Formal Foundations of Computer Science 1

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Wolfgang.Schreiner@risc.uni-linz.ac.at

October 4, 1999

Abstract

We introduce basic mathematical domains and techniques as a foundation for the formal treatment of various concepts in computer science and engineering. Mathematical domains like numbers, functions, and relations are constructed on the basis of the fundamental notions of set theory thus exhibiting their inner structure and properties. As for mathematical techniques, we present the language of logic for the precise description of mathematical objects and facts and give rules for correct reasoning and arguing. The constructions are partially implemented with the help of an interpreter that executes a simple subset of logic and set theory.

Contents

Chapter 1

Introduction

AGEOMETRETOS MEDEIS EISITO
Let no one without skill in geometry enter.
(Inscription over the portal of Plato's Academy
in the sense of "Study mathematics first")

We introduce in this report basic mathematical domains and techniques as a foundation for the formal treatment of many concepts in computer science and engineering. The relationship between computer science and mathematics is still a subject of debate. Nevertheless it is clear that the mathematical way of problem solving is at the core of every scientific and technical activity whose goal is to describe phenomena of the real world and to influence these, e.g. by a computer program that fulfills a particular purpose.

The key idea in this process is *abstraction* (see Figure 1.1): we construct a simplified image (model, theory, domain) of the real world that captures all those aspects that we consider of relevance for our problem but neglects those facets that seem without influence. While being not a duplicate of the real world, this domain has the advantage that it can be unambiguously described and that we can operate in it (ask questions, derive knowledge, define new concepts) without resorting to the objects of the real world. An *interpretation* of the solution developed in this domain yields real world effects that (provided that our abstraction has been faithful and that our operations have been correct) meet our expectations.

While the real world has all kinds of different objects and phenomena, their mathematical images are ultimately described in terms of a relatively small number of notions, such as *numbers, functions, and relations* which in turn can be all reduced to the fundamental concept of *sets*. The constructions of

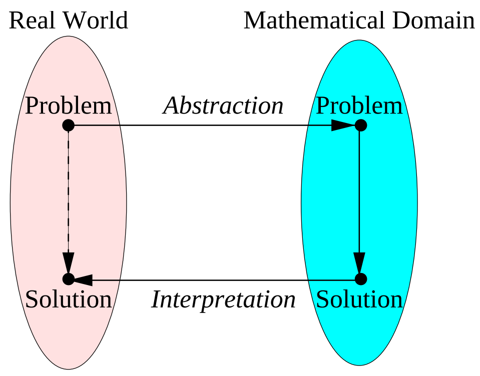


Figure 1.1: The Mathematical Process

these domains, their resulting inner structures, and their consequent properties reveal much of how mathematics works; we will elaborate them in this report in detail.

For understanding the constructions in a deeper sense it is important to learn not only mathematical contents but also mathematical techniques, in particular the *language* and the *reasoning rules* provided by mathematical logic. If these are mastered, we read and understand definitions and statements in any scientific area and judge whether a given argument is complete and correct. We therefore put in this report much emphasis on formally correct definitions and propositions in addition to the semi-formal ones that can be usually found in a textbook. Likewise we give proofs in a greater degree of formality and detail as is common practice; our goal is to enable the reader to check and to understand statements and claims, not to learn them by heart.

This report is therefore not only a collection of mathematical "cookbook recipes" that can be applied without any deeper insight (although it contains a lot of definitions and propositions); it is intended to make the reader understand "how things work" in order to enable her to operate in all formal domains that she will encounter in the future.

To emphasize this "no magic" approach, we use a small piece of software, the Logic Evaluator, a Java applet/application whose source can be found at

`http://www.risc.uni-linz.ac.at/software/formal`.

This interpreter allows to write function and predicate definitions in an executable subset of first order predicate logic with set theory and to evaluate terms and propositions. Many of the concepts introduced in this paper are implemented in this way; demonstration examples as the basis of own experiments can be found throughout the electronic version of this document (with screenshots in the hardcopy). The Java classes for formulas and terms are listed to illustrate the operational interpretation of logic formulas.

Much inspiration and some of the contents of this report have been derived from the textbook

> B. Buchberger, F. Lichtenberger
> Mathematik für Informatik I — Die Methode der Mathematik
> Zweite, korrigierte Auflage, Springer, Berlin, 1981

as well as from the lecture notes

> F. Lichtenberger
> Mathematik für Informatiker 2
> Johannes Kepler Universität, Sommersemester 1999

based on material from former courses of Bruno Buchberger. A subset of the Theorema software

> `http://www.theorema.org`

being developed by Bruno Buchberger and his working group has been a source of inspiration for the Logic Evaluator. Many thanks to Werner Danielczyk-Landerl, Peter Kulczycki, Michael Petz, and Wolfgang Windsteiger for discussions on and/or corrections of this report.

This document contains two parts, the main body oriented towards mathematical domains (Chapters 2–7) and an appendix oriented towards mathematical techniques (Appendices A and B). We recommend to study the material in the order

$$2 \to \mathrm{A} \to 3 \to 4 \to 5 \to \mathrm{B} \to 6 \to 7$$

skipping the details of the proofs in Chapters 3 and 4 on first reading and studying them again after having read Appendix B.

Chapter 2

The Language of Logic

There are two main aspects of mathematical logic that are important for computer science[1].

1. Logic provides a precise *language* for formulating statements (Aussagen) and denoting objects (Gegenstände). Since computer science deals with highly abstract entities with complicated properties, it is important to have such a precise language in which critical issues can be expressed without ambiguity. The role of natural language in scientific work is to communicate basic ideas and intuitions; the role of logic is to resolve ambiguities and to clarify details.

2. Logic provides an intellectual framework for correct reasoning (Schlußfolgern, Schließen) and arguing (Argumentieren). These activities can be ultimately reduced to a finite number of rules that are formulated so precisely that their correct application can be checked automatically. Computer programs are probably the most complex human artifacts; to reason about their properties is a central task of computer science.

A precise language and sound reasoning belong to the most valuable intellectual tools of a computer scientist or engineer. Only if she masters these tools thoroughly, she can utilize her creative potential in an effective and goal-directed way. Please note that it is *not* the heavy use of symbolism that characterizes precise expression and clear arguments. A computer scientist should be able to recognize and to present the underlying logical concepts

[1]In this chapter, we focus on the *language* aspect of logic; the *reasoning* aspect is investigated in Appendix B. A third aspect, the automatization of problem solving on computers, is discussed in another course "Logic for Computer Science".

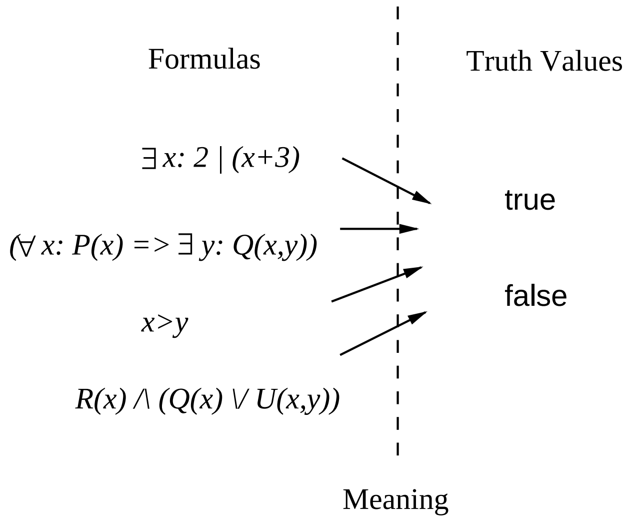


Figure 2.1: Formulas and Truth Values

on various language levels and in various notations. She can then flexibly adapt form and preciseness of her statements to the demands of a particular situation while preserving the essential contents. We therefore strive to demonstrate these concepts in various styles, from natural language ("every Ferrari is red") to purely symbolic ($\forall x : F(x) \Rightarrow R(x)$).

2.1 Preliminaries

The fundamental entities of logic are introduced in the following definition (see Figure 2.1).

Definition 1 (Truth Value, Formula) A *truth value (Wahrheitswert)* or *Boolean value (Boolescher Wert)* is one of the values true or false.

A *(logical) formula (logische Formel)* or *proposition (Aussage)* is a syntactic phrase whose meaning is a truth value.

This definition introduces two layers of a formula: its *syntax (Syntax)*, i.e., the concrete external representation, and its *semantics (Semantik)*, i.e., the underlying meaning.

Operational Interpretation In the Logic Evaluator, every formula is an instance of a Java class that implements the following interface:

```
public interface Formula
{
  boolean eval() throws EvalException;
}
```

For every `Formula` f, the evaluation of f`.eval()` returns the boolean value denoted by f (or throws an exception if some error has occurred).

For stating relationships between formulas, we will use the following notion.

Definition 2 (Equivalence of Formulas) Let A and B be formulas. We write

$$A \text{ iff } B$$

to indicate that A has the same meaning as B in any context; we read this as "A is *equivalent* to B" ("A ist äquivalent zu B") or as "A if and only if B" ("A genau dann wenn B").

2.2 Propositional Logic

Propositional logic is that part of mathematical logic that deals with the composition of formulas. The composition starts with basic formulas that are considered as "black boxes": we are interested in their truth value ("true or false?") but not in what properties they actually describe. Various operations then allow to combine simpler formulas to more complex ones.

Definition 3 (Logical Connective) A *(logical) connective (Junktor)* is a syntactic operator that combines formulas to a new formula.

In more detail, the formulas of propositional logic are constructed from the connectives 'F', 'T', '$\neg$', '$\wedge$', '$\vee$', '$\Rightarrow$', '$\Leftrightarrow$' as follows:

Proposition 1 (Formulas of Propositional Logic)

- The following *logical constants (Logische Konstanten)* are formulas[2]:

  False "false" ("falsch")

  $$\mathrm{F}$$

  True "true" ("wahr")

  $$\mathrm{T}$$

- If A is a formula, then the following is a formula:

  Negation "not A" ("nicht A")

  $$\neg A$$

- If A and B are formulas, then the following are formulas

  Conjunction "A and B" ("A und B")

  $$(A \wedge B)$$

  Disjunction "A or B" ("A oder B")

  $$(A \vee B)$$

  Implication "A implies B" ("A impliziert B")

  $$(A \Rightarrow B)$$

  Equivalence "A is equivalent to B" ("A ist äquivalent zu B")

  $$(A \Leftrightarrow B)$$

  We omit the parentheses if A and B are unambiguous (usually x is free in A).

Above propositions provide the basis for a hierarchical construction of formulas.

Example The formula

$$(\mathrm{T} \wedge \mathrm{F}) \Rightarrow \neg(\mathrm{F} \vee \neg\mathrm{F})$$

has the following syntactic structure:

[2]Logical constants can be considered as connectives of arity 0, i.e., as connectives that do not take any arguments.

$$\overbrace{(\overbrace{\overbrace{\mathrm{T}}^{\text{Constant}} \wedge \overbrace{\mathrm{F}}^{\text{Constant}}}^{\text{Conjunction}}) \Rightarrow \overbrace{\neg(\overbrace{\overbrace{\mathrm{F}}^{\text{Constant}} \vee \overbrace{\neg \overbrace{\mathrm{F}}^{\text{Constant}}}^{\text{Negation}}}^{\text{Disjunction}})}^{\text{Negation}}}^{\text{Implication}}$$

The syntactic structure of the formula

$$\mathrm{T} \wedge \mathrm{F} \Rightarrow \mathrm{F}$$

is ambiguous; it may be understood as $(\mathrm{T} \wedge \mathrm{F}) \Rightarrow \mathrm{F}$, i.e.,

$$\overbrace{\overbrace{\overbrace{\mathrm{T}}^{\text{Constant}} \wedge \overbrace{\mathrm{F}}^{\text{Constant}}}^{\text{Conjunction}} \Rightarrow \overbrace{\mathrm{F}}^{\text{Constant}}}^{\text{Implication}}$$

or as $\mathrm{T} \wedge (\mathrm{F} \Rightarrow \mathrm{F})$, i.e,

$$\overbrace{\overbrace{\mathrm{T}}^{\text{Constant}} \wedge \overbrace{\overbrace{\mathrm{F}}^{\text{Constant}} \Rightarrow \overbrace{\mathrm{F}}^{\text{Constant}}}^{\text{Implication}}}^{\text{Conjunction}}.$$

In the following sections, we will discuss the meanings of these formulas.

2.2.1 Logical Constants

The following logical constants are formulas:

F,
T.

Syntactic Forms Logical constants may appear in various forms, e.g. as

- 0, 1;
- "wrong", "right";
- "incorrect", "correct";

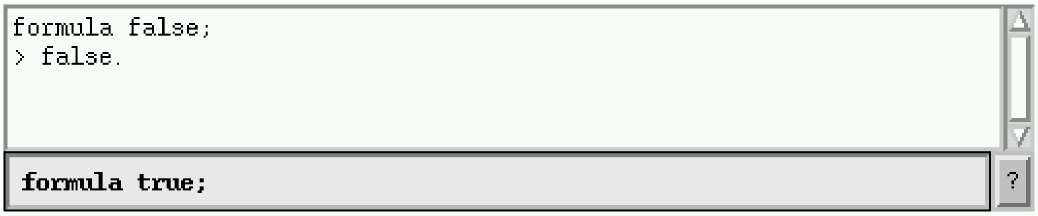
- false, true.

The last form is most frequently used; please do not confuse these *formulas* with the *truth values* true and false.

Definition 4 (Semantics of Logical Constants) The *meaning* of 'F' is false and the *meaning* of 'T' is true.

Again, please note the distinction between the two layers of syntax ('F', 'T') and semantics (true, false).

The Logic Evaluator uses the syntax `false` and `true` both for (the input of) logical constants and for (the output of) truth values. However, by above explanation it should be clear that the input denotes a *formula* while the output prefixed by the token '`>`' denotes the *meaning* of this formula.



2.2.2 Negations

If A is a formula, then

$$\neg A$$

is a formula.

Alternative Forms Other syntactic forms of negation are

- $\overline{A}$, $\sim A$, $-A$, $!A$;
- non-A;
- "not A", "never A", "in no case A";
- `not(`A`)`.

The last line denotes the input syntax of the Logic Evaluator.

Negation of infix atomic formulas is often expressed by crossing the predicate symbol, e.g.

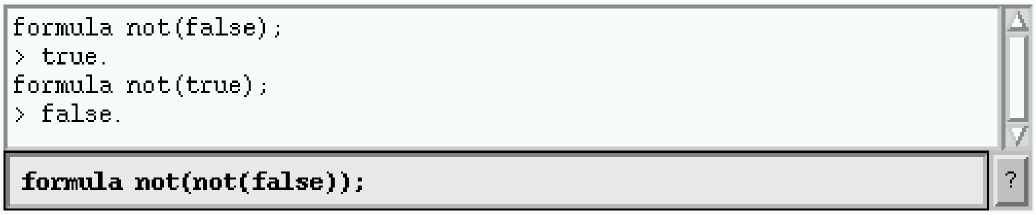
$$x \not< y$$

represents $\neg(x < y)$.

Definition 5 (Semantics of Negation) For every formula A, the meaning of $\neg A$ is denoted by the following *truth table (Wahrheitstabelle)* that lists the possible truth values for A in the first column and the truth value of its negation in the second column:

A	$\neg A$
false	true
true	false

In other words, $\neg A$ is true if and only if A is false.



Operational Interpretation A logical formula can be given an operational interpretation by a program that computes its truth value. For instance, in the Logic Evaluator, a negation is represented by an object of Java type

```
public final class Not implements Formula
{
  private Formula formula;

  public Not(Formula _formula)
  {
    formula = _formula;
  }
```

```
  public boolean eval() throws EvalException
  {
    if (formula.eval())
      return false;
    else
      return true;
  }
}
```

The Java expression `(new Not(`A`)).eval()` thus computes the truth value of the negation of A. The result is true only if the truth value of A is false.

We conclude the discussion of negation by stating a simple law and proving its correctness.

Proposition 2 (Inversion of Negation) For every formula A, we have

$\neg\neg A$ iff A.

Proof Let A be an arbitrary formula. We have to show that the meaning of $\neg\neg A$ is the same as the meaning of A, i.e., that they have the same truth values. From Definition 5, we can construct the following truth table.

A	$\neg A$	$\neg\neg A$
false	true	false
true	false	true

Since the last column coincides with the first column, we are done.

2.2.3 Conjunctions

If A and B are formulas, then

$(A \wedge B)$

is a formula.

Alternative Forms A conjunction of A and B may also appear in other syntactic forms, e.g. as

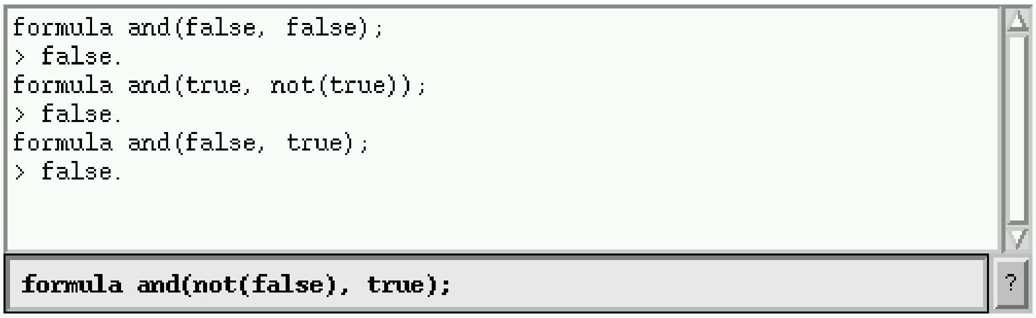
- A, B (comma);
- $A * B$, A & B, A `&&` B;
- "A and B", "A as well as B" ("sowohl A als auch B");
- `and(`A`,` B`)`.

The last form is the input syntax for the Logic Evaluator.

Definition 6 (Semantics of Conjunction) Let A and B be formulas. The *meaning of* $A \wedge B$ is defined by the following truth table:

A	B	$A \wedge B$
false	false	false
false	true	false
true	false	false
true	true	true

In other words, $A \wedge B$ is true if and only if both A and B are true.



Operational Interpretation In the Logic Evaluator, a conjunction is represented by an object of the Java type

```
public final class And implements Formula
{
  private Formula formula0;
  private Formula formula1;

  public And(Formula _formula0, Formula _formula1)
```

```
    {
      formula0 = _formula0;
      formula1 = _formula1;
    }

    public boolean eval() throws EvalException
    {
      if (formula0.eval())
      {
        if (formula1.eval())
          return true;
        else
          return false;
      }
      else
        return false;
    }
  }
```

The Java expression `(new And(`A`, `B`)).eval()` computes the truth value of $A \wedge B$. As one can see, if A evaluates to false, the result is immediately false, i.e., the truth value of B does not matter any more. Only if A evaluates to true, also B is evaluated; the result is true only if both formulas are true.

From Definition 6, we can deduce the following properties of conjunctions.

Proposition 4 (Conjunctive Laws) Conjunction is *commutative*, i.e., for all formulas A and B, we have

$$A \wedge B \text{ iff } B \wedge A.$$

Conjunction is also *associative*, i.e., for all formulas A, B, and C, we have

$$A \wedge (B \wedge C) \text{ iff } (A \wedge B) \wedge C.$$

We will now argue for the correctness of the second part of this proposition, i.e., the associativity of conjunction.

Proof Let A, B, and C be arbitrary formulas. We have to show that, for every possible truth value of A, B, and C, the formulas $A \wedge (B \wedge C)$ and $(A \wedge B) \wedge C$ have the same truth value.

From the truth table of $A \wedge B$ we can determine the meaning of $A \wedge (B \wedge C)$:

A	B	C	$B \wedge C$	$A \wedge (B \wedge C)$
false	false	false	false	false
false	false	true	false	false
false	true	false	false	false
false	true	true	true	false
true	false	false	false	false
true	false	true	false	false
true	true	false	false	false
true	true	true	true	true

Likewise, we can determine the meaning of $(A \wedge B) \wedge C$:

A	B	C	$A \wedge B$	$(A \wedge B) \wedge C$
false	false	false	false	false
false	false	true	false	false
false	true	false	false	false
false	true	true	false	false
true	false	false	false	false
true	false	true	false	false
true	true	false	true	false
true	true	true	true	true

Since the final results are identical in all lines of both tables, the meanings of both formulas are the same.

Convention Because of associativity, it does not matter in which particular way parentheses are placed in nestings of conjunctive formulas. We will therefore write $A \wedge B \wedge C$ instead of $A \wedge (B \wedge C)$ respectively $(A \wedge B) \wedge C$ and, in general,

$$A_0 \wedge A_1 \wedge \ldots \wedge A_{n-1}$$

for conjunctions of n formulas (for every n). Also the Logic Evaluator allows conjunctions

`and(`A_0`,` A_1`, ...,` A_{n-1}`)`

of an arbitrary number of formulas.

2.2.4 Disjunctions

If A and B are formulas, then

$$A \vee B$$

is a formula.

Alternative Forms A disjunction of A and B may also appear in other syntactic forms, e.g. as

- $A + B$, $A \mid B$, $A \mid\mid B$;
- "A or B";
- `or(`A`, `B`)`.

The last form is the input syntax for the Logic Evaluator.

Definition 7 (Semantics of Disjunction) Let A and B be formulas. The *meaning of* $A \vee B$ is defined by the following truth table:

A	B	$A \vee B$
false	false	false
false	true	true
true	false	true
true	true	true

In other words, $A \vee B$ is false if and only if both A and B are false.

Since $A \vee B$ is also true if both A and B are true, it must *not* be read as "either A or B" (which would indicate otherwise). Such an "exclusive or" connective needs an extra definition (which is given at the end of this section).

```
formula or(true, false);
> true.
formula or(false, not(false));
> true.
formula or(true, true);
> true.

formula or(not(true), and(true, false));
```

Operational Interpretation In the Logic Evaluator, a disjunction is represented by an object of the Java type

```
public final class Or implements Formula
{
  private Formula formula0;
  private Formula formula1;

  public Or(Formula _formula0, Formula _formula1)
  {
    formula0 = _formula0;
    formula1 = _formula1;
  }

  public boolean eval() throws EvalException
  {
    if (formula0.eval())
      return true;
    else
      if (formula1.eval())
        return true;
      else
        return false;
  }
}
```

The Java expression `(new Or(`A`, `B`)).eval()` computes the truth value of $A \vee B$. As one can see, if A evaluates to true, the result is immediately true, i.e., the truth value of B does not matter any more. Only if A evaluates to false, also B is evaluated; the result is false only if both formulas are false.

From Definition 7, we can deduce the following properties of disjunctions.

Proposition 6 (Disjunctive Laws) Disjunction is *commutative*, i.e., for all formulas A and B, we have

$A \vee B$ iff $B \vee A$.

Disjunction is also *associative*, i.e., for all formulas A, B, and C, we have

$A \vee (B \vee C)$ iff $(A \vee B) \vee C$.

We have an important *duality* between conjunctions and disjunctions expressed by the following law.

Proposition 7 (De Morgan's Laws) For all formulas A and B, the following holds:

$$\begin{array}{lll} \neg(A \wedge B) & \text{iff} & \neg A \vee \neg B, \\ \neg(A \vee B) & \text{iff} & \neg A \wedge \neg B. \end{array}$$

Consequence From above laws, we have

$$A \vee B \text{ iff } \neg(\neg A \wedge \neg B).$$

The disjunction $A \vee B$ is therefore frequently defined just as a *syntactic abbreviation* of $\neg(\neg A \wedge \neg B)$.

Convention Because of associativity, it does not matter in which particular way parentheses are placed in nestings of disjunctive formulas. We will therefore write $A \vee B \vee C$ instead of $A \vee (B \vee C)$ respectively $(A \vee B) \vee C$ and, in general,

$$A_0 \vee A_1 \vee \ldots \vee A_{n-1}$$

for conjunctions of n formulas (for every n). Also the Logic Evaluator allows disjunctions

`or(`A_0`, `A_1`, ..., `A_{n-1}`)`

of an arbitrary number of formulas.

Exclusive Disjunction Finally we give the definition of the "exclusive or" connective mentioned above.

Definition 8 (Exclusive Disjunction) If A and B are formulas, then also

$$(A \text{ xor } B)$$

is a formula, the *exclusive disjunction (Ausschließende Oder-Verknüpfung)* of A and B, The meaning of this formula is defined by the following truth table:

A	B	A xor B
false	false	false
false	true	true
true	false	true
true	true	false

In other words, A xor B is true, if and only if exactly one of A or B is true.

We then have the following relationship between both kinds of disjunctions.

Proposition 8 (Inclusive and Exclusive Disjunction) For all formulas A and B, we have

$$(A \text{ xor } B) \text{ iff } (A \wedge \neg B) \vee (\neg A \wedge B).$$

Because of this law, the formula $(A \text{ xor } B)$ is frequently just defined as a *syntactic abbreviation* for $(A \wedge \neg B) \vee (\neg A \wedge B)$.

2.2.5 Implications

If A and B are formulas, then

$$(A \Rightarrow B)$$

is a formula.

Alternative Forms An implication $A \Rightarrow B$ is often also expressed as

- "A implies B" ("A impliziert B");
- "if A, then B" ("wenn A, dann B");
- "B, (only) if A" ("B (nur) dann, wenn A");
- "B follows from A" ("aus A folgt B");
- "A is sufficient for B" ("A ist hinreichend für B");
- "B is necessary for A" ("B ist notwendig für A");

- `implies(`A`, `B`)`.

The last line denotes the input syntax of the Logic Evaluator.

Definition 9 (Semantics of Implication) Let A and B be formulas. The *meaning of* $A \Rightarrow B$ is defined by the following truth table:

A	B	$A \Rightarrow B$
false	false	true
false	true	true
true	false	false
true	true	true

In other words, $A \Rightarrow B$ is false if and only if A is true and B is false.

Please note that an implication is always true if its premise is false. This may appear strange at first glance, because it makes sentences like "if 2 is odd, then 3 is even" true. However, by an implication we actually want to express the fact "*if* the premise holds, then also the conclusion holds", i.e.,

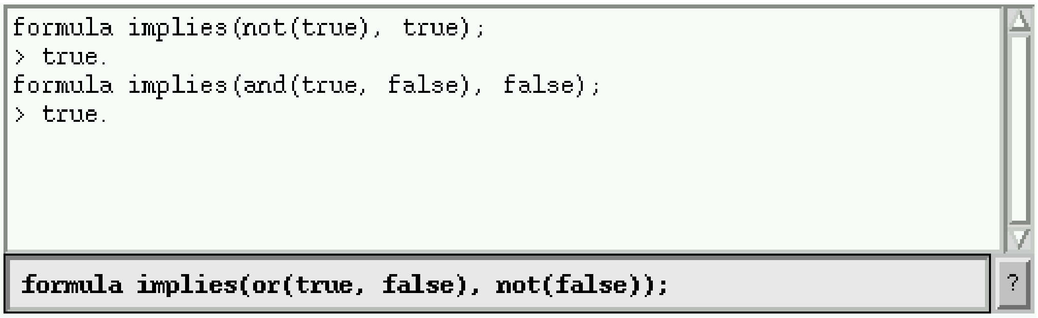
- either the premise does not hold,
- or (the premise holds and) the conclusion holds.

This behavior is also expressed by the first of the following laws.

Proposition 9 (Implicative Laws) For all formulas A and B, we have

$$\begin{array}{rll} A \Rightarrow B & \text{iff} & \neg A \vee B \\ A \Rightarrow B & \text{iff} & \neg B \Rightarrow \neg A \\ \neg(A \Rightarrow B) & \text{iff} & A \wedge \neg B. \end{array}$$

Because of the first law, the implication $A \Rightarrow B$ is frequently defined just as a *syntactic abbreviation* for $\neg A \vee B$.



Operational Interpretation In the Logic Evaluator, an implication is represented by an object of the Java type

```
public final class Implies implements Formula
{
  private Formula formula0;
  private Formula formula1;

  public Implies(Formula _formula0, Formula _formula1)
  {
    formula0 = _formula0;
    formula1 = _formula1;
  }

  public boolean eval() throws EvalException
  {
    if (formula0.eval())
    {
      if (formula1.eval())
        return true;
      else
        return false;
    }
    else
      return true;
  }
}
```

The Java expression `(new Implies(`A`, `B`)).eval()` computes the truth value of $A \Rightarrow B$. As one can see, if A evaluates to false, the result is immediately true, i.e., the truth value of B does not matter any more. Only if A evaluates to true, also B is evaluated; the result is false only if A is true and B is false.

2.2.6 Equivalences

If A and B are formulas, then

$$(A \Leftrightarrow B)$$

is a formula.

Alternative Forms An equivalence $A \Leftrightarrow B$ is often also expressed as

- $A = B$, $A \sim B$;
- "A and B are equivalent" ("A und B sind äquivalent");
- "A if and only if B" ("A genau dann, wenn B", "A dann und nur dann, wenn B");
- "A iff B" ("A gdw B");
- "A is necessary and sufficient for B" ("A ist notwendig und hinreichend für B");
- `equiv(`A`, `B`)`.

The last line denotes the input syntax of the Logic Evaluator.

Definition 10 (Semantics of Equivalence) Let A and B be formulas. The *meaning of* $A \Leftrightarrow B$ is defined by the following truth table:

A	B	$A \Leftrightarrow B$
false	false	true
false	true	false
true	false	false
true	true	true

In other words, $A \Leftrightarrow B$ is true if and only if A and B have the same truth value.

```
formula equiv(not(true), true);
> false.
formula equiv(implies(not(true), false), true);
> true.
```

```
formula implies(equiv(not(true), false), true);
```

Operational Interpretation In the Logic Evaluator, an implication is represented by an object of the Java type

```
public final class Equiv implements Formula
{
  private Formula formula0;
  private Formula formula1;

  public Equiv(Formula _formula0, Formula _formula1)
  {
    formula0 = _formula0;
    formula1 = _formula1;
  }

  public boolean eval() throws EvalException
  {
    return (formula0.eval() == formula1.eval());
  }
}
```

The Java expression `(new Equiv(`A`, `B`)).eval()` computes the truth value of $A \Leftrightarrow B$. As one can see, the result is `true` only if A and B have the same truth value.

Proposition 10 (Equivalence Laws) For all formulas A and B, the following holds:

$$
\begin{array}{lll}
A \Leftrightarrow B & \text{iff} & B \Leftrightarrow A \\
A \Leftrightarrow B & \text{iff} & (A \wedge B) \vee (\neg A \wedge \neg B) \\
A \Leftrightarrow B & \text{iff} & (A \Rightarrow B) \wedge (B \Rightarrow A)
\end{array}
$$

Because of the last relationship, the equivalence $A \Leftrightarrow B$ is frequently defined just as a *syntactic abbreviation* for $(A \Rightarrow B) \wedge (B \Rightarrow A)$.

Proposition 11 (Equivalence) For all formulas A and B, the formula $A \Leftrightarrow B$ is true if and only if

$$A \text{ iff } B$$

holds.

This law introduces a relationship between '$\Rightarrow$' and the notion of equivalence that justifies to call this operator by the same name.

2.2.7 Summary

Let A and B be formulas. We summarize the meanings of all connectives presented so far in a single truth table:

A	B	F	T	$\neg A$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
false	false	false	true	true	false	false	true	true
false	true	false	true	true	false	true	true	false
true	false	false	true	false	false	true	false	false
true	true	false	true	false	true	true	true	true

Furthermore, we have the following relationships between connectives:

$$\begin{array}{rll} A \vee B & \text{iff} & \neg(\neg A \wedge \neg B) \\ A \Rightarrow B & \text{iff} & \neg A \vee B \\ A \Leftrightarrow B & \text{iff} & (A \Rightarrow B) \wedge (B \Rightarrow A) \end{array}$$

All connectives can therefore be ultimately reduced to negation and conjunction.

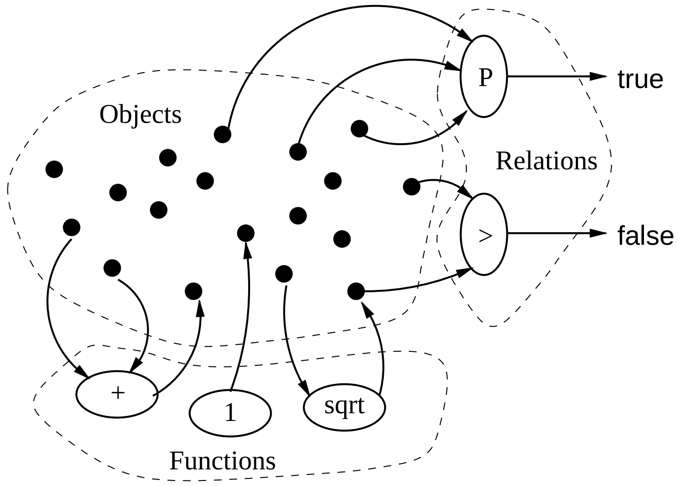


Figure 2.2: A Domain

2.3 Predicate Logic

Propositional logic is just concerned about the composition of formulas; it does not investigate the properties described by these formulas. We now enter the realm of *predicate logic* that is able to describe objects and properties of a domain, i.e., of some abstract model of the real world (see Figure 2.2).

Definition 11 (Domain) A *domain (Bereich)* or *structure (Struktur)* consists of

1. A collection of *values (Werte)* or *objects (Gegenstände)*; the number of objects may be finite or infinite (but not zero).

2. A collection of *functions (Funktionen)*, also called *mappings (Abbildungen, Zuordnungen)*; each function takes a certain number of values, the *arguments (Argumente)*, and returns a value, the *result (Resultat, Ergebnis)*. The number of arguments is the *arity* of the function.

   A function with arity 0 is also called a *constant (Konstante)*.

3. A collection of *predicates (Prädikate)*, also called *relations (Relationen, Beziehungen)*, *properties (Eigenschaften)*, *attributes (Attribute)*; each predicate takes a certain number of values, the predicate *arguments*, and returns a truth value.

If the predicate returns `true`, it is said to be true, to *hold (halten)*, or to be *valid (gültig)* for these values.

Example Various examples of domains are

- the set of natural numbers as values with constants "zero" and "one" and functions "addition", "multiplication", and predicate "is less than';
- the people in Austria as "values" with functions "mother of" and "father of" and predicates, "is male", "is female", "are parents of";
- the set of Java values of type `int[]` and `int` with function "indexed array access" and predicate "is sorted".

Predicate logic can be considered an extension of propositional logic, i.e., all propositional logic formulas are also predicate logic formulas and all properties derived for propositional logic also hold (in a generalized form) in predicate logic. While in propositional logic the only elementary formulas are the logical constants 'T' and 'F', predicate logic allows elementary formulas that express properties of objects. For this purpose, predicate logic introduces means to denote objects in a given domain.

2.3.1 Terms

Definition 12 (Term) A *term (Term, Begriff)* is a syntactic phrase whose meaning is a value (of the domain being considered).

The meaning of a term therefore depends on the domain; changing the domain also changes the meaning of a term.

Operational Interpretation In the Logic Evaluator, every term is an instance of a Java class that implements the following interface:

```
public interface Term
{
  Value eval() throws EvalException;
}
```

For every `Term` t, the evaluation of `t.eval()` returns the value denoted by t (or throws an exception if some error has occurred).

Definition 13 (Variable, Assignment) A *variable (Variable)* is a name that may represent any value of the domain.

A *variable assignment (Zuweisung, Belegung)* maps every variable to a value of the domain.

The value of a variable therefore depends on a given assignment, e.g. the assignment $[x \mapsto \text{“}one'', y \mapsto \text{“}two'']$ on the domain of natural numbers maps the variable y to the value “two”.

Please note that variables may be only mapped to values, i.e., to *first order objects (Objekte erster Ordnung)*, not to relations or functions, i.e., to *higher order objects (Objekte höherer Ordnung)*. The version of predicate logic that imposes this restriction is called *first order predicate logic (Prädikatenlogik erster Stufe)*. As we will see in Chapter 3, we can encode functions and relations as first order objects, such that this restriction does not limit the expressiveness of the language from the practical point of view.

Definition 14 (Function Constant) A *function constant (Funktionskonstante)* is a name that may represent some function. The *arity* of the function constant determines the arity of the functions it may stand for.

We call a function constant of arity 0 an *object constant (Objektkonstante)*.

We use the notion *name* for any combination of characters (“x_2”) or symbols (“$\sqrt{\ }$”). Since both variables and function constants are names, we apply in this document the syntactic convention that only names starting with the letters x, y, z denote variables (unless explicitly stated otherwise).

Example The following are function constants:

- 0, 1, 3.14, “Wolfgang Schreiner”, “Austria”, π (object constants);
- $|\ |$, sin(), $\sqrt{\ }$, “mother of” (unary constants);
- $+$, $\circ$, $[\ ]$ (binary constants);

Terms are then constructed as follows.

Proposition 12 (Syntax of Terms)

- Every variable is a term.
- If fc is a function constant of arity n and $t_0, \ldots, t_{n-1}$ are terms, then

  $$fc(t_0, \ldots, t_{n-1})$$

  is a term, called an *elementary term (elementarer Term)* or *function application (Funktionsanwendung)*.

Terms do not necessarily appear in the "standard form" denoted above but they may appear in various syntactic formats.

Example The following are terms:

- 0, "Wolfgang Schreiner", π (constant terms);
- $\sqrt{2}$, $\sin(x)$, $\text{sum}(2, s)$, "the mother of Thomas" (prefix terms);
- $|1|$, $a \circ b$, $a[i]$, $1 + 2$ (infix terms);

A natural language term

the roof of the house of her father

can be written in prefix form as

roof(house(father(she)))

with function constants "roof", "house", "father" and a variable "she".

Definition 15 (Semantics of Terms) The *meaning* of a term t under an assignment in some domain is defined as follows:

- If t is a variable, then its meaning is the value to which the variable is mapped by the assignment.
- If t is an elementary term $fc(t_0, \dots, t_{n-1})$, then its meaning is the result of the application of the function f denoted by the function constant fc to the values of the terms t_i for the given assignment.

The value of a term therefore depends on the domain (determining the *interpretation* of the constants as functions of the domain) as well as on the variable assignment (determining the values of the variables).

Example Take the term

$$x + (y + 0).$$

In the domain "natural numbers" with object constant 0 interpreted as the number "zero", the function constant + interpreted as addition and under the variable assignment $[x \mapsto 1, y \mapsto 2]$, the value of this term is the number "three". However, under the variable assignment $[x \mapsto 1, y \mapsto 0]$, its meaning is "one".

In the domain "character strings" with object constant 0 interpreted as the empty string, the function constant + interpreted as string concatenation and under the variable assignment $[x \mapsto \text{“}hi,\ '', y \mapsto \text{“}babe'']$, the value of the term is "hi, babe".

The Logic Evaluator has the domain "natural numbers" builtin with object constants `0`, `1`, ... and function constants `+` and `*`. Since terms must be entered always in prefix notation, terms can be evaluated as follows:

```
term 0;
> 0.
term +(1, 0);
> 1.
term +(2, *(3, 4));
> 14.
```

```
term +(*(2, 3), *(4, 5));
```

Operational Interpretation Elementary terms are implemented by the following Java class:

```
public final class Application implements Term
{
  private String name;
  private Term[] arguments;

  public Application(String _name, Term[] _arguments)
  {
    name = _name;
    arguments = _arguments;
  }

  public Value eval() throws EvalException
  {
    Function function = Model.getFunction(name, arguments.length);
    if (function == null)
      throw new EvalException("unknown function " + name +
                              "/" + arguments.length);
    Value[] values = new Value[arguments.length];
    for (int i=0; i< values.length; i++)
      values[i] = arguments[i].eval();
    return function.apply(values);
  }
}
```

The Java term `new Application(`f`,` *args*`).eval()` evaluates to the meaning of the term $f(args)$ (where f is a function constant and *args* is the list of arguments). As one can see, first we determine the interpretation *function* of the function constant in the given domain (`Model`), then we evaluate the arguments and apply the function to the results.

The operational interpretation of variables will be explained in Section 2.3.4.

2.3.2 Atomic Formulas

Constants may not only denote functions but also predicates.

Definition 16 (Predicate Constant) A *predicate constant (Prädikatenkonstante)* is a name that may represent some predicate. The *arity* of the predicate constant determines the arity of the predicates it may stand for.

Predicate constants may appear in various syntactic formats.

Example The following are predicate constants:

- "is positive" (unary predicate constant);
- $\leq$, $|$ (binary predicate constants);
- "is father of" (binary predicate constant);
- "is a child of ... and of ..." (ternary predicate constant).

We now introduce formulas that allow us to express properties about objects.

Proposition 13 (Atomic Formula) If pc is a predicate constant of arity n and $t_0, \ldots, t_{n-1}$ are terms, then

$$pc(t_0, \ldots, t_{n-1})$$

is a formula, called *atomic formula (atomare Aussage)*.

Atomic formulas may appear in various syntactic formats.

Example The following are atomic formulas:

- "1 is positive";
- $2 \leq \sqrt{x+3}$;
- $2|5$;
- "Thomas is the father of Susanne";
- "Susanne is a child of Thomas and of Birgit".

A natural language formula

Bill Clinton is a better president than his predecessor.

can be written in prefix form as

isBetterPresident(Bill Clinton, Predecessor(Bill Clinton))

with predicate constant "isBetterPresident" and function constants "Predecessor" and "Bill Clinton".

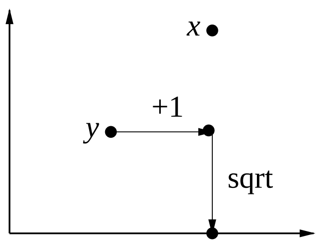
Definition 17 (Semantics of Atomic Formulas) The *meaning* of an atomic formula $pc(t_0, \ldots, t_{n-1})$ under an assignment in some domain is the truth value of the predicate denoted by the predicate constant pc for the values of the terms t_i under the given assignment.

The truth value of an atomic formula thus depends on the domain (determining the interpretation of the predicate constant and the interpretation of the function constants in the argument terms) as well as on the variable assignment (determining the values of the variables).

Example Take the formula

$$x \leq \sqrt{y+1}$$

in the domain of real numbers with '$\leq$' interpreted as the "less than or equal" relation, '$\sqrt{\ }$' interpreted as "square root", '$+$' interpreted as addition, and '1' interpreted as "one". For assignment $[x \mapsto 3, y \mapsto 8]$, above formula is true; for assignment $[x \mapsto 4, y \mapsto 3]$, it is false.



Now take the domain of points on a plane with '$\leq$' interpreted as "is not farther from the zero point than", '$\sqrt{\ }$' interpreted as "projection of a point to the horizontal axis", '$+$' interpreted as addition of point coordinates, '1' interpreted as point $(1, 0)$. Is the formula true for the assignment $[x \mapsto (2, 2), y \mapsto (1, 1)]$ or not?

The Logic evaluator has the predicates `=` and `<=` on natural numbers built in; all formulas have to be entered in prefix notation:

```
formula =(3, +(2, 2));
> false.
formula <=(3, +(2, 2));
> true.

formula <=(*(2, 3), +(1, 5));
```

Operational Interpretation Atomic formulas are implemented by the following Java class:

```
public final class Atomic implements Formula
{
  private String name;
  private Term[] arguments;

  public Atomic(String _name, Term[] _arguments)
  {
    name = _name;
    arguments = _arguments;
  }

  public boolean eval() throws EvalException
  {
    Predicate predicate = Model.getPredicate(name,arguments.length);
    if (predicate == null)
      throw new EvalException("unknown predicate " + name +
                              "/" + arguments.length);
    Value[] values = new Value[arguments.length];
    for (int i=0; i<values.length; i++)
      values[i] = arguments[i].eval();
    return predicate.apply(values);
  }
}
```

The Java term `new Atomic(`*pc*`, `*args*`).eval()` evaluates to the meaning of the predicate $f(args)$ (where *pc* is a predicate constant and *args* is the list of arguments). As one can see, first we determine the interpretation *predicate* of the predicate constant in the given domain (`Model`), then we evaluate the arguments and apply the predicate to the results.

2.3.3 Equality

Typically we operate in a version of predicate logic that provides a binary predicate constant '=' with the following properties (independently of the domain being considered).

Proposition 14 (Equality) For all x, y, and z, the following holds:

Reflexivity (Reflexivität)

$x = x$;

Symmetry (Symmetrie)

$x = y \Rightarrow y = x$;

Transitivity (Transitivität)

$(x = y \wedge y = z) \Rightarrow x = z$.

Furthermore, let f be a function constant and p be a predicate constant. Then we have:

Equality Axioms (Gleichheitsaxiome)

$$\begin{array}{lll} x = y & \Rightarrow & f(\ldots, x, \ldots) = f(\ldots, y, \ldots); \\ x = y & \Rightarrow & p(\ldots, x, \ldots) \Leftrightarrow p(\ldots, y, \ldots). \end{array}$$

If we know $x = y$, the equality axioms allow us to replace x by y in any phrase without changing the meaning of the phrase.

2.3.4 Quantified Formulas

In elementary terms and atomic formulas (applications of functions and predicates), the meaning of variables depends on a given assignment. We say that these variables are *free (frei)* in the corresponding phrases. We now introduce a concept to give phrases with variables a meaning independently of any variable assignment.

Definition 18 (Quantifier) A *quantifier (Quantor)* is a syntactic operator that combines a variable and a phrase to form a new phrase. The meaning of the new phrase does not depend on a given assignment to determine the value of the quantified variable; we say that this variable is *bound (gebunden)* by the quantifier.

If every variable in a syntactic phrase is bound by a quantifier, the phrase is said to be *closed (geschlossen)*.

The meaning of a phrase does not depend on the name of the bound variable; we can replace this name by any other name without making a difference. Above definition implies that the meaning of a closed phrase does not depend on a given variable assignment but only on the considered domain.

In predicate logic, we have two important kinds of quantified formulas.

Proposition 15 (Quantified Formulas) If x is a variable and A is a formula then the following are formulas with bound variable x:

Universal Quantification "for all x, A" ("für alle x, A")

$$(\forall x : A)$$

Existential Quantification "there exists x, A" ("es existiert x, A")

$$(\exists x : A)$$

We omit the parentheses, if A is unambiguous.

When writing a quantified formula like $(\forall f : A)$ it is clear that any occurrence of f within A should denote a variable (not a constant). Consequently, we do not need to rely on any special convention for the names of quantified variables to distinguish them from function constants.

Alternative Forms The formula $\forall x : A$ appears in various syntactic forms:

- $\forall_x\ A$
- $\bigwedge_x A$;
- "for all x we have A" ("für alle x gilt A");

- "every x has A" ("jedes x hat die Eigenschaft A");
- "A, for all x" ("A gilt für alle x");
- `forall(`x`:`A`)`.

The last line denotes the input syntax of the Logic Evaluator.

Likewise, $\exists x : A$ may be expressed as:

- $\exists_x \; A$
- $\bigvee\limits_{x} A$;
- "there exists x with A" ("es existiert ein x mit Eigenschaft A");
- "there is some x with A" ("es gibt ein x mit Eigenschaft A");
- "some x has A" ("ein x hat die Eigenschaft A");
- "A, for some x" ("A gilt für ein x").
- `exists(`x`:`A`)`.

Again, the last line is the input syntax of the Logic Evaluator.

Example The following formulas are quantified:

- $\forall x : \text{even}(x) \vee \text{odd}(x)$;
- $\forall y : x \leq y$;
- "every x is non-negative"

  $$\forall x : \neg(x < 0);$$

- "prime numbers exist"

  $$\exists x : \text{isPrime}(x).$$

The correct syntactic structure of formulas with nested quantifiers has to be often deduced from the free variables; parentheses should be used whenever necessary to resolve ambiguities.

Example The formula

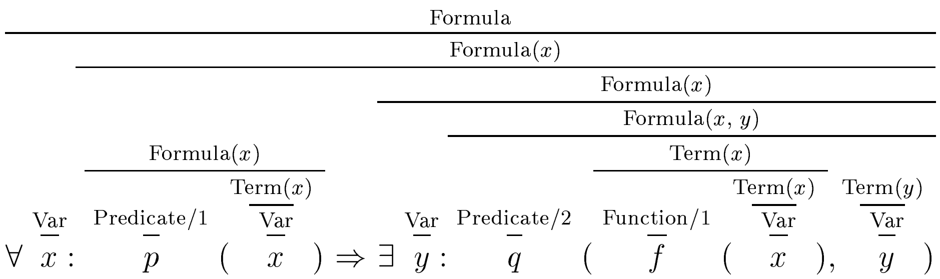
$$\forall x : p(x) \Rightarrow \exists y : q(f(x), y)$$

is to be interpreted as

$$\forall x : (p(x) \Rightarrow \exists y : q(f(x), y)).$$

and not as $(\forall x : p(x)) \Rightarrow \exists y : q(f(x), y)$.

Its syntactic structure is as follows (we list in parentheses the free variables of every term/formula and denote by the suffix '$/n$' the arity n of function/predicate constants):



Definition 19 (Semantics of Quantified Formulas) Let x be a variable and A be a formula. The meaning of a quantified formula under an assignment in some domain is defined as follows:

- The formula $\forall x : A$ is true if and only if A is true for *every* value of x (i.e., A is true in every assignment extension where x is mapped to a value from the domain).
- The formula $\exists x : A$ is true if and only if A is true for *some* value of x (i.e., A is true in some assignment extension where x is mapped to a value from the domain).

Example The universal formula $\forall x : y \leq x$ is true in assignment $[y \mapsto 0]$ over the domain of natural numbers with the usual interpretation of '$\leq$', because $0 \leq x$ is true for every natural number x:

$$0 \leq 0, 0 \leq 1, 0 \leq 2, \ldots$$

Likewise, the existential formula $\exists x : x|15$ is true for every assignment over the natural numbers with '|' interpreted as 'divides', because $x|15$ is true for some natural number x (e.g. for 3):

$$3|15.$$

Example Take the formula

$$\forall x : a(x) \Rightarrow b(x, f(x)).$$

In the domain of integer numbers with a interpreted as "is non-negative", f interpreted as "square of" and b interpreted as "is less than or equal", the formula expresses the (true) statement

> Every non-negative integer number is less than or equal its square.

In the domain of character strings with a interpreted as "is non-empty", f interpreted as "string without first character" and b interpreted as "is longer than", the formula expresses the (true) statement

> Every non-empty character string is longer than the string without the first character.

Free and Bound Variables As said above, the meaning of a formula does not depend on the name of a *bound* variable. For instance, the meaning of

$$\forall x : \exists y : x * y \leq z$$

is the same as the meaning of

$$\forall y : \exists w : y * w \leq z$$

but it is *not* the same as the meaning of

$$\forall x : \exists y : x * y \leq w.$$

(because a free variable must not be renamed).

When stating a proposition, we usually operate with closed formulas, since their meaning is independent of any variable assignment. For instance, assume that we consider the domain of natural numbers with '$\leq$' interpreted as usual. The statement

$$\forall y : x \leq y$$

is true, if the current assignment maps the free variable x to 0, and false, otherwise. On the contrary, the closed formula

$$\exists x : \forall y : x \leq y$$

is true *independently* of the variable assignment. Sometimes, it is assumed that all free variables in a formula are universally quantified such that a formula

$$\forall x : \forall y : x \leq x + y$$

could be also written as

$$x \leq x + y.$$

We recommend to always write all quantifiers to avoid misinterpretations.

We will sometimes use the following notation.

Definition 20 (Free Variable Substitution) For every syntactic phrase P, variable x, and term T we denote by

$$P[x \leftarrow T]$$

the phrase we get from P by replacing every *free* occurrence of x by T.

Example Let $F := p(x) \wedge r(x,y) \wedge \forall x : p(x)$. Then

$$F[x \leftarrow 1] = p(1) \wedge r(1,y) \wedge \forall x : p(x)$$

Patterns A frequently occuring formula pattern is $\forall x : A_x \Rightarrow B$ where A_x is a formula with free variable x. Such a formula is often written as

$$\forall A_x : B$$

provided that the quantified variable x is clear from the context. Another such pattern is $\exists x : A_x \wedge B$ which is typically written as

$$\exists A_x : B.$$

Example

- The formula

  $$\forall x : \exists 0 \leq y \leq x : |x - 2 * y| \leq 1$$

  is an abbreviation for the formula

  $$\forall x : (\exists y : (0 \leq y) \wedge (y \leq x) \wedge (|x - 2 * y| \leq 1)).$$

- The natural language sentence

  "All Ferraris are red"

  is an abbreviation for the sentence

  "It holds for all objects that, if the object is a Ferrari, then the object is red"

  which is represented by the formula

  $$\forall x : \text{isFerrari}(x) \Rightarrow \text{isRed}(x).$$

- The natural language sentence

  "Some Ferraris are black"

  is an abbreviation for the sentence

  "There exists an object such that the object is a Ferrari and the object is black"

  which is represented by the formula

  $$\exists x : \text{isFerrari}(x) \wedge \text{isBlack}(x).$$

The Logic Evaluator can evaluate quantified formulas that are input in prefix form; however, we have to denote the domain that we consider (which must be finite). In the following example, our domain is the range of natural numbers less than or equal 100.

```
option universe = nat(0, 100);
> universe of discourse set.
formula forall(x: <=(x, 1000));
> true.
formula exists(x: =(*(x, 2), 56));
> true.
formula exists(x: =(*(x, 2), 49));
> false.
formula forall(x: exists(y:
  or(=(*(2, y), x),
     =(*(2, y), +(x, 1)))));
> true.
```

```
formula exists(x: forall(y: <=(y, x)));
```

We may however also explicitly set the domain for a quantified variable:

```
formula forall(x in nat(0, 100): <=(x, 1000));
> true.
formula forall(x in nat(0, 100): exists(y in nat(0, x):
  or(=(*(2, y), x),
     =(*(2, y), +(x, 1)))));
> true.
```

```
formula exists(x in nat(0, 100): forall(y in nat(0, 100): <=(y, x)
```

Operational Interpretation Variables are implemented by the following Java class:

```
public final class Variable implements Term
{
  private String variable;

  public Variable(String _variable)
  {
    variable = _variable;
  }

  public String name()
  {
    return variable;
  }

  public Value eval() throws EvalException
  {
    Value value = Context.get(variable);
    if (value == null) throw
      new EvalException("no variable " + variable + " in context");
    return value;
  }
}
```

As we can see, the value of a variable is just determined by the context (the current variable assignment). This context is, e.g. provided by a universally quantified formula as an instance of the following Java class:

```
public final class ForAll implements Formula
{
  private String variable;
  private Term domain;
  private Formula formula;

  public ForAll(String _variable, Term _domain, Formula _formula)
  {
    variable = _variable;
    domain = _domain;
    formula = _formula;
  }

  public boolean eval() throws EvalException
  {
    Iterator iterator = Model.iterator(domain);
    while (iterator.hasNext())
    {
```

```
        Context.begin(variable, iterator.next());
        boolean result = formula.eval();
        Context.end();
        if (!result) return false;
      }
      return true;
    }
  }
```

The Java expression `(new Forall(`x`, `d`, `A`)).eval()` computes the truth value of $\forall x : A$ in domain d. As one can see, we iterate over d and establish new contexts in which x is assigned each value of the domain in turn. In each context, we evaluate A. Only if the result is true for every value, the overall result is true.

Likewise, we have the following implementation of existential quantification:

```
public final class Exists implements Formula
{
  private String variable;
  private Term domain;
  private Formula formula;

  public Exists(String _variable, Term _domain, Formula _formula)
  {
    variable = _variable;
    domain = _domain;
    formula = _formula;
  }

  public boolean eval() throws EvalException
  {
    Iterator iterator = Model.iterator(domain);
    while (iterator.hasNext())
    {
      Context.begin(variable, iterator.next());
      boolean result = formula.eval();
      Context.end();
      if (result) return true;
    }
    return false;
  }
}
```

To compute the value of the expression `(new Exists(`x`, `d`, `A`)).eval()`, we iterate over d and establish all contexts in which A is evaluated. If we find some value for which this result is true, the overall result is true.

Convention Multiple quantifiers of the same kind are often "merged", i.e., instead of writing $\forall x : \forall y : A$ we often write

$$\forall x, y : A$$

and instead of writing $\exists x : \exists y : A$ we write

$$\exists x, y : A.$$

The same is true for the Logic Evaluator where you can write

```
forall(x0, ..., xn-1: A)
exists(x0, ..., xn-1: A)
```

with any number of quantified variables.

Duality Existential and universal quantification are dual concepts:

Proposition 16 (De Morgan's Laws) For every variable x and formula A, we have

$$\begin{array}{lll} \neg\forall x : A & \text{iff} & \exists x : \neg A \\ \neg\exists x : A & \text{iff} & \forall x : \neg A \end{array}$$

In other words, "not all x have A" means the same as "there exists some x such that not A" and "there does not exist an x with A" means the same as "for all x, not A".

From these laws, it follows that $\exists x : A$ means the same as $\neg\forall x : \neg A$; therefore frequently existential quantification is defined just as a *syntactic abbreviation* in terms of universal quantification (or vice versa).

2.3.5 Local Definitions

Apart from universal and existential quantification of formulas, there exist several other frequently occurring quantifiers, most notably quantifiers for introducing local definitions.

Proposition 17 (Local Definitions) Let x be a variable, T and S be terms, and A be a formula. Then the following are formulas with bound variable x:

$$(\textbf{let } x = T : A)$$
$$(A \textbf{ where } x = T)$$

The following are terms:

$$(\textbf{let } x = T : S)$$
$$(S \textbf{ where } x = T)$$

We omit the parentheses, if A, T, and S are unambiguous.

This definition shows that also terms (not only formulas) may be quantified.

Alternative Forms Various syntactic forms express local definitions.

- Let x be T. Then we have A.
- Let x be T in A.
- A, where $x = T$.
- $f(x) := S_{x,y}$, where $y = T_x$;
- `let(`x` = `T`:  `A`)`

The last line denotes the input syntax for the Logic Evaluator.

Proposition 18 (Semantics of Local Definitions) Let x be a variable, A be a formula, and T and S be terms. Under a given assignment in a domain,

- the meaning of $(\textbf{let } x = T : A)$ respectively that of $(A \textbf{ where } x = T)$ is the meaning of A in the extended assignment where x is mapped to the value of T;
- the meaning of $(\textbf{let } x = T : S)$ respectively that of $(S \textbf{ where } x = T)$ is the value of S in the extended assignment where x is mapped to the value of T.

Thus both operators **let** and **where** are just different syntactic forms for the same concept.

Example Take the domain of natural numbers and assignment $[x \mapsto 1]$. Then the proposition

$$\textbf{let } y = 0 : x \leq y$$

is false because $1 \leq 0$ does not hold.

The function f defined as

$$f(x) := s * x + s \textbf{ where } s = x + 1$$

has the same meaning as if it were defined as

$$f(x) := (x + 1) * x + (x + 1).$$

The last example shows that local definitions are just convenient forms for structuring formulas by giving names to values. We can always remove the quantifier by replacing any free occurrence of the quantified variable x by its defining term T.

Example The formula

$$\begin{array}{l} x|y \wedge \exists z : y|z \\ \quad \textbf{where } y = 2x \end{array}$$

is equivalent to

$$x|2x \wedge \exists z : 2x|z$$

Likewise, the formula

$$\begin{array}{l} x|y \wedge \exists y : y|x \\ \quad \textbf{where } y = 2x \end{array}$$

is equivalent to

$$x|2x \wedge \exists y : y|x$$

However, *it is not equivalent to*

$$x|2x \wedge \exists y : 2x|x$$

because the second occurrence of y is within the scope of another quantifier that binds this variable.

Actually, local definitions can be replaced by existential respectively universal quantifiers.

Proposition 19 (Replacement of Local Definitions) Let x be a variable, T be a term, and A be a formula. Then we have

$$\begin{array}{lll} (\textbf{let } x = T : A) & \text{iff} & (\forall x = T : A) \\ (\textbf{let } x = T : A) & \text{iff} & (\exists x = T : A) \end{array}$$

The universal/existential formulas in above proposition are the previously introduced shortcuts for $(\forall x : x = T \Rightarrow A)$ and $(\exists x : x = T \wedge A)$. The proposition is true, because for every term T there exists exactly one x such that $x = T$.

In the Logic Evaluator, local definitions are implemented for formulas as well as for terms as shown below:

```
formula let(x = +(2, 3), y = +(x, x): <=(0, y));
> true.
term let(x = +(2, 3), y = +(x, x): y);
> 10.

formula <=(let(x = +(2, 3): *(x, x), let(x = +(1, 4): *(x, x)));
```

Operational Interpretation A term with a local definition is implemented in the evaluator by an object of the following Java class:

```
public final class LetTerm implements Term
{
  private String variable;
  private Term term;
  private Term body;

  public LetTerm(String _variable, Term _term, Term _body)
  {
    variable = _variable;
    term = _term;
    body = _body;
  }

  public Value eval() throws EvalException
  {
    Context.begin(variable, term.eval());
    Value result = body.eval();
    Context.end();
    return result;
  }
}
```

The Java expression `new LetTerm(`x`, `T`, `S`)` thus evaluates to the meaning of **let** $x = T : S$. As one can see, we construct a new context in which x is bound to the value of T and evaluate S in this context.

Formulas with local definitions are implemented analogously.

Convention We merge multiple local definitions, i.e., instead of writing (**let** $x = 1$: **let** $y = 2 : (x + y) * (x - y)$), we usually write

$$\textbf{let } x = 1, y = 2 : (x + y) * (x - y)$$

or correspondingly

$$(x + y) * (x - y) \textbf{ where } x = 1, y = 2$$

Also the Logic Evaluator implements local definitions

`let(`x_0` = `T_0`, ..., `x_{n-1}` = `T_{n-1}`: `A`)`

with any number of quantified variables.

2.4 Example

We demonstrate the beauty and usefulness of the language of logic on a simple example. Suppose we are given the following problem statement:

> Write a program that takes a number and returns the next prime number.

Specification The task can be formally specified as follows:

- **Input:** n such that $\mathit{isNumber}(n)$;
- **Output:** p such that $\mathit{isNextPrime}(n, p)$.

Here *isNumber* is the *input condition (Eingabebedingung)*, a unary predicate that describes the property that every well-formed input n has; *isNextPrime* is the *output condition (Ausgabebedingung)*, a binary predicate that describes how the output p shall be related to n. Our goal is to construct these two conditions.

As for the input condition, we can reasonably assume from the problem statement that the input is a natural number. We denote by $\mathbb{N}$ the set of natural numbers and define

$$\mathit{isNumber}(n) :\Leftrightarrow n \in \mathbb{N}$$

using the set theoretic predicate '$\in$' ("is element of") to be introduced in the next chapter and the notion '$:\Leftrightarrow$' to define a new predicate.

The output condition is a bit more complicated. Apparently, the result p shall be again an object of the same kind as input n; but it shall be also a prime and it shall be the "next" such number. Thus we define

$$\mathit{isNextPrime}(n, p) :\Leftrightarrow \mathit{isNumber}(p) \wedge \mathit{isPrime}(p) \wedge \mathit{isNextP}(n, p).$$

A natural number p greater than 1 is a prime if and only if there is no number n between 1 and p that divides p

$$\begin{array}{l} \mathit{isPrime}(p) :\Leftrightarrow \\ \quad 1 < p \wedge \\ \quad \neg(\exists 1 < n < p : n|p). \end{array}$$

This can be written a bit more elegantly as

$$\begin{aligned}&\mathit{isPrime}(p) :\Leftrightarrow\\&\quad 1 < p \wedge\\&\quad \forall 1 < n < p : n \nmid p.\end{aligned}$$

The number n divides m if there exists some p such that the product of n and p is m:

$$n|m :\Leftrightarrow \exists p : n * p = m.$$

Thus we have reduced the description of the predicate *isPrime* to formulas that only use the function constant $*$ and the predicates $=$ and $<$ with their usual interpretation on natural numbers.

Now we proceed to the question what n's *next* prime p is. A little thinking reveals that the notion "next" intends to express that there is no other number between n and p that is also a prime. But what if n is itself prime? Shall we then return n itself or the smallest prime *greater* than n? The natural language term "next" is somewhat ambiguous in this respect, so we discuss with our client and learn that the first interpretation is the desired one. Thus we write:

$$\begin{aligned}&\mathit{isNextP}(n,p) :\Leftrightarrow\\&\quad n \leq p \wedge\\&\quad \neg(\exists n \leq q < p : \mathit{isPrime}(q)).\end{aligned}$$

The predicate *isNumber* is rather trivial and the predicate *isNextP* in isolation does not describe a very interesting property; thus we insert the corresponding formulas to yield the final specification:

- **Input:** $n \in \mathbb{N}$;
- **Output:** $p \in \mathbb{N}$ such that $\mathit{isNextPrime}(n,p)$.

$$\begin{aligned}&\mathit{isNextPrime}(n,p) :\Leftrightarrow\\&\quad n \leq p \wedge \mathit{isPrime}(p) \wedge\\&\quad \neg(\exists n \leq q < p : \mathit{isPrime}(q))\end{aligned}$$

$$\begin{aligned}&\mathit{isPrime}(p) :\Leftrightarrow\\&\quad 1 < p \wedge\\&\quad \forall 1 < n < p : n \nmid p\end{aligned}$$

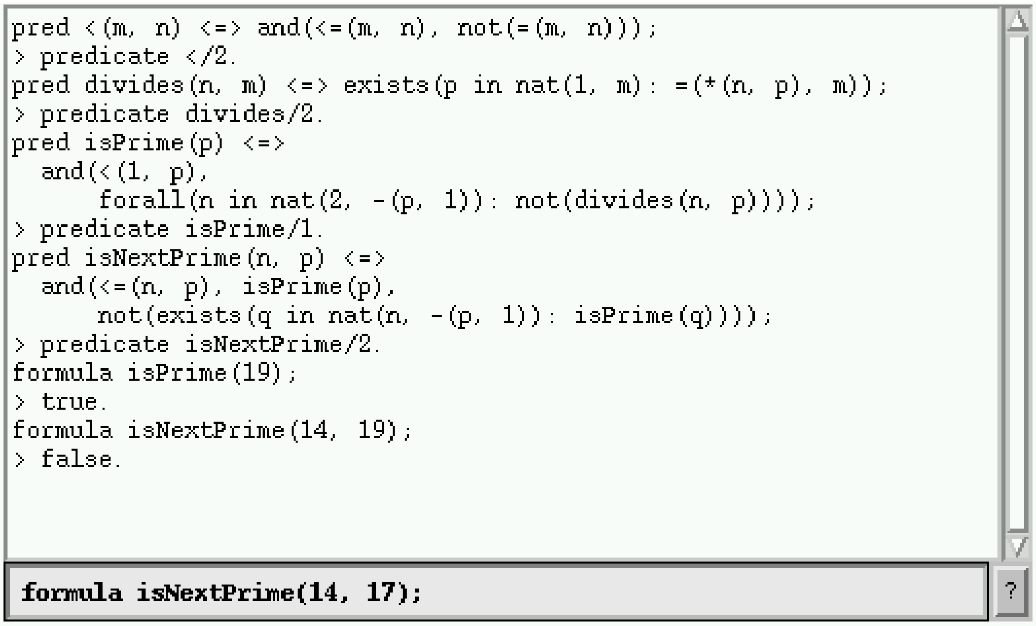
$$n|m :\Leftrightarrow \exists p : n * p = m.$$

These predicates can be defined in the Logic Evaluator as follows:

```
pred <(m, n) <=> and(<=(m, n), not(=(m, n)));
pred divides(n, m) <=> exists(p in nat(1, m): =(*(n, p), m));
pred isPrime(p) <=>
  and(<(1, p),
      forall(n in nat(2, -(p, 1)): not(divides(n, p))));
pred isNextPrime(n, p) <=>
  and(<=(n, p), isPrime(p),
      not(exists(q in nat(n, -(p, 1)): isPrime(q))));
```

The only difference to the mathematical definitions is that we have to explicitly restrict the range of the existentially quantified variable p in the definition of `divides` (which is easy, because any divisor of m is in the range $[1 \ldots m]$).

We can use these definitions to check the validity of input/output pairs:



```
pred <(m, n) <=> and(<=(m, n), not(=(m, n)));
> predicate </2.
pred divides(n, m) <=> exists(p in nat(1, m): =(*(n, p), m));
> predicate divides/2.
pred isPrime(p) <=>
  and(<(1, p),
      forall(n in nat(2, -(p, 1)): not(divides(n, p))));
> predicate isPrime/1.
pred isNextPrime(n, p) <=>
  and(<=(n, p), isPrime(p),
      not(exists(q in nat(n, -(p, 1)): isPrime(q))));
> predicate isNextPrime/2.
formula isPrime(19);
> true.
formula isNextPrime(14, 19);
> false.

formula isNextPrime(14, 17);
?
```

We will now use the "such" quantifier (see page 292)

such(x: A, T)

which returns a value of term T such that A holds (for some value of x, provided that such an x exists). With the help of this quantifier we can define a function

```
fun program(n) =
  such(p in nat(n, *(2, n)): isNextPrime(n, p), p);
```

We restrict our search for the next prime to the range $[n \dots 2n]$ applying the mathematical knowledge that between every number and its double there exists at least one prime number. Thus we can simulate a program that satisfies our specification:

```
pred <(m, n) <=> and(<=(m, n), not(=(m, n)));
> predicate </2.
pred divides(n, m) <=> exists(p in nat(1, m): =(*(n, p), m));
> predicate divides/2.
pred isPrime(p) <=>
  and(<(1, p),
      forall(n in nat(2, -(p, 1)): not(divides(n, p))));
> predicate isPrime/1.
pred isNextPrime(n, p) <=>
  and(<=(n, p), isPrime(p),
      not(exists(q in nat(n, -(p, 1)): isPrime(q))));
> predicate isNextPrime/2.
fun program(n) = such(p in nat(n, *(2, n)): isNextPrime(n, p), p);
> function program/1.
term program(14);
> 17.
term program(50);
> 53.

term program(90);
```

Operational Interpretation From the formal specification constructed above, one can already derive a straight-forward program that solves the stated problem. We recall the operational interpretations of formulas introduced in the previous sections and encode every formula $(\forall a \leq x \leq b : A)$ as the following piece of code

```
boolean forallX = true;
for (int x = a; x <= b; x++)
{
  // value of A(x)
  boolean isTrue = ...;
  ...

  if (!isTrueA)
  {
    forallX = false;
```

```
    break;
  }
}
```

Then the variable `forallX` is true after the termination of the loop if and only if the universal formula is true.

Correspondingly, we can encode $(\exists a \leq x \leq b : A)$ as

```
boolean existsX = false;
for (int x = a; x <= b; x++)
{
  // value of A(x)
  boolean isTrue = ...;
  ...

  if (isTrueA)
  {
    existsX = true;
    break;
  }
}
```

Here the variable `existsX` is true after the termination of the loop if and only if the existential formula is true.

Based on this idea, we implement the predicates as the following Java methods:

```
// ----------------------------------------------------------
// divides(m, n) :<=> (exists p in 1..n: m*p = n)
// ----------------------------------------------------------
boolean divides(int m, int n)
{
  // exists p in 1..n
  boolean existsP = false;
  for (int p = 1; p <= n; p++)
  {
    // m*p = n
    boolean equalsMPN = (m*p == n);
    if (equalsMPN)
    {
      existsP = true;
      break;
    }
  }
  return existsP;
}
```

```
// ----------------------------------------------------
// isPrime(p) :<=>
//   1 < p /\
//   (forall n in 2..p-1: ~divides(n, p))
// ----------------------------------------------------
boolean isPrime(int p)
{
  // p > 1 /\
  if (p > 1)
  {
    // forall n in 2..p-1
    boolean forallN = true;
    for (int n = 2; n <= p-1; n++)
    {
       // ~divides(n, p)
       boolean notDividesNP = !divides(n, p);
       if (!notDividesNP)
       {
          forallN = false;
          break;
       }
    }
    return forallN;
  }
  else
    return false;
}

// ----------------------------------------------------
// isNextPrime(n, p) :<=>
//   n <= p /\ isPrime(p) /\
//   ~exists(q in n..p-1: isPrime(q))));
// ----------------------------------------------------
boolean isNextPrime(int n, int p)
{
  // n <= p /\
  if (n <= p)
  {
    // isPrime(p) /\
    if (isPrime(p))
    {
      // ~exists q in nat(n, p-1):
      boolean existsQ = false;
      for (int q = n; q <= p-1; q++)
      {
        boolean isPrimeQ = isPrime(q);
        if (isPrimeQ)
        {
```

```
          existsQ = true;
          break;
        }
      }
      if (existsQ)
        return false;
      else
        return true;
    }
    else
     return false;
  }
  else
    return false;
}

// ----------------------------------------------------------
// program(n) := such(p in n..2*n: isNextPrime(n, p))
// ----------------------------------------------------------
int program(int n)
{
  for (int p = n; p <= 2*n; p++)
  {
    if (isNextPrime(n, p))
      return p;
  }
  return -1; // not reached
}
```

While above program solves the stated problem, a little bit of thinking reveals that we can do this more efficiently. The first prime number found is exactly the wanted "next" prime such that we do not need an extra check for this property. Thus we write:

```
// ---------------------------------------------------------
// program(n) := such(p in n..2*n: isPrime(p))
// ---------------------------------------------------------
int program(int n)
{
  for (int p = n; p <= 2*n; p++)
  {
    if (isPrime(p))
      return p;
  }
  return -1;
}
```

Chapter 3

Sets, Relations, and Functions

Frequently, logical formulas are interpreted in a domain whose objects are of the datatype *set (Menge)*. The importance of this domain stems from its universality: virtually all other types of objects that occur in mathematical work (relations, functions, numbers, arrays, lists, trees, databases, ...) can with the help of a few basic constructions be modelled as sets. The properties of these objects are then determined entirely by the properties of sets; the theory of sets thus provides the building material for most other theories.

Intuitively, a set is a collection of elements. However, since sets shall serve as a fundamental kind of objects, there is no point in asking what (other object) a set *is* (if we could answer this question, we would have a more fundamental kind of object). Sets are therefore not defined by what they *are* but by what one *knows* about (respectively can *do* with) them. In other words, the domain of sets is characterized by various *axioms (Axiome)*, i.e., propositions that are stipulated to be true.

A common axiomatization of *set theory (Mengenlehre)* is due to Zermelo and Fraenkel; this form of set theory is called ZF set theory. We will not list all ZF axioms but focus on their consequences for practical work.

3.1 The Datatype Set

The domain of sets has a single binary predicate *is element of (ist Element von)* denoted by the infix constant '$\in$'. All other predicates and functions are defined by this predicate.

The formula $x \in y$ is read as

- x is element of y;
- x is part of y;
- y contains x.

Please note that in set theory *all* objects are sets (e.g. also the number 1 is just a particular set). Thus in a statement $x \in y$, both x and y denote sets the first of which is element of the second one.

The following axiom states that sets have no other property than being element containers:

Axiom 1 (Equality of Sets) Two sets are equal if and only if they have the same elements, i.e., for all x and y, we have

$$x = y \Leftrightarrow (\forall z : z \in x \Leftrightarrow z \in y).$$

Furthermore, we have a "minimal" set:

Axiom 2 (Empty Set) There exists a set that is empty, i.e., that does not contain any elements:

$$\exists x : \forall y : y \notin x.$$

Because of Axiom 1, there is only one empty set such that we can define

$$\emptyset := \textbf{such } x : \forall y : y \notin x.$$

Finite sets (i.e., sets with a finite number of elements) can be constructed by explicit enumeration.

Proposition 20 (Set Enumeration) Let T_0, T_1, $\ldots$, T_{n-1} be terms (for any n). Then

$$\{T_0, T_1, \ldots, T_{n-1}\}$$

is a term that denotes the set of all the T_i values, i.e., for every x, we have

$$x \in \{T_0, T_1, \ldots, T_{n-1}\} \Leftrightarrow (x = T_0 \vee x = T_1 \vee \ldots \vee x = T_{n-1}).$$

As a special case, we have $\{\} = \emptyset$.

Example The set $S := \{1, \emptyset, \{1, 2\}, a\}$ contains the values

$$1, \emptyset, \{1, 2\}, a,$$

i.e., $1 \in S$, $\emptyset \in S$, $\{1, 2\} \in S$, $a \in S$.

We have the following set equalities:

$$\{1, 2\} = \{2, 1\};$$
$$\{1, 2\} = \{1, 1, 2, 1\}.$$

Logic Evaluator The predicate "is element of" is denoted by (prefix) `in` and the empty set is denoted by $\{\}$. The binary function `join` adds an element to a set, i.e.,

$$\texttt{join}(T_0,\ \texttt{join}(T_1,\ \ldots,\ \texttt{join}(T_{n-1},\ \{\}))) = \{T_0, T_1, \ldots, T_{n-1}\}.$$

The binary predicate `=` also denotes set equality (we will see later how we can define a corresponding predicate ourselves).

```
term {};
> {}.
formula in(1, {});
> false.
fun S = join(1, join(2, join (3, join(2, {}))));
> function S/0.
term S;
> {2, 3, 1}.
formula in(2, S);
> true.
term join({}, {});
> {{}}.
term join(1, join(join(2, {}), join({}, {})));
> {{}, {2}, 1}.
formula =(S, join(3, join(2, join(1, {}))));
> true.
```

```
formula =({}, join({}, {}));
```

Sets have an important “ordering” relationship:

Definition 21 (Subset) x is a *subset (Teilmenge)* of y, if every element of x is also an element of y, i.e.,

$$x \subseteq y :\Leftrightarrow (\forall z \in x : z \in y).$$

```
pred isSubset(x, y) <=> forall(z in x: in(z, y));
> predicate isSubset/2.
formula isSubset({}, {});
> true.
formula isSubset({}, join(1, {}));
> true.
formula isSubset(join(2, {}), join(1, {}));
> false.
```

```
formula isSubset(join(1, join(2, {})), join(2, join(3, join(1, {})
```

The subset relationship is an ordering in the following sense.

Proposition 21 (Subset Properties) For every x, y, and z we have:

Minimum (Kleinstes Element)

$\emptyset \subseteq x$;

Reflexivity (Reflexivität)

$x \subseteq x$;

Antisymmetry (Antisymmetrie)

$(x \subseteq y \wedge y \subseteq x) \Rightarrow y = x$;

Transitivity (Transitivität)

$(x \subseteq y \wedge y \subseteq z) \Rightarrow x \subseteq z$.

Furthermore, we have the following relationship between equality and the subset predicate:

Proposition 22 (Equality and Subset) For every x and y, we have

$$x = y \Leftrightarrow (x \subseteq y \wedge y \subseteq x).$$

Proof Take arbitrary x and y. We have to prove $x = y \Leftrightarrow (x \subseteq y \wedge y \subseteq x)$.

- We prove $x = y \Rightarrow (x \subseteq y \wedge y \subseteq x)$. Assume $x = y$, i.e., by definition of '$=$',

  (1) $\forall z : z \in x \Leftrightarrow z \in y$.

  We have to prove $x \subseteq y \wedge y \subseteq x$.

  - We prove $x \subseteq y$, i.e., by definition of '$\subseteq$', $\forall z \in x : z \in y$. Take arbitrary z. We have to prove $z \in x \Rightarrow z \in y$. Assume (2) $z \in x$. We have to prove $z \in y$ which is a consequence of (1) and (2).
  - The proof of $y \subseteq x$ proceeds analogously.

- We prove $(x \subseteq y \wedge y \subseteq x) \Rightarrow x = y$. Assume $x \subseteq y \wedge y \subseteq x$, i.e., by definition of '$\subseteq$'

  (1) $\forall z \in x : z \in y$;
  (2) $\forall z \in y : z \in x$.

  We prove $x = y$, i.e., by definition of '$=$', $\forall z : z \in x \Leftrightarrow z \in y$. Take arbitrary z. We have to prove $z \in x \Leftrightarrow z \in y$.

  - We prove $z \in x \Rightarrow z \in y$. Assume (3) $z \in x$. We have to prove $z \in y$ which is a consequence of (1) and (3).
  - We prove $z \in y \Rightarrow z \in x$. Assume (4) $z \in y$. We have to prove $z \in x$ which is a consequence of (2) and (4).

Logic Evaluator Using the knowledge proved above, we may define equality of sets as follows:

```
pred isSubset(x, y) <=> forall(z in x: in(z, y));
> predicate isSubset/2.
pred equals(x, y) <=> and(isSubset(x, y), isSubset(y, x));
> predicate equals/2.
formula equals(join(1, {}), join(1, join(2, {})));
> false.
```

```
ormula equals(join(1, join(2, {})), join(2, join(1, join(1, {}))))
```

Most frequently, sets are constructed with the help of the following special quantifier that builds a term from a term and a formula:

Proposition 24 (Set Quantifier) Let x be a variable, S a term, and A a formula; then the following is a term:

$$\{x \in S : A\}.$$

The value of this term is the set of all elements x in S such that A holds, i.e., for all x, we have

$$x \in \{x \in S : A\} \Leftrightarrow (x \in S \wedge A).$$

We call S the *variable domain (Variablenbereich)* of x. If S is clear from the context, we may just write $\{x : A\}$.

Example Let S be $\{1, 2, 3, \ldots, 10\}$. Then the set

$$\{x \in S : x \leq 3 \vee x \text{ is even}\}$$

is $\{1, 2, 3, 4, 6, 8, 10\}$.

Remark The set quantifier allows us to construct sets only from elements of a (possibly implicitly) given set. We might want to lift this restriction, i.e., work with a general quantifier

$$\{x : A\},$$

such that, for every x, $x \in \{x : A\} \Leftrightarrow A$. However, with such a general quantifier we could also define a set $S := \{x : x \notin x\}$, i.e., the set of all sets that do not contain themselves as elements. Then we might ask ourself whether $S \in S$. If yes, then this would contradict the construction principle. If no, then $S \in S$ would be true by the construction principle, which would contradict our assumption. In any case, we would have a contradiction, therefore neither $S \in S$ nor $S \notin S$ could hold. This so called *Russel Paradox* is avoided by restricting the quantifier as shown above.

Generalization The set quantifier is often used in a more general form. Let T_x be a term with free variable x, S be a term, and A be a proposition. Then the term

$$\{T_x : x \in S \wedge A\}$$

denotes the set of all values of T_x such that A holds where x is an element of S, i.e., it is the same as

$$\{y : (\exists x \in S : y = T_x \wedge A)\}.$$

If S is clear, the formula $x \in S$ is often skipped; the bound variable x then has to be deduced from the context.

Example The term

$$\{2 * x : 1 \leq x \leq 5\}$$

is usually interpreted as

$$\{2 * x : x \in \mathbb{N} \wedge 1 \leq x \leq 5\}$$

(where $\mathbb{N}$ is the set of natural numbers) which denotes the set $\{2, 4, 6, 8, 10\}$. Likewise, the term

$$\{x + y : 1 \leq x \leq 5\}$$

denotes in assignment $[y \mapsto 1]$ the set $\{2, 3, 4, 5, 6\}$ (assuming that y is not bound by the set quantifier).

Logic Evaluator The general form of the set quantifier is available as

$$\texttt{set}(x \text{ in } S\text{: } A,\, T) = \{T_x : x \in S \wedge A\}.$$

```
term set(x in nat(1, 10): true, x);
> {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}.
term set(x in nat(1, 10): true, +(x, x));
> {2, 4, 6, 8, 10, 12, 14, 16, 18, 20}.
pred divides(m, n) <=> exists(p in nat(1, n): =(n, *(p, m)));
> predicate divides/2.
```

```
term set(x in nat(1, 10): divides(2, x), *(x, x));
```

Generalization Set quantifiers may also range over multiple variables, e.g.

$$\{Z_{x,y} : x \in S \wedge y \in T \wedge A\}$$

with the meaning

$$\{z : (\exists x \in S, y \in T : z = Z_{x,y} \wedge A)\}.$$

Again, the formula $x \in S \wedge y \in T$ is frequently skipped and the quantified variables have to be deduced from the context.

Example The term

$$\{x + y : 1 \leq x \leq 3 \wedge 0 \leq y \leq 2\}$$

is usually interpreted as

$$\{x + y : x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge 1 \leq x \leq 3 \wedge 0 \leq y \leq 2\}.$$

Its value is the same as that of the term

$$\{s : (\exists x \in \mathbb{N}, y \in \mathbb{N} : s = x + y \wedge 1 \leq x \leq 3 \wedge 0 \leq y \leq 2)\}.$$

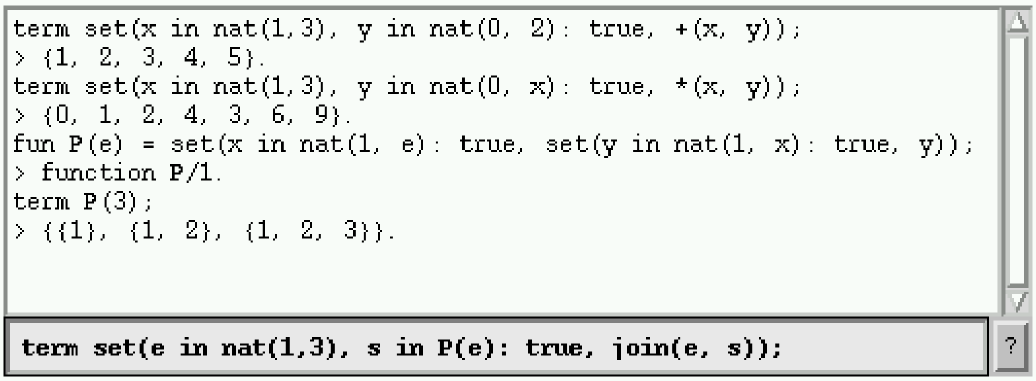
which is the set

$$\{1+0, 1+1, 1+2, 2+0, 2+1, 2+2, 3+0, 3+1, 3+2\}$$

which again equals

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

Logic Evaluator The set quantifier also binds multiple variables:



Operational Interpretation In the Logic Evaluator, set quantifiers with single variables are instances of the following Java class:

```
public final class SetTerm implements Term
{
  private String variable;
  private Term domain;
  private Formula formula;
  private Term element;

  public SetTerm(String _variable, Term _domain,
                 Formula _formula, Term _element)
  {
    variable = _variable;
    domain = _domain;
    formula = _formula;
    element = _element;
  }

  public Value eval() throws EvalException
  {
    Set set = new Set();
    Iterator iterator = Model.iterator(domain);
    while (iterator.hasNext())
    {
      Context.begin(variable, iterator.next());
      if (formula.eval())
        set.addElement(element.eval());
      Context.end();
    }
    return set;
  }
}
```

The evaluation of the Java term `(new SetTerm(`x`, `S`, `A`, `T`)).eval()` results in the set $\{T_x : x \in S \wedge A\}$. We iterate over the set denoted by S and establish new contexts in which x is mapped to each element of S in turn. In each context, we evaluate the formula A and, if the result is `true`, evaluate the term T whose value is added to the set.

With the help of the set quantifier, we can define a number of important functions on sets[1]:

Definition 22 (Set Functions) We define the following functions on sets:

Union (Vereinigung)

"the union of x and y"

$$x \cup y := \{z : z \in x \vee z \in y\}$$

also denoted by $x + y$;

"the union of all elements of x"

$$\bigcup x := \{z : (\exists y \in x : z \in y)\}$$

also used as a quantor $\bigcup_{x \in S \wedge A} T := \bigcup\{T : x \in S \wedge A\}$.

Intersection (Durchschnitt)

"the intersection of x and y"

$$x \cap y := \{z \in x : z \in y\}$$

also denoted by $x * y$;

"the intersection of all elements of x"

$$\bigcap x := \{z \in \bigcup x : (\forall y \in x : z \in y)\}$$

also used as a quantor $\bigcap_{x \in S \wedge A} T := \bigcap\{T : x \in S \wedge A\}$.

Difference (Differenz)

"the difference of x and y" ("the complement of y in x").

$$x \backslash y := \{z \in x : z \notin y\}$$

also denoted by $x - y$;

[1] See the followup comments on the definition of "union" and "powerset".

Powerset (Potenzmenge)

"the powerset of x"

$$\mathbb{P}(x) := \{y : y \subseteq x\}$$

also denoted by 2^x.

Example Let $S := \{1,2,3,4,5\}$, $T := \{2,5,7\}$, $U := \{1,3,5,7,9\}$. Then we have

$$\begin{aligned}
&S \cap T = \{2,5\};\\
&S \cup T = \{1,2,3,4,5,7\};\\
&\bigcap\{S,T,U\} = \{5\};\\
&\bigcup\{S,T,U\} = \{1,2,3,4,5,7,9\};\\
&\mathbb{P}(T) = \{\emptyset, \{2\}, \{5\}, \{7\}, \{2,5\}, \{2,7\}, \{5,7\}, \{2,5,7\}\}.
\end{aligned}$$

Let $\mathbb{N}$ denote the set of natural numbers and $\mathbb{N}_n := \{x \in \mathbb{N} : x < n\}$. Then we have

$$\begin{aligned}
&\bigcup_{i\in\mathbb{N}} \mathbb{N}_i = \mathbb{N};\\
&\bigcap_{i\in\mathbb{N}} \mathbb{N}_i = \{\};
\end{aligned}$$

The syntactic structure of $\bigcup_{i\in\mathbb{N}} \mathbb{N}_i$ is as follows (with free variables denoted in parentheses):

$$\overbrace{\overbrace{\bigcup}^{\text{Quantor}}\ \overbrace{\overbrace{i}^{\text{Var}} \in \overbrace{\mathbb{N}}^{\text{Term}}}^{\text{Formula}(i)}\ \overbrace{\mathbb{N}_{\overbrace{i}^{\text{Var}}}}^{\text{Term}(i)}}^{\text{Term}}.$$

Remark Above definitions of union and powerset make use of the general set quantor whose potential problems have been discussed before. Actually, in set theory union and powerset are introduced in a different way: there is an *axiom* that stipulates the existence of a unary function $\bigcup$ with the property that, for every x,

$$(\forall z : z \in \bigcup x \Leftrightarrow (\exists y \in x : z \in y))$$

which is equivalent to the definition given above. Based on this axiom, binary union is then defined as $x \cup y := \bigcup\{x, y\}$. Likewise, there is an axiom that stipulates the existence of a unary function '$\mathbb{P}$' with the property that, for every x,

$$(\forall y : y \in \mathbb{P}(x) \Leftrightarrow (\forall z \in y : z \in x))$$

which is equivalent to the definition given above.

Logic Evaluator Also the implementation of unary and binary union as well as of powersets is not based on the set quantifier but on set reduction (see page 304). We can use these functions ($\cup$ is denoted by `++`, $\mathbb{P}$ by `Powerset`) by loading a file `set.txt`.

```
read set;
> file 'set.txt' read.
fun S = join(1, join(2, join(3, join(4, join(5, {})))));
> function S/0.
fun T = join(2, join(5, join (7, {})));
> function T/0.
fun U = join(1, join(3, join(5, join(7, join(9, {})))));
> function U/0.
term ++(S, T);
> {7, 5, 2, 4, 3, 1}.
term ++(join(S, join(T, join(U, {}))));
> {9, 7, 5, 3, 1, 2, 4}.
```

```
term Powerset(T);
```

However, binary and unary intersection can be defined with the help of the set quantifier as follows ($\cap$ is denoted by `**`, please compare with the mathematical definitions):

```
read set;
> file 'set.txt' read.
fun **(x, y) = set(z in x: in(z, y), z);
> function **/2.
fun **(x) = set(z in ++(x): forall(y in x: in(z, y)), z);
> function **/1.
fun S = join(1, join(2, join(3, join(4, join(5, {})))));
> function S/0.
fun T = join(2, join(5, join (7, {})));
> function T/0.
fun U = join(1, join(3, join(5, join(7, join(9, {})))));
> function U/0.
term **(S, T);
> {5, 2}.
```

```
term **(join(S, join(T, join(U, {}))));
```

We have a number of laws that relate the basic set operations; some of them are listed below.

Proposition 25 (Set Identities) For every A, B, and C we have:

Idempotency, Identity and Domination

$$A \cup A = A,\ \ A \cup \emptyset = A,$$
$$A \cap A = A,\ \ A \cap \emptyset = \emptyset;$$

Commutativity

$$A \cup B = B \cup A,$$
$$A \cap B = B \cap A;$$

Associativity

$$A \cup (B \cup C) = (A \cup B) \cup C,$$
$$A \cap (B \cap C) = (A \cap B) \cap C;$$

Distributivity

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C),$$
$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C);$$

Cancellation

$$A \cup (A \cap B) = A,$$
$$A \cap (A \cup B) = A;$$

De Morgan

$$C \backslash (A \cup B) = C \backslash A \cap C \backslash B,$$
$$C \backslash (A \cap B) = C \backslash A \cup C \backslash B.$$

Because of associativity, we may write $A \cup B \cup C$ respectively $A \cap B \cap C$ without parentheses.

We now argue for the correctness of $A \cup B = B \cup A$.

Proof Take arbitrary A and B. By definition of $=$, we have to prove

(1) $\forall x : x \in A \cup B \Leftrightarrow x \in B \cup A$.

Take arbitrary x.

- We prove $x \in A \cup B \Rightarrow x \in B \cup A$. Assume

  (2) $x \in A \cup B$.

  We have to prove $x \in B \cup A$. By definition of $\cup$, we have to prove

  (3) $x \in B \vee x \in A$.

  If $x \in B$, we are done. Thus assume (4) $x \notin B$. By (2) and definition of $\cup$, we have (5) $x \in A \vee x \in B$. From (4) and (5), we have $x \in A$ and thus (3).

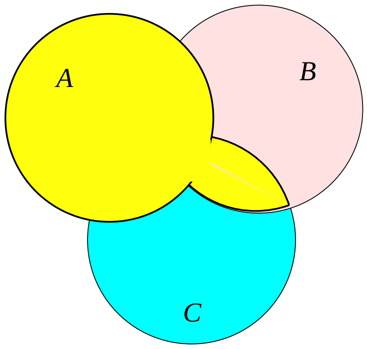
- The proof of $x \in B \cup A \Rightarrow x \in A \cup B$ proceeds analogously.

Many other set identities may be proved with the help of above identities by "equality reasoning" (which relies on the transitivity of $=$).

Example For all A, B, C we have:

$$(B \cup A) \cap (C \cup A) \quad = \quad (B \cap C) \cup A.$$

as illustrated by the following *Venn diagram (Venn-Diagramm)*:



We prove this claim as follows:

$$\begin{array}{rcl} (B \cup A) \cap (C \cup A) & = & \text{(commutativity)} \\ (A \cup B) \cap (A \cup C) & = & \text{(distributivity)} \\ A \cup (B \cap C) & = & \text{(commutativity)} \\ (B \cap C) \cup A. & & \end{array}$$

3.2 Tuples

The elements in a set do not have a particular order, i.e., there is no "first", "second", or "last" set element. The concept of containers that preserve a particular order among their components needs some additional mechanisms.

Proposition 27 (Tuple) For every n, there is an n-ary function $\langle\ \rangle$ (the *tuple constructor*) and n unary functions $._0, ._1, \ldots, ._{n-1}$ (the *tuple selectors* or *projections*) such that for all $x_0, x_1, \ldots, x_{n-1}$ we have

$$\begin{array}{l} \langle x_0, x_1, \ldots, x_{n-1} \rangle_0 = x_0; \\ \langle x_0, x_1, \ldots, x_{n-1} \rangle_1 = x_1; \\ \ldots \\ \langle x_0, x_1, \ldots, x_{n-1} \rangle_{n-1} = x_{n-1}. \end{array}$$

We call $\langle x_0, x_1, \ldots, x_{n-1} \rangle$ an *n-tuple (Tupel)*.

Please do not confuse in above proposition a variable name x_i with the application of the tuple selector $._i$ to a tuple t, denoted as t_i. Please also note that we use the same symbol $\langle\ \rangle$ for the tuple constructors of all arities and the same function symbol $\cdot_i$ for the selection of the i-th component of every n-tuple with $i < n$. Frequently, tuple construction is just denoted by $(x_0, x_1, \ldots, x_{n-1})$.

Example Let $T := \langle 1, 2 \rangle$. Then

$$T_0 = 1;$$
$$T_1 = 2.$$

Let $U := \langle 2, T, \{1\} \rangle$. Then

$$U_0 = 2;$$
$$U_1 = T;$$
$$U_2 = \{1\}.$$

Proposition 28 (Equality of Tuples) Two n-tuples are equal if and only if their components are equal, i.e., for every n and all $x_0, x_1, \ldots, x_{n-1}$ and all $y_0, y_1, \ldots, y_{n-1}$, we have

$$\langle x_0, x_1, \ldots, x_{n-1} \rangle = \langle y_0, y_1, \ldots, y_{n-1} \rangle \Leftrightarrow$$
$$(x_0 = y_0 \wedge x_1 = y_1 \wedge \ldots x_{n-1} = y_{n-1}).$$

Example

$$\langle 1, 2 \rangle \neq \langle 2, 1 \rangle;$$
$$\langle 1, 2 \rangle \neq \langle 1, 2, 2 \rangle.$$

Construction from Sets Tuples are not a basic concept: they can be built from sets as follows. First, we define the constructor for 2-tuples as

$$\langle x, y \rangle := \{\{x\}, \{y, x\}\}.$$

If $x = y$, then t has a single element $\{x\}$ that determines both x and y. If $x \neq y$, then t contains two elements $\{x\}$ and $\{x, y\}$. The 1-element set determines x and its complement in the 2-element set determines y. Thus the selectors can be uniquely defined as:

$$t_0 := \textbf{such } x : \exists y : t = \langle x, y \rangle;$$
$$t_1 := \textbf{such } y : \exists x : t = \langle x, y \rangle.$$

An n-ary tuple is then constructed from 2-tuples as

$$\langle x_0, x_1, \ldots, x_{n-1} \rangle := \langle x_0, \langle x_1, \langle \ldots, \langle x_{n-2}, x_{n-1} \rangle \rangle \rangle \rangle$$

with the selectors being defined accordingly.

Logic Evaluator The constructor $\langle\ \rangle$ is represented by the function `tuple` with selector $._i$ being represented as (prefix) $.i$; the predicate `=` can be also applied to tuples. The unary predicate `Tuple` returns true for tuples only.

```
fun T = tuple(1, 2);
> function T/0.
formula =(T, tuple(2, 1));
> false.
fun S = tuple(2, T, join(1, {}));
> function S/0.
term S;
> <2, <1, 2>, {1}>.
formula Tuple(S);
> true.
term .0(S);
> 2.
term .1(S);
> <1, 2>.

term tuple(.0(S), .1(S), .2(S));
```

Definition 23 (Cartesian Product) The *Cartesian Product (Kartesisches Produkt)* of n sets $S_0, \ldots, S_{n-1}$ is the set of all n-tuples whose i-th component is in S_i:

$$S_0 \times \ldots \times S_{n-1} := \{\langle x_0, \ldots, x_{n-1} \rangle : x_0 \in S_0 \wedge \ldots \wedge x_{n-1} \in S_{n-1}\}.$$

In the Logic Evaluator, we define the binary Cartesian product as follows:

```
fun x(A, B) = set(a in A, b in B: true, tuple(a, b));
> function x/2.
fun N(n) = set(x in nat(0, n): true, x);
> function N/1.
term N(3);
> {0, 1, 2, 3}.
term N(4);
> {0, 1, 2, 3, 4}.
term x(N(3), N(4));
> {<0, 0>, <0, 1>, <0, 2>, <0, 3>, <0, 4>, <1, 0>, <1, 1>, <1, 2>,
<1, 3>, <1, 4>, <2, 0>, <2, 1>, <2, 2>, <2, 3>, <2, 4>, <3, 0>, <3,
1>, <3, 2>, <3, 3>, <3, 4>}.
```

```
term x(N(2), x(N(3), N(4)));
```

3.3 Predicates as Sets

The language of first order predicate logic introduced in Chapter 2 provides variables that represent objects of the considered domain; a variable may not stand for a predicate or a function. Consequently, we cannot state propositions like "for all predicates p" or "there is a function f", which is necessary in a lot of situations. Set theory however provides means to represent predicates and functions as sets, such that we can talk about "higher order objects" (predicates and functions) in a first order framework (using the domain of sets).

Definition 24 (Relation) A *relation (Relation, Beziehung)* is a subset of a Cartesian product:

$$R \text{ is a relation between } S_0, \ldots, S_{n-1} :\Leftrightarrow R \subseteq S_0 \times \ldots \times S_{n-1}.$$

An n-ary relation is a subset of a Cartesian product of n sets:

R is a relation of arity n (an n-ary relation) $:\Leftrightarrow$
  $\exists S_0, \ldots, S_{n-1} : R$ is a relation between $S_0, \ldots, S_{n-1}$;
R is an n-ary relation on S $:\Leftrightarrow$
  $R \subseteq S \times \ldots \times S$ (cartesian product of n instances of S).
R is a relation on S $:\Leftrightarrow$ R is a 2-ary (*binary*) relation on S.

Let R be an n-ary relation. We define the $(n+1)$-ary predicate

$$R(x_0, \ldots, x_{n-1}) :\Leftrightarrow \langle x_0, \ldots, x_{n-1}\rangle \in R.$$

and read this as R *holds on (ist gültig für)* $x_0, \ldots, x_{n-1}$. If R is a binary relation, we may write this as $x_0 R x_1$.

A relation is thus the set-theoretic counterpart of a predicate; the notation $R(x_0, \ldots, x_{n-1})$ is convenient to hide the distinction between relations and predicate constants. We may now quantify over "predicates" as in

$$\forall x : \exists R : (\forall y : R(x, y) \Rightarrow x = y)$$

with the understanding that this is interpreted in the domain of sets as

$$\forall x : \exists R : (\forall y : \langle x, y\rangle \in R \Rightarrow x = y)$$

where R represents a relation, i.e., a set.

Example Take the relation $R := \{\langle x, y\rangle : x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x \leq y\}$ on the natural numbers $\mathbb{N}$. R is also a relation on the integer numbers $\mathbb{Z}$, because $\mathbb{N} \subseteq \mathbb{Z}$.

Take $S := \{\langle x, x/2\rangle : x \in \mathbb{N}\}$. S is a relation between $\mathbb{N}$ and $\mathbb{Q}$ (the set of all rational numbers) and it is also a relation on $\mathbb{Q}$; however, it is not a relation on $\mathbb{N}$ (because $1/2 \notin \mathbb{N}$).

The empty set $\emptyset$ is a relation on every set S.

Logic Evaluator We define the predicates `Relation`(R) (R is a relation), `isRelation`(R, A, B) (R is a relation between A and B) as shown below:

```
pred Relation(R) <=> // R is a relation
  forall(x in R: and(Tuple(x), =(length(x), 2)));
> predicate Relation/1.
pred isRelation(R: Relation, A, B) <=> // relation between A and B
  forall(x in R: and(in(.0(x), A), in(.1(x), B)));
> predicate isRelation/3.
formula Relation({});
> true.
fun R = set(x in nat(0,2), y in nat(0,3): <=(x,*(2,y)), tuple(x,y));
> function R/0.
term R;
> {<0, 0>, <0, 1>, <0, 2>, <0, 3>, <1, 1>, <1, 2>, <1, 3>, <2, 1>,
<2, 2>, <2, 3>}.
formula Relation(R);
> true.
fun S = set(x in nat(0, 3): true, x);
> function S/0.
term S;
> {0, 1, 2, 3}.
formula isRelation(R, S, S);
> true.

formula isRelation(R, S, join(0, join(1, {})));
```

Relations can be represented in various forms.

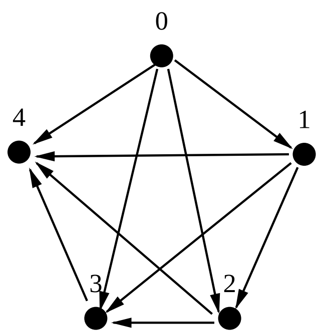
Example Take the relation $R := \{\langle x, y\rangle : x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x < y < 5\}$. It may be represented by enumerating its elements

$$\{\langle 0,1\rangle, \langle 0,2\rangle, \langle 0,3\rangle, \langle 0,4\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle, \langle 2,3\rangle, \langle 2,4\rangle, \langle 3,4\rangle\}.$$

Another representation is a truth matrix where an entry true for $\langle x, y\rangle$ indicates xRy (all entries not shown are false):

$x \backslash y$	0	1	2	3	4	...
0	false	true	true	true	true	
1	false	false	true	true	true	
2	false	false	false	true	true	
3	false	false	false	false	true	
4	false	false	false	false	false	
...						

A graphical representation is that by a *directed graph* where an arrow $x \to y$ indicates xRy:



We now introduce a notion by which we can denote those values that are affected by a relation.

Definition 25 (Domain, Range) Let R be a binary relation. The *domain (Definitionsbereich)* of R is the set of all first components of the tuples in R:

$$\text{domain}(R) := \{r_0 : r \in R\}.$$

The *range (Wertebereich)* of R is the set of all second components of the tuples in R:

$$\text{range}(R) := \{r_1 : r \in R\}.$$

An equivalent characterization is given by the following law:

Proposition 29 (Domain, Range) Let R be a binary relation. Then

$$(\forall x : x \in \text{ domain}(R) \Leftrightarrow \exists y : \langle x, y\rangle \in R);$$
$$(\forall y : y \in \text{ range}(R) \Leftrightarrow \exists x : \langle x, y\rangle \in R).$$

Example Take the relation $R := \{\langle 0,0\rangle, \langle 0,1\rangle, \langle 0,2\rangle, \langle 1,2\rangle\}$. We have $\text{domain}(R) = \{0,1\}$ and $\text{range}(R) = \{0,1,2\}$.

For the empty relation, we have $\text{domain}(\emptyset) = \text{range}(\emptyset) = \emptyset$.

Logic Evaluator The functions `domain` and `range` are defined as shown in the corresponding mathematical definitions:

```
read relation;
> file 'set.txt' read.
> file 'relation.txt' read.
fun domain(R: Relation) = set(x in R: true, .0(x));
> function domain/1.
fun range(R: Relation) = set(x in R: true, .1(x));
> function range/1.
fun R = set(x in nat(0,2), y in nat(0,3): <=(x,*(2,y)), tuple(x,y));
> function R/0.
term R;
> {<0, 0>, <0, 1>, <0, 2>, <0, 3>, <1, 1>, <1, 2>, <1, 3>, <2, 1>,
<2, 2>, <2, 3>}.
term domain(R);
> {0, 1, 2}.
term range(R);
> {0, 1, 2, 3}.
formula isSubset(R, x(domain(R), range(R)));
> true.
```

```
formula =(R, x(domain(R), range(R)));
```

The order of elements in a binary relation can be inverted.

Definition 26 (Inverse of a Relation) Let $R \subseteq A \times B$. The *inverse* of R is the following relation between B and A:

$$R^{-1} := \{\langle b, a\rangle : a \in A \wedge b \in B \wedge \langle a, b\rangle \in R\}.$$

Above definition uses the notion $R \subseteq A \times B$ to denote that R is a binary relation between A and B (as is common mathematical practice). The clause $a \in A \wedge b \in B$ is only added to make the set quantifier conform to the syntax given on page 66.

Example Take the relation $R := \{\langle 0,0\rangle, \langle 0,1\rangle, \langle 0,2\rangle, \langle 1,2\rangle\}$. We have $R^{-1} = \{\langle 0,0\rangle, \langle 1,0\rangle, \langle 2,0\rangle, \langle 2,1\rangle\}$.

Logic Evaluator We can define the function `^-1`(R) (i.e., R^{-1}) as shown below:

```
read relation;
> file 'set.txt' read.
> file 'relation.txt' read.
fun ^-1(R: Relation) = set(x in R: true, tuple(.1(x), .0(x)));
> function ^-1/1.
fun R0 = set(x in nat(0,2),y in nat(0,3): <=(x,*(2,y)), tuple(x,y));
> function R0/0.
fun R1 = set(x in nat(0,2),y in nat(0,3): <=(x,*(2,y)), tuple(y,x));
> function R1/0.
term R0;
> {<0, 0>, <0, 1>, <0, 2>, <0, 3>, <1, 1>, <1, 2>, <1, 3>, <2, 1>,
<2, 2>, <2, 3>}.
term ^-1(R0);
> {<0, 0>, <1, 0>, <2, 0>, <3, 0>, <1, 1>, <2, 1>, <3, 1>, <1, 2>,
<2, 2>, <3, 2>}.
term R1;
> {<0, 0>, <1, 0>, <2, 0>, <3, 0>, <1, 1>, <2, 1>, <3, 1>, <1, 2>,
<2, 2>, <3, 2>}.
formula =(^-1(R0), R1);
> true.

formula =(R0, ^-1(^-1(R0)));
```

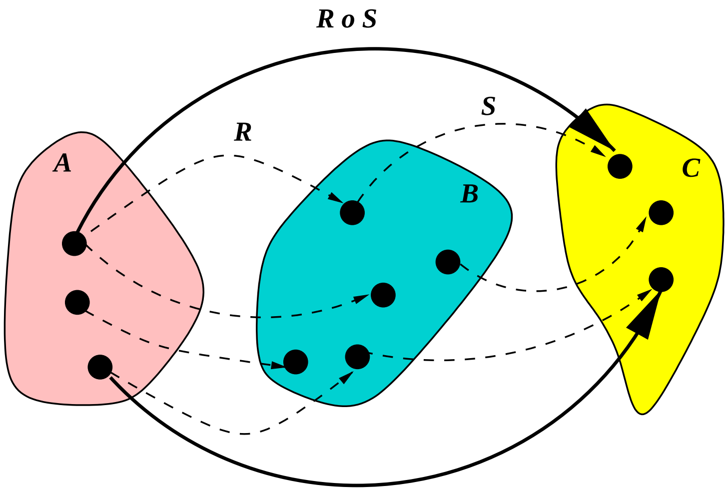
Two relations may be combined via an intermediate set to form another relation.

Definition 27 (Composition of Relations) Let $R \subseteq A \times B$ and $S \subseteq B \times C$. The *composition (Verknüpfung, Produkt)* of R and S is the following relation between A and C:

$$\begin{aligned}&R \circ S := \{\langle a, c\rangle : a \in \text{domain}(R) \wedge c \in \text{range}(S) \wedge \\ &\quad (\exists b : \langle a, b\rangle \in R \wedge \langle b, c\rangle \in S)\}.\end{aligned}$$

The clause $a \in A \wedge c \in C$ may be omitted in above definition; it is only used to make the set quantifier conform to the syntax given on page 66.

Example The composition of relations $R \subseteq A \times B$ and $S \subseteq B \times C$ can be visualized by the composition of arrows:



An arrow between an element of A and an element of C exists only if there exist two arrows that are correspondingly connected via some intermediate point in B.

Logic Evaluator We can define o(R, S) (i.e, $R \circ S$) as follows:

```
read relation;
> file 'set.txt' read.
> file 'relation.txt' read.
fun o(R: Relation, S: Relation) =
  let(A = domain(R), B = **(range(R), domain(S)), C = range(S):
    set(a in A, c in C:
      exists(b in B: and(in(tuple(a, b), R), in(tuple(b, c), S))),
      tuple(a, c)));
> function o/2.
fun R = set(x in nat(0, 5): true, tuple(x, *(2, x)));
> function R/0.
term R;
> {<0, 0>, <1, 2>, <2, 4>, <3, 6>, <4, 8>, <5, 10>}.
fun S = set(x in nat(0,2), y in nat(0,3): <=(x,*(2,y)), tuple(x,y));
> function S/0.
term S;
> {<0, 0>, <0, 1>, <0, 2>, <0, 3>, <1, 1>, <1, 2>, <1, 3>, <2, 1>,
<2, 2>, <2, 3>}.
term o(R, S);
> {<0, 0>, <0, 1>, <0, 2>, <0, 3>, <1, 1>, <1, 2>, <1, 3>}.
term o(S, R);
> {<0, 0>, <0, 2>, <0, 4>, <0, 6>, <1, 2>, <1, 4>, <1, 6>, <2, 2>,
<2, 4>, <2, 6>}.
```

```
term o(R, ^-1(R));
```

Proposition 30 (Relation Laws) For every A, B, C, D and every $R \subseteq A \times B$, $S \subseteq B \times C$, and $T \subseteq C \times D$, we have:

$$(R^{-1})^{-1} = R;$$
$$R \circ (S \circ T) = (R \circ S) \circ T;$$
$$(R \circ S)^{-1} = S^{-1} \circ R^{-1}.$$

We argue for the correctness of the first law.

Proof Take arbitrary A, B and $R \subseteq A \times B$. We prove

(1) $(R^{-1})^{-1} = R$.

Because of the definition of $=$, we have to show

(2) $\forall r : r \in (R^{-1})^{-1} \Leftrightarrow r \in R$.

Take arbitrary r.

- We prove $r \in (R^{-1})^{-1} \Rightarrow r \in R$. We assume

  (3) $r \in (R^{-1})^{-1}$

  and show $r \in R$.

  We have $R \subseteq A \times B$, therefore we know by definition of $^{-1}$ that $R^{-1} \subseteq B \times A$ and $(R^{-1})^{-1} \subseteq A \times B$. Thus we can find by (3) some $a \in A$ and $b \in B$ such that $r = \langle a, b \rangle$. By definition of $^{-1}$, we have $\langle b, a \rangle \in R^{-1}$. Again, by definition of $^{-1}$, we know that $\langle a, b \rangle \in R$, therefore $r \in R$.
- The proof that $r \in R \Rightarrow r \in (R^{-1})^{-1}$ proceeds in a similar way.

Logic Evaluator We may test whether our definitions of inversion and composition of relations conform to these laws:

```
read relation;
> file 'set.txt' read.
> file 'relation.txt' read.
fun R = set(x in nat(0, 5): true, tuple(x, *(2, x)));
> function R/0.
term R;
> {<0, 0>, <1, 2>, <2, 4>, <3, 6>, <4, 8>, <5, 10>}.
fun S = set(x in nat(0,2), y in nat(0,3): <=(x,*(2,y)), tuple(x,y));
> function S/0.
term S;
> {<0, 0>, <0, 1>, <0, 2>, <0, 3>, <1, 1>, <1, 2>, <1, 3>, <2, 1>,
<2, 2>, <2, 3>}.
formula =(^-1(o(R, S)), o(^-1(S), ^-1(R)));
> true.
fun T = set(x in nat(0, 5): true, tuple(x, *(3, x)));
> function T/0.

formula =(o(R, o(S, T)), o(R, o(S, T)));
```

3.4 Functions as Sets

Like relations provide a set theoretic substitute for predicates, we can also model functions by sets. In fact, functions are just considered as special relations.

Definition 28 (Function) A *function (Funktion)* is a binary relation where every element of the domain is in relation to at most one element of the range:

$$
\begin{aligned}
&f \text{ is a function} :\Leftrightarrow \\
&\quad f \text{ is a binary relation} \wedge \\
&\quad \forall x, y_0, y_1 : (\langle x, y_0\rangle \in f \ \wedge \langle x, y_1\rangle \in f) \Rightarrow y_0 = y_1.
\end{aligned}
$$

A *partial function (partielle Funktion/Abbildung)* from A to B is a function that is a relation between A and B:

$$
\begin{aligned}
&f : A \overset{\text{partial}}{\longrightarrow} B :\Leftrightarrow \\
&\quad f \text{ is a function} \wedge \\
&\quad f \subseteq A \times B.
\end{aligned}
$$

A *(total) function (totale Funktion/Abbildung)* from A to B is a partial function from A to B where every element of A is in relation to some element of B:

$$\begin{aligned}
&f : A \to B :\Leftrightarrow \\
&\quad f : A \stackrel{\text{partial}}{\longrightarrow} B \land \\
&\quad \forall x \in A : \exists y \in B : \langle x, y \rangle \in f.
\end{aligned}$$

Since a function is a relation, the notions *domain*, *range*, *inverse*, and *composition* also apply to functions (but see the comments on function inversion below).

While above definition only introduces unary functions, the case of n-ary functions (for every n) is reduced to the unary case by taking a Cartesian product as the function domain: e.g. a function

$$f : A \times B \to C$$

maps a tuple $\langle a, b \rangle$ with $a \in A$ and $b \in B$ to some value $c \in C$.

Example The following are functions:

- $\emptyset$;
- $\{\langle 0, 0\rangle, \langle 1, 0\rangle, \langle 2, 1\rangle, \langle 3, 2\rangle\}$;
- $\{\langle x, x^2\rangle : x \in \mathbb{N}\}$;
- $\{\langle x, x^2\rangle : x \in \mathbb{N} \land x \text{ is even}\} \cup \{\langle x, 0\rangle : x \in \mathbb{N} \land x \text{ is odd}\}$;
- $\{\langle x, y\rangle : x \in \mathbb{N} \land y \in \mathbb{N} \land x + y = 100\}$;
- $\{\langle\langle x, y\rangle, x + y\rangle : x \in \mathbb{N} \land y \in \mathbb{N}\}$.

The following are *not* functions:

- $\{\langle 0, 0\rangle, \langle 0, 1\rangle\}$;
- $\{\langle x, x^2\rangle : x \in \mathbb{N} \land x \text{ is prime}\} \cup \{\langle x, 0\rangle : x \in \mathbb{N} \land x \text{ is odd}\}$;
- $\{\langle x, y\rangle : x \in \mathbb{N} \land y \in \mathbb{N} \land x + y \leq 100\}$;

Logic Evaluator We define the corresponding predicates `Function`(f) (i.e., f is a function), `isPartialFunction`$(f,\ A,\ B)$ (i.e., $f : A \overset{\text{partial}}{\longrightarrow} B$) and `isFunction`$(f,\ A,\ B)$ (i.e., $f : A \to B$) as follows:

```
read relation;
> file 'set.txt' read.
> file 'relation.txt' read.
pred Function(f) <=>
  and(Relation(f),
    forall(t0 in f, t1 in f:
      let(x0 = .0(t0), y0 = .1(t0), x1 = .0(t1), y1 = .1(t1):
        implies(=(x0, x1), =(y0, y1)))));
> predicate Function/1.
pred isPartialFunction(f, A, B) <=>
  and(isRelation(f, A, B), Function(f));
> predicate isPartialFunction/3.
pred isFunction(f, A, B) <=>
  and(isPartialFunction(f, A, B),
    forall(x in A: exists(y in B: in(tuple(x, y), f))));
> predicate isFunction/3.
fun N(n) = set(x in nat(1, n): true, x);
> function N/1.
fun f = set(x in N(10): true, tuple(x, *(x, x)));
> function f/0.
term f;
> {<1, 1>, <2, 4>, <3, 9>, <4, 16>, <5, 25>, <6, 36>, <7, 49>, <8,
64>, <9, 81>, <10, 100>}.
formula isPartialFunction(f, N(15), N(100));
> true.
formula isPartialFunction(f, N(15), N(50));
> false.
formula isFunction(f, N(10), N(100));
> true.

formula isFunction(f, N(15), N(100));
```

The result of a function application is determined by "looking up" the set that represents the function.

Definition 29 (Function Application) Let f be a function and let $x \in \text{domain}(f)$. The *value (Wert)* of f at x is the y such that $\langle x, y\rangle \in f$:

$$f(x) := \textbf{such } y \in \text{range}(f) : \langle x, y\rangle \in f.$$

Since f is a function, above statement indeed defines a unique $y \in \text{range}(f)$ for every $x \in \text{domain}(f)$.

Please note that this definition introduces a binary function '()' ("apply") that takes two arguments f and x. This syntax is convenient to hide the distinction between set-theoretic functions and function constants. We are now able to quantify over functions as in

$$\forall y : \exists f : \forall x : f(x) = y$$

with the understanding that f denotes a particular set and above formula has to be interpreted as

$$\forall y : \exists f : \forall x : \langle x, y \rangle \in f.$$

Instead of writing $f(\langle x_0, \ldots, x_{n-1} \rangle)$, we usually simply write $f(x_0, \ldots, x_{n-1})$.

Logic Evaluator We can define the binary function `apply`(f, x) (i.e., $f(x)$) as follows:

```
read relation;
> file 'set.txt' read.
> file 'relation.txt' read.
fun apply(f: Function, x) =
  such(y in range(f): in(tuple(x, y), f), y);
> function apply/2.
fun f = set(x in nat(1, 10): true, tuple(x, *(x, x)));
> function f/0.
term apply(f, 2);
> 4.
term apply(f, apply(f, 2));
> 16.
term apply(f, 11);
> ERROR: no such value.
```

```
term let(x = apply(f, 3): +(x, apply(f, x)));
```

Notation Functions are usually defined in one of the following formats (where x is a variable and T is a term):

- $$\begin{aligned} f : A &\to B \\ x &\mapsto T \end{aligned}$$
- $$\begin{aligned} &f : A \to B \\ &f(x) := T \end{aligned}$$
- $$\begin{aligned} &f : A \to B \\ &f := \lambda x.T \end{aligned}$$
- $$f(x : A) : B = T$$

All of this is to be interpreted as

$$f : A \to B \ \wedge \ \forall x \in A : f(x) = T$$

or, equivalently, as

$$f = \{\langle x, T\rangle : x \in A\} \ \wedge \ \forall x \in A : T \in B.$$

If the function domain is a Cartesian product, we usually write

$$\begin{aligned} &f : A_0 \times \ldots \times A_{n-1} \to B \\ &f(x_0, \ldots, x_{n-1}) := T \end{aligned}$$

to denote

$$\begin{aligned} &f : A_0 \times \ldots \times A_{n-1} \to B \\ &f(t) := \mathbf{let}\ x_0 = t_0, \ldots, x_{n-1} = t_{n-1} : T. \end{aligned}$$

Example The statement

$$\begin{aligned} &\mathrm{div} : \mathbb{N} \times \mathbb{N} \to \mathbb{Q} \\ &\mathrm{div}(x, y) := x/y; \end{aligned}$$

should be read as

$$\begin{aligned} &\mathrm{div} := \{\langle\langle x, y\rangle, x/y\rangle : x \in \mathbb{N}, y \in \mathbb{N}\} \wedge \\ &\forall x \in \mathbb{N}, y \in \mathbb{N} : \mathrm{div}(x, y) \in \mathbb{Q}. \end{aligned}$$

Function Inversion While the inverse of a function is well defined, it is not necessarily a function.

Example Take function $f = \{\langle 0,0\rangle, \langle 1,0\rangle, \langle 2,1\rangle\}$. Then

$$f^{-1} = \{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,2\rangle\}$$

is a binary relation but not a function, because it contains $\langle 0,0\rangle$ and $\langle 0,1\rangle$.

We will learn in Proposition 70 on page 170 under which condition the inverse of a function is indeed a function.

On the other hand, we have the following result.

Proposition 32 (Function Composition) The composition of two functions is also a function, i.e., for all A, B, and C, and all $f : A \to B$ and $g : B \to C$, we have:

$$f \circ g : A \to C.$$

Proof Take arbitrary A, B, C, $f : A \to B$, and $g : B \to C$. We have to prove $f \circ g : A \to C$, i.e., by definition of $\circ$, that

(1) $(f \circ g) : A \overset{\text{partial}}{\longrightarrow} C$;
(2) $\forall x \in A : \exists y \in C : \langle x, y\rangle \in (f \circ g)$.

- We prove (1), i.e., by definition of $\overset{\text{partial}}{\longrightarrow}$, that

  (3) $(f \circ g) \subseteq A \times C$;
  (4) $\forall x, y_0, y_1 : (\langle x, y_0\rangle \in (f \circ g) \ \wedge \langle x, y_1\rangle \in (f \circ g)) \Rightarrow y_0 = y_1$.

  We know (3) from the definition of $\circ$; we still have to show (4). Take arbitrary x, y_0, y_1 and assume

  (5) $\langle x, y_0\rangle \in (f \circ g)$;
  (6) $\langle x, y_1\rangle \in (f \circ g)$.

We have to show $y_0 = y_1$.

From (5), (6), and the definition of $\circ$, we know $y_0 \in C$, $y_1 \in C$, and

(7) $\exists b \in B : \langle x, b\rangle \in f \wedge \langle b, y_0\rangle \in g$;
(8) $\exists b \in B : \langle x, b\rangle \in f \wedge \langle b, y_1\rangle \in g$.

By (7), we have some $b_0 \in B$ such that $\langle x, b_0\rangle \in f \wedge \langle b_0, y_0\rangle \in g$; by (8), we have some $b_1 \in B$ such that $\langle x, b_1\rangle \in f \wedge \langle b_1, y_0\rangle \in g$. From $\langle x, b_0\rangle \in f$, $\langle x, b_1\rangle \in f$ and the fact that f is a function, we know that $b_0 = b_1$. From this and from $\langle b_0, y_0\rangle \in g$, $\langle b_1, y_1\rangle \in g$, and the fact that g is a function, we know that $y_0 = y_1$.

- We prove (2). Take arbitrary (3) $x \in A$. We have to show

  (4) $\exists y \in C : \langle x, y\rangle \in (f \circ g)$.

  From (3) and $f : A \to B$, we know (5) $\langle x, f(x)\rangle \in f$ and (6) $f(x) \in B$. From (6) and $g : B \to C$, we know (7) $\langle f(x), g(f(x))\rangle \in g$ and (8) $g(f(x)) \in C$.

  To show (4), we take (9) $y := g(f(x))$ and show

  (10) $y \in C$;
  (11) $\langle x, y\rangle \in (f \circ g)$.

  We know (10) from (8) and (9). To show (11), we have to show, by definition of $\circ$, that

  (12) $x \in A \wedge y \in C$;
  (13) $\exists b : \langle x, b\rangle \in f \wedge \langle b, y\rangle \in g$.

  We know (12) from (3), (8), and (9). To show (13), we take (14) $b := f(x)$ and have to show:

  (15) $\langle x, b\rangle \in f$;
  (16) $\langle b, y\rangle \in g$.

  We know (15) from (5) and (14). We know (16) from (7), (9), and (14).

As a consequence, we have the following "direct" characterization of function composition.

Proposition 34 (Function Composition) For all A, B, and C, and all $f : A \to B$ and $g : B \to C$, we have:

$$\forall x \in A : (f \circ g)(x) = g(f(x)).$$

We then have the following result:

Proposition 35 (Associativity of Function Composition) The composition of two functions is associative, i.e.,

$$\begin{array}{l} \forall A, B, C, D, f : A \to B, g : B \to C, h : C \to D : \\ \quad f \circ (g \circ h) = (f \circ g) \circ h \end{array}$$

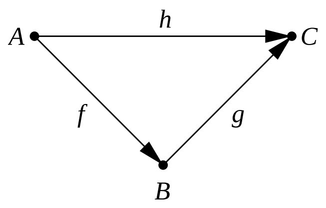
Proof Take arbitrary A, B, C, D, $f : A \to B$, $g : B \to C$, and $h : C \to D$. We have to show

$$\forall x \in A : (f \circ (g \circ h))(x) = ((f \circ g) \circ h)(x).$$

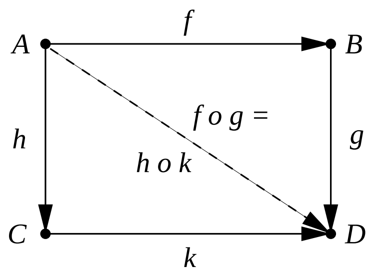
Take arbitrary $x \in A$. Then we have by Proposition 32

$$\begin{array}{rl} (f \circ (g \circ h))(x) & = \\ (g \circ h)(f(x)) & = \\ h(g(f(x))) & = \\ h((f \circ g)(x)) & = \\ ((f \circ g) \circ h)(x). & \end{array}$$

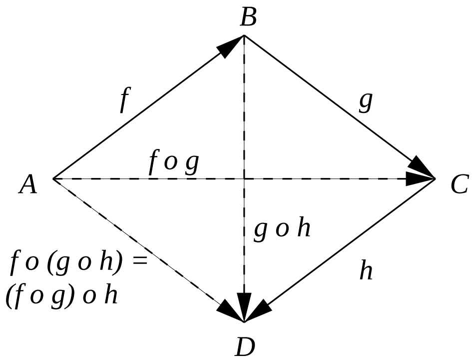
Commutative Diagrams Relationships between functions and composite functions are often stated by a diagram where the nodes represent sets and an arrow f between nodes A and B indicates that f is a function from A to B. A path from one node to another represents the composition of all the functions represented by the individual arcs on the graph. The diagram is said to *commute* if any two paths with same initial node and same terminal node represent the same function, e.g., the diagram



asserts that $h = f \circ g$ for $f : A \to B$ and $g : B \to C$ and $h : A \to C$. The diagram



says that $f \circ g = h \circ k$, for $f : A \to B$, $g : B \to D$, $h : A \to C$, $k : C \to D$. Proposition 35 is asserted by the following diagram:



3.5 Sequences and Matrices

Tuples are intended to model sequences of objects of a fixed length; for every n-tuple, there are n unary functions to select each component of the tuple but there is no binary function that takes a tuple t and an arbitrary index i and returns the i-th component of t.

While we could extend tuples accordingly, we have a more convenient concept to model sequences of objects of arbitrary (possibly infinite) length:

Definition 30 (Sequence) A *sequence (Folge)* of length n over S is a function from $\mathbb{N}_n$, the set of the first n natural numbers, to S:

$$s \text{ is a sequence of length } n \text{ over } S :\Leftrightarrow s : \mathbb{N}_n \to S.$$

The *length* of a finite sequence s is that n such that s maps $\mathbb{N}_n$ to some S:

$$\begin{array}{l}\text{length}(s) := \\ \quad \textbf{such } n \in \mathbb{N}\text{: } \exists S : s \text{ is a sequence of length } n \text{ over } S.\end{array}$$

A *finite sequence* over S is a sequence of length n over S, for some n:

$$\begin{array}{l}s \text{ is a finite sequence over } S :\Leftrightarrow \\ \quad \exists n \in N : s \text{ is a sequence of length } n \text{ over } S.\end{array}$$

An *infinite sequence* over S is a function from $\mathbb{N}$, the natural numbers, to S:

$$s \text{ is an infinite sequence over } S :\Leftrightarrow s : \mathbb{N} \to S.$$

Sequences appear under various names:

- tables,
- arrays,
- vectors,
- lists.

The i-th component $s(i)$ of a sequence s is also denoted as

- s_i;
- $s[i]$;
- $s.i$.

Example Take the sequence $S := \{\langle 0,2\rangle, \langle 1,3\rangle, \langle 2,4\rangle, \langle 3,4\rangle, \langle 4,5\rangle\}$ typically written as

$$[2,3,4,4,5].$$

The length of S is 5 and we have

$$S_0 = 2, S_1 = 3, S_2 = 4, S_3 = 4, S_4 = 5.$$

We also have

$$\forall i \in \mathbb{N}_5 : 0 < S_i < 10.$$

Take the infinite sequence $T := \{\langle i, i^2\rangle : i \in \mathbb{N}\}$, i.e.,

$$[0,1,4,9,16,25,\ldots].$$

We have

$$\forall i \in \mathbb{N} : S_i = i^2.$$

The same concept may be extended to two dimensions.

Definition 31 (Matrix) A *matrix (Matrix)* with m rows and n columns over S is a function from $\mathbb{N}_m \times \mathbb{N}_n$ to S:

$$M \text{ is a } m \times n\text{-matrix over } S :\Leftrightarrow M : \mathbb{N}_m \times \mathbb{N}_n \to S.$$

A matrix component $M(\langle i,j\rangle)$ is often denoted as:

- $M(i,j)$;
- $M[i,j]$;
- $M_{i,j}$.

Example Take the matrix

$$\begin{aligned}&M := \{\\&\quad\langle\langle 0,0\rangle,1\rangle,\langle\langle 0,1\rangle,2\rangle,\langle\langle 0,2\rangle,3\rangle,\\&\quad\langle\langle 1,0\rangle,4\rangle,\langle\langle 1,1\rangle,5\rangle,\langle\langle 1,2\rangle,6\rangle,\\&\quad\langle\langle 2,0\rangle,7\rangle,\langle\langle 2,1\rangle,8\rangle,\langle\langle 2,2\rangle,9\rangle\}\end{aligned}$$

which is typically depicted as:

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$$

We have

$$\forall i \in \mathbb{N}_3, j \in \mathbb{N}_3 : M_{i,j} = 3i + j + 1.$$

The matrix concept can be extended to an arbitrary number of dimensions in an alogous way.

When constructing sequences or matrices by iterative updates, the following notations come handy:

Definition 32 (Sequence Updates) The n-element *a-sequence* is the sequence of length n that maps every index to a:

$$\mathrm{new(n, a)} := \{\langle i, a\rangle : i \in \mathbb{N}_n\}.$$

The sequence constructed from s by setting i to a is the same as s with the exception that i is mapped to a:

$$s[i \mapsto a] := \{\langle i, a\rangle\} \cup \{t \in s : t_0 \neq i\}.$$

Example Take the sequence

$$A := \mathrm{new}(5,0)[3 \mapsto 1][2 \mapsto 5][4 \mapsto 1][2 \mapsto 3].$$

Then we have length$(A) = 5$ and

$$A_0 = 0, A_1 = 0, A_2 = 3; A_3 = 1; A_4 = 2.$$

Logic Evaluator We can only represent finite sequences; we define the sequence functions `len`(s) (based on the function `#`(s) that gives the number of elements in a set s), `new`(n, a) and `[]`(s, i, a) (i.e, $s[i \mapsto a]$).

```
read relation;
> file 'set.txt' read.
> file 'relation.txt' read.
pred Sequence(f) <=>
  and(Function(f),
    let(n = #(f):
      forall(x in f: let(i = .0(x): and(Nat(i), <=(+(i, 1), n))))));
> predicate Sequence/1.
fun slength(f: Sequence) = #(f);
> function slength/1.
fun new(n: Nat, a) = set(i in nat(0, -(n, 1)): true, tuple(i, a));
> function new/2.
fun [](s: Sequence, i: Nat, a) =
  join(tuple(i, a), set(t in s: not(=(.0(t), i)), t));
> function []/3.
fun S = []([]([]([](new(5, 0), 3, 1), 2, 5), 4, 1), 2, 3);
> function S/0.
term slength(S);
> 5.
term apply(S, 0);
> 0.
term apply(S, 2);
> 3.
term apply(S, 4);
> 1.
```

```
term apply([](S, 0, 7), 0);
```

Chapter 4

Numbers

Much of our knowledge about numbers consists of rules (heuristics, methods, algorithms) that tell us how to compute the values of arithmetic expressions ("what is (27*56)/13?") or solve equations for some unknowns ("for which x does $x^2+x = 15.75$ hold?"). This is what we have been trained in school and practice in daily life. However when being asked "What exactly *is* a (natural, integer, rational, real, complex) number?" or "*Why* is $x + y = y + x$?", our school wisdom does usually not suffice to give a satisfactory answer.

The goal of this chapter is to provide some deeper *insight* into the hierarchy of number domains, i.e., to look below the surface of arithmetic rules and to shed some light on the internal structure of the domains. Much of what we have been taught to believe in school can here be easily justified; it is not necessary to learn a lot of rules by heart if one is able to derive them on demand from a few basic principles.

We will focus in this chapter on a *constructive* approach that explicitly defines a domain by concepts that have an algorithmic interpretation (with set theory as the fundamental basis). The constructions themselves are interesting from the computer science point of view, because they illustrate how a particular hierarchy of datatypes is systematically developed. Another approach that demonstrates some more abstract mathematical techniques will be discussed in Chapter 7.

4.1 The Natural Numbers

Intuitively, the natural numbers are the numbers of *counting* distinct objects: there may be no object, one object, two objects, ... While the domain of natural numbers looks very simple and familiar, it is surprisingly powerful; e.g. every computer program can be encoded as a natural number. However, only *discrete* realities can be described by natural numbers; there are also *continuous* realities where the distinction between objects is so “fine” that they cannot be modelled by this domain.

Peano Arithmetic We may consider the natural numbers as an elementary domain by a *theory* (see p. 281) that is due to the mathematician Peano.

Axiom 3 (Peano Arithmetic) The theory of natural numbers has an object constant 0 (*zero*) and a unary function constant $'$, the *successor (Nachfolger)*, that satisfy the following axioms.

1. 0 is not the successor of any natural number:

   $$\forall x : x' \neq 0.$$

2. Different natural numbers have different successors:

   $$\forall x, y : x' = y' \Rightarrow x = y.$$

3. For every formula F, we have an axiom that states: F holds for every natural number, if F holds for 0 and with every number also for its successor, i.e.,

   $$(F[x \leftarrow 0] \wedge (\forall x : F \Rightarrow F[x \leftarrow x+1])) \Rightarrow \forall x : F.$$

   We call these axioms the *induction axioms (Induktionsaxiome)*.

The Peano axioms essentially state that the natural numbers are a single infinite chain

$$0 \to 0' \to 0'' \to 0''' \to \ldots$$

We will see later how all operations on the natural numbers can be defined just on the basis of 0 and $'$.

Construction from Sets It is not necessary to consider the natural numbers as elementary objects; they can be *defined* within the framework of set theory that provides a more fundamental kind of objects.

Definition 33 (Natural Numbers from Set) We define a constant 0 and a unary function $'$ as follows:

$$0 := \emptyset;$$
$$x' := x \cup \{x\}.$$

The first two axioms of Peano arithmetic are consequences of this definition.

Proof We sketch the proof of the first two laws of Peano.

- We prove $\forall x : x' \neq 0$. Take arbitrary x. By definition of 0 and $'$, we have to prove

  $$x \cup \{x\} \neq \emptyset$$

  which is true because $x \in (x \cup \{x\})$ but $x \neq \emptyset$.

- We prove $\forall x, y : x' = y' \Rightarrow x = y$. Take arbitrary x and y. We assume

  $$(1)\ x' = y'$$

  and have to show $x = y$. We assume (2) $x \neq y$ and show a contradiction. From (1), we know by the definition of $'$

  $$(3)\ x \cup \{x\} = y \cup \{y\}.$$

  By the definition of $\cup$ and set enumeration, we know (4) $x \in x \cup \{x\}$ and (5) $y \in y \cup \{y\}$. From (3), (4), and the definition of $=$, we have (6) $x \in y \cup \{y\}$ which implies with (2)

  $$(7)\ x \in y.$$

  Likewise, we have from (3), (5), and the definition of $=$ that (8) $y \in x \cup \{x\}$ which implies with (2)

  $$(9)\ y \in x.$$

  The conjunction of (7) and (9) is in contradiction to the set-theoretic axiom of *regularity (Regularität)* that prohibits such "cycles" (in general any infinitely descending chains) with respect to $\in$.

The construction of natural numbers proceeds with the definition of a set $\mathbb{N}$ (which we do not give) such that the following proposition holds.

Proposition 38 (Natural Numbers) $\mathbb{N}$, the set of *natural numbers (natürliche Zahlen)*, is the smallest set that satisfies the following properties:

$$\begin{array}{l} 0 \in \mathbb{N}; \\ \forall x \in \mathbb{N} : x' \in \mathbb{N}; \\ \text{for every formula } F : \\ \quad (F(0) \wedge \forall x \in \mathbb{N} : F(x) \Rightarrow F(x+1)) \Rightarrow \forall x \in \mathbb{N} : F(x). \end{array}$$

Thus also the third law of Peano holds in the set-based construction of natural numbers. Sometimes the following notations will be used.

Definition 34 (Natural Number Subsets)

$$\begin{array}{rcl} \mathbb{N}_{>0} & := & \{x \in \mathbb{N} : x > 0\}; \\ \mathbb{N}_n & := & \{x \in \mathbb{N} : x < n\}. \end{array}$$

Arithmetic Operations Independently of how we have introduced the natural numbers, we can define the basic arithmetic operations with the help of the functions 0 and $'$ and the following auxiliary operation.

Definition 35 (Predecessor) The *predecessor (Vorgänger)* of a natural number x is the natural number whose successor is x:

$$x^- := \textbf{such } y : x = y'.$$

Because of the second Peano law, the predecessor is uniquely defined (if it exists). However, because of the first Peano law, there is no x such that $x' = 0$ and thus 0^- is undefined (and we must not assume $0^{-\prime} = 0$).

We now introduce the arithmetic operations by A.5. In the rest of this section, all variables denote natural numbers.

Definition 36 (Natural Numbers Operations) We define the following functions and predicates (for all recursive definitions, a corresponding termination function is $r(x, y) := y$).

Constants

$$1 := 0',\ 2 := 1';$$

Addition

$$x + y := \textbf{if } y = 0 \textbf{ then } x \textbf{ else } (x + y^-)'$$

Multiplication

$$x * y := \textbf{if } y = 0 \textbf{ then } 0 \textbf{ else } x + (x * y^-)$$

Total Order

$$\begin{array}{l} x \leq y :\Leftrightarrow \\ \quad \textbf{if } x = 0 \textbf{ then } \mathrm{T} \\ \quad \textbf{else if } y = 0 \textbf{ then } \mathrm{F} \\ \quad \textbf{else } x^- \leq y^- \end{array}$$

The recursive definitions are constructive in the sense that they can be executed by repeated "unfolding" of the definition.

Example We compute (1+2):

$$\begin{array}{rcl} 1 + 2 & = & \text{(definitions of 1 and 2)} \\ 0' + 0'' = & = & \text{(definition of } +) \\ (\textbf{if } 0'' = 0 \textbf{ then } 0' \textbf{ else } (0' + 0''^-)') & = & \text{(second Peano law)} \\ (0' + 0''^-)' & = & \text{(definition of } ^-) \\ (0' + 0')' & = & \text{(definition of } +) \\ (\textbf{if } 0' = 0 \textbf{ then } 0' \textbf{ else } (0' + 0'^-)')' & = & \text{(second Peano law)} \\ (0' + 0'^-)'' & = & \text{(definition of } ^-) \\ (0' + 0)'' & = & \text{(definition of } +) \\ (\textbf{if } 0 = 0 \textbf{ then } 0' \textbf{ else } (0' + 0^-)')'' & = & \text{(reflexivity of } =) \\ 0'''. & & \end{array}$$

Natural Number Laws Immediate consequences of these definitions are the following laws.

Proposition 39 (Natural Numbers Operations) For all natural numbers x and y, we have:

Addition

$$\begin{aligned} x + 0 &= x, \\ x + y' &= (x + y)'; \end{aligned}$$

Multiplication

$$\begin{aligned} x * 0 &= 0, \\ x * y' &= x + (x * y); \end{aligned}$$

Total Order

$$\begin{aligned} 0 \leq x &\Leftrightarrow \mathrm{T}, \\ x \leq 0 &\Leftrightarrow x = 0, \\ x' \leq y' &\Leftrightarrow x \leq y. \end{aligned}$$

Furthermore, the following well known laws hold.

Proposition 40 (Natural Number Laws) For all natural numbers x, y, z, we have:

$$\begin{aligned} x + 0 &= x, \\ x * 1 &= x, \\ x + y &= y + x, \\ x * y &= y * x, \\ x + (y + z) &= (x + y) + z, \\ x * (y * z) &= (x * y) * z, \\ x * (y + z) &= (x * y) + (x * z), \\ x &\leq x, \\ (x \leq y \wedge y \leq x) &\Rightarrow x = y, \\ (x \leq y \wedge y \leq z) &\Rightarrow x \leq z. \end{aligned}$$

We will use the following abbreviations for every domain that provides a binary predicate $\leq$.

Definition 37 (Order Predicates)

$$\begin{aligned} &x < y :\Leftrightarrow x \leq y \wedge x \neq y; \\ &x > y :\Leftrightarrow x \not\leq y; \\ &x \geq y :\Leftrightarrow x \not< y. \end{aligned}$$

We often write $a \leq x < b$ to denote $x \leq a \wedge x < b$ and similar for all other combinations of the order predicates.

Logic Evaluator Although there is a builtin implementation of the natural numbers, we are going to simulate the construction on the base of 0 and and $'$; we use the letter `N` to distinguish the natural number function and predicate constants from those that are going to be introduced in the following sections. These and the subsequent definitions are collected in Appendix C.3 (file `natural.txt`).

```
pred N(x) <=> Nat(x);
> predicate N/1.

fun N0 = 0;
> function N0/0.
fun '(x: N) = +(x, 1);
> function '/1.
fun ^-(x: N) = such(n in nat(0, x): =(x, '(n)), n);
> function ^-/1.

fun N1 = '(N0);
> function N1/0.
fun N2 = '(N1);
> function N2/0.

fun +N(x: N, y: N) recursive y =
  if(=(y, N0), x, '(+N(x, ^-(y))));
> function +N/2.
fun *N(x: N, y: N) recursive y =
  if(=(y, N0), N0, +N(x, *N(x, ^-(y))));
> function *N/2.
pred <=N(x: N, y: N) recursive y <=>
  if(=(x, N0), true, if(=(y, N0), false, <=N(^-(x), ^-(y))));
> predicate <=N/2.

term ^-(0);
> ERROR: no such value.
term +N(5, 2);
> 7.
term *N(5, 3);
> 15.

formula <=(5, 3);
```

More Operations Since we have addition, we would also like to have the inverse operation.

Definition 38 (Difference) z is a difference of x and y if $x = y + z$.

$$x - y := \textbf{such } z : x = z + y.$$

Please note that a difference between two natural numbers does not always exist, e.g. $1 - 2$ is undefined because there is no z with $1 = z + 2$ (and we must not assume $1 = (1 - 2) + 2$). However, we have the following result.

Proposition 41 (Difference) If a difference exists, it is unique:

$$\forall x, y, z_0, z_1 : (x = z_0 + y \land x = z_1 + y) \Rightarrow z_0 = z_1.$$

If $x \geq y$, the difference of x and y is defined:

$$\forall x, y : x \geq y \Rightarrow x = (x - y) + y.$$

At the moment, we state these propositions without proofs. In Chapter 5, we will discuss in detail how to prove properties about natural numbers.

While the natural numbers do not have an operation for exact division, they provide the following pair of functions.

Definition 39 (Quotient and Remainder) The *quotient (Quotient)* and *remainder (Rest)* of two natural numbers are defined as follows:

$$\begin{array}{rcl} x \text{ div } y & := & \textbf{such } q : \exists r : r < y \land x = (q * y) + r; \\ x \text{ mod } y & := & \textbf{such } r : \exists q : r < y \land x = (q * y) + r. \end{array}$$

By above definition $(x \text{ div } 0)$ and $(x \text{ mod } 0)$ are undefined for every x. On the other side, we have the following positive results.

Proposition 42 (Quotient and Remainder) If quotient respectively remainder of two numbers exist, they are unique.

$$\begin{array}{l} \forall x, y, q_0, q_1, r_0 < y, r_1 < y : \\ \quad (x = (q_0 * y) + r_0 \land x = (q_1 * y) + r_1) \Rightarrow (q_0 = q_1 \land r_0 = r_1). \end{array}$$

If the divisor is not null, quotient and remainder exist:

$$\forall x, y \neq 0 : (\exists q, r : r < y \wedge x = (q * y) + r).$$

We thus have the following relationship between quotient and remainder:

$$\forall x, y \neq 0 : x = (x \text{ div } y) * y + (x \text{ mod } y).$$

We define exponentiation as shown below.

Definition 40 (Exponentiation) We define recursively with termination function $r(x, n) := n$:

$$\cdot^{\cdot} : \mathbb{N} \times \mathbb{N} \to \mathbb{N},$$
$$x^n := \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } x * x^{n^-}.$$

A corresponding construction can be applied in every domain D with a constant 1 and a binary function $*$ such as the number domains that will be introduced in the following sections; we will then assume the existence of a corresponding function $\cdot^{\cdot} : D \times \mathbb{N} \to D$.

Logic Evaluator Difference, quotient, remainder and exponentiation are defined as shown below.

```
option silent = true;read natural;

fun -N(x: N, y: N) = such(z in nat(0, x): =(x, +N(z, y)), z);
fun divN(x: N, y: N) =
  such(q in nat(0, x), r in nat(0, ^-(y)):
    =(x, +N(*N(q, y), r)), q);
fun modN(x: N, y: N) =
  such(q in nat(0, x), r in nat(0, ^-(y)):
    =(x, +N(*N(q, y), r)), r);
fun ^N(x: N, n: N) recursive n =
  if(=(n, N0), N1, *N(x, ^N(x, ^-(n))));

term -N(5, 3);
> 2.
term -N(3, 5);
> ERROR: no such value.
term divN(5, 3);
> 1.

term modN(5, 3);
```

Other important notions on natural numbers can be introduced as follows.

Definition 41 (Natural Number Operations)

- x *divides* y if $x * z = y$ for some z:

$$x|y :\Leftrightarrow \exists z : x * z = y.$$

- The *greatest common divisor (größter gemeinsamer Teiler)* of x and y is the largest number that divides both x and y:

$$\mathrm{gcd}(x, y) := \textbf{such } z : z|x \wedge z|y \wedge (\forall w : (w|x \wedge w|y) \Rightarrow w \leq z).$$

- The *least common multiple (kleinstes gemeinsames Vielfaches)* of x and y is the smallest number that both x and y divide:

$$\mathrm{lcm}(x, y) := \textbf{such } z : x|z \wedge y|z \wedge (\forall w : (x|w \wedge y|w) \Rightarrow z \leq w).$$

- Two numbers are *relatively prime (relativ prim)* if their greatest common divisor is 1:

$$x \text{ and } y \text{ are relatively prime } :\Leftrightarrow \mathrm{gcd}(x, y) = 1.$$

- A number greater than 1 is *prime (prim)* if its only divisors are are 1 and itself:

$$x \text{ is prime } :\Leftrightarrow x > 1 \wedge (\forall y : y|x \Rightarrow (y = 1 \vee y = x)).$$

The definition of prime numbers is motivated by the fact that every positive natural number n has a unique prime number factorization (which will be stated formally in Proposition 60 on page 134).

Logic Evaluator Above notions are defined as shown below (to speed up computation, we we revert to builtin natural number arithmetic using the file `natural0.txt`.

```
option silent = true;
read natural0;

pred divides(x, y) <=> exists(z in nat(N0, y): =(*N(x, z), y));
fun gcd(x, y) =
  let(m = if(=(x, N0), y, x):
    such(z in nat(N0, m):
      and(divides(z, x), divides(z, y),
        forall(w in nat(+N(z, N1), m):
          or(not(divides(w, x)), not(divides(w, y))))),
      z));
fun lcm(x, y) = such(z in nat(N1, *N(x, y)):
  and(divides(x, z), divides(y, z),
    forall(w in nat(x, -(z, N1)):
      or(not(divides(x, w)), not(divides(y, w))))),
  z);
pred relprime(x, y) <=> =(gcd(x, y), N1);
pred isprime(x) <=>
  and(not(<=N(x, N1)),
    forall(y in nat(N0, x):
      implies(divides(y, x), or(=(y, N1), =(y, x)))));
formula divides(4, 12);
> true.

term gcd(6, 9);
> 3.
term lcm(10, 6);
> 30.
formula relprime(6, 4);
> false.

formula isprime(17);
```

4.2 The Integer Numbers

As we have seen in the previous section, the domain of natural numbers is "open" in the sense that, for certain arguments, some important operations have no value in this domain. For instance, there does not exist a natural number x with

$$0 = x + 1,$$

i.e., $(0 - 1)$ is undefined.

We therefore introduce a domain $\mathbb{Z}$ of *integer numbers* such that

1. $\mathbb{N}$ can be "embedded" into $\mathbb{Z}$, and

2. for all integers a and b there is an integer x with $a = x + b$ (and consequently $a - b$ is defined).

We are going to give a corresponding set-theoretic construction of $\mathbb{Z}$ that meets these requirements. For the purpose of this endeavor (which is based on the construction of the naturals given in the previous section), we will denote by a constant $c_{\mathbb{N}}$ the function or predicate c in $\mathbb{N}$.

Definition 42 (Integer Numbers) The set $\mathbb{Z}$ of *integer numbers (ganze Zahlen)* is defined as the following subset of $\mathbb{N} \times \mathbb{N}$:

$$
\begin{aligned}
&\mathbb{Z} := \mathbb{Z}_{\geq 0} \cup \mathbb{Z}_{<0};\\
&\mathbb{Z}_{\geq 0} := \{\langle x, 0_{\mathbb{N}}\rangle : x \in \mathbb{N}\};\\
&\mathbb{Z}_{<0} := \{\langle 0_{\mathbb{N}}, x\rangle : x \in \mathbb{N}\backslash\{0_{\mathbb{N}}\}\};\\
&\mathbb{Z}_{>0} := \mathbb{Z}_{\geq 0}\backslash\{\langle 0_{\mathbb{N}}, 0_{\mathbb{N}}\rangle\}.
\end{aligned}
$$

We define a corresponding constructor function:

$$
\begin{aligned}
&\mathrm{I} : \mathbb{N} \times \mathbb{N} \to \mathbb{Z}\\
&\mathrm{I}(x, y) := \textbf{if } x \geq_{\mathbb{N}} y \textbf{ then } \langle x -_{\mathbb{N}} y, 0_{\mathbb{N}}\rangle \textbf{ else } \langle 0_{\mathbb{N}}, y -_{\mathbb{N}} x\rangle;
\end{aligned}
$$

The intuition of this construction is that a tuple $\langle a, 0_{\mathbb{N}}\rangle$ represents a non-negative integer a, while a tuple $\langle 0_{\mathbb{N}}, b\rangle$ represents a negative integer $-b$. This particular representation has the advantage that the definition of the arithmetic operations on integers becomes relatively easy.

Definition 43 (Integer Operations)

Constants

$$0 := \mathrm{I}(0_{\mathbb{N}}, 0_{\mathbb{N}});\ 1 := \mathrm{I}(1_{\mathbb{N}}, 0_{\mathbb{N}});\ 2 := \mathrm{I}(2_{\mathbb{N}}, 0_{\mathbb{N}}).$$

Basic Arithmetic

$$
\begin{aligned}
&x + y := \mathrm{I}(x_0 +_{\mathbb{N}} y_0, x_1 +_{\mathbb{N}} y_1);\\
&x * y := \mathrm{I}((x_0 *_{\mathbb{N}} y_0) +_{\mathbb{N}} (x_1 *_{\mathbb{N}} y_1), (x_0 *_{\mathbb{N}} y_1) +_{\mathbb{N}} (x_1 *_{\mathbb{N}} y_0));\\
&-x := \langle x_1, x_0\rangle;\\
&x - y := x + (-y).
\end{aligned}
$$

Total Order

$$x \leq y :\Leftrightarrow (x_0 + y_1 <_{\mathbb{N}} y_0 + x_1).$$

More Arithmetic

$$\begin{aligned}
&|x| := \textbf{if } 0 \leq x \textbf{ then } x \textbf{ else } -x;\\
&\mathrm{sign}(x) := \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else if } 0 \leq x \textbf{ then } 1 \textbf{ else } -1;\\
&x \text{ div } y := \textbf{ such } q : \exists r :\\
&\quad |r| < |y| \land x = q * y + r \land (\mathrm{sign}(r) = 0 \lor \mathrm{sign}(r) = \mathrm{sign}(y));\\
&x \text{ mod } y := \textbf{ such } r : \exists q :\\
&\quad |r| < |y| \land x = q * y + r \land (\mathrm{sign}(r) = 0 \lor \mathrm{sign}(r) = \mathrm{sign}(y)).
\end{aligned}$$

Logic Evaluator The implementation of basic arithmetic correspond to the mathematical definitions (see Appendix C.4, file `integer.txt`).

```
option silent = true;
read integer;

pred Z(x) <=> and(Tuple(x), =(2, length(x)), N(.0(x)), N(.1(x)));
fun I(x: N, y: N) = if(<=N(y, x), tuple(-N(x, y), N0), tuple(N0,
-N(y, x)));

fun Z0 = I(N0, N0);
fun Z1 = I(N1, N0);
fun Z2 = I(N2, N0);

fun +Z(x: Z, y: Z) = I(+N(.0(x), .0(y)), +N(.1(x), .1(y)));
fun *Z(x: Z, y: Z) =
  I(+N(*N(.0(x), .0(y)), *N(.1(x), .1(y))),
    +N(*N(.0(x), .1(y)), *N(.1(x), .0(y))));
fun -Z(x: Z) = tuple(.1(x), .0(x));
fun -Z(x: Z, y: Z) = +Z(x, -Z(y));
pred <=Z(x: Z, y: Z) <=> =(.1(-Z(y, x)), N0);

term Z2;
> <2, 0>.
term -Z(Z1, Z2);
> <0, 1>.
term +Z(Z1, Z2);
> <3, 0>.
term *Z(Z2, -Z(Z2));
> <0, 4>.
```

```
formula <=Z(-Z(Z1), Z2);
```

For the implementation of quotient and remainder, we need to define the "search range" using an auxiliary function.

```
option silent = true; read integer;

fun ||(x: Z) = if(<=Z(Z0, x), x, -Z(x));
fun sign(x: Z) = if(=(x, Z0), Z0, if(<=Z(Z0, x), Z1, -Z(Z1)));

fun Z[](x: Z) = // the integers with absolute value <= x
  let(y = .0(||(x)):
    ++(set(z in nat(N0, y): true, tuple(z, N0)),
       set(z in nat(N1, y): true, tuple(N0, z))));

fun divZ(x: Z, y: Z) =
  such(q in Z[](||(x)), r in  Z[](-Z(||(y), Z1)):
    and(=(x, +Z(*Z(q, y), r)),
      or(=(sign(r), Z0), =(sign(r), sign(y)))), q);
fun modZ(x: Z, y: Z) =
  such(q in Z[](||(x)), r in  Z[](-Z(||(y), Z1)):
    and(=(x, +Z(*Z(q, y), r)),
      or(=(sign(r), Z0), =(sign(r), sign(y)))), r);

fun Z3 = +Z(Z2, Z1);
fun Z5 = +Z(*Z(Z2, Z2), Z1);
term divZ(Z5, -Z(Z3));
> <0, 2>.

term modZ(Z5, -Z(Z3));
```

One can now show that the operations satisfy the laws stated for their counterparts in $\mathbb{N}$ (Propositions 40, 41, and 42); we give an example.

Proof We show

$$\forall x \in \mathbb{Z}, y \in \mathbb{Z} : x + y = y + x.$$

Take arbitrary $x \in \mathbb{Z}, y \in \mathbb{Z}$. We have

$$\begin{array}{rcl} x + y & = & \text{(definition of } +) \\ \mathrm{I}(x_0 +_{\mathbb{N}} y_0, x_1 +_{\mathbb{N}} y_1) & = & \text{(commutativity of } +_{\mathbb{N}}) \\ \mathrm{I}(y_0 +_{\mathbb{N}} x_0, y_1 +_{\mathbb{N}} x_1) & = & \text{(definition of } +) \\ y + x. & & \end{array}$$

In addition, however, the domain $\mathbb{Z}$ is *closed* with respect to the computation of differences.

Proposition 44 (Difference) For every integer x and y the difference is defined:

$$\forall x \in \mathbb{Z}, y \in \mathbb{Z} : x = (x - y) + y.$$

Proof Take arbitrary $x \in \mathbb{Z}$ and $y \in \mathbb{Z}$. We have

$$\begin{array}{rcl} (x-y)+y & = & \text{(definition of } -) \\ (x+(-y))+y & = & \text{(associativity of} +) \\ x+((-y+y)) & = & (*) \\ x+0 & = & \text{(definition of } + \text{ and } 0) \\ x. & & \end{array}$$

(*) We show $-y + y = 0$:

$$\begin{array}{rcl} -y+y & = & \text{(definition of } -) \\ \langle y_1, y_0 \rangle + y & = & \text{(definition of } +) \\ \mathrm{I}(y_1 +_{\mathbb{N}} y_0, y_0 +_{\mathbb{N}} y_1) & = & \text{(definition of } \mathrm{I}, \text{computation in } \mathbb{N}) \\ (0_{\mathbb{N}}, 0_{\mathbb{N}}) & = & \text{(definition of } 0) \\ 0. & & \end{array}$$

Thus our definition of $\mathbb{Z}$ meets one of the two requirements stated at the begin of this section. In what sense also the other requirement (the embedding of $\mathbb{N}$ into $\mathbb{Z}$) is met will be discussed in Section 6.3. Despite the fact that such an embedding is possible, we will also see in Section 6.2 that there are actually no "more" integer numbers than natural numbers.

We will give in Chapter 7 another construction of $\mathbb{Z}$ which is in some sense "structurally equal" to the definition above but mathematically more elegant (because it is an instance of a more general technique that can be applied in many situations). The definition above however has the advantage that it is *constructive*, i.e., it gives rise to algorithms that can be used for implementation on a computer.

4.3 The Rational Numbers

The integer numbers are not yet closed with respect to all operations. For instance, there does not exist a natural number x with

$$1 = x * 2.$$

i.e., the quotient $1/2$ is undefined.

We therefore introduce a domain $\mathbb{Q}$ of *rational numbers* such that

1. $\mathbb{Z}$ can be "embedded" into $\mathbb{Q}$, and
2. for all integers a and b there is an integer x with $a = x * b$ (and consequently a/b is defined).

We proceed by a set-theoretic construction that represents a rational r as a tuple of integers $\langle x, y\rangle$ such that $r * y = x$ (i.e., $r = x/y$). In the following, we denote by $c_{\mathbb{Z}}$ the operation c in $\mathbb{Z}$.

First, we need some auxiliary functions.

Definition 44 (Integer Conversions) We define the following conversion functions between natural numbers and integers.

$$\begin{aligned}&\mathrm{Z} : \mathbb{N} \to \mathbb{Z}_{\geq 0},\ \mathrm{Z}(x) := \langle x, 0_{\mathbb{N}}\rangle;\\ &\mathrm{N} : \mathbb{Z} \to \mathbb{N},\ \mathrm{N}(x) := |x|_0;\end{aligned}$$

It is easy to see that, for every $n \in \mathbb{N}$, $\mathrm{N}(\mathrm{Z}(n)) = n$ and, for every $i \in \mathbb{N}$, $\mathrm{Z}(\mathrm{N}(i)) = |i|$.

Definition 45 (Rational Numbers) The set $\mathbb{Q}$ of *rational numbers (rationale Zahlen)* is the following subset of $\mathbb{Z} \times \mathbb{Z}$:

$$\mathbb{Q} := \{\langle x, y\rangle : x \in \mathbb{Z} \wedge y \in \mathbb{Z}_{>0} \wedge \mathrm{N}(x) \text{ and } \mathrm{N}(y) \text{ are relatively prime}\}.$$

We define a corresponding constructor function $\frac{*}{*} : \mathbb{Z} \times \mathbb{Z}_{>0} \to \mathbb{Q}$

$$\frac{x}{y} := \langle \mathrm{sign}(x *_{\mathbb{Z}} y) *_{\mathbb{Z}} (|x| \operatorname{div}_{\mathbb{Z}} g), |y| \operatorname{div}_{\mathbb{Z}} g\rangle$$

where $g = \mathrm{Z}(\mathrm{gcd}(\mathrm{N}(x), \mathrm{N}(y)))$.

Let $r \in \mathbb{Q}$ and take x and y such that $r = \langle x, y \rangle$. We call x the *numerator (Zähler)* of r and y its *denominator (Nenner)*:

$$\begin{array}{rcl} \text{numerator}(r) & := & \textbf{such } x : \exists y : r \in \mathbb{Q} \wedge r = \langle x, y \rangle; \\ \text{denominator}(r) & := & \textbf{such } y : \exists x : r \in \mathbb{Q} \wedge r = \langle x, y \rangle. \end{array}$$

This construction makes the tuple components relatively prime and places the sign into the first component.

Example We have for $1 \in \mathbb{Z}$, $2 \in \mathbb{Z}$:

$$\frac{1}{2} = \langle 1, 2 \rangle, \frac{1}{-2} = \langle -1, 2 \rangle, \frac{2}{2} = \langle 1, 1 \rangle.$$

Consequently, every rational number is represented by a unique tuple.

Proposition 46 (Uniqueness) We have for all integers a, b, c, d:

$$(b \neq 0_{\mathbb{Z}} \wedge d \neq 0_{\mathbb{Z}} \wedge a *_{\mathbb{Z}} d = b *_{\mathbb{Z}} c) \Rightarrow \frac{a}{b} = \frac{c}{d}.$$

This gives us the well known simplification rule for every integer a, b, and c:

$$(a \neq 0_{\mathbb{Z}} \wedge c \neq 0_{\mathbb{Z}}) \Rightarrow \frac{a *_{\mathbb{Z}} b}{a *_{\mathbb{Z}} c} = \frac{b}{c}.$$

Logic Evaluator The conversion functions and the basic constructors are implemented as shown below (`@`x, y denotes $\frac{x}{y}$); all rational operations are collected in Appendix C.5, file `rational.txt`).

```
option silent = true;
read integer;

fun Z(x: N) = tuple(x, N0);
fun N(x: Z) = .0(||(x));
pred Q(x) <=>
  and(Tuple(x), =(2, length(x)),
    Z(.0(x)), Z(.1(x)), not(<=Z(.1(x), Z0)),
    relprime(N(.0(x)), N(.1(x))));
fun @(x: Z, y: Z) =
  let(g = Z(gcd(N(x), N(y))):
    tuple(*Z(sign(*Z(x, y)), divZ(||(x), g)), divZ(||(y), g)));
fun numerator(r: Z) = .0(r);
fun denominator(r: Z) = .1(r);

term @(Z(3), Z(2));
> <<3, 0>, <2, 0>>.
term @(Z(3), -Z(Z(2)));
> <<0, 3>, <2, 0>>.
```

```
term @(Z(3), Z(0));
```

We are now ready to define the usual operations on rational numbers.

Definition 46 (Rational Number Operations)

Constants

$$0 := \frac{0_{\mathbb{Z}}}{1_{\mathbb{Z}}};\ 1 := \frac{1_{\mathbb{Z}}}{1_{\mathbb{Z}}};\ 2 := \frac{2_{\mathbb{Z}}}{1_{\mathbb{Z}}}.$$

Arithmetic

$$x + y := \frac{(x_0 *_{\mathbb{Z}} y_1) +_{\mathbb{Z}} (x_1 *_{\mathbb{Z}} y_0)}{x_1 *_{\mathbb{Z}} y_1};$$

$$x * y := \frac{x_0 *_{\mathbb{Z}} y_0}{x_1 *_{\mathbb{Z}} y_1};$$

$$-x := \frac{-_{\mathbb{Z}}\ x_0}{x_1};$$

$$x - y := x + (-y);$$

$$x^{-1} := \frac{x_1}{x_0};$$

$$x/y := x * y^{-1}.$$

Total Order

$$x \leq y :\Leftrightarrow x_0 *_{\mathbb{Z}} y_1 \leq_{\mathbb{Z}} y_0 *_{\mathbb{Z}} x_1.$$

Logic Evaluator The corresponding executable definitions are listed below.

```
option silent = true;
read rational;

fun Q0 = @(Z0, Z1);
fun Q1 = @(Z1, Z1);
fun Q2 = @(Z2, Z1);

fun +Q(x: Q, y: Q) =
  @(+Z(*Z(.0(x), .1(y)), *Z(.1(x), .0(y))), *Z(.1(x), .1(y)));
fun *Q(x: Q, y: Q) =
  @(*Z(.0(x), .0(y)), *Z(.1(x), .1(y)));
fun -Q(x: Q) = @(-Z(.0(x)), .1(x));
fun -Q(x: Q, y: Q) = +Q(x, -Q(y));
fun ^-1Q(x: Q) = @(.1(x), .0(x));
fun /Q(x: Q, y: Q) = *Q(x, ^-1Q(y));

pred <=Q(x: Q, y: Q) <=> <=Z(*Z(.0(x), .1(y)), *Z(.0(y), .1(x)));

term +Q(@(Z(2), Z(3)), @(Z(3), Z(5)));
> <<19, 0>, <15, 0>>.
term *Q(@(Z(2), Z(3)), @(Z(3), Z(5)));
> <<2, 0>, <5, 0>>.
term /Q(@(Z(2), Z(3)), @(Z(3), Z(5)));
> <<10, 0>, <9, 0>>.

term ^-1Q(Q0);
```

Rational Number Laws Again we can show that the operations satisfy the laws stated for their counterparts in $\mathbb{N}$ (Propositions 40, 41, and 42). In addition, however, the domain $\mathbb{Q}$ is *closed* with respect to the computation of quotients.

Proposition 47 (Quotient) For all rationals x and $y \neq 0$ the quotient is defined:

$$\forall x \in \mathbb{Q}, y \in \mathbb{Q}\backslash\{0\} : x = (x/y) * y.$$

Proof Take arbitrary $x \in \mathbb{Q}$ and $y \in \mathbb{Q}\backslash\{0\}$. We have

$$\begin{array}{rcl}
(x/y) * y & = & \text{(definition of } /) \\
(x * y^{-1}) * y & = & \text{(associativity of } *) \\
x * (y^{-1} * y) & = & (*) \\
x * 1 & = & \text{(definition of } * \text{ and } 1) \\
x. & &
\end{array}$$

(*) We show $y^{-1} * y = 1$:

$$\begin{array}{rcl}
y^{-1} * y & = & \text{(definition of } {}^{-1}) \\
\frac{y_1}{y_0} * y & = & \text{(definition of } \frac{*}{*}) \\
\langle \mathrm{sign}(y_1 *_{\mathbb{Z}} y_0) *_{\mathbb{Z}} (|y_1| \ \mathrm{div}_{\mathbb{Z}}\ g), |y_0| \ \mathrm{div}_{\mathbb{Z}}\ g\rangle * y & = & \text{(def. } *, g \neq 0) \\
\frac{(\mathrm{sign}(y_1 *_{\mathbb{Z}} y_0) *_{\mathbb{Z}} (|y_1| \ \mathrm{div}_{\mathbb{Z}}\ g)) *_{\mathbb{Z}} y_0}{(|y_0| \ \mathrm{div}_{\mathbb{Z}} g) *_{\mathbb{Z}} y_1} & = & \text{(Prop. 46, } g \neq 0) \\
\frac{\mathrm{sign}(y_1 *_{\mathbb{Z}} y_0) *_{\mathbb{Z}} |y_1| *_{\mathbb{Z}} y_0}{|y_0| *_{\mathbb{Z}} y_1} & = & \text{(Prop. 46, } y \neq 0) \\
\frac{\mathrm{sign}(y_1 *_{\mathbb{Z}} y_0) *_{\mathbb{Z}} \mathrm{sign}(y_0)}{\mathrm{sign}(y_1)} & = & \text{(arithmetic in } \mathbb{Z}) \\
\frac{\mathrm{sign}(y_1) *_{\mathbb{Z}} \mathrm{sign}(y_0) *_{\mathbb{Z}} \mathrm{sign}(y_0)}{\mathrm{sign}(y_1)} & = & \text{(Pr. 46, } y_1 \neq 0) \\
\frac{\mathrm{sign}(y_0) *_{\mathbb{Z}} \mathrm{sign}(y_0)}{1_{\mathbb{Z}}} & = & \text{(arithmetic in } \mathbb{Z}) \\
\frac{1_{\mathbb{Z}}}{1_{\mathbb{Z}}} & = & \text{(definition of } 1) \\
1. & &
\end{array}$$

Our definition of $\mathbb{Q}$ meets one of the two requirements stated at the begin of this section. How $\mathbb{Z}$ is embedded into $\mathbb{Q}$ will be discussed in Section 6.3. In Chapter 3, we will give a construction of $\mathbb{Q}$ which is "structurally equal" to the definition above (and which is another instance of the general technique that is used for a corresponding definition of $\mathbb{Z}$).

Another characteristic property that distinguishes $\mathbb{Q}$ from $\mathbb{Z}$ is stated next.

Proposition 49 () Between any two rational numbers, there is another rational number:

$$\forall x \in \mathbb{Q}, y \in \mathbb{Q} : x < y \Rightarrow \exists z \in \mathbb{Q} : x < z < y.$$

Proof Take any $x \in \mathbb{Q}$ and $y \in \mathbb{Q}$ with $x < y$. Then $x < (x + y)/2 < y$.

As a consequence, between any two rational numbers there are *infinitely* many other rational numbers. $\mathbb{Q}$ can be therefore used for modelling arbitrarily "fine-grained" realities. However, we will see in Section 6.2 that there are not more rational numbers than there are integer numbers (or natural numbers). Furthermore, the next section shows that there are truly "continuous" realities that cannot be modelled even by rationals.

4.4 The Real Numbers

The rational numbers are still not complete with respect to the basic operations: there does not exist an $x \in \mathbb{Q}$ with

$$\mathrm{x} * \mathrm{x} = 2.$$

i.e., there is no square root of 2 in $\mathbb{Q}$.

Proof We show

$$\forall x \in \mathbb{Q} : x * x \neq 2.$$

Take arbitrary $x \in \mathbb{Q}$. We assume (1) $x * x = 2$ and show a contradiction. From the construction of $\mathbb{Q}$, we know $x = \frac{a}{b}$ for some $a \in \mathbb{Z}$ and $b \in \mathbb{Z}_{>0}$ such that (2) $\mathrm{N}(a)$ and $\mathrm{N}(b)$ are relatively prime. We have $\frac{a\ *_{\mathbb{Z}}\ a}{b\ *_{\mathbb{Z}}\ b} = 2$ and thus (from now on we operate in $\mathbb{Z}$ and drop the corresponding subscripts):

$$(3)\ a * a = 2 * b * b.$$

From (3) we know $\mathrm{N}(2)|\mathrm{N}(a * a)$ and thus also (4) $\mathrm{N}(2)|\mathrm{N}(a)$ (a proposition that has to be proved extra). Therefore there exists some $c \in \mathbb{Z}$ such that

(5) $a = 2 * c$.

From (3) and (5) we have $2 * c * 2 * c = 2 * b * b$, i.e., $2 * c * c = b * b$, thus (6) $\mathrm{N}(2)|\mathrm{N}(b * b)$ and therefore (7) $\mathrm{N}(2)|\mathrm{N}(b)$. (4) and (7) contradict (2).

We will now introduce the domain $\mathbb{R}$ of real numbers which provides a solution for the equation stated above. While there exist also relatively concrete set-theoretic constructions of $\mathbb{R}$ (e.g. as a set of infinite decimal fractions), the usual construction is another instance of the kind that will be stated for $\mathbb{Z}$ and $\mathbb{Q}$ in Chapter 3.

In this section, we will characterize $\mathbb{R}$ in an inconstructive way by a number of axioms.

Axiom 4 (Real Numbers) The domain of *real numbers (reelle Zahlen)* is a set $\mathbb{R}$ with constants $0 \in \mathbb{R}$ and $1 \in \mathbb{R}$, functions $+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, $- : \mathbb{R} \to \mathbb{R}$, $* : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, ${}^{-1} : \mathbb{R} \backslash \{0\} \to \mathbb{R}$ and predicate $\leq \subseteq \mathbb{R} \times \mathbb{R}$ such that, for all $x \in \mathbb{R}$, $y \in \mathbb{R}$, and $z \in \mathbb{R}$, the following holds:

Field Axioms (Körperaxiome)

$$\begin{array}{rcl}
x + y & = & y + x, \\
x + (y + z) & = & (x + y) + z, \\
x + (-y) & = & 0, \\
x * y & = & y * x, \\
x * (y * z) & = & (x * y) * z, \\
x * (x^{-1}) & = & 1, \\
x * 1 & = & x, \\
x * (y + z) & = & (x * y) + (x * z);
\end{array}$$

Order Axioms (Ordnungsaxiome)

$$\begin{array}{l}
x \leq x, \\
x \leq y \lor y \leq z, \\
(x \leq y \land y \leq z) \Rightarrow x \leq z, \\
x \leq y \Rightarrow x + z \leq y + z, \\
(x \leq y \land 0 \leq z) \Rightarrow x \leq y * z.
\end{array}$$

Completeness Axiom (Vollständigkeitsaxiom)

Every non-empty subset of $\mathbb{R}$ that has an upper bound also has an upper limit.

$$\begin{array}{l} \forall S \subseteq \mathbb{R} : S \neq \emptyset \Rightarrow \\ \quad ((\exists B \in \mathbb{R} : B \text{ is upper bound of } S) \Rightarrow \\ \quad\quad (\exists L \in \mathbb{R} : L \text{ is upper limit of } S)). \end{array}$$

The predicates used in the completeness axiom are defined below.

Definition 47 (Bounds and Limits) An *upper bound (obere Schranke)* of S is as large as every element of S:

$$B \text{ is upper bound of } S :\Leftrightarrow \forall x \in S : x \leq B.$$

An *upper limit (obere Grenze)* of S is the smallest upper bound of S:

$$\begin{array}{l} L \text{ is upper limit of } S :\Leftrightarrow \\ \quad L \text{ is upper bound of } S \wedge \\ \quad (\forall M : M \text{ is upper bound of } S \Rightarrow B \leq M). \end{array}$$

Intuitively, the completeness axiom makes the real numbers much more "dense" than the rational numbers where above property does not hold.

Example Take the set $S := \{s_i : i \in \mathbb{N}\}$ where s_i is the sum of the first i components of the sequence

$$\begin{array}{l} a : \mathbb{N} \rightarrow \mathbb{Q} \\ a(i) = \frac{1}{i!} \end{array}$$

and ! denotes the factorial function, i.e.,

$$a = [\frac{1}{1}, \frac{1}{1}, \frac{1}{1*2}, \frac{1}{1*2*3}, \frac{1}{1*2*3*4}, \ldots]$$

and

$$S = \{\frac{1}{1}, (\frac{1}{1} + \frac{1}{1}), (\frac{1}{1} + \frac{1}{1} + \frac{1}{1*2}), (\frac{1}{1} + \frac{1}{1} + \frac{1}{1*2} + \frac{1}{1*2*3}), \ldots\}.$$

S has an upper bound $\frac{3}{1}$ but no upper limit in $\mathbb{Q}$. However, it has an upper limit in $\mathbb{R}$ traditionally denoted by the constant e $(= 2.718281828\ldots)$.

Then we have the following result which implies that the problem stated at the begin of this section can be solved.

Proposition 52 (Existence of Real Roots) In $\mathbb{R}$ every non-negative number has an n-th root, i.e., for $\mathbb{R}_{\geq 0} := \{x \in \mathbb{R} : x \geq 0\}$ the following holds:

$$\forall a \in \mathbb{R}_{\geq 0}, n \in \mathbb{N}_{>0} : \exists x \in \mathbb{R} : x^n = a.$$

Since we can show that, for every $x \in \mathbb{R}$, $x * x = x^{2_\mathbb{N}}$, this implies the existence of a square root of 2 in $\mathbb{R}$.

Definition 48 (Real Root Function)

$$\sqrt[n]{x} := \textbf{such } y : x^n = y$$
$$\sqrt{x} := \sqrt[2_\mathbb{N}]{x}.$$

As a consequence of Proposition 52, we have

$$\forall a \in \mathbb{R}_{\geq 0}, n \in \mathbb{N}_{>0} : (\sqrt[n]{a})^n = a.$$

Contiguous subsets of the reals are often denoted as shown below.

Definition 49 (Intervals) We define the following *intervals (Intervalle)*:

$$[a, b] := \{x \in \mathbb{R} : a \leq x \leq b\};$$
$$[a, b[:= \{x \in \mathbb{R} : a \leq x < b\};$$
$$]a, b] := \{x \in \mathbb{R} : a < x \leq b\};$$
$$]a, b[:= \{x \in \mathbb{R} : a < x < b\}.$$

For all $a \in \mathbb{R}$ and $b \in \mathbb{R}$, the intervals $[a, b]$ and $[a, b[$ are called *left closed (linksseitig abgeschlossen)*, and the intervals $[a, b]$ and $]a, b]$ are called *right closed (rechtsseitig abgeschlossen)*.

Intervals are contiguous in the sense that if two values are in an interval S, then also every intermediate value is in this interval, i.e.,

$$(\forall x \in S, y \in S : x \leq y \Rightarrow (\forall z \in \mathbb{R} : x \leq z \leq y \Rightarrow z \in S)).$$

The set $\mathbb{R}$ "embeds" $\mathbb{Q}$ in the sense that will be discussed in Section 6.3. However, unlike the relationship between $\mathbb{Z}$ and $\mathbb{Q}$ (which are essentially of the "same" size), there are considerably "more" reals than there are rationals. Actually we will see in Section 6.2 that there are so many elements of $\mathbb{R}$ that this set cannot be represented in a computer even with an *infinite* number of memory cells.

Consequently the Logic Evaluator does in file `real.txt` not implement the reals but fakes real number operations (without square root computation) by rational number operations (see Appendix 339).

In Chapter 3 we will sketch a "structurally equal" construction that is an instance of the general technique that can be applied for the construction of $\mathbb{Z}$, and $\mathbb{Q}$.

4.5 The Complex Numbers

Even $\mathbb{R}$ does not allow to solve all equations that can be formulated with $+$ and $*$, e.g. there is no $x \in \mathbb{R}$ such that

$$x * x = -1.$$

Proof We prove $\forall x \in \mathbb{R} : x * x \neq -1$. Take arbitrary $x \in \mathbb{R}$. If $x \geq 0$, then $x * x \geq 0$. If $x < 0$, then also $x * x \geq 0$. Since $-1 \not\geq 0$, $x * x \neq 0$.

We introduce a domain in which all equations of this kind can be solved.

Definition 50 (Complex Numbers) The set $\mathbb{C}$ of *complex numbers (komplexe Zahlen)* is defined as follows:

$$\mathbb{C} := \mathbb{R} \times \mathbb{R}.$$

We define a corresponding constructor

$$
\begin{aligned}
&\_ + \_\mathrm{i} : \mathbb{R} \times \mathbb{R} \to \mathbb{C} \\
&x + y\mathrm{i} := \langle x, y \rangle.
\end{aligned}
$$

Let $c \in \mathbb{C}$ and take x and y such that $c = \langle x, y \rangle$. We call x the *real part (Realteil)* of c and y its *imaginary part (Imaginärteil)*:

$$
\begin{aligned}
\mathrm{real}(c) &:= \textbf{such } x : \exists y : c \in \mathbb{C} \wedge c = \langle x, y \rangle; \\
\mathrm{imaginary}(c) &:= \textbf{such } y : \exists x : c \in \mathbb{C} \wedge c = \langle x, y \rangle.
\end{aligned}
$$

So the question what a complex number $x + y\mathrm{i}$ is, can be easily answered: it is the tuple $\langle x, y \rangle$.

Logic Evaluator The corresponding executable definitions (on the basis of rationals instead of reals) are given in Appendix C.7, file `complex.txt`.

```
option silent = true;
read real;

pred C(x) <=>
  and(Tuple(x), =(2, length(x)), R(.0(x)), R(.1(x)));
fun +i(x: R, y: R) = tuple(x, y);
fun real(r: C) = .0(r);
fun imaginary(r: C) = .1(r);

term +i(R1, R2);
> <<<1, 0>, <1, 0>>, <<2, 0>, <1, 0>>>.
term real(+i(R1, R2));
> <<1, 0>, <1, 0>>.
```

```
term imaginary(+i(R1, R2));
```

The usual operations on complex numbers are then defined as follows:

Definition 51 (Complex Number Operations)

Constants

$$0 := 0_{\mathbb{R}} + 0_{\mathbb{R}}\mathrm{i};\ 1 := 1_{\mathbb{R}} + 0_{\mathbb{R}}\mathrm{i};\ 2 := 2_{\mathbb{R}} + 0_{\mathbb{R}}\mathrm{i};$$

The *imaginary constant (imaginäre Konstante)*:

$$i := 0_{\mathbb{R}} + 1_{\mathbb{R}}\mathrm{i}.$$

Arithmetic

$$\begin{array}{l}
x + y := (x_0 +_{\mathbb{R}} y_0) + (x_1 +_{\mathbb{R}} y_1)\mathrm{i};\\
x - y := (x_0 -_{\mathbb{R}} y_0) + (x_1 -_{\mathbb{R}} y_1)\mathrm{i};\\
-x := (-_{\mathbb{R}}x_0) + (-_{\mathbb{R}}x_1)\mathrm{i};\\
x * y := ((x_0 *_{\mathbb{R}} y_0) -_{\mathbb{R}} (x_1 *_{\mathbb{R}} y_1)) + ((x_0 *_{\mathbb{R}} y_1) + (x_1 *_{\mathbb{R}} y_0))\mathrm{i};\\
x/y :=\\
\quad (((x_0 *_{\mathbb{R}} y_0) +_{\mathbb{R}} (x_1 *_{\mathbb{R}} y_1))/_{\mathbb{R}}\ d)+\\
\quad (((x_1 *_{\mathbb{R}} y_0) -_{\mathbb{R}} (x_0 *_{\mathbb{R}} y_1))/_{\mathbb{R}}\ d)\mathrm{i}\\
\qquad \textbf{where } d = (y_0 *_{\mathbb{R}} y_0) +_{\mathbb{R}} (y_1 *_{\mathbb{R}} y_1);\\
x^{-1} := 1/x.
\end{array}$$

Special Operations

The *complex conjugate (konjugiert komplexe Zahl)*:

$$\overline{x} := x_0 + (-_{\mathbb{R}}x_1)\mathrm{i}.$$

The *absolute value (Absolutbetrag)*:

$$|x| := \sqrt{(x_0 *_{\mathbb{R}} x_0) +_{\mathbb{R}} (x_1 *_{\mathbb{R}} x_1)}.$$

Please note that an application $x+y\mathrm{i}$ of the constructor function denotes the tuple $\langle x, y\rangle$ while the imaginary constant i denotes $(0_{\mathbb{R}} + 1_{\mathbb{R}}\mathrm{i})$, i.e., $\langle 0_{\mathbb{R}}, 1_{\mathbb{R}}\rangle$. This imaginary constant is the "missing value" that solves the equation stated at the begin of this section.

Proof We prove $i * i = -1$.

$$\begin{array}{rcl}
i * i & = & (\text{definition of } i)\\
(0_{\mathbb{R}} + 1_{\mathbb{R}}\mathrm{i}) *_{\mathbb{R}} (0_{\mathbb{R}} + 1_{\mathbb{R}}\mathrm{i}) & = & (\text{definition of } *)\\
((0_{\mathbb{R}} *_{\mathbb{R}} 0_{\mathbb{R}}) -_{\mathbb{R}} (1_{\mathbb{R}} *_{\mathbb{R}} 1_{\mathbb{R}}))+ & & \\
((0_{\mathbb{R}} *_{\mathbb{R}} 1_{\mathbb{R}}) +_{\mathbb{R}} (1_{\mathbb{R}} *_{\mathbb{R}} 0_{\mathbb{R}}))\mathrm{i} & = & (\text{arithmetic in } \mathbb{R})\\
(-_{\mathbb{R}}1_{\mathbb{R}}) + (-_{\mathbb{R}}0_{\mathbb{R}})\mathrm{i} & = & (\text{definition of } -)\\
-(1_{\mathbb{R}} + 0_{\mathbb{R}}\mathrm{i}) & = & (\text{definition of } 1)\\
-1. & &
\end{array}$$

Logic Evaluator Function `conj` computes the complex conjugate and `||^2` returns the square of the absolute value:

```
option silent = true;
read complex;

fun C0 = +i(R0, R0);
fun C1 = +i(R1, R0);
fun C2 = +i(R2, R0);
fun i  = +i(R0, R1);

fun +C(x: C, y: C) = +i(+R(.0(x), .0(y)), +R(.1(x), .1(y)));
fun -C(x: C, y: C) = +i(-R(.0(x), .0(y)), -R(.1(x), .1(y)));
fun -(x: C) = -C(C0, x);
fun *C(x: C, y: C) =
  +i(-R(*R(.0(x), .0(y)), *R(.1(x), .1(y))),
     +R(*R(.0(x), .1(y)), *R(.1(x), .0(y))));
fun /C(x: C, y: C) =
  let(d = +R(*R(.0(y), .0(y)), *R(.1(y), .1(y))):
    +i(/R(+R(*R(.0(x), .0(y)), *R(.1(x), .1(y))), d),
       /R(-R(*R(.1(x), .0(y)), *R(.0(x), .1(y))), d)));
fun ^-1C(x: C) = /C(C1, x);

fun conj(x: C) = +i(.0(x), -R(.1(x)));
fun ||^2(x: C) = +R(*R(.0(x), .0(x)), *R(.1(x), .1(x)));

term *C(i, i);
> <<<0, 1>, <1, 0>>, <<0, 0>, <1, 0>>>.
term *C(+i(R1, R2), +i(R2, R2));
> <<<0, 2>, <1, 0>>, <<6, 0>, <1, 0>>>.
term *C(+i(R0, R1), ^-1C(+i(R0, R1)));
> <<<1, 0>, <1, 0>>, <<0, 0>, <1, 0>>>.

term let (x = +i(R1, R2): -C(x, +i(real(x), imaginary(x))));
```

The following proposition tells us that our quest for a closed number domain has reached its goal.

Proposition 55 (Fundamental Theorem of Algebra) For every $a_0 \in \mathbb{C}, \ldots, a_{n-1} \in \mathbb{C}$, there exists an $x \in \mathbb{C}$ such that

$$a_0 * x^0 + \ldots + a_{n-1} * x^{n-1} = 0.$$

We now introduce the complex root function analogously to the corresponding operation on the reals.

Definition 52 (Complex Root Function)

$$\sqrt[n]{x} := \textbf{such } y : x^n = y$$

As a consequence of Proposition 55, we then have

$$\forall a \in \mathbb{C}, n \in \mathbb{N}_{>0} : (\sqrt[n]{a})^n = a.$$

For the special case of the square root function, we give a definition that immediately allows us to "compute" the square of a complex number.

Definition 53 (Complex Square Root)

$$\begin{array}{l}
\sqrt{.} : \mathbb{C} \to \mathbb{C} \\
\sqrt{x} := \\
\quad \textbf{if } x_1 \geq_{\mathbb{R}} 0_{\mathbb{R}} \textbf{ then } u + v\mathrm{i} \textbf{ else } u + (-_{\mathbb{R}}v)\mathrm{i} \\
\quad \textbf{where} \\
\qquad u = \sqrt{(x_0 +_{\mathbb{R}} \sqrt{{x_0}^2 +_{\mathbb{R}} {x_1}^2}) \ /_{\mathbb{R}} \ 2_{\mathbb{R}}} \\
\qquad v = \sqrt{(-_{\mathbb{R}}x_0 +_{\mathbb{R}} \sqrt{{x_0}^2 +_{\mathbb{R}} {x_1}^2}) \ /_{\mathbb{R}} \ 2_{\mathbb{R}}}.
\end{array}$$

The equation $x * x = a$ has in general two solutions, the one denoted by the definition above and the one we get by multiplying the value with (-1). In other words, the following holds:

$$\begin{array}{l}
\forall x \in \mathbb{C} : \\
\quad \textbf{let } r = \sqrt{x} : \\
\qquad x = r * r \wedge x = (-r) * (-r).
\end{array}$$

Absolute Value Out definition of the absolute value implies, for every $x \in \mathbb{R}$, $|x + 0\mathrm{i}| = \sqrt{x *_{\mathbb{R}} x}$, i.e.,

$$|x + 0\mathrm{i}| = \textbf{if } x \leq_{\mathbb{R}} 0_{\mathbb{R}} \textbf{ then } -_{\mathbb{R}} x \textbf{ else } x$$

which corresponds to the usual definition of $|\ |$ on $\mathbb{R}$. The absolute value satisfies the following equations and inequalities.

Proposition 56 (Absolute Value) For every $x \in \mathbb{C}, y \in \mathbb{C}, z \in \mathbb{C}$, the following holds:

$$\begin{array}{rcl}
|x| & \geq_{\mathbb{R}} & 0_{\mathbb{R}}, \\
|-x| & = & |x|, \\
|x| = 0_{\mathbb{R}} & \Leftrightarrow & x = 0, \\
|x+y| & \leq_{\mathbb{R}} & |x| +_{\mathbb{R}} |y|, \\
|x-y| & \leq_{\mathbb{R}} & |x| +_{\mathbb{R}} |y|, \\
||x| -_{\mathbb{R}} |y|| & \leq_{\mathbb{R}} & |x+y|, \\
||x| -_{\mathbb{R}} |y|| & \leq_{\mathbb{R}} & |x-y|, \\
|x * y| & = & |x| *_{\mathbb{R}} |y|, \\
y \neq 0 \;\Rightarrow\; |x/y| & = & |x| /_{\mathbb{R}} |y|.
\end{array}$$

Likewise we have the following laws for the complex conjugate.

Proposition 57 (Complex Conjugate) For every $x \in \mathbb{C}, y \in \mathbb{C}, z \in \mathbb{C}$, the following holds:

$$\begin{array}{rcl}
\overline{\overline{x}} & = & x, \\
\overline{x+y} & = & \overline{x} + \overline{y}, \\
\overline{x*y} & = & \overline{x} * \overline{y}, \\
y \neq 0 \Rightarrow \overline{x/y} & = & \overline{x}/\overline{y}, \\
x_1 = 0_{\mathbb{R}} & \Leftrightarrow & x = \overline{x}, \\
x & = & ((x+\overline{x})/2) + ((x-\overline{x})/2)\mathrm{i}, \\
|x| & = & \sqrt{z * \overline{z}}
\end{array}$$

The set $\mathbb{C}$ "embeds" $\mathbb{R}$ in the sense that will be discussed in Section 6.3.

4.6 Relationships between Number Domains

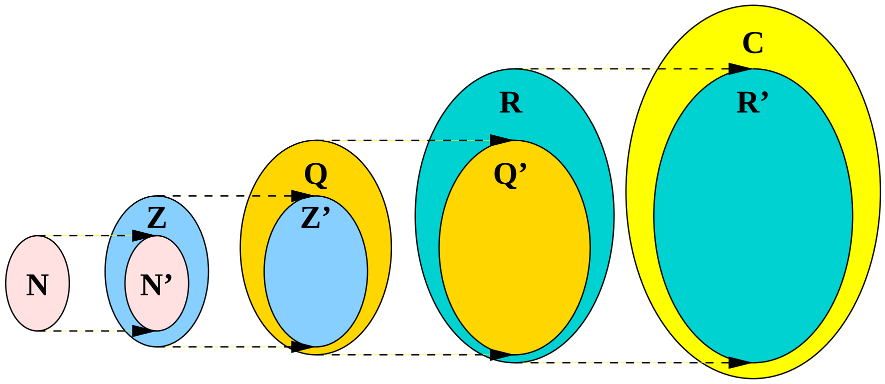
In the previous sections we have introduced the sets $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ with the property

$$\mathbb{N} \subseteq^0 \mathbb{Z}, \mathbb{Z} \subseteq^1 \mathbb{Q}, \mathbb{Q} \subseteq^2 \mathbb{R}, \mathbb{R} \subseteq^3 \mathbb{C}$$

where the relations $\subseteq^i$ denote the corresponding embeddings that we will discuss in Section 6.3. These embeddings preserve all relevant notions (functions and predicates) and the corresponding properties such that we can operate in the domains as if we actually had the relationship

$$\mathbb{N} \subseteq \mathbb{Z}, \mathbb{Z} \subseteq \mathbb{Q}, \mathbb{Q} \subseteq \mathbb{R}, \mathbb{R} \subseteq \mathbb{C}.$$

Indeed there exists for every domain D an "identical twin" D' which is a subset of the next larger domain:



We will from now on operate with these twins pretending that they are the siblings that we have actually defined.

Consequently, we do not any more bother whether $+$ denotes $+_{\mathbb{Z}}$ or $+_{\mathbb{Q}}$; they have essentially the same properties with respect to computing and reasoning. Of course, we must still take care of that some functions have no result in a particular domain (the difference of two natural numbers is not necessarily a natural number) but only in an enclosing domain (the difference of two natural numbers is always an integer number).

4.7 Arithmetic Notions

We introduce additional notions that can be used in all domains that have constants $0, 1, +, *, \leq$ with "number-like" properties.

4.7.1 Minimum and Maximum

Definition 54 (Minimum and Maximum Quantifier) If x is a variable and F is a formula, then the following are terms with bound variable x:

$$\min_x F;$$
$$\max_x F.$$

The value of the first term is the smallest value of x such that F holds; the value of the second term is the largest such value, i.e.,

$$\min_x F := \textbf{such } x : F \wedge (\forall y : F[x \leftarrow y] \Rightarrow x \leq y);$$
$$\max_x F := \textbf{such } x : F \wedge (\forall y : F[x \leftarrow y] \Rightarrow x \geq y).$$

Usually the variable x is dropped and has to be deduced from the context.

We can use this notion to define corresponding functions on sets as follows.

Definition 55 (Minimum and Maximum Functions)

$$\begin{array}{rcl} \min(S) & := & \min_x x \in S; \\ \max(S) & := & \max_x x \in S; \end{array}$$

Please note that the definition of "min" and "max" depends on the choice of the predicate $\leq$. Furthermore, minimum and maximum are not necessarily defined.

Example

- We have

  $$\max_x(\text{prime}(x) \wedge x|100) = 5.$$

- The value of

  $$\min(\{1/x : x \in \mathbb{N}_{>0}\})$$

  is undefined because for every x in $\{1/1,\ 1/2,\ 1/3,\ 1/4,\ \ldots\}$ there is always an y in this set with $y < x$, namely $1/(x+1)$.

4.7.2 Arithmetic Quantifiers

Frequently we want to construct the sum of a collection of values.

Definition 56 (Sum Quantifier) If x is a variable, F is a formula and T is a term, then the following is a term with bound variable x:

$$\sum_{x,F} T.$$

The value of this term is 0, if F does not hold for any x; otherwise it is, for some x that satisfies F, the sum of the value of T and of the value of the term for all other x, i.e.,

$$\begin{array}{rcl} (\forall x : \neg F) & \Rightarrow & \sum_{x,F} T = 0; \\ (\forall y : F[x \leftarrow y] & \Rightarrow & \sum_{x,F} T = T[x \leftarrow y] + \sum_{x,F \wedge x \neq y} T). \end{array}$$

Please note that the value of the term is only defined if there do not exist infinitely many x such that F holds.

Usually F contains the condition $x \in \mathbb{N}$, which is therefore not written explicitly. Also the variable x is usually not written but has to be deduced from the context. Frequently also the form

$$\sum_{i=a}^{b} T$$

is used to denote $\sum_{0 \leq i \leq a} T$.

Example

$$\sum_{1 \leq i \leq n} i^2 = \sum_{i,(i \in \mathbb{N} \wedge 1 \leq i \wedge i \leq n)} i^2;$$

$$\sum_{1 \leq i \leq 5} i^2 = 1^2 + 2^2 + 3^2 + 4^2 + 5^2;$$

$$\sum_{1 \leq i \leq 0} i^5 = 0;$$

$$\sum_{1 \leq i \leq 9} (x-i)^2 = (x-1)^2 + \sum_{2 \leq i \leq 9} (x-i)^2$$

$$\sum_{1 \leq i \leq n} x^i = \sum_{1 \leq i \leq n \wedge \mathrm{even}(i)} x^i + \sum_{1 \leq i \leq n \wedge \mathrm{odd}(i)} x^i$$

Example Let $a := [3,1,2,9,0,7]$. We have

$$\sum_{0 \leq i \leq 5} a_i * 10^i = 709213.$$

In general, for any finite sequence d of "decimal digits" the term

$$\sum_{0 \leq i < \mathrm{length}(d)} d_i * 10^i$$

denotes the value of this sequence in the decimal number system. Likewise, for any finite sequence b of binary digits 0 and 1, the value

$$\sum_{0 \leq i < \mathrm{length}(b)} b_i * 2^i$$

denotes the value of this sequence in the binary number system, e.g., the value of $[0,1,1,0,1]$ is

$$1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 = 22.$$

The quantifier may also bind multiple variables.

Example

$$\sum_{1 \leq i \leq 5, 1 \leq j \leq 3} i * j = 1*1 + 1*2 + 1*3 + \sum_{2 \leq i \leq 5, 1 \leq j \leq 3} i * j$$

$$\sum_{1 \leq i \leq 3, 1 \leq j \leq i} i * j = 1*1 + 2*1 + 2*2 + 3*1 + 3*2 + 3*3.$$

We have a number of important identities.

Proposition 58 (Sum Quantifier) For all variables i and j and formulas F (in which j does not occur freely), G (in which i does not occur freely), and H and terms T and U we have:

$$\sum_{i,F} T * \sum_{j,G} U = \sum_{i,F} \sum_{j,G} T * U.$$

$$\sum_{i,F} \sum_{j,G} T = \sum_{j,G} \sum_{i,F} T = \sum_{i,j,F \wedge G} T.$$

$$\sum_{i,F} T + \sum_{i,H} T = \sum_{i,F \vee H} T + \sum_{i,F \wedge H} T.$$

Furthermore, if term C is a term in which i does not occur freely, we have:

$$\sum_{i,F} C * T = C * \sum_{i,F} T.$$

$$\sum_{i,F} C = n * C \text{ (where } n \text{ is the number of } i \text{ for which } F \text{ holds).}$$

Example

$$\sum_{1 \leq i \leq n} x^i * \sum_{1 \leq j \leq m} x^j = \sum_{1 \leq i \leq n} \sum_{1 \leq j \leq m} x^{i+j} = \sum_{1 \leq i \leq n \wedge 1 \leq j \leq m} x^{i+j}.$$

$$\sum_{1 \leq i \leq n} \sum_{1 \leq j \leq m} i * x^j = \sum_{1 \leq i \leq n} (i * \sum_{1 \leq j \leq m} x^j).$$

Likewise we often wish to construct the product of a collection of values.

Definition 57 (Product Quantifier) If x is a variable, F is a formula and T is a term, then the following is a term with bound variable x:

$$\prod_{x,F} T.$$

The value of this term is 1, if F does not hold for any x; otherwise it is, for some x that satisfies F, the product of the value of T and of the value of the term for all other x, i.e.,

$$\begin{array}{rcl} (\forall x : \neg F) & \Rightarrow & \prod_{x,F} T = 1; \\ (\forall y : F[x \leftarrow y] & \Rightarrow & \prod_{x,F} T = T[x \leftarrow y] + \prod_{x,F \wedge x \neq y} T). \end{array}$$

The same remarks apply as for the definition of the sum quantifier.

Example

$$\prod_{1 \leq i \leq 0} i^5 = 1;$$

$$\prod_{0 \leq i \leq 5} a_i = a_0 * a_1 * a_2 * a_3 * a_4 * a_5$$

$$\prod_{0 \leq i \leq 100} a_i = a_0 * a_1 * a_2 * \prod_{3 \leq i \leq 100} a_i$$

$$\prod_{0 \leq i \leq 100} a_i = \prod_{0 \leq i \leq 50} a_i * \prod_{51 \leq i \leq 100} a_i$$

$$\prod_{\substack{1 \leq i \leq 3 \\ 1 \leq j \leq i}} i + j = (1+1) * (2+1) * (2+2) * (3+1) * (3+2) * (3+3).$$

We have the following identities.

Proposition 59 (Product Quantifier) For all variables i and j and formulas F (in which j does not occur freely), G (in which i does not occur freely), and H and terms T and U we have:

$$\prod_{i,F}\prod_{j,G} T = \prod_{j,G}\prod_{i,F} T = \prod_{i,j,F\wedge G} T.$$

$$\prod_{i,F} T * \prod_{i,H} T = \prod_{i,F\vee H} T * \prod_{i,F\wedge H} T.$$

Furthermore, if term C is a term in which i does not occur freely, we have:

$$\prod_{i,F} C * T = C^n \prod_{i,F} T \text{ (where } n \text{ is the number of } i \text{ for which } F \text{ holds).}$$

$$\prod_{i,F} C = C^n \text{ (where } n \text{ is the number of } i \text{ for which } F \text{ holds)}$$

With the help of the product quantifier we can now formulate the uniqueness of prime number composition stated in Section 4.1.

Proposition 60 (Prime Number Factorization) For every natural number $n \neq 0$, there exists a unique prime number factorization:

$$\forall n \in \mathbb{N}_{>0} : (\exists p : \text{pf}(n,p) \wedge (\forall q : \text{pf}(n,q) \Rightarrow p = q)).$$

A prime number factorization of n is a sequence of pairs (p, e) ordered by the primes p such that n is the product of all p^e.

$$\begin{aligned}
&\text{pf}(n,p) :\Leftrightarrow \\
&\quad p : \mathbb{N} \to \mathbb{N} \times \mathbb{N} \wedge \\
&\quad (\forall i \in \mathbb{N} : p(i)_0 \text{ is prime } \wedge p(i)_0 < p(i+1)_0) \wedge \\
&\quad (\exists k \in \mathbb{N} : n = \textstyle\prod_{0\leq i\leq k} p(i)_0{}^{p(i)_1} \wedge \forall i > k : p(i)_1 = 0)
\end{aligned}$$

Example Because $300 = 2^2 * 3^1 * 5^2$, we have for 300 the prime number factorization

$$[\langle 2,2\rangle, \langle 3,1\rangle, \langle 5,2\rangle, \langle 7,0\rangle, \langle 11,0\rangle, \langle 13,0\rangle, \ldots].$$

Likewise, 1 has the prime number factorization

$$[\langle 2,0\rangle, \langle 3,0\rangle, \langle 5,0\rangle, \langle 7,0\rangle, \langle 11,0\rangle, \langle 13,0\rangle, \ldots].$$

4.7.3 Binomials

In combinatorics (the branch of mathematics that solves questions of the kind "how many objects do exist such that ...?"), the following notions are of importance.

Definition 58 (Factorial) The *factorial (Fakultät)* of a natural number n is the product of all non-zero numbers less than or equal n:

$$n! := \prod_{1 \le i \le n} i.$$

Please note that the definition of $\prod$ implies $0! = 1$.

Definition 59 (Binomial Coefficient) The *binomial coefficient (Binomialkoeffizient)* of two natural numbers is defined as follows:

$$\binom{n}{k} := \textbf{if } 0 \le k \le n \textbf{ then } \frac{n!}{k! * (n-k)!} \textbf{ else } 0.$$

We read this as "n choose k" ("n über k").

The name "n choose k" stems from the fact that $\binom{n}{k}$ is the number of ways to choose a k element set from an n-element set.

Example The set $\{0,\ 1,\ 2,\ 3\}$ has $6 = \binom{4}{2}$ subsets with 2 elements:

$$\{0,1\}, \{0,2\}, \{0,3\}, \{1,2\}, \{1,3\}, \{2,3\}.$$

By above definition we have for every n and k with $0 \leq k \leq n$

$$\binom{n}{k} = \frac{\prod_{n-k+1 \leq i \leq n} i}{\prod_{1 \leq i \leq n} i}.$$

Furthermore, we have the following important identities.

Proposition 61 (Binomial Identities) For every $n \in \mathbb{N}$ and $k \in \mathbb{N}$ with $0 \leq k \leq n$, the following holds:

$$\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1},$$

$$\binom{n}{k} = \binom{n}{n-k},$$

$$\binom{n}{0} = \binom{n}{n} = 1.$$

These three laws give rise to *Pascal's triangle*:

$$\begin{array}{ccccccc} & & & 1 & & & \\ & & 1 & & 1 & & \\ & 1 & & 2 & & 1 & \\ 1 & & 3 & & 3 & & 1 \\ \multicolumn{7}{c}{\cdots\cdots\cdots\cdots} \end{array} \quad = \quad \begin{array}{ccccccc} & & & \binom{0}{0} & & & \\ & & \binom{1}{0} & & \binom{1}{1} & & \\ & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & \\ \binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} \\ \multicolumn{7}{c}{\cdots\cdots\cdots\cdots} \end{array}$$

This triangle is bounded by sides of 1 and where every interior element is the sum of both parents:

$$
\begin{array}{c}
\binom{n}{k} \ \binom{n}{k+1} \\
\ldots\ldots \binom{n+1}{k+1} \ldots\ldots\ldots
\end{array}
$$

4.7.4 Matrix Operations

We define on matrices (see p. 95) over $\mathbb{R}$ the following arithmetic operations.

Definition 60 (Matrices over the Reals) The domain of *real matrices* of dimension $m \times n$ is defined as follows:

$$\mathbb{M}_{m,n} := \mathbb{N}_m \times \mathbb{N}_n \to \mathbb{R}.$$

Definition 61 (Matrix Operations) For every $m \in \mathbb{N}$, $n \in \mathbb{N}$, and $p \in \mathbb{N}$ we define the following operations on real matrices.

Constants

null matrix (Nullmatrix):

$$
\begin{aligned}
&0 : \mathbb{N}_m \times \mathbb{N}_n \to \mathbb{R}, \\
&0_{i,j} := 0.
\end{aligned}
$$

unity matrix (Einheitsmatrix):

$$
\begin{aligned}
&1 : \mathbb{N}_n \times \mathbb{N}_n \to \mathbb{R}, \\
&1_{i,j} := \textbf{if } i = j \textbf{ then } 1 \textbf{ else } 0.
\end{aligned}
$$

Addition

$$
\begin{aligned}
&+ : \mathbb{M}_{m,n} \times \mathbb{M}_{m,n} \to \mathbb{M}_{m,n} \\
&A + B := \textbf{ such } C \in \mathbb{M}_{m,n} : \\
&\quad \forall i \in \mathbb{N}_m, j \in \mathbb{N}_n : C_{i,j} = A_{i,j} + B_{i,j}.
\end{aligned}
$$

short: $(A+B)_{i,j} := A_{i,j} + B_{i,j}$.

Scalar Product

$$
\begin{array}{l}
* : \mathbb{R} \times \mathbb{M}_{m,n} \to \mathbb{M}_{m,n} \\
c * A := \ \textbf{such}\ C \in \mathbb{M}_{m,n} : \\
\quad \forall i \in \mathbb{N}_m, j \in \mathbb{N}_n : C_{i,j} = c * A_{i,j}.
\end{array}
$$

short: $(c * A)_{i,j} := c * A_{i,j}$.

Matrix Product

$$
\begin{array}{l}
+ : \mathbb{M}_{m,n} \times \mathbb{M}_{n,p} \to \mathbb{M}_{m,p} \\
A * B := \ \textbf{such}\ C \in \mathbb{M}_{m,p} : \\
\quad \forall i \in \mathbb{N}_m, j \in \mathbb{N}_p : C_{i,j} = \sum_{0 \le k < n} A_{i,k} * B_{k,j}.
\end{array}
$$

short: $(A + B)_{i,j} := \sum_{0 \le k < n} A_{i,k} * B_{k,j}$.

Determinant

$$
\begin{array}{l}
|\,.\,| : \mathbb{M}_{n,n} \to \mathbb{R}, \\
\text{if n } = 1 : \\
\quad |A| := A_{0,0}, \\
\text{if n } > 1 : \\
\quad |A| := \sum_{0 \le j < n} A_{0,j} * (-1)^j * |B| \\
\qquad \textbf{where}\ B = \ \textbf{such}\ B \in \mathbb{M}_{n-1,n-1} : \\
\qquad\quad \forall k \in \mathbb{N}_{n-1}, l \in \mathbb{N}_{n-1} : \\
\qquad\qquad B_{k,l} = (\textbf{if}\ l < j\ \textbf{then}\ A_{k+1,l}\ \textbf{else}\ A_{k+1,l+1}).
\end{array}
$$

The determinant of an $n \times n$ matrix A is defined by the determinants of a number of $(n-1) \times (n-1)$ matrices B that are constructed from A by deleting the first row and some column j.

Example

$$
\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix} = 1 * \begin{vmatrix} 5 & 6 \\ 8 & 9 \end{vmatrix} - 2 * \begin{vmatrix} 4 & 6 \\ 7 & 9 \end{vmatrix} + 3 * \begin{vmatrix} 4 & 5 \\ 7 & 8 \end{vmatrix}
$$

$$
\begin{vmatrix} 5 & 6 \\ 8 & 9 \end{vmatrix} = 5 * 9 - 6 * 8 = -3.
$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} * \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} = \begin{bmatrix} 1a+2c+3e & 1b+2d+3f \\ 4a+5c+6e & 4b+5d+6f \end{bmatrix}$$

The matrix operations satisfy many equations that also hold for numbers, e.g., $A * (B + C) = A * B + A * C$. However, unlike in number domains, matrix multiplication is *not* commutative.

Example

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix} = \begin{bmatrix} 7 & 9 \\ 17 & 19 \end{bmatrix}$$

$$\begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix} * \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 6 & 10 \\ 14 & 20 \end{bmatrix}$$

4.7.5 Polynomials

We are used to think of a polynomial $x + 1$ as a term with a variable x; the term can be manipulated according to various rules that preserve the value of the polynomial for any substitution of the variable. However, we need not introduce concepts like "terms" and "variables" in order to deal with polynomials; we can construct them directly from the basic material (functions and numbers) that we have introduced in the previous sections.

Definition 62 (Polynomials over the Reals) A *polynomial (Polynom)* over the reals is an infinite sequence of real numbers, the *coefficients (Koeffizienten)*, of which only finitely many are different from 0:

$$p \text{ is polynomial } :\Leftrightarrow p : \mathbb{N} \to \mathbb{R} \wedge (\exists k \in \mathbb{N} : \forall i \geq k : p_i = 0).$$

The *degree (Grad)* of a polynomial is zero, if all coefficients are zero; otherwise, it is the index of the largest non-zero coeffient:

$$\begin{array}{l} \deg(p) := \\ \quad \textbf{if } \forall i \in \mathbb{N} : p_i = 0 \\ \qquad \textbf{then } 0 \\ \qquad \textbf{else } (\textbf{such } k \in \mathbb{N} : p_k \neq 0 \wedge (\forall i > k : p_i = 0)). \end{array}$$

We denote by "Poly" the set of all polynomials over the reals:

$$\text{Poly} := \{p \in \mathbb{N} \to \mathbb{R} : p \text{ is polynomial}\}.$$

We then can define the following "number-like" operations on polynomials.

Definition 63 (Polynomial Operations)

$$\cdot_{\text{Poly}} : \mathbb{R} \to \text{Poly}$$
$$c_{\text{Poly}} := \textbf{such } p \in \text{Poly} : p_0 = c \wedge (\forall i > 0 : p_i = 0)$$

$$\mathsf{x} := \textbf{such } p \in \text{Poly} : p_0 = 0 \wedge p_1 = 1 \wedge (\forall i > 1 : p_i = 0)$$

$$+ : \text{Poly} \times \text{Poly} \to \text{Poly}$$
$$(p+q)_i := p_i + q_i$$

$$- : \text{Poly} \times \text{Poly} \to \text{Poly}$$
$$(p-q)_i := p_i - q_i$$

$$- : \text{Poly} \to \text{Poly}$$
$$(-p)_i := -(p_i)$$

$$* : \text{Poly} \times \text{Poly} \to \text{Poly}$$
$$(p*q)_i := \sum_{0 \le j \le i} p_j * q_{i-j}$$

One can show that the polynomial operations satisfy the laws that also hold for the corresponding operations on numbers, e.g., $p*1_{\text{Poly}} = p$ and $p*q = q*p$.

The first operation maps a real number c in a unique way to a polynomial c_{Poly}. Usually we write just c instead of c_{Poly}, when the meaning is clear from the context, e.g., if p is a polynomial, then $p+1$ actually denotes $p+1_{\text{Poly}}$.

Example We have the following polynomial identities:

$$\begin{array}{rcl}
3_{\text{Poly}} & = & [3,0,0,0,0,\ldots] \\
\mathsf{x} & = & [0,1,0,0,0,\ldots] \\
\mathsf{x}+3 & = & [3,1,0,0,0,\ldots] \\
\mathsf{x}*\mathsf{x} & = & [0,0,1,0,0,\ldots] \\
\mathsf{x}*\mathsf{x}+2*\mathsf{x}+1 & = & [1,2,1,0,0,\ldots] \\
(\mathsf{x}+1)*(\mathsf{x}+2) & = & [2,3,1,0,0,\ldots]
\end{array}$$

The polynomial operations are closely related to their counterparts on $\mathbb{R}$.

Proposition 62 (Constant Polynomials) For all real numbers a and b we have

$$\begin{aligned} a_{\text{Poly}} + b_{\text{Poly}} &= (a +_{\mathbb{R}} b)_{\text{Poly}}, \\ a_{\text{Poly}} - b_{\text{Poly}} &= (a -_{\mathbb{R}} b)_{\text{Poly}}, \\ -a_{\text{Poly}} &= (-_{\mathbb{R}} a)_{\text{Poly}}, \\ a_{\text{Poly}} * b_{\text{Poly}} &= (a *_{\mathbb{R}} b)_{\text{Poly}}. \end{aligned}$$

In other words, a property like $1 + 1 = 2$ also holds for polynomials 1_{Poly} and 2_{Poly} and + interpreted as the polynomial addition. Above law says that $\mathbb{R}$ can be embedded into "Poly" in a sense that is discussed in Section 6.3.

Definition 63 answers the question what a polynomial $\mathsf{x} + 1$ is: it is the sum of two polynomials x and $\mathsf{1}$, i.e., the "variable" x in a polynomial is nothing but a particular (*constant*) polynomial! However, the view of x as a variable that can be substituted by a value is provided by the following operation.

Definition 64 (Polynomial Evaluation)

$$\begin{aligned} &[\,] : \text{Poly} \times \mathbb{R} \to \mathbb{R} \\ &p[a] := \textstyle\sum_{0 \le i \le \deg(p)} p_i * a^i. \end{aligned}$$

Example Let $p := 2 + 3 * x + 4 * x * x$, i.e., $p = [2, 3, 4, 0, 0, 0, 0, \ldots]$. Then

$$p[5] = 2 * 5^0 + 3 * 5^1 + 4 * 5^2 = 117.$$

The evaluation operation satisfies the following laws.

Proposition 63 (Evaluation Laws) For all polynomials p and q and all real numbers c and a, we have:

$$\begin{aligned} c_{\text{Poly}}[a] &= c_{\text{Poly}}, \\ \mathsf{x}[a] &= a_{\text{Poly}}, \\ (p + q)[a] &= p[a] + q[a], \\ (p * q)[a] &= p[a] * q[a]. \end{aligned}$$

When evaluating a polynomial p on a real number a, we thus just need to substitute a_{Poly} for every occurence of x in the term describing p and then evaluate polynomial addition and multiplication. We can do this by simply using the rules for addition and multiplication over the real numbers.

Example We have the following polynomial identities:

$$\begin{array}{rcl}(\mathsf{x}+1)[2] & = & 2+1\ (=3_{\text{Poly}}) \\ (\mathsf{x}*\mathsf{x})[2] & = & 2*2\ (=4_{\text{Poly}}) \\ (\mathsf{x}*\mathsf{x}+2*\mathsf{x}+1)[3] & = & 3*3+2*3+1\ (=16_{\text{Poly}})\end{array}$$

In mathematical practice, we usually do not work with a fixed "polynomial variable" x. Instead we rather introduce a set $\mathbb{R}[x]$ of polynomials with coefficients in $\mathbb{R}$ (and correspondingly for other coefficent domains) such that the constant x denotes the polynomial $[0,1,0,0,0,\ldots]$.

Example $2y+1$ is an element of $\mathbb{R}[y]$ denoting the polynomial

$$[1,2,0,0,0,\ldots].$$

A multivariate polynomial is interpreted as a univariate polynomial whose coefficents are themselves polynomials.

Example The bivariate polynomial

$$x^2y+xy^2+3x+2y+1$$

which can be written as

$$y*x^2+(y^2+3)*x+(2y+1)$$

is an element of $\mathbb{R}[y,x]$, i.e., $(\mathbb{R}[y])[x]$:

$$[2y+1,y^2+3,y,0,0,0,\ldots].$$

The coefficient $2y+1$ is an element of $\mathbb{R}[y]$:

$$[1,2,0,0,0,\ldots].$$

Chapter 5

Induction and Recursion

The domain of natural numbers has a property (implied by the first two Peano axioms) that allows us to *define functions and predicates inductively*, i.e., by sets of equations respectively equivalences with a particular structure. Likewise, the third Peano axiom give us *induction* as a technique for proving properties about natural numbers. In this chapter, we investigate these two aspects of induction, defining and proving, and demonstrate their relationship by proving properties of inductively/recursively defined functions. We also generalize the concept of inductive definitions and proofs from natural numbers to *inductively defined sets* which are prominent in computer science.

5.1 Inductive Definitions

We have introduced in Section 4.1 multiplication over $\mathbb{N}$ by the following recursive definition (with termination function $r(x, y) := y$)

$$\begin{array}{l} * : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \\ x * y := \mathbf{if}\ y = 0\ \mathbf{then}\ 0\ \mathbf{else}\ x + (x * y^-) \end{array}$$

from which we can infer the following knowledge (Proposition 39 on page 102)

$$\begin{array}{rcl} x * 0 & = & 0, \\ x * y' & = & x + (x * y). \end{array}$$

or in another form

$$\begin{array}{rcl} x * 0 & = & 0, \\ x * (y + 1) & = & x + (x * y). \end{array}$$

Actually also the inverse is true: this pair of equations uniquely determines the function $*$: the first Peano law implies that the left hand side of only one equation "matches" a particular application $a * b$ (because b is either 0 or $y + 1$ for some y but not both) and the second Peano law implies that argument b determines the value of parameter y uniquely.

We can therefore write the definition also in the form

$$
\begin{array}{l}
* : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\
x * 0 := 0, \\
x * (y+1) := x + (x * y).
\end{array}
$$

The syntactic properties of these equations guarantee that the corresponding function is well defined.

Definition 65 (Inductive Function Definition) An *inductive definition (induktive Definition)* over $\mathbb{N}$ of an n-ary function f is a pair of equations

$$
\begin{array}{rcl}
f(x_0, \ldots, 0, \ldots, x_{n-1}) & := & T_{\mathrm{b}}, \\
f(x_0, \ldots, x_i + 1, \ldots, x_{n-1}) & := & T_{\mathrm{r}}
\end{array}
$$

where T_{b} (whose free variables are among $x_0, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{n-1}$) is the *base term* in which f does not occur and where every application of f in the *recursion term* T_{r} (whose free variables are among $x_0, \ldots, x_{n-1}$) has the form

$$f(T_0, \ldots, x_i, \ldots, T_{n-1})$$

for some terms $T_0, \ldots, T_{n-1}$. We say that the induction *runs over* x_i.

If there exist $A_0, \ldots, A_{i-1}, A_{i+1}, \ldots, A_{n-1}, B$ with

$$
\begin{array}{l}
\forall x_0 \in A_0, \ldots, x_{i-1} \in A_{i-1}, x_i \in \mathbb{N}, x_{i+1} \in A_{i+1}, \ldots, x_{n-1} \in A_{n-1} : \\
\quad T_{\mathrm{b}} \in B \wedge T_{\mathrm{r}} \in B
\end{array}
$$

then above definition introduces a function

$$f : A_0 \times \ldots \times A_{i-1} \times \mathbb{N} \times A_{i+1} \times \ldots \times A_{n-1} \to B$$

such that the following holds:

$$
\begin{array}{l}
\forall x_0 \in A_0, \ldots, x_{i-1} \in A_{i-1}, x_i \in \mathbb{N}, x_{i+1} \in A_{i+1}, \ldots, x_{n-1} \in A_{n-1} : \\
\quad f(x_0, \ldots, 0, \ldots, x_{n-1}) = T_{\mathrm{b}} \wedge \\
\quad f(x_0, \ldots, x_i + 1, \ldots, x_{n-1}) = T_{\mathrm{r}}.
\end{array}
$$

A function introduced by an inductive definition is uniquely defined.

One can generalize inductive definitions in various ways. For instance, it is not necessary that the induction term is decreased by exactly 1. We may use any non-zero decrement value as long as every possible case is covered by a corresponding equation. For instance, in the definition of the *Fibonacci Numbers (Fibonacci-Zahlen)*

$$\begin{aligned}
&\mathrm{fib}(0) := 1,\\
&\mathrm{fib}(1) := 1,\\
&\mathrm{fib}(x+2) := \mathrm{fib}(x) + \mathrm{fib}(x+1)
\end{aligned}$$

we have two base equations because in the last equation the induction term x is decreased by 2.

Furthermore, we may let the induction not just run over a single parameter but over a *combination* of parameters as in

$$\begin{aligned}
&f(0,0) := 0,\\
&f(x+1,0) := 1 + f(x,0),\\
&f(x,y+1) := 1 + f(x,y).
\end{aligned}$$

which defines a function that computes (in an artificially complex way) the sum of both arguments x and y. In this case, the induction is guarded by the *termination term* $x + y$ whose value is decreased by one in every recursive application of f.

If the induction runs over multiple arguments, we have to take special care that every possible argument pattern is covered by an equation. For instance, the definition

$$\begin{aligned}
&f(0,0) := 0,\\
&f(x+1,0) := 1 + f(x,0),\\
&f(0,y+1) := 1 + f(0,y),\\
&f(x+1,y+1) := 2 + f(x,y),
\end{aligned}$$

uses four equations to cover all possible cases.

The more complicated the recursive structure of a function definition is, the more important it becomes to show that a set of equations indeed defines a total function. This can be done by giving a termination term or, more general, by constructing a well-founded ordering (see page 298).

Example *Ackerman's function* is defined as follows:

$$
\begin{aligned}
&A(0, y) := y + 1,\\
&A(x+1, 0) := A(x, 1),\\
&A(x+1, y+1) := A(x, A(x+1, y)).
\end{aligned}
$$

We have $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ because we can construct the well-founded *lexicographic ordering (lexikographische Ordnung)* on the function arguments

$$x \prec y :\Leftrightarrow x_0 < y_0 \vee (x_0 = x_0 \wedge x_1 < y_1)$$

(e.g. $\langle 0, 0\rangle \prec \langle 0, 1\rangle$ and $\langle 0, 1\rangle \prec \langle 1, 0\rangle$) and show that the arguments decrease in every recursive application with respect to this ordering, i.e., for the three recursive applications in the equations above:

1. $\langle x, 1\rangle \prec \langle x+1, 0\rangle$,
2. $\langle x+1, y\rangle \prec \langle x+1, y+1\rangle$,
3. $\langle x, A(x+1, y)\rangle \prec \langle x+1, y+1\rangle$.

All three statements are true by definition of $\prec$.

Also predicates can be defined inductively.

Definition 66 (Inductive Predicate Definition) An *inductive definition (Induktive Definition)* over $\mathbb{N}$ of an n-ary predicate p is a pair of equivalences

$$
\begin{aligned}
p(x_0, \ldots, 0, \ldots, x_{n-1}) &\quad :\Leftrightarrow \quad F_{\mathrm{b}},\\
p(x_0, \ldots, x_i + 1, \ldots, x_{n-1}) &\quad :\Leftrightarrow \quad F_{\mathrm{r}}
\end{aligned}
$$

where F_{b} (whose free variables are among $x_0, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{n-1}$) is the *base formula* in which f does not occur and where every application of f in the *recursion formula* F_{r} (whose free variables are among $x_0, \ldots, x_{n-1}$) has the form

$$p(T_0, \ldots, x_i, \ldots, T_{n-1})$$

for some terms $T_0, \ldots, T_{n-1}$. We say that the induction *runs over* x_i.
Above definition defines a predicate

$$p \subseteq A_0 \times \ldots \times A_{i-1} \times \mathbb{N} \times A_{i+1} \times \ldots \times A_{n-1}.$$

such that the following holds:

$$\begin{aligned}
&\forall x_0 \in A_0, \ldots, x_{i-1} \in A_{i-1}, x_i \in \mathbb{N}, x_{i+1} \in A_{i+1}, \ldots, x_{n-1} \in A_{n-1} : \\
&\quad (p(x_0, \ldots, 0, \ldots, x_{n-1}) \Leftrightarrow F_{\mathrm{b}}) \wedge \\
&\quad (p(x_0, \ldots, x_i + 1, \ldots, x_{n-1}) \Leftrightarrow F_{\mathrm{r}}).
\end{aligned}$$

A predicate introduced by an inductive definition is uniquely defined.

Example We can introduce the predicate $\mathrm{iseven}(x) :\Leftrightarrow 2|x$ also by the following recursive definition:

$$\begin{aligned}
&\mathrm{iseven}(0) :\Leftrightarrow \mathrm{T}, \\
&\mathrm{iseven}(x+1) :\Leftrightarrow \neg\mathrm{iseven}(x).
\end{aligned}$$

We can define the same predicate also in the following way with two base cases and an induction term x that is decreased by 2:

$$\begin{aligned}
&\mathrm{iseven}(0) :\Leftrightarrow \mathrm{T}, \\
&\mathrm{iseven}(1) :\Leftrightarrow \mathrm{F}, \\
&\mathrm{iseven}(x+2) :\Leftrightarrow \mathrm{iseven}(x).
\end{aligned}$$

5.2 Induction as a Proof Technique

The Peano Axioms (page 99) include for every formula F an axiom

$$(F[x \leftarrow 0] \wedge (\forall x \in \mathbb{N} : F \Rightarrow F[x \leftarrow x+1])) \Rightarrow \forall x \in \mathbb{N} : F.$$

We may apply this axiom for *proving* properties of formulas involving natural numbers.

Proposition 64 (Mathematical Induction) In order to prove

$$\forall x \in \mathbb{N} : F,$$

it suffices to prove

1. $F[x \leftarrow 0]$,
2. $(\forall x \in \mathbb{N} : F \Rightarrow F[x \leftarrow x + 1])$.

In natural language, this proof strategy is usually indicated by the following triple of statements:

1. **Induction Base:** We show $F[x \leftarrow 0]$.
2. **Induction Hypothesis:** We take arbitrary $x \in \mathbb{N}$ and assume F.
3. **Induction Step:** We show $F[x \leftarrow x + 1]$.

Several examples are given below.

Example We prove

$$\forall n \in \mathbb{N} : n < 2^n$$

by induction on n.

The induction base holds because $0 < 1 = 2^0$.

Now we take arbitrary $n \in \mathbb{N}$ and assume

(1) $n < 2^n$.

We have to show

(2) $n + 1 < 2^{n+1}$.

By (1) we have

(3) $n + 1 < 2^n + 1$

and therefore

(4) $n + 1 < 2^n + 1 \leq 2^n + 2^n = 2 * 2^n = 2^n + 1$

which implies (2).

(The derivation of (2) from (1) and $1 \leq 2^n$ are knowledge that we assume granted in this proof).

Example We prove

$$\forall n \in \mathbb{N} : 3 | n^3 + 2n$$

by induction on n.

The induction base holds because $3|0$ and $0 = 0^3 + 2 * 0$.

We take arbitrary $n \in \mathbb{N}$ and assume

(1) $3|n^3 + 2n$.

We have to show

(2) $3|(n+1)^3 + 2(n+1)$.

By (1) and definition of $|$ we have some $a \in \mathbb{N}$ such that

(3) $3a = n^3 + 2n$.

We therefore have

$$\begin{array}{rcl} (n+1)^3 + 2(n+1) & = & \\ (n^3 + 3n^2 + 3n + 1) + (2n + 2) & = & \\ (n^3 + 2n) + (3n^2 + 3n + 3) & = & (3) \\ 3a + 3(n^2 + n + 1) & = & \\ 3(a + n^2 + n + 1) & & \end{array}$$

which implies (2) by definition of $|$.

(In this proof we assume the computing rules in $\mathbb{N}$ as granted).

Example We prove

$$\forall n \in \mathbb{N} : \sum_{1 \leq i \leq n} i = \frac{(n+1)n}{2}$$

by induction on n.

The induction base holds because

$$\sum_{1\leq i\leq 0} i = 0 = \frac{(0+1)*0}{2}.$$

We take arbitrary $n \in \mathbb{N}$ and assume

$$(1)\quad \sum_{1\leq i\leq n} i = \frac{(n+1)n}{2}.$$

We have to show

$$(2)\quad \sum_{1\leq i\leq n+1} i = \frac{((n+1)+1)(n+1)}{2}.$$

We have

$$\begin{array}{rcl}
\sum_{1\leq i\leq n+1} i & = & (\text{definition } \sum)\\
\sum_{1\leq i\leq n} i + (n+1) & = & (1)\\
\frac{(n+1)n}{2} + (n+1) & = & \\
\frac{(n+1)n+2(n+1)}{2} & = & \\
\frac{(n+1)(n+2)}{2} & = & \\
\frac{(n+1)((n+1)+1)}{2}. & &
\end{array}$$

which implies (2).

(In this proof we assume the computing rules in $\mathbb{Q}$ as granted).

In particular, we can prove by induction the "computing laws" stated in Section 4.1 on page 103.

Example We prove

$$\forall x \in \mathbb{N}, y \in \mathbb{N}, z \in \mathbb{N} : x + (y + z) = (x + y) + z.$$

We take arbitrary $x \in \mathbb{N}$ and $y \in \mathbb{N}$ and prove

$$\forall z \in \mathbb{N} : x + (y + z) = (x + y) + z$$

by induction on z.

We have by definition of $+$

$$x + (y + 0) = x + y = (x + y) + 0$$

and thus the induction base.

We assume

$$(1)\ x + (y + z) = (x + y) + z$$

and show

$$(2)\ x + (y + (z + 1)) = (x + y) + (z + 1).$$

We have

$$\begin{array}{lcl} (x+y)+(z+1) & = & \text{(definition +)} \\ ((x+y)+z)+1 & = & (1) \\ (x+(y+z))+1 & = & \text{(definition +)} \\ x+((y+z)+1) & = & \text{(definition +)} \\ x+(y+(z+1)) & & \end{array}$$

which implies (2).

Similar to inductive definitions with termination terms that are decreased by values greater than one, proofs by induction may use a more general induction rule which allows to rely on the truth of the formula to be proved for all predecessors of the current number.

Proposition 65 (Complete Induction) In order to prove

$$\forall x \in \mathbb{N} : F$$

it suffices to prove

$$(\forall x \in \mathbb{N} : (\forall n < x : F[x \leftarrow n]) \Rightarrow F).$$

In complete induction, we use a generalized induction hypothesis in which we assume that F holds for all previous values of x (not just for $x - 1$). There is no need for a separate induction base, because for $x = 0$ the hypothesis $(\forall n < x : F[x \leftarrow n])$ collapses to T.

The application of this proof strategy is usually indicated as follows:

1. **Induction Hypothesis**. We take arbitrary $x \in \mathbb{N}$ and assume

$$\forall n < x : F[x \leftarrow n].$$

2. **Induction Step**: We show F.

Complete induction can be applied not only in $\mathbb{N}$ but in every domain A with a well-founded ordering $\prec \subseteq A \times A$ (see page 298).

Example We prove that every natural number greater than 1 can be factorized into a sequence of prime numbers, i.e.,

$$\begin{array}{l}\forall n \in \mathbb{N} : n > 1 \Rightarrow \\ \quad (\exists k \in \mathbb{N}, f : \mathbb{N}_k \to \mathbb{N} : n = \prod_{0 \leq i < k} f(i) \wedge \forall i \in \mathbb{N}_k : f(i) \text{ is prime}).\end{array}$$

We proceed by complete induction over n.

We take arbitrary $n \in \mathbb{N}$ and assume

$$\begin{array}{l}(1)\ \forall m < n : m > 1 \Rightarrow \\ \quad (\exists k \in \mathbb{N}, f : \mathbb{N}_k \to \mathbb{N} : m = \prod_{0 \leq i < k} f(i) \wedge \forall i \in \mathbb{N}_k : f(i) \text{ is prime}).\end{array}$$

We have to show

$$\begin{array}{l}n > 1 \Rightarrow \\ \quad (\exists k \in \mathbb{N}, f : \mathbb{N}_k \to \mathbb{N} : n = \prod_{0 \leq i < k} f(i) \wedge \forall i \in \mathbb{N}_k : f(i) \text{ is prime}).\end{array}$$

We assume (3) $n > 1$ and show

$$(4)\ (\exists k \in \mathbb{N}, f : \mathbb{N}_k \to \mathbb{N} : n = \prod_{0 \leq i < k} f(i) \wedge \forall i \in \mathbb{N}_k : f(i) \text{ is prime}).$$

If n is prime, then take $k := 1$ and $f_0 := n$ and we are done.

If n is not prime, then, by definition of primality, there exists some a with (5) $1 < a < n$ and (6) $a|n$. By (6) and definition of $|$, there exists some b such that

$$(7)\ a * b = n.$$

From (5) and (7), we have (8) $1 < b < n$.

By (1) and (5), we have some $k_a \in \mathbb{N}$ and $f_a : \mathbb{N}_{k_a} \to \mathbb{N}$ with

$$(9)\ a = \prod_{0 \leq i < k_a} f_a(i) \land \forall i \in \mathbb{N}_k : f_a(i) \text{ is prime.}$$

Likewise we have by (1) and (8) some $k_b \in \mathbb{N}$ and $f_b : \mathbb{N}_{k_b} \to \mathbb{N}$ with

$$(10)\ b = \prod_{0 \leq i < k_b} f_b(i) \land \forall i \in \mathbb{N}_k : f_b(i) \text{ is prime.}$$

Take $k := k_a + k_b$ and $f(i) :=$ **if** $i < k_a$ **then** $f_a(i)$ **else** $f_b(i)$. By (7), (9), and (10), we have

$$(11)\ n = \prod_{0 \leq i < k} f(i) \land \forall i \in \mathbb{N}_k : f(i) \text{ is prime}$$

which implies (4).

Similar to inductive definitions where the induction term may be a combination of the parameters, proofs by induction may proceed over the value of a term.

Proposition 66 (Induction over Term Values) In order to prove a formula F, it suffices to prove

$$(\forall y \in \mathbb{N} : y = T \Rightarrow F)$$

where y is a variable that does not occur freely in T or F and T is a term that denotes a natural number.

To prove above formula, we may proceed by induction on y (using one of the variants described above).

In particular, in order to prove

$$\forall x_0, \ldots, x_{n-1} : F$$

we may prove

$$(\forall x_0, \ldots, x_{n-1}, y \in \mathbb{N} : y = T \Rightarrow F)$$

where T is a term with free variables $x_0, \ldots, x_{n-1}$. We can then take arbitrary $x_0, \ldots, x_{n-1}$ and prove $(\forall y \in \mathbb{N} : y = T \Rightarrow F)$ by induction on y.

Actually, we do not need an explicite variable y; it suffices to state "we prove by induction on T"

and then proceed with an induction proof on the value of T (which must be a natural number). An example is given in the following section in the verification of Euclid's algorithm.

5.3 Properties of Recursive Definitions

The close relationship between inductive definitions and induction as a proof strategy makes it natural that the same method can be applied to *verify (verifizieren)* the correctness of inductively/recursively defined functions. Given the definition of a function

$$f : A \to B$$

we would like to show that the formula

$$\forall x \in A : O(x, f(x))$$

holds where O is a binary predicate which expresses the desired relationship between input x and output $f(x)$. A more general statement of the problem is given below.

Definition 67 (Specification) For every function $f : A \to B$, a relation $I \subseteq A$ and a relation $O \subseteq A \times B$, we call the formula

$$\forall x : I(x) \Rightarrow O(x, f(x))$$

a *specification (Spezifikation)* of function f with *input condition (Eingabebedingung)* I and *output condition (Ausgabebedingung)* O.

If the domain of a function parameter is $\mathbb{N}$, then the verification may proceed by induction over this parameter.

Example Take the exponentiation function defined on page 106:

$$x^0 := 1,$$
$$x^{n+1} := x * x^n.$$

We want to verify that the expected relationship

$$\forall x, n \in \mathbb{N} : x^n = \prod_{1 \le i \le n} x.$$

is indeed true for above definition.

Take arbitrary x; we proceed by induction over n.

We have $x^0 = 1 = \prod_{1 \le i \le 0} x$ and thus the induction base holds.

We take arbitrary $n \in \mathbb{N}$ and assume

$$(1)\ x^n = \prod_{1 \le i \le n} x.$$

We have to prove

$$(2)\ x^{n+1} = \prod_{1 \le i \le n+1} x.$$

We know

$$\begin{array}{rcl} x^{n+1} & = & \text{(definition exponentiation)} \\ x * x^n & = & (1) \\ x * \prod_{1 \le i \le n} x & = & (\text{definition} \prod) \\ \prod_{1 \le i \le n+1} x & & \end{array}$$

which implies (2).

The usual purpose of verification is to show that a particular definition of a function matches a specification that is defined in a more abstract away; the function definition can be thus considered as an *implementation* of the specification. While the specification may be inconstructive (i.e., it does not directly yield an algorithm that can be implemented on a computer) or constructive but algorithmically inefficient, the implementation is more concrete and/or more efficient.

The proof that an implementation matches the specification typically requires the derivation of additional *mathematical knowledge*; more knowledge yields better algorithms.

Example We have defined the greatest common divisor on page 107 as

$$\gcd(x, y) := \textbf{such } z \in \mathbb{N} : z|x \wedge z|y \wedge (\forall w : (w|x \wedge w|y) \Rightarrow w \le z).$$

The corresponding implementation in the Logic Evaluator

```
fun gcd(x, y) =
  let(m = if(=(x, N0), y, x):
    such(z in nat(N0, m):
      and(divides(z, x), divides(z, y),
        forall(w in nat(+N(z, N1), m):
          or(not(divides(w, x)), not(divides(w, y))))),
      z));
```

is very inefficient because it tests every $z \in \mathbb{N}_m$ (for some m) whether it is a divisor of the arguments and if it is the greatest such divisor.

Now we may realize (as the Greek mathematician Euclid did) that the following property holds (which is proved in Chapter B on page 326):

$$(0)\ \forall m \in \mathbb{N}, n \leq m : \gcd(m, n) = \gcd(m - n, n).$$

Based on this knowledge, we may define the following function

$$\begin{array}{l} \text{Euclid}(m, n) := \\ \quad \textbf{if } m = 0 \textbf{ then } n \\ \quad \textbf{else if } n = 0 \textbf{ then } m \\ \quad \textbf{else if } n \leq m \textbf{ then } \text{Euclid}(m - n, n) \\ \quad \textbf{else } \text{Euclid}(m, n - m). \end{array}$$

which we claim to satisfy the specification

$$\forall m \in \mathbb{N}, n \in \mathbb{N} : (m \neq 0 \lor n \neq 0) \Rightarrow \text{Euclid}(m, n) = \gcd(m, n).$$

First we check that above equation defines a function

$$\text{Euclid} : \text{Nat} \times \text{Nat} \to \mathbb{N}$$

by constructing a termination term $m+n$ (and check that it is appropriately decreased in every recursive application).

Second we verify that this function meets the specification by *complete induction* on the term $m+n$: we take arbitrary $m \in \mathbb{N}$ and $n \in \mathbb{N}$ and assume

$$\begin{array}{l} (1)\ \forall x \in \mathbb{N}, y \in \mathbb{N} : x + y < m + n \Rightarrow \\ \qquad (x \neq 0 \lor y \neq 0) \Rightarrow \text{Euclid}(x, y) = \gcd(x, y). \end{array}$$

We have to prove

(2) $(m \neq 0 \vee n \neq 0) \Rightarrow \text{Euclid}(m,n) = \text{gcd}(m,n)$.

We assume (3) $(m \neq 0 \vee n \neq 0)$ and prove (4) $\text{Euclid}(m,n) = \text{gcd}(m,n)$.

We have, according to the definition of 'Euclid', four cases

1. $m = 0$.
   By (3), we have $n \neq 0$ and, by definition of 'gcd' and 'Euclid',
   $$\text{gcd}(m,n) = n = \text{Euclid}(m,n)$$
   which implies (4).
2. $m \neq 0 \wedge n = 0$.
   We have, by definition of 'gcd' and 'Euclid',
   $$\text{gcd}(m,n) = m = \text{Euclid}(m,n)$$
   which implies (4).
3. $m \neq 0 \wedge n \neq 0 \wedge n \leq m$.
   We know
   $$\begin{array}{rcl} \text{gcd}(m,n) & = & (0) \\ \text{gcd}(m-n,n) & = & (1) \\ \text{Euclid}(m-n,n) & = & (\text{definition Euclid}) \\ \text{Euclid}(m,n) & & \end{array}$$
   which implies (4).
4. $m \neq 0 \wedge n \neq 0 \wedge n \not\leq m$.
   The proof is analogous to the previous case.

Thus we have shown that 'Euclid' indeed computes the 'gcd' (in an algorithmically much more efficient way than suggested by the definition of 'gcd').

From (0) we can derive the even more powerful property

$(0')$ $\forall m \in \mathbb{N}, n \neq 0 : \text{gcd}(m,n) = \text{gcd}(m, m \bmod n)$

which triggers the following recursive function definition

$$\begin{aligned}
&\text{Euclid}'(m, n) := \\
&\quad \textbf{if } m = 0 \textbf{ then } n \\
&\quad \textbf{else if } n = 0 \textbf{ then } m \\
&\quad \textbf{else if } n \leq m \textbf{ then } \text{Euclid}'(m \bmod n, n) \\
&\quad \textbf{else } \text{Euclid}'(m, n \bmod m)
\end{aligned}$$

with induction term $m + n$.

The proof that this (algorithmically even more efficient) function also meets the specification

$$\forall m \in \mathbb{N}, n \in \mathbb{N} : (m \neq 0 \lor n \neq 0) \Rightarrow \text{Euclid}'(m, n) = \gcd(m, n).$$

is similar to the proof given above.

One can give an algorithmically (essentially) equivalent but shorter definition

$$\begin{aligned}
&\text{Euclid}''(m, n) := \\
&\quad \textbf{if } n = 0 \textbf{ then } m \\
&\quad \textbf{else } \text{Euclid}''(n, m \bmod n)
\end{aligned}$$

which however requires the construction of an well-founded ordering (see page 298) of the arguments to show termination (please try).

```
option silent = true;
read natural0;

fun Euclid(m: N, n: N) recursive +(m, n) =
  if(=(m, 0), n,
  if(=(n, 0), m,
  if(<=(m, n), Euclid(m, -(n, m)),
    Euclid(-(m, n), n))));
fun Euclid'(m: N, n: N) recursive +(m, n) =
  if(=(m, 0), n,
  if(=(n, 0), m,
  if(<=(m, n), Euclid'(m, modN(n, m)),
    Euclid'(modN(m, n), n))));

term gcd(112, 96);
> 16.
term Euclid(112, 96);
> 16.
term Euclid'(112, 96);
> 16.
```

```
term gcd(300, 215);
```

A definition of *Euclid's algorithm (Euklidscher Algorithmus)* in the Logic Evaluator is given above. With larger input values, one soon notices the difference to the efficiency of the naive implementation of the gcd.

5.4 Induction on Sets

The concept of induction is not limited to $\mathbb{N}$; it can be applied to any set whose construction is based on a corresponding principle.

Definition 68 (Inductive Set Definition) An inductive definition of a set S is a collection of formulas

$$
\begin{array}{l}
(\forall x_1, \ldots, x_{m_1}, y_1 \in S, \ldots, y_{n_1} \in S : \\
\quad f_1(x_1, \ldots, x_{m_1}, y_1, \ldots, y_{n_1}) \in S) \\
, \ \ldots \ , \\
(\forall x_1, \ldots, x_{m_c}, y_1 \in S, \ldots, y_{n_c} \in S : \\
\quad f_c(x_1, \ldots, x_{m_c}, y_1, \ldots, y_{n_c}) \in S)
\end{array}
$$

with object constant S and function constants $f_1, \ldots, f_c$ which denote the *constructors (Konstruktoren)* of S.

By this definition, S denotes the smallest set on which the conjunction of these formulas holds.

The notion "smallest" used in above definition implies that every element of the defined set is denoted by a constructor term

$$f_i(T_1, \ldots, T_{m_i}, S_1, \ldots, S_{n_i})$$

for some terms $T_1, \ldots, T_{m_i}, S_1, \ldots, S_{n_i}$ where the $S_1, \ldots, S_{n_i}$ are also such constructor terms.

Example

- The set $\mathbb{N}$ is inductively defined by

$$
\begin{array}{l}
0 \in \mathbb{N}, \\
\forall x \in \mathbb{N} : x' \in \mathbb{N}
\end{array}
$$

for some constructors 0 and '. Every element of $\mathbb{N}$ is of the form

$$0'^{\cdots\prime},$$

e.g. $0''''$ is interpreted as the number 4 in $\mathbb{N}$.

- For every set T, the set "List(T)" is defined by

$$\begin{array}{l} \text{nil} \in \text{List}(T), \\ \forall e \in T, l \in \text{List}(T) : \text{cons}(e,l) \in \text{List}(T). \end{array}$$

  for some constructors "nil" and "cons". Every element of List(T) is of the form

$$\text{cons}(e_0, \ldots, \text{cons}(e_{n-1}, \text{nil})),$$

  e.g. "cons(2, cons(3, nil))" is interpreted as the list $[2,3]$ in List($\mathbb{N}$).

- For every set T, the set "Tree(T)" is defined by

$$\begin{array}{l} \text{empty} \in \text{Tree}(T), \\ \forall e \in T, l \in \text{Tree}(T), r \in \text{List}(T) : \text{node}(e,l,r) \in \text{Tree}(T). \end{array}$$

  for some constructors "empty" and "tree". Every element of Tree(T) is of the form

$$\text{node}(n_0, \text{node}(n_{11}, \ldots), \text{node}(n_{21}, \ldots)),$$

  e.g. "node(1, node(2, node(3, empty, empty), node(4, empty, empty)), node(5, empty, empty))" is interpreted as the following tree in Tree($\mathbb{N}$):

$$\begin{array}{ccccc} & & 1 & & \\ & 2 & & & 5 \\ 3 & & 4 & & \end{array}$$

- The set "Term" is defined by

$$\begin{array}{l} 0 \in \text{Term}, \\ 1 \in \text{Term}, \\ \forall x \in \text{Term} : -x \in \text{Term}, \\ \forall x \in \text{Term}, y \in \text{Term} : x + y \in \text{Term}, \\ \forall x \in \text{Term}, y \in \text{Term} : x * y \in \text{Term} \end{array}$$

  for some constructors $0, 1, -, +, *$. An element of "Term" is "$1 + (1 + 0) * 1$".

- The set "Formula" is defined by

$$\begin{aligned}
&\mathrm{T} \in \mathrm{Formula} \\
&\forall x \in \mathrm{Formula} : \mathrm{not}(x) \in \mathrm{Formula}, \\
&\forall x \in \mathrm{Formula}, y \in \mathrm{Formula} : \mathrm{and}(x, y) \in \mathrm{Formula}, \\
&\forall x \in \mathrm{Variable}, y \in \mathrm{Formula} : \mathrm{forall}(x, y) \in \mathrm{Formula}
\end{aligned}$$

  for some constructors "T", "not", "and", "forall". An element of "Formula" is "forall(X, and(T, or(T, F)))" (assuming X ∈ "Variable″).

Inductive set definitions may appear in various notations; e.g., the set "Term" in above example is more conveniently defined as

$$\mathrm{Term} ::= 0 \mid 1 \mid -\mathrm{Term} \mid \mathrm{Term}{+}\mathrm{Term} \mid \mathrm{Term}{*}\mathrm{Term}.$$

in the syntax of *BNF* (*Backus Naur Form*) that is used to describe *grammars (Grammatiken)* of formal languages.

We now constrain our notion of inductive sets such that the elements preserve "knowledge" about their construction.

Definition 69 (Term Algebra) An inductively defined set is a *term algebra (Term Algebra)* if we have for every constructor f of this set

$$\forall x, y : f(x) = f(y) \Rightarrow x = y$$

i.e., f is *injective (injektiv).* Furthermore, for all constructors f and g

$$\forall x, y : f(x) \neq g(y)$$

i.e., different constructors yield different results.

Every element of a term algebra is denoted by *one and only one* constructor term

$$f_i(T_1, \ldots, T_{m_i}, S_1, \ldots, S_{n_i})$$

for some terms $T_1, \ldots, T_{m_i}, S_1, \ldots, S_{n_i}$ where the $S_1, \ldots, S_{n_i}$ are also constructor terms. The name *term algebra* stems from this one to one correspondence between set elements and the terms that denote these elements.

Example Because of the first two Peano axioms (page 99), $\mathbb{N}$ is a term algebra with constructors 0 and $'$.

A nice property of a term algebra is that they allow inductive function and predicate definitions in the style of inductive definitions over $\mathbb{N}$; we do not introduce these definitions formally but give some examples.

Example

- Take the set $\text{List}(T)$ defined in the previous example and assume that it is a term algebra. We define the length of a list as

  $$\begin{aligned}&\text{length} : \text{List}(T) \to \mathbb{N}\\&\text{length}(\text{nil}) := 0\\&\text{length}(\text{cons}(e, l)) := 1 + \text{length}(l).\end{aligned}$$

  Then we have length(cons(1, cons(2, nil))) $= 2$.

- Take the set $\text{List}(T)$ defined in the previous example and assume that it is a term algebra. We define the depth of a tree as

  $$\begin{aligned}&\text{depth} : \text{Tree}(T) \to \mathbb{N}\\&\text{depth}(\text{empty}) := 0\\&\text{depth}(\text{node}(n, l, r)) := 1 + \max(\text{depth}(l), \text{depth}(r)).\end{aligned}$$

  Then we have depth(node(1, empty, node(2, node(3, empty, empty), empty))) $= 3$.

- Take the set Term defined in the previous example and assume that it is a term algebra. We define the value of a term as

  $$\begin{aligned}&\text{value} : \text{Term} \to \mathbb{N}\\&\text{value}(0) := 0_{\mathbb{N}}\\&\text{value}(1) := 1_{\mathbb{N}}\\&\text{value}(-x) := -_{\mathbb{N}}\text{value}(x)\\&\text{value}(x + y) := \text{value}(x) +_{\mathbb{N}} \text{value}(y)\\&\text{value}(x * y) := \text{value}(x) *_{\mathbb{N}} \text{value}(y)\end{aligned}$$

  Then we have $\text{value}(1 + (1 + 0) * 1) = 2$.

On inductively defined sets (not necessarily term algebras) we have the following induction principle.

Proposition 67 (Structural Induction) In order to prove a property

$$\forall x \in S : F$$

for an inductively defined set S, it suffices to prove

$$\begin{aligned}
&\forall x_1, \ldots, x_{m_i}, y_1 \in S, \ldots, y_{n_i} \in S : \\
&\quad (F[x := y_1] \wedge \ldots \wedge F[x := y_{n_i}]) \Rightarrow \\
&\qquad F[x := f_i(x_1, \ldots, x_{m_i}, y_1, \ldots, y_{n_i})]
\end{aligned}$$

for every constructor f_i of S.

Intuitively, we let the induction run over the "structure" of every term

$$f_i(x_1, \ldots, x_{m_i}, y_1, \ldots, y_{n_i}).$$

denoting some element x in S. We assume that F holds for every "S-component" y_j of x and show that F is propagated to x itself.

Example Take the set "List(T)" defined inductively as

$$\begin{aligned}
&\text{nil} \in \text{List}(T), \\
&\forall e \in T, l \in \text{List}(T) : \text{cons}(e, l) \in \text{List}(T).
\end{aligned}$$

We define

$$\begin{aligned}
&\text{append} : \text{List}(T) \times \text{List}(T) \to \text{List}(T) \\
&\text{append}(\text{nil}, y) := y \\
&\text{append}(\text{cons}(e, x), y) := \text{cons}(e, \text{append}(x, y))
\end{aligned}$$

and claim that (for the function "length" defined in the previous example) the following holds:

$$\begin{aligned}
&\forall x \in \text{List}(T), y \in \text{List}(T) : \\
&\quad \text{length}(\text{append}(x, y)) = \text{length}(x) + \text{length}(y).
\end{aligned}$$

We proceed by structural induction on x:

Case $x = \text{nil}$**:** We have to show

$$\begin{array}{l}\forall y \in \text{List}(T): \\ \quad \text{length}(\text{append}(\text{nil}, y)) = \text{length}(\text{nil}) + \text{length}(y).\end{array}$$

Take arbitrary $y \in \text{List}(T)$. We have

$$\begin{array}{rcl}\text{length}(\text{append}(\text{nil}, y)) & = & \text{(definition append)} \\ \text{length}(y) & = & \\ 0 + \text{length}(y) & = & \text{(definition length)} \\ \text{length}(\text{nil}) + \text{length}(y). & & \end{array}$$

Case $x = \text{cons}(e, l)$**:** Take arbitrary $e \in T$ and $l \in \text{List}(T)$.

We assume (induction hypothesis)

$$\begin{array}{l}\forall y \in \text{List}(T): \\ \quad \text{length}(\text{append}(l, y)) = \text{length}(l) + \text{length}(y)\end{array}$$

and have to show

$$\begin{array}{l}\forall y \in \text{List}(T): \\ \quad \text{length}(\text{append}(\text{cons}(e, l), y)) = \text{length}(\text{cons}(e, l)) + \text{length}(y).\end{array}$$

Take arbitrary $y \in \text{List}(T)$. We have

$$\begin{array}{rcl}\text{length}(\text{append}(\text{cons}(e, l), y)) & = & \text{(definition append)} \\ \text{length}(\text{cons}(e, \text{append}(l, y))) & = & \text{(definition length)} \\ 1 + \text{length}(\text{append}(l, y)) & = & \text{(induction hypothesis)} \\ 1 + (\text{length}(l) + \text{length}(y)) & = & \\ (1 + \text{length}(l)) + \text{length}(y) & = & \text{(definition length)} \\ \text{length}(\text{cons}(e, l)) + \text{length}(y). & & \end{array}$$

Several formal concepts in computer science, e.g. datatypes and formal languages, can be reduced to inductively defined sets; the principle of structural induction is therefore of great importance.

Chapter 6

More on Functions

We have modelled in Chapter 3 functions as special relations, i.e., as sets of tuples. In this chapter, we will elaborate this notion and its properties in more detail and describe some important constructions and applications: we will count set elements and compare the sizes of (also infinite) sets, embed domains into other domains such that the properties of their operations are preserved, model numeric sequences and series, discuss special functions over the reals, and finally compare the asymptotic behavior of functions.

6.1 Further Notions

We start by introducing some additional notions that help to describe the "shape" of functions.

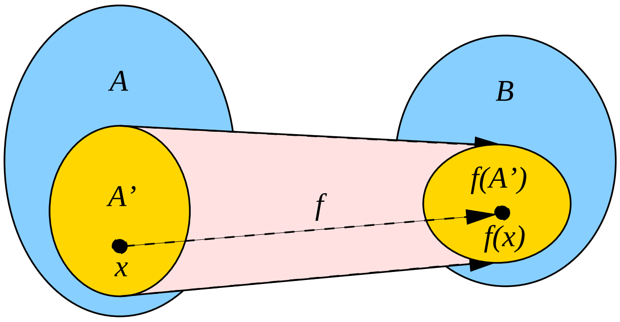
Definition 70 (Image) Let $f : A \to B$, $A' \subseteq A$. The *image (Bild)* of A' at f is the set of all values to which elements of A' are mapped by f:

$$f(A') := \textbf{if } A' \subseteq \text{domain}(f) \textbf{ then } \{f(x) : x \in A'\}.$$

The *inverse image (Urbild)* of B' at f is the set of all elements that are mapped to some elements of B' by f:

$$f^{-1}(B') := \{x \in \text{domain}(f) : f(x) \in B'\}.$$

The set $f(A')$ is visualized in the following figure (where $f^{-1}(f(A')) = A'$):



Logic Evaluator The corresponding definitions are as follows:

```
option silent = true; read relation; read integer;

fun image(f: Function, A': Set) =
  if(isSubset(A', domain(f)), set(x in A': true, apply(f, x)));
fun image^-1(f: Function, B': Set) =
  set(x in domain(f): in(apply(f, x), B'), x);

// all integers with absolute value less than or equal x
fun Z(x: Nat) =
  ++(set(x in nat(0, x): true, I(x, N0)),
     set(x in nat(0, x): true, I(N0, x)));
// the square of all integers x with abs(x) <= n
fun f(n: Nat) = set(x in Z(n): true, tuple(x, *Z(x, x)));

term image(f(6), Z(4));
> {<0, 0>, <1, 0>, <4, 0>, <9, 0>, <16, 0>}.
term image^-1(f(6), Z(4));
> {<0, 0>, <0, 1>, <0, 2>, <1, 0>, <2, 0>}.
term image(f(6), domain(f(6)));
> {<0, 0>, <1, 0>, <4, 0>, <9, 0>, <16, 0>, <25, 0>, <36, 0>}.

term image^-1(f(6), range(f(6)));
```

Next we define a particular classification of functions.

Definition 71 (Function Properties) Let f: $A \to B$. f is *injective (injektiv)* (also called *one-to-one*) if it does not map different arguments to the same result:

$$f : A \overset{\text{injective}}{\longrightarrow} B :\Leftrightarrow$$
$$f : A \to B \wedge (\forall x_0 \in A, x_1 \in A : f(x_0) = f(x_1) \Rightarrow x_0 = x_1).$$

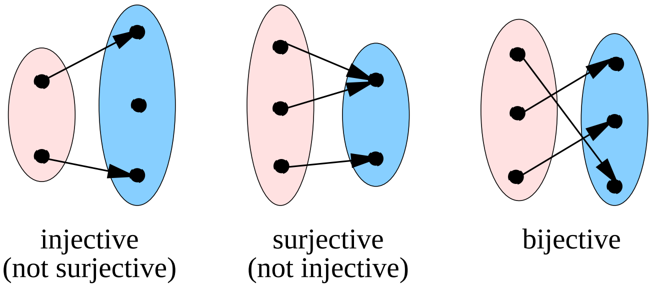
f is *surjective (surjektiv)* (also called *onto*) if it every element of B is hit by some argument:

$$f : A \overset{\text{surjective}}{\longrightarrow} B :\Leftrightarrow$$
$$f : A \to B \wedge (\forall y \in B : \exists x \in A : f(x) = y).$$

f is *bijective (bijektiv)* if it is injective and surjective:

$$f : A \overset{\text{bijective}}{\longrightarrow} B :\Leftrightarrow$$
$$f : A \overset{\text{injective}}{\longrightarrow} B \wedge f : A \overset{\text{surjective}}{\longrightarrow} B.$$

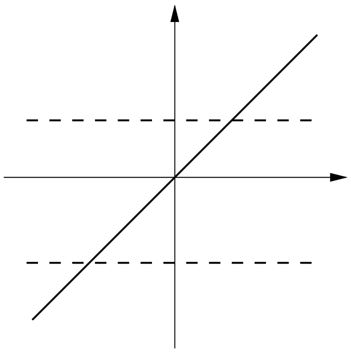
These new notions are visualized in the following figure:



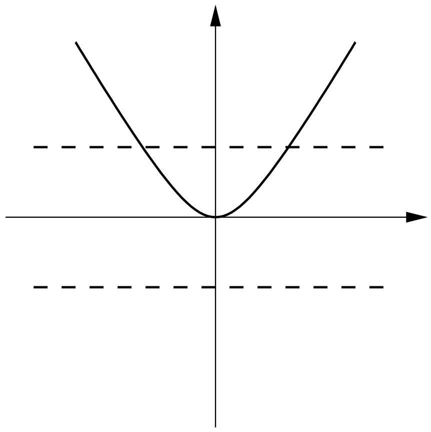
For an injective function, the image of the function domain is at least as large as the domain; for a surjective function, the domain is at least as large as the image. If a function is injective and surjective, the domain and its image thus have the same size.

Example For functions on $\mathbb{R}$, above properties can be demonstrated by considering the number of intersections of the function graph with horizontal lines:

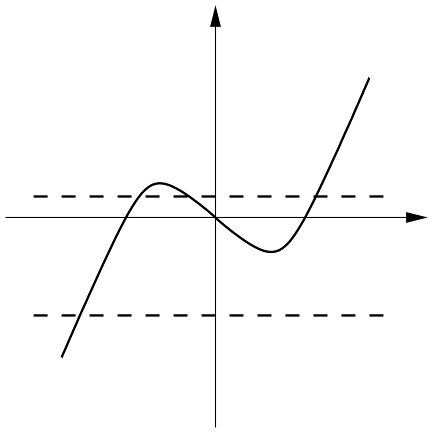
- The *identity function (Identität)* $f(x) := x$ is *bijective* because it intersects every horizontal line exactly once:



- The *square function (Quadratfunktion)* $f(x) := x^2$ is *neither* injective (because the horizontal lines with positive vertical coordinates are intersected twice) *nor* surjective (because the horizontal lines with negative coordinates are not intersected at all).



- The function $f(x) := x^3 - x$ is *surjective* but not injective (every horizontal line is intersected at least once, some horizontal lines are intersected more than once):



Logic Evaluator The new predicates are defined below.

```
option silent = true; read relation; read integer;

pred isInjective(f: Function, A: Set, B: Set) <=>
  and(isFunction(f, A, B),
    forall(x0 in A, x1 in A:
      implies(=(apply(f, x0), apply(f, x1)), =(x0, x1))));
pred isSurjective(f: Function, A: Set, B: Set) <=>
  and(isFunction(f, A, B),
    forall(y in B:
      exists(x in A: =(apply(f, x), y))));
pred isBijective(f: Function, A: Set, B: Set) <=>
  and(isInjective(f, A, B), isSurjective(f, A, B));

// all non-negative integers less than or equal x
fun N(x: Nat) = set(x in nat(0, x): true, I(x, N0));
// all integers with absolute value less than or equal x
fun Z(x: Nat) = ++(N(x), set(y in N(x): true, -Z(y)));

fun F(n: Nat) = set(x in Z(n): true, tuple(x, x));
fun G(n: Nat) = set(x in Z(n): true, tuple(x, ||(x)));
fun H(n: Nat) = set(x in Z(n): true, tuple(x, *Z(x, x)));

formula isBijective(F(3), Z(3), Z(3));
> true.
formula isBijective(F(3), Z(3), Z(4));
> false.
formula isSurjective(G(4), Z(4), Z(4));
> false.
formula isSurjective(G(4), Z(4), N(4));
> true.
```

```
formula isInjective(H(4), Z(4), N(4));
```

Bijectivity is preserved by function composition.

Proposition 68 (Composition of Bijective Functions) The composition of two bijective functions is also bijective:

$$\forall A, B, C, f : A \xrightarrow{\text{bijective}} B, g : B \xrightarrow{\text{bijective}} C :$$
$$f \circ g : A \xrightarrow{\text{bijective}} C.$$

Proof Take arbitrary $f : A \xrightarrow{\text{bijective}} B, g : B \xrightarrow{\text{bijective}} C$.

- We show $f \circ g$ is injective. Take arbitrary $x_0 \in A$ and $x_1 \in A$ with $(f \circ g)(x_0) = (f \circ g)(x_1)$. We have to show $x_0 = x_1$.

  We know, by definition of $\circ$, that $g(f(x_0)) = g(f(x_1))$ and thus, because g is injective, $f(x_0) = f(x_1)$. Since f is injective, we then have $x_0 = x_1$.

- We show $f \circ g$ is surjective. Take arbitrary $z \in C$; we have to find some x such that $(f \circ g)(x) = z$.

  Since g is surjective, we have some $y \in B$ such that $g(y) = z$. Since f is surjective, we have some $x \in A$ such that $f(x) = y$. Thus $(f \circ g)(x) = g(f(x)) = g(y) = z$.

As we have seen in Section 3.4, the inverse of a function is not necessarily a function. However, we have the following result.

Proposition 70 (Inverse of a Function) If a function is injective, its inverse is also a function:

$$
\begin{aligned}
&\forall A, B, f : A \overset{\text{injective}}{\longrightarrow} B : \\
&\quad f^{-1} : B \to A.
\end{aligned}
$$

Proof Take arbitrary $f : A \overset{\text{injective}}{\longrightarrow} B$. We have to show $f^{-1} : B \to A$.

By Proposition 26 on page 81, we have $f^{-1} \subseteq B \times A$. By Definition 28 on page 85, it remains to be shown

$$(\forall x, y_0, y_1 : (\langle x, y_0 \rangle \in f^{-1} \wedge \langle x, y_1 \rangle \in f^{-1}) \Rightarrow y_0 = y_1).$$

Take arbitrary x, y_0, and y_1 and assume

(1) $\langle x, y_0 \rangle \in f^{-1} \wedge \langle x, y_1 \rangle \in f^{-1}$.

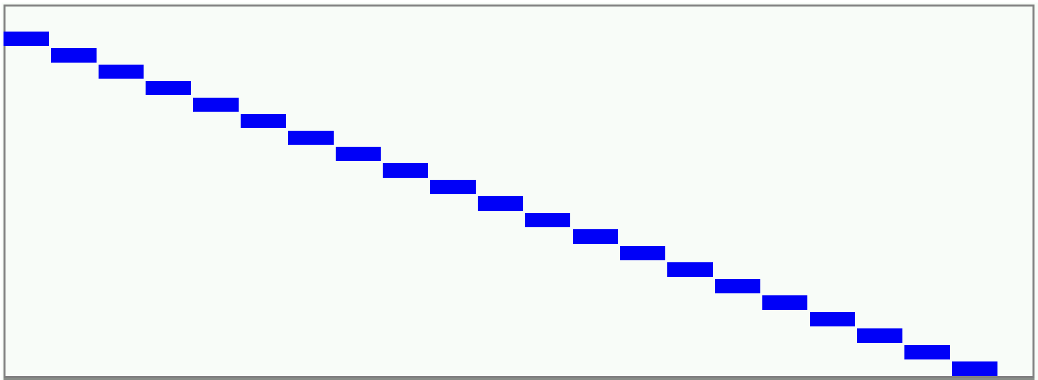
We have to show (2) $y_0 = y_1$.

From (1) and Definition 26 on page 81, we know

(3) $\langle y_0, x \rangle \in f \ \wedge \langle y_1, x \rangle \in f$,

i.e., $f(y_0) = x$ and $f(y_1) = x$. Since f is injective, we thus know (2).

Logic Evaluator The fact that the inverse of an injective function is also a function is demonstrated below at the example of negation on $\mathbb{Z}$(which is injective) and of the absolute value (which is not injective) using the definitions in file `function.txt` (see Appendix C.8).



```
option silent = true; read function; read integer;

fun N(x: Nat) = set(x in nat(0, x): true, I(x, N0));
fun Z(x: Nat) = ++(N(x), set(y in N(x): true, -Z(y)));

// negation and absolute value
fun F(n: Nat) = set(x in Z(n): true, tuple(x, -Z(x)));
fun G(n: Nat) = set(x in Z(n): true, tuple(x, ||(x)));

// shift function points into plot range
fun points(f, n: Nat) = set(t in f: true,
  tuple(.0(||(+Z(.0(t), I(n, 0)))), .0(||(+Z(.1(t), I(n, 0))))));

formula isInjective(F(5), Z(5), Z(5));
> true.
plot points(^-1(F(10)), 10);
> done.

formula isInjective(G(5), Z(5), Z(5));
> false.
```

```
plot points(^-1(G(10)), 10);
```

?

The following function is the result of the composition of a function with its inverse.

Definition 72 (Identity Function) 1_A is the function that maps every $x \in A$ to itself:

$$1_A : A \to A$$
$$1_A(x) := x$$

Proposition 72 (Inverse Function Properties) For every $A, B, f : A \to B$, we have

$$\begin{array}{rcl} f \circ 1_B & = & f \\ 1_A \circ f & = & f. \end{array}$$

If f is injective, then we have

$$f \circ f^{-1} \;=\; 1_A$$

If f is also surjective (i.e., bijective), then we have

$$f^{-1} \circ f \;=\; 1_B.$$

We can consider the set of all functions over some set A as a domain with a binary operation $\circ$ (which is associative), a unary operation $^{-1}$ (the inverse) and a neutral element 1_A.

Logic Evaluator We demonstrate the properties of the identity function listed above.

```
option silent = true; read function; read integer;

fun id(A: Set) = set(x in A: true, tuple(x, x));

fun N(x: Nat) = set(x in nat(0, x): true, I(x, N0));
fun Z(x: Nat) = ++(N(x), set(y in N(x): true, -Z(y)));
fun F(n: Nat) = set(x in Z(n): true, tuple(x, +Z(Z2, x)));

formula isInjective(F(5), Z(5), Z(7));
> true.
formula =(id(Z(5)), o(F(5), ^-1(F(5))));
> true.

formula isSurjective(F(5), Z(5), Z(7));
> false.
```

```
formula =(id(Z(7)), o(^-1(F(5)), F(5)));
```

6.2 Counting Set Elements

The notions introduced in Chapter 3 do not yet allow us to talk about the number of elements in a set. This concept is usually introduced in the following (inconstructive) way:

Definition 73 (Number of Set Elements) A set is *finite (endlich)* if it is empty or there exists a bijection to $\mathbb{N}_n$ for some $n \in \mathbb{N}_{>0}$. We then call 0 respectively n the *size (Größe)* or *cardinality (Kardinalität)* of the set:

$$\begin{aligned} &S \text{ is finite} :\Leftrightarrow S = \emptyset \vee \\ &\quad (\exists n \in \mathbb{N}_{>0}, f : f : \mathbb{N}_n \stackrel{\text{bijective}}{\longrightarrow} S); \\ &|S| := \textbf{if } S = \emptyset \textbf{ then } 0 \textbf{ else} \\ &\quad (\textbf{such } n \in \mathbb{N}_{>0} : \exists f : f : \mathbb{N}_n \stackrel{\text{bijective}}{\longrightarrow} S). \end{aligned}$$

A set is *infinite (unendlich)* if is not finite:

$$S \text{ is infinite } :\Leftrightarrow \neg S \text{ is finite.}$$

Sometimes the set size $|S|$ is also denoted as $\#S$. Please note that $|S|$ is only defined, if S is finite. In this case, however, it is uniquely defined because the number of elements is independent of the order in which we count them, i.e., independent of the particular bijection chosen.

Proposition 73 (Unicity of Bijection) If S is not empty and both $f : \mathbb{N}_n \rightarrow S$ and $g : \mathbb{N}_m \rightarrow S$ are bijections, then $n = m$:

$$\begin{aligned} &(\forall S \neq \emptyset, n \in \mathbb{N}, m \in \mathbb{N}, f : \mathbb{N}_n \stackrel{\text{bijective}}{\longrightarrow} S, g : \mathbb{N}_m \stackrel{\text{bijective}}{\longrightarrow} S : \\ &\quad m = n). \end{aligned}$$

Proof Take arbitrary $S \neq \emptyset$, $n \in \mathbb{N}$, $m \in \mathbb{N}$, $f : \mathbb{N}_n \stackrel{\text{bijective}}{\longrightarrow} S$, $g : \mathbb{N}_m \stackrel{\text{bijective}}{\longrightarrow} S$ and assume $m \neq n$. We show a contradiction.

Assume $m < n$ (the case $m > n$ proceeds analogously). We know $f \circ g^{-1} : \mathbb{N}_n \stackrel{\text{bijective}}{\longrightarrow} \mathbb{N}_m$. However, since $\mathbb{N}_n$ has n elements, $\mathbb{N}_m$ has m elements, and

$m < n$, $f \circ g^{-1}$ cannot be injective (*pigeonhole principle*, a fact that has to be proved separately).

Example

- The set $S := \{0, 2, 4\}$ is finite; its size is 3 because we can define a function $f : \mathbb{N}_3 \xrightarrow{\text{bijective}} S$ as

$$\begin{array}{rcl} f(0) & := & 0 \\ f(1) & := & 2 \\ f(2) & := & 4 \end{array}$$

  i.e., $f = [0, 2, 4]$. The length of f is the same as the length of $[0, 4, 2]$, $[4, 2, 0]$ or of any other bijection to S.

- The set $\mathbb{N}$ is infinite. If it were finite, we had some $n \in \mathbb{N}$ and some $f : \mathbb{N}_n \xrightarrow{\text{bijective}} \mathbb{N}$. Take $k := 1 + \max\{f(i) : i \in \mathbb{N}_n\}$. Then, by construction, $k \in \mathbb{N}$ but $\forall i \in \mathbb{N}_n : f(i) \neq k$, i.e., f is not surjective on $\mathbb{N}$.

A constructive way to determine the number of elements of a set is shown by the following recursive definition:

$$\begin{array}{l} \text{size}(S) := \\ \quad \textbf{if } S = \emptyset \textbf{ then } 0 \\ \quad \textbf{else let } e = (\textbf{such } x : x \in S) : \\ \qquad 1 + \text{size}(S - e) \end{array}$$

We then have $|S| = \text{size}(S)$, for every finite set S. However, to show that above recursive definition indeed defines a function itself relies on the termination term $|S|$: therefore we give a constructive definition in terms of set reduction (see page 304):

$$\begin{array}{l} |S| := \text{reduce}(\text{count}, S, 0); \\ \text{count}(e, i) := i + 1. \end{array}$$

Logic Evaluator The constructive definitions introduced above can be implemented as follows:

```
option silent = true; read set;

fun count(e, i: Nat) = +(i, 1);
fun #(S: Set) = reduce(count, S, 0);

fun size(S: Set) recursive #(S) =
  if(=(S, {}), 0,
    let(e = such(x in S: true, x): +(1, size(--(S, {}(e))))));

fun S(n) = set(x in nat(1, n): true, x);

term #(S(10));
> 10.
term size(S(10));
> 10.
```

```
term size(--(S(12), S(5)));
```

We sometimes count the number of values for which a proposition holds.

Definition 74 (Number Quantifier) For every variable x and formula F, the phrase

$$\#x : F$$

is a term where x is bound and whose value equals

$$|\{x : F\}|.$$

Please note that the value of such a "number" term is only defined if the base formula is true for a finite number of assignments for the bound variable.

We state the following facts (whose proofs can be easily given by induction).

Proposition 75 (Set Sizes) If A and B are disjoint with sizes m and n, respectively, then the size of their union is $m + n$:

$$\begin{array}{l}\forall A, B, m \in \mathbb{N}, n \in \mathbb{N} : \\ \quad (A \cap B = \emptyset \wedge |A| = m \wedge |B| = n) \Rightarrow |A \cup B| = m + n.\end{array}$$

The size of the Cartesian product of two sets is the product of their sizes:

$$\begin{array}{l}\forall A, B, m \in \mathbb{N}, n \in \mathbb{N} : \\ \quad (|A| = m \wedge |B| = n) \Rightarrow |A \times B| = m * n.\end{array}$$

If A and B have size m and n, respectively, then the size of the set of functions from A to B is n^m:

$$\begin{array}{l}\forall A, B, m \in \mathbb{N}, n \in \mathbb{N} : \\ \quad (|A| = m \wedge |B| = n) \Rightarrow |A \rightarrow B| = n^m.\end{array}$$

If A is of size n, then A has 2^n subsets:

$$\begin{array}{l}\forall A, n \in \mathbb{N} : \\ \quad |A| = n \Rightarrow |\{S \in \mathbb{P}(S) : \mathrm{T}\}| = 2^n.\end{array}$$

It may seem surprising, but we are able to distinguish between different kinds of "infinity", i.e., some sets are "more infinite" than others.

Definition 75 (Countable Sets) A set is *countable (abzählbar unendlich)* if it has an *enumeration (Aufzählung)*, i.e., a bijective mapping from $\mathbb{N}$:

$$S \text{ is countable } :\Leftrightarrow \exists f : f : \mathbb{N} \overset{\text{bijective}}{\longrightarrow} S.$$

Example

- The set $\mathbb{Z}$ is infinite but it is countable because we can define a function $f : \mathbb{N} \overset{\text{bijective}}{\longrightarrow} \mathbb{Z}$ as

  $$f(x) := \textbf{if } x \text{ is even } \textbf{then } -x/2 \textbf{ else } (x+1)/2$$

  i.e.,

  $$f = [0, 1, -1, 2, -2, 3, -3, \ldots].$$

- The set $\mathbb{Q}$ is infinite but countable; we sketch the construction of a corresponding enumeration. We can list all positive rationals in an infinite matrix that holds at position $\langle i, j\rangle$ the rational $\frac{i+1}{j+1}$:

$$\begin{array}{ccccccccc}
\frac{1}{1} & & \frac{1}{2} & & \frac{1}{3} & & \frac{1}{4} & & \cdots \\
& \swarrow & & \swarrow & & \swarrow & & & \\
\frac{2}{1} & & \frac{2}{2} & & \frac{2}{3} & & \cdots & & \cdots \\
& \swarrow & & \swarrow & & & & & \\
\frac{3}{1} & & \frac{3}{2} & & \cdots & & \cdots & & \cdots \\
& \swarrow & & & & & & & \\
\frac{4}{1} & & \cdots & & \cdots & & \cdots & & \cdots \\
& & & & & & & & \\
\cdots & & \cdots & & \cdots & & \cdots & & \cdots
\end{array}$$

  We can enumerate all elements in this matrix by first traversing all fractions $\frac{x}{y}$ with $x + y = 2$, then all fractions with $x + y = 3$, then all fractions with $x + y = 4$, and so on (always enumerating the fractions $x + y = c$ in the order of increasing x).

  From this sequence, we have to remove all "doubles" (such as $\frac{2}{2}$ which has already appeared previously as $\frac{1}{1}$) constructing a corresponding sequence $f' : \mathbb{N} \to \mathbb{Q}$ that contains each positive rational number in *exactly* one position. Finally we can define an enumeration of the rationals by $g : \mathbb{N} \overset{\text{bijective}}{\longrightarrow} \mathbb{Q}$ defined as

  $g(x) :=$
  if $x = 0$ **then** 0
  else if x is even
  then $-f'(x/2)$
  else $f'((x-1)/2)$

  i.e.,

$$g = [0, 1, -1, \frac{1}{2}, -\frac{1}{2}, \frac{2}{1}, -\frac{2}{1}, \frac{1}{3}, -\frac{1}{3}, \frac{3}{1}, -\frac{3}{1}, \ldots].$$

- The set of all infinite sequences over $\{0, 1\}$ is *not* countable: if it were, we had some $f : \mathbb{N} \overset{\text{bijective}}{\longrightarrow} (\mathbb{N} \to \{0, 1\})$. Take $s : \mathbb{N} \to \{0, 1\}$ defined as

$$s(i) := \overline{f(i)_i}$$

where $\overline{d} := 1 - d$. Then s differs from $f(i)$ in the i-th digit (for every $i \in \mathbb{N}$), thus s is not contained in f.

$$\begin{array}{lllllll}
f(0) = & [\boldsymbol{f(0)_0} & f(0)_1 & f(0)_2 & f(0)_3 & \ldots] \\
f(1) = & [f(1)_0 & \boldsymbol{f(1)_1} & f(1)_2 & f(1)_3 & \ldots] \\
f(2) = & [f(2)_0 & f(2)_1 & \boldsymbol{f(2)_2} & f(2)_3 & \ldots] \\
f(3) = & [f(3)_0 & f(3)_1 & f(3)_2 & \boldsymbol{f(3)_3} & \ldots] \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots
\end{array}$$

$$s := [\overline{f(0)_0}, \overline{f(1)_1}, \overline{f(2)_2}, \overline{f(3)_3}, \ldots]$$

The argument called *diagonalization (Diagonalisierung)* has been invented by Cantor to show the following result.

- The set $\mathbb{R}$ is *not* countable: every infinite sequence d of decimal digits represents a real number $0.d_0d_1d_2\ldots$. Since the set of all infinite sequences is not countable (and every real number is represented by a countable set of such sequences), also $\mathbb{R}$ is not countable.

Above example shows us that there exists a bijection between $\mathbb{N}$ and $\mathbb{Z}$, i.e., there is one distinct natural number for every integer number and vice versa. It is therefore natural to consider $\mathbb{N}$ and $\mathbb{Z}$ of the *same size*[1]. The concept of bijections can therefore be used to compare the size of sets.

Proposition 76 (Set Cardinalities) Two sets have *same size* or *same cardinality*, if there is a bijection between them:

$$A \text{ and } B \text{ are of same size } :\Leftrightarrow \exists f : f : A \overset{\text{bijective}}{\longrightarrow} B.$$

One set is *not larger than* another set, if there exists an injection from the first set into the second set:

$$A \text{ is not larger than } B :\Leftrightarrow \exists f : f : A \overset{\text{injective}}{\longrightarrow} B.$$

[1]This does not contradict the statement that for every positive $n \in \mathbb{N}$, there is a unique pair $-n \in \mathbb{Z}$ and $+n \in \mathbb{Z}$. Rather it shows us that the union of two disjoint copies of an infinite set S still has the same size as S, i.e., that our intuition of set sizes cannot be automatically applied to the infinite case.

One set is *smaller than* another set, if they are not of same size and the second one is not larger than the first one.

$$\begin{aligned}&A \text{ is smaller than } B :\Leftrightarrow\\&\quad (A \text{ is not larger than } B) \wedge \neg(A \text{ and } B \text{ have same size}).\end{aligned}$$

For finite sets, above notions clearly coincide with the definition of $|\ |$.

Proposition 77 (Finite Sets) For all finite sets A and B, the following holds:

$$\begin{aligned}|A| = |B| &\Rightarrow A \text{ and } B \text{ have same size;}\\|A| \leq |B| &\Rightarrow A \text{ is not larger than } B;\\|A| < |B| &\Rightarrow A \text{ is smaller than } B.\end{aligned}$$

However the new notions also allow us to compare the sizes of *infinite* sets.

Example By the arguments given in the previous example, we have the following results:

- $\mathbb{N}$ has the same size as $\mathbb{Z}$.
- $\mathbb{Z}$ has the same size as $\mathbb{Q}$.
- $\mathbb{Q}$ is smaller than $\mathbb{R}$.

One can also easily show

- $\mathbb{R}$ has the same size as $\mathbb{C}$.

The number domains introduced in Section 4 therefore fall into two classes:

1. $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ (the enumerable ones),
2. $\mathbb{R}$, $\mathbb{C}$ (the not enumerable ones).

The hierarchy of "infiniteness" is not limited, because we can construct for every (also infinite) set a still larger set.

Proposition 78 (Size of Powerset) Every set is smaller than its powerset:

$$\forall S : S \text{ is smaller than } \mathbb{P}(S).$$

Proof Take arbitrary S. S is not larger than $\mathbb{P}(S)$ because we can define

$$f : S \stackrel{\text{injective}}{\longrightarrow} \mathbb{P}(S)$$
$$f(x) := \{x\}.$$

Assume that S and $\mathbb{P}(S)$ are of the same size, i.e., there exists some $f : S \stackrel{\text{bijective}}{\longrightarrow} \mathbb{P}(S)$. We show a contradiction.

Take $A := \{x \in S : x \notin f(x)\}$. Since f is surjective and $A \subseteq S$, i.e., $A \in \mathbb{P}(S)$, we have some $a \in S$ with $f(a) = A$. But then we know

$$a \in A \Leftrightarrow a \notin f(a) \Leftrightarrow a \notin A.$$

Consequently, we know that $\mathbb{N}$ is smaller than $\mathbb{P}(\mathbb{N})$ which is smaller than $\mathbb{P}(\mathbb{P}(\mathbb{N}))$ which is smaller than $\mathbb{P}(\mathbb{P}(\mathbb{P}(\mathbb{N})))$ which is ....

We also have the following result.

Proposition 80 (Size of Function Space) Every set is smaller than the set of all functions into it:

$$\forall A, B : B \text{ is smaller than } A \to B.$$

Permutations Bijections on subsets of $\mathbb{N}$ also of importance for sorting elements in a particular order.

Definition 76 (Permutation) A *permutation (Permutation)* of length n is a bijective function from $\mathbb{N}_n$ to $\mathbb{N}_n$:

$$p \text{ is permutation of length } n :\Leftrightarrow p : \mathbb{N}_n \xrightarrow{\text{bijective}} \mathbb{N}_n.$$

Example Take the sequence $s = [a, b, c, d, e]$ and the permutation $p = [1, 0, 4, 3, 2]$. Then we have

$$p \circ s = [b, a, e, d, c].$$

Logic Evaluator Permutations can be easily implemented as follows.

```
option silent = true; read function;

pred isPermutation(p: Function, n: Nat) <=>
  let(N = if(=(n, 0), {}, set(x in nat(0, -(n, 1)): true, x)):
    isBijective(p, N, N));

fun j(x, y, e) = join(tuple(x, y), e);
fun p = j(0, 3, j(1, 2, j(2, 0, j(3, 1, {}))));
fun s = j(0, {}(0), j(1, {}(1), j(2, {}(2), j(3, {}(3), {}))));

formula isPermutation(p, 4);
> true.
term s;
> {<3, {3}>, <2, {2}>, <1, {1}>, <0, {0}>}.
term o(p, s);
> {<3, {1}>, <2, {0}>, <1, {2}>, <0, {3}>}.

term o(p, p);
```

Example The problem of *sorting (Sortieren)* a sequence of elements can be specified as follows:

- **Input**:

– A ... the element domain,
– $\leq \subseteq A \times A$ ... a reflexive, antisymmetric, and transitive relation,
– $n \in \mathbb{N}$ ... the length of the sequence,
– $s : \mathbb{N}_n \to A$ ... a sequence of length n on A.

- **Output**: $t : \mathbb{N}_n \to A$ such that
  – t is permutation of s,
  – t is sorted with respect to $\leq$.

with the following auxiliary predicates:

$$\begin{array}{l} t \text{ is permutation of } s :\Leftrightarrow \\ \quad \textbf{let } n = \text{length}(t) : \\ \quad\quad n = \text{length}(s) \land \\ \quad\quad \exists p : p \text{ is permutation of length } n \land p \circ s = t; \end{array}$$

$$\begin{array}{l} t \text{ is sorted with respect to } \leq :\Leftrightarrow \\ \quad \forall 0 \leq i < \text{length}(t) : t_i \leq t_{i+1}. \end{array}$$

The set of permutations is closed under function composition.

Proposition 81 (Composition of Permutations) The composition of two permutations of the same length is also a permutation of this length:

$$\begin{array}{l} \forall n \in \mathbb{N}, p_0, p_1 : \\ \quad (p_0 \text{ is permutation of length } n \land \\ \quad p_1 \text{ is permutation of length } n) \Rightarrow \\ \quad\quad p_0 \circ p_1 \text{ is permutation of length } n. \end{array}$$

The inverse of a permutation is a permutation of the same length:

$$\begin{array}{l} \forall n \in \mathbb{N}, p : \\ \quad p \text{ is permutation of length } n \Rightarrow \\ \quad\quad p^{-1} \text{ is permutation of length } n. \end{array}$$

Example Take the permutations $p = [1, 0, 4, 3, 2]$ and $q = [2, 1, 3, 4, 0]$. Then

$$p \circ q = [1, 2, 0, 4, 3],$$

e.g., $(p \circ q)(2) = q(p(2)) = q(4) = 0$. We also have

$$p^{-1} = [2, 0, 1, 4, 3]$$

e.g., $p^{-1}(2) = 1$ because $p(1) = 2$, and thus

$$p \circ p^{-1} = p^{-1} \circ p = [0, 1, 2, 3, 4].$$

```
option silent = true; read function;

pred isPermutation(p: Function, n: Nat) <=>
  let(N = if(=(n, 0), {}, set(x in nat(0, -(n, 1)): true, x)):
    isBijective(p, N, N));

fun j(x, y, e) = join(tuple(x, y), e);
fun p = j(0, 3, j(1, 2, j(2, 0, j(3, 1, {}))));

formula isPermutation(o(p, p), 4);
> true.
formula isPermutation(^-1(p), 4);
> true.
term o(p, ^-1(p));
> {<3, 3>, <2, 2>, <1, 1>, <0, 0>}.
```

```
formula isPermutation(o(p, ^-1(p)), 4);
```

The following establishes some basic combinatorial knowledge.

Proposition 82 (Number of Permutations) The number of permutations of length n is 2^n:

$$\forall n \in \mathbb{N} : |\{f : f \text{ is permutation of length } n\}| = 2^n.$$

Proof We proceed by induction on n.

If n=0, then the only permutation is $p = [\,]$.

Assume $|\{f : f \text{ is permutation of length } n\}| = 2^n$.

We define

$$
\begin{array}{l}
\text{insert}(x, i, f) := \\
\quad \textbf{such } s : \\
\quad\quad \text{length}(s) = 1 + \text{length}(f) \land \\
\quad\quad \forall j \in \mathbb{N}_{n+1} : \\
\quad\quad\quad j < i \Rightarrow s(j) = f(j) \land \\
\quad\quad\quad j = i \Rightarrow s(j) = x \land \\
\quad\quad\quad j > i \Rightarrow s(j) = f(j-1)
\end{array}
$$

which returns the sequence constructed from f by inserting element x at position i.

Then we have

$$
\begin{array}{rl}
|\{f : f \text{is permutation of length } n+1\}| & = \\
|\{\text{insert}(n+1, i, f) : i \in \mathbb{N}_{n+1} \land f \text{ is permutation of length } n\}| & = \\
(n+1) * |\{f : f \text{ is permutation of length } n\}| & = \\
(n+1) * n & = \\
(n+1)! &
\end{array}
$$

6.3 Embedding Sets

We have introduced in Chapter 4 a sequence of number domains and stated rather vaguely that each domain can be "embedded" into the successor domain such that the properties of the corresponding operations are preserved. This statement can be made more precisely by the following concept.

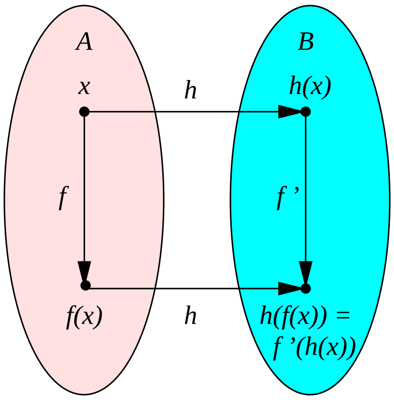
Definition 77 (Homomorphism) Let $f : A^n \to A$ and $f' : B^n \to B$. We call h a *homomorphism (Homomorphismus)* from A to B (with respect to f and f') if, for every $x_0 \in A, \ldots, x_{n-1} \in A$, $h(f(x_0, \ldots, x_{n-1})) = f'(h(x_0), \ldots, f'(x_{n-1}))$:

$$
\begin{array}{l}
h : A \overset{\mathrm{hom}(f,f')}{\longrightarrow} B :\Leftrightarrow \\
\quad h : A \to B \wedge \\
\quad (\exists n \in \mathbb{N} : \\
\quad\quad f : A^n \to A \wedge f' : B^n \to B \wedge \\
\quad\quad (\forall x \in A^n : h(f(x_0, \ldots, x_{n-1})) = f'(h(x_0), \ldots, h(x_{n-1})))).
\end{array}
$$

A *isomorphism (Isomorphismus)* is a bijective homomorphism.

$$
\begin{array}{l}
h : A \overset{\mathrm{iso}(f,f')}{\longrightarrow} B :\Leftrightarrow \\
\quad h : A \overset{\mathrm{hom}(f,f')}{\longrightarrow} B \wedge \\
\quad h : A \overset{\mathrm{bijective}}{\longrightarrow} B.
\end{array}
$$

A homomorphism makes the following diagram commute:



In other words, it does not matter whether we first perform the operation f in A and let h map the result to B or whether we let h map the arguments of f to B and apply f' to them. In this sense, h *embeds* A into B and their operations are *structurally equal.* If h is an isomorphism, then this embedding works in both directions: A and B become "identical twins" that cannot be distinguished by the behavior of f and f'.

Example

- Take $h : \mathbb{N} \to \mathbb{Z}$ defined as (see Definition 42 on page 109)

  $$h(x) := \langle x, 0 \rangle.$$

  Then we have, for all $x \in \mathbb{N}$ and $y \in \mathbb{N}$,

  $$\begin{array}{rcl} h(x +_{\mathbb{N}} y) & = & h(x) +_{\mathbb{Z}} h(y); \\ h(x *_{\mathbb{N}} y) & = & h(x) *_{\mathbb{Z}} h(y) \end{array}$$

  i.e., h is a homomorphism from $\mathbb{N}$ to $\mathbb{Z}$ (for the operations $+$ and $*$).

- Take $h : \mathbb{Z} \to \mathbb{Q}$ defined as (see Definition 45 on page 113):

  $$h(x) := \frac{x}{1_{\mathbb{Z}}}.$$

  Then we have, for all $x \in \mathbb{Z}$ and $y \in \mathbb{Z}$,

  $$\begin{array}{rcl} h(x +_{\mathbb{Z}} y) & = & h(x) +_{\mathbb{Q}} h(y); \\ h(x -_{\mathbb{Z}} y) & = & h(x) -_{\mathbb{Q}} h(y); \\ h(x *_{\mathbb{Z}} y) & = & h(x) *_{\mathbb{Q}} h(y) \end{array}$$

  i.e., h is a homomorphism from $\mathbb{Z}$ to $\mathbb{Q}$ (for operations $+$, $-$, and $*$).

- Take the domain "List(T)" with functions "length" and "append" (see the Examples on page 162 and 163).

  We have for all $x \in \text{List}(T)$ and $y \in \text{List}(T)$

  $$\text{length}(\text{append}(x, y)) = \text{length}(x) + \text{length}(y)$$

  i.e., "length" is a homomorphism from "List(T)" to $\mathbb{N}$ with respect to "append" and $+$.

- Take the set of polynomials. We have, for all polynomials x and y and $a \in \mathbb{R}$,

  $$\begin{array}{rcl} (x + y)[a] & = & x(a) +_{\mathbb{R}} y(a) \\ (x - y)[a] & = & x(a) -_{\mathbb{R}} y(a) \\ (-x)[a] & = & -_{\mathbb{R}} x(a) \\ (x * y)[a] & = & x(a) *_{\mathbb{R}} y(a) \end{array}$$

  i.e., polynomial evaluation is a homomorphism from the set of polynomials to $\mathbb{R}$ (for operations $+$, $-$, $*$).

As the previous example suggests, we have a homomorphism from each number domain introduced in Chapter 4 to its successor. Actually, each domain is *isomorphic* to some subset of its successor, i.e.,

$$\mathbb{N} \subseteq' \mathbb{Z} \subseteq' \mathbb{Q} \subseteq' \mathbb{R} \subseteq' \mathbb{C}$$

where

$$A \subseteq' B :\Leftrightarrow \exists h, B' \subseteq B : h : A \overset{\mathrm{iso}(O_A, O'_B)}{\longrightarrow} B'.$$

Here O_A denotes the considered operations on A, O'_B denotes the corresponding operations on B' and $h : A \overset{\mathrm{iso}(O_A, O'_B)}{\longrightarrow} B'$ states that h is an isomorphism between A and B for each operation pair.

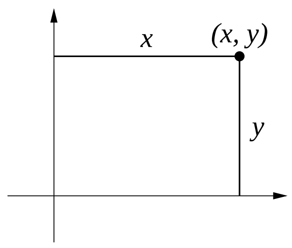
Logic Evaluator We demonstrate that (a finite subset of) $\mathbb{N}$ is isomorphic to a subset of $\mathbb{Z}$:

```
option silent = true; read function; read integer;
pred isBinaryHomomorphism(h, A, B, f, f') <=>
  and(isFunction(h, A, B),
    isFunction(f, x(A, A), A), isFunction(f', x(B, B), B),
    forall(x0 in A, x1 in A:
      =(apply(h, apply(f, tuple(x0, x1))),
        apply(f', tuple(apply(h, x0), apply(h, x1))))));

fun N = 4;
fun A = set(x in nat(0, N): true, x);
fun Z = I(N, 0);
fun B = set(x in nat(0, N): true, I(x, 0));

// functions as sets
fun t(x, y, z) = tuple(tuple(x, y), z);
fun minN(x) = if (<=N(x, N), x, N);
fun +N' = set(x in A, y in A: true, t(x, y, minN(+N(x, y))));
fun *N' = set(x in A, y in A: true, t(x, y, minN(*N(x, y))));
fun minZ(x) = if (<=Z(x, Z), x, Z);
fun +Z' = set(x in B, y in B: true, t(x, y, minZ(+Z(x, y))));
fun *Z' = set(x in B, y in B: true, t(x, y, minZ(*Z(x, y))));

// the isomorphism
fun h = set(x in A: true, tuple(x, I(x, 0)));
formula isBinaryHomomorphism(h, A, B, +N', +Z');
> true.
formula isBijective(h, A, B);
> true.
```

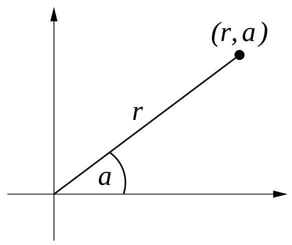
```
formula isBinaryHomomorphism(h, A, B, *N', *Z');
```

Isomorphisms are also useful to demonstrate the equivalence of different constructions as shown below.

Another Construction of the Complex Numbers Every element $x+y\mathrm{i}$ of $\mathbb{C}$ can be visualized as a point in the plane whose *Cartesian coordinates (kartesische Koordinaten)* are denoted by the real part and by the imaginary part of the complex number, respectively:



In *polar coordinates (Polarkoordinaten)*, each point is represented by a pair $\langle r, \alpha \rangle$ where r denotes the distance of the point from the origin and α denotes the angle to the horizontal axis:



Thus we can define a "polar" variant of $\mathbb{C}$

$$\mathbb{C}' := \mathbb{R} \times [0, 2\pi[$$

assuming that angles are uniquely expressed in "radians" (i.e., $\pi = 180°$).

A translation from the new domain into the original one is given by

$$\begin{aligned}&\text{cartesian} : \mathbb{C}' \to \mathbb{C}\\&\text{cartesian}(z) = z_0 * \cos(z_1) + z_0 * \sin(z_1)\mathrm{i}.\end{aligned}$$

where "cos" and "sin" represent cosine and sine, respectively.

The other direction is a bit more complicated:

$$
\begin{array}{l}
\text{polar} : \mathbb{C} \to \mathbb{C}' \\
\text{polar}(z) = (\sqrt{z_0^2 + z_1^2}, \alpha) \\
\quad \textbf{where } \alpha = \\
\quad\quad \textbf{if } z_0 = 0 \textbf{ then} \\
\quad\quad\quad \textbf{if } z_1 > 0 \textbf{ then } \pi/2 \textbf{ else } 3\pi/2 \\
\quad\quad \textbf{else} \\
\quad\quad\quad \textbf{let } a = \arctan(z_1/z_0) : \\
\quad\quad\quad \text{shift}(\textbf{if } z_0 \geq 0 \textbf{ then } a \textbf{ else } \pi - a)
\end{array}
$$

where "arctan" is the inverse ("arcus") tangent. Since the inverse tangent does not determine the result uniquely, we have to make a case distinction and normalize the result by the following auxiliary function:

$$
\begin{array}{l}
\text{shift} : \mathbb{R} \to [0, 2\pi[\\
\text{shift}(a) := (\textbf{such } b : b \in [0, 2\pi[\, \wedge \, \exists i \in \mathbb{Z} : a - b = 2\pi i)
\end{array}
$$

One can now show

$$
\begin{array}{l}
\text{cartesian} \circ \text{polar} = 1_{\mathbb{C}}, \\
\text{polar} \circ \text{cartesian} = 1_{\mathbb{C}'}
\end{array}
$$

i.e., "cartesian" and "polar" are bijections.

We may define the following "multiplication" on ℂ':

$$x *_{\mathbb{C}'} y := \langle x_0 * y_0, \text{shift}(x_1 + y_1) \rangle$$

We then have

$$
\begin{array}{l}
\text{cartesian} : \mathbb{C} \overset{\text{iso}(*_{\mathbb{C}}, *_{\mathbb{C}'})}{\longrightarrow} \mathbb{C}' \\
\text{polar} : \mathbb{C}' \overset{\text{iso}(*_{\mathbb{C}'}, *_{\mathbb{C}})}{\longrightarrow} \mathbb{C}
\end{array}
$$

i.e., ℂ and ℂ' are *isomorphic* with respect to their respective notions of multiplication.

Proof We show that "cartesian" is a homomorphism with respect to multiplication:

$$\forall x \in \mathbb{C}', y \in \mathbb{C}' : \text{cartesian}(x *_{\mathbb{C}'} y) = \text{cartesian}(x) *_{\mathbb{C}} \text{cartesian}(y).$$

Take arbitrary $x \in \mathbb{C}'$ and $y \in \mathbb{C}'$. We then have

$$\begin{array}{rcl}
\text{cartesian}(x *_{\mathbb{C}'} y) & = & \\
\text{cartesian}(x_0 y_0, \text{shift}(x_1 + y_1)) & = & \\
x_0 y_0 \cos(\text{shift}(x_1 + y_1)) + (x_0 y_0 \sin(\text{shift}(x_1 + y_1)))\mathrm{i} & = & (*) \\
x_0 y_0 \cos(x_1 + y_1) + (x_0 y_0 \sin(x_1 + y_1))\mathrm{i}. & &
\end{array}$$

(*) holds because of the definition of "shift" and because, for every $x \in \mathbb{R}$,

$$\begin{array}{rcl}
\sin(x + 2\pi) & = & \sin(x), \\
\cos(x + 2\pi) & = & \cos(x)
\end{array}$$

which we assume as granted knowledge.

We also have

$$\begin{array}{rcl}
\text{cartesian}(x) *_{\mathbb{C}} \text{cartesian}(y) & = & \\
(x_0 \cos(x_1) + x_0 \sin(x_1)\mathrm{i}) *_{\mathbb{C}} (y_0 \cos(y_1) + y_0 \sin(y_1)\mathrm{i}) & = & \\
(x_0 y_0 \cos(x_1)\cos(y_1) - x_0 y_0 \sin(x_1)\sin(y_1)) + & & \\
(x_0 y_0 \cos(x_1)\sin(y_1) + x_0 y_0 \sin(x_1)\cos(y_1))\mathrm{i} & = & \\
x_0 y_0 (\cos(x_1)\cos(y_1) - \sin(x_1)\sin(y_1)) + & & \\
x_0 y_0 (\cos(x_1)\sin(y_1) + x_0 y_0 \sin(x_1)\cos(y_1))\mathrm{i} & &
\end{array}$$

We assume the knowledge

$$\begin{array}{rcl}
\cos(x_1 + y_1) & = & \cos(x_1)\cos(y_1) - \sin(x_1)\sin(y_1) \\
\sin(x_1 + y_1) & = & \cos(x_1)\sin(y_1) + \sin(x_1)\cos(y_1)
\end{array}$$

as granted and are therefore done.

We may also define corresponding notions of addition, subtraction, and division in $\mathbb{C}$' such that they are isomorphic to their counterparts in $\mathbb{C}$.

Consequently, we may operate in $\mathbb{C}$ or in $\mathbb{C}$', whatever is more convenient, and translate the results into the other direction. In particular, if the computation of some operation is rather awkward in one domain, it may be simpler to translate the arguments into the other domain, perform the operation there, and translate the result back into the original domain. This is in particular true for the computation of complex roots.

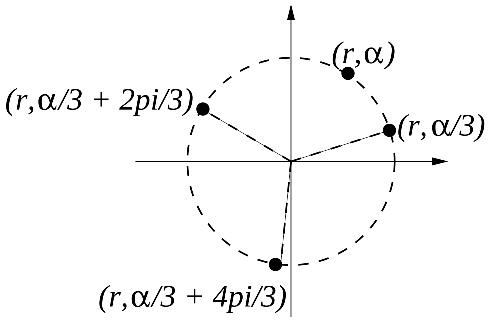
Definition 78 (Complex Root)

$$\begin{array}{l}
\sqrt{} : (\mathbb{N} \times \mathbb{C}') \to \mathbb{C}' \\
\sqrt[n]{z} := \langle \sqrt[n]{z_0}, z_1/n \rangle.
\end{array}$$

Proposition 85 (Complex Roots) For every $n \in \mathbb{N}_{>0}$ and $z \in \mathbb{C}'$, the n-th roots of z are $\sqrt[n]{z}$ and the n-1 values that have the same distance from the origin and their angle shifted by multiples of $2\pi/n$:

$$
\begin{array}{l}
\forall n \in \mathbb{N}_{>0}, z \in \mathbb{C}' : \\
\quad \textbf{let } r = \sqrt[n]{z} : \\
\quad (\forall s \in \mathbb{C}' : z = s^n \Leftrightarrow \exists i \in \mathbb{N} : s = \langle r_0, \text{shift}(r_1 + 2\pi i/n)\rangle).
\end{array}
$$

In other words, the n-th roots of a complex number are on the same circle and have equal distance from each other as shown in the following diagram for the cube root of $\langle r, \alpha\rangle$.



Since roots can be computed in ℂ' most easily, one usually does not bother to introduce this concept in $\mathbb{C}$ but simply defines:

$$
\begin{array}{l}
\sqrt[\cdot]{} : (\mathbb{N}_{>0} \times \mathbb{C}) \rightarrow \mathbb{C} \\
\sqrt[n]{z} := \text{cartesian}(\sqrt[n]{\text{polar}(z)}).
\end{array}
$$

Vice versa, addition and subtraction are done more easily in $\mathbb{C}$ than in ℂ':

$$
\begin{array}{l}
+_{\mathbb{C}'} : (\mathbb{C}' \times \mathbb{C}') \rightarrow \mathbb{C}' \\
x +_{\mathbb{C}'} y := \text{polar}(\text{cartesian}(x) +_{\mathbb{C}} \text{cartesian}(y)); \\
-_{\mathbb{C}'} : (\mathbb{C}' \times \mathbb{C}') \rightarrow \mathbb{C}' \\
x -_{\mathbb{C}'} y := \text{polar}(\text{cartesian}(x) +_{\mathbb{C}} \text{cartesian}(y)).
\end{array}
$$

6.4 Sequences and Series

In this section, we discuss infinite sequences over $\mathbb{R}$; we often write them in the following form.

Definition 79 (Sequence Quantor) For every variable i and term T, the phrase

$$[T]_i$$

is a term with bound variable i whose value is the sequence

$$[T]_i : \mathbb{N} \to \mathbb{R}$$
$$[T]_i(i) := T.$$

Example $[a^2 + c]_a = [0 + c, 1 + c, 4 + c, 9 + c, \ldots]$.

The sequence elements may be ordered in a way that is compatible with the order of the element domain.

Definition 80 (Monotonicity) Let f be an infinite sequence over $\mathbb{R}$. f is *monotonically increasing (monoton wachsend)* if every element of f is less than or equal the next element:

$$f \text{ is monotonically increasing } :\Leftrightarrow f : \mathbb{N} \to \mathbb{R} \wedge \forall i \in \mathbb{N} : f_i \leq f_{i+1}.$$

f is *strictly monotonically increasing (streng monoton wachsend)* if every element of f is less than the next element:

$$f \text{ is strictly monotonically increasing } :\Leftrightarrow$$
$$f : \mathbb{N} \to \mathbb{R} \wedge \forall i \in \mathbb{N} : f_i < f_{i+1}.$$

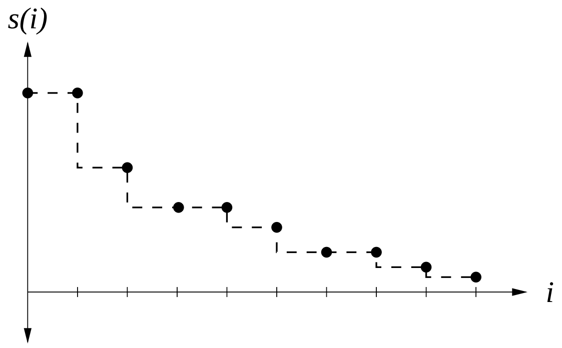
f is *monotonically decreasing (monoton wachsend)* if every element of f is greater than or equal the next element:

$$f \text{ is monotonically decreasing } :\Leftrightarrow f : \mathbb{N} \to \mathbb{R} \wedge \forall i \in \mathbb{N} : f_i \geq f_{i+1}.$$

f is *strictly monotonically decreasing (streng monoton wachsend)* if every element of f is greater than the next element:

$$\begin{aligned}&f \text{ is strictly monotonically decreasing } :\Leftrightarrow\\&\quad f : \mathbb{N} \to \mathbb{R} \wedge \forall i \in \mathbb{N} : f_i > f_{i+1}.\end{aligned}$$

The following figure illustrates a monotonically (not strictly monotonically) decreasing sequence:



Example

- $[i]_i$ is strictly monotonically increasing:

  $$[i]_i = [0, 1, 2, 3, \ldots]$$

- $[\frac{1}{i+1}]_i$ is strictly monotonically decreasing:

  $$[\frac{1}{i+1}]_i = [1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots]$$

- $[i \text{ div } 2]_i$ is monotonically increasing:

  $$[i \text{ div } 2]_i = [0, 0, 1, 1, 2, 2, \ldots]$$

- $[(-1)^i]_i$ is neither monotonically increasing nor decreasing:

  $$[(-1)^i]_i = [1, -1, 1, -1, \ldots]$$

The elements of an infinite sequence either become arbitrarily large or their size is bounded.

Definition 81 (Upper and Lower Bounds) Let f be an infinite sequence over $\mathbb{R}$. f has an *upper bound (obere Schranke)* U if every element of f is less than or equal U:

$$U \text{ is upper bound of } f :\Leftrightarrow f : \mathbb{N} \to \mathbb{R} \land \forall i \in \mathbb{N} : f_i \leq U.$$

f has a *lower bound (untere Schranke)* L if every element of f is less than or equal L:

$$L \text{ is lower bound of } f :\Leftrightarrow f : \mathbb{N} \to \mathbb{R} \land \forall i \in \mathbb{N} : f_i \leq L.$$

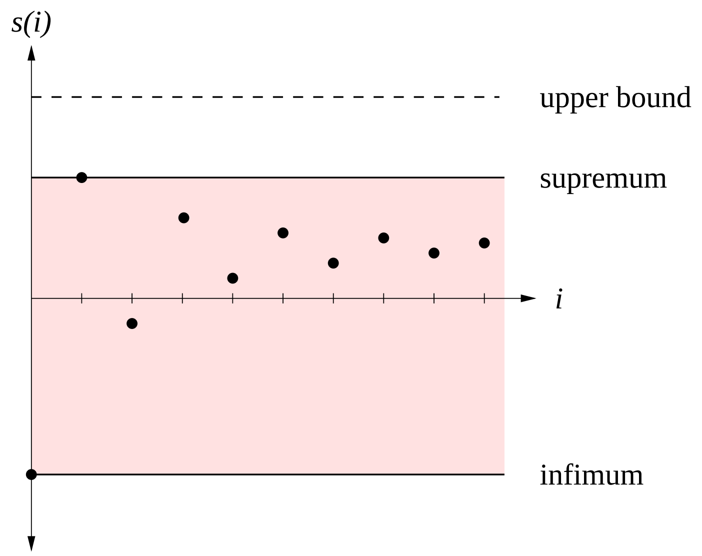
The *supremum (Supremum)* of f is the smallest upper bound of f:

$$\begin{aligned}&\sup(f) := \textbf{such } S :\\&\quad S \text{ is upper bound of } f \land\\&\quad (\forall S' : S' \text{ is upper bound of } f \Rightarrow S \leq S').\end{aligned}$$

The *infimum (Infimum)* of f is the greatest lower bound of f:

$$\begin{aligned}&\inf(f) := \textbf{such } I :\\&\quad I \text{ is lower bound of } f \land\\&\quad (\forall I' : I' \text{ is lower bound of } f \Rightarrow I \geq I').\end{aligned}$$

The notions are visualized below:



While there are infinitely many upper and lower bounds (if any), supremum and infimum are uniquely defined.

Proof We define

$$\begin{aligned}&S \text{ is supremum of } f :\Leftrightarrow\\&\quad S \text{ is upper bound of } f \wedge\\&\quad (\forall S' : S' \text{ is upper bound of } f \Rightarrow S \leq S')\end{aligned}$$

(i.e., $\sup(f) = \textbf{such } S : S \text{ is supremum of } f$) and show

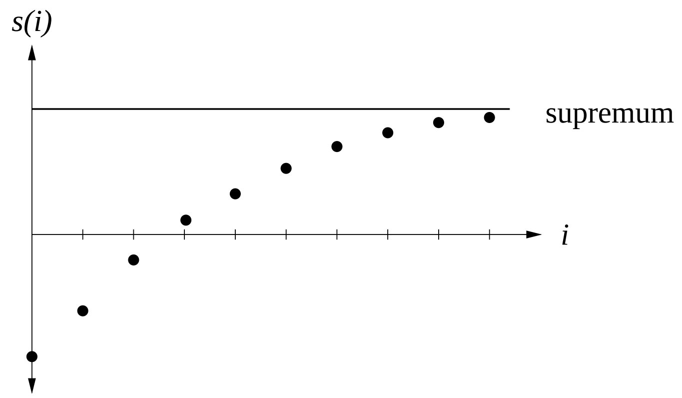
$$\begin{aligned}&\forall f, S_0, S_1 :\\&\quad S_0 \text{ is supremum of } f \wedge S_1 \text{ is supremum of } f \Rightarrow S_0 = S_1.\end{aligned}$$

Take arbitrary f and suprema S_0 and S_1 of f. Since S_0 is a supremum and S_1 is an upper bound of f, we have $S_0 \leq S_1$. Conversely, since S_1 is a supremum and S_0 is an upper bound of f, we have $S_1 \leq S_0$. Since $S_0 \leq S_1$ and $S_1 \leq S_0$, we have $S_0 = S_1$.

Example

- $[i]_i$ has infimum 0 but no upper bound.
- $[1 - i]_i$ has supremum 1 but no lower bound.
- $[\frac{1}{i+1}]_i$ has supremum 1 and infimum 0.
- $[i^3]_i$ has no upper bound and no lower bound.

As above example shows, the infimum and supremum of a sequence need not be among the values of the sequence; in this case the sequence values arbitrarily close approach the bound without ever actually reaching it:



Logic Evaluator We simulate some of the notions introduced above by considering a finite initial segment of the sequence.

```
option silent = true; read real; read function;

// is sequence increasing/decreasing in first n members?
pred isMonoIncreasing(f, n) <=>
  forall(i in nat(0, n): <=R(apply(f, i), apply(f, +(i, 1))));
pred isMonoDecreasing(f, n) <=>
  forall(i in nat(0, n): <=R(apply(f, +(i, 1)), apply(f, i)));

// is B upper/lower bound of f in first n members of sequence s?
pred isUpperBound(B, s, n) <=>
  forall(i in nat(0, n): <=R(apply(s, i), B));
pred isLowerBound(B, s, n) <=>
  forall(i in nat(0, n): <=R(B, apply(s, i)));

// first n+1 members of sequence 1/i
fun s(n) =
  set(i in nat(0, +(n, 1)): true, tuple(i, /R(R1, +R(R1, R(i)))));
fun n = 4;
fun s = s(n);

formula isMonoIncreasing(s, n);
> false.
formula isMonoDecreasing(s, n);
> true.
formula isUpperBound(R1, s, n);
> true.
formula isUpperBound(/R(R1, R2), s, n);
> false.

formula isLowerBound(R0, s, n);
```

Please note that the predicate `isMonoIncreasing`(s, n) only checks whether the sequence s is monotonically increasing in the first n elements and correspondingly for the other predicates. In general we can describe in the Logic Evaluator infinite notions only by finite approximations in the style demonstrated above.

Correspondingly, the following definitions determine a pseudo-supremum and pseudo-infimum among the first n values of the given sequence ; please note that for infinite sequences supremum respectively infimum need *not* be among the values of the sequence; please compare with the actual definitions of “sup” and “inf” on page 194.

```
option silent = true; read realfunction;

fun sup(s, n) =
  such(U in range(s):
    and(isUpperBound(U, s, n),
      forall(U' in range(s):
        implies(isUpperBound(U', s, n), <=R(U, U')))), U);
fun inf(s, n) =
  such(L in range(s):
    and(isLowerBound(L, s, n),
      forall(L' in range(s):
         implies(isLowerBound(L', s, n), <=R(L', L)))), L);

// first n+1 members of sequence 1/i
fun s(n) =
  set(i in nat(0, +(n, 1)): true, tuple(i, /R(R1, +R(R1, R(i)))));
fun n = 4;
fun s = s(n);

term sup(s, n);
> <<1, 0>, <1, 0>>.
```

```
term inf(s, n);
```

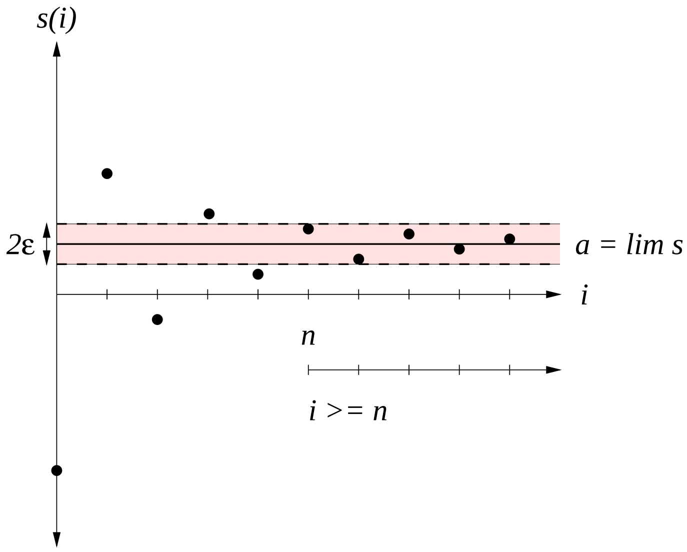
Definition 82 (Convergence and Limit) An infinite sequence s over $\mathbb{R}$ *converges (konvergiert)* to *limit (Grenzwert, Limes)* a, if its members approach a arbitrarily close:

$$s \text{ converges to } a :\Leftrightarrow \forall \epsilon > 0 : \exists n \in \mathbb{N} : \forall i \geq n : |s_i - a| < \epsilon;$$
$$\lim(s) := \textbf{such } a : s \text{ converges to } a.$$

A non-convergent series is called *divergent (divergent)*:

$$s \text{ is divergent} :\Leftrightarrow \neg\exists a : s \text{ converges to } a.$$

A geometric interpretation of convergence is given by the following picture that shows that, for every $\epsilon > 0$, all members of a convergent sequence are eventually in an "ϵ-tunnel" around limit a, i.e., in the interval $]a - \epsilon, a + \epsilon[$:



Example Let $s := [(-1)^i * \frac{1}{i+1}]_i$. We show that s converges to 0. Take arbitrary $\epsilon > 0$. We have to find some $n \in \mathbb{N}$ such that

$$\forall i \geq n : |(-1)^i * \frac{1}{i+1} - 0| < \epsilon$$

which can be simplified to

$$\forall i \geq n : \frac{1}{i} < \epsilon.$$

Take $n :=$ **such** $n \in \mathbb{N} : \frac{1}{\epsilon} < n$. Because $\mathbb{N}$ is unbounded (i.e., $\forall r \in \mathbb{R} : \exists n \in \mathbb{N} : r < n$), we know $\frac{1}{\epsilon} < n$.

Take arbitrary $i \geq n$. We then have

$$\frac{1}{i} \leq \frac{1}{n} < \frac{1}{\frac{1}{\epsilon}} = \epsilon.$$

Example Let $s := [(-1)^i]_i$. We show that s is divergent. Assume that s is convergent, i.e., s converges to some limit $a \in \mathbb{R}$. We show a contradiction.

Let $\epsilon := \frac{1}{2}$. There exists some $n \in \mathbb{N}$ such that

$$\forall i \geq n : |(-1)^i - a| < \frac{1}{2}.$$

We thus have

$$|1 - a| < \tfrac{1}{2} \wedge |-1 - a| < \tfrac{1}{2}$$

(substituting for i the values n and $n+1$ one of which is even and one of which is odd).

We then have (see the absolute value laws on page 127)

$$\begin{aligned} &1 = \tfrac{1}{2} + \tfrac{1}{2} > |1 - a| + |-1 - a| \\ &= |1 - a| + |1 + a| \geq |(1 - a) + (1 + a)| = 2 \end{aligned}$$

which represents a contradiction.

The limit of a convergent sequence is uniquely defined.

Proposition 87 (Unicity of Limits) If s converges to both a_0 and a_1, then $a_0 = a_1$:

$$\forall s, a_0, a_1 : (s \text{ converges to } a_0 \wedge s \text{ converges to } a_1) \Rightarrow a_0 = a_1.$$

Proof Take arbitrary s, a_0 and a_1 such that s converges to both a_0 and a_1. Assume $a_0 \neq a_1$. We show a contradiction.

Let $\epsilon := |\frac{a_1 - a_0}{2}|$. We know $\epsilon > 0$, therefore there exist $n_0 \in \mathbb{N}$ and $n_1 \in \mathbb{N}$ such that

$$(\forall i \geq n_0 : |s_i - a_0| < \epsilon) \wedge (\forall i \geq n_1 : |s_i - a_1| < \epsilon).$$

Let $n := \max(n_0, n_1)$. We then know

$$|s_n - a_0| < \epsilon \wedge |s_n - a_1| < \epsilon.$$

Consequently, we have (see the absolute value laws on page 127)

$$\begin{aligned}|a_1 - a_0| = 2\epsilon = \epsilon + \epsilon > |s_n - a_0| + |s_n - a_0| \\ \geq |(s_n - a_0) - (s_n - a_1)| = |a_1 - a_0|\end{aligned}$$

which represents a contradiction.

We often write limits in form of a quantor.

Definition 83 (Limit Quantor) For every variable i and term T, the following phrase represents a term with bound variable i:

$$\lim_{i\to\infty} T$$

The value of this term is that of the term

$$\lim([T]_i).$$

We then have the following laws for the manipulation of limit terms.

Proposition 89 (Limit Laws) For all convergent sequences $[A]_i$ and $[B]_i$, we have

$$\begin{aligned}\lim_{i\to\infty}(A+B) &= (\lim_{i\to\infty} A) + (\lim_{i\to\infty} B),\\ \lim_{i\to\infty}(A-B) &= (\lim_{i\to\infty} A) - (\lim_{i\to\infty} B),\\ \lim_{i\to\infty}(A*B) &= (\lim_{i\to\infty} A) * (\lim_{i\to\infty} B).\end{aligned}$$

If $\lim_{i\to\infty} B \neq 0$, we also have

$$\lim_{i\to\infty}(\frac{A}{B}) = \frac{\lim_{i\to\infty} A}{\lim_{i\to\infty} B}.$$

Furthermore, we have for every $c \in \mathbb{R}$

$$\lim_{i\to\infty}(c*A) = c*\lim_{i\to\infty} A.$$

Proof Take arbitrary convergent sequences $[A]_i$ and $[B]_i$. We prove

$$\lim_{i\to\infty}(A+B) = (\lim_{i\to\infty} A) + (\lim_{i\to\infty} B).$$

Let $a := \lim_{i\to\infty} A$ and $b := \lim_{i\to\infty} B$. We have to show

$$\forall \epsilon > 0 : \exists n \in \mathbb{N} : \forall i > n : |A + B - (a + b)| < \epsilon.$$

Take arbitrary $\epsilon > 0$. We know that there exist $n_0 \in \mathbb{N}$ and $n_1 \in \mathbb{N}$ such that

$$\begin{array}{l} \forall i \geq n_0 : |A - a| < \frac{\epsilon}{2}, \\ \forall i \geq n_1 : |B - b| < \frac{\epsilon}{2}. \end{array}$$

Let $n := \max(n_0, n_1)$. We have to show

$$\forall i \geq n : |A + B - (a + b)| < \epsilon.$$

Take arbitrary $i \geq n$. We know

$$\begin{array}{rcl} |A + B - (a + b)| & = & \\ |(A - a) + (B - b)| & \leq & \\ |A - a| + |B - b| & < & \\ \frac{\epsilon}{2} + \frac{\epsilon}{2} & = & \epsilon. \end{array}$$

Example

$$\lim_{i\to\infty}(\frac{i+2}{i-3}) = \lim_{i\to\infty}(\frac{1+\frac{2}{i}}{1-\frac{3}{i}}) = \frac{\lim_{i\to\infty}(1+\frac{2}{i})}{\lim_{i\to\infty}(1-\frac{3}{i})}$$

$$= \frac{\lim_{i\to\infty}(1) + \lim_{i\to\infty}(\frac{2}{i})}{\lim_{i\to\infty}(1) - \lim_{i\to\infty}(\frac{3}{i})} = \frac{1+0}{1-0} = 1.$$

An important mathematical constant is defined as a limit.

Definition 84 (Euler's Number)

$$\mathrm{e} := \lim_{i\to\infty} \left(1 + \frac{1}{i}\right)^i.$$

One can show that the sequence $[(1 + \frac{1}{i})^i]_i$ is convergent; thus e is indeed well-defined with value is $2.71828\ldots$.

We now construct a new sequence by summing up the elements of another sequence.

Definition 85 (Series) Let a be an infinite sequence over $\mathbb{R}$. The *series (Reihe)* corresponding to a is the sequence where every element s_n is the sum of the first $n+1$ elements of a:

$$\begin{array}{l} \text{series} : (\mathbb{N} \to \mathbb{R}) \to (\mathbb{N} \to \mathbb{R}) \\ \text{series}(a)_n := \sum_{0 \le i \le n} a_i. \end{array}$$

If $a = [T]_i$, then series(a) is denoted by the term $[\sum_{0\le i\le n} T]_n$.

Example Let $a = [i^2]_i$:

$$a = [0, 1, 4, 9, 16, 25, \ldots].$$

Then series$(a) = [\sum_{0\le i\le n} i^2]_n$:

$$\text{series}(a) = [0, 1, 5, 14, 30, 55, \ldots].$$

Logic Evaluator A series can be simply defined as follows.

```
option silent = true; read function;

// the sum of a[i..n]
fun sum(a, i, n) recursive -(n, i) =
  if(<=(n, i), 0, +(apply(a, i), sum(a, +(i, 1), n)));

// the first n+1 members of the series constructed from a
fun series(a, n) = set(i in nat(0, n): true,
  tuple(i, sum(a, 0, i)));

fun s(n) = set(i in nat(0, n): true, tuple(i, *(i, i)));

term s(10);
> {<0, 0>, <1, 1>, <2, 4>, <3, 9>, <4, 16>, <5, 25>, <6, 36>, <7,
49>, <8, 64>, <9, 81>, <10, 100>}.
term series(s(10), 10);
> {<0, 0>, <1, 0>, <2, 1>, <3, 5>, <4, 14>, <5, 30>, <6, 55>, <7,
91>, <8, 140>, <9, 204>, <10, 285>}.
```

```
term series(series(s(10), 10), 10);
```

The following example illustrates some important classes of series.

Example

Arithmetic Series For every $c \in \mathbb{R}$, the sequence $[i * c]_i$ is called an *arithmetic sequence (arithmetische Folge)*. Correspondingly, $\text{series}([i * c]_i)$ is called an *arithmetic series (arithmetische Reihe)*. We have, for every $n \in \mathbb{N}$,

$$\text{series}([i * c]_i)_n = \sum_{0 \le i \le n} i * c = c * \frac{n(n+1)}{2}.$$

(which can be proved by induction on n).

For instance, for $c = 1$, we have

$$\begin{array}{rcl} [i]_i & = & [0,1,2,3,4,5,\ldots], \\ \text{series}([i]_i) & = & [0,1,3,6,10,15,\ldots]. \end{array}$$

Geometric Series For every $q \in \mathbb{R}$, the sequence $[q^i]_i$ is called a *geometric sequence (geometrische Folge)*. Correspondingly, $\text{series}([q^i]_i)$ is called a *geometric series (geometrische Reihe)*. We have, for every $n \in \mathbb{N}$,

$$\text{series}([q^i]_i)_n = \sum_{0 \le i \le n} q^i = \frac{q^n - 1}{q - 1}.$$

(which can be proved by induction on n).

For instance, for $q = 2$, we have

$$\begin{aligned} [2^i]_i &= [1,2,4,8,16,\ldots], \\ \text{series}([2^i]_i) &= [1,3,7,15,31]. \end{aligned}$$

The limit of a series is usually denoted as follows.

Definition 86 (Series Limit Quantor) For every variable i and term T, the following phrase denotes a term with bound variable i:

$$\sum_{i=0}^{\infty} T$$

The value of this term is

$$\lim_{n\to\infty} \sum_{0\leq i\leq n} T.$$

Please note that the value $\sum_{i=0}^{\infty} T$ is only well defined if $[\sum_{0\leq i\leq n} T]_n$ is convergent.

We then have the following well-known result.

Proposition 91 (Limit of Geometric Series) For every $q \in \mathbb{R}$ with $|q| < 1$, the series $[\sum_{0\leq i\leq n} q^i]_n$ converges to $\frac{1}{1-q}$:

$$\forall q \in \mathbb{R} : |q| < 1 \Rightarrow \sum_{i=0}^{\infty} q^i = \tfrac{1}{1-q}.$$

Proof Take arbitrary $q \in \mathbb{R}$ with $|q| < 1$. We then have

$$\begin{aligned} \sum_{i=0}^{\infty} q^i &= \\ \lim_{n\to\infty} \sum_{0\leq i\leq n} T &= \\ \lim_{n\to\infty} \frac{q^n-1}{q-1} &= \\ \frac{\lim_{n\to\infty}\ (q^n-1)}{\lim_{n\to\infty}\ (q-1)} &= \\ \frac{(\lim_{n\to\infty}\ q^n)-(\lim_{n\to\infty}\ 1)}{q-1} &= \quad (*) \\ \frac{0-1}{q-1} &= \\ \frac{1}{1-q}. \end{aligned}$$

(*) The fact $\lim_{n\to\infty}\ q^n = 0$ has to be shown in a separate proof.

6.5 Special Functions

We now investigate the behavior of several important functions over $\mathbb{R}$. Most of the notations defined for sequences in the previous section (monotonicity, upper and lower bound, supremum, infimum, convergence, limit) apply to such functions as well; we do not repeat the corresponding definitions.

Real functions can be constructed in a modular manner by composing more complex functions from simpler ones. The composition operations are defined "pointwise", i.e., the value of a composed function at point x is determined only by the values of the component functions at this point.

Definition 87 (Pointwise Function Definition) Let RealFun $= \mathbb{R} \to \mathbb{R}$. We define the following function composition operations:

$$
\begin{aligned}
&. : \mathbb{R} \to \text{RealFun} \\
&c(x) := c; \\[1em]
&+ : (\text{RealFun} \times \text{RealFun}) \to \text{RealFun} \\
&(f+g)(x) := f(x) + g(x); \\[1em]
&- : (\text{RealFun} \times \text{RealFun}) \to \text{RealFun} \\
&(f+g)(x) := f(x) - g(x); \\[1em]
&: (\text{RealFun} \times \text{RealFun}) \to \text{RealFun} \\
&(f * g)(x) := f(x) * g(x); \\[1em]
&\dot{\overline{\cdot}} : (\text{RealFun} \times \text{RealFun}) \to (\mathbb{R} \stackrel{\text{partial}}{\longrightarrow} \mathbb{R}) \\
&(\tfrac{f}{g})(x) := \tfrac{f(x)}{g(x)}; \\[1em]
&\cdot^{\cdot} : (\text{RealFun} \times \mathbb{Z}) \to (\mathbb{R} \stackrel{\text{partial}}{\longrightarrow} \mathbb{R}) \\
&(f^n)(x) := f(x)^n; \\[1em]
&\sqrt{\ } : (\mathbb{Z} \times \text{RealFun}) \to (\mathbb{R} \stackrel{\text{partial}}{\longrightarrow} \mathbb{R}) \\
&(\sqrt[n]{f})(x) := \sqrt[n]{f(x)};
\end{aligned}
$$

The first composition operation makes a function from a constant, e.g., the number 5 can be interpreted as the constant function $5(x) := 5$. The domain of a quotient function f/g is the set of all reals x such that $g(x) \neq 0$. The domain of f^n and of $\sqrt[n]{f}$ is $\mathbb{R}$, if $n \geq 0$. Otherwise, their domain is the set of all reals x such that $f(x) > 0$.

Example

- Let $f(x) := 2x^2$ and $g(y) := y + 4$. Then, for every $x \in \mathbb{R}$,

$$(\sqrt{f+g})(x) = \sqrt{2x^2 + x + 4}.$$

- The function $f(x) := 3x^2 + x\sqrt{x}$ equals

$$3 * g + h$$

where $g(x) := x^2$ and $h(x) = x\sqrt{x}$.

- The function $\frac{{1_{\mathbb{R}}}^2 + \sqrt{\ }}{1_{\mathbb{R}} + 1}$ equals the function

$$f(x) := \frac{x^2 + \sqrt{x}}{x+1}.$$

Integer Functions Various functions map real numbers to integers.

Definition 88 (Floor and Ceiling) The *floor* of a real number x is the largest integer less than or equal x.

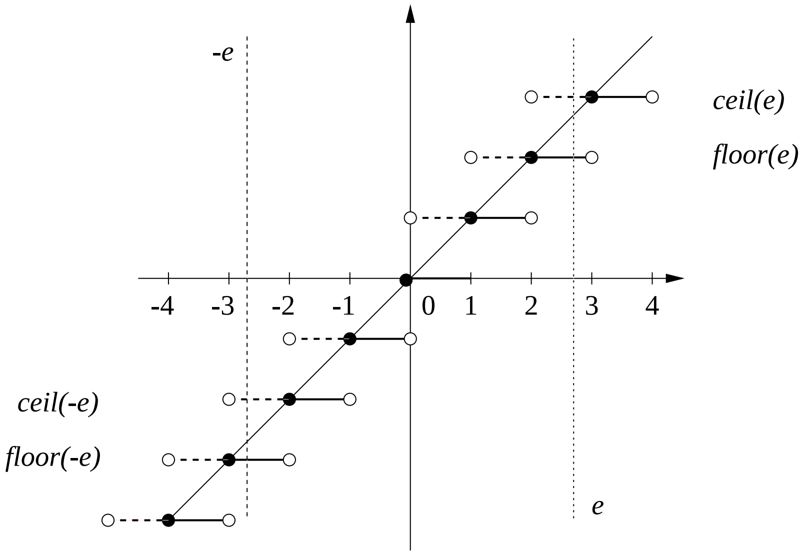
$$\lfloor x \rfloor := \max\{y \in \mathbb{Z} : y \leq x\}.$$

Analogously, the *ceiling* of x is the smallest integer greater than or equal x:

$$\lceil x \rceil := \min\{y \in \mathbb{Z} : y \geq x\}.$$

In above definition, we consider $\mathbb{Z}$ as a subset of $\mathbb{R}$ (instead of writing $h(\mathbb{Z})$ where h is the homomorphism that actually maps $\mathbb{Z}$ into $\mathbb{R}$).

The graphs of these functions are depicted by the following diagram (with the floor denoted by a solid line and the ceiling denoted by the dotted line):



We see that the floor function lies on or below the first diagonal while the ceiling lies above the diagonal. If we shift the diagonal line down one unit, it lies completely below the floor function; if we shift it up one unit, it lies completely above the floor function. Furthermore, functions intersect with each other at the diagonal exactly for the integer points. Finally, the functions are reflections of each other about both axes. We thus have the following three results.

Proposition 93 (Floor and Ceiling) For every $x \in \mathbb{R}$, we have:

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1,$$
$$\lfloor x \rfloor = x \Leftrightarrow x \in \mathbb{Z} \Leftrightarrow \lceil x \rceil = x,$$
$$\lfloor -x \rfloor = -\lceil x \rceil, \lceil -x \rceil = -\lfloor x \rfloor.$$

(We use in above proposition $A \Leftrightarrow B \Leftrightarrow C$ to denote $(A \Leftrightarrow B) \wedge (B \Leftrightarrow C)$).

Furthermore, we have the following property that allows us to shift integer terms out of a floor or a ceiling.

Proposition 94 (Floor and Ceiling Shifts) For every $x \in \mathbb{R}$ and $i \in \mathbb{Z}$, we have:

$$\lfloor x + i \rfloor = \lfloor x \rfloor + i,$$
$$\lceil x + i \rceil = \lceil x \rceil + i.$$

Please note that above rule does not hold for multiplication, i.e., there exist $x \in \mathbb{R}$ and $i \in \mathbb{Z}$ such that $\lfloor x * i \rfloor \neq \lfloor x \rfloor * i$ (e.g., $2 * \lfloor 0.5 \rfloor = 0 \neq 1 = \lfloor 2 * 0.5 \rfloor$).

The following rules help us to get rid of the floor and ceiling brackets under certain circumstances.

Proposition 95 (Floor and Ceiling Removal) For every $x \in \mathbb{R}$ and $i \in \mathbb{Z}$, we have:

$$\begin{array}{lcl} x < i & \Leftrightarrow & \lfloor x \rfloor < i, \\ i < x & \Leftrightarrow & i < \lceil x \rceil, \\ x \leq i & \Leftrightarrow & \lceil x \rceil \leq i, \\ i \leq x & \Leftrightarrow & i \leq \lfloor x \rfloor. \end{array}$$

Computer languages and pocket calculators sometimes provide operations to truncate the fractional part of a number.

Proposition 96 (Truncation) The *truncated part (abgeschnittener Teil)* of a real number is the number without its fractional part:

$$\mathrm{trunc}(x) := \textbf{if } x < 0 \textbf{ then } \lceil x \rceil \textbf{ else } \lfloor x \rfloor.$$

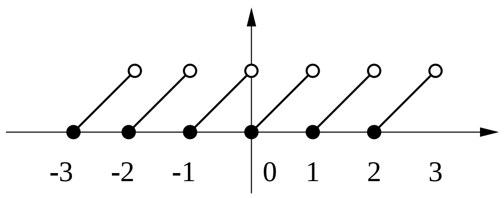
For this function, we have the following result.

Proposition 97 (Truncation) For every $x \in \mathbb{R}$, the truncated part of the negation of x is the negation of the truncated part of x:

$$\forall x \in \mathbb{R} : \mathrm{trunc}(-x) = -\mathrm{trunc}(x).$$

Apart from that, truncation does not have many nice mathematical properties (as we have for floor and ceiling), therefore it is rarely used in mathematical practice. A number of other interesting functions can be defined with the help of floor and ceiling.

Example The graph of $f(x) := x - \lfloor x \rfloor$ is depicted by



The graph in above example is an infinite replication of the graph in the interval $[0, 1[$, i.e., the corresponding function has period 1.

Definition 89 (Periodic Functions) A function has *period (Periode)* a, if the function values are repeated in intervals of width a:

$$\begin{aligned} &f \text{ has period } a :\Leftrightarrow \\ &\quad f : \mathbb{R} \to \mathbb{R} \wedge a \in \mathbb{R} \wedge \forall x \in \mathbb{R} : f(x+a) = f(x). \end{aligned}$$

A function is *periodic (periodisch)* if it has some period:

$$f \text{ is periodic } :\Leftrightarrow \exists a \in \mathbb{R} : f \text{ has period } a.$$

With the help of the floor function, we can also perform "integer division" (see Proposition 43 on page 109) of two reals x and y as $\lfloor x/y \rfloor$. A corresponding remainder operation is then defined as follows.

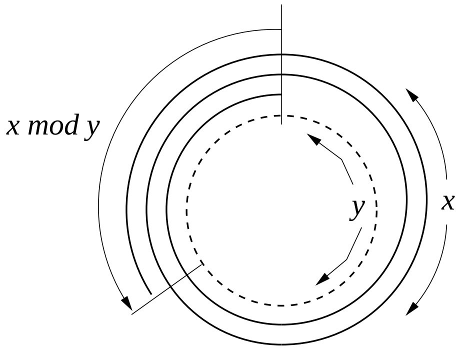
Definition 90 (Real Remainder)

$$\begin{aligned} &\text{mod} : (\mathbb{R} \times \mathbb{R}) \to \mathbb{R} \\ &x \text{ mod } y := x - y * \lfloor x/y \rfloor. \end{aligned}$$

As an immediate consequence of this definition, we have, for every $x \in \mathbb{R}$ and $y \in \mathbb{R}$ with $y \neq 0$,

$$x = y * \lfloor x/y \rfloor + x \text{ mod } y.$$

For positive x and y, the intuitive meaning of integer quotient and remainder in $\mathbb{R}$ can be easily grasped by imagining a circle of circumference y on which we travel from a denoted starting point distance x:



The number of times that we traverse the circle is $\lfloor x/y \rfloor$, the distance of the final point from the starting point is x mod y.

Example

- $\lfloor 5.44/4 \rfloor = 1$; $5.44 \bmod 4 = 5.44 - 4 * 1 = 1.44$.
- $\lfloor -5.44/4 \rfloor = -2$; $-5.44 \bmod 4 = -5.44 - 4 * (-2) = 2.56$.
- $\lfloor 5.44/-4 \rfloor = -2$; $5.44 \bmod -4 = 5.44 - (-4) * (-2) = -2.56$.
- $\lfloor -5.44/-4 \rfloor = 1$; $-5.44 \bmod -4 = -5.44 - (-4) * 1 = -1.44$.

From above examples, we realize the following facts about the remainder operation.

Proposition 98 (Remainder Laws) For every $x \in \mathbb{R}$ and $y \in \mathbb{R}$, we have

$$y > 0 \Rightarrow (0 \le x \bmod y < y),$$
$$y < 0 \Rightarrow (0 \ge x \bmod y > y).$$

Polynomial Functions and Rational Functions Functions may be defined by polynomials.

Definition 91 (Polynomial Function) For every $n \geq 0$ and $a : \mathbb{N}_{n+1} \to \mathbb{R}$, we call $p : \mathbb{R} \to \mathbb{R}, p(x) := \sum_{0 \leq i \leq n} a_i x^i$ a *polynomial function (Polynomfunktion)* of degree n with coefficients a:

p is polynomial function of degree n with coefficients a :⇔
  $n \in \mathbb{N} \wedge a : \mathbb{N}_{n+1} \to \mathbb{R} \wedge$
  $\forall x \in \mathbb{R} : p(x) = \sum_{0 \leq i \leq n} a_i$;
p is polynomial function :⇔
  $\exists n \in \mathbb{N}, a : p$ is polynomial function of degree n with coeff. a;
degree(p) := **such** $n \in \mathbb{N} : \exists a : \mathbb{N}_{n+1} \in \mathbb{R}$:
  p is polynomial function of degree n with coefficients a;
coefficients(p) := **such** $a : \exists n \in \mathbb{N}$:
  $a : \mathbb{N}_{n+1} \in \mathbb{R} \wedge$
  p is polynomial function of degree n with coefficients a.

Degree and coefficients of a polynomial function are uniquely defined (which we do not show here). Every polynomial function can be pointwise defined from identity and constant functions just by use of $+$, $-$, $*$ and exponentiation (with non-negative exponents); vice versa, any function defined in such a way is a polynomial function.

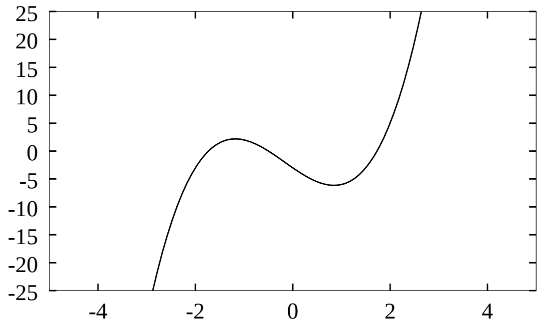
Example

- The *constant function (konstante Funktion)* $f(x) := c$ (for any $c \in \mathbb{R}$) is a polynomial function of degree 0 and with coefficients $[c]$.
- The *linear function (lineare Funktion)* $f(x) := ax + b$ (for any $a \in \mathbb{R}$ and $b \in \mathbb{R}$) is a polynomial function of degree 1 with coefficients $[a, b]$.
- The *square function (Quadratfunktion)* $f(x) := x^2$ is a polynomial function of degree 2 with coefficients $[1, 0, 0]$.
- $f(x) := x^5 - 3x^2 + 1$ is a polynomial function with degree 5 and coefficients $[5, 0, 0, -3, 0, 1]$.
- The function $f(x) := (2x+1)*(x^2-3)$ is a polynomial function because it can be represented (by simplification of the defining term) as

  $$f(x) := 2x^3 + x^2 - 6x - 3.$$

  It thus is of degree 3 and has coefficients $\langle 2, 1, -6, -3 \rangle$.

Every polynomial function has a function graph with a single "smooth" path such as the graph for the function in the last last example:



.

By allowing also division as a function composition operation, we get the following class of real functions:

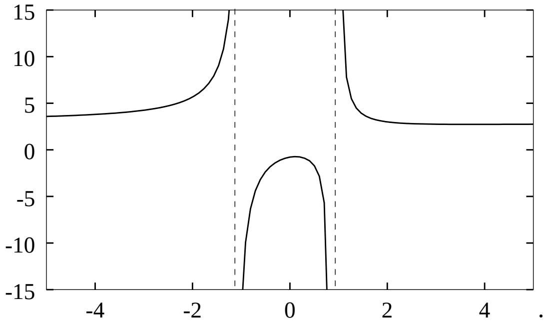
Definition 92 (Rational Function) For every pair of polynomial functions p and q, we call the function $r : R \to \mathbb{R}, r(x) := p(x)/q(x)$ with $R := \mathbb{R} - \{x \in \mathbb{R} : q(x) = 0\}$ a *rational function (rationale Funktion)*:

$$\begin{aligned} &r \text{ is a rational function} :\Leftrightarrow \exists p, q : \\ &\quad p \text{ is polynomial function} \wedge \\ &\quad q \text{ is polynomial function} \wedge \\ &\quad \forall x \in \mathbb{R} : r(x) = \tfrac{p(x)}{q(x)}. \end{aligned}$$

Example

- The *reciprocal function (Reziprokfunktion)* $f(x) := \frac{1}{x}$ is a rational function with domain $\mathbb{R} - \{0\}$.
- $f(x) := \frac{3x^2-2x+1}{x^2+1}$ is a rational function with domain $\mathbb{R}$.
- $f(x) := \frac{3x^2-2x+1}{x^2-1}$ is a rational function with domain $\mathbb{R} - \{-1, 1\}$.

Rational functions may have function graphs with multiple paths separated by those horizontal coordinates where the denominator becomes zero; e.g. for the rational function in the last example we have:



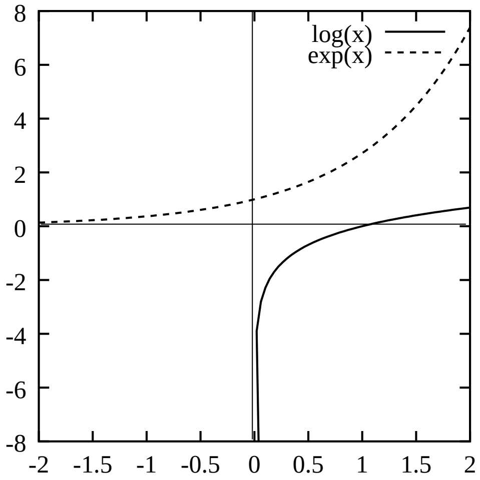
The dashed lines denote the discontinuities at -1 and 1 where the rational function is not defined.

Exponentiation and Natural Logarithm The sequence $[e^i]_i$ (where e denotes Euler's number) can be smoothly extended to a function over $\mathbb{R}$.

Definition 93 (Exponential Function and Logarithm Function)

$$\begin{array}{ll} \exp : \mathbb{R} \to \mathbb{R}_{>0} & \ln : \mathbb{R}_{>0} \to \mathbb{R} \\ \exp(x) := \sum_{i=0}^{\infty} \frac{x^i}{i!}; & \ln(x) := \exp^{-1}(x). \end{array}$$

One can show that, for every $x \in \mathbb{R}$, $[\sum_{0 \le i \le n} \frac{x^i}{i!}]_n$ converges and that $\exp(x)$ is thus well defined. Furthermore, the function is bijective and thus, for every $x > 0$, $\ln(x)$ is well-defined. The graphs of both functions are depicted as follows:



where we see for instance $\exp(0) = 1$ and $\ln(1) = 0$. More general, we have the following properties.

Proposition 99 (Exponential and Logarithm Properties) For every $x \in \mathbb{R}$ and $y \in \mathbb{R}$, we have:

$$\begin{array}{rcl@{\qquad}rcl}
\exp(x+y) & = & \exp(x) * \exp(y), & \ln(x*y) & = & \ln(x) + \ln(y), \\
\exp(x-y) & = & \exp(x)/\exp(y), & \ln(x/y) & = & \ln(x) - \ln(y), \\
\exp(-x) & = & 1/\exp(x), & \ln(1/x) & = & -\ln(x), \\
\exp(x*y) & = & \exp(x)^y, & \ln(x^y) & = & y * \ln(x), \\
\exp(0) & = & 1, & \ln(e) & = & 1, \\
\exp(1) & = & e, & \ln(1) & = & 0.
\end{array}$$

Proof Take arbitrary $x \in \mathbb{R}$ and $y \in \mathbb{R}$. We show

$$\exp(x+y) = \exp(x) * \exp(y).$$

We have

$$\begin{array}{rcl}
\exp(x+y) & = & \\
(\sum_{i=0}^{\infty} \frac{x^i}{i!}) + (\sum_{i=0}^{\infty} \frac{y^i}{i!}) & = & \\
\sum_{k=0}^{\infty} \sum_{0 \leq i \leq k} \frac{x^i}{i!} \frac{y^{k-i}}{(k-i)!} & = & \\
\sum_{k=0}^{\infty} \frac{1}{k!} \sum_{0 \leq i \leq k} \frac{k!}{i!(k-i)!} x^i y^{k-i} & = & \\
\sum_{k=0}^{\infty} \frac{1}{k!} \sum_{0 \leq i \leq k} \begin{pmatrix} k \\ i \end{pmatrix} x^i y^{k-i} & = & (*) \\
\sum_{k=0}^{\infty} \frac{(x+y)^k}{k!} & = & \\
\exp(x) + \exp(y). & &
\end{array}$$

(*) The fact $\sum_{0 \leq i \leq k} \begin{pmatrix} k \\ i \end{pmatrix} x^i y^{k-i} = (x+y)^k$ has to be shown in a separate proof (by induction over k).

Most properties about exponentiation are easy consequences of the first fact in above proposition, e.g. we have, for every $x \in \mathbb{R}$,

$$\exp(0) = \exp(x+0) = \exp(x) * \exp(0)$$

and consequently $\exp(0) = 1$. Properties of the logarithm are direct consequences of its definition as the inverse of exponentiation.

Proof Take arbitrary $x \in \mathbb{R}$ and $y \in \mathbb{R}$. We show

$$\ln(x * y) = \ln(x) + \ln(y).$$

We have

$$\begin{array}{rl} \ln(x) + \ln(y) & = \\ \ln(\exp(\ln(x) + \ln(y))) & = \\ \ln(\exp(\ln(x)) * \exp(\ln(y))) & = \\ \ln(x * y). & \end{array}$$

On the basis of "exp" and "ln" a more general kind of exponentiation and logarithm can be defined.

Definition 94 (General Exponentiation and Logarithm Function)

$$\begin{array}{ll} \exp : (\mathbb{R}_{>0} \times \mathbb{R}) \to \mathbb{R}_{>0} & \log : (\mathbb{R}_{>0} \times \mathbb{R}_{>0}) \to \mathbb{R} \\ \exp_a(x) := \exp(x * \ln(a)), & \log_a(x) := \ln(x)/\ln(a). \end{array}$$

Both functions are the inverse of each other, e.g. we have for every $a \in \mathbb{R}_{>0}$ and $x \in \mathbb{R}_{>0}$

$$\exp_a(\log_a(x)) = \exp_a(\ln(x)/\ln(a)) = (\ln(x)/\ln(a) * \ln(a) = x.$$

The graphs of generalized exponentiation and logarithm look like their basic counterparts but are scaled by a constant factor $\ln(a)$.

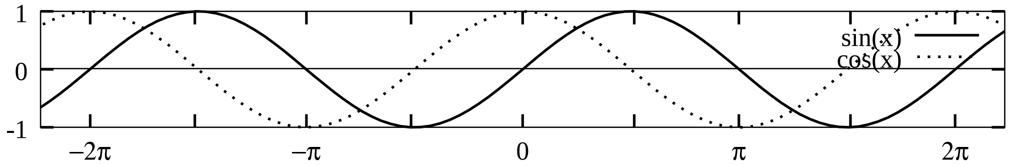
Trigonometric Functions The trigonometric functions are defined as limits of some power series.

Definition 95 (Trigonometric Functions) The functions *sine (Sinus)*, *cosine (Cosinus)*, *tangent (Tangens)*, and *cotangent (Cotangens)* are defined as follows:

$$\sin : \mathbb{R} \to [-1,1] \qquad \cos : \mathbb{R} \to [-1,1]$$
$$\sin(x) := \sum_{i=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}; \qquad \cos(x) := \sum_{i=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!};$$

$$\tan : \mathbb{R} \overset{\text{partial}}{\longrightarrow} \mathbb{R} \qquad \cot : \mathbb{R} \overset{\text{partial}}{\longrightarrow} \mathbb{R}$$
$$\tan(x) := \frac{\sin(x)}{\cos(x)}; \qquad \cot(x) := \frac{\cos(x)}{\sin(x)}.$$

The function graphs of sine and cosine are depicted as follows:



We see that the sine is zero for every multiple of π and alternates between its maximum $+1$ and minimum -1 at the points in the middles between; the function has period 2π. The cosine is just a copy of the sine shifted by $\frac{\pi}{2}$ to the left.

Proposition 102 (Sine and Cosine Values) For all $i \in \mathbb{Z}$, we have:

$$\begin{array}{rclrcl}
\sin(2\pi i) & = & 0, & \cos(2\pi i) & = & 1, \\
\sin(2\pi i + \pi/2) & = & 1, & \cos(2\pi i + \pi/2) & = & 0, \\
\sin(2\pi i + \pi) & = & 0, & \cos(2\pi i + \pi) & = & -1, \\
\sin(2\pi i + 3\pi/2) & = & -1, & \cos(2\pi i + 3\pi/2) & = & 0.
\end{array}$$

Many properties of sine and cosine can be derived from the following fundamental proposition (which we state without proof).

Proposition 103 (Sine and Cosine) For every $x \in \mathbb{R}$ and $y \in \mathbb{R}$, we have

$$\begin{array}{rcl}
\sin(x+y) & = & \sin(x) * \cos(y) + \cos(x) * \sin(y), \\
\cos(x+y) & = & \cos(x) * \cos(y) - \sin(x) * \sin(y).
\end{array}$$

We then have the following results.

Proposition 104 (Sine and Cosine) For every $x \in \mathbb{R}$ and $y \in \mathbb{R}$, we have:

$$\begin{aligned}
\sin(x+2\pi) &= \sin(x), & \cos(x+2\pi) &= \cos(x),\\
\sin(x+\tfrac{\pi}{2}) &= \cos(x), & \cos(x+\tfrac{\pi}{2}) &= -\sin(x),\\
\sin(\tfrac{\pi}{2}-x) &= \cos(x), & \cos(\tfrac{\pi}{2}-x) &= -\sin(x),\\
\sin(-x) &= -\sin(x), & \cos(-x) &= \cos(x),\\
\sin(2x) &= 2\sin(x)\cos(x), & \cos(2x) &= \cos^2(x)-\sin^2(x).
\end{aligned}$$

$$\cos^2(x)+\sin^2(x)=1,$$

$$\begin{gathered}
2\ \sin(x)*\sin(y)=\cos(x-y)-\cos(x+y),\\
2\ \cos(x)*\cos(y)=\cos(x-y)+\cos(x+y),\\
2\ \sin(x)*\cos(y)=\sin(x-y)+\sin(x+y).
\end{gathered}$$

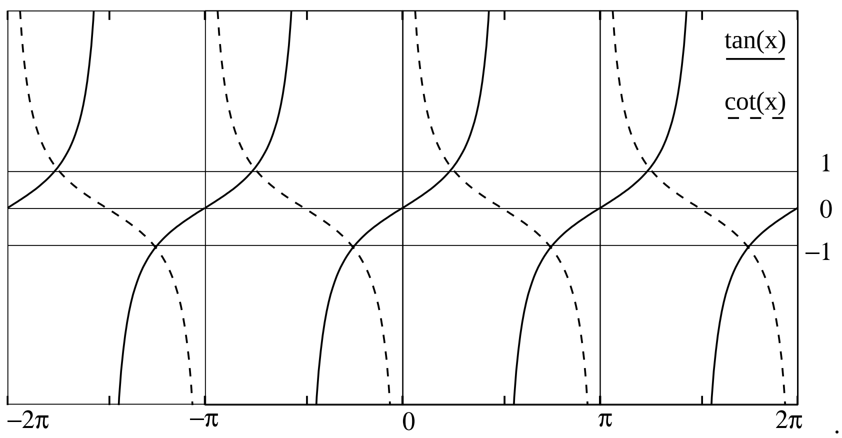
Proof Take arbitrary $x \in \mathbb{R}$. We show

$$\sin(x+2\pi)=\sin(x).$$

We have

$$\begin{aligned}
&\sin(x+2\pi)=\sin(x)*\cos(2\pi)+\cos(x)*\sin(2\pi)\\
&\quad=\sin(x)*1+\cos(x)*0=\sin(x).
\end{aligned}$$

The function graphs of tangent and cotangent look as follows:



.

The tangent function has period π, its value is 0 at every multiple of π and undefined at every odd multiple of $\frac{\pi}{2}$; furthermore the function is unbounded in both directions. The cotangent is the tangent horizontally flipped and shifted left by $\frac{\pi}{2}$. Both functions meet at odd multiples of $\frac{\pi}{4}$ at values $+/-1$.

Proposition 106 (Tangent and Cotangent Values) For all $i \in \mathbb{Z}$, we have:

$$\begin{array}{rclrcl} \tan(\pi i) & = & 0, & \cot(\pi i + \frac{\pi}{2}) & = & 0, \\ \tan(\pi i + \frac{\pi}{4}) & = & 1, & \cot(\pi i + \frac{\pi}{4}) & = & 1, \\ \tan(\pi i - \frac{\pi}{4}) & = & -1, & \cot(\pi i - \frac{\pi}{4}) & = & -1. \end{array}$$

Many properties of tangent and cotangent can be derived from the following proposition (which we state without proof).

Proposition 107 (Tangent and Cotangent) For every $x \in \mathbb{R}$ and $y \in \mathbb{R}$, we have

$$\tan(x+y) = \frac{\tan(x) + tan(y)}{1 - \tan(x) * \tan(y)},\ \cot(x+y) = \frac{\cot(x) * \cot(y) - 1}{\cot(x) + \cot(y)}.$$

We than also have the following results.

Proposition 108 (Tangent and Cotangent Properties) For every $x \in \mathbb{R}$ and $y \in \mathbb{R}$, we have:

$$\begin{array}{rclrcl} \tan(-x) & = & -\tan(x), & \cot(-x) & = & -\cot(x), \\ \tan(x) & = & -\cot(x + \frac{\pi}{2}), & \tan(x) & = & -\tan(x - \frac{\pi}{2}). \end{array}$$

Proof Take arbitrary $x \in \mathbb{R}$. We show

$$\tan(-x) = -\tan(x).$$

We have $\tan(-x) = \frac{\sin(-x)}{\cos(-x)} = \frac{-\sin(x)}{\cos(x)} = -\frac{\sin(x)}{\cos(x)} = -\tan(x)$.

6.6 Asymptotic Bounds

When discussing the execution time of an algorithm, we are often sloppy by stating that an algorithm takes, for problem size n, not more than n^2 steps while it may actually run in $7n^2 + 3n + 19$ machine instructions. However, in some sense the sequence $[n^2]_n$ already bounds the growth rate of $[7n^2 + 3n + 19]_n$; the factor 7. the addendum 19, and even the term $3n$ do not contribute to this rate any more. This concept of "asymptotic function bounds" is generally formalized as follows.

Definition 96 (Big O Quantor) For every variable n and terms S and T, the phrase

$$S = \mathrm{O}_n(T)$$

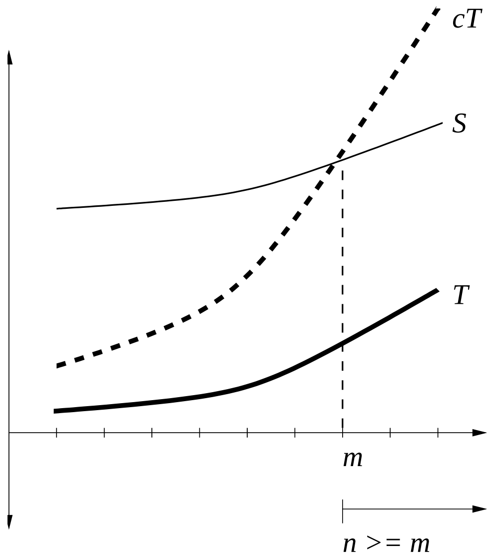
is a proposition with bound variable n which is read as "S is big O of T" or "*T asymptotically dominates (dominiert asymptotisch) S*".

Its meaning is equivalent to the proposition

$$\exists c \in \mathbb{R}, m \in \mathbb{N} : \forall n \geq m : |S| \leq c * |T|.$$

Usually the subscript n is dropped and the bound variable has to be deduced from the context.

The proposition $S = \mathrm{O}_n(T)$ states that, from a certain point m on and scaled by some factor c, the absolute value of T is at least as large as the absolute value of S. The function $[T]_n$ thus grows at least as fast as $[S]_n$.



For instance, instead of saying that the execution time T of some algorithm A in dependence on input size n is $T_A(n) = 7n^2 + 3n + 19$, we may say

$$T_A(n) = \mathrm{O}(n^2),$$

i.e., that the execution time is asymptotically dominated by the square function and ignore all constant factors and minor terms.

Sometimes, we also see propositions like

$$S_0 = S_1 + \mathrm{O}(T)$$

which is to be understood as $S_0 - S_1 = \mathrm{O}(T)$. For instance, by saying

$$f(n) = 5n^3 + 2n^2 + \mathrm{O}(n)$$

we state that $f(n)$ differs from $5n^3+2n^2$ not more than by some linear factor.

Example

- We have $10n + 100 = \mathrm{O}(n)$:

  Let $c := 110$ and $m := 1$ and take arbitrary $n \geq m$. We have to show

  $$|10n + 100| \leq c * |n|.$$

  We know

  $$|10 * n + 100| = 10 * n + 100 \leq 110n = 110|n| = c|n|.$$

- We have *not* $n^2 = \mathrm{O}(n)$:

  We suppose $n^2 = \mathrm{O}(n)$ and show a contradiction. By the assumption, we have some $c \in \mathbb{R}$ and $m \in \mathbb{N}$ with

  $$\forall n \geq m : |n^2| \leq c|n|.$$

  If $c < 0$, let $k := \max(1, m)$; then we have $k \geq m$ but

  $$|k^2| > 0 > c|k|$$

  and thus a contradiction.

  If $c \geq 0$, let $k := \max(m, \lceil c + 2 \rceil)$. Then we have $k \geq m$ but

  $$\begin{aligned} |k^2| &= k^2 \geq \lceil c+2 \rceil^2 \geq (c+2)^2 = c^2 + 4c + 4 \\ &> c^2 + 3c = c(c+3) \geq c\lceil c+2 \rceil \geq ck = c|k| \end{aligned}$$

  and thus a contradiction.

- We have $n^2 = \mathrm{O}(2^n)$:

  We assume *not* $n^2 = \mathrm{O}(2^n)$ and show a contradiction. By assumption, we have $\neg\exists c \in \mathbb{R}, m \in \mathbb{N} : \forall n \geq m : |n^2| \leq c|2^n|$, i.e.,

  $$\forall c \in \mathbb{R}, m \in \mathbb{N} : \neg\forall n \geq m : |n^2| \leq c|2^n|.$$

  We prove

  $$\forall n \geq 0 : n^2 \leq 2 * 2^n$$

  by induction on n and thus have a contradiction.

  If $n = 0$, we have $n^2 = 0 < 2 = 2 * 2^0 = 2 * 2^n$.

  We assume $n \geq 1 \Rightarrow n^2 \leq 2 * 2^n$ and show

  $$n + 1 \geq 1 \Rightarrow (n+1)^2 \leq 2 * 2^{n+1}.$$

  Assume $n + 1 \geq 1$. We show

  $$(n+1)^2 \leq 2 * 2^{n+1}.$$

  If $n < 1$, then $n = 0$ and

  $$(n+1)^2 = (0+1)^2 = 1 < 4 = 2 * 2^{0+1} = 2 * 2^{n+1}.$$

  If $n \geq 1$, then, by the induction hypothesis, $n^2 \leq 2 * 2^n$. We then have

  $$\begin{aligned}(n+1)^2 &= n^2 + 2n + 1 \leq n^2 + 2^n + 2^n \ (*)\\ &\leq 2 * 2^n + 2^n + 2^n = 2^{n+1} + 2^{n+1} = 2(2^{n+1}).\end{aligned}$$

  (*) The fact $2n \leq 2^n$ has to be shown in a separate proof.

It is easy to derive a number of laws for the manipulation of O terms.

Proposition 110 (O Manipulation) For every variable n, terms S and T, $c \in \mathbb{R}$, and $d \in \mathbb{R}$ with $d \geq c$, we can replace the following terms on the left hand side by the corresponding terms on the right hand side:

$$
\begin{array}{rcl}
S & \to & \mathrm{O}_n(S), \\
c + \mathrm{O}_n(S) & \to & \mathrm{O}_n(S), \\
c * \mathrm{O}_n(S) & \to & \mathrm{O}_n(S), \\
\mathrm{O}_n(S) + \mathrm{O}_n(S) & \to & \mathrm{O}_n(S), \\
\mathrm{O}_n(\mathrm{O}_n(S)) & \to & \mathrm{O}_n(S), \\
\mathrm{O}_n(S) + \mathrm{O}_n(T) & \to & \mathrm{O}_n(|S| + |T|), \\
\mathrm{O}_n(S) * \mathrm{O}_n(T) & \to & \mathrm{O}_n(S * T), \\
\mathrm{O}_n(S) * \mathrm{O}_n(T) & \to & S * \mathrm{O}_n(T), \\
\mathrm{O}_n(S^c) & \to & \mathrm{O}_n(S^d).
\end{array}
$$

Thus we can replace a statement

$$f(n) = 2n^2 - 7n + 5$$

via the transformations

$$
\begin{array}{l}
2n^2 - 7n + 5 \to 2 * \mathrm{O}(n^2) - 7 * \mathrm{O}(n) + 5 * \mathrm{O}(1) \to \\
\quad \mathrm{O}(n^2) + O(n) + O(1) \to \mathrm{O}(n^2) + \mathrm{O}(n^2) + \mathrm{O}(n^2) \to \mathrm{O}(n^2)
\end{array}
$$

by the statement

$$f(n) = \mathrm{O}(n^2).$$

This example illustrates that the asymptotic behavior of a polynomial sequence is asymptotically dominated by the polynomial term with the highest degree.

Proposition 111 (Asymptotic of Polynomial Sequences) Every polynomial sequence of degree n is dominated by $[x^n]_x$:

$$
\begin{array}{l}
\forall n \in \mathbb{N}, a : \mathbb{N}_n \to \mathbb{R} : \\
\quad \sum_{0 \leq i \leq n} a x^i = \mathrm{O}_n(x^n).
\end{array}
$$

For comparing the asymptotic behavior of classes of real functions, the following notation comes handy.

Definition 97 (Strictly Dominated) For every variable n and terms S and T, the phrase

$$S \prec_n T$$

is a proposition with bound variable n which is read as "S is strictly dominated by T" and that is equivalent to

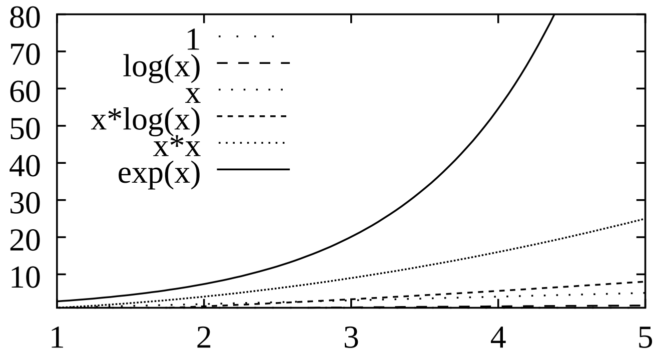
$$S = \mathrm{O}_n(T) \wedge T \neq O_n(S).$$

We then have the following chain of function classes:

Proposition 112 (Asymptotic Classes) For every $k \in \mathbb{N}$, $c \in \mathbb{R}$, and $a \in \mathbb{R}_{>0}$, we have:

$$1 \prec_n \log_a(n) \prec_n n \prec_n n\log_a(n) \prec_n n^k \prec_n c^n \prec_n n!$$

The corresponding function graphs are visualized below.



To get some "feeling" for the growth rate of these functions, assume that we have a machine that performs 10^6 operations per second. We are given six algorithms whose number of execution steps in dependence of the problem size n is determined by one of the six functions denoted above. Then the maximum size of a problem that can be solved in a fixed amount of time is

Execution Steps	1 second	1 minute	1 hour
$\log_2(n)$	2^{10^6}	2^{6*10^7}	2^{36*10^8}
n	10^6	$6*10^7$	$3.6*10^9$
$n\log_2(n)$	62746	$2.8*10^6$	$1.3*10^8$
n^2	1000	7746	60000
2^n	23	26	32
$n!$	9	11	12

A sorting algorithm that takes, for some logarithmic base, $O(n\log(n))$ steps (such as Quicksort) is therefore much better than an n^2 algorithm (such as Insertion Sort); the difference in computation time may be not significant for small inputs but the larger the inputs become, the more the asymptotic complexity distinguishes the algorithms.

Also we can see that, for algorithms with exponential (or higher) complexity, the only problems of extremely small size can be solved. Furthermore, it does not help much to allow say 1000 times more execution time (or use a 1000 times faster machine or apply a parallel computer with 1000 processors); the manageable problem size does not increase significantly. Problems for which only exponential time algorithms exist are therefore called *intractable (ungefügig)*. For instance, to find out whether a propositional formula with n variables is a tautology is such an intractable problem.

Chapter 7

More on Relations

In Chapter 3, we have introduced relations as subsets of Cartesian products. In this chapter, we focus on binary relations and demonstrate some important applications of this notion. First, we introduce equivalence relations as a tool to partition sets into classes and show how in this way modular arithmetic and various number domains can be defined. Then we deal with relations that help us to impose order structures on sets; finally we show how graphs can be formalized as binary relations.

7.1 Equivalence Relations and Partitions

7.1.1 Basic Notions

Equivalence relations are binary relations that have many (but not all) of the properties of the equality relation. First we name these properties.

Definition 98 (Relation Properties) R is *reflexive (reflexiv)*, *symmetric (symmetrisch)*, respectively *transitive (transitiv)* on a set S, if it satisfies the following properties:

R is reflexive on $S :\Leftrightarrow$
$\quad \forall x \in S : \langle x, x\rangle \in R$;
R is symmetric on $S :\Leftrightarrow$
$\quad \forall x, y : \langle x, y\rangle \in R \Rightarrow \langle y, x\rangle \in R$;
R is transitive on $S :\Leftrightarrow$
$\quad \forall x, y, z : (\langle x, y\rangle \in R \wedge \langle y, z\rangle \in R) \Rightarrow \langle x, z\rangle \in R$.

Definition 99 (Equivalence Relation) Let R be a binary relation on S. R is an *equivalence relation (Äquivalenzrelation)* on S, if it is reflexive, symmetric, and transitive on S:

$$\begin{array}{l} R \text{ is equivalence relation on } S :\Leftrightarrow \\ \quad R \subseteq S \times S \land \\ \quad R \text{ is reflexive on } S \land \\ \quad R \text{ is symmetric on } S \land \\ \quad R \text{ is transitive on } S. \end{array}$$

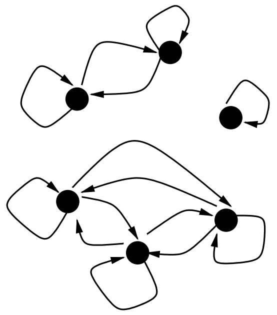
Example

- $=$ (equality) is an equivalence relation on every set.
- Let $p(x, y) :\Leftrightarrow x + y$ is even. Then p is an equivalence relation on $\mathbb{N}$.
- Let $q(x, y) :\Leftrightarrow x \bmod 5 = \text{y} \bmod 5$. Then q is an equivalence relation on $\mathbb{N}$.
- Let $r(x, y) :\Leftrightarrow x_0 + y_0 = x_1 + y_1$. Then r is an equivalence relation on $\mathbb{R} \times \mathbb{R}$.
- Let $s(x, y) :\Leftrightarrow x$ is parallel to (or coincides with) y. Then s is an equivalence relation on the set of all lines in the plane.
- Let $t(x, y) :\Leftrightarrow x$ has the same age as y. Then t is an equivalence relation on the set of all people.

The visualization of an equivalence relation by a directed graph has the following properties:

- every node has an arrow to itself (reflexivity),
- if there is an arrow from node a to node b, then there is also an arrow from b to a (symmetry),
- if there is an arrow from node a to node b and an arrow from b to some node c, then there is also an arrow from a to c (transitivity).

Example The following graph denotes an equivalence relation:



The equivalence relation imposed on the set of six nodes determines three partitions in each of which every node is connected to every other node. Actually, there is a close relationship between equivalence relations and set partitions that will be elaborated later.

We demonstrate how to prove that a relation is an equivalence relation.

Proof Let $p(x, y) :\Leftrightarrow x + y$ is even. We prove that p is an equivalence relation on $\mathbb{N}$.

1. p is clearly a binary relation on $\mathbb{N}$.

2. We prove p is reflexive on $\mathbb{N}$. Take arbitrary $x \in \mathbb{N}$. We have to show $x + x$ is even, i.e, $2x$ is even, which clearly holds.

3. We prove p is symmetric on $\mathbb{N}$. Take arbitrary $x \in \mathbb{N}$ and $y \in \mathbb{N}$. We assume $x + y$ is even. Then $y + x$ is even, because $x + y = y + x$.

4. We prove p is transitive on $\mathbb{N}$. Take arbitrary $x \in \mathbb{N}$, $y \in \mathbb{N}$, and $z \in \mathbb{N}$. We assume

   (1) $x + y$ is even $\wedge$ $y + z$ is even.

   We have to show

   (2) $x + z$ is even

From (1), we have some $a \in \mathbb{N}$ and $b \in \mathbb{N}$ such that

(3) $2a = x + y \wedge 2b = y + z$.

Thus we know

$$x + z = (x + y) + (y + z) - 2y = 2a + 2b - 2y = 2(a + b - y)$$

such that (2) holds.

Logic Evaluator Above definitions can be implemented as shown below, see Appendix C.10 (file `equiv.txt`).

```
option silent = true; read relation;

pred isReflexive(R: Relation, S) <=>
  forall(x in S: in(tuple(x, x), R));
pred isSymmetric(R: Relation, S) <=>
  forall(t in R: in(tuple(.1(t), .0(t)), R));
pred isTransitive(R: Relation, S) <=>
  forall(s in R, t in R:
    implies(=(.1(s), .0(t)), in(tuple(.0(s), .1(t)), R)));
pred isEquivalence(R: Relation, S) <=>
  and(isRelation(R, S, S),
      isReflexive(R, S),
      isSymmetric(R, S),
      isTransitive(R, S));

fun S(n) = set(x in nat(0, n): true, x);
fun j(x, y, S) = join(tuple(x, y), S);
fun R0 = j(0, 0, j(1, 1, j(2, 2, j(3, 3, {}))));
fun R1 = j(0, 1, j(1, 0, j(1, 2, j(3, 3, R0))));
fun R2 = j(0, 2, R1);
fun R3 = j(2, 0, j(2, 1, R2));

formula isTransitive(R1, S(3));
> false.
formula isTransitive(R2, S(3));
> true.
formula isSymmetric(R2, S(3));
> false.
```

```
formula isEquivalence(R3, S(3));
```

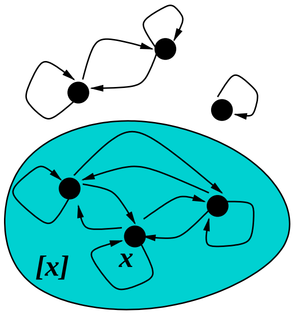
We may construct the group of all objects that are related to some object.

Definition 100 (Class) The *class (Klasse)* of x with respect to R is the set of all elements that are related to x by R:

$$[x]_R := \{y \in \text{range}(R) : \langle x, y \rangle \in R\}.$$

We may just write $[x]$, if R is clear from the context. If R is an equivalence relation, we call $[x]_R$ the *equivalence class (Äquivalenzklasse)* of x with respect to R.

In the visualization of an (equivalence) relation by a directed graph, the (equivalence) class of a node is the set of all nodes to which the node is connected:



By the reflexivity of an equivalence relation, we have the following result.

Proposition 114 (Non-Empty Equivalence Classes) If R is an equivalence relation on S, then $[x]_R$ contains x, for every $x \in S$:

$$\forall S, R : R \text{ is equivalence class on } R \ \Rightarrow x \in [x]_R.$$

Example

- Let $p \subseteq \mathbb{N} \times \mathbb{N}$ such that $p(x, y) \Leftrightarrow x + y$ is even. Then we have

$$
\begin{array}{lcl}
[0]_p & = & \{0, 2, 4, 6, 8, 10, \ldots\}, \\
[1]_p & = & \{1, 3, 5, 7, 9, 11, \ldots\}, \\
[2]_p & = & \{0, 2, 4, 6, 8, 10, \ldots\}, \\
[3]_p & = & \{1, 3, 5, 7, 9, 11, \ldots\}, \\
[4]_p & = & \{0, 2, 4, 6, 8, 10, \ldots\}, \\
\ldots & &
\end{array}
$$

We see that $[0]_p \cup [1]_p = \mathbb{N}$ and $[0]_p \cap [1]_p = \emptyset$.

- Let $q \subseteq \mathbb{N} \times \mathbb{N}$ such that $q(x, y) \Leftrightarrow x \bmod 5 = \mathrm{y} \bmod 5$. Then we have

$$
\begin{array}{lcl}
[0]_q & = & \{0, 5, 10, 15, 20, 25, \ldots\}, \\
[1]_q & = & \{1, 6, 11, 16, 21, 26, \ldots\}, \\
[2]_q & = & \{2, 7, 12, 17, 22, 27, \ldots\}, \\
[3]_q & = & \{3, 8, 13, 18, 23, 28, \ldots\}, \\
[4]_q & = & \{4, 9, 14, 19, 24, 29, \ldots\}, \\
[5]_q & = & \{0, 5, 10, 15, 20, 25, \ldots\}, \\
\ldots & &
\end{array}
$$

We see that $[0]_q \cup [1]_q \cup [2]_q \cup [3]_q \cup [4]_q = \mathbb{N}$ and that any two of these sets are disjoint.

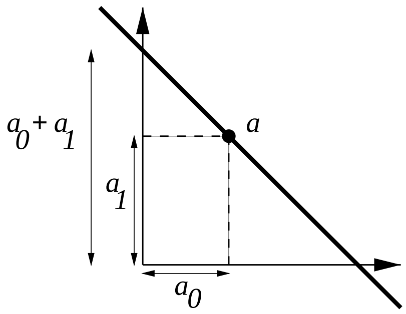
- Let $r \subseteq \mathbb{R} \times \mathbb{R}$ such that $r(x, y) \Leftrightarrow x_0 + x_1 = y_0 + y_1$. Then we have

$$[a]_r = \{b \in \mathbb{R} \times \mathbb{R} : a_0 + a_1 = b_0 + b_1\}$$

i.e.,

$$[a]_r = \{b \in \mathbb{R} \times \mathbb{R} : b_1 = -b_0 + (a_0 + a_1)\}$$

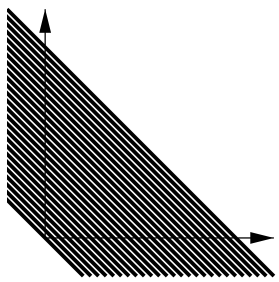
If we consider $\mathbb{R} \times \mathbb{R}$ as the set of all points in the plane, then $[a]_r$ denotes the line with slope -1 that goes through a:



We thus have

$$\mathbb{R} \times \mathbb{R} = \bigcup_{a \in \mathbb{R} \times \mathbb{R}} [a]_r$$

i.e., the plane is partitioned into the set of all these lines:



We can determine a "canonical representative" for each such line (the point on this line whose first coordinate is zero) and thus have

$$\mathbb{R} \times \mathbb{R} = \bigcup_{y \in \mathbb{R}} [\langle 0, y \rangle]_r$$

As illustrated by the example, different equivalences classes do not overlap.

Proposition 115 (Disjoint Equivalence Classes) Let R be an equivalence relation on S and x and y be elements of S. The equivalence classes of x and y with respect to R are either identical or disjoint:

$$\begin{array}{l} \forall S, R : R \text{ is equivalence relation on } S \;\Rightarrow \\ \quad \forall x \in S, y \in S : \\ \qquad [x]_R = [y]_R \vee [x]_R \cap [y]_R = \emptyset. \end{array}$$

Proof Take arbitrary S, an equivalence relation R on S, $x \in S$ and $y \in S$. We assume

(1) $[x]_R \neq [y]_R$,
(2) $[x]_R \cap [y]_R \neq \emptyset$

and show a contradiction. From (2), we have some z such that

(3) $z \in [x]_R \wedge z \in [y]_R$.

We thus know

(4) $\langle x, z\rangle \in R$,
(5) $\langle y, z\rangle \in R$.

We will now show

(6) $[y]_R \subseteq [x]_R$,
(7) $[x]_R \subseteq [y]_R$

which contradicts (1).

We show (6): From (5) and the symmetry of R, we know

(8) $\langle z, y\rangle \in R$.

From (4), (8), and the transitivity of R, we know

(9) $\langle x, y\rangle \in R$.

From (9) we know by the transitivity of R that

(10) $\forall z : \langle y, z\rangle \in R \Rightarrow \langle x, z\rangle \in R$

and therefore by definition of $[y]_R$

(11) $\forall z \in [y]_R : \langle x, z\rangle \in R$

and thus by definition of $[x]_R$

(12) $\forall z \in [y]_R : z \in [x]_R$

which gives us (6).

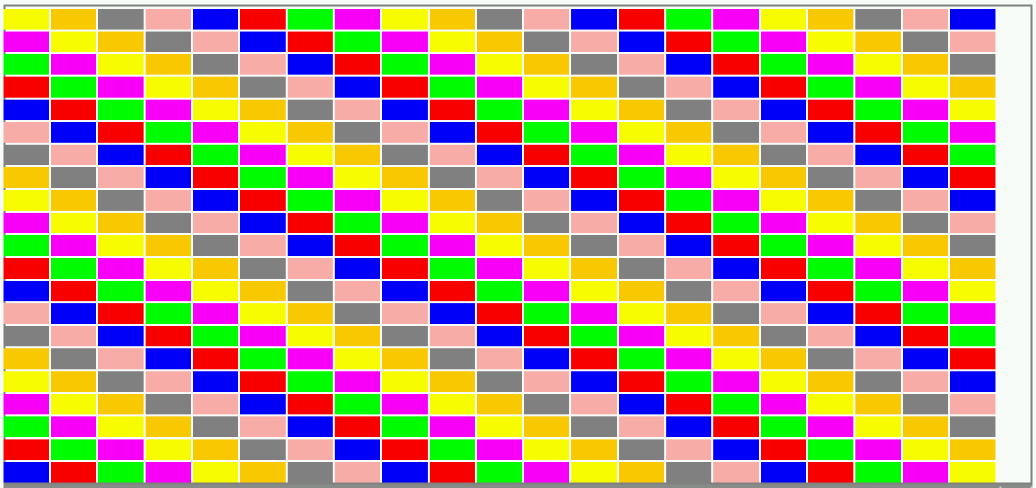
With the same line of reasoning, we can show (7) and are done.

As a consequence of Propositions 114 and 115, we can decompose every set S by an equivalence relation into a set of non-empty disjoint subsets.

Definition 101 (Quotient Set) The *quotient set (Quotientenmenge)* of S with respect to R is the set of all classes induced on S by R:

$$S/R := \{[x]_R : x \in S\}.$$

Logic Evaluator The quotient set construction can be implemented as shown below, see Appendix C.10 (file `equiv.txt`).



```
option silent = true; read relation;

fun [](x, R: Relation) = // class [x]_R
  set(y in range(R): in(tuple(x, y), R), y);
fun /(S, R: Relation) =  // quotient set S/R
  set(x in S: true, [](x, R));

fun n = 20;fun D = set(x in nat(0, n): true, x);
fun S = set(x in D, y in D: true, tuple(x, y));
pred r(x, y) <=> =(+(.0(x), .1(x)), +(.0(y), .1(y)));
fun R = set(x in S, y in S: r(x, y), tuple(x, y));
fun C = /(S, R);

fun [](x) = set(y in S: r(x, y), y);
fun C' = set(x in nat(0,*(2,n)): true, [](tuple(0, x)));
plots C';
```

```
fun n = 4; term C; term C';
```

We use in this example a relation on $S := \mathbb{N}_n \times \mathbb{N}_n$ whose elements are interpreted as points in a plane. Since the computation of $C := S/R$ takes too long, we resort to an equivalent but faster construction C' where a representative of each equivalence class is manually selected, and the relation R as a set has been replaced by predicate r. Each color in the resulting picture denotes an equivalence class on S.

We expect this decomposition to satisfy the following properties.

Definition 102 (Partition) D is a *partition (Partitionierung)* or *decomposition (Zerlegung)* of S, if its elements, the *blocks (Blöcke)*, are non-empty and disjoint and their union equals S:

$$\begin{array}{l} D \text{ is partition of } S :\Leftrightarrow \\ \quad (\forall x \in D : x \neq \emptyset) \wedge \\ \quad (\forall x \in D, y \in D : x = y \vee x \cap y = \emptyset) \wedge \\ \quad \bigcup D = S. \end{array}$$

Example

- We define

$$\begin{array}{lcl} A & := & \{x \in \mathbb{N} : x \text{ is even}\}, \\ B & := & \{x \in \mathbb{N} : x > 2 \wedge x \text{ is prime}\}, \\ B & := & \{x \in \mathbb{N} : x \text{ is odd} \wedge x \text{ is not prime}\}. \end{array}$$

  Then the set $\{A, B, C\}$ is a partition of $\mathbb{N}$.

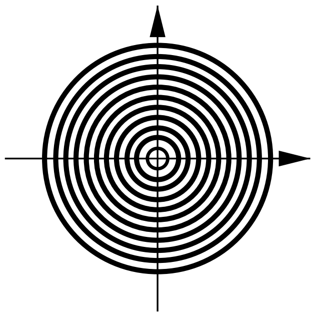
- We define

$$\text{circle}(r) := \{p \in \mathbb{R} \times \mathbb{R} : {p_0}^2 + {p_1}^2 = r^2\}.$$

  Then the set

$$\{\text{circle}(r) : r \in \mathbb{R}_{\geq 0}\}$$

  is a partition of $\mathbb{R} \times \mathbb{R}$ as denoted by the following picture:



The following result shows that an equivalence relation indeed defines a partition.

Proposition 117 (Equivalence Relation Defines Partition) Let R be an equivalence relation on S. The quotient set of S with respect to R is a partition of S:

$$\begin{aligned}&\forall S, R : R \text{ is equivalence relation on } S \;\Rightarrow\\ &\quad S/R \text{ is partition of } S.\end{aligned}$$

Proof Take arbitrary S and equivalence relation R on S. We show S/R is a partition of S:

1. Take arbitrary $x \in S/R$. By definition of S/R, we have some $a \in S$ such that $x = [a]_R$. By Proposition 114, $a \in [a]_R$, therefore $x \neq \emptyset$.

2. Take arbitrary $x \in S/R$ and $y \in S/R$. By definition of S/R, we have some $a \in S$ and $b \in S$ such that $x = [a]_R$ and $y = [b]_R$. By Proposition 115, $[a]_R = [b]_R \vee [a]_R \cap [b]_R = \emptyset$.

3. By definition of S/R, $\bigcup S/R \subseteq S$. We show $S \subseteq \bigcup S/R$. Take arbitrary $x \in S$. Then from Proposition 114 we have $x \in [x]_R$ and from the definition of S/R we have $[x]_R \in S/R$, thus $x \in \bigcup S/R$.

We can also proceed in the other direction and construct a relation from a partition.

Definition 103 (Induced Relation) The relation induced by a partition D is the set of all pairs of elements of the same block of D:

$$x \sim_D y :\Leftrightarrow \exists d \in D : x \in d \wedge y \in d.$$

The constructed relation is an equivalence relation.

Proposition 119 (Partition Defines Equivalence Relation) Let D be a partition of S. The relation induced by D is an equivalence relation on S:

$$\forall S, D : D \text{ is partition of } S \ \Rightarrow$$
$$\sim_D \text{ is equivalence relation on } S.$$

Proof Take arbitrary S and partition D on S. We show that $\sim_D$ is an equivalence relation on S.

Binary Relation By definition, $\sim_D$ is a binary relation. If $x \sim_D y$, then we have some $d \in D$ such that $x \in d$ and $y \in d$. Since $\bigcup D = S$, we have $x \in S$ and $y \in S$ and $\sim_D$ is a relation on S.

Reflexivity Take arbitrary $x \in S$. Since $\bigcup D = S$, we have some $d \in D$ such that $x \in d$ and thus $x \sim_D x$.

Symmetry Take arbitrary $x \in S$ and $y \in S$ and assume $x \sim_D y$. Then we have some $d \in D$ such that $x \in d \wedge y \in d$ which gives us $y \sim_d x$.

Transitivity Take arbitrary $x \in S$, $y \in S$, and $z \in S$ and assume $x \sim_D y$ and $y \sim_D z$. Then we have some $d \in D$ such that $x \in d \wedge y \in d$ and some $e \in D$ such that $y \in d \wedge z \in d$. Since $y \in d$ and $y \in e$, Proposition 115 gives us $d = e$ and thus $x \sim_D z$.

As shown above, equivalence relations and partitions are just different notions for the same concept; we can always construct from one description the other one. Actually, each construction is the inverse of the other one as stated by the following result (which can be easily proved).

Proposition 121 (Equivalence Relations and Partitions) Let R be an equivalence relation on S. Then R is the relation induced by the quotient set of S with respect to R:

$$\forall S, R : R \text{ is equivalence relation on } S \;\Rightarrow\; R = \sim_{S/R} .$$

Let D be a partition of S. Then D is the quotient set of the relation induced by D:

$$\forall S, D : D \text{ is partition of } S \;\Rightarrow\; D = S/\sim_D .$$

Logic Evaluator We can implement the new notions as shown below, see Appendix C.10 (file `equiv.txt`).

```
option silent = true; read equiv;

pred isPartition(D, S) <=>
  and(forall(x in D: not(=(x, {}))),
    forall(x in D, y in D: or(=(x, y), =(**(x, y), {}))),
    =(++(D), S));

fun ~(D) = ++(set(d in D: true,
  set(x in d, y in d: true, tuple(x, y))));

fun S = join(0, join(1, join(2, join(3, join(4, {})))));
fun D0 = join(0, join(1, {}));
fun D1 = join(2, join(3, join(4, {})));
fun D = join(D0, join(D1, {}));

formula isPartition(D, S);
> true.
term ~(D);
> {<4, 4>, <4, 3>, <4, 2>, <3, 4>, <3, 3>, <3, 2>, <2, 4>, <2, 3>,
<2, 2>, <1, 1>, <1, 0>, <0, 1>, <0, 0>}.
formula isEquivalence(~(D), S);
> true.
```

```
term isPartition(/(S, ~(D)));
```

The partitioning of sets into equivalence classes is an important methodology with many applications some of which are demonstrated in the following subsections.

7.1.2 Modular Arithmetic

The domain of integer numbers introduced in Chapter 4 has infinite size, i.e., there is no upper bound for an integer. However, computer processors can only operate with numbers of a maximum size determined by the length of a computer word on the corresponding architecture, e.g., a 64 bit architecture only supports 2^{64} different numbers.

Apparently the question arises how to define the result of an operation whose value is outside the range that can be represented by such a word, e.g., the result of $2^{40} * 2^{30}$. While we may consider such an operation "illegal" (and raise a hardware interrupt), there is a more elegant solution provided by *modular arithmetic*: we define the result of every operation as the result of the unbounded integer operation *modulo* the number m of elements that can be represented by a computer word.

We start with a preliminary definition of modular arithmetic, which will later be replaced by a more suitable one.

Definition 104 (Modular Arithmetic, Direct Approach) Let $m \in \mathbb{Z}_{>0}$ and $\mathbb{Z}_m := \{x \in \mathbb{Z} : 0 \leq x < m\}$.

$$+_m : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}_m$$
$$x +_m y := (x + y) \bmod m$$

$$-_m : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}_m$$
$$x -_m y := (x - y) \bmod m$$

$$-_m : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}_m$$
$$-_m x := (-x) \bmod m$$

$$*_m : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}_m$$
$$x *_m y := (x * y) \bmod m$$

Example These tables describe arithmetic modulo 3 for a subset of $\mathbb{Z}$:

$+_3$	-4	-3	-2	-1	0	1	2	3	4
-4	1	2	0	1	2	0	1	2	0
-3	2	0	1	2	0	1	2	0	1
-2	0	1	2	0	1	2	0	1	2
-1	1	2	0	1	2	0	1	2	0
0	2	0	1	2	0	1	2	0	1
1	0	1	2	0	1	2	0	1	2
2	1	2	0	1	2	0	1	2	0
3	2	0	1	2	0	1	2	0	1
4	0	1	2	0	1	2	0	1	2

$*_3$	-4	-3	-2	-1	0	1	2	3	4
-4	1	0	2	1	0	2	1	0	2
-3	0	0	0	0	0	0	0	0	0
-2	2	0	1	2	0	1	2	0	1
-1	1	0	2	1	0	2	1	0	2
0	0	0	0	0	0	0	0	0	0
1	2	0	1	2	0	1	2	0	1
2	1	0	2	1	0	2	1	0	2
3	0	0	0	0	0	0	0	0	0
4	2	0	1	2	0	1	2	0	1

In above definition, the arguments may be arbitrary integers while the result is one of m values. However, a little investigation of the function tables reveals that the same pattern of result values is repeated every m lines and every m columns. Consequently, it does not matter whether we compute with an argument a or with $a+3$ or with $a-3$ or with $a+3i$ for any $i \in \mathbb{Z}$. We may therefore define the set of all values

$$[a]_m := \{a + im : i \in \mathbb{Z}\}$$

that cannot be distinguished from a by modular arithmetic modulo m. However, this definition is just a special case of the construction of equivalence classes, as we are going to elaborate now. This construction will provide us with a suitable domain for modular arithmetic.

First we define the relation that links two integers that "are the same" with respect to arithmetic modulo m.

Definition 105 (Modular Congruence) Two integers x and y are *congruent (kongruent)* modulo m if they have the same remainder when divided by m:

$$x \equiv_m y :\Leftrightarrow (x \bmod m) = (y \bmod m).$$

Clearly $\equiv_m$ is an equivalence relation, for every $m \in \mathbb{Z}_{>0}$. The equivalence classes induced by this relation collect all the integers that cannot be distinguished by arithmetic modulo m.

Definition 106 (Residue Class) The *residue class (Restklasse)* of a modulo m is the set of all integer numbers that are congruent to a modulo m.

$$[a]_m := [a]_{\equiv_m}.$$

It is easy to see that $[a]_m = \{a + im : i \in \mathbb{Z}\}$ as was our original intuition. However, the new construction has the advantage that we may construct the quotient set of $\mathbb{Z}$ with respect to $\equiv_m$, which gives a suitable domain for modular arithmetic.

Definition 107 (Modular Integer Numbers) The set of integers modulo m is the quotient set of $\mathbb{Z}$ with respect to congruence modulo m:

$$\mathbb{Z}_m := \mathbb{Z}/\equiv_m.$$

$\mathbb{Z}_m$ has m elements each of which is represented by a natural number less than m, i.e.,

$$\mathbb{Z}_m = \{[0]_m, [1]_m, [2]_m, \ldots, [m-1]_m\}.$$

Before we are going to define the arithmetic operations on this domain, we emphasize an important property of our construction.

Proposition 122 (Congruence Properties) Let a and a' be two integers that are in the same residue class and b and b' be two integers that are also in the same residue class. Then the result of any integer operation involving a and b is in the same residue class as if the operation were performed with a' and b' instead:

$$\begin{array}{l}
\forall m \in \mathbb{Z}_{>0}, a \in \mathbb{Z}, a' \in \mathbb{Z}, b \in \mathbb{Z}, b' \in \mathbb{Z}: \\
\quad [a]_m = [a']_m \wedge [b]_m = [b']_m \;\Rightarrow \\
\quad\quad [a+b]_m = [a'+b']_m \wedge \\
\quad\quad [a-b]_m = [a'-b']_m \wedge \\
\quad\quad [-a]_m = [-a']_m \wedge \\
\quad\quad [a*b]_m = [a'*b']_m.
\end{array}$$

Proof Take arbitrary $m \in \mathbb{Z}_{>0}$, $a \in \mathbb{Z}$, $a' \in \mathbb{Z}$, $b \in \mathbb{Z}$, $b' \in \mathbb{Z}$ such that $[a]_m = [a']_m$ and $[b]_m = [b']_m$. We show

$$[a+b]_m = [a'+b']_m.$$

We know that there exist quotients x, y, z, w and remainders r, s such that

$$\begin{array}{l}
a = xm + r, a' = ym + r, \\
b = zm + s, b' = wm + s,
\end{array}$$

which implies

$$\begin{array}{rcl}
[a+b]_m & = & \\
[(xm+r)+(ym+s)]_m & = & \\
[(x+y)m+(r+s)]_m & = & (*) \\
[(r+s)]_m & = & \\
[(z+w)m+(r+s)]_m & = & (*) \\
[(zm+r)+(wm+s)]_m & = & \\
[a'+b']_m. & &
\end{array}$$

(*) It remains to be shown that

$$\forall m \in \mathbb{N}_{>0}, x \in \mathbb{Z}, y \in \mathbb{Z}: [xm+y]_m = [y]_m.$$

Example We have $[7]_5 = [2]_5$ and $[9]_5 = [4]_5$. Consequently,

$$[7+9]_5 = [2+4]_5 = [6]_5 = [1]_5.$$

The importance of the congruence properties is that we may choose *any* representative of a value of $\mathbb{Z}_m$ in order to compute with that value, e.g., when dealing with

$$[3]_5 = \{\ldots, -7, -2, 3, 8, 13, \ldots\}$$

we need not take the "canonical" representative 3 but may also choose -2 or 8 as the representative for computation. Consequently, the following definition determines the function results uniquely.

Definition 108 (Modular Arithmetic, Residue Classes) Let $m \in \mathbb{Z}_{>0}$ and define the selector function

$$\overline{x} := \textbf{such } a \in \mathbb{Z} : x = [a]_m.$$

We then define the operations of *modular arithmetic (modulare Arithmetik)*:

$$\begin{array}{l} +_m : \mathbb{Z}_m \times \mathbb{Z}_m \to \mathbb{Z}_m \\ x +_m y := [\overline{x} +_{\mathbb{Z}} \overline{y}]_m \\ (\text{or} : [a]_m +_m [b]_m := [a +_{\mathbb{Z}} b]_m) \end{array}$$

$$\begin{array}{l} -_m : \mathbb{Z}_m \times \mathbb{Z}_m \to \mathbb{Z}_m \\ x -_m y := [\overline{x} -_{\mathbb{Z}} \overline{y}]_m \\ (\text{or} : [a]_m -_m [b]_m := [a -_{\mathbb{Z}} b]_m) \end{array}$$

$$\begin{array}{l} -_m : \mathbb{Z}_m \to \mathbb{Z}_m \\ -_m\ x := [-_{\mathbb{Z}}\ \overline{x}]_m \\ (\text{or} :\ -_m [a]_m := [-_{\mathbb{Z}}\ a]_m) \end{array}$$

$$\begin{array}{l} *_m : \mathbb{Z}_m \times \mathbb{Z}_m \to \mathbb{Z}_m \\ x *_m y := [\overline{x} *_{\mathbb{Z}} \overline{y}]_m \\ (\text{or} : [a]_m *_m [b]_m := [a *_{\mathbb{Z}} b]_m) \end{array}$$

For performing arithmetic on some $x \in \mathbb{Z}_m$,

1. we apply the selector function to determine a representative $\overline{x} \in \mathbb{Z}$,
2. perform the corresponding operation in $\mathbb{Z}$ to yield the result $r \in \mathbb{Z}$,

3. and then determine the residue class $[r]_m \in \mathbb{Z}_m$.

Example We consider arithmetic in $\mathbb{Z}_5 = \{[0]_5, [1]_5, [2]_5, [3]_5, [4]_5\}$.

$$\begin{aligned}
[17]_5 +_5 [24]_5 &= [2]_5 +_5 [4]_5 = [6]_5 = [1]_5 \\
[7]_5 -_5 [10]_5 &= [2]_5 -_5 [0]_5 = [3]_5 \\
[-7]_5 &= [3]_5 \\
[6]_5 *_5 [9]_5 &= [1]_5 *_5 [4]_5 = [4]_5 \\
[-3]_5 *_5 [6]_5 &= [2]_5 *_5 [1]_5 = [2]_5
\end{aligned}$$

Logic Evaluator We can implement modular arithmetic by equivalence classes as shown below, see Appendix C.11 (file `modular.txt`): the result of a modular operation is an equivalence class of integers $\langle x, y\rangle$ whose difference $x - y$ denotes the corresponding value.

```
option silent = true; option check = false;
read integer; read equiv;

// consider just subset of Z
fun Z' = Z[](I(4, 0));
pred Z+(x) <=> and(Z(x), =(sign(x), Z1));

// x = y mod m
fun =m(m: Z+) =
  set(x in Z', y in Z': =(modZ(x, m), modZ(y, m)), tuple(x, y));

// Z_m
fun Z(m: Z+) = /(Z', =m(m));

// selector function and equivalence class of x modulo m
fun _(x) = such(a in x: true, a);
fun []m(m: Z+, x: Z) = [](x, =m(m));

fun +m(m: Z+, x in Z(m), y in Z(m)) = []m(m, +Z(_(x), _(y)));
fun -m(m: Z+, x in Z(m), y in Z(m)) = []m(m, -Z(_(x), _(y)));
fun -m(m: Z+, x in Z(m))            = []m(m, -Z(_(x)));
fun *m(m: Z+, x in Z(m), y in Z(m)) = []m(m, *Z(_(x), _(y)));

fun M = I(3, 0);
term []m(M, Z2);
> {<0, 1>, <0, 4>, <2, 0>}.
term +m(M, []m(M, Z2), []m(M, Z2));
> {<0, 2>, <1, 0>, <4, 0>}.

term *m(M, []m(M, Z2), []m(M, Z2));
```

7.1.3 Another Construction of Number Domains

Integer Numbers We have introduced in Section 4.2 an integer as a pair $\langle x, y\rangle$ of natural numbers x and y such that their difference denotes the desired value. In order to determine this pair uniquely, either x or y has been chosen zero; we are now going to lift this artificial restriction by defining an integer as the *class* of all pairs with the same difference.

Definition 109 (Integer Numbers)

$$x \sim_{\mathbb{Z}} y :\Leftrightarrow (x_0 +_{\mathbb{N}} y_1 = y_0 +_{\mathbb{N}} x_1)$$

$$\mathbb{Z} := (\mathbb{N} \times \mathbb{N})/ \sim_{\mathbb{Z}}$$

$$0 := [\langle 0_{\mathbb{N}}, 0_{\mathbb{N}}\rangle];\ 1 := [\langle 1_{\mathbb{N}}, 0_{\mathbb{N}}\rangle];\ 2 := [\langle 2_{\mathbb{N}}, 0_{\mathbb{N}}\rangle]$$

$$\overline{x} := \textbf{such } a \in \mathbb{N} \times \mathbb{N} : x = [a]$$

$$\begin{array}{rcl}
x + y & := & [\langle \overline{x}_0 +_{\mathbb{N}} \overline{y}_0, \overline{x}_1 +_{\mathbb{N}} \overline{y}_1\rangle] \\
-x & := & [\langle \overline{x}_1, \overline{x}_0\rangle] \\
x - y & := & [\langle \overline{x}_0 +_{\mathbb{N}} \overline{y}_1, \overline{y}_0 +_{\mathbb{N}} \overline{x}_1\rangle] \\
x * y & := & [\langle (\overline{x}_0 *_{\mathbb{N}} \overline{y}_0) +_{\mathbb{N}} (\overline{x}_1 *_{\mathbb{N}} \overline{y}_1), (\overline{x}_0 *_{\mathbb{N}} \overline{y}_1) +_{\mathbb{N}} (\overline{x}_1 *_{\mathbb{N}} \overline{y}_0)\rangle] \\
x \leq y & :\Leftrightarrow & \overline{x}_0 + \overline{y}_1 \leq_{\mathbb{N}} \overline{y}_0 + \overline{x}_1
\end{array}$$

In above definition, $[x]$ denotes $[x]_{\sim_{\mathbb{Z}}}$. It is easy to see that $\sim_{\mathbb{Z}}$ is an equivalence relation, therefore $\mathbb{Z}$ is a partition of $\mathbb{N} \times \mathbb{N}$. The corresponding equivalence classes satisfy congruence properties similar to those of Proposition 122 such that the results of all operations are uniquely defined.

Example

$$\begin{array}{rcl}
5 & = & [\langle 7, 2\rangle] = \{\langle 5, 0\rangle, \langle 6, 1\rangle, \langle 7, 2\rangle, \langle 8, 3\rangle, \ldots\} \\
-6 & = & [\langle 3, 9\rangle] = \{\langle 0, 6\rangle, \langle 1, 7\rangle, \langle 2, 8\rangle, \langle 3, 9\rangle, \ldots\} \\
5 + (-6) & = & [\langle 7, 2\rangle] + [\langle 3, 9\rangle] = [\langle 10, 11\rangle] = [\langle 0, 1\rangle] = -1 \\
5 * (-6) & = & [\langle 7, 2\rangle] * [\langle 3, 9\rangle] = [\langle 39, 69\rangle] = [\langle 0, 30\rangle] = -30 \\
5 \leq -6 & \Leftrightarrow & [\langle 7, 2\rangle] \leq [\langle 3, 9\rangle] \Leftrightarrow 16 \leq 5 \Leftrightarrow \mathrm{F}.
\end{array}$$

The new construction is isomorphic to the old one such that the properties of all operations are preserved.

Proposition 124 (Isomorphism of Integer Constructions) Let ℤ' denote the old construction of the integers and ℤ denote the new one. The function

$$\begin{array}{l} i : \mathbb{Z}' \to \mathbb{Z} \\ i(x) := [x] \end{array}$$

is an isomorphism with respect to $0, +, -, *, <$, i.e., i is bijective and for all $x \in \mathbb{Z}'$ and $y \in \mathbb{Z}'$, we have:

$$\begin{array}{rcl} i(0_{\mathbb{Z}'}) & = & 0_{\mathbb{Z}}, \\ i(x +_{\mathbb{Z}'} y) & = & x +_{\mathbb{Z}} y, \\ i(-_{\mathbb{Z}'} x) & = & -_{\mathbb{Z}} x, \\ i(x -_{\mathbb{Z}'} y) & = & x -_{\mathbb{Z}} y, \\ i(x *_{\mathbb{Z}'} y) & = & x *_{\mathbb{Z}} y, \\ x \leq_{\mathbb{Z}'} y & \Leftrightarrow & i(x) \leq_{\mathbb{Z}} i(y). \end{array}$$

It is easy to see that the inverse isomorphism is denoted by

$$\begin{array}{l} j : \mathbb{Z} \to \mathbb{Z}' \\ j(x) := \mathrm{I}(\overline{x}) \end{array}$$

where "I" denotes the constructor function on $\mathbb{Z}'$.

Proof Take arbitrary $x \in \mathbb{Z}'$ and $y \in \mathbb{Z}'$. We prove

$$i(x +_{\mathbb{Z}'} y) = x +_{\mathbb{Z}} y.$$

We know

$$i(x +_{\mathbb{Z}'} y) \quad = \quad i(\mathrm{I}(x_0 +_{\mathbb{N}} y_0, x_1 +_{\mathbb{N}} y_1)).$$

Case $x_0 +_{\mathbb{N}} y_0 \geq x_1 +_{\mathbb{N}} y_1$**:** We have

$$\begin{array}{rcl} i(\mathrm{I}(x_0 +_{\mathbb{N}} y_0, x_1 +_{\mathbb{N}} y_1)) & = & \\ i(\langle (x_0 +_{\mathbb{N}} y_0) - (x_1 +_{\mathbb{N}} y_1) \rangle) & = & \\ [\langle (x_0 +_{\mathbb{N}} y_0) - (x_1 +_{\mathbb{N}} y_1) \rangle] & = & \text{(definition } \sim_{\mathbb{Z}}) \\ [\langle x_0 +_{\mathbb{N}} y_0, x_1 +_{\mathbb{N}} y_1 \rangle] & = & \\ x +_{\mathbb{Z}} y. & & \end{array}$$

Case $x_0 +_{\mathbb{N}} y_0 < x_1 +_{\mathbb{N}} y_1$**:** Proceeds analogously.

An implementation of this construction in the Logic Evaluator is shown below, see Appendix C.12 (file `integer2.txt`).

```
option silent = true; option check = false;
read equiv; read natural0;
fun Z' = set(x in nat(0, 6), y in nat(0, 6): true, tuple(x, y));

fun ~Z = set(x in Z', y in Z':
  =(+N(.0(x), .1(y)), +N(.0(y), .1(x))), tuple(x, y));
fun Z = /(Z', ~Z);

fun _(x) = such(a in x: true, a);
fun []Z(x in Z') = [](x, ~Z);

fun Z0 = []Z(tuple(N0, N0));
fun Z1 = []Z(tuple(N1, N0));
fun Z2 = []Z(tuple(N2, N0));

fun +Z(x in Z, y in Z) =
  []Z(tuple(+N(.0(_(x)), .0(_(y))), +N(.1(_(x)), .1(_(y)))));
fun -Z(x in Z) = []Z(tuple(.1(_(x)), .0(_(x))));
fun -Z(x in Z, y in Z) =
  []Z(tuple(+N(.0(_(x)), .1(_(y))), +N(.0(_(y)), .1(_(x)))));
fun *Z(x in Z, y in Z) =
  []Z(tuple(+N(*N(.0(_(x)), .0(_(y))), *N(.1(_(x)), .1(_(y)))),
            +N(*N(.0(_(x)), .1(_(y))), *N(.1(_(x)), .0(_(y))))));
pred <=Z(x in Z, y in Z) <=>
  <=N(+N(.0(_(x)), .1(_(y))), +N(.0(_(y)), .1(_(x))));

term +Z(Z1, Z2);
> {<3, 0>, <4, 1>, <5, 2>, <6, 3>}.
term -Z(Z1, Z2);
> {<0, 1>, <1, 2>, <2, 3>, <3, 4>, <4, 5>, <5, 6>}.
term *Z(Z2, Z2);
> {<4, 0>, <5, 1>, <6, 2>}.
```

```
formula <=Z(-Z(Z2), Z1);
```

Rational Numbers In Section 4.3, we have defined a rational as a pair $\langle x, y\rangle$ of integer numbers x and y such that their quotient denotes the desired value. In order to determine this pair uniquely, x and y have been chosen relatively prime with y being positive. We will now lift this restriction by defining a rational as the class of all pairs with the same quotient.

Definition 110 (Rational Numbers)

$$x \sim_{\mathbb{Q}} y :\Leftrightarrow (x_0 *_{\mathbb{Z}} y_1 = y_0 *_{\mathbb{Z}} x_1)$$

$$\mathbb{Q} := (\mathbb{Z} \times \mathbb{Z}_{\neq 0}) / \sim_{\mathbb{Q}}$$

$$0 := [\langle 0_{\mathbb{Z}}, 1_{\mathbb{Z}} \rangle];\ 1 := [\langle 1_{\mathbb{Z}}, 1_{\mathbb{Z}} \rangle];\ 2 := [\langle 2_{\mathbb{Z}}, 1_{\mathbb{Z}} \rangle];$$

$$\overline{x} := \textbf{such } a \in \mathbb{Z} \times \mathbb{Z} : x = [a]$$

$$\begin{array}{rcl}
x + y & := & [\langle (\overline{x}_0 *_{\mathbb{Z}} \overline{y}_1) +_{\mathbb{Z}} (\overline{y}_0 *_{\mathbb{Z}} \overline{x}_1), \overline{x}_1 *_{\mathbb{Z}} \overline{y}_1 \rangle] \\
-x & := & [\langle -_{\mathbb{Z}}\, \overline{x}_0, \overline{x}_1 \rangle] \\
x - y & := & [\langle (\overline{x}_0 *_{\mathbb{Z}} \overline{y}_1) -_{\mathbb{Z}} (\overline{y}_0 *_{\mathbb{Z}} \overline{x}_1), \overline{x}_1 *_{\mathbb{Z}} \overline{y}_1 \rangle] \\
x * y & := & [\langle \overline{x}_0 *_{\mathbb{Z}} \overline{y}_0, \overline{x}_1 *_{\mathbb{Z}} \overline{y}_1 \rangle] \\
x^{-1} & := & [\langle \overline{x}_1, \overline{x}_0 \rangle] \\
x / y & := & [\langle \overline{x}_0 *_{\mathbb{Z}} \overline{y}_1, \overline{y}_0 *_{\mathbb{Z}} \overline{x}_1 \rangle] \\
x \leq y & :\Leftrightarrow & \overline{x}_0 *_{\mathbb{Z}} \overline{y}_1 \leq_{\mathbb{Z}} \overline{y}_0 *_{\mathbb{Z}} \overline{x}_1
\end{array}$$

It is easy to see that $\sim_{\mathbb{Q}}$ is an equivalence relation on $\mathbb{Z} \times \mathbb{Z}_{\neq 0}$ where $\mathbb{Z}_{\neq 0} := \{x \in \mathbb{Z} : x \neq 0\}$; we have a corresponding partition that also satisfies the necessary congruence properties such that the results of all operations are uniquely defined.

Example

$$\begin{array}{rcl}
\frac{4}{6} & = & [\langle 4, 6 \rangle] = \{\langle 2, 3 \rangle, \langle -2, -3 \rangle, \langle 4, 6 \rangle, \langle -4, -6 \rangle, \langle 6, 9 \rangle, \ldots\} \\
-\frac{2}{5} & = & -[\langle 2, 5 \rangle] = [\langle -2, 5 \rangle] \\
 & = & \{\langle 2, -5 \rangle, \langle -2, 5 \rangle, \langle 4, -10 \rangle, \langle -4, 10 \rangle, \langle 6, -15 \rangle, \ldots\} \\
\frac{4}{6} + (-\frac{2}{5}) & = & [\langle 4, 6 \rangle] + [\langle -2, 5 \rangle] = [\langle 8, 30 \rangle] = [\langle 4, 15 \rangle] = \frac{4}{15} \\
\frac{4}{6} * (-\frac{2}{5}) & = & [\langle 4, 6 \rangle] * [\langle -2, 5 \rangle] = [\langle -8, 30 \rangle] = [\langle -4, 15 \rangle] = -\frac{4}{15}
\end{array}$$

Again the new construction is isomorphic to the old one such that the properties of all operations are preserved.

Proposition 126 (Isomorphism of Rational Constructions) Let $\mathbb{Q}$' denote the old construction of the integers and $\mathbb{Q}$ denote the new one. The function

$$\begin{array}{l} i : \mathbb{Q}' \to \mathbb{Q} \\ i(x) := [x] \end{array}$$

is an isomorphism with respect to $0, +, -, *, ', <$, i.e., i is bijective and for all $x \in \mathbb{Q}'$ and $y \in \mathbb{Q}'$, we have:

$$\begin{array}{rcl} i(0_{\mathbb{Q}'}) & = & 0_{\mathbb{Q}}, \\ i(x +_{\mathbb{Q}'} y) & = & x +_{\mathbb{Q}} y, \\ i(-_{\mathbb{Q}'} x) & = & -_{\mathbb{Q}} x, \\ i(x -_{\mathbb{Q}'} y) & = & x -_{\mathbb{Q}} y, \\ i(x *_{\mathbb{Q}'} y) & = & x *_{\mathbb{Q}} y, \\ i(x^{-1}) & = & x^{-1}, \\ i(x /_{\mathbb{Q}'} y) & = & x /_{\mathbb{Q}} y, \\ x \leq_{\mathbb{Q}'} y & \Leftrightarrow & i(x) \leq_{\mathbb{Q}} i(y). \end{array}$$

It is easy to see that the inverse isomorphism is denoted by

$$\begin{array}{l} j : \mathbb{Q} \to \mathbb{Q}' \\ j(x) := \frac{\overline{x}_0}{\overline{x}_1} \end{array}$$

where $\frac{*}{*}$ denotes the constructor function on $\mathbb{Q}'$.

An implementation of this construction in the Logic Evaluator is shown on the next page, see Appendix C.13 (file `rational2.txt`).

Real Numbers While in Section 4.4 the real numbers have been only axiomatically characterized, also a direct construction is possible. The square of no rational number yields 2, but can define an infinite *sequence* of rationals that approaches such a number arbitrarily closely. The basic idea is to partition the set of all "well-behaved" infinite sequences of rationals by an equivalence relation that is true for any two sequences that approach each other arbitrarily closely. Each such equivalence class denotes a real number with arithmetic being defined point-wise on the sequence.

For this construction, we first define the class of "well-behaved" sequences.

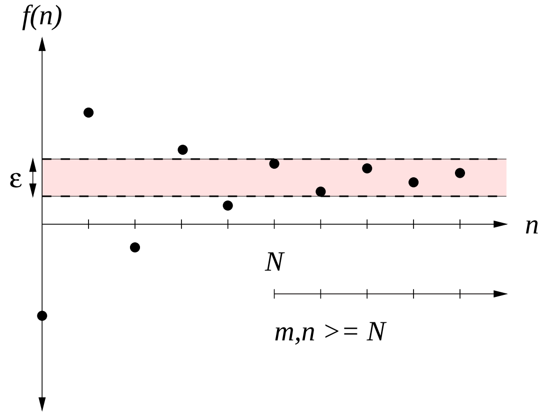
```
option silent = true; option check = false;
read equiv; read integer;
fun Q' = set(x in Z[](I(3, 0)), y in Z[](I(3, 0)):
            not(=(y, Z0)), tuple(x, y));

fun ~Q = set(x in Q', y in Q':
  =(*Z(.0(x), .1(y)), *Z(.0(y), .1(x))), tuple(x, y));
fun Q = /(Q', ~Q);

fun _(x) = such(a in x: true, a);
fun []Q(x in Q') = [](x, ~Q);

fun Q0 = []Q(tuple(Z0, Z1));
fun Q1 = []Q(tuple(Z1, Z1));
fun Q2 = []Q(tuple(Z2, Z1));

fun +Q(x in Q, y in Q) =
  []Q(tuple(+Z(*Z(.0(_(x)), .1(_(y))), *Z(.0(_(y)), .1(_(x)))),
            *Z(.1(_(x)), .1(_(y)))));
fun -Q(x in Q) = []Q(tuple(-Z(.0(_(x))), .1(_(x))));
fun -Q(x in Q, y in Q) =
  []Q(tuple(-Z(*Z(.0(_(x)), .1(_(y))), *Z(.0(_(y)), .1(_(x)))),
            *Z(.1(_(x)), .1(_(y)))));
fun *Q(x in Q, y in Q) =
  []Q(tuple(*Z(.0(_(x)), .0(_(y))), *Z(.1(_(x)), .1(_(y)))));
fun ^-1Q(x in Q) = []Q(tuple(.1(_(x)), .0(_(x))));
fun /Q(x in Q, y in Q) =
  []Q(tuple(*Z(.0(_(x)), .1(_(y))), *Z(.0(_(y)), .1(_(x)))));
pred <=Q(x in Q, y in Q) <=>
  <=Z(*Z(.0(_(x)), .1(_(y))), *Z(.0(_(y)), .1(_(x))));

term +Q(Q1, /Q(Q1, Q2));
> {<<0, 3>, <0, 2>>, <<3, 0>, <2, 0>>}.
```

```
formula <=(/Q(Q1, Q2), Q1);
```

Definition 111 (Cauchy-Sequence) An infinite sequence over the reals is a *Cauchy-sequence (Cauchy-Folge)*, if its members approach each other arbitrarily closely:

$$\begin{aligned}
&f \text{ is Cauchy-sequence} :\Leftrightarrow \\
&\quad f : \mathbb{N} \to \mathbb{Q} \wedge \\
&\quad \forall \epsilon \in \mathbb{Q}_{>0} : \exists N \in \mathbb{N} : \forall m \geq N, n \geq N : |f_m - f_n| < \epsilon.
\end{aligned}$$

In other words, for every ϵ, from a certain position N on, all members of a Cauchy-sequence stay in a “tunnel” of width ϵ:



The position of this tunnel determines a real number; two sequences define the same number if the difference of their corresponding members becomes arbitrarily small. We define this equivalence relation and the arithmetic operations as shown below.

Definition 112 (Real Numbers)

$$f \sim_{\mathbb{R}} g :\Leftrightarrow \forall \epsilon \in \mathbb{Q}_{>0} : \exists N \in \mathbb{N} : \forall n \geq N : |f_n - g_n| < \epsilon.$$

$$C := \{f : \mathbb{N} \to \mathbb{Q} : f \text{ is Cauchy-sequence}\}$$
$$\mathbb{R} := C/\sim_{\mathbb{R}}$$

$$\begin{aligned} 0 &:= [\textbf{such } f \in C : \forall n \in \mathbb{N} : f_n = 0_{\mathbb{Q}}] \\ 1 &:= [\textbf{such } f \in C : \forall n \in \mathbb{N} : f_n = 1_{\mathbb{Q}}] \\ 2 &:= [\textbf{such } f \in C : \forall n \in \mathbb{N} : f_n = 2_{\mathbb{Q}}] \end{aligned}$$

$$\overline{x} := \textbf{such } f \in C : x = [f]$$

$$\begin{aligned} x + y &:= [[\overline{x}_n +_{\mathbb{Q}} \overline{y}_n]_n] \\ -x &:= [[-_{\mathbb{Q}} \overline{x}_n]_n] \\ x - y &:= [[\overline{x}_n -_{\mathbb{Q}} \overline{y}_n]_n] \\ x * y &:= [[\overline{x}_n *_{\mathbb{Q}} \overline{y}_n]_n] \\ x^{-1} &:= [[\overline{x}_n^{-1}]_n] \\ x/y &:= [[\overline{x}_n /_{\mathbb{Q}} \overline{y}_n]_n] \end{aligned}$$

$$\begin{aligned} x \text{ is positive} &:\Leftrightarrow \exists \epsilon \in \mathbb{Q}_{>0}, N \in \mathbb{N} : \forall n \geq N : \overline{x} > \epsilon \\ x \leq y &:\Leftrightarrow \exists z \in \mathbb{R} : z \text{ is positive} \wedge x + z = y \end{aligned}$$

We use the notation $[T]_n$ for the sequence f defined as $f(n) := T$ (not to confuse with the notation $[x]$ for the equivalence class of x). One can show that $\sim_{\mathbb{R}}$ is an equivalence relation that satisfies the necessary congruence properties to determine unique result values for the functions defined above. Furthermore, one can show that the domain $\mathbb{R}$ defined in such a way satisfies the axioms stated in Section 4.4, i.e., we have indeed the real numbers.

Complex Numbers Also $\mathbb{C}$ can be defined along the lines demonstrated above. The intuition is to partition the set of all univariate polynomials (considered as equations with righthand side zero) by the equivalence relation that is true for two polynomials if their difference is divisible by $x^2 + 1$ (the polynomial that defines the imaginary unit). We omit the details.

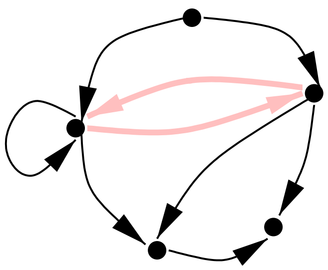
7.2 Order Relations

Partial Orders A binary relation is a partial order if it has some (not necessarily all) properties of the relation $\leq$ on $\mathbb{R}$. In addition to some of the properties introduced in the previous section, we need the following notion.

Definition 113 (Antisymmetric) A binary relation is *antisymmetric (antisymmetrisch)*, if different elements are not mutually related:

$$
\begin{aligned}
&R \text{ is antisymmetric on } S :\Leftrightarrow \\
&\quad \forall x \in S, y \in S : (\langle x, y\rangle \in R \wedge \langle y, x\rangle \in R) \Rightarrow x = y.
\end{aligned}
$$

In the graph representation of an antisymmetric relation, the only loops are from a node to itself, i.e., the following relation is *not* antisymmetric:



Definition 114 (Partial Order) A binary relation on S is a *partial order (partielle Ordnung, Halbordnung)*, if it is reflexive, antisymmetric, and transitive:

$$\begin{aligned}&R \text{ is partial order on } S :\Leftrightarrow\\ &\quad R \subseteq S \times S \wedge\\ &\quad R \text{ is reflexive on } S \wedge\\ &\quad R \text{ is antisymmetric on } S \wedge\\ &\quad R \text{ is transitive on } S.\end{aligned}$$

Example

- $\leq$ is a partial order on $\mathbb{N}$.
- $\subseteq$ is a partial order on $\mathbb{P}(S)$, for every set S.
- $\mid$ (divides) is a partial order on $\mathbb{N}$.

Quasi Orders Frequently, a partial order is denoted as infix $\preceq$ (or simply as $\leq$); a corresponding relation $\prec$ is then defined as

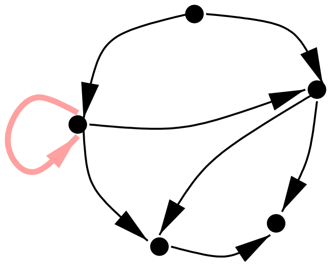
$$x \prec y :\Leftrightarrow x \preceq y \wedge x \neq y.$$

While $\prec$ is also antisymmetric and transitive, it is not reflexive, and thus not a partial order. Actually, it even satisfies the following property.

Definition 115 (Irreflexive) A binary relation is *irreflexive (irreflexiv)*, if no element is related to itself:

$$\begin{aligned}&R \text{ is irreflexive on } S :\Leftrightarrow\\ &\quad \forall x \in S : \langle x, x\rangle \notin R.\end{aligned}$$

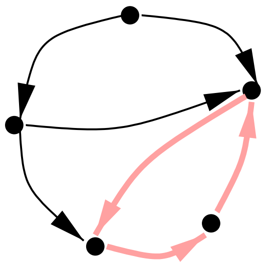
The graph representation of an irreflexive relation does not contain direct loops from a node to itself, i.e., the following relation is *not* irreflexive:



Definition 116 (Quasi Order) A binary relation on S is a *quasi order (Quasi-Ordnung)*, if it is irreflexive, antisymmetric, and transitive:

$$\begin{aligned}&R \text{ is quasi order on } S :\Leftrightarrow\\ &\quad R \subseteq S \times S \land\\ &\quad R \text{ is irreflexive on } S \land\\ &\quad R \text{ is antisymmetric on } S \land\\ &\quad R \text{ is transitive on } S.\end{aligned}$$

The graph representation of an quasi order does not contain any (direct or indirect) loops, i.e., the following is *not* a quasi-order:



Proposition 127 (Quasi Orders Are Antisymmetric) Every quasi order is antisymmetric:

$$\begin{aligned}&\forall S, \prec : \prec \text{ is quasi order on } S \;\Rightarrow\\ &\quad \prec \text{ is antisymmetric on } S.\end{aligned}$$

Proof Take arbitrary S and quasi order $\prec$ on S. Assume there exist $x \in S$ and $y \in S$ with $x \neq y$ such that $x \prec y$ and $y \prec x$. By transitivity, we then have $x \prec x$ which contradicts the irreflexivity of $\prec$.

Consequently, the only difference between partial orders and quasi orders is reflexivity versus irreflexivity. Furthermore, the following holds.

Proposition 129 (Quasi Order from Partial Order) If $\preceq$ is a partial order, then $\prec$ is a quasi-order:

$$\begin{aligned}&\forall S, \preceq : \preceq \text{ is partial order on } S \Rightarrow\\&\quad \prec \text{ is quasi order on } S.\end{aligned}$$

Therefore adding equality to a quasi order makes it a partial order and removing equality from a partial order makes it a quasi order.

Example

- $<$ is a quasi-order on $\mathbb{R}$.
- $\subset$ (proper subset) is a quasi-order on $\mathbb{P}(S)$, for every set S.

In a partial order or quasi order, not all elements are necessarily comparable, e.g., in $\mathbb{P}(\{0,1,2\})$ neither $\{0,1\} \subseteq \{1,2\}$ nor $\{1,2\} \subseteq \{0,1\}$.

Definition 117 (Incomparable) Two elements x and y are incomparable with respect to a relation $\preceq$ if neither $x \preceq y$ nor $y \preceq x$:

$$\begin{aligned}&x \text{ and } y \text{ are incomparable w.r.t. } \preceq :\Leftrightarrow\\&\quad x \not\preceq y \wedge y \not\preceq x.\end{aligned}$$

Total Orders Some partial orders do not have incomparable elements.

Definition 118 (Total Order) A partial order is a *total order (totale Ordnung)* or *linear order (lineare Ordnung)* or *chain (Kette)* if no elements are incomparable with respect to this order:

$$\begin{array}{l} R \text{ is total order on } S \;:\Leftrightarrow \\ \quad R \text{ is partial order on } S \land \\ \quad \neg\exists x \in S, y \in S : x \text{ and } y \text{ are incomparable w.r.t. } R. \end{array}$$

Example

- $\leq$ is a total order on $\mathbb{R}$.
- The *lexicographic order (lexikographische Ordnung)*

  $$x \preceq y :\Leftrightarrow x_0 < y_0 \lor (x_0 = y_0 \land x_1 \leq y_1)$$

  is a total order on $\mathbb{N} \times \mathbb{N}$. In this order, we have

  $$\langle 0,0\rangle \prec \langle 0,1\rangle \prec \langle 0,2\rangle \prec \ldots \prec \langle 1,0\rangle \prec \langle 1,1\rangle \prec \langle 1,2\rangle \prec \ldots$$

  i.e., tuples are ordered first by their first component and then by their second component.
- Let $\preceq$ be a total order on an alphabet A, let $W_n := \mathbb{N}_n \to A$ be the set of all words with n letters, and let

  $$\overline{w} := \textbf{such } u \in W_{n-1} : \forall i \in \mathbb{N}_{n-1} : u_i = w_{i+1}$$

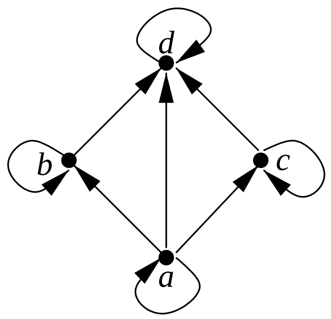
  be the function that removes the first letter from word w. The lexicographic order $\preceq_n \subseteq W_n \times W_n$ defined as

  $$w \preceq_n u :\Leftrightarrow n = 0 \lor w_0 \prec u_0 \lor (w_0 = u_0 \land \overline{w} \leq_{n-1} \overline{u})$$

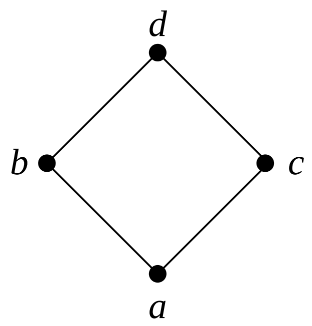
  is a total order. If A is the set of Roman letters, we have

  $$\text{“back”} \prec_4 \text{“bare”} \prec_4 \text{“base”} \prec_4 \text{“bear”} \prec_4 \text{“bend”} \prec_4 \text{“care”}$$

Hasse Diagrams Like any binary relation, a partial order can be represented by a directed graph, e.g.



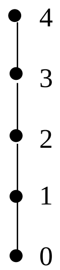
A more economical visualization that utilizes the properties of partial order is provided by a *Hasse diagram (Hasse Diagramm)*. Such a diagram is an undirected graph were all edges are considered as arrows from bottom to top, i.e., smaller elements are placed lower:



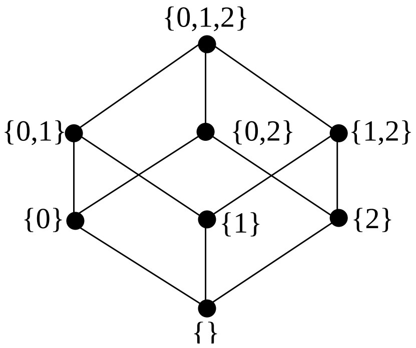
The graph for the corresponding partial order is constructed by directing the edges from bottom to top, by adding loops to all nodes (because of reflexivity) and by adding direct arrows between all nodes that are indirectly connected by two or more arrows (because of transitivity). E.g., the relationship $a \preceq d$ is not denoted by an extra edge but can be deduced from the relationship $a \preceq b$ and $b \preceq d$ in the diagram.

Example

- This Hasse diagram illustrates the total order $\leq$ on $\mathbb{N}_5$.

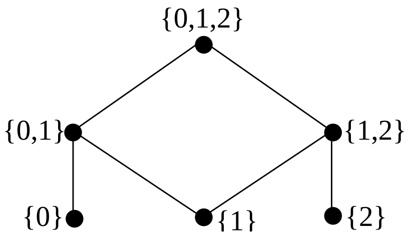


- This Hasse diagram illustrates the partial order $\subseteq$ on $\mathbb{P}(\{1, 2, 3\})$:

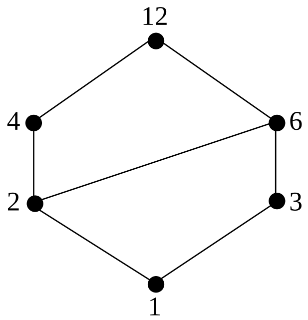


- This Hasse diagram illustrates the partial order $\subseteq$ on

$$\{\{0\}, \{1\}, \{2\}, \{0, 1\}, \{1, 2\}, \{0, 1, 2\}\}:$$



- Let $F_n := \{m \in \mathbb{N} : m|n\}$ (the factors of n). The following Hasse diagram illustrates the partial order $|$ on F_{12}:



Logic Evaluator Above definitions can be implemented as shown below (see Appendix C.14 (file `order.txt`)):

```
option silent = true; read equiv;

pred isAntiSymmetric(R: Relation, S) <=>
  forall(x in S, y in S:
    implies(and(in(tuple(x, y), S), in(tuple(y, x), S)), =(x, y)));
pred isPartialOrder(R: Relation, S) <=>
  and(isRelation(R, S, S), isReflexive(R, S),
      isAntiSymmetric(R, S), isTransitive(R, S));
pred isIrreflexive(R: Relation, S) <=>
  forall(x in S: not(in(tuple(x, x), R)));
pred isQuasiOrder(R: Relation, S) <=>
  and(isRelation(R, S, S), isIrreflexive(R, S),
      isAntiSymmetric(R, S), isTransitive(R, S));
pred areIncomparable(x, y, R) <=>
  and(not(in(tuple(x, y), R)), not(in(tuple(y, x), R)));
pred isTotalOrder(R: Relation, S) <=>
  and(isRelation(R, S, S),
      not(exists(x in S, y in S: areIncomparable(x, y, R))));

fun S(n) = set(x in nat(0, n): true, x);fun j(x, y, S) =
join(tuple(x, y), S);
fun R0 = j(0, 1, j(0, 2, j(2, 3, j(0, 3, {}))));
fun R1 = j(0, 0, j(1, 1, j(2, 2, j(3, 3, R0))));
fun R2 = j(1, 2, j(1, 3, R1));

formula isQuasiOrder(R0, S(3));
> true.
formula isPartialOrder(R1, S(3));
> true.

formula isTotalOrder(R2, S(3));
```

More Notions Various notions that we have introduced for the total ordering $\leq$ on $\mathbb{R}$ can be generalized to partial orders.

Definition 119 (Least and Greatest Element) If x is an element of S such that x is less than or equal to any element of S, then x is called the *least element (kleinstes Element)* of S:

$$\begin{aligned}&x \text{ is least element of } S \text{ w.r.t. } \preceq \;:\Leftrightarrow\\&\quad x \in S \wedge \forall y \in S : x \preceq y.\end{aligned}$$

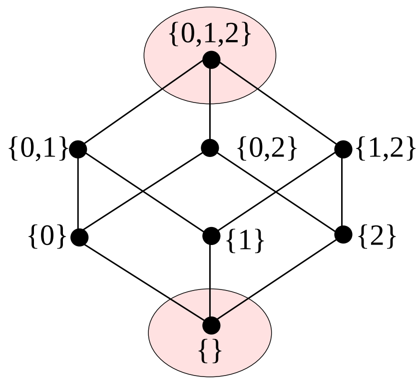
Correspondingly, if x is an element of S such that x is greater than or equal to any element of S with respect to a partial ordering, then x is called the *greatest element (größtes Element)* of S:

$$x \text{ is greatest element of } S \text{ w.r.t. } \preceq \;:\Leftrightarrow$$
$$x \in S \wedge \forall y \in S : y \preceq x.$$

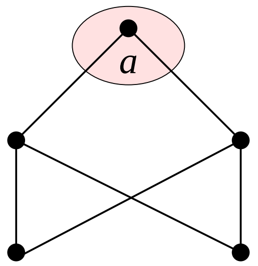
We can show that a least respectively greatest element is unique, if such an element exists.

Example

- $\emptyset$ is the least element of $\mathbb{P}(\{1,2,3\})$ with respect to $\subseteq$, $\{1,2,3\}$ is the greatest element:



- The following Hasse diagram denotes a partial order with greatest element a but without a least element:



Definition 120 (Minimal and Maximal Element) If x is an element of S such that no element of S is smaller than S, then x is called a *minimal element (minimales Element)* of S.

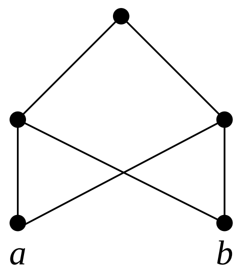
$$\begin{aligned}&x \text{ is minimal element of } S \text{ w.r.t. } \preceq \;:\Leftrightarrow\\&\quad x \in S \land \forall y \in S : y \preceq x \Rightarrow y = x.\end{aligned}$$

Correspondingly, if x is an element of S such that no element of S is greater than S, then x is called a *maximal element (maximales Element)* of S.

$$\begin{aligned}&x \text{ is maximal element of } S \text{ w.r.t. } \preceq \;:\Leftrightarrow\\&\quad x \in S \land \forall y \in S : x \preceq y \Rightarrow x = y.\end{aligned}$$

Minimal and maximal elements are not necessarily unique: distinct incomparable minimal respectively maximal elements may exist.

Example The following Hasse diagram denotes a partial order with two minimal elements a and b:



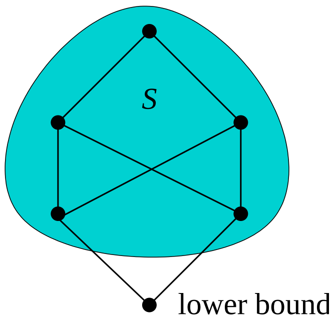
Definition 121 (Upper and Lower Bound) A value x is a *lower bound (untere Schranke)* of S, if it is less than or equal any element of S:

$$\begin{aligned}&x \text{ is lower bound of } S \text{ w.r.t. } \preceq \;:\Leftrightarrow\\&\quad \forall y \in S : x \preceq y.\end{aligned}$$

Correspondingly, a value x is an *upper bound (obere Schranke)* of S, if it is greater than or equal any element of S:

$$x \text{ is upper bound of } S \text{ w.r.t. } \preceq \;:\Leftrightarrow$$
$$\forall y \in S : y \preceq x.$$

Please note that an lower (respectively upper bound) of a set S need not be an element of S:



Definition 122 (Infimum and Supremum) A value x is a *infimum (Infimum)* of S, if it is the greatest lower bound of S:

$$x \text{ is infimum of } S \text{ w.r.t. } \preceq \;:\Leftrightarrow$$
$$x \text{ is lower bound of } S \text{ w.r.t. } \preceq \;\wedge$$
$$\forall y : y \text{ is lower bound of } S \text{ w.r.t. } \preceq \;\Rightarrow y \preceq x.$$

Correspondingly, a value x is a *supremum (Supremum)* of S, if it is the least upper bound of S:

$$x \text{ is supremum of } S \text{ w.r.t. } \preceq \;:\Leftrightarrow$$
$$x \text{ is upper bound of } S \text{ w.r.t. } \preceq \;\wedge$$
$$\forall y : y \text{ is upper bound of } S \text{ w.r.t. } \preceq \;\Rightarrow x \preceq y.$$

Even the supremum respectively infimum of a set need not be an element of this set. However, we have the following relationships between the concepts defined above.

Proposition 130 (Order Laws) Let $\preceq$ be a partial order on S. Then for every x, the following holds:

$$
\begin{aligned}
&x \text{ is least (greatest) element of } S \text{ w.r.t. } \preceq \Rightarrow\\
&\quad x \text{ is minimal (maximal) element of } S \text{ w.r.t. } \preceq,\\
&x \text{ is least (greatest) element of } S \text{ w.r.t. } \preceq \Rightarrow\\
&\quad x \text{ is infimum (supremum) of } S \text{ w.r.t. } \preceq,\\
&x \text{ is lower (upper) bound of } S \text{ w.r.t. } \preceq \wedge\, x \in S \;\Rightarrow\\
&\quad x \text{ is least (greatest) element of } S \text{ w.r.t. } \preceq .
\end{aligned}
$$

Logic Evaluator Some of above definitions can be implemented as shown below, see Appendix C.14 (file `order.txt`).

```
option silent = true; read order;

pred isLeast(x, S, R) <=>
  and(in(x, S), forall(y in S: in(tuple(x, y), R)));
pred isMinimal(x, S, R) <=>
  and(in(x, S), forall(y in S:
    implies(in(tuple(y, x), R), =(y, x))));
pred isLowerBound(x, S, R) <=>
  forall(y in S: in(tuple(x, y), R));
pred isInfimum(x, S, R) <=>
  and(isLowerBound(x, S, R), forall(y in domain(R):
    implies(isLowerBound(y, S, R), in(tuple(y, x), R))));

fun S0 = join(2, join(3, {}));
fun S1 = join(0, join(1, S0));

fun j(x, y, S) = join(tuple(x, y), S);
fun R0 = j(0, 0, j(1, 1, j(2, 2, j(3, 3, {}))));
fun R1 = j(0, 2, j(0, 3, j(2, 3, R0)));
fun R  = j(1, 2, j(1, 3, j(2, 3, R1)));

formula isLeast(2, S0, R);
> true.
formula isMinimal(0, S1, R);
> true.
formula isLeast(0, S1, R);
> false.
formula isLowerBound(0, S0, R);
> true.
```

```
formula isInfimum(0, S1, R);
```

7.3 Graphs

Basic Notions Throughout this document, we have used graphs as visual illustrations of binary relations. However, we can also consider graphs themselves as mathematical objects that are represented by binary relations. The theory of graphs has many applications in various scientific areas; they can be used to model railroad plans as well as electrical circuits, molecules as well as brain structures.

Definition 123 (Directed Graph) A *directed graph (gerichteter Graph)* is a pair $\langle V, E\rangle$ of a set V of *vertices (Knoten)* or *nodes* and a set of E of *edges (Kanten)* or *arcs* where E is a binary relation on S:

$$
\begin{aligned}
&G \text{ is directed graph} :\Leftrightarrow \\
&\quad \exists V, E : \\
&\qquad G = \langle V, E\rangle \wedge \\
&\qquad E \subseteq V \times V.
\end{aligned}
$$

An element $\langle x, y\rangle \in E$ expresses the fact that node x is connected to node y where x is called the *initial node* of the corresponding edge and y is called its *terminal node*. Both nodes are then *adjazent*.

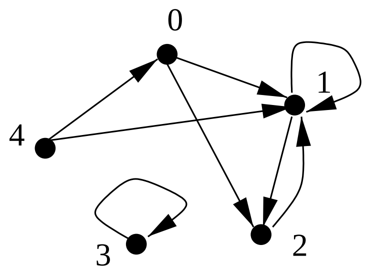
Please note that in above definition of a graph there may be at most one edge between any pair of nodes. A graph with potentially more than one edge between two nodes is called a *multigraph (Multigraph)* which may be used to describes e.g. multiple roads between two cities. Furthermore, the edges in a graph may be labelled with real values; such a graph is called a *weighted graph (gewichteter Graph)* and may describe lengths of roads or capacities of water pipes. However, we will not consider the formalization of such graphs in this document.

Example

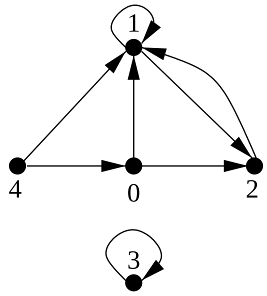
- The graph $\langle \mathbb{N}_5, E\rangle$ with

  $$E = \{\langle 0,1\rangle, \langle 0,2\rangle, \langle 1,1\rangle, \langle 1,2\rangle, \langle 2,1\rangle, \langle 3,3\rangle, \langle 4,0\rangle, \langle 4,1\rangle\}$$

  can be depicted as



or as



i.e., the visual representation of a graph is not unique; vice versa, different visual objects may describe the same graph.

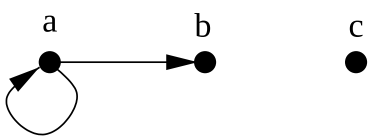
A representation of this graph by a boolean matrix (every edge is denoted by a matrix entry true) is the following:

	0	1	2	3	4
0	false	true	true	false	false
1	false	true	true	false	false
2	false	true	false	false	false
3	false	false	false	true	false
4	true	true	false	false	false

- The graph $\langle\{\mathsf{a},\mathsf{b},\mathsf{c}\}, E\rangle$ with

  $$E = \{\langle \mathsf{a},\mathsf{a}\rangle, \langle \mathsf{a},\mathsf{b}\rangle\}$$

  can be depicted as



and is represented by the following matrix:

	a	b	c
a	true	true	false
b	false	false	false
c	false	false	false

The matrix representation of a graph used in above example can be formalized as follows.

Definition 124 (Adjacency Matrix) Let $G = \langle V, E \rangle$ be a directed graph with $|V| = n$. The *adjacency matrix (Adjazenzmatrix, Nachbarschaftsmatrix)* of G is the boolean $n * n$ matrix M where $M(x, y) = \mathsf{true}$ if and only if $\langle x, y \rangle \in E$:

$$\begin{array}{l} \text{adjacency}(G) := \\ \quad \textbf{let } V = G_0, E = G_1 : \\ \qquad \textbf{such } M \in V \times V \to \{\mathsf{true}, \mathsf{false}\} : \\ \qquad\quad (\forall x \in V, y \in V : M(x, y) = \mathsf{true} \Leftrightarrow \langle x, y \rangle \in E). \end{array}$$

The adjacency matrix is the typical implementation of a graph in a computer; most graph algorithms operate with this representation.

Example Let R and S be binary relations on $\mathbb{N}_n$ for some $n \in \mathbb{N}$. The composition of R and S is

$$R \circ S = \{\langle a, c \rangle : a \in \mathbb{N}_n \wedge c \in \mathbb{N}_n \wedge (\exists b : \langle a, b \rangle \in R \wedge \langle b, c \rangle \in S)\}.$$

For the corresponding adjacency matrix, we thus have

$$\begin{array}{l} \forall i \in \mathbb{N}_n, j \in \mathbb{N}_n : \\ \quad \text{adjacency}(\langle \mathbb{N}_n, R \circ S \rangle)_{i,j} = \mathsf{true} \Leftrightarrow \\ \qquad \exists k \in \mathbb{N}_n : A_{i,k} = \mathsf{true} \wedge B_{k,j} = \mathsf{true} \\ \qquad \textbf{where } A = \text{adjacency}(\langle \mathbb{N}_n, R \rangle), B = \text{adjacency}(\langle \mathbb{N}_n, S \rangle). \end{array}$$

Written as a Java method, the composition of two adjacency matrices A and B giving a result matrix C resembles matrix multiplication:

```
void compose(int n, boolean[][] A, boolean[][] B, boolean[][] C)
{
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
    {
      C[i][j] = false;
      for (int k=0; k<n; k++)
        C[i][j] = C[i][j] || (A[i][k] && B[k][j]);
    }
  return C;
}
```

In some graphs, the direction of the edges does not matter.

Definition 125 (Undirected Graph) An *undirected graph (ungerichteter Graph)* is a directed graph whose edge relation is symmetric:

$$
\begin{aligned}
&G \text{ is undirected graph} :\Leftrightarrow\\
&\quad \exists V, E:\\
&\qquad G = \langle V, E\rangle,\\
&\qquad E \subseteq V \times V,\\
&\qquad E \text{ is symmetric on } V.
\end{aligned}
$$

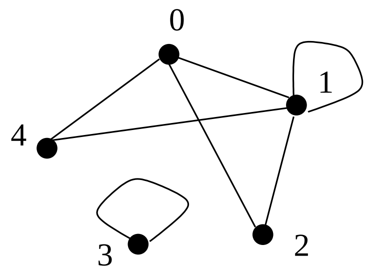
According to this definition, an undirected graph is a special directed graph; therefore all the following definitions and results for directed graphs apply to undirected graphs as well. Because of symmetry, the only information carried by an undirected graph is whether two nodes x and y are connected or not; in the visual representation it suffices to draw a single undirected edge between two connected nodes.

Example

- The graph $\langle \mathbb{N}_5, E\rangle$ with

  $$
  \begin{aligned}
  E = \{&\langle 0,1\rangle, \langle 1,0\rangle, \langle 0,2\rangle, \langle 2,0\rangle, \langle 1,1\rangle, \langle 1,2\rangle,\\
  &\langle 2,1\rangle, \langle 3,3\rangle, \langle 0,4\rangle, \langle 4,0\rangle, \langle 1,4\rangle, \langle 4,1\rangle\}
  \end{aligned}
  $$

  can be depicted as follows:



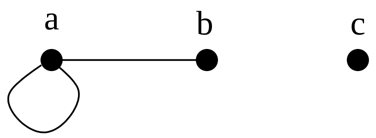
We just need to fill the elements of the upper right triangle of the matrix representation; the other elements are determined by symmetricity:

	0	1	2	3	4
0	false	true	true	false	true
1		true	true	false	true
2			false	false	false
3				true	false
4					false

- The graph $\langle\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}, E\rangle$ with

  $$\mathsf{E} = \{\langle\mathsf{a}, \mathsf{a}\rangle, \langle\mathsf{a}, \mathsf{b}\rangle, \langle\mathsf{b}, \mathsf{a}\rangle\}$$

  can be depicted as

  and has the following matrix representation:

	a	b	c
a	true	true	false
b		false	false
c			false

An important notion is the number of edges that enter or leave a node.

Definition 126 (Degree) In a directed graph, the *indegree (eingehender Grad)* of x is the number of edges whose terminal node is x:

$$\mathrm{indeg}_G(x) := |\{y \in V : \langle y, x\rangle \in E\}|$$
$$\textbf{where } V = G_0, E = G_1.$$

The *outdegree (ausgehender Grad)* of x is the number of edges whose initial node is x:

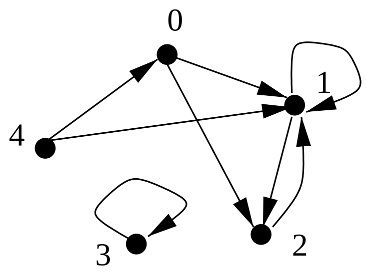
$$\text{outdeg}_G(x) := |\{y \in V : \langle x, y\rangle \in E\}|$$
$$\textbf{where } V = G_0, E = G_1.$$

The *total degree (Grad)* of x is the sum of its indegree and its outdegree:

$$\deg_G(x) := \text{indeg}_G(x) + \text{outdeg}_G(x).$$

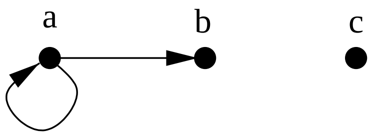
Example

- In the graph



  the indegree of node 1 is 4 and its outdegree is 2.

- In the graph



  the indegree and the outdegree of node c are both 0.

Applied to undirected graphs, above definition makes the indegree of a node equal its outdegree and the total degree be even; since in undirected graphs the only interesting information is the total degree, this number is then usually divided by two in correspondence with the visual representation.

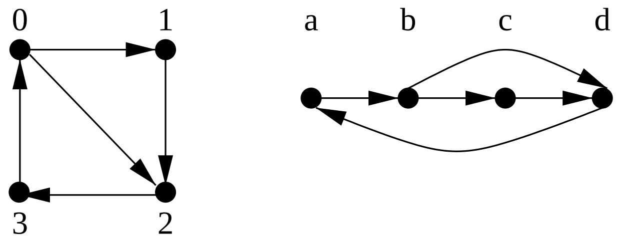
Even if two graphs are different, they may have the same structure.

Definition 127 (Graph Isomorphism) Two graphs are *isomorphic (isomorph)* if there exists a bijection between the nodes of the two graphs which preserves the edge structure:

$$
\begin{aligned}
&G \text{ and } G' \text{ are isomorphic} :\Leftrightarrow \\
&\quad G \text{ is directed graph} \wedge G' \text{ is directed graph} \wedge \\
&\quad \exists f : f : V \overset{\mathrm{iso}(E,E')}{\longrightarrow} V' \\
&\qquad \textbf{where } V = G_0, E = G_1, V' = {G'}_0, E' = {G'}_1.
\end{aligned}
$$

Example

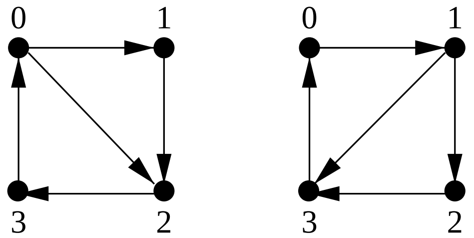
- The graphs



are isomorphic with isomorphism

$$f = \{\langle 0, \mathsf{b}\rangle, \langle 1, \mathsf{c}\rangle, \langle 2, \mathsf{d}\rangle, \langle 3, \mathsf{a}\rangle\}.$$

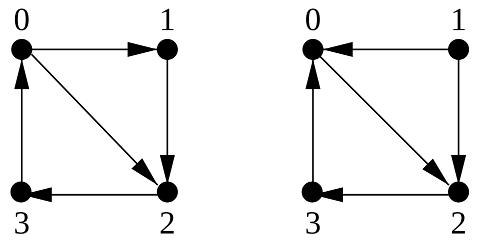
- The graphs



are isomorphic with isomorphism

$$f = \{\langle 0, 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, \langle 3, 0\rangle\}.$$

- The graphs



are not isomorphic.

The edges of a node yield paths that can be used to wander among nodes.

Definition 128 (Path) A *path (Pfad)* in a graph is a sequence of nodes connected by edges:

$$\begin{aligned}&p \text{ is path in } G :\Leftrightarrow\\&\quad (\exists n \in \mathbb{N}_{>0} : p : \mathbb{N}_n \to V \,\wedge\\&\quad\quad \forall i \in \mathbb{N}_{n-1} : \langle p_i, p_{i+1}\rangle \in E)\\&\quad \textbf{where } V = G_0, E = G_1.\end{aligned}$$

The *length (Länge)* of a path is the number of edges it contains:

$$\text{length}(p) := \textbf{such } n \in \mathbb{N} : p : \mathbb{N}_{n+1} \to V.$$

A path *from* x to y has initial node x and terminal node y :⇔

$$\begin{aligned}&p \text{ is path from } x \text{ to } y :\Leftrightarrow\\&\quad p_0 = x \wedge p_n = y\\&\quad\quad \textbf{where } n = \text{length}(p).\end{aligned}$$

A path is *simple (einfach)* if it does not contain any edge twice:

$$\begin{aligned}&p \text{ is simple} :\Leftrightarrow\\&\quad \forall i \in \mathbb{N}_n, j \in \mathbb{N}_n : \langle p_i, p_{i+1}\rangle = \langle p_j, p_{j+1}\rangle \Rightarrow i = j\\&\quad\quad \textbf{where } n = \text{length}(p).\end{aligned}$$

A path is *elementary (elementar)* if it does not contain any node twice:

$$\begin{aligned}
&p \text{ is elementary} :\Leftrightarrow \\
&\quad \forall i \in \mathbb{N}_n, j \in \mathbb{N}_n : p_i = p_j \Rightarrow i = j \\
&\quad\quad \textbf{where } n = \text{length}(p).
\end{aligned}$$

A path is a *cycle (Kreis)* or *circuit* if it terminates in its initial node:

$$p \text{ is cycle} :\Leftrightarrow \exists x : p \text{ is path from } x \text{ to } x.$$

Reachability We will now investigate how to determine whether two nodes are directly or indirectly connected in a graph.

Definition 129 (Reachable) A node y is *reachable (erreichbar)* from a node x in a graph G if there is a path in G from x to y:

$$\begin{aligned}
&y \text{ is reachable from } x \text{ in } G :\Leftrightarrow \\
&\quad \exists p : p \text{ is path in } G \land p \text{ is path from } x \text{ to } y.
\end{aligned}$$

The predicate "is reachable" is, for fixed graph $G{=}\langle V, E\rangle$, a binary relation on V, as well as the edge relation E is. Actually we can derive from E the reachability relation by a general construction that we are going to elaborate.

Definition 130 (Reflexive and Transitive Closure) Let R be a binary relation on S. The *reflexive closure (reflexiver Abschluß)* of R on S is the smallest relation that contains R and is reflexive on S:

$$\begin{aligned}
&\text{reflexive}_S(R) := \\
&\quad \textbf{such } R' \subseteq S \times S : \\
&\quad\quad R \subseteq R' \land R' \text{ is reflexive on } S \land \\
&\quad\quad \forall R'' : (R \subseteq R'' \land R'' \text{ is reflexive on } S) \Rightarrow R' \subseteq R''.
\end{aligned}$$

The *transitive closure (transitiver Abschluß)* of R on S is the smallest relation that contains R and is transitive on S:

$$\begin{aligned}
&\text{transitive}_S(R) := \\
&\quad \textbf{such } R' \subseteq S \times S : \\
&\quad\quad R \subseteq R' \land R' \text{ is transitive on } S \land \\
&\quad\quad \forall R'' : (R \subseteq R'' \land R'' \text{ is transitive on } S) \Rightarrow R' \subseteq R''.
\end{aligned}$$

We then can derive the following result.

Proposition 131 (Reachability is Closure of Edge Relation) We define the reachability relation

$$R_G := \{\langle x, y\rangle \in G_0 \times G_0 : y \text{ is reachable from } x \text{ in } G\}.$$

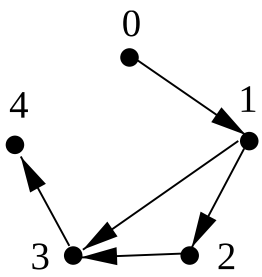
Then, for any directed graph $\langle V, E\rangle$, $R_{\langle V,E\rangle}$ is the reflexive and transitive closure of E on V:

$$\begin{aligned}&\forall V, E : \langle V, E\rangle \text{ is directed graph} \Rightarrow\\&\quad R_{\langle V,E\rangle} = \text{reflexive}_V(\text{transitive}_V(E)).\end{aligned}$$

Example Take the graph $\langle \mathbb{N}_5, E\rangle$ with

$$E = \{\langle 0, 1\rangle, \langle 1, 2\rangle, \langle 1, 3\rangle, \langle 2, 3\rangle, \langle 3, 4\rangle\}$$

which can be depicted as:



Since $\langle 0, 1\rangle \in E$ and $\langle 1, 3\rangle \in E$, $\langle 0, 3\rangle$ is in the transitive closure of E. Since also $\langle 3, 4\rangle$ is in E, also $\langle 0, 4\rangle$ is in the transitive closure of E, i.e., node 4 is reachable from 0. In total, we have:

$$\begin{aligned}&\text{transitive}_{\mathbb{N}_5}(E) =\\&\quad \{\langle 0, 1\rangle, \langle 0, 2\rangle, \langle 0, 3\rangle, \langle 0, 4\rangle, \langle 1, 2\rangle, \langle 1, 3\rangle, \langle 1, 4\rangle, \langle 2, 3\rangle, \langle 2, 4\rangle\}.\end{aligned}$$

In addition, every node is reachable from itself:

$$\text{reflexive}_{\mathbb{N}_5}(E) = \{\langle 0, 0\rangle, \langle 1, 1\rangle, \langle 2, 2\rangle, \langle 3, 3\rangle, \langle 4, 4\rangle\}.$$

Above definitions of closures are too inefficient for a practical algorithm that decides the reachability of a node; a direct construction of the reflexive closure is provided by the following result which is easy to establish.

Proposition 132 (Reflexive Closure) The reflexive closure of R on S is the union of R and the identity relation on S:

$$\begin{aligned}&\forall S, R : R \subseteq S \times S \;\Rightarrow\\&\quad \text{reflexive}_S(R) = R \cup (S \times S).\end{aligned}$$

An efficient computation of the transitive closure is based on the following operation that composes a relation R an arbitrary number of times.

Definition 131 (Exponentiation of Relations)

$$\begin{aligned}&R^0_S := \{\langle x, x\rangle : x \in S\}\\&R^{i+1}_S := R^i_S \circ R.\end{aligned}$$

R^0_S is the identity relation and R^1_S is R. Consequently, e.g.

$$R^3_S = R \circ R \circ R.$$

The transitive closure is then characterized by the following result which we state without proof.

Proposition 133 (Transitive Closure)

$$\begin{aligned}&\forall S, R : R \subseteq S \times S \Rightarrow\\&\quad \text{transitive}_S(R) = \bigcup\{R^i_S : i \in \mathbb{N}_{>0}\}\end{aligned}$$

In other words, the transitive closure of R is the union of all members of the sequence

$$[R, R \cup (R \circ R), R \cup (R \circ R) \cup (R \circ R \circ R), \ldots]$$

that starts with R and computes each subsequent member by adding to its predecessor the transitive consequences of R. Since every member is contained in its successor, the transitive closure is the "limit" of this sequence. This approach is constructive because one can show that the sequence converges, i.e., from a certain point on no more elements are added; the final sequence member represents the transitive closure.

Proposition 134 (Transitive Closure) Let R be a binary relation on S where S has n elements. Then $\bigcup_{1 \leq i \leq n} R^i_S$ is the transitive closure of R:

$$\begin{array}{l} \forall S, R : R \subseteq S \times S \;\Rightarrow \\ \quad \text{transitive}_S(R) = \bigcup_{1 \leq i \leq n} R^i_S \\ \quad \textbf{where } n = |S|. \end{array}$$

Since the reflexive and transitive closure of the edge relation describes the reachability relation, we thus have a constructive method of computing the reachability relation of a graph $\langle V, E \rangle$; we start with the identity relation and add n times the transitive consequences of R:

$$\begin{array}{l} \text{reachability}(V, E): \\ \quad n = |V| \\ \quad R^0 = \{\langle x, x \rangle : x \in V\} \\ \quad \textbf{for}(i = 0; i < n; i{+}{+}) \\ \quad\quad R^{i+1} = R^i \cup (R^i \circ E) \\ \quad \textbf{return } R^n \end{array}$$

For an actual implementation, we usually turn to the representation of the graph as a $n*n$ matrix M. Since in this representation the composition of two relations resembles matrix multiplication the computation of reachability is basically a sequence of matrix multiplications (see the example on page 105).

Trees Trees are a special kind of graphs that are of particular relevance in computer science: they provide a way to represent hierarchical structures such as a family genealogy, an organizational chart, or a file system.

Definition 132 (Tree) A *tree (Baum)* is a directed graph such that there is exactly one node, the *root (Wurzel)*, that has indegree zero, every other node has indegree one, and every node can be reached from the root.

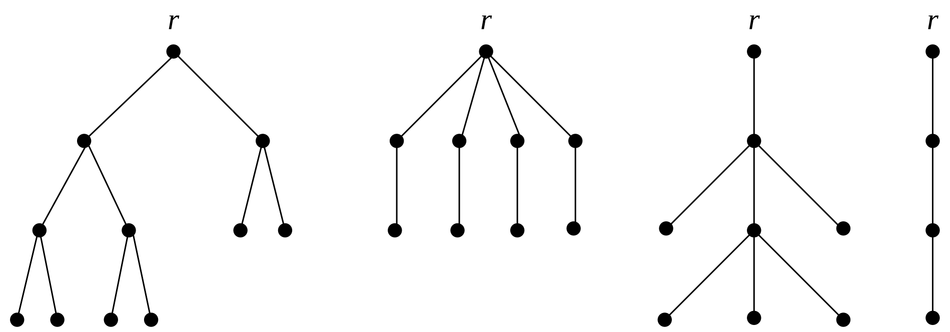
$$
\begin{array}{l}
T \text{ is tree} :\Leftrightarrow \\
\quad T \text{ is directed graph} \wedge \\
\quad \exists r \in V : \mathrm{indeg}(r) = 0 \wedge \\
\quad\quad \forall x \in V - \{r\} : \\
\quad\quad\quad \mathrm{indeg}(x) = 1 \wedge \\
\quad\quad\quad x \text{ is reachable from } r \text{ in } T \\
\quad \textbf{where } V = T_0.
\end{array}
$$

$$\mathrm{root}(T) := (\textbf{such } r \in V : \mathrm{indeg}(r) = 0) \textbf{ where } V = T_0.$$

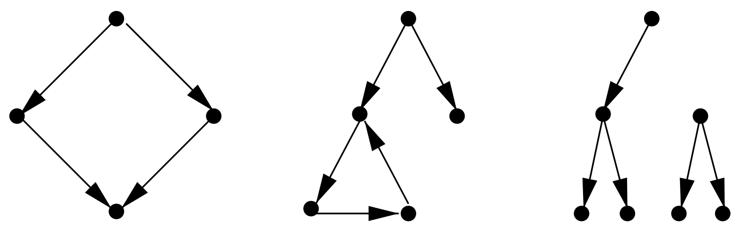
Trees are depicted with the root node at the top and all arcs directed downwards (such that arrowheads need not be given)[1].

Example

- The following diagrams depict trees with root r:

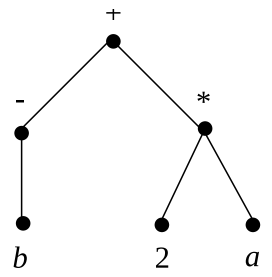


- The following directed graphs are *not* trees:



[1]In computer science, trees grow downwards! ;-)

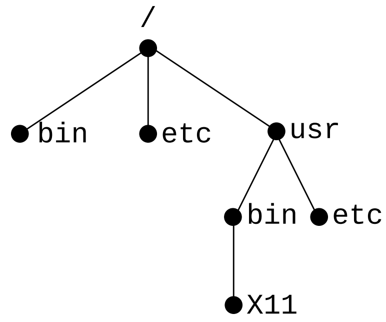
- The term $-b+2a$ is represented by the following tree:



- A file system with directories

  `/`, `/bin`, `/etc`, `/usr`, `/usr/bin`, `/usr/bin/X11`, `/usr/etc`

  is represented by the following tree:



We state without proof the following important property of trees.

Proposition 135 (Trees and Cycles) A tree has only cycles of length 0:

$$\begin{array}{l}\forall T : T \text{ is tree } \Rightarrow \\ \quad \neg(\exists p : p \text{ is path in } T \wedge \text{length}(p) > 0 \ \wedge p \text{ is cycle}).\end{array}$$

The following notions describe relations among tree nodes.

Definition 133 (Node Relations) Let T be a tree. A node y is called a *child (Kind)* of x if there is an edge from x to y in T:

$$\begin{array}{l}y \text{ is child of } x \text{ in } T :\Leftrightarrow \\ \quad \langle x, y\rangle \in E \textbf{ where } E = T_1.\end{array}$$

x is then called the *parent (Elternteil)* of y:

$$\begin{aligned}&\mathrm{parent}_T(y) := \textbf{such } x \in V : \langle x, y\rangle \in E\\&\quad \textbf{where } V = T_0, E = T_1.\end{aligned}$$

A node x is a *leaf (Blatt)*, if it does not have children:

$$\begin{aligned}&x \text{ is leaf in } T :\Leftrightarrow x \in V \wedge \neg\exists y : y \text{ is child of } x \text{ in } T\\&\quad \textbf{where } V = T_0.\end{aligned}$$

A node x is an *ancestor (Vorfahre)* of y if there is a path from x to y in T:

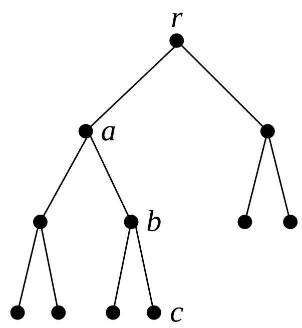
$$\begin{aligned}&x \text{ is ancestor of } y \text{ in } T :\Leftrightarrow\\&\quad \exists p : p \text{ is path in } T \wedge p \text{ is path from } x \text{ to } y.\end{aligned}$$

y is then called a *descendant (Nachkomme)* of x:

$$y \text{ is descendant of } x \text{ in } T :\Leftrightarrow x \text{ is ancestor of } y \text{ in } T.$$

Please note that in a tree every node (apart from the root) has a parent and that this parent is unique. Every node is a descendant of the root; the only descendant of a leaf node is the node itself.

Example In the tree



node a is the parent of b and an ancestor of leaf c.

In a tree, all paths from the root are unique.

Proposition 136 (Unique Root Paths) Let T be a tree with root r. Then the path from r to every node of T is unique:

$$\begin{array}{l} \forall T : T \text{ is tree } \Rightarrow \\ \quad \forall x \in V : \\ \quad\quad \exists p : p \text{ is path in } T \ \wedge p \text{ is path from } r \text{ to } x \wedge \\ \quad\quad\quad \forall p' : (p' \text{ is path in } T \wedge p' \text{ is path from } r \text{ to } x) \Rightarrow p' = p \\ \quad \textbf{where } V = T_0, r = \text{root}(T). \end{array}$$

Because of this result, the following notions are uniquely defined.

Definition 134 (Level and Height) The *level (Stufe)* of a node x in a tree is the length of the path from the root of the tree to x:

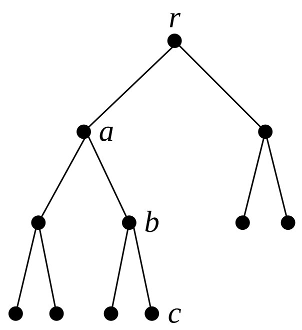
$$\begin{array}{l} \text{level}_T(x) := \text{length}(p) \\ \quad \textbf{where } p = \textbf{such } p : p \text{ is path in } T \wedge p \text{ is path from root}(T) \text{ to } x. \end{array}$$

The *height (Höhe)* of a tree is the maximum level of its nodes:

$$\text{height}(T) := \max\{\text{level}_T(x) : x \in V\} \textbf{ where } V = T_0.$$

The definition implies that the root node has level 0 and the level of a node is one plus the level of its parent.

Example In the tree



a has level 1, b has level 2, and c has level 3. The height of the tree is 3.

We conclude this section by a result which relates the degrees of nodes to the heights of trees.

Proposition 137 (Height of Binary Trees) Let T be a tree of height h where every node has an outdegree of at most 2. Then the number of nodes in the tree is less than 2^{h+1}:

$$\forall T : (T \text{ is tree} \land \forall x \in V : \text{outdeg}(x) \leq 2) \Rightarrow |V| < 2^{h+1}$$
$$\quad \textbf{where } V = T_0, h = \text{height}(T).$$

Proof Let T be a tree where every node has an outdegree of at most 2. We proceed by complete induction on the height of T.

1. Assume the height is $h = 0$. Then $|V| = 1 < 2 = 2^{h+1}$.
2. Assume the height is $h > 0$. Consequently the root of T has a child that is the root of a tree of height $h - 1$ and possibly a second child that is the root of a tree of height less than or equal $h - 1$. By the induction hypothesis, we thus have

$$|V| \leq 1 + (2^h - 1) + (2^h - 1) = 2^{h+1} - 1 < 2^{h+1}.$$

It is easy to generalize above result to trees whose outdegrees are bounded by an arbutrary limit k (how?).

Appendix A

Defining New Notions

When formulating propositions for a particular domain, it becomes soon cumbersome to commit oneself to the predicates and functions provided by the domain. For instance, while set theory (see Chapter 3) offers a single binary predicate '$\in$' that in principle suffices to express all facts about sets, formulas become very large and difficult to understand if they are expressed only with the help of this predicate.

Therefore we need mechanisms that allow us to introduce new predicate and function constants that capture under a single name properties respectively objects whose description with the help of the basic notions is rather large. By this *abstraction (Abstraktion)*, we can refer to a notion just by its name without having to repeat its description in different contexts over and over again. If descriptions are similar but not identical, we may extract by *parameterization (Parameterisierung)* a common core or a more general notion that is suitable for abstraction.

The new constants may be again used for the introduction of other constants, thus we can build in an iterative *bottom up (von unten nach oben)* process new layers of abstractions on top of previously constructed ones.

Actually when formalizing aspects of the real world or specifying problems, this process typically proceeds *top down (von oben nach unten)*: we introduce a name for the notion of interest and then construct a description that reduces this notion to simpler notions. We may use these simpler notions just as "black boxes" or again describe them in terms of simpler notions until we reach a level where the notions are sufficiently well understood. This is in particular the case, if these notions are the elementary predicates and functions of a formally characterized domain (such as set theory).

This method of constructing descriptions by *stepwise refinement (schrittweise Verfeinerung)* is of particular importance in computer science that has to deal with complex and often only vaguely understood aspects of the real world.

A.1 Preliminaries

In mathematics, a formal description is called a *definition*; a definition constructs a new *theory* from a given one.

Definition 135 (Theory) A *theory (Theorie)* consists of a set of constants and a set of formulas that only use constants from the given set. We call these formulas the *axioms (Axiome)* of the theory.

The axioms of a theory express those propositions that are stipulated to be true; from these propositions other true propositions can be derived.

Example

- The theory of Peano Arithmetic consists of constants

  0 (zero), $'$ (successor)

  and the axioms that are stated in Section 4.1.

- The theory of sets (see Chapter 3) has a single binary predicate constant

  $\in$ (is element of)

  and the Axioms of Zermelo and Fraenkel.

- The theory of real numbers has constants

  $$0, 1, +, *, \leq$$

  and the axioms that are stated in Section 4.4.

- The theory of *lists* has the constants

  nil, cons, isnil, head, tail

  and axioms like $(\forall e, l : \text{head}(\text{cons}(e, l)) = e)$.

Definition 136 (Definition) A *definition (Definition)* is a statement that introduces into a theory a new constant and new axioms such that from the extended set of axioms no formula can be derived that could not be derived from the original axioms (provided that this formula only uses constants from the original theory).

A definition just introduces syntactic abbreviations for entities that are already implicitly given (but not explicitly named) by a theory; they do not add any conceptual power to a theory (nor do they remove some).

Example The axioms of the set theory demand the existence of a set that does not contain any elements. We introduce the constant $\emptyset$ to denote this set. We may conclude $(\forall x : x \notin \emptyset)$.

In the following, we will discuss various techniques to write definitions.

A.2 Explicit Predicate Definitions

Definition 137 (Explicit Predicate Definition) Let p be a predicate constant of arity n that is not yet in use, $x_0, \ldots, x_{n-1}$ be n distinct variables, and F be a formula in which p does not occur and with no other free variables than $x_0, \ldots, x_{n-1}$. Then the phrase

$$p(x_0, \ldots, x_{n-1}) :\Leftrightarrow F$$

is an *explicit predicate definition (explizite Prädikatsdefinition)* which introduces an n-ary predicate constant p together with an axiom

$$\forall x_0, \ldots, x_{n-1} : p(x_0, \ldots, x_{n-1}) \Leftrightarrow F.$$

We call $p(x_0, \ldots, x_{n-1})$ the *definiendum* ("the one to be defined") and F the *definiens* ("the defining one").

A predicate p is uniquely characterized by an explicit definition, i.e., there is exactly one p that satisfies the new axiom (because for every $x_0, \ldots, x_{n-1}$, F has a single truth value).

The restriction that p does not occur in F is necessary, e.g. to avoid bogus "definitions" such as

$$p(x) :\Leftrightarrow \neg p(x)$$

which lead to a contradiction. The restriction that all free variables of the definiens must occur in the definiendum is to avoid "definitions" as

$$p(x) :\Leftrightarrow x \leq y.$$

where $p(1)$ may hold or not, depending on the current assignment of y.

A predicate definition in natural language is usually expressed as

x is (called) a P, if ...,

introducing the unary predicate $P(x)$, or

x and y are P, if ...

introducing the binary predicate $P(x, y)$, or

x is a P of y, if ...

which also introduces $P(x, y)$.

The new constant being defined is often typeset in *italics* or <u>underlined</u>.

Example

- The statement

  p is *prime*, if its only divisors are 1 and p.

  introduces the unary predicate

  p <u>is prime</u> $:\Leftrightarrow (\forall p : p$ is a divisor of $x \Rightarrow (p = 1 \vee p = x))$.

- The statement

  x is a *divisor* of y, if $x * z = y$, for some z.

  introduces the binary predicate

$$x \underline{\text{ is a divisor of }} y :\Leftrightarrow \exists z : x * z = y.$$

- The statement

  x is a *divisor* of $a * y$, if $x * z = a * y$, for some z.

  is *not* a definition, because on the left hand side of the formula

  $$x \text{ is a divisor of } a * y \Leftrightarrow \exists z : x * z = a * y.$$

  a general term $a * y$ appears where only a variable is allowed.

Logic Evaluator A predicate is explicitly defined by a statement

pred $p(x_0$, ..., $x_{n-1})$ <=> F;

which is to be read as $p(x_0, \ldots, x_{n-1}) :\Leftrightarrow F$.

```
pred divides(x, y) <=>
  and(Nat(x), Nat(y),
    exists(z in nat(1, y): =(*(x, z), y)));
> predicate divides/2.
pred even(x) <=> not(divides(2, x));
> predicate even/1.
pred isprime(x) <=>
  and(Nat(x), forall(p in nat(1, x):
    implies(divides(p, x), or(=(p, 1), =(p, x)))));
> predicate isprime/1.
formula divides(3, 12);
> true.
formula divides(3, 13);
> false.
formula isprime(13);
> true.
formula isprime(97);
> true.

formula forall(x in nat(3, 30): implies(isprime(x), even(x)));
```

Constrained Predicate Definitions Sometimes a predicate definition conjoins some "side-conditions" by saying

Let x be such that P. Then x is a Q if F.

or, shorter,

A P is a Q if F.

This is to be interpreted as the definition of a predicate by a conjunction:

$$Q(x) :\Leftrightarrow P(x) \wedge F$$

which we usually express in the form defined below (provided that we operate in the domain of sets).

Definition 138 (Constrained Predicate Definition) Let p be a predicate constant of arity n that is not yet in use and S be a term in which p does not occur, $x_0, \ldots, x_{n-1}$ be n distinct variables, and F be a formula in which p does not occur and with no other free variables than $x_0, \ldots, x_{n-1}$. Then

$$\begin{array}{l} p \subseteq S \\ p(x_0, \ldots, x_{n-1}) :\Leftrightarrow F \end{array}$$

is a *constrained predicate definition* which introduces an n-ary predicate p such that

$$\forall x_0, \ldots, x_{n-1} : p(x_0, \ldots, x_{n-1}) \Leftrightarrow (\langle x_0, \ldots, x_{n-1} \rangle \in S \wedge F).$$

Since a constraint just adds a condition, the corresponding predicate is still uniquely defined.

Example

- The definition

  $$\begin{array}{l} | \subseteq \mathbb{N} \times \mathbb{N} \\ x|y :\Leftrightarrow \exists p \in \mathbb{N} : x * p = y \end{array}$$

  is to be understood as

  $$x|y :\Leftrightarrow x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge \exists p \in \mathbb{N} : x * p.$$

- The definition

  > Let f be a binary relation. f is a *partial function* if every element of its domain is in relation to at most one element of its range.

  as well as the definition

  > A *partial function* is a binary relation where every element of the domain is in relation to at most one element of the range.

  are to be read as the definition of a unary predicate

  $$\begin{array}{l} f \text{ is a partial function} :\Leftrightarrow \\ \quad f \text{ is a binary relation} \wedge \\ \quad \forall x, y_0, y_1 : (\langle x, y_0 \rangle \in f \wedge \langle x, y_1 \rangle \in f) \Rightarrow y_0 = y_1. \end{array}$$

A.3 Explicit Function Definitions

Definition 139 (Explicit Function Definition) Let f be a function constant of arity n that is not yet in use, $x_0, \ldots, x_{n-1}$ be n distinct variables, and T be a term in which f does not occur and with no other free variables than $x_0, \ldots, x_{n-1}$. Then the phrase

$$f(x_0, \ldots, x_{n-1}) := T$$

is an *explicit function definition (explizite Funktionsdefinition)* which introduces an n-ary function constant f together with an axiom

$$\forall x_0, \ldots, x_{n-1} : f(x_0, \ldots, x_{n-1}) = T.$$

We call $f(x_0, \ldots, x_{n-1})$ the *definiendum* ("the one to be defined") and T the *definiens* ("the defining one").

A function f introduced by an explicit definition is uniquely characterized, i.e., there is exactly one f that satisfies the stated law (because for every $x_0, \ldots, x_{n-1}$, T has a single value).

The restriction that f must not occur in T avoids contradictions like in

$$f(x) := 1 + f(x).$$

All free variables of the definiendum must occur in the definens in order to avoid a dependence on the current variable assignment, such as in

$$f(x) := x + y$$

where $f(1)$ may be 2, 3, or any other number (depending on the current assignment of y).

Logic Evaluator A function is explicitly defined by a statement

fun f(x_0, ..., x_{n-1}) = T;

which is to be read as $f(x_0, \ldots, x_{n-1}) := T$.

```
fun f(x) = +(1, f(x))> ERROR: undefined function f/1.
;
> definition is ignored.
fun ^2(x) = *(x, x);
> function ^2/1.
fun square(m, n) = set(x in nat(m, n): true, ^2(x));
> function square/2.
term square(3, 15);
> {9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225}.
read set;
> file 'set.txt' read.

term ++(square(3, 15), square(10, 20));
```

Conditional Definitions The value of a function is sometimes defined in different ways for different kinds of arguments, which can be expressed as

$$f(x_0, \ldots, x_{n-1}) := \begin{array}{ll} T_0, & \textbf{if } F \\ T_1, & \textbf{else (otherwise)} \end{array}$$

with terms T_0 and T_1 and formula F in all of which f does not occur and with no other free variables than $x_0, \ldots, x_{n-1}$. This is another form of writing

$$f(x_0, \ldots, x_{n-1}) := \textbf{if } F \textbf{ then } T_0 \textbf{ else } T_1$$

where the phrase (**if** F **then** T_0 **else** T_1) is introduced below.

Definition 140 (Conditional Term) For every formula F and terms T_0 and T_1, the phrase

$$(\textbf{if } F \textbf{ then } T_0 \textbf{ else } T_1)$$

is a term whose value is the value of T_0, if F holds, and the value of T_1, otherwise. We omit the parentheses, if T_1 is clear.

Example The definition

$$|x| := \begin{array}{l} -x, \textbf{ if } x < 0 \\ x, \textbf{ else} \end{array}$$

is another form of writing

$$|x| := \textbf{if } x < 0 \textbf{ then } -x \textbf{ else } x$$

which denotes the *absolute value* of x.

Logic Evaluator A conditional term is denoted as

`if(`F`, `T_0`, `T_1`)`

which is to be read as (**if** F **then** T_0 **else** T_1).

```
pred divides(x, y) <=>
  and(Nat(x), Nat(y),
    exists(z in nat(1, y): =(*(x, z), y)));
> predicate divides/2.
pred even(x) <=> divides(2, x);
> predicate even/1.
fun f(x) = if(even(x), *(x, 2), -(x, 1));
> function f/1.
term f(10);
> 20.
term f(11);
> 10.
```

```
term if(even(f(5)), f(f(5)), f(4));
```

Constrained Function Domains In a definition $f(x_0, \ldots, x_{n-1}) := T$, the value of T may not be defined for some $x_0, \ldots, x_{n-1}$, i.e., there is no knowledge that relates this value to any other value.

Example We define the function over the real numbers

$$f(x) := 1/x.$$

We do not know anything about $f(0)$ because we do not know anything about the value $1/0$.

In order to make undefined function values explicitly visible, we often write a definition in the following format (provided that we operate in the domain of sets):

Definition 141 (Constrained Function Definition) Let f be a function constant of arity n that is not yet in use, A and B be terms in which f does not occur, $x_0, \ldots, x_{n-1}$ be n distinct variables, and T be a term in which f does not occur and with no other free variables than $x_0, \ldots, x_{n-1}$. Then

$$f : A \to B$$
$$f(x_0, \ldots, x_{n-1}) := T$$

is a *constrained function definition* which introduces an n-ary function constant f such that

$$\forall x_0, \ldots, x_{n-1}, y : f(x_0, \ldots, x_{n-1}) = y \Leftrightarrow (\langle x_0, \ldots, x_{n-1}\rangle \in A \wedge y = T).$$

Above definition characterizes f uniquely.

Please note that, for every $x_0, \ldots, x_{n-1}$, we have to prove

$$\langle x_0, \ldots, x_{n-1}\rangle \in A$$

before we are allowed to deduce

$$f(x_0, \ldots, x_{n-1}) = T.$$

Furthermore, we still have to prove

$$(\forall x_0, \ldots, x_{n-1} : \langle x_0, \ldots, x_{n-1}\rangle \in A \Rightarrow T \in B)$$

before we are allowed to assume

$$f(x_0, \ldots, x_{n-1}) \in B.$$

Example We define

$$\begin{aligned} &|\ \ | : \mathbb{R} \to \mathbb{R}_{\geq 0} \\ &|x| := -x,\ \textbf{if } x < 0 \\ &\qquad\ \ x,\ \textbf{else} \end{aligned}$$

where $\mathbb{R}$ denotes the set of real numbers and $\mathbb{R}_{\geq 0}$ denotes the set of non-negative real numbers. For every $a \in \mathbb{R}$, we have

$$|a| = \textbf{if } a < 0 \textbf{ then } -a \textbf{ else } a.$$

However, the value of $|i|$ (where i denotes the imaginary number) is undefined because $i \notin \mathbb{R}$.

Furthermore, we can prove

$$\forall x : (\textbf{if } x < 0 \textbf{ then } -x \textbf{ else } x) \in \mathbb{R}_{\geq 0}$$

because, for a real $x < 0$, $-x \in \mathbb{R}_{\geq 0}$, and for a real $x \not< 0$, $x \in \mathbb{R}_{\geq 0}$. Thus we indeed have

$$|\ \ | : \mathbb{R} \to \mathbb{R}_{\geq 0}.$$

Logic Evaluator A constraint on the function domain can be expressed in various ways:

- For a unary predicate P, a parameter declaration

  f(*x*: *P*)

  lets the evaluation of a function application $f(a)$ abort with an error message, if $P(a)$ does not hold. We have the predefined unary predicates `Nat`, `Set`, `Tuple` that test whether a value is a natural number, a set, or a tuple, respectively.

- Likewise for a term S denoting a set, a parameter declaration

  f(*x* `in` *S*)

  lets the evaluation of a function application $f(a)$ abort if $a \notin S$.

- We may use the binary function `if` to define a function as

  `fun` f(*x*) `= if(`*F*`,` *T*`);`

  If the condition F does not hold on the input arguments, the evaluation of the function is aborted with an error message.

The checking of constraints can be switched off by the command

```
option check = false;
```

which may speed up the evaluation.

```
fun f(x: Nat) = x;
> function f/1.
term f(1);
> 1.
term f({});
> ERROR: value {} does not satisfy predicate Nat/1 for parameter x.
option check = false;
> execution checks are off.
term f({});
> {}.
option check = true;
> execution checks are on.
pred NonZero(x) <=> and(Nat(x), not(=(x, 0)));
> predicate NonZero/1.
fun dec(x: NonZero) = -(x, 1);
> function dec/1.
term dec(4);
> 3.

term dec(0);
```

A.4 Implicit Function Definitions

In some situations, it is more convenient to describe a function rather by a condition between the argument values and the result value than by a term that denotes the result value. Such an *implicit function definition (implizite Funktionsdefinition)* is typically written in the form.

> Let $f(x_0, \ldots, x_{n-1})$ be a y such that F.

However, this is just another way of writing an explicit definition

$$f(x_0, \ldots, x_{n-1}) := \textbf{such } y : F$$

where (**such** y: F) is a term in which f does not occur and with no other free variables than $x_0, \ldots, x_{n-1}$ (y is bound).

Definition 142 (Such Quantifier) For every variable x and formula F, the phrase

$$(\textbf{such } x : F)$$

is a term with bound variable x.

The value of this term is some x such that F holds (if such an x exists). We omit the parentheses if F is clear.

Frequently, we write

$$\textbf{such } x \in S : F$$

to denote **such** x: $x \in S \wedge F$.

Reasoning When reasoning about implicitly defined objects with have to take special care because the definition may be inconsistent (i.e., no object with the denoted property exists) or not unique (multiple objects satisfy the property).

All that we actually have is the knowledge

$$(\exists x : F) \Rightarrow (\forall x : x = (\textbf{such } x : F) \Rightarrow F),$$

i.e., if an object with property F exists, then the object denoted by the “such term” satisfies F as well.

Consequently, we have for a function definition

$$f(x_0, \ldots, x_{n-1}) := \textbf{such } y : F$$

the knowledge

$$(\forall x_0, \ldots, x_{n-1} : (\exists y : F) \Rightarrow (\forall y : y = f(x_0, \ldots, x_{n-1}) \Rightarrow F)),$$

i.e., for any tuple of function arguments the result value satisfies F, provided that such a value indeed exists.

Therefore, for given $x_0, \ldots, x_{n-1}$, we first have to prove

$$\exists y : F.$$

before we may assume that, for every $y = f(x_0, \ldots, x_{n-1})$, F holds.

Example We define

$$x/y := \textbf{such } q : q * y = x.$$

Then $12/4 = 3$ because $3 * 4 = 12$.

Take $a := 1/0$. We must *not* assume that $a * 0 = 1$, because we cannot prove $\exists q : q * 0 = 1$.

Such Terms and Predicates An alternative to defining a function

$$f(x) := \textbf{such } y : F$$

is always to define a predicate

$$p(x, y) :\Leftrightarrow F.$$

Apparently, we then have the relationship

$$f(x) := \textbf{such } y : p(x, y).$$

However, it may be easier to understand (and to reason about) statements that are formulated with the help of the predicate than with the corresponding "such" term: for every formula G, the statement

$$G[y \leftarrow f(x)]$$

is equivalent to

$$(\forall y : p(x, y) \Rightarrow G)$$

which may be preferrable in some situations.

Example We define in Section 6.4 the limit of a sequence s as

$$\lim(s) := \textbf{such } a : s \text{ converges to } a.$$

We then wish to state that, for all sequences s and t,

$$\lim(s + t) = \lim(s) + \lim(t)$$

where, for every $i \in \mathbb{N}$, $(s + t)_i = s_i + t_i$.

Inserting the definition of "lim" gives us the statement

$$\begin{aligned}&(\textbf{such } a : s + t \text{ converges to } a) =\\ &\quad(\textbf{such } b : s \text{ converges to } b) + (\textbf{such } c : t \text{ converges to } c)\end{aligned}$$

which is rather difficult to prove.

However, above statement can be also formulated as

$$\begin{aligned}&\forall a, b, c :\\ &\quad(s + t \text{ converges to } a \wedge s \text{ converges to } b \;\wedge t \text{ converges to } c) \;\Rightarrow\\ &\qquad a = b + c.\end{aligned}$$

which is much more amenable to a formal proof.

As the previous example shows, it may be advisable to "switch" from the formulation of a statement with "such terms" to a statement with corresponding predicates to understand the subtleties of the statement.

Logic Evaluator We have a restricted form of the "such" quantifier that allows to express the value of the term (**such** $x \in S$: F) as

such(x in S: F, x)

In general, the quantifier binds multiple variables $x_0, \ldots, x_{n-1}$ and returns a value of term T where F holds:

such(x_0 ∈ T_0, ..., x_{n-1} ∈ T_{n-1}: F, T)

The value of this term can be expressed by the standard quantifier as

$$\begin{aligned}&(\textbf{let}\\ &\quad t = \;\textbf{such } t \textbf{ that}\\ &\qquad(\exists x_0 \in T_0, \ldots, x_{n-1} \in T_{n-1} : t = \langle x_0, \ldots, x_{n-1}\rangle \wedge F)\\ &\quad x_0 = t_0,\\ &\quad \ldots\\ &\quad x_{n-1} = t_{n-1} :\\ &T)\end{aligned}$$

where t is a variable that does not occur in T and F.

```
pred divides(x, y) <=>
  exists(z in nat(1, y): =(*(x, z), y));
> predicate divides/2.
fun divisor(x) =
  such(y in nat(2, x): divides(y, x), y);
> function divisor/1.
term divisor(6);
> 2.
term divisor(49);
> 7.
fun div(x, y) = such(z in nat(1, x): =(*(y, z), x), z);
> function div/2.
term div(70, 14);
> 5.
term div(70, 15);
> ERROR: no such value.
fun divquot(n) = such(x in nat(2, -(n, 1)), y in nat(1, x):
  =(n, *(x, y)), tuple(x, y));
> function divquot/1.
term divquot(85);
> <17, 5>.
```

```
term divquot(17);
```

Operational Interpretation A `such` expression is an instance of the Java class shown below.

```
public final class SuchThat implements Term
{
  private String variable;
  private Term domain;
  private Formula formula;
  private Term term;

  public SuchThat(String _variable, Term _domain,
                  Formula _formula, Term _term)
  {
    variable = _variable;
    domain = _domain;
    formula = _formula;
    term = _term;
  }

  public Value eval() throws EvalException
  {
    Iterator iterator = Model.iterator(domain);
    while (iterator.hasNext())
    {
```

```
        Value result = null;
        Context.begin(variable, iterator.next());
        if (formula.eval())
          result = term.eval();
        Context.end();
        if (result != null) return result;
      }
      throw new EvalException("no such value");
    }
  }
```

The Java expression `(new SuchThat(`x`, `d`, `F`, `T`)).eval()` computes the value of **let** $x = (\textbf{such } x \in d : F) : T$. As one can see, we iterate over d and establish new contexts in which x is assigned each value of the domain in turn. In each context, we evaluate F. If the formula is true, we evaluate T and return its value.

Unique Implicit Function Definitions The "such quanifier" does not necessarily determine a unique value, consequently an implicit function definition not always determines a function uniquely.

Example We define

Let $f(x)$ be a y such that $y|x$.

or, in other words,

$$f(x) := \textbf{such } y : y|x$$

where $|$ denotes the predicate "divides". Then all we know is that $f(12)|12$ (because $\exists y : y|12$), and consequently $f(12) \in \{1, 2, 3, 4, 6, 12\}$.

Frequently, however an implicit definition is given in the form

let $f(x_0, \ldots, x_{n-1})$ be <u>the</u> y such that F

which implies that there exists at most one y that satisfies F for given $x_0, \ldots, x_{n-1}$.

This is equivalent to stating, in addition to

$$f(x_0, \ldots, x_{n-1}) := \textbf{such } y : F,$$

also the condition (which has to be proved)

$$(\forall x_0, \ldots, x_{n-1}, y : F \Rightarrow y = f(x_0, \ldots, x_{n-1})).$$

Example It is *wrong* to write

> Let $f(x)$ be the y such that $y|x$.

because from $6|12$ we must not conclude $f(12) = 6$.

However, we may define

> Let x/y be the q such that $q * y = x$.

Since $3 * 4 = 12$, we may conclude $12/4 = 3$.

A.5 Recursive Definitions

The definitions discussed in the previous sections are constrained by the fact that the new notion has to be defined in terms of other notions only. However, there are also situations in which it is necessary to reduce the value of a predicate or functions to other values of the same predicate or function. Such a *recursive* definition makes sense if this reduction does not continue forever, i.e., if eventually a situation is reached where the value can be defined directly in terms of other (previously introduced) notions.

In order to bound the number of recursive unfoldings, we introduce the following concept.

Definition 143 (Well-Founded Ordering) A binary relation $\prec$ is well-founded if there is no infinitely decreasing chain with respect to $\prec$:

$$\prec \text{ is well-founded} :\Leftrightarrow \neg\exists S, s : \mathbb{N} \to S : \forall i \in \mathbb{N} : s_{i+1} \prec s_i.$$

We now drop the constraint that the definiens must not refer to the definiendum.

Definition 144 (Recursive Function Definition) Let f be a function constant of arity n that is not yet in use, $x_0, \ldots, x_{n-1}$ be n distinct variables, and T be a term with no other free variables than $x_0, \ldots, x_{n-1}$. Then

$$f(x_0, \ldots, x_{n-1}) := T$$

is a *recursive function definition (rekursive Funktionsdefinition)*, if there exists a well-founded ordering $\prec$ with the following property: for all $x_0, \ldots, x_{n-1}$ and every occurrence of $f(T_0, \ldots, T_{n-1})$ in T that is needed to determine the value of $f(x_0, \ldots, x_{n-1})$, we have

$$\langle T_0, \ldots, T_{n-1} \rangle \prec \langle x_0, \ldots, x_{n-1} \rangle.$$

This definition introduces an n-ary function constant f with the axiom

$$\forall x_0, \ldots, x_{n-1} : f(x_0, \ldots, x_{n-1}) = T.$$

The predicate "is needed" in above definition is a bit vague; for our purposes it suffices to state that every subterm of a term T is needed to determine the value of T, *unless* T is of the form

$$\textbf{if } F \textbf{ then } T_0 \textbf{ else } T_1.$$

In this case, only the value of T_0 is needed if F is true, and only the value of T_1 is needed, otherwise. A "definition" like

$$f(x) := 1 + f(x)$$

can therefore not make sense because every subterm of the definiens is needed to compute the function result.

The existence of a well-founded ordering ensures that for every argument only a finite number of recursive "unfoldings" of a function definition is required to determine the function value. Frequently, such an ordering is constructed by defining a corresponding *termination function (Terminationsfunktion)* r such that, for every $x_0, \ldots, x_{n-1}$,

$$r(x_0, \ldots, x_{n-1}) \in \mathbb{N}.$$

Then the well-founded ordering is defined as

$$x \prec y :\Leftrightarrow r(x) < r(y)$$

where $<$ is the usual ordering of the natural numbers.

Example We define

$$\begin{aligned}
&\text{any}(S) := \textbf{such } x : x \in S \\
&\text{rest}(S) := \{x \in S : x \neq \text{any}(S)\} \\
&\#S := \text{the number of elements in a finite set } S.
\end{aligned}$$

- The recursive definition

$$\begin{aligned}
&\text{sum}: \ \text{FiniteSet} \to \mathbb{N} \\
&\text{sum}(S) := \\
&\quad \textbf{if } S = \emptyset \\
&\qquad \textbf{then } 0 \\
&\qquad \textbf{else } \text{any}(S) + \text{sum}(\text{rest}(S))
\end{aligned}$$

  computes the sum of the elements of a finite set of numbers S; a corresponding termination function is

$$r(S) := \#S.$$

  For finite S, $\text{sum}(S) \in \mathbb{N}$. If $S = \emptyset$, no other application of 'sum' is needed to determine $\text{sum}(S)$. Otherwise, we need $\text{sum}(\text{rest}(S))$ with $r(\text{rest}(S)) = r(S) - 1 < r(S)$.

- The recursive definition

$$\begin{aligned}
&* : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\
&x * y := \\
&\quad \textbf{if } y = 0 \\
&\qquad \textbf{then } 0 \\
&\qquad \textbf{else } x + x * (y - 1)
\end{aligned}$$

  introduces multiplication over the natural numbers; a corresponding termination function is

$$r(x, y) := y.$$

  For $x \in \mathbb{N}$ and $y \in \mathbb{N}$, $r(x, y) \in \mathbb{N}$. If $y = 0$, no other application of $*$ is needed to determine $x * y$. If $y \neq 0$, we need $x * (y - 1)$ with $r(x, y - 1) = y - 1 < y = r(x, y)$.

- The function defined as

$$
\begin{array}{l}
\text{dsum} : \text{ FiniteSet } \times \text{ FiniteSet } \to \mathbb{N} \\
\text{dsum}(A, B) := \\
\quad \textbf{if } A = \emptyset \textbf{ then} \\
\quad\quad \text{any}(B) + \text{dsum}(A, \text{rest}(B)) \\
\quad \textbf{else if } B = \emptyset \textbf{ then} \\
\quad\quad \text{any}(A) + \text{dsum}(\text{rest}(A), B) \\
\quad \textbf{else} \\
\quad\quad \text{any}(A) + \text{any}(B) + \text{dsum}(\text{rest}(A), \text{rest}(B))
\end{array}
$$

  computes the sum of the elements of two finite sets A and B. A corresponding termination function is

$$r(A, B) := \#A + \#B.$$

Logic Evaluator We may define a function recursively by a statement

fun f(x_0, ..., x_{n-1}) recursive R = T;

where R is a term that denotes the value of a termination function.

```
fun f(x) recursive x = +(1, f(x));
> function f/1.
term f(3);
> ERROR: new induction value 3 of function f/1 is not less than old
value 3.

fun *(x: Nat, y: Nat) recursive y =
  if(=(y, 0), 0, +(x, *(x, -(y, 1))));
> function */2.
term *(7, 8);
> 56.

read set;
> file 'set.txt' read.
fun any(S: Set) = such(x in S: true, x);
> function any/1.
fun rest(S: Set) =
  let(e = any(S): set(x in S: not(=(e, x)), x));
> function rest/1.
fun prod(S: Set) recursive #(S) =
  if(=(S, {}), 1, *(any(S), prod(rest(S))));
> function prod/1.
term prod(set(x in nat(1, 5): true, x));
> 120.
```

```
term prod(set(x in nat(1, 5): true, -(x, 1)));
```

The evaluator checks that in every recursive invocation function this term is appropriately decreased (unless the potentially dangerous statement `option check = false` is executed).

Also a predicate can be defined in a recursive way.

Definition 145 (Recursive Predicate Definition) Let p be a predicate constant of arity n that is not yet in use, $x_0, \ldots, x_{n-1}$ be n distinct variables, and F be a formula and with no other free variables than $x_0, \ldots, x_{n-1}$. Then

$$p(x_0, \ldots, x_{n-1}) :\Leftrightarrow F$$

is a *recursive predicate definition (rekursive Prädikatsdefinition)* provided that there exists a well-founded ordering $\prec$ with the following property: for all $x_0, \ldots, x_{n-1}$, and every occurrence of $p(T_0, \ldots, T_{n-1})$ in F that is needed to determine the truth value of $p(x_0, \ldots, x_{n-1})$ we have

$$\langle T_0, \ldots, T_{n-1} \rangle \prec \langle x_0, \ldots, x_{n-1} \rangle.$$

This definition introduces an n-ary predicate constant p with the axiom

$$\forall x_0, \ldots, x_{n-1} : p(x_0, \ldots, x_{n-1}) \Leftrightarrow F.$$

As for functions, the predicate "is needed" needs some refinement. We introduce a new kind of formula.

Definition 146 (Conditional Formula) For every formula F and terms F_0 and F_1, the phrase

$$(\textbf{if } F \textbf{ then } F_0 \textbf{ else } F_1)$$

is a formula whose truth value is the value of F_0, if F holds, and the value of F_1, otherwise. We omit the parentheses, if F_1 is clear.

By above definition, we have the relationship

$$(\textbf{if } F \textbf{ then } F_0 \textbf{ else } F_1) \Leftrightarrow ((F \Rightarrow F_0) \wedge (\neg F \Rightarrow F_1)).$$

We now state that every subformula of a formula F is needed to determine the value of F, *unless* F is of the form

$$\textbf{if } F \textbf{ then } F_0 \textbf{ else } F_1.$$

In this case, only the value of F_0 is needed if F is true, and only the value of F_1 is needed, otherwise.

Example

- The definition

  $$\begin{array}{l} \text{iseven} \subseteq \mathbb{N} \\ \text{iseven}(x) :\Leftrightarrow \\ \quad \textbf{if } x = 0 \\ \qquad \textbf{then } \mathrm{T} \\ \qquad \textbf{else } \neg\text{iseven}(x-1) \end{array}$$

  introduces a unary predicate on natural numbers; a corresponding termination function is

  $$r(x) := x.$$

- The definition

  $$\begin{array}{l} \leq \subseteq \mathbb{N} \times \mathbb{N} \\ x \leq y :\Leftrightarrow \\ \quad \textbf{if } x = 0 \textbf{ then } \mathrm{T} \\ \quad \textbf{else if } y = 0 \textbf{ then } \mathrm{F} \\ \quad \textbf{else } x - 1 \leq y - 1 \end{array}$$

  introduces the usual ordering on natural numbers; a corresponding termination function is

  $$r(x, y) := x.$$

Logic Evaluator Similar to recursive function definitions, predicates may be recursively defined by a statement `pred` p`(`x_0`, ..., `x_{n-1}`) recursive` R `<=>` F`;`

where R is a term that denotes the value of a termination function. For writing recursive predicates, we also need the conditional formula

`if(`F`, `F_0`, `F_1`)`

whose value is (**if** F **then** F_0 **else** F_1).

```
pred iseven(x: Nat) recursive x <=>
  if(=(x, 0), true, not(iseven(-(x, 1))));
> predicate iseven/1.
formula iseven(0);
> true.
formula iseven(1);
> false.
formula iseven(17);
> false.
pred <=(x: Nat, y: Nat) recursive x <=>
 if(=(x, 0), true,
   if(=(y, 0), false,
     <=(-(x, 1), -(y, 1))));
> predicate <=/2.
formula <=(0, 3);
> true.
formula <=(10, 5);
> false.

formula <=(5, 10);
```

Reduction of Sets Frequently functions over sets are recursively defined as follows:

$$
\begin{array}{l}
g(S) := \\
\quad \textbf{if } S = \emptyset \textbf{ then } b \\
\quad \textbf{else let } e = \textbf{such } x : x \in S : \\
\qquad f(e, g(S - \{e\})).
\end{array}
$$

We call this a *reduction (Reduktion)* of S by f with base b. If f is commutative and associative, the result of a reduction is uniquely defined.

Example Let $S := \{1, 2, 3\}$. Then the reduction of S by $+$ with base 0 is

$$6 = (1 + (2 + 3)) + 0 = (2 + (1 + 3)) + 0 = (2 + 1) + (3 + 0).$$

The reduction of S by $*$ with base 1 is

$$6 = (1 * (2 * 3)) * 1 = (2 * (1 * 3)) * 1 = ((1 * 1) * 2) * 3.$$

The Logic Evaluator provides the function

`reduce(`f`, `S`, `b`)`

which returns the reduction of S by f with base b which is more efficiently evaluated than a corresponding recursive function defined by the user (compare `sum` and `sum'` in the example below).

```
option silent = true;fun sum(S) = reduce(+, S, 0);
fun product(S) = reduce(*, S, 1);
fun S(n) = set(x in nat(1, n): true, x);
term S(10);
> {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}.
term sum(S(10));
> 55.
term product(S(10));
> 3628800.

fun +1(x, y) = +(1, y);
fun #(S)  = reduce(+1, S, 0);
term #(S(10));
> 10.

fun sum'(S) recursive #(S) =
  if(=(S, {}), 0,
    let(e = such(x in S: true, x):
      +(e, sum'(set(x in S: not(=(x, e)), x))));
term sum'(S(10));
> 55.
```

```
term sum'(S(100));
```

With the help of this operator also set union and powerset can be implemented as follows:

```
option silent = true;
fun ++(A: Set, B: Set) =
  reduce(join, A, B);
fun combine(e, S: Set) =
  ++(S, set(x in S: true, join(e, x)));
fun Powerset(S: Set) =
  reduce(combine, S, join({}, {}));
fun S(m, n) = set(x in nat(m, n): true, x);
term ++(S(1, 10), S(5, 15));
> {5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 1, 2, 3, 4}.
term Powerset(S(1, 4));
> {{1, 2, 3, 4}, {2, 3, 4}, {1, 3, 4}, {3, 4}, {1, 2, 4}, {2, 4},
{1, 4}, {4}, {1, 2, 3}, {2, 3}, {1, 3}, {3}, {1, 2}, {2}, {1}, {}}.
```

```
term combine(1, set(x in Powerset(S(2, 4)): true, x));
```

A.6 Evaluating Definitions

With the help of the notions *theory* (page 281) and *definition* (page 282), we can state very precisely what "evaluating a term" or "computing the value of a term" or just "computing" means.

Definition 147 (Computing) We are given a theory with constants C and a set of definitions that extend the theory by additional constants C'. Let T' be a term constructed from the constants in $C \cup C'$.

The problem of *computing (berechnen)* the value of T' is to find a term T that is constructed from the constants in C such that from the axioms of the extended theory

$$T = T'$$

can be derived.

"Computing" the value of a term is therefore a reduction of the notions of the extended theory to the notions of the core theory. If the definitions are given in particular "constructive" forms, the evaluation may proceed automatically by the application of certain substitution rules.

Example

- Take the theory of natural numbers with function constants 0 and $'$. We define a new function constant

$$x + y := \textbf{if } y = 0 \textbf{ then } x \textbf{ else } (x + y^{-})'$$

The value of the term $((0''' + (0' + 0')) + 0'')$ is $0'''''''$.

- Take the theory of lists with function constants "nil" and "cons". We define new function constants

$$\begin{aligned}
&\text{first}(x) := \textbf{such } a : \exists b : x = \text{cons}(a, b);\\
&\text{rest}(x) := \textbf{such } b : \exists a : x = \text{cons}(a, b);\\
&\text{append}(x, y) := \textbf{if } x = \text{nil} \textbf{ then } y\\
&\quad \textbf{else } \text{cons}(\text{first}(x), \text{append}(\text{rest}(x), y)).
\end{aligned}$$

The value of "append(rest(cons(1, cons(2, nil))), cons(3, cons(4, nil)))" is "cons(2, cons(3, cons(4, nil)))".

Appendix B

Proving Propositions

A proposition is a formula that is claimed to be true. However, a claim is only as good as the substantiating argument is. Since different people may disagree on the quality of an argument, we need some objective criteria whether an argument is correct or not. Surprisingly, there exist a number of *structural* rules such that any proposition is true that has a correspondingly formed argument. We call such an argument a *proof (Beweis)*.

It is always possible to decide whether a given argument represents a proof or not; once a proof is given, there is therefore no more dispute about the validity of a proposition. Proving is thus at the heart of any critical discourse and the knowledge about the correct application of proof rules is a must for a scientist or engineer: if we derive new knowledge, we must be able to proof it; if we are given some knowledge, we must able to check it (i.e., its proof).

To *invent* a proof is a creative activity. The proof rules give some guiding principles that help in the first steps; however, from a certain point on, some insight is required to discover the "killer argument" that "slays" the proposition. To *check* a proof is simply craft; everybody who understands "proof terminology" is able to read a proof and judge its correctness.

This chapter is dedicated to the demonstration of formal proof rules and the various forms of their appearance. The application of these rules is demonstrated by the various proofs given throughout this document.

B.1 Proof Levels

A proof can be given on different levels of detail; on the lowest level, the individual reasoning steps are so small that their correct application can be even automatically verified by a computer. However, such proofs become very large and are rarely written by humans.

On a higher level, the reasoning steps are larger and less self-evident; still the *structure* of a (less detailed) high-level proof matches the structure of the (more detailed) low-level proof as it is determined by the proof rules. If some opponent questions our high-level proof, we can correspondingly *refine* it by decomposing a large reasoning step into some smaller steps. This process can be repeated until we reach the lowest level (which is automatically checkable) or our opponent is satisfied. Even if we are not writing all details in a high-level proof, the skeleton provided by the proof rules help us to maintain its power of persuasion and justify our confidence in its correctness. This is the kind of proofs that humans typically write in scientific works.

To choose the right size of reasoning steps is a tradeoff between demonstrating the "key ideas" of a proof (by skipping details that the reader is supposed to be able to fill in herself) and giving convincing arguments (by providing sufficient details that enable the reader to understand the line of reasoning). However, while we may easily bore our audience with too many details, we may even more easily lose its understanding by giving too few. Also for the sake of correctness, it is better to elaborate more details than we may be originally inclined: many textbook proofs are wrong[1], because the author tried to save some work: she used reasoning steps that were so large that she could ultimately not grasp them herself, or she took wrong shortcuts in the application of proof rules.

If a proof is not understood, the proof uses too large reasoning steps, has bad style, or is simply wrong (the boundaries are fluid). This is in general not the fault of the audience; the presenter has the duty to adapt her presentation to the available knowledge, to make a clear argument and to refine it down to a satisfying level of detail. Science is not theology: no one is obliged to believe our claim unless we give a compelling argument.

B.2 Preliminaries

The proof of a proposition is always relative to given knowledge. Initially, this is knowledge that characterizes the considered domain (*axioms*), that is derived from a "harmless" extension (*definitions*), that is true in any domain (*tautologies*) or that has been derived by another proof (*propositions*). In the course of a proof, new knowledge is gradually added (*assumptions*).

[1]The corresponding propositions may be nevertheless true.

A *proof situation* consists of the available knowledge K (a set of formulas) and the goal G to be proved (a formula). We represent such a situation graphically as

$$\begin{array}{|c|}\hline K \\ \hline G \\ \hline \end{array}$$

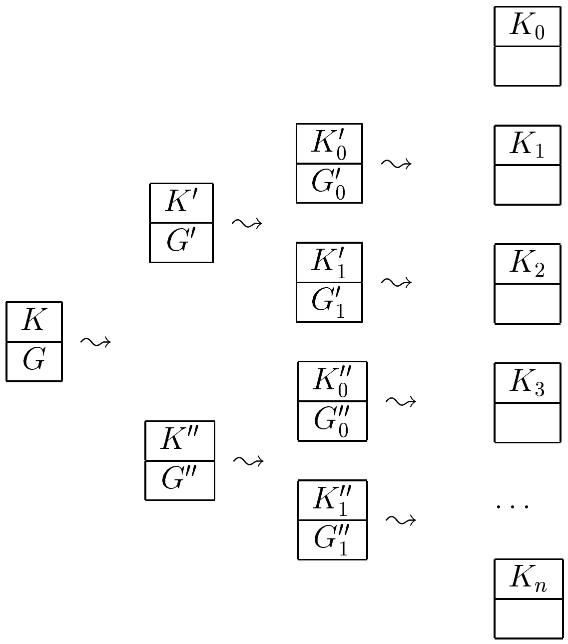
and read this as "we (have to) prove G with knowledge K". In a natural language proof, the knowledge that is available in a particular proof situation has to be deduced from the context: it consists of the knowledge at the beginning of the proof extended by all the temporary definitions and assumptions that were made in the branch of the proof that led to the current situation.

A *proof rule* reduces a proof situation to one or more other situations. We denote this reduction by $\rightsquigarrow$ and read

$$\begin{array}{|c|}\hline K_0 \\ \hline G_0 \\ \hline \end{array} \rightsquigarrow \begin{array}{|c|}\hline K_1 \\ \hline G_1 \\ \hline \end{array}$$

as "in order to prove G_0 with knowledge K_0 it suffices to prove G_1 with knowledge K_1".

A proof is the reduction of the start situation to other situations that are again reduced to other situations until we have only situations in which nothing is left to be proved:



The only way to reach a leaf in this *proof tree* (i.e., to complete a particular proof branch) is by the following rule.

Proposition 139 (Proof Completion) For proving with knowledge $K \cup \{G\}$ the goal G, nothing has to be done any more.

$$\begin{array}{|c|}\hline K \cup \{G\} \\ \hline G \\ \hline \end{array} \rightsquigarrow \begin{array}{|c|}\hline K \cup \{G\} \\ \hline \\ \hline \end{array}$$

In other words, if the goal is in our "knowledge base", we are done.

B.3 General Strategies

Direct Proof Given some knowledge K, the usual situation is that we want to find out whether a formula G is true or not, i.e., we *don't know* the truth of G in advance. Therefore we have two possibilities of a *direct proof*:

1. We try $\begin{array}{|c|}\hline K \\ \hline G \\ \hline \end{array}$. If we are successful, then G holds.
2. We try $\begin{array}{|c|}\hline K \\ \hline \neg G \\ \hline \end{array}$. If we are successful, then $\neg G$ holds.

If the first proof does not work out, it may give sufficient insight in order to establish the second proof. If we can show $\neg G$, we may have gained additional insight that enables us to transform G into some G' whose validity we can actually prove.

Indirect Proof If we are not successful with direct attempts, we may try an *indirect proof* where the goal is to derive a *contradiction*.

Proposition 140 (Proof by Contradiction) For proving with knowledge K the goal G, it suffices to prove F(alse) with additional knowledge $\neg G$.

$$\begin{array}{|c|}\hline K \\ \hline G \\ \hline \end{array} \rightsquigarrow \begin{array}{|c|}\hline K \cup \{\neg G\} \\ \hline \mathrm{F(alse)} \\ \hline \end{array}$$

In natural language, a proof of G by contradiction is usually indicated as

We assume $\neg G$ and show a contradiction.

A contradiction is usually derived by establishing a proof situation

$$\begin{array}{|c|}\hline K \cup \{G, \neg G\} \\ \hline \mathrm{F(alse)} \\ \hline \end{array}$$

because we then immediately have

$$\begin{array}{|c|}\hline K \cup \{G, \neg G\} \\ \hline \mathrm{F(alse)} \\ \hline \end{array} \rightsquigarrow \begin{array}{|c|}\hline K \cup \{\mathrm{F(alse)}\} \\ \hline \mathrm{F(alse)} \\ \hline \end{array} \rightsquigarrow \begin{array}{|c|}\hline K \cup \{\mathrm{F(alse)}\} \\ \hline \\ \hline \end{array}.$$

Examples of such proofs are given in the proof of the Peano laws from the set-theoretic construction of the natural numbers (page 100) or the proof that the square root of 2 is not a rational number (page 118).

Again, since we don't know the truth of G for sure, we generally we have two possibilities:

1. We try $\begin{array}{|c|}\hline K \cup \{\neg G\} \\ \hline \mathrm{F(alse)} \\ \hline \end{array}$. If we are successful, then G holds.
2. We try $\begin{array}{|c|}\hline K \cup \{G\} \\ \hline \mathrm{F(alse)} \\ \hline \end{array}$. If we are successful, then $\neg G$ holds.

If all attempts have failed, we have hopefully gained more insight to start a new round of attempts with refined ideas.

Proof Directions When pursuing a proof, we may proceed in two ways:

1. **Top-Down** by decomposing the goal into simpler formulas with corresponding subproofs.

   $$\begin{array}{|c|}\hline K \\ \hline G \\ \hline \end{array} \rightsquigarrow \begin{array}{|c|}\hline K_0 \\ \hline G_0 \\ \hline \end{array} \ldots \begin{array}{|c|}\hline K_{n-1} \\ \hline G_{n-1} \\ \hline \end{array}$$

2. **Bottom-Up** by deriving new knowledge from the given knowledge such that the goal ultimately becomes part of the knowledge.

   $$\begin{array}{|c|}\hline K \\ \hline G \\ \hline \end{array} \rightsquigarrow \begin{array}{|c|}\hline K \cup \{F\} \\ \hline G \\ \hline \end{array}$$

In the course of a proof we mix top-down steps with bottom-up steps; usually we begin with the top-down strategy.

B.4 Decomposing the Goal

In a proof situation $\begin{array}{|c|}\hline K\\\hline G\\\hline\end{array}$ where the goal G is not an atomic formula, the usual strategy is to decompose the proof into subproofs in a way that is determined by the outermost quantifier respectively connective of G.

Proposition 141 (Decomposition of Universal Quantifications) For proving with knowledge K the goal $\forall x : G$, it suffices to prove $G[x \leftarrow a]$ where a is an object constant that does not appear in K and not in G.

$$\begin{array}{|c|}\hline K\\\hline \forall x : G\\\hline\end{array} \rightsquigarrow \begin{array}{|c|}\hline K\\\hline G[x \leftarrow a]\\\hline\end{array} \; (a \text{ not in } K \cup \{G\})$$

In the proof of a universal formula we thus replace the variable by a constant about which no assumptions have been made yet. This may be indicated in a natural language proof in a very verbose form as

> We have to prove $(\forall x : G)$. We take an arbitrary (but fixed) constant a and show $G[x \leftarrow a]$.

Usually, we choose instead of a some name that indicates the name of the corresponding variable, e.g., $\overline{x}$. Even more frequently, we simply chose x as the name of the constant provided that x *does not appear freely* in K or G. We then simply write

> We have to prove $(\forall x : G)$. We take arbitrary x and show G.

or even shorter

> We have to prove $(\forall x : G)$. Take arbitrary x. Then ... (proof of G).

However, we have to understand that x now represents an arbitrary (but fixed) constant. Allmost all proofs in this document use this style, see for instance the proof of Proposition 32 on page 90 or the proof of Proposition 44 on page 112.

Frequently also the *indirect method* is applied in such a situation. Instead of proving $(\forall x : G)$ we assume $(\neg \forall x : G)$, i.e., $(\exists x : \neg G)$ and proceed to derive a contradiction with the help of a constant a such that G holds (see Proposition 151 on page 321):

$$\begin{array}{|c|}\hline K \\ \hline \forall x : G \\ \hline \end{array} \leadsto \begin{array}{|c|}\hline K \cup \{\exists x : \neg G\} \\ \hline \text{F(alse)} \\ \hline \end{array} \leadsto \begin{array}{|c|}\hline K \cup \{\neg G[x \leftarrow a]\} \\ \hline \text{F(alse)} \\ \hline \end{array}$$

This is typically indicated as

> We have to prove $(\forall x : G)$. Assume $\neg G$ for some x. Then ... (derivation of a contradiction with additional knowledge $\neg G$).

In this formulation, x now represents a constant for the rest of the proof.

Proposition 142 (Decomposition of Existential Quantifications) For proving with knowledge K the goal $\exists x : G$, it suffices to prove $G[x \leftarrow T]$ for some term T.

$$\begin{array}{|c|}\hline K \\ \hline \exists x : G \\ \hline \end{array} \leadsto \begin{array}{|c|}\hline K \\ \hline G[x \leftarrow T] \\ \hline \end{array}$$

An existential proof requires creativity. We have to find a *witness (Zeugen)*, i.e., a value for x that makes G true. This value can be only constructed from those function and object constants about which we have some knowledge in K such that we can proceed to prove G.

In natural language, such a proof is usually indicated as

> We have to prove $(\exists x : G)$. We prove $G[x \leftarrow T]$.

or, if x occurs multiple times in G, shorter as

> We have to prove $(\exists x : G)$. Take $a := T$. We prove $G[x \leftarrow a]$.

where a is an object constant that does not appear in K and for which we assume the additional knowledge $a = T$. Even shorter, we may write

> We have to prove $(\exists x : G)$. Take $x := T$. We then have …(proof of G with additional knowledge $x = T$).

This is possible if x *does not occur freely in* K.

For examples, see the proof of Proposition 49 on page 118 or the proof of the existence of a factorization into prime numbers on page 134.

The *indirect method* may be applied by assuming $(\neg\exists x : G)$, i.e., $(\forall x : \neg G)$ and deriving a contradiction, see Proposition 150 on page 320:

$$\begin{array}{|c|}\hline K \\ \hline \exists x : G \\ \hline\end{array} \rightsquigarrow \begin{array}{|c|}\hline K \cup \{\forall x : \neg G\} \\ \hline \mathrm{F(alse)} \\ \hline\end{array}.$$

This is usually indicated as follows:

> We have to prove $(\exists x : G)$. Assume $(\forall x : \neg G)$. Then … (derivation of a contradiction with additional knowledge $(\forall x : \neg G)$).

Proposition 143 (Decomposition of Equivalences) For proving with knowledge K the goal $G_0 \Leftrightarrow G_1$, it suffices to prove both $G_0 \Rightarrow G_1$ and $G_1 \Rightarrow G_0$:

$$\begin{array}{|c|}\hline K \\ \hline G_0 \Leftrightarrow G_1 \\ \hline\end{array} \rightsquigarrow \begin{array}{|c|}\hline K \\ \hline G_0 \Rightarrow G_1 \\ \hline\end{array}\ \begin{array}{|c|}\hline K \\ \hline G_1 \Rightarrow G_0 \\ \hline\end{array}$$

An equivalence is shown by proving the implication "from left to right" and "from right to left", sometimes denoted as

> We have to prove $G_0 \Leftrightarrow G_1$:
>
> - $\Rightarrow$: …(proof of $G_0 \Rightarrow G_1$).
> - $\Leftarrow$: …(proof of $G_1 \Rightarrow G_0$).

If we have to show $G_0 \Leftrightarrow G_1 \Leftrightarrow G_2$, i.e., $(G_0 \Leftrightarrow G_1) \wedge (G_1 \Leftrightarrow G_0)$, it suffices to "traverse the circle", i.e., to show $(G_0 \Rightarrow G_1) \wedge (G_1 \Rightarrow G_2) \wedge (G_2 \Rightarrow G_0)$ (and analogously for an arbitrary number of equivalences).

Examples are shown in the proofs of Proposition 22 on page 64 and of Proposition 30 on page 84.

Proposition 144 (Decomposition of Implications) For proving with knowledge K the goal $G_0 \Rightarrow G_1$, it suffices to prove G_1 with additional knowledge G_0:

$$\begin{array}{|c|}\hline K \\ \hline G_0 \Rightarrow G_1 \\ \hline\end{array} \rightsquigarrow \begin{array}{|c|}\hline K \cup \{G_0\} \\ \hline G_1 \\ \hline\end{array}$$

Almost every proof contains applications of above rule indicated as

> We have to show $G_0 \Rightarrow G_1$. Assume G_0. Then ... (proof of G_1 with additional knowledge G_0).

Because of $(G_0 \Rightarrow G_1)$ iff $(\neg G_1 \Rightarrow \neg G_0)$, an alternative is to apply the rule

$$\begin{array}{|c|}\hline K \\ \hline G_0 \Rightarrow G_1 \\ \hline\end{array} \rightsquigarrow \begin{array}{|c|}\hline K \cup \{\neg G_1\} \\ \hline \neg G_0 \\ \hline\end{array}$$

indicated as

> We have to show $G_0 \Rightarrow G_1$. Assume $\neg G_1$. Then ... (proof of $\neg G_0$ with additional knowledge $\neg G_1$).

Because of $\neg(G_0 \Rightarrow G_1)$ iff $(G_0 \wedge \neg G_1)$, the indirect method yields

$$\begin{array}{|c|}\hline K \\ \hline G_0 \Rightarrow G_1 \\ \hline\end{array} \rightsquigarrow \begin{array}{|c|}\hline K \cup \{G_0 \wedge \neg G_1\} \\ \hline \text{F(alse)} \\ \hline\end{array}$$

which is typically indicated as

> We have to show $G_0 \Rightarrow G_1$. Assume $G_0 \wedge \neg G_1$. Then we have ... (derivation of a contradiction).

Examples are given in the proofs of Proposition 22 on page 64 and of the Second Peano Law on page 100.

Proposition 145 (Decomposition of Conjunctions) For proving with knowledge K the goal $G_0 \wedge G_1$, it suffices to prove both G_0 and G_1:

$$\begin{array}{|c|}\hline K \\ \hline G_0 \wedge G_1 \\ \hline \end{array} \rightsquigarrow \begin{array}{|c|}\hline K \\ \hline G_0 \\ \hline \end{array} \begin{array}{|c|}\hline K \\ \hline G_1 \\ \hline \end{array}$$

A conjunction is shown by showing both conjuncts in turn:

> We have to show $G_0 \wedge G_1$.
>
> 1. …(proof of G_0).
> 2. …(proof of G_1).

Because of $\neg(G_0 \wedge G_1)$ iff $\neg G_0 \vee \neg G_1$, the *indirect method* leads to

$$\begin{array}{|c|}\hline K \\ \hline G_0 \wedge G_1 \\ \hline \end{array} \rightsquigarrow \begin{array}{|c|}\hline K \cup \{\neg G_0 \vee \neg G_1\} \\ \hline \mathrm{F(alse)} \\ \hline \end{array} \rightsquigarrow \begin{array}{|c|}\hline K \cup \neg G_0 \\ \hline \mathrm{F(alse)} \\ \hline \end{array} \begin{array}{|c|}\hline K \cup \neg G_1 \\ \hline \mathrm{F(alse)} \\ \hline \end{array}$$

applying the technique of "case distinction" explained on page 319:

> We have to prove $G_0 \wedge G_1$.
>
> 1. Assume $\neg G_0$. Then …(derivation of contradiction with additional assumption $\neg G_0$).
> 2. Assume $\neg G_1$. Then …(derivation of contradiction with additional assumption $\neg G_1$).

Examples of proofs of conjunctions are given for Proposition 32 on page 90 for the existence of a factorization into prime numbers on page 134.

Proposition 146 (Decomposition of Disjunctions) For proving with knowledge K the goal $G_0 \vee G_1$, it suffices to prove G_1 with additional knowledge $\neg G_0$.

$$\begin{array}{|c|}\hline K \\ \hline G_0 \vee G_1 \\ \hline \end{array} \rightsquigarrow \begin{array}{|c|}\hline K \cup \neg G_0 \\ \hline G_1 \\ \hline \end{array}$$

This rule is a consequence of "$(G_0 \vee G_1)$ iff $(\neg G_0 \Rightarrow G_1)$" such that the same techniques can be applied as for the decomposition of implications (see Proposition 144):

> We have to show $G_0 \vee G_1$. Assume $\neg G_0$. Then … (proof of G_1).

Of course, the roles of G_0 and G_1 can be inverted.

Frequently, in a particular proof situation, the additional assumption is not required and one simply says

> We have to show $G_0 \vee G_1$. We have … (proof of G_0 or of G_1).

Proposition 147 (Explicitly Defined Predicates) For proving an atomic formula $p(a_0, \ldots, a_{n-1})$ where p is predicate explicitly defined as

$$p(x_0, \ldots, x_{n-1}) :\Leftrightarrow G,$$

and $a_0, \ldots, a_{n-1}$ are terms, it suffices to prove $G[x_0 \leftarrow a_0, \ldots, x_{n-1} \leftarrow a_{n-1}]$:

$$\begin{array}{|c|}\hline K \cup \{\forall x_0, \ldots, x_{n-1} : p(x_0, \ldots, x_{n-1}) \Leftrightarrow G\} \\ \hline p(a_0, \ldots, a_{n-1}) \\ \hline \end{array} \rightsquigarrow$$

$$\begin{array}{|c|}\hline K \cup \{\forall x_0, \ldots, x_{n-1} : p(x_0, \ldots, x_{n-1}) \Leftrightarrow G\} \\ \hline G[x_0 \leftarrow a_0, \ldots, x_{n-1} \leftarrow a_{n-1}] \\ \hline \end{array}$$

We may thus just insert the definition of a predicate into a goal formula:

> We have to prove $p(a_0, \ldots, a_{n-1})$. By definition of p, it suffices to prove $G[x_0 \leftarrow a_0, \ldots, x_{n-1} \leftarrow a_{n-1}]$.

The proofs of Proposition 22 on page 64 and of Proposition 32 on page 90 show applications of above rule.

Proposition 148 (Explicitly Defined Functions) For proving the goal $F[x \leftarrow G(a_0, \ldots, a_{n-1})]$ where f is a function explicitly defined as

$$G(x_0, \ldots, x_{n-1}) := T,$$

it suffices to prove $G[x \leftarrow T[x_0 \leftarrow a_0, \ldots, x_{n-1} \leftarrow a_{n-1}]]$ (for some terms $T, a_0, \ldots, a_{n-1}$):

$$\begin{array}{|c|}\hline K \cup \{\forall x_0, \ldots, x_{n-1} : G(x_0, \ldots, x_{n-1}) = T\} \\ \hline G[x \leftarrow G(a_0, \ldots, a_{n-1})] \\ \hline \end{array} \rightsquigarrow$$

$$\begin{array}{|c|}\hline K \cup \{\forall x_0, \ldots, x_{n-1} : G(x_0, \ldots, x_{n-1}) = T\} \\ \hline G[x \leftarrow T[x_0 \leftarrow a_0, \ldots, x_{n-1} \leftarrow a_{n-1}]] \\ \hline \end{array}$$

Above rule simply tells us that, if a goal contains an elementary term with an explicitly defined function, we are allowed to "insert the definition" of the function:

> We have to prove $F[x \leftarrow G(a_0, \ldots, a_{n-1})]$. By definition of G, it suffices to prove $G[x \leftarrow T[x_0 \leftarrow a_0, \ldots, x_{n-1} \leftarrow a_{n-1}]]$.

Examples are given by the proofs of Proposition 32 on page 90 and of Proposition 47 on page 117.

B.5 Deriving New Knowledge

At some point (at least if the goal is an atomic formula with a predicate of the underlying theory), we have to start to operate with the available knowledge in order to derive a situation in which the goal is part of the knowledge.

Proposition 149 (Proof by Case Distinction) For proving with knowledge K the goal G, it suffices to prove G with additional knowledge F and to prove G with additional knowledge $\neg F$ (for some formula F).

$$\begin{array}{|c|}\hline K \\ \hline G \\ \hline \end{array} \rightsquigarrow \begin{array}{|c|}\hline K \cup \{F\} \\ \hline G \\ \hline \end{array} \begin{array}{|c|}\hline K \cup \{\neg F\} \\ \hline G \\ \hline \end{array}$$

A proof by case distinction decomposes the "universe of situations" into those where a particular assumption F holds and those where it does not hold:

> We have to prove G.

> 1. Assume F. Then ... (proof of G with additional knowledge F).
> 2. Assume $\neg F$. Then ... (proof of G with additional knowledge $\neg F$).

Frequently a case distinction is introduced in situations where we have a formula $(F_0 \vee \ldots \vee \ldots F_{n-1})$ in our knowledge:

$$\begin{array}{|c|}\hline K \cup \{F_0 \vee \ldots \vee F_{n-1}\} \\ \hline G \\ \hline \end{array} \rightsquigarrow \begin{array}{|c|}\hline K \cup \{F_0\} \\ \hline G \\ \hline \end{array} \ldots \begin{array}{|c|}\hline K \cup \{F_{n-1}\} \\ \hline G \\ \hline \end{array}$$

which is indicated as

> We have to prove G. Since we know $(F_0 \vee \ldots \vee F_{n-1})$, it suffices to consider the following cases:
>
> - Case F_0: ... (proof of G with additional knowledge F_0).
> - ...
> - Case F_{n-1}: ... (proof of G with additional knowledge F_{n-1}).

An example of a proof with case distinction is the verification of Euclid's Algorithm on page 154.

Proposition 150 (Universal Quantification in Knowledge) For proving with knowledge $K \cup \{\forall x : F\}$ the goal G, it suffices to prove G with additional knowledge $F[x \leftarrow T]$ for any term T:

$$\begin{array}{|c|}\hline K \cup \{\forall x : F\} \\ \hline G \\ \hline \end{array} \rightsquigarrow \begin{array}{|c|}\hline K \cup \{\forall x : F, F[x \leftarrow T]\} \\ \hline G \\ \hline \end{array}$$

A universal formula $(\forall x : F)$ in the knowledge base is a "machine" that takes any T and prodcues additional knowledge $F[x \leftarrow T]$. Whenever we are in need of an arbitrary instance of F in our knowledge base, we can start this machine:

> We have to prove G. Since we know $(\forall x : F)$, we have $F[x \leftarrow T]$ and thus ... (proof of G with additional knowledge $F[x \leftarrow T]$).

Applications of this rule are shown in the proof of Proposition 22 on page 64.

Proposition 151 (Existential Quantification in Knowledge) For proving with knowledge $K \cup \{\exists x : F\}$ the goal G, it suffices to prove G with additional knowledge $F[x \leftarrow a]$ for some object constant a that does not appear in K, G, or F:

$$\begin{array}{|c|}\hline K \cup \{\exists x : F\} \\ \hline G \\ \hline\end{array} \rightsquigarrow \begin{array}{|c|}\hline K \cup \{\exists x : F, F[x \leftarrow a]\} \\ \hline G \\ \hline\end{array} (a \text{ not in } \; K \cup \{G, F\})$$

An existential formula $(\exists x : F)$ in the knowledge base is an "engine" which returns a constant a about the only thing we know is $F[x \leftarrow a]$. Whenever we are in need of such a constant, we can invoke the machine:

> We have to prove G. Since we know $(\exists x : F)$, we have have some a with $F[x \leftarrow a]$. Thus ... (proof of G with additional knowledge $F[x \leftarrow a]$ for some new constant a).

The application of this rule is demonstrated in the proof of Proposition 32 on page 90 and the proof that the square root of 2 is not rational on page 4.4.

Proposition 152 (Additional Knowledge) For proving with knowledge K the goal G, it suffices to prove G with additional knowledge F, if F holds in every domain in which (some of) the formulas in K holds.

$$\begin{array}{|c|}\hline K \\ \hline G \\ \hline\end{array} \rightsquigarrow \begin{array}{|c|}\hline K \cup F \\ \hline G \\ \hline\end{array} (F \text{ holds in every domain in which } K \text{ holds}).$$

This last rule is a "placeholder" for a number of ways to infer

$$\begin{array}{|c|}\hline K \\ \hline F \\ \hline\end{array}$$

(respectively $\begin{array}{|c|}\hline S \\ \hline F \\ \hline\end{array}$ for some $S \subseteq K$):

1. This has been shown in a previous proof or is shown as a subproof.
2. This holds because F is a *propositional consequence* of K, i.e., the conclusion holds independently of the truth values of the atomic formulas and quantified formulas contained in K and F.
3. This is an instance of some *quantifier consequence* wich give true conclusions in every domain.
4. This is derived by applying *substitution* rules from known equalities and equivalences.

In the first way, we can decompose a proof in a *modular* way into a number of smaller proofs or reuse the knowledge represented by previously proved propositions. The other ways are explained in the following subsections.

B.5.1 Propositional Consequences

The conclusion

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

holds independently of the interpretation of p and consequently independently of the truth value of $(\forall x : \exists y : p(x, y))$. This is because this conclusion is an instance of the pattern

$\neg\neg G$
G

which is true independently of the truth value of G. We call such a conclusion a *propositional consequence (aussagenlogische Folge).*

Proposition 153 (Propositional Consequences) The following conclusions are propositional consequences for every formula A and B:

Negation

$\neg\neg A$
A

A
$\neg\neg A$

And Introduction and Or Elimination

$$\boxed{\dfrac{A \land B}{A}} \quad \boxed{\dfrac{A}{A \lor B}}$$

De Morgan

$$\boxed{\dfrac{\neg(A \land B)}{\neg A \lor \neg B}} \quad \boxed{\dfrac{\neg(A \lor B)}{\neg A \land \neg B}} \quad \boxed{\dfrac{\neg A \lor \neg B}{\neg(A \land B)}} \quad \boxed{\dfrac{\neg A \land \neg B}{\neg(A \lor B)}}$$

Modus Ponens

$$\boxed{\dfrac{A, A \Rightarrow B}{B}}$$

Contraposition

$$\boxed{\dfrac{A \Rightarrow B}{\neg B \Rightarrow \neg A}} \quad \boxed{\dfrac{\neg A \Rightarrow \neg B}{B \Rightarrow A}} \quad \boxed{\dfrac{A \Leftrightarrow B}{\neg A \Leftrightarrow \neg B}} \quad \boxed{\dfrac{\neg A \Leftrightarrow \neg B}{A \Leftrightarrow B}}$$

There are (infinitely) many other propositional consequences. A general strategy to show that $\boxed{\dfrac{A}{B}}$ is a propositional consequence is to show that $A \Rightarrow B$ is a propositional tautology.

Proposition 154 (Tautology) A propositional formula with variables is a *propositional tautology (aussagenlogische Tautologie)* if it is true for every assignment of truth values to the variables.

We can show that a propositional formula with variables is a tautology by constructing a truth table (see page 15) or by applying the *indirect method.*

Example We show that

$$((A \lor B) \land (A \Rightarrow C) \land (B \Rightarrow C)) \Rightarrow C.$$

is a tautology by assuming that its truth value is false and then deriving a contradiction:

$$\overset{\text{false}}{\overline{\overset{\text{true}}{\overline{(\overset{\text{true}}{\overline{(A \lor B)}} \land (\overset{\text{true}}{\overline{\overset{\text{false}}{\overline{A}} \Rightarrow \overset{\text{false}}{\overline{C}}}}) \land (\overset{\text{true}}{\overline{\overset{\text{false}}{\overline{B}} \Rightarrow \overset{\text{false}}{\overline{C}}}}))}} \Rightarrow \overset{\text{false}}{\overline{C}}}}$$

Because the implication is false, C is false and the conjuncts are true. Thus A and B must be false (since C is false, the implications cannot be true otherwise). Therefore $A \vee B$ is false, which contradicts above derivation.

B.5.2 Quantifier Consequences

There are various patterns of conclusions that occur in proofs so frequently that they are considered as "basic knowledge".

Proposition 155 (Quantifier Consequences) For every formula A and B, the following conclusions hold:

Universal Quantification and Conjunction

$$\frac{(\forall x : A \wedge B)}{(\forall x : A) \wedge (\forall x : B)} \qquad \frac{(\forall x : A) \wedge (\forall x : B)}{(\forall x : A \wedge B)}$$

Existential Quantification and Disjunction

$$\frac{(\exists x : A \vee B)}{(\exists x : A) \vee (\exists x : B)} \qquad \frac{(\exists x : A) \vee (\exists x : B)}{(\exists x : A \vee B)}$$

Universal and Disjunction, Existential and Conjunction

$$\frac{(\forall x : A) \vee (\forall x : B)}{(\forall x : A \vee B)} \qquad \frac{(\exists x : A) \wedge (\exists x : B)}{(\exists x : A \wedge B)}$$

Universal and Existential Quantification

$$\frac{\exists x : \forall y : A}{\forall y : \exists x : A}$$

De Morgan Laws

$$\frac{\neg \forall x : A}{\exists x : \neg A} \qquad \frac{\exists x : \neg A}{\neg \forall x : A} \qquad \frac{\neg \exists x : A}{\forall x : \neg A} \qquad \frac{\forall x : \neg A}{\neg \exists x : A}$$

Such Quantifier

$$\frac{\exists x : A}{A[x \leftarrow \textbf{such } x : A]}$$

$$\frac{(\forall y_0, y_1 : (A[x \leftarrow y_0] \wedge A[x \leftarrow y_1]) \Rightarrow y_0 = y_1)}{(\forall x : A \Rightarrow x = \textbf{such } x : A)}$$

The last two facts about the such quantifier tell us that

1. there must exist some x such that A holds before we are allowed to conclude that $(\textbf{such}\ x : A)$ satisfies A;
2. to prove that some y with $A[x \leftarrow y]$ equals $(\textbf{such}\ x : A)$, we also have to show that only one such y exists.

These conclusions can be proved by the proving techniques explained before; an example is given below.

Proof We show for arbitrary formula A

$$(\neg\forall x : A) \Rightarrow (\exists x : \neg A)$$

by showing (contraposition)

$$(\neg\exists x : \neg A) \Rightarrow (\neg\neg\forall x : A)$$

i.e. (propositional consequence and substitution, see next subsection)

$$(\neg\exists x : \neg A) \Rightarrow (\forall x : A).$$

We assume

$$(*)\ \neg\exists x : \neg A$$

and show $\forall x : A$. Take arbitrary and assume $\neg A$. Then we have $(\exists x : \neg A)$ which contradicts (*).

B.5.3 Substitutions

The equality axioms introduced on page 38 and the properties of logical equivalence allow us to draw the following conclusions.

Proposition 157 (Substitutions) For all terms S and T, formulas A and B, variables x and formula patterns C with variable F, the following holds:

Equality Subsitutions

$S = T \wedge A[x \leftarrow S]$
$A[x \leftarrow T]$

Equivalence Substitutions

$A \Leftrightarrow B \wedge C[F \leftarrow A]$
$C[F \leftarrow B]$

The substitution laws allow us to replace in a goal "terms by equal terms" and "formulas by equivalent formulas". These laws are applied when we say

> We have to prove $F[x \leftarrow T]$. Because $T{=}U$, it suffices to prove $F[x \leftarrow U]$.

and thus reduce a problem to an equivalent subproblem.

B.6 Example

We conclude this chapter by demonstrating in detail the proof of a proposition that is used in the verification of Euclid's algorithm on page 154. First we repeat the definitions

$$x|y :\Leftrightarrow \exists z \in \mathbb{N} : x * z = y;$$
$$\gcd(x, y) := \textbf{such } z \in \mathbb{N} : z|x \wedge z|y \wedge (\forall g : (g|x \wedge g|y) \Rightarrow g \leq z).$$

Our goal is to prove

$$\forall m \in \mathbb{N}, n \leq m : \gcd(m, n) = \gcd(m - n, n),$$

i.e., the greatest common divisor of m and n is also the greatest common divisor of $m - n$ and n.

An argument as it can be found in a typical text book is the following.

Proof Let g be the gcd of $m-n$ and n. Since g divides $m-n$ and g divides n, g also divides $(m - n) + n = m$.

Take h such that h divides m and h divides n. Then h also divides $m - n$. Since g is the gcd of m and $m - n$, h is less than or equal g.

This "proof skeleton" captures on a very high-level the *key idea* that represents the basis of a "real" proof. However, this idea is only understandable with the knowledge of the rules introduced in the previous sections. Then it becomes also clear that the skeleton actually reflects the structure of a more detailed proof such as the one given below (with additional explanations that are not part of the proof added in *italics*).

Proof Take arbitrary $m \in \mathbb{N}$ and $n \leq m$.

We have to prove a universally quantified implication. We introduce arbitrary constants m and n and add the implication hypothesis $(m \in \mathbb{N} \wedge n \leq m)$ *to our knowledge. We now have to show the goal* $\gcd(m, n) = \gcd(m - n)$.

If $m = 0 \wedge n = 0$, we are done.

We proceed by case distinction. In the first case $m = 0 \wedge n = 0$, *we know* $\gcd(m, n) = \gcd(0, 0) = \gcd(0 - 0, 0) = \gcd(m - n)$. *By transitivity of* $=$, *we have the goal.*

Otherwise, $m \neq 0 \vee n \neq 0$. We can show that in this case there exists a unique $z \in \mathbb{N}$ with

$$z|m \wedge z|n \wedge (\forall g : (g|m \wedge g|n) \Rightarrow g \leq z).$$

The rest of the proof deals with the second case $m \neq 0 \vee n \neq 0$. *We claim that in this case above formula holds (which requires a separate proof) and use this as knowledge from now on.*

For every $z' \in \mathbb{N}$, in order to prove

$$z' = \gcd(m, n),$$

it therefore suffices to prove

$$\mathrm{z}'|m \wedge \mathrm{z}'|n \wedge (\forall g : (g|m \wedge g|n) \Rightarrow g \leq \mathrm{z}').$$

We justify a reasoning step that will be used below: we have to prove that a particular object equals $\gcd(m, n)$. *By the rule of substitution, we may insert the definition of* gcd *and thus have to prove that the object equals the value of* *a "such term". We thus remind the reader of the proof rule for this kind of terms on page 325 and instantiate the rule with the concrete formula. The application of this rule requires that only one object satisfies the formula in the "such term", which is exactly what we have added as knowledge above.*

Let $g := \gcd(m-n, n)$. By $n \leq m$, we know

$$(1)\ g|(m-n) \wedge g|n \wedge (\forall h : (h|(m-n) \wedge h|n) \Rightarrow h \leq g).$$

We define an abbreviation g for $\gcd(m-n, n)$. Since $n \leq m$, we have $m - n \in \mathbb{N}$. Since $m \neq 0 \vee n \neq 0$, also $m - n \neq 0 \vee n \neq 0$ (which has to be shown in a small subproof). By the knowledge added above, we thus have an object that satisfies the formula in the definition of $\gcd(m-n, n)$. Therefore we may apply the corresponding rule for "such terms" and assume that g satisfies this formula as well.

We have to show $g = \gcd(m, n)$, i.e., by the explanation above

$$(2)\ g|m \wedge g|n \wedge (\forall h : (h|m \wedge h|n) \Rightarrow h \leq g).$$

We apply the reasoning step explained above. Now we have to prove the conjunction by showing each conjunct in turn.

1. We show (3) $g|m$. Because of (1), we have some a and b with $ga = m-n$ and $gb = n$. Then we know

   $$g(a+b) = ga + gb = (m-n) + n = m;$$

   which implies (3).

   We insert the definition of $|$ into knowledge (1), which gives us an existential formula $(\exists a : ga = m)$ and $(\exists b : gb = n)$. The application of these existential formulas gives us new constants a and b with the corresponding properties. Then we apply equational reasoning which gives us our goal by the transitivity of $=$.

2. We know (4) $g|n$ from (1).

   The goal is part of our knowledge.

3. We show (5) $\forall h : (h|m \wedge h|n) \Rightarrow h \leq g$. Take arbitrary h and assume

   $$(6)\ h|m \wedge h|n.$$

We show (7) $h \leq g$.

We have to show a universally quantified implication. We introduce an arbitrary constant h and assume the implication hypothesis. We now have to show the new goal (7).

By (6) we have some a' and b' with $ha' = m$ and $hb' = n$.

We insert the definition of | into knowledge (6) which gives two existential formulas that we can apply to introduce the constants a' and b' with the knowledge denoted above.

Then we know

$$h(a' - b') = ha' - hb' = m - n$$

and thus (8) $h|(m - n)$.

We apply equational reasoning to yield a formula in our knowledge that is equivalent to the definition of | for arguments h and $m - n$.

With (6) and (8), (1) implies (7).

We apply the rule of "universal quantification in knowledge" to instantiate the universally quantified implication in (1) by term h. Then we apply the rule of "modus ponens" to this implication and to the conjunction of (6) and (8), which gives us the goal.

While above proof is very detailed, it still relies on the claim that the greatest common divisor is uniquely defined (which has to be shown in an extra proof). The verbosity of the proof stems from the fact that it considers details like "Is $m - n$ actually defined?" or "Is the gcd unique?" that were neglected in the high-level proof skeleton.

Whether one is willing to bother with details is a matter of whether one is actually interested in the truth of a formula *as it stands* or whether one just wants to know whether the "intention is correct" (whatever this means). For a computer scientist, exactness *is* important (a "roughly" correct program may be the cause of an airplane crash); she should not be *anxious* but *eager* to find an error in a proposition and thus also check all the details that are often neglected in typical mathematical texts.

Appendix C

Logic Evaluator Definitions

C.1 Sets

```
// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// some set-theoretic notions
//
// (c) 1999, Wolfgang Schreiner, see file COPYRIGHT
// http://www.risc.uni-linz.ac.at/software/formal
// ----------------------------------------------------------------

// singleton set
fun {}(x) =
  join(x, {});

// A is a subset of B iff every element of A is also in B
pred isSubset(A: Set, B: Set) <=>
  forall(x in A: in(x, B));

// two sets are equal if each is a subset of the other
pred equals(A: Set, B: Set) <=>
  and(isSubset(A, B), isSubset(B, A));

// the intersection of two sets
fun **(A: Set, B: Set) =
  set(x in A: in(x, B), x);

// the difference of two sets
fun --(A: Set, B: Set) =
  set(x in A: not(in(x, B)), x);

// the union of two sets
```

```
fun ++(A: Set, B: Set) =
  reduce(join, A, B);

// the union of all sets n A
fun ++(A: Set) =
  reduce(++, A, {});

// the product of two sets A and B
fun x(A: Set, B: Set) =
  set(a in A, b in B: true, tuple(a, b));

// cardinality of S is determined by iteration over the elements
fun count(e, i: Nat) = +(i, 1);
fun #(S: Set) = reduce(count, S, 0);

// the union of S and of e joined to all elements of S
fun combine(e, S: Set) =
  ++(S, set(x in S: true, join(e, x)));

// the set of all subsets of S
fun Powerset(S: Set) =
  reduce(combine, S, {}({}));

// alternative recursive definition
fun PowersetR(S: Set) recursive #(S) =
  if (=(S, {}), join({}, {}),
    let(e = such(x in S: true, x):
      combine(e, Powerset(--(S, {}(e))))));

// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// ----------------------------------------------------------------
```

C.2 Relations and Functions

```
// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// binary relations
//
// (c) 1999, Wolfgang Schreiner, see file COPYRIGHT
// http://www.risc.uni-linz.ac.at/software/formal
// ----------------------------------------------------------------

read set;

// R is a relation
pred Relation(R) <=>
  forall(x in R: and(Tuple(x), =(length(x), 2)));

// R is a relation between A and B
pred isRelation(R, A, B) <=>
  and(Relation(R), forall(x in R: and(in(.0(x), A), in(.1(x), B))));

// domain of R
fun domain(R: Relation) =
  set(x in R: true, .0(x));

// range of R
fun range(R: Relation) =
  set(x in R: true, .1(x));

// inverse of R
fun ^-1(R: Relation) =
  set(x in R: true, tuple(.1(x), .0(x)));

// composition of R and S
fun o(R: Relation, S: Relation) =
  let(A = domain(R), B = **(range(R), domain(S)), C = range(S):
    set(a in A, c in C:
      exists(b in B: and(in(tuple(a, b), R), in(tuple(b, c), S))),
      tuple(a, c)));

// f is a function
pred Function(f) <=>
  and(Relation(f),
    forall(t0 in f, t1 in f:
      let(x0 = .0(t0), y0 = .1(t0), x1 = .0(t1), y1 = .1(t1):
        implies(=(x0, x1), =(y0, y1)))));

// f is a partial function from A to B
pred isPartialFunction(f, A, B) <=>
  and(isRelation(f, A, B), Function(f));
```

```
// f is a total function from A to B
pred isFunction(f, A, B) <=>
  and(isPartialFunction(f, A, B),
    forall(x in A: exists(y in B: in(tuple(x, y), f))));

// application of f to x
fun apply(f: Function, x) =
  such(y in range(f): in(tuple(x, y), f), y);

// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// ----------------------------------------------------------------
```

C.3 Natural Numbers

```
// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// the natural numbers
//
// (c) 1999, Wolfgang Schreiner, see file COPYRIGHT
// http://www.risc.uni-linz.ac.at/software/formal
// ----------------------------------------------------------------

// we use the builtin type
pred N(x) <=> Nat(x);

// the constructors
fun N0 = 0;
fun '(x: N) = +(x, 1);

// predecessor
fun ^-(x: N) = such(n in nat(0, x): =(x, '(n)), n);

// constants
fun N1 = '(N0);
fun N2 = '(N1);

// addition, multiplication, comparison
fun +N(x: N, y: N) recursive y =
  if(=(y, N0), x, '(+N(x, ^-(y))));
fun *N(x: N, y: N) recursive y =
  if(=(y, N0), N0, +N(x, *N(x, ^-(y))));
pred <=N(x: N, y: N) recursive y <=>
  if(=(x, N0), true, if(=(y, N0), false, <=N(^-(x), ^-(y))));

// difference
fun -N(x: N, y: N) = such(z in nat(0, x): =(x, +N(z, y)), z);

// quotient and remainder
fun divN(x: N, y: N) =
  such(q in nat(0, x), r in nat(0, ^-(y)):
    =(x, +N(*N(q, y), r)), q);
fun modN(x: N, y: N) =
  such(q in nat(0, x), r in nat(0, ^-(y)):
    =(x, +N(*N(q, y), r)), r);

// exponentiation
fun ^N(x: N, n: N) recursive n =
  if(=(n, N0), N1, *N(x, ^N(x, ^-(n))));

// more notions
pred divides(x, y) <=> exists(z in nat(N0, y): =(*N(x, z), y));
```

```
fun gcd(x, y) =
  let(m = if(=(x, N0), y, x):
    such(z in nat(N0, m):
      and(divides(z, x), divides(z, y),
        forall(w in nat(+N(z, N1), m):
          or(not(divides(w, x)), not(divides(w, y))))),
      z));
fun lcm(x, y) = such(z in nat(N1, *N(x, y)):
  and(divides(x, z), divides(y, z),
    forall(w in nat(x, -(z, N1)):
      or(not(divides(x, w)), not(divides(y, w))))),
  z);
pred relprime(x, y) <=> =(gcd(x, y), N1);
pred isprime(x) <=>
  and(not(<=N(x, N1)),
    forall(y in nat(N0, x):
      implies(divides(y, x), or(=(y, N1), =(y, x)))));

// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// ----------------------------------------------------------------


// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// the natural numbers using builtin arithmetic
//
// (c) 1999, Wolfgang Schreiner, see file COPYRIGHT
// http://www.risc.uni-linz.ac.at/software/formal
// ----------------------------------------------------------------

read natural;

// redefine addition, multiplication, comparison to builtin
fun +N(x, y) = +(x, y);
fun *N(x, y) = *(x, y);
pred <=N(x, y) <=> <=(x, y);

// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// ----------------------------------------------------------------
```

C.4 Integer Numbers

```
// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// the integer numbers
//
// (c) 1999, Wolfgang Schreiner, see file COPYRIGHT
// http://www.risc.uni-linz.ac.at/software/formal
// ----------------------------------------------------------------

read set;
read natural0;

// integers are pairs of naturals
pred Z(x) <=>
  and(Tuple(x), =(2, length(x)), N(.0(x)), N(.1(x)));

// constructor function
fun I(x: N, y: N) =
  if(<=N(y, x), tuple(-N(x, y), N0), tuple(N0, -N(y, x)));

// constants
fun Z0 = I(N0, N0);
fun Z1 = I(N1, N0);
fun Z2 = I(N2, N0);

// basic arithmetic
fun +Z(x: Z, y: Z) = I(+N(.0(x), .0(y)), +N(.1(x), .1(y)));
fun *Z(x: Z, y: Z) =
  I(+N(*N(.0(x), .0(y)), *N(.1(x), .1(y))),
    +N(*N(.0(x), .1(y)), *N(.1(x), .0(y))));
fun -Z(x: Z) = tuple(.1(x), .0(x));
fun -Z(x: Z, y: Z) = +Z(x, -Z(y));

// total order
pred <=Z(x: Z, y: Z) <=> =(.1(-Z(y, x)), N0);

// absolute value and sign
fun ||(x: Z) = if(<=Z(Z0, x), x, -Z(x));
fun sign(x: Z) = if(=(x, Z0), Z0, if(<=Z(Z0, x), Z1, -Z(Z1)));

// the set of integers in interval [-x, x]
fun Z[](x: Z) =
  let(y = .0(x):
    ++(set(z in nat(N0, y): true, tuple(z, N0)),
       set(z in nat(N0, y): true, tuple(N0, z))));

// quotient and remainder
fun divZ(x: Z, y: Z) =
```

```
  such(q in Z[](||(x)), r in  Z[](-Z(||(y), Z1)):
    and(=(x, +Z(*Z(q, y), r)),
      or(=(sign(r), Z0), =(sign(r), sign(y)))), q);
fun modZ(x: Z, y: Z) =
  such(q in Z[](||(x)), r in  Z[](-Z(||(y), Z1)):
    and(=(x, +Z(*Z(q, y), r)),
      or(=(sign(r), Z0), =(sign(r), sign(y)))), r);

// -----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// -----------------------------------------------------------------
```

C.5 Rational Numbers

```
// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// the rational numbers
//
// (c) 1999, Wolfgang Schreiner, see file COPYRIGHT
// http://www.risc.uni-linz.ac.at/software/formal
// ----------------------------------------------------------------
read integer;

// conversion predicates
fun Z(x: N) = tuple(x, N0);
fun N(x: Z) = .0(||(x));

// rationals are pairs of integers
pred Q(x) <=>
  and(Tuple(x), =(2, length(x)),
    Z(.0(x)), Z(.1(x)), not(<=Z(.1(x), Z0)),
    relprime(N(.0(x)), N(.1(x))));

// rational constructor and selectors
fun @(x: Z, y: Z) =
  let(g = Z(gcd(N(x), N(y))):
    tuple(*Z(sign(*Z(x, y)), divZ(||(x), g)), divZ(||(y), g)));
fun numerator(r: Z) = .0(r);
fun denominator(r: Z) = .1(r);

// constants
fun Q0 = @(Z0, Z1);
fun Q1 = @(Z1, Z1);
fun Q2 = @(Z2, Z1);

// basic arithmetic
fun +Q(x: Q, y: Q) =
  @(+Z(*Z(.0(x), .1(y)), *Z(.1(x), .0(y))), *Z(.1(x), .1(y)));
fun *Q(x: Q, y: Q) =
  @(*Z(.0(x), .0(y)), *Z(.1(x), .1(y)));
fun -Q(x: Q) = @(-Z(.0(x)), .1(x));
fun -Q(x: Q, y: Q) = +Q(x, -Q(y));
fun ^-1Q(x: Q) = @(.1(x), .0(x));
fun /Q(x: Q, y: Q) = *Q(x, ^-1Q(y));

// total order
pred <=Q(x: Q, y: Q) <=> <=Z(*Z(.0(x), .1(y)), *Z(.0(y), .1(x)));

// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// ----------------------------------------------------------------
```

C.6 Real Numbers

```
// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// the real numbers (faked by rationals)
//
// (c) 1999, Wolfgang Schreiner, see file COPYRIGHT
// http://www.risc.uni-linz.ac.at/software/formal
// ----------------------------------------------------------------

read rational;

pred R(x) <=> Q(x);
fun R(x: N) = @(Z(x), Z1);

fun R0 = Q0;
fun R1 = Q1;
fun R2 = Q2;

fun +R(x: R, y: R) = +Q(x, y);
fun *R(x: R, y: R) = *Q(x, y);
fun -R(x: R) = -Q(x);
fun -R(x: R, y: R) = -Q(x, y);
fun ^-1R(x: R) = ^-1Q(x);
fun /R(x: R, y: R) = /Q(x, y);

pred <=R(x: R, y: R) <=> <=Q(x, y);

// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// ----------------------------------------------------------------
```

C.7 Complex Numbers

```
// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// the complex numbers
//
// (c) 1999, Wolfgang Schreiner, see file COPYRIGHT
// http://www.risc.uni-linz.ac.at/software/formal
// ----------------------------------------------------------------
read real;

// complex numbers are pairs of reals
pred C(x) <=>
  and(Tuple(x), =(2, length(x)), R(.0(x)), R(.1(x)));

// complex constructors and
fun +i(x: R, y: R) = tuple(x, y);
fun real(r: R) = .0(r);
fun imaginary(r: R) = .1(r);

// constants
fun C0 = +i(R0, R0);
fun C1 = +i(R1, R0);
fun C2 = +i(R2, R0);
fun i  = +i(R0, R1);

// arithmetic
fun +C(x: C, y: C) = +i(+R(.0(x), .0(y)), +R(.1(x), .1(y)));
fun -C(x: C, y: C) = +i(-R(.0(x), .0(y)), -R(.1(x), .1(y)));
fun -(x: C) = -C(C0, x);
fun *C(x: C, y: C) =
  +i(-R(*R(.0(x), .0(y)), *R(.1(x), .1(y))),
     +R(*R(.0(x), .1(y)), *R(.1(x), .0(y))));
fun /C(x: C, y: C) =
  let(d = +R(*R(.0(y), .0(y)), *R(.1(y), .1(y))):
    +i(/R(+R(*R(.0(x), .0(y)), *R(.1(x), .1(y))), d),
       /R(-R(*R(.1(x), .0(y)), *R(.0(x), .1(y))), d)));
fun ^-1C(x: C) = /C(C1, x);

// complex conjugate
fun conj(x: C) = +i(.0(x), -R(.1(x)));

// square of absolute value
fun ||^2(x: C) = +R(*R(.0(x), .0(x)), *R(.1(x), .1(x)));

// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// ----------------------------------------------------------------
```

C.8 More on Functions

```
// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// functions
//
// (c) 1999, Wolfgang Schreiner, see file COPYRIGHT
// http://www.risc.uni-linz.ac.at/software/formal
// ----------------------------------------------------------------

read relation;

// the image of A' at f
fun image(f: Function, A': Set) =
  if(isSubset(A', domain(f)), set(x in A': true, apply(f, x)));

// the inverse image of B' at f
fun image^-1(f: Function, B': Set) =
  set(x in domain(f): in(apply(f, x), B'), x);

// f is injective from A to B
pred isInjective(f: Function, A: Set, B: Set) <=>
  and(isFunction(f, A, B),
    forall(x0 in A, x1 in A:
      implies(=(apply(f, x0), apply(f, x1)), =(x0, x1))));

// f is surjective from A to B
pred isSurjective(f: Function, A: Set, B: Set) <=>
  and(isFunction(f, A, B),
    forall(y in B:
      exists(x in A: =(apply(f, x), y))));

// f is bijective between A and B
pred isBijective(f: Function, A: Set, B: Set) <=>
  and(isInjective(f, A, B), isSurjective(f, A, B));

// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// ----------------------------------------------------------------
```

C.9 Real Functions

```
// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// functions over the reals
//
// (c) 1999, Wolfgang Schreiner, see file COPYRIGHT
// http://www.risc.uni-linz.ac.at/software/formal
// ----------------------------------------------------------------

read real;
read function;

// is B upper/lower bound of s in first n members of sequence s?
pred isUpperBound(B, s, n) <=>
  forall(i in nat(0, n): <=R(apply(s, i), B));
pred isLowerBound(B, s, n) <=>
  forall(i in nat(0, n): <=R(B, apply(s, i)));

// supremum/infimum of s considering only the first n members
fun sup(s, n) =
  such(U in range(s):
    and(isUpperBound(U, s, n),
      forall(U' in range(s):
        implies(isUpperBound(U', s, n), <=R(U, U')))), U);
fun inf(s, n) =
  such(L in range(s):
    and(isLowerBound(L, s, n),
      forall(L' in range(s):
        implies(isLowerBound(L', s, n), <=R(L', L)))), L);

// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// ----------------------------------------------------------------
```

C.10 Equivalence Relations

```
// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// equivalence relations
//
// (c) 1999, Wolfgang Schreiner, see file COPYRIGHT
// http://www.risc.uni-linz.ac.at/software/formal
// ----------------------------------------------------------------
read relation;

// R is reflexive on S
pred isReflexive(R: Relation, S) <=>
  forall(x in S: in(tuple(x, x), R));

// R is symmetric on S
pred isSymmetric(R: Relation, S) <=>
  forall(t in R: in(tuple(.1(t), .0(t)), R));

// R is transitive on S
pred isTransitive(R: Relation, S) <=>
  forall(s in R, t in R:
    implies(=(.1(s), .0(t)), in(tuple(.0(s), .1(t)), R)));

// R is equivalence relation on S
pred isEquivalence(R: Relation, S) <=>
  and(isRelation(R, S, S), isReflexive(R, S),
      isSymmetric(R, S), isTransitive(R, S));

// class [x]_R
fun [](x, R: Relation) =
  set(y in range(R): in(tuple(x, y), R), y);

// quotient set S/R
fun /(S, R: Relation) = set(x in S: true, [](x, R));

// D is partition on S
pred isPartition(D, S) <=>
  and(forall(x in D: not(=(x, {}))),
      forall(x in D, y in D: or(=(x, y), =(**(x, y), {}))),
      =(++(D), S));

// induced relation ~D
fun ~(D) =
  ++(set(d in D: true, set(x in d, y in d: true, tuple(x, y))));

// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// ----------------------------------------------------------------
```

C.11 Modular Arithmetic

```
// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// modular arithmetic
//
// (c) 1999, Wolfgang Schreiner, see file COPYRIGHT
// http://www.risc.uni-linz.ac.at/software/formal
// ----------------------------------------------------------------

read integer;
read equiv;

// positive integers
pred Z+(x) <=> and(Z(x), =(sign(x), Z1));

// just consider subset of Z
fun Z' = Z[](I(5, 0));

// x = y mod m
fun =m(m: Z+) =
  set(x in Z', y in Z': =(modZ(x, m), modZ(y, m)), tuple(x, y));

// Z_m
fun Z(m: Z+) = /(Z', =m(m));

// selector function
fun _(x) = such(a in x: true, a);

// equivalence class of x modulo m
fun []m(m: Z+, x: Z) = [](x, =m(m));

// modular arithmetic
fun +m(m: Z+, x in Z(m), y in Z(m)) = []m(m, +Z(_(x), _(y)));
fun -m(m: Z+, x in Z(m), y in Z(m)) = []m(m, -Z(_(x), _(y)));
fun -m(m: Z+, x in Z(m))            = []m(m, -Z(_(x)));
fun *m(m: Z+, x in Z(m), y in Z(m)) = []m(m, *Z(_(x), _(y)));

// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// ----------------------------------------------------------------
```

C.12 Integers as Equivalence Classes

```
// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// the integer numbers as equivalence classes
//
// (c) 1999, Wolfgang Schreiner, see file COPYRIGHT
// http://www.risc.uni-linz.ac.at/software/formal
// ----------------------------------------------------------------
read equiv; read natural0;

// just consider subset of N x N
fun Z' = set(x in nat(0, 6), y in nat(0, 6): true, tuple(x, y));

// x0-x1 = y0-y1
fun ~Z = set(x in Z', y in Z':
  =(+N(.0(x), .1(y)), +N(.0(y), .1(x))), tuple(x, y));

// Z as a quotient set
fun Z = /(Z', ~Z);

// selector function
fun _(x) = such(a in x: true, a);

// equivalence class of x
fun []Z(x in Z') = [](x, ~Z);

// constants
fun Z0 = []Z(tuple(N0, N0));
fun Z1 = []Z(tuple(N1, N0));
fun Z2 = []Z(tuple(N2, N0));

// arithmetic
fun +Z(x in Z, y in Z) =
  []Z(tuple(+N(.0(_(x)), .0(_(y))), +N(.1(_(x)), .1(_(y)))));
fun -Z(x in Z) = []Z(tuple(.1(_(x)), .0(_(x))));
fun -Z(x in Z, y in Z) =
  []Z(tuple(+N(.0(_(x)), .1(_(y))), +N(.0(_(y)), .1(_(x)))));
fun *Z(x in Z, y in Z) =
  []Z(tuple(+N(*N(.0(_(x)), .0(_(y))), *N(.1(_(x)), .1(_(y)))),
            +N(*N(.0(_(x)), .1(_(y))), *N(.1(_(x)), .0(_(y))))));

// order
pred <=Z(x in Z, y in Z) <=>
  <=N(+N(.0(_(x)), .1(_(y))), +N(.0(_(y)), .1(_(x))));

// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// ----------------------------------------------------------------
```

C.13 Rationals as Equivalence Classes

```
// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// the rational numbers as equivalence classes
//
// (c) 1999, Wolfgang Schreiner, see file COPYRIGHT
// http://www.risc.uni-linz.ac.at/software/formal
// ----------------------------------------------------------------

read equiv;
read integer;

// just consider subset of Z x Z
fun Q' =
  set(x in Z[](I(3, 0)), y in Z[](I(3, 0)):
    not(=(y, Z0)), tuple(x, y));

// x0/x1 = y0/y1
fun ~Q =
  set(x in Q', y in Q':
    =(*Z(.0(x), .1(y)), *Z(.0(y), .1(x))), tuple(x, y));

// Q as a quotient set
fun Q = /(Q', ~Q);

// selector function
fun _(x) = such(a in x: true, a);

// equivalence class of x
fun []Q(x in Q') = [](x, ~Q);

// constants
fun Q0 = []Q(tuple(Z0, Z1));
fun Q1 = []Q(tuple(Z1, Z1));
fun Q2 = []Q(tuple(Z2, Z1));

// arithmetic
fun +Q(x in Q, y in Q) =
  []Q(tuple(+Z(*Z(.0(_(x)), .1(_(y))), *Z(.0(_(y)), .1(_(x)))),
            *Z(.1(_(x)), .1(_(y)))));
fun -Q(x in Q) =
  []Q(tuple(-Z(.0(_(x))), .1(_(x))));
fun -Q(x in Q, y in Q) =
  []Q(tuple(-Z(*Z(.0(_(x)), .1(_(y))), *Z(.0(_(y)), .1(_(x)))),
            *Z(.1(_(x)), .1(_(y)))));
fun *Q(x in Q, y in Q) =
  []Q(tuple(*Z(.0(_(x)), .0(_(y))), *Z(.1(_(x)), .1(_(y)))));
fun ^-1Q(x in Q) =
```

```
  []Q(tuple(.1(_(x)), .0(_(x))));
fun /Q(x in Q, y in Q) =
  []Q(tuple(*Z(.0(_(x)), .1(_(y))), *Z(.0(_(y)), .1(_(x)))));

// order
pred <=Q(x in Q, y in Q) <=>
  <=Z(*Z(.0(_(x)), .1(_(y))), *Z(.0(_(y)), .1(_(x))));

// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// ----------------------------------------------------------------
```

C.14 Order Relations

```
// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// order relations
//
// (c) 1999, Wolfgang Schreiner, see file COPYRIGHT
// http://www.risc.uni-linz.ac.at/software/formal
// ----------------------------------------------------------------

read equiv;

// R is antisymmetric on S
pred isAntiSymmetric(R: Relation, S) <=>
  forall(x in S, y in S:
    implies(and(in(tuple(x, y), S), in(tuple(y, x), S)), =(x, y)));

// R is partial order on S
pred isPartialOrder(R: Relation, S) <=>
  and(isRelation(R, S, S),
      isReflexive(R, S),
      isAntiSymmetric(R, S),
      isTransitive(R, S));

// R is irreflexive on S
pred isIrreflexive(R: Relation, S) <=>
  forall(x in S: not(in(tuple(x, x), R)));

// R is partial order on S
pred isQuasiOrder(R: Relation, S) <=>
  and(isRelation(R, S, S),
      isIrreflexive(R, S),
      isAntiSymmetric(R, S),
      isTransitive(R, S));

// x and y are incomparable with respect to R
pred areIncomparable(x, y, R) <=>
  and(not(in(tuple(x, y), R)), not(in(tuple(y, x), R)));

// R is total order on S
pred isTotalOrder(R: Relation, S) <=>
  and(isRelation(R, S, S),
      not(exists(x in S, y in S: areIncomparable(x, y, R))));

// x is least/greatest element in S with respect to R
pred isLeast(x, S, R) <=>
  and(in(x, S), forall(y in S: in(tuple(x, y), R)));
pred isGreatest(x, S, R) <=>
  and(in(x, S), forall(y in S: in(tuple(y, x), R)));
```

```
// x is minimal/maximal element of S with respect to R
pred isMinimal(x, S, R) <=>
  and(in(x, S), forall(y in S:
    implies(in(tuple(y, x), R), =(y, x))));
pred isMaximal(x, S, R) <=>
  and(in(x, S), forall(y in S:
    implies(in(tuple(x, y), R), =(x, y))));

// x is lower/upper bound of S with respect to R
pred isLowerBound(x, S, R) <=>
  forall(y in S: in(tuple(x, y), R));
pred isUpperBound(x, S, R) <=>
  forall(y in S: in(tuple(y, x), R));

// x is infimum/supremum of S with respect to R
pred isInfimum(x, S, R) <=>
  and(isLowerBound(x, S, R),
      forall(y in domain(R):
        implies(isLowerBound(y, S, R), in(tuple(y, x), R))));
pred isSupremum(x, S, R) <=>
  and(isUpperBound(x, S, R),
      forall(y in range(R):
        implies(isUpperBound(y, S, R), in(tuple(x, y), R))));

// ----------------------------------------------------------------
// $Id: evaluator.tex,v 1.7 1999/09/29 12:28:26 schreine Exp $
// ----------------------------------------------------------------
```

Index