS) *Seien H_1 und H_2 aussagenlogische Formeln, so sind auch $(H_1 \wedge H_2)$, $(H_1 \vee H_2)$, $\neg H_1$, $(H_1 \leftrightarrow H_2)$, $(H_1 \to H_2)$ und $(H_1 \oplus H_2)$ aussagenlogische Formeln. Nichts sonst ist eine aussagenlogische Formel.*

Bei diesen Beispielen ahnt man schon, dass solche Techniken zur präzisen und eleganten Definition von Programmiersprachen und Dateiformaten gute Dienste leisten. Es zeigt sich, dass die im Compilerbau verwendeten Chomsky-Grammatiken eine andere Art von induktiven Definitionen darstellen. Darüber hinaus bieten induktive Definitionen noch weitere Vorteile, denn man kann oft relativ leicht Induktionsbeweise konstruieren, die Aussagen über induktiv definierte Objekte belegen / beweisen.

E.6.3. Die strukturelle Induktion

Theorem 79: *Die Anzahl der öffnenden Klammern eines arithmetischen Ausdrucks stimmt mit der Anzahl der schließenden Klammern überein.*

Es ist offensichtlich, dass diese Aussage richtig ist, denn in Ausdrücken wie $(x + y)/2$ oder $x + ((y/2) * z)$ muss ja zu jeder öffnenden Klammer eine schließende Klammer existieren. Der nächste Beweis verwendet diese Idee um die Aussage von Satz 79 mit Hilfe einer *strukturellen Induktion* zu zeigen.
Beweis: Wir bezeichnen die Anzahl der öffnenden Klammern eines Ausdrucks E mit $\#_[(E)$ und verwenden die analoge Notation $\#_](E)$ für die Anzahl der schließenden Klammern.
(IA) Die einfachsten Ausdrücke sind Buchstaben und Zahlen. Die Anzahl der öffnenden und schließenden Klammern ist in beiden Fällen gleich 0.
(IV) Sei E ein Ausdruck, dann gilt $\#_[(E) = \#_](E)$.
(IS) Für einen Ausdruck $E + F$ gilt $\#_[(E + F) = \#_[(E) + \#_[(F) \overset{\textbf{IV}}{=} \#_](E) + \#_](F) = \#_](E + F)$. Völlig analog zeigt man dies für $E * F$. Für den Ausdruck $[E]$ ergibt sich $\#_[([E]) = \#_[(E) + 1 \overset{\textbf{IV}}{=} \#_](E) + 1 = \#_]([E])$. In jedem Fall ist die Anzahl der öffnenden Klammern gleich der Anzahl der schließenden Klammern. #

Mit Hilfe von Satz 79 können wir nun leicht ein Programm entwickeln, das einen Plausibilitätscheck (z.B. direkt in einem Editor) durchführt und die Klammern zählt, bevor die Syntax von arithmetischen Ausdrücken überprüft wird. Definiert man eine vollständige Programmiersprache induktiv, dann werden ganz ähnliche Induktionsbeweise möglich, d.h. man kann die Techniken aus diesem Beispiel relativ leicht auf die Praxis der Informatik übertragen.
Man überlegt sich leicht, dass die natürlichen Zahlen auch induktiv definiert werden können (vgl. Peano-Axiome). Damit zeigt sich, dass die vollständige Induktion eigentlich nur ein Spezialfall der strukturellen Induktion ist.

Glossary

Calculus of Constructions "The calculus of constructive proofs is a natural deduction style." https://doi.org/10.1016/0890-5401(88)90005-3. 2

Hadoop open-source software project for relaiblae, scalable und distrubuted computing. https://hadoop.apache.org . 2

Isabelle A generic proof assitant https://isabelle.in.tum.de/ . 2

model checker Checking weather a model meets a given specification. 2

SAT-solver An algorithm to solve a Boolean satisfubility problem called SAT. 2

SMT-solver An algorithm solving the SMT-problem, which is determining if a fomula of first order Logic is staisfiable. 2

References

[Can95] G. Cantor. Beiträge zur Begründung der transfiniten Mengenlehre. *Mathematische Annalen*, 46(4):481–512, 1895.

[Coqa] found online at https://coq.inria.fr/ accessed at the 17th February 2020.

[COQb] Documentation of the coq integrated development environment. found online at https://coq.inria.fr/refman/practical-tools/coqide.html acessed at 17th July 2019.

[Cpp] found online at https://www.cplusplus.com/doc/tutorial/structures accessed at the 7th February 2019.

[CSB16] F. Cerqueira, F. Stutz, and B. Brandenburg. Prosa: A case for readable mechanized schedulability analysis. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems*, ECRTS 2016, page 273–284, New York, NY, USA, 2016. Association for Computing Machinery.

[Ext] Extraction of programs in ocaml and haskell. found online at https://coq.inria.fr/refman/addendum/extraction.html?highlight=coding accessed 11th February 2020.

[Gal] The gallina language specification. found online at https://coq.inria.fr/refman/language/gallina-specification-language.html accessed at the 27th March 2020.

[Gia] N. Giannarakis. Personal github profile. found online at https://github.com/nickgian/thesis/lstcoq.sty, accessed at the 19th September 2020.

[PdAC+19] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. Software foundations. In *Logical Foundations*, volume Volume 1 of *5*. 2019. found online at https://softwarefoundations.cis.upenn.edu/current/lf-current/index.html.

[PROa] Proof general - a generic Emacs interface for proof assistants. found online at = https://proofgeneral.github.io/ Accessed: February 14th 2020.

[PROb] Prosa - ECRTS'16 Artifact Evaluation. Accessed: 2020.

[Rei] S. Reith. Vorlesung security a1. lecture notes. found online at https://www.cs.hs-rm.de/~reith/lehre/oloff2017wahrscheinlichkeitsrechnungSecurity19/page-9/styled-32, accessed at the July 10th 2019.

[Ull17] C. Ullenboom. *Java ist auch eine Insel, Einführung, Ausbildung, Praxis*. Rheinwerk Computing, 2017.