CS_376 Programming with Abstract Data Types

Ulrich Berger 1

Lecture Notes, Autumn 2001

1 Introduction

This course gives an introduction to some advanced methods for the development of reliable software. By “reliable software” we mean computer programs that are

(i) adequate, i.e. solve the customers problem,

(ii) correct, i.e. are free of bugs and thus behave the way the programmer wants them to behave,

(iii) easy to maintain, i.e. can be easily modified or extended without introducing new errors.

That conventional programming techniques to a large extent fail to produce software meeting these requirements follows from the fact that approximately 80% of the total time and money presently invested into software development is estimated to be spent for finding errors and amending incorrect or poorly designed software. Hence there is an obvious need for better programming methodologies.

The program development methods studied in this course will mainly rely on mathematical modeling, formal specification and formal reasoning techniques. These issues also played an important role in many of the first and second year courses, for example in Modeling Computing Systems, Logic Programming, System Specification, Theory of Programming Languages, Logic and Foundations of Mathematics, and Mathematics for Computation.

Literature mainly used in this course.


1u.berger@swan.ac.uk, tel. 205678-4517, room 303 Dept. of Computer Science. These are preliminary course notes; comments are welcome.
1.1 The software design process

In conventional software design one writes a program that is supposed to solve a given problem. The program is then tested and altered until no errors are unveiled by the tests. After that the program is put into practical use. At this stage often new errors pop up or the program appears to be inadequate. The process of maintaining is then started by repeating the different design steps. This methodology is illustrated by the so-called software life-cycle model (figure 1). It has at least two deficiencies. First, being based on tests, it can only confirm the existence of errors, not their absence. Hence testing fails to prove the correctness of a program. A second deficiency of testing is the fact that results are compared with expectations resulting from one’s own understanding of the problem. Hence testing may fail to unveil inadequacies of the program.

The goal of a better methodology for software design is to avoid errors and inadequacies as far as possible, or at least to try to detect and correct them in an early stage of the design. The main idea is to derive a program from a problem in several controlled steps as illustrated in figure 2.

![Software Life-Cycle Model Diagram](image-url)
1. From a careful analysis of the customer’s problem one derives an informal specification abstracting from all unnecessary details.

2. The informal specification is formalized i.e. written in a formal language. If the specification is of a particularly simple form (equational for term rewriting, or Horn-clausal form for logic programming) it will be executable (rapid prototyping) and can be used for detecting inadequacies of the specification at an early stage.

3. From the formal specification a certain method of program development leads to a program that is provably correct, i.e. it can be proven that it meets the specification (verification). In the course we will discuss different such methods, two of them being illustrated in the example in section 1.2.

4. The derived program can be compiled and executed and the results can be used to test the program.
1.2 Example

Problem

Compute the $gcd$ of two positive natural numbers $m$, $n$.

Informal specification

$gcd(m, n)$ is a number $k$ that divides $m$ and $n$, such that if $l$ is any other number $l$ also dividing $m$ and $n$ then $l$ divides $k$.

Formal specification

$$ k = gcd(m, n) \iff k \mid m \land k \mid n \land \forall l \mid m \land l \mid n \rightarrow l \mid k \land k \mid m \iff \exists q \, k \ast q = m $$

Transformation

<table>
<thead>
<tr>
<th>Formal specification 1</th>
<th>Formal specification 2</th>
<th>Formal specification 3</th>
</tr>
</thead>
</table>
| $\exists r \mid r < n \land \exists q \mid m = q \ast n + r \land r = 0 \rightarrow gcd(m, n) = n \land r > 0 \rightarrow gcd(m, n) = gcd(n, r)$ | $mod(m, n) < n \land \exists q \mid m = q 
\ast n + mod(m, n) \land mod(m, n) = 0 \rightarrow gcd(m, n) = n \land mod(m, n) > 0 \rightarrow gcd(m, n) = gcd(n, mod(m, n))$ | $m < n \rightarrow mod(m, n) = m \land m \geq n \rightarrow mod(m, n) = mod(m - n, n) \land mod(m, n) = 0 \rightarrow gcd(m, n) = n \land mod(m, n) > 0 \rightarrow gcd(m, n) = gcd(n, mod(m, n))$ |

Program extraction

<table>
<thead>
<tr>
<th>Prove the formula</th>
<th>From a formal proof</th>
<th>provably satisfying the specification, that is, the formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall m &gt; 0 \forall n &gt; 0 \exists k \mid k \mid m \land k \mid n \land \forall l \mid m \land l \mid n \rightarrow l \mid k$</td>
<td>extract a program $gcd$</td>
<td>$\forall m &gt; 0 \forall n &gt; 0 \forall l \mid m \land gcd(m, n) \mid n \land gcd(m, n)$</td>
</tr>
</tbody>
</table>

Program

```plaintext
function mod (m,n:integer, m,n>0) : integer;
begin
  if m < n then mod := m
  else mod := mod(m-n,n)
end

function gcd (m,n:integer, n>0) : integer;
begin
  r := mod(m,n);
  if r=0 then gcd := n
  else gcd := gcd(n,r)
end
```
1.3 Programming with Abstract data Types

In the previous example 1.2 the program development on the left hand side proceeds by a stepwise refinement of the original formal specification.

The first step introduces the essential algorithmic idea due to Euclid. Formally it has the effect that the universal quantifier, $\forall l$, in the original specification is eliminated.

In the second step the existential quantifier, $\exists r$, is replaced by introducing the function symbol $\text{mod}$ for the modulus function computing the remainder in an integer division.

In the third step the remaining existence quantifier, $\exists q$, in the specification of the modulus is replaced by an equational description embodying the algorithmic idea for computing the modulus.

The third specification contains no quantifiers and has the form of a conjunction of conditional equations. This specification can automatically be transformed into corresponding recursive programs computing the modulus and the greatest common divisor.

In order to make this program development complete one has to establish its correctness, which means that one has to prove the implications

\[
\text{Formal specification} \\
\uparrow \\
\text{Formal specification 1} \\
\uparrow \\
\text{Formal specification 2} \\
\uparrow \\
\text{Formal specification 3}
\]

Finally, a proof is required that the derived program terminates on all legal inputs.

Formal specifications as they occur in this program development are often called algebraic specifications because their natural interpretations, or models, are (many-sorted) algebras. The class of models of an algebraic specification forms an abstract data type (ADT). In the literature (but not in this course) algebraic specifications and abstract data types are often identified.

Program development using ADTs is a well-established technique for producing reliable software. Its main methodological principles are

- **abstraction**, i.e. the description of unnecessary details is avoided,
- **modularization**, i.e. the programming task is divided into small manageable pieces that can be solved independently.
While our example was too simple to show the effect of abstraction, modularization took place by dividing the development into four relatively small steps and separating the problem of computing the modulus as an independent programming task.


1.4 Program Extraction from Proofs

The right hand side of example 1.2 indicates how to develop a program using the method of program extraction from formal proofs. This method can be described in general (and somewhat simplified) as follows:

1. We assume that the programming problem is given in the form

\[ \forall x \exists y A(x, y) \]

(in our example, \( \forall m, n \exists k \) (k is the greatest common divisor of \( m \) and \( n \)), where \( m, n \) range over positive natural numbers).

2. One finds (manually, or computer-aided) a constructive formal proof of the formula

\[ \forall x \exists y A(x, y) \]

3. From the proof a program \( p \) (in or example the program for \( \text{gcd} \)) is extracted (fully automatically) that provable meets the specification, that is,

\[ \forall x A(x, p(x)) \]

is provable (in our example, \( \forall m, n \) (\( \text{gcd}(m, n) \) is the greatest common divisor of \( m \) and \( n \))).

The concept of a constructive proof as an alternative foundation for logic and mathematics has been advocated first by L Kronecker, L E J Brouwer and A Kolmogorov in the beginning of the 20th century, and was formalized by Brouwer’s student A Heyting. The algorithmic interpretation of constructive proofs was formulated first by Brouwer, Heyting and Kolmogorov and is therefore often called BHK-interpretation (cf [5] p. 156). In the Computer Science community the names Curry-Howard-interpretation (after the American mathematicians H B Curry and W Howard), or proofs-as-programs paradigm are more popular. According to the proofs-as-programs paradigm we have the following correspondences

\[
\begin{align*}
\text{formula} & \equiv \text{data type} \\
\text{constructive proof of formula } A & \equiv \text{program of data type } A
\end{align*}
\]

The constructive proof calculus studied in this course will be natural deduction. We will mainly follow the textbook [5].

There exist a number of systems supporting program extraction from proofs (e.g. Agda, Coq, Fred, Minlog, NuPrl, PX). Time permitting, we will look at some of these systems in this course and carry out small case studies of program extraction.
2 Logic

In this chapter we study the syntax and semantics of many-sorted first-order predicate logic.

2.1 Signatures and algebras

The purpose of a signature is to provide names for objects and operations and fix their format. Hence a signature can be viewed as the mathematical counterpart to the programming concept of an interface.

2.1.1 Definition

A many-sorted signature (signature for short), is a pair $\Sigma = (S, \Omega)$ such that the following conditions are satisfied.

- $S$ is a set. The elements $s \in S$ are called sorts.
- $\Omega$ is a set whose elements are called operations, and which are of the form

$$f : s_1 \times \ldots \times s_n \to s,$$

where $n \geq 0$ and $s_1, \ldots, s_n, s \in S$.

$s_1 \times \ldots \times s_n \to s$ is called arity of $f$, with argument sorts $s_1, \ldots, s_n$ and target sort $s$.

Operations of the form $c : \to s$ (i.e. $n = 0$) are called constants of sort $s$. For constants we often use the shorter notation $c : s$ (i.e. we omit the arrow).

Signatures are interpreted by mathematical structures called algebras. An algebra can be viewed as the mathematical counterpart to the programming concept of a concrete data type.

2.1.2 Definition

A many-sorted algebra $A$ (algebra for short) for a signature $\Sigma = (S, \Omega)$ is given by the following.

- For each sort $s$ in $S$ a set $A_s$, called the carrier set of the sort $s$.
- For each constant $c : s$ in $\Omega$ an element $c^A \in A_s$.
- For each operation $f : s_1 \times \ldots \times s_n \to s$ in $\Omega$ a function

$$f^A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$$
2.1.3 Remarks

1. In the definition of an algebra (2.1.1) the expression \( f : s_1 \times \ldots \times s_n \rightarrow s \) is meant *symbolically*, i.e. ‘\( \times \)' and ‘\( \rightarrow \)' are to be read as uninterpreted symbols. In the definition of an algebra (2.1.2), however, we used the familiar mathematical notation for set-theoretic functions to communicate by \( f^A : A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_s \) a *semantical* object, namely a function \( f^A \) whose domain is the cartesian product of the sets \( A_{s_i} \) and whose range is \( A_s \).

2. It is common, and we will do so, to call the elements \( c^A \) *constants*, and the functions \( f^A \) *operations*. Hence the words ‘constant’ and ‘operation’ have a double meaning. However, it should always be clear from the context what is meant.

3. By a \( \Sigma \)-algebra we mean an algebra for the signature \( \Sigma \).

4. In logicians jargon a signature is called a *many-sorted first-order language* and an algebra is called a *many-sorted first-order structure*.

2.1.4 Example

Consider the signature \( \Sigma := (S, \Omega) \), where

\[
S = \{ \text{nat, boole} \}
\]

\[
\Omega = \{ 0 : \text{nat, T: boole, F: boole, add: nat \times nat \to nat, le: nat \times nat \to boole} \}
\]

For better readability we display this signature by

<table>
<thead>
<tr>
<th>Signature</th>
<th>( \Sigma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>nat, boole</td>
</tr>
<tr>
<td>Constants</td>
<td>0 : nat, T : boole, F : boole</td>
</tr>
<tr>
<td>Operations</td>
<td>add : nat \times nat \to nat</td>
</tr>
<tr>
<td></td>
<td>\leq : nat \times nat \to boole</td>
</tr>
</tbody>
</table>

The \( \Sigma \)-algebra \( A \) of natural numbers with 0, and addition and the relation \( \leq \) is given by

the carrier sets \( N = \{ 0, 1, 2, \ldots \} \) and \( B = \{ T, F \} \), i.e.

\[
A_{\text{nat}} = N, \quad A_{\text{boole}} = B,
\]

the constants \( 0, T, F \), i.e.

\[
0^A = 0, \quad T^A = T, \quad F^A = F,
\]

the operations of addition on \( N \) and the comparison relation \( < \) viewed as a boolean function, i.e. for all \( n, m \in N \),
add^A(n, m) = n + m,
≤^A (n, m) = \begin{cases} T & \text{if } n \leq m \\ F & \text{otherwise} \end{cases}

Again we use a notation that is better readable:

<table>
<thead>
<tr>
<th>Algebra</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carriers</td>
<td>N, B</td>
</tr>
<tr>
<td>Constants</td>
<td>0, T, F</td>
</tr>
<tr>
<td>Operations</td>
<td>+: N × N → N</td>
</tr>
<tr>
<td></td>
<td>≤: N × N → B</td>
</tr>
</tbody>
</table>

For the signature Σ we may also consider another algebra, B, with carrier \( N^+ := N \setminus \{0\} \) (= \{1, 2, 3, 4, \ldots\}), the constants 1, T, F, multiplication restricted to M, and the divisibility relation \( \cdot \mid \cdot \). Hence we have

\[ B_{\text{nat}} = N^+, \quad B_{\text{bool}} = B, \]
\[ 0^B = 1, \quad T^B = T, \quad F^B = F, \]
\[ \text{add}^B(n, m) = n \cdot m \text{ for all } n, m \in N^+. \]
\[ \leq^A (n, m) = \begin{cases} T & \text{if } n \mid m \\ F & \text{otherwise} \end{cases} \]

Written in a box

<table>
<thead>
<tr>
<th>Algebra</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carriers</td>
<td>N^+, B</td>
</tr>
<tr>
<td>Constants</td>
<td>1, T, F</td>
</tr>
<tr>
<td>Operations</td>
<td>*: N^+ × N^+ → N^+</td>
</tr>
<tr>
<td></td>
<td>#: N^+ × N^+ → B</td>
</tr>
</tbody>
</table>
2.1.5 Remarks

1. The display of signatures and algebras via boxes has to be handled with some care. If, for example, in the box displaying the signature $\Sigma$ in example 2.1.4 we would have exchanged the order of the sorts $\text{nat}$ and $\text{boole}$, we would have defined the same signature. But then the box displaying the algebra $A$ would not be well-defined, since then the sort $\text{boole}$ would be associated with the set $\mathbb{N}$ and $\text{nat}$ with $\mathbb{B}$, and consequently the arities of the operations of $\text{Sigma}$ would not fit with the operations of the algebra $A$.

Moral: When displaying signatures and algebras in boxes order matters.

2. Operations with target sort boole are often called predicates.

2.2 Terms

The constants and operations of a signature $\Sigma$ can be used to build formal expressions, called terms, which denote elements of a $\Sigma$-algebra.

2.2.1 Definition

Let $\Sigma = (S, \Omega)$ be a signature, and let $X = (X_s)_{s \in S}$ be a family of pairwise disjoint sets. The elements of $X_s$ are called variables of sort $s$. We define terms and their sorts by the following rules.

(i) Every variable $x \in X_s$ is a term of sort $s$.

(ii) Every constant $c$ in $\Sigma$ of sort $s$ is a term of sort $s$.

(iii) If $f : s_1 \times \ldots \times s_n \to s$ is an operation in $\Sigma$, and $t_1, \ldots, t_n$ are (previously defined) terms of sorts $s_1, \ldots, s_n$, respectively, then the formal expression

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

is a term of sort $s$.

The set of all terms of sort $s$ is denoted by $T(\Sigma, X)_s$.

A term is closed if it doesn’t contain variables, i.e. is built without the use of rule (i).

The set of all closed terms of sort $s$ is denoted by $T(\Sigma)_s$. Clearly $T(\Sigma)_s = T(\Sigma, \emptyset)_s$.

2.2.2 Example

For the signature $\Sigma$ of example 2.1.4 and the set of variables $X := \{x, y\}$ the following are examples of terms in $T(\Sigma, X)$:

$$x$$
0
add(0, y)
add(add(0, x), y)
add(add(0, 0), add(x, x))
add(0, add(0, add(0, 0)))

The second and the last of these terms are closed.

In order to declare the semantics of terms in a \( \Sigma \)-algebra \( A \) we have to define for each term \( t \) of sort \( s \) its value in \( A_s \), i.e. the element in \( A_s \) that is denoted by \( t \). The value of \( t \) will in general depend on the values assigned to the variables occurring in \( t \).

### 2.2.3 Definition (Semantics of terms)

Let \( A \) be an algebra for the signature \( \Sigma = (S, \Omega) \), and let \( X = (X_s)_{s \in S} \) a set of variables.

A variable assignment \( \alpha : X \to A \) is a function assigning to every variable \( x \in X_s \) an element \( \alpha(x) \in A_s \).

Given a variable assignment \( \alpha : X \to A \) we define for each term \( t \in T(\Sigma, X) \) its value

\[
t^{A,\alpha} \in A_s
\]

by the following rules.

(i) \( x^{A,\alpha} := \alpha(x) \).

(ii) \( c^{A,\alpha} := c^A \).

(iii) \( f(t_1, \ldots, t_n)^{A,\alpha} := f^A(t_1^{A,\alpha}, \ldots, t_n^{A,\alpha}) \).

For closed terms \( t \), i.e. \( t \in T(\Sigma) (= T(\Sigma, \emptyset)) \) the variable assignment \( \alpha \) and rule (i) are obsolete and we write \( t^A \) instead of \( t^{A,\alpha} \).

### 2.2.4 Remark

The definition of \( t^{A,\alpha} \) is by recursion on the term structure (also called structural recursion). In general a function on terms can be defined by recursion on the term structure by defining it for atomic terms, i.e. constants and variables (rules (i) and (ii)), and recursively for a compound term \( f(t_1, \ldots, t_n) \) using the values of the function at the components \( t_1, \ldots, t_n \).

### 2.2.5 Exercise

Define the set \( \text{var}(t) \) of all variables occurring in a term \( t \) by structural recursion on \( t \).
2.2.6 Example

Let us calculate the values of the terms in example 2.1.4 in the $\Sigma$-algebra $A$ under the variable assignment $\alpha : \{x, y\} \to \mathbb{N}$, $\alpha(x) := 3$ and $\alpha(y) := 5$.

$$x^{A,\alpha} = \alpha(x) = 3$$
$$0^{A,\alpha} = 0^A = 0$$
$$\text{add}(0, y)^{A,\alpha} =$$
$$\text{add}(\text{add}(0, x), y)^{A,\alpha} =$$
$$\text{add}(\text{add}(0, 0), \text{add}(x, x))^{A,\alpha} =$$
$$\text{add}(0, \text{add}(0, \text{add}(0, 0)))^{A,\alpha} =$$

Terms can be used to construct to every signature and variable set a ‘canonical’ algebra.

2.2.7 Definition

Let $\Sigma = (S, \Omega)$ a signature and $X$ a set of variables for $\Sigma$. We define a $\Sigma$-algebra $T(\Sigma, X)$, called term algebra, as follows.

<table>
<thead>
<tr>
<th>Algebra</th>
<th>$T(\Sigma, X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carriers</td>
<td>$T(\Sigma, X)_s$ (s $\in S$)</td>
</tr>
<tr>
<td>Constants</td>
<td>$c^{T(\Sigma, X)} := c$</td>
</tr>
<tr>
<td>Operations</td>
<td>$f^{T(\Sigma, X)}(t_1, \ldots, t_n) := f(t_1, \ldots, t_n)$</td>
</tr>
</tbody>
</table>

In the special case $X = \emptyset$ we write $T(\Sigma)$ for $T(\Sigma, X)$ and call this the closed term algebra.

2.3 Formulas

In a similar way as terms are syntactic constructs denoting objects, formulas are syntactic construct to denote propositions.
2.3.1 Definition

The set of formulas over a signature $\Sigma = (S, \Omega)$ and a set of variables $X = (X_s)_{s \in S}$ is defined inductively by the following rules.

(i) $\bot$ is a formula, called absurdity.

(ii) $t_1 = t_2$ is a formula for each pair of terms $t_1, t_2 \in T(\Sigma, X)$ of the same sort.

(iii) If $P$ and $Q$ are formulas then $P \rightarrow Q$, $P \land Q$, and $P \lor Q$ are formulas.

(iv) If $P$ is a formula then $\forall x P$ and $\exists x P$ are formulas for every variable $x \in X$.

Formulas over a signature $\Sigma$ are also called $\Sigma$-formulas.

A free occurrence of a variable $x$ in a formula $P$ is an occurrence of $x$ in $P$ which is not in the scope of a quantifier $\forall x$ or $\exists x$. We let $\text{FV}(P)$ denote the set of free variables of $P$, i.e. the set of variables with a free occurrence in $P$. A formula $P$ is closed if $\text{FV}(P) = \emptyset$.

We set

$$\mathcal{L}(\Sigma, X) := \{P \mid P \text{ is a } \Sigma\text{-formula, } \text{FV}(P) \subseteq X\}$$

and use the abbreviation

$$\mathcal{L}(\Sigma) := \mathcal{L}(\Sigma, \emptyset),$$

i.e. $\mathcal{L}(\Sigma)$ is the set of closed $\Sigma$-formulas.

2.3.2 Remarks and Notations

1. Formulas as defined above are usually called first-order formulas, since we allow quantification over object variables only. If we would also quantify over set variables we would obtain second-order formulas.

2. A formula is quantifier free, qf for short, if it doesn’t contain quantifiers.

3. An equation is a formula of the form $t_1 = t_2$.

4. A formula is universal if it is of the form $\forall x_1 \ldots \forall x_n P$ where $P$ is quantifier free.
2.3.3 Abbreviations

<table>
<thead>
<tr>
<th>Formula</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg P )</td>
<td>( P \rightarrow \bot ) (negation)</td>
</tr>
<tr>
<td>( \forall x_1 \forall x_2 \ldots \forall x_n \ P )</td>
<td>( \forall x_1, x_2, \ldots, x_n P )</td>
</tr>
<tr>
<td>( \exists x_1 \exists x_2 \ldots \exists x_n \ P )</td>
<td>( \exists x_1, x_2, \ldots, x_n P )</td>
</tr>
<tr>
<td>( \forall x_1, \ldots, x_n P ), where ( {x_1, \ldots, x_n} = \text{FV}(P) )</td>
<td>( \forall P ) (closure of ( P ))</td>
</tr>
<tr>
<td>( (P \rightarrow Q) \land (Q \rightarrow P) )</td>
<td>( P \leftrightarrow Q ) (equivalence)</td>
</tr>
<tr>
<td>( t = T )</td>
<td>( t ) (provided ( t ) is of sort boole)</td>
</tr>
</tbody>
</table>

2.3.4 Examples

\[ P_1 : \equiv \ T = F \]
\[ P_2 : \equiv \ x = 0 \rightarrow \text{add}(y, x) = y \]
\[ P_3 : \equiv \ \exists y \ (x = y \rightarrow \forall z \ x = z) \]
\[ P_4 : \equiv \ \forall x \ (0 \leq x = T) \]

\( P_1 \) and \( P_2 \) are quantifier free.
\( P_1 \) is an equation. \( P_3 \) is universal.
\( P_1 \) and \( P_4 \) are closed.
\( \text{FV}(P_2) = \{x, y\}, \text{FV}(P_3) = \{x\} \).
\( P_4 \) can be abbreviated \( \forall x \ (0 \leq x) \).

2.3.5 Exercise

Formalization exercises !!!

In order to precisely declare the semantics of a formula we define what it means for a formula to be true in an algebra.
2.3.6 Definition (Semantics of formulas)

Let $\Sigma = (S, \Omega)$ be a signature, $X = (X_s)_{s \in S}$ a set of variables, $A$ a $\Sigma$-algebra $\alpha : X \to A$, and $P \in \mathcal{L}(\Sigma, X)$.

We define the relation

$$A, \alpha \models P$$

which is to be read ‘$P$ is true in $A$ under $\alpha$’, or ‘$A, \alpha$ is a model of $P$', by structural recursion on the formula $P$.

(i) $A, \alpha \not\models \bot$, i.e. $A, \alpha \models \bot$ does not hold.

(ii) $A, \alpha \models t_1 = t_2$ iff $t_1^A \alpha = t_2^A \alpha$.

(iii) $A, \alpha \models P \land Q$ iff $A, \alpha \models P$ and $A, \alpha \models Q$.

$A, \alpha \models P \lor Q$ iff $A, \alpha \models P$ or $A, \alpha \models Q$.

$A, \alpha \models P \rightarrow Q$ iff $A, \alpha \models P$ implies $A, \alpha \models Q$ (i.e. $A, \alpha \not\models P$ or $A, \alpha \models Q$).

(iv) $A, \alpha \models \forall x P$ iff $A, \alpha^a_x \models P$ for all $a \in A_s$ (provided $x$ is of sort $s$).

$A, \alpha \models \exists x P$ iff $A, \alpha^a_x \models P$ for at least one $a \in A_s$ (provided $x$ is of sort $s$).

In (iii) we used the updated variable assignment $\alpha^a_x$ defined by $\alpha^a_x(x) = a$ and $\alpha^a_x(y) = \alpha(y)$ for every variable different from $x$.

For closed $\Sigma$-formulas $P$ the variable assignment is obviously redundant and we write

$$A \models P$$

for $A, \alpha \models P$. For a set $\Gamma$ of closed $\Sigma$-formulas we say that the $\Sigma$-algebra $A$ is a model of $\Gamma$, written

$$A \models \Gamma$$

if $A \models P$ for all $P \in \Gamma$.

2.3.7 Definition (Logical consequence)

Let $\Gamma$ be a set of closed formulas and $P$ a closed formula. We say that $P$ is a logical consequence of $\Gamma$, or $\Gamma$ logically implies $A$, written

$$\Gamma \models P,$$

if $P$ is true in all models of $\Gamma$, that is,

$$A \models \Gamma \iff A \models P$$

for all $\Sigma$-algebras $A$. 
2.3.8 Definition (Logical validity)

A closed $\Sigma$-formula $P$ is said to be (logically) valid, written

$$\models P$$

if $P$ is true in all $\Sigma$-algebras, that is $A \models P$ for all $\Sigma$-algebras $A$. Valid formulas is also called a tautologies.

Obviously, $P$ is valid if and only if it is a logical consequence of the empty set of formulas.

2.3.9 Definition (Satisfiability)

A set of closed $\Sigma$-formulas $\Gamma$ is called satisfiable if it has a model, that is, there exists a $\Sigma$-algebra $A$ in which all formulas of $\Gamma$ are true ($A \models \Gamma$).

2.3.10 Exercise

Show that validity and satisfiability are are related by the following equivalences:

- $P$ valid $\iff$ $\{\neg P\}$ unsatisfiable (that is, not satisfiable)
- $P$ satisfiable $\iff$ $\{\neg P\}$ not valid

2.3.11 Theorem (A Church)

It is undecidable whether or not a closed formula is valid.

This theorem can be proven by reducing the halting problem to the validity problem (i.e. coding Turing machines into logic).

Although the validity problem is undecidable, there is an effective procedure generating all valid formulas (technically: the set of valid formulas is recursively enumerable). We will study such a generation process in the next chapter.

2.3.12 Examples

Examples of logical consequence, tautologies and satisfiable formulas !!!
2.4.1 Definition

Let $\Sigma = (S, \Omega)$ be a signature and $X = \{X_s\}_{s \in S}$, $Y = \{Y_s\}_{s \in S}$, two sets of variables.

A substitution is a mapping $\theta: X \to T(\Sigma, Y)$ that respects sorts, i.e. the variable $x$ and the term $\theta(x)$ have the same sorts for all $x \in X$.

Given a substitution $\theta$ we define for every $t \in T(\Sigma, X)$ a term $t\theta \in T(\Sigma, Y)$ by

$$t\theta := \text{the result of replacing every occurrence of a variable } x \text{ in } t \text{ by } \theta(x)$$

Equivalently $t\theta$ can be defined by recursion on the term structure:

(i) $x\theta := \theta(x)$.

(ii) $c\theta := c$.

(iii) $f(t_1, \ldots, t_n)\theta := f(t_1\theta, \ldots, t_n\theta)$.

This also yields a recursive algorithm for computing $t\theta$.

Notation

(a) By $\{t_1/x_1, \ldots, t_n/x_n\}$ we denote the substitution $\theta$ such that $\theta(x_i) = t_i$ for $i = 1, \ldots, n$ and $\theta(x) = x$ if $x \notin \{x_1, \ldots, x_n\}$. Of course this implicitly assumes that $x_1$ and $t_1$ have the same sort, and the variables $x_i$ are all distinct.

(b) If $\theta: X \to T(\Sigma, Y)$ and $\sigma: Y \to T(\Sigma, Z)$ are substitutions, then we define the substitution $\theta\sigma: X \to T(\Sigma, Z)$ by

$$(\theta\sigma)(x) := \theta(x)\sigma$$

It can be easily proved that

$$t(\theta\sigma) = (t\theta)\sigma$$

for all terms $t \in T(\Sigma, X)$ (see proof below).

(c) If $\theta: X \to T(\Sigma, Y)$ is a substitution and $\alpha: Y \to A$ is a variable assignment, then the variable assignments $\theta^{A,\alpha}: X \to A$ is defined by

$$\theta^{A,\alpha}(x) := (\theta(x))^{A,\alpha}$$

Note that for a substitution $\{t/x\}$ we simply have

$$\{t/x\}^{A,\alpha} = \alpha_{x}^{t^{A,\alpha}}$$

Induction

The equation $t(\theta\sigma) = (t\theta)\sigma$ in (b) above can be proved by induction on terms. By this we mean the following proof principle. Let $P(t)$ be a statement about terms $t$ in (b) above we have for example $P(t) : \iff t(\theta\sigma) = (t\theta)\sigma$. In order to prove that $P(t)$ holds for all terms $t$ one has to prove the following.
- **Induction base.** $P(t)$ holds for all atomic terms, i.e. variables and constants.

- **Induction step.** If $P(t_1), \ldots, P(t_n)$ hold (induction hypothesis), then also $P(f(t_1, \ldots, t_n))$ holds.

Let us use this principle to prove that $t(\theta \sigma) = (t \theta) \sigma$ for all terms $t$.

(i) **Induction base (variables).** $x(\theta \sigma)(x) = (x \theta) \sigma$

(ii) **Induction base (constants).** $c(\theta \sigma) = c = (c \theta) \sigma$

(iii) **Induction step.** The induction hypothesis is $t_i(\theta \sigma) = (t_i \theta) \sigma$ for $i = 1, \ldots, n$.

$$f(t_1, \ldots, t_n)(\theta \sigma) = f(t_1(\theta \sigma), \ldots, t_n(\theta \sigma))$$

$$= f((t_1 \theta) \sigma, \ldots, (t_n \theta) \sigma) \quad \text{by induction hypothesis}$$

$$= f(t_1, \ldots, t_n) \theta \sigma$$

$$= (f(t_1, \ldots, t_n) \theta) \sigma$$

2.4.2 Exercise

Let $\theta: Y \to \mathbb{T}(\Sigma, X)$ be a substitution. Note that $\theta$ can also be viewed as a variable assignment in the term algebra $\mathbb{T}(\Sigma, X)$ (see definition 2.2.7). Prove by induction on terms that for any term $t \in \mathbb{T}(\Sigma, Y)$

$$t_{\mathbb{T}(\Sigma, X), \theta} = t \theta$$

2.4.3 Example

Let $\Sigma$ and $A$ be as in example 2.1.4 and $X := \{x, y\}$. Define the substitution $\theta: X \to \mathbb{T}(\Sigma, X)$ by $\theta := \{\text{add}(y, y)/x\}$, i.e. $\theta(x) = \text{add}(0, y)$, $\theta(y) = y$. Now we have for example

$$\text{add}(x, x) \theta = \text{add}(\text{add}(y, y), \text{add}(y, y))$$

Define a assignment $\alpha: X \to A$ by $\alpha(x) := 3$, $\alpha(y) := 7$. Now $\theta^{A, \alpha}: X \to A$ is a variable assignment with

$$\theta^{A, \alpha}(x) = \text{add}(y, y)^{A, \alpha} = 7 + 7 = 14$$

$$\theta^{A, \alpha}(y) = y^{A, \alpha} = 7$$

Now for example

$$\text{add}(x, x)^{A, (\theta^{A, \alpha})} = 14 + 14 = 28.$$ 

Note that also

$$(\text{add}(x, x) \theta)^{A, \alpha} = \text{add}(\text{add}(y, y), \text{add}(y, y))^{A, \alpha} = (7 + 7) + (7 + 7) = 28.$$ 

The following lemma proves that the observation above is not a coincidence.
2.4.4 Substitution lemma for terms

Let \( A \) be an algebra and \( X \) a set of variables, both for a signature \( \Sigma = (S, \Omega) \).

For any term \( t \in T(\Sigma, X) \), substitution \( \theta : X \rightarrow T(\Sigma, Y) \) and variable assignment \( \alpha : Y \rightarrow A \)
\[
(t\theta)^{A,\alpha} = t^{A,\theta^{A,\alpha}}
\]

For a substitution \( \{r/x\} \) this means
\[
t\{r/x\}^{A,\alpha} = t^{A,\alpha}_{x}^{A,\alpha}
\]

**Proof.** The equation is proved by induction on terms.

1. **Induction base (variables).** \((x\theta)^{A,\alpha} = \theta(x)^{A,\alpha} = x^{A,\theta^{A,\alpha}}\)

2. **Induction base (constants).** \((c\theta)^{A,\alpha} = c^{A,\alpha} = c^{A,\theta^{A,\alpha}}\)

3. **Induction step.** The induction hypothesis is \((t_{i}\theta)^{A,\alpha} = t_{i}^{A,\theta^{A,\alpha}}\) for \( i = 1, \ldots, n \).

\[
(f(t_{1}, \ldots, t_{n})\theta)^{A,\alpha} = f(t_{1}\theta, \ldots, t_{n}\theta)^{A,\alpha} = f^{A}(t_{1}\theta)^{A,\alpha}, \ldots, (t_{n}\theta)^{A,\alpha}) = f^{A}(t_{1}^{\alpha}, \ldots, t_{n}^{\alpha}) \quad \text{by induction hypothesis}
\]

2.4.5 Definition (Applying substitutions to formulas)

Let \( \theta : X \rightarrow T(\Sigma, Y) \) be a substitution and \( P \) a formula with \( \text{FV}(P) \subseteq X \). The intuitive definition of applying the \( \theta \) to \( A \) is

\[
P\theta := \text{the result of replacing every free occurrence of a variable } x \text{ in } P \text{ by } \theta(x),
\]
possibly renaming the bound variables of \( P \) in order to avoid variable clashes

So, for example, \((\forall y(x + y = y))\{y/x\}\) is \((\forall z(y + z = z))\{y/x\}\) and *not* \((\forall y(y + y = y))\{y/x\}\).

\(P\theta\) can be defined more precisely by recursion on \( P\)

**Exercise:** Carry this out.

2.4.6 Substitution lemma for formulas

Let \( A \) be an algebra and \( X \) a set of variables, both for a signature \( \Sigma = (S, \Omega) \).

For any formula \( P \) with \( \text{FV}(P) \subseteq X \), substitution \( \theta : X \rightarrow T(\Sigma, Y) \) and variable assignment \( \alpha : Y \rightarrow A \)
\[
A, \alpha \models P\theta \iff A, \alpha^{A,\alpha} \models P
\]

For a substitution \( \{r/x\} \) this means
\[
A, \alpha \models P\{r/x\} \iff A, \alpha^{x}_{x} \models P
\]

**Proof.** Induction on \( P \).
(i) The case that $P$ is $\bot$ is trivial.

(ii) Case $P$ is an equation $t_1 = t_2$. Note that $(t_1 = t_2)\theta$ is the formula $t_1 \theta = t_2 \theta$. By the substitution lemma for terms we have

$$ (t_1 \theta)^A,\alpha = t_i^{A,(\theta^{A,\alpha})} $$

for $i = 1, 2$. Therefore

$$ A, \alpha \models (t_1 = t_2)\theta \iff A, \alpha \models t_1 \theta = t_2 \theta \iff (t_1 \theta)^A,\alpha = (t_2 \theta)^A,\alpha \iff t_1^{A,(\theta^{A,\alpha})} = t_2^{A,(\theta^{A,\alpha})} \iff A, \theta^{A,\alpha} \models t_1 = t_2 $$

(iii) The induction steps are easy and left to the reader.

2.4.7 Lemma (Replacing equals by equals)

Let $r, r'$ be terms and $x$ a variable, all of the same sort, let $A$ be an algebra and $\alpha$ a variable assignment. If

$$ r^{A,\alpha} = r'^{A,\alpha} $$

then for every term $t$ and every formula $P$

$$ t\{r/x\}^{A,\alpha} = t\{r'/x\}^{A,\alpha} $$

$$ A, \alpha \models P\{r/x\} \iff A, \alpha \models P\{r'/x\} $$

**Proof.** Assume $r^{A,\alpha} = r'^{A,\alpha}$. By the substitution lemma for formulas, 2.4.4, we have

$$ t\{r/x\}^{A,\alpha} = t\{r'/x\}^{A,\alpha} = A, \alpha_x^{A,\alpha} = A, \alpha_x'^{A,\alpha} $$

Similarly, by the substitution lemma for formulas, 2.4.4, we have

$$ A, \alpha \models P\{r/x\} \iff A, \alpha_x^{A,\alpha} \models P \iff A, \alpha_x'^{A,\alpha} \models P \iff A, \alpha \models P\{r'/x\} $$
3 Proofs

We are now going to study a proof calculus for deriving facts of the form

$$\Gamma \models \mathcal{P}$$

In order to motivate the rules of this calculus we make some simple observations about the relation $\Gamma \models \mathcal{P}$.

In some cases it is very easy to prove that $\Gamma \models \mathcal{P}$ holds, i.e. that $\mathcal{P}$ is a logical consequence of $\Gamma$. For example:

$$\Gamma \models \mathcal{P} \quad \text{if } \mathcal{P} \text{ is an element of } \Gamma$$

We also have

(1) $\{\mathcal{P},\mathcal{Q}\} \models \mathcal{P} \land \mathcal{Q}$

and

(2) $\{\mathcal{P} \land \mathcal{Q}\} \models \mathcal{P}$

A useful method to obtain new statements of the form $\Gamma \models \mathcal{P}$ from old ones is given by the following rule:

Transitivity of logical consequence

If $\Gamma \models \mathcal{P}_1$ and ... and $\Gamma \models \mathcal{P}_n$ and $\{\mathcal{P}_1,\ldots,\mathcal{P}_n\} \models \mathcal{Q}$ then $\Gamma \models \mathcal{Q}$

Proof. Assume $\Gamma \models \mathcal{P}_i$ for $i = 1,\ldots,n$. We have to show $\Gamma \models \mathcal{Q}$. By definition we have to show that $\mathcal{P}$ is true in all models of $\Gamma$. So, let $\mathcal{A}$ be an arbitrary model of $\Gamma$. We have to show that $\mathcal{P}$ is true in $\mathcal{A}$. Since we assumed $\Gamma \models \mathcal{P}_i$ we may conclude that all $\mathcal{P}_i$ are true in $\mathcal{A}$, that is, $\mathcal{A}$ is a model of $\{\mathcal{P}_1,\ldots,\mathcal{P}_n\}$. Using the assumption $\{\mathcal{P}_1,\ldots,\mathcal{P}_n\} \models \mathcal{Q}$ we conclude that $\mathcal{Q}$ is true in $\mathcal{A}$.

Using transitivity of logical consequence we immediately obtain from (1) and (2) above the following rules:

If $\Gamma \models \mathcal{P}$ and $\Gamma \models \mathcal{Q}$ then $\Gamma \models \mathcal{P} \land \mathcal{Q}$

If $\Gamma \models \mathcal{P} \land \mathcal{Q}$ then $\Gamma \models \mathcal{P}$

Another, less trivial, rule is

If $\Gamma \cup \{\mathcal{P}\} \models \mathcal{Q}$ then $\Gamma \models \mathcal{P} \rightarrow \mathcal{Q}$. 
For verifying this rule we cannot use transitivity of consequence. Instead we may argue as follows: Assume \( \Gamma \cup \{ P \} \models Q \). We have to show \( \Gamma \models P \rightarrow Q \). Let \( A \) be a model of \( \Gamma \). We have to show that \( P \rightarrow Q \) is true in \( A \). By definition, we have to prove

\[
\text{if } P \text{ is true in } A \text{ then } Q \text{ is true in } A
\]

So, let us assume that \( P \) is true in \( A \). We have to show that \( Q \) is true in \( A \). Because \( A \) is a model of \( \Gamma \) this means that \( A \) is a model of \( \Gamma \cup \{ P \} \). Because \( \Gamma \cup \{ P \} \models Q \), by assumption, it follows that \( Q \) is true in \( A \).

The main result in this chapter, due to K Gödel, will be that a bunch of simple rules, similar to the rules above, suffices to derive all true statements of the form \( \Gamma \models P \). In particular all tautologies can be derived.

### 3.1 Natural deduction rules for propositional connectives

#### 3.1.1 Definition (Sequent)

If \( \Gamma \) is a finite set of formulas and \( P \) a formula then

\[
\Gamma \vdash P
\]

is called a **sequent**. The set \( \Gamma \) is called **context**, or **antecedent** of the sequent. The elements of the context are called **assumptions**. The formula \( P \) is called **succedent** of the sequent.

In the following we will write \( P_1, \ldots, P_n \) for the set \( \{ P_1, \ldots, P_n \} \). We also write \( \Gamma, P \) for \( \Gamma \cup \{ P \} \).

#### 3.1.2 The assumption rule

\[
\text{Assumption} \quad \frac{}{\Gamma, P \vdash P}^{\text{use}}
\]

The assumption rule allows us to begin a formal proof. Another way to express this rule would be to say that we can derive \( \Gamma \vdash P \) if \( P \) is an element of \( \Gamma \).

Using the following rules for the logical connectives \( \land, \rightarrow, \lor \), one can build complex formal proofs. For each connective there are **introduction** and **elimination** rules.

#### 3.1.3 Rules for conjunction

**Conjunction Introduction**

\[
\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \land Q}^{\land^+}
\]

**Conjunction Elimination**

\[
\frac{\Gamma \vdash P \land Q}{\Gamma \vdash P}^{\land_1} \quad \frac{\Gamma \vdash P \land Q}{\Gamma \vdash Q}^{\land_2}
\]
A derivation (also called formal proof) of a sequent \( \Gamma \vdash P \) consists of a tree of sequents built according to these rules whose root is \( \Gamma \vdash P \). We also call such a derivation a derivation of \( P \) from assumptions \( \Gamma \).

### 3.1.4 Examples and abbreviations

The following is a derivation of the sequent \( P \land Q \vdash P \land Q \).

\[
\begin{align*}
\frac{P \land Q \vdash P \land Q}{P \land Q \vdash Q} & \quad \text{use } \land T \\
\frac{P \land Q \vdash P \land Q}{P \land Q \vdash P} & \quad \text{use } \land I \\
\frac{P \land Q \vdash Q \land P}{P \land Q \vdash P \land Q} & \quad \land^+
\end{align*}
\]

To improve readability we write this derivation in the following short form

\[
\begin{align*}
\frac{P \land Q}{Q} & \quad \land T \\
\frac{P \land Q}{P} & \quad \land I \\
\frac{Q \land P}{Q \land P} & \quad \land^+
\end{align*}
\]

i.e. we only write down the succedents of the sequents and omit the bar of the assumption rule. Another example of a derivation, written in short form, is

\[
\begin{align*}
\frac{P \land Q}{P} & \quad \land T \\
\frac{P \land Q}{Q} & \quad \land I \\
\frac{Q \land Q \land R}{Q \land Q \land R} & \quad \land^+
\end{align*}
\]

To reconstruct the contexts of the sequents in this derivation we collect the formulas attached to the leaves of this derivation tree. We obtain the context \( \{P \land Q, R\} \). Therefore the full derivation reads:

\[
\begin{align*}
\frac{P \land Q, R \vdash P \land Q}{P \land Q, R \vdash P} & \quad \text{use } \land T \\
\frac{P \land Q, R \vdash P \land Q}{P \land Q, R \vdash Q} & \quad \text{use } \land I \\
\frac{P \land Q, R \vdash R}{P \land Q, R \vdash P \land Q} & \quad \land^+
\end{align*}
\]

We see that this is a derivation of the sequent \( P \land Q, R \vdash P \land (Q \land R) \).

### 3.1.5 Rules for implication

**Implication Introduction**

\[
\begin{align*}
\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \rightarrow Q} & \quad \rightarrow^+
\end{align*}
\]

**Implication Elimination**

\[
\begin{align*}
\frac{\Gamma \vdash P \rightarrow Q \quad \Gamma \vdash P}{\Gamma \vdash Q} & \quad \rightarrow^-
\end{align*}
\]
3.1.6 Examples and abbreviations

Let us derive the sequent $\Gamma \vdash Q \rightarrow R$, where $\Gamma := \{ P \land Q \rightarrow R, P \}$

\[
\begin{align*}
\Gamma, Q \vdash P \land Q & \rightarrow R \quad \text{use} \\
\Gamma, Q \vdash P & \quad \text{use} \\
\Gamma, Q \vdash Q & \quad \text{use} \\
\hline
\Gamma, Q \vdash P & \rightarrow^+ \\
\Gamma, Q \vdash Q & \rightarrow^+ \\
\hline
\Gamma, Q \vdash P \land Q & \rightarrow^+ \\
\Gamma, Q \vdash R & \rightarrow^+ \\
\hline
\Gamma \vdash Q \rightarrow R & \rightarrow^+ \\
\end{align*}
\]

In order to write this derivation in a nice short form we introduce a label, $u : Q$, for the assumption $Q$ removed from the context in the last step where the rule $\rightarrow^+$ was applied.

\[
\begin{align*}
P \land Q & \rightarrow R \\
\hline
\Gamma, Q \vdash P & \rightarrow^+ \\
\Gamma, Q \vdash Q & \rightarrow^+ \\
\hline
\Gamma \vdash R & \rightarrow^+ \\
\Gamma \vdash Q & \rightarrow^+ \\
\hline
P & \rightarrow^+ u : Q \\
\end{align*}
\]

Next let us derive $P \land Q \rightarrow R \vdash P \rightarrow (Q \rightarrow R)$. The easiest way to find the derivations is to construct it “bottom up”.

\[
\begin{align*}
P \land Q & \rightarrow R \\
\hline
\Gamma, Q \vdash P & \rightarrow^+ \\
\Gamma, Q \vdash Q & \rightarrow^+ \\
\hline
\Gamma \vdash R & \rightarrow^+ v : Q \\
\Gamma \vdash (Q \rightarrow R) & \rightarrow^+ u : P \\
\end{align*}
\]

To reconstruct the contexts of the sequents in this derivation we again collect the formulas attached to the leaves of this derivation tree. We obtain the context $\Gamma := \{ P \land Q \rightarrow R \}$ (as expected), and the full derivation reads:

\[
\begin{align*}
\Gamma, P, Q \vdash P \land Q & \rightarrow R \quad \text{use} \\
\Gamma, P, Q \vdash P & \quad \text{use} \\
\Gamma, P, Q \vdash Q & \quad \text{use} \\
\hline
\Gamma, P, Q \vdash P \land Q & \rightarrow^+ \\
\Gamma, P, Q \vdash R & \rightarrow^+ \\
\hline
\Gamma, P \vdash R & \rightarrow^+ \\
\Gamma \vdash P \rightarrow (Q \rightarrow R) & \rightarrow^+ \\
\end{align*}
\]

We see that this is a derivation of the sequent $P \land Q \rightarrow R \vdash P \rightarrow (Q \rightarrow R)$.

3.1.7 Rules for disjunction

**Disjunction Introduction**

\[
\begin{align*}
\Gamma \vdash P & \quad \vee_1^+ \\
\Gamma \vdash Q & \quad \vee_2^+ \\
\Gamma \vdash P \lor Q & \quad \vee^+ \\
\end{align*}
\]

**Disjunction Elimination**

\[
\begin{align*}
\Gamma \vdash P \lor Q & \\
\Gamma \vdash P & \rightarrow R \\
\Gamma \vdash Q \rightarrow R & \rightarrow^- \\
\end{align*}
\]
3.1.8 Examples and abbreviations

We derive the sequent \( P \lor Q \vdash Q \lor P \) using our short notation.

\[
\begin{align*}
P \lor Q & \quad \frac{u}{P \to Q \lor P} \quad P \lor Q \vdash u : P \\
& \quad \frac{v}{Q \lor P} \quad Q \lor P \vdash v : Q
\end{align*}
\]

Here’s the full derivation:

\[
\begin{align*}
P \lor Q \vdash P \lor Q & \quad \text{use} \\
& \quad \frac{P \lor Q, Q \vdash P}{P \lor Q \lor Q \to Q \lor P} \quad \frac{u}{P \lor Q \vdash u : P} \\
& \quad \frac{P \lor Q, Q \vdash Q}{P \lor Q \lor P} \quad \frac{v}{Q \lor P} \quad Q \lor P \vdash v : Q
\end{align*}
\]

Next we derive (one half of) one of de-Morgan’s laws, \( P \land (Q \lor R) \to (P \land Q) \lor (P \land R) \), without assumptions.

\[
\begin{align*}
P \land Q & \quad \frac{u}{P \land Q} \quad \frac{v}{v^+} \\
& \quad \frac{w}{w^+} \quad \frac{(P \land Q) \lor (P \land R) \vdash u : P \land (Q \lor R)}{Q \to (P \land Q) \lor (P \land R)} \quad \frac{Q \vdash (P \land Q) \lor (P \land R)}{Q \to (P \land Q) \lor (P \land R)} \quad \frac{(P \land Q) \lor (P \land R) \vdash v : R}{Q \to (P \land Q) \lor (P \land R)} \quad \frac{(P \land Q) \lor (P \land R) \vdash w : R}{Q \to (P \land Q) \lor (P \land R)}
\end{align*}
\]

3.1.9 Rules for falsity

**Ex-falso-quodlibet**

\[
\begin{align*}
\Gamma \vdash \bot & \quad \text{efl} \\
\Gamma \vdash P
\end{align*}
\]

**Reductio-ad-absurdum**

\[
\begin{align*}
\Gamma \vdash \neg P & \quad \text{raa} \\
\Gamma \vdash P
\end{align*}
\]

Recall that \( \neg P \) is shorthand for \( P \to \bot \), and therefore \( \neg \neg P \) stands for \( (P \to \bot) \to \bot \).

3.1.10 Example

We derive Peirce’s law \( P \to Q \to P \vdash P \) using short notation.
3.2 Quantifier rules

In the formulation of the rules for the quantifiers \( \forall \) and \( \exists \) we write \( P(x) \) for a formula that possibly contains \( x \) free. The substitution of a term \( t \) for \( x \) is then written \( P(t) \). In other words, instead of the official notation \( P \) and \( P\{t/x\} \) we use the slightly imprecise, but more suggestive notation \( P(x) \) and \( P(t) \), where it is implicitly assumed that the variable \( x \) and the term \( t \) are of the same sort.

3.2.1 Rules for universal quantification

\[ \forall \text{- introduction} \quad \frac{\Gamma \vdash P(x)}{\Gamma \vdash \forall x P(x)} \quad \text{\( \forall^+ \) provided \( x \) is not free in \( \Gamma \) (variable condition)} \]

\[ \forall \text{- elimination} \quad \frac{\Gamma \vdash \forall x P(x)}{\Gamma \vdash P(t)} \quad \text{\( \forall^- \)} \]

3.2.2 Example

We derive \( \forall x P(x) \vdash \forall y P(y + 1) \) (w.l.o.g. we may assume that \( y \) is not free in \( P(x) \)):

\[ \frac{\forall x P(x)}{P(y + 1)} \quad \text{\( \forall^- \)} \]

\[ \frac{\forall y P(y + 1)}{\forall x P(x) \vdash \forall y P(y + 1)} \quad \text{\( \forall^+ \)} \]

Here’s the full proof:

\[ \frac{\forall x P(x) \vdash \forall x P(x) \quad \text{use}}{\forall x P(x) \vdash P(y + 1)} \quad \text{\( \forall^- \)} \]

\[ \frac{\forall x P(x) \vdash \forall y P(y + 1)}{\forall x P(x) \vdash \forall y P(y + 1)} \quad \text{\( \forall^+ \)} \]

We see that in the application of \( \forall^+ \) the variable condition is satisfied, because \( y \) is not free in \( \forall x P(x) \).
3.2.3 Exercise

Find out what’s wrong with the following two ‘derivations’.

\[
\dfrac{\forall y(x < y + 1) \quad 0}{x < 1 + 0} \quad \forall^-
\]
\[
\dfrac{\forall x(x < 1 + 0)}{\forall x(x < 1 + 0)} \quad \forall^+
\]

\[
\dfrac{\forall x(\forall y(x < y + 1) \rightarrow x = 0) \quad y}{\forall y(y < y + 1) \rightarrow y = 0} \quad \forall^-(y < y + 1) \rightarrow
\]
\[
\dfrac{y = 0}{\forall y(y = 0)} \quad \forall^+
\]

3.2.4 Rules for existential quantification

\[
\exists-\text{introduction} \quad \dfrac{\Gamma \vdash P(t)}{\Gamma \vdash \exists x \ P(x)} \quad \forall^+
\]

\[
\exists-\text{elimination} \quad \dfrac{\Gamma \vdash \exists x \ P(x) \quad \Gamma \vdash \forall x \ (P(x) \rightarrow Q)}{\Gamma \vdash Q} \quad \exists^- \quad \text{provided } x \text{ is not free in } Q
\]

3.2.5 Examples

\[
(x - 1) + 1 = x \vdash \exists y(y + 1 = x)
\]
\[
\dfrac{(x - 1) + 1 = x}{\exists y(y + 1 = x)} \quad \exists^+
\]

\[
\exists x \ P(x), \forall x \ (P(x) \rightarrow Q(f(x))) \vdash \exists y \ Q(y)
\]
\[
\dfrac{\forall x \ (P(x) \rightarrow Q(f(x)) \quad x}{P(x) \rightarrow Q(f(x))} \quad \forall^- \quad u \rightarrow
\]
\[
\dfrac{Q(f(x)) \quad \exists^+}{\exists y \ Q(y)} \quad \rightarrow^+ u: P(x)
\]
\[
\dfrac{P(x) \rightarrow \exists y \ Q(y) \quad \forall^+}{\exists x \ P(x)} \quad \forall^-
\]
\[
\dfrac{\forall x \ (P(x) \rightarrow \exists y \ Q(y)) \quad \forall^+}{\exists y \ Q(y)} \quad \exists^-
\]
We see that in the application of $\exists^-$ the variable condition is satisfied, because $x$ is not free in $\exists y Q(y)$.

In figure 3 on page 29 the rules of natural deduction are summarized leaving out rules for equality which will be treated next. In the short version of the $\forall$-introduction rule the square brackets indicate that every occurrence of the assumption $P$ is canceled and replaced by the (new) label $u$. The (*) indicates the appropriate variable condition explained in 3.2.1 and 3.2.4.

### 3.3 Equality rules

**Reflexivity**

$$\Gamma \vdash t = t \text{ refl}$$

**Symmetry**

$$\Gamma \vdash s = t \text{ sym}$$

**Transitivity**

$$\Gamma \vdash r = s \quad \Gamma \vdash s = t \quad \Gamma \vdash r = t \text{ trans}$$

**Compatibility**

$$\Gamma \vdash s_1 = t_1 \quad \ldots \quad \Gamma \vdash s_n = t_n \text{ comp}$$

$$\Gamma \vdash f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$$

for every operation $f$ of $n$ arguments.

### 3.3.1 Lemma

Let $t(x)$ be a term possibly containing the variable $x$, and let $r, s$ be terms of the same sort as $x$. Then the following sequent is derivable.

$$r = s \vdash t(r) = t(s)$$

**Proof.** Induction on $t(x)$.

If $t(x)$ is a constant or a variable different from $x$, then $t(r)$ and $t(r)$ are the same term $t$. Hence the assertion is $r = s \vdash t(t) = t$ which is an instance of the reflexivity rule.

If $t(x)$ is the variable $x$ then $t(r)$ is $r$ and $t(s)$ is $s$, and the assertion becomes $r = s \vdash r = s$ which is an instance of the assumption rule.

Finally, consider $t(x)$ of the form $f(t_1(x), \ldots, t_n(x))$. By induction hypothesis we may assume that we already have a derivation of $r = s \vdash t_i(r) = t_i(s)$ for $i = 1, \ldots, n$. One application of the compatibility rule yields the required sequent.
<table>
<thead>
<tr>
<th></th>
<th>Full version</th>
<th>Short version</th>
</tr>
</thead>
<tbody>
<tr>
<td>use</td>
<td>$\Gamma, P \vdash P$</td>
<td>$P$</td>
</tr>
<tr>
<td>$\land^+$</td>
<td>$\Gamma \vdash P \quad \Gamma \vdash Q \quad \Gamma \vdash P \land Q$</td>
<td>$\frac{P \land Q}{\Gamma \vdash P \land Q} \quad \land^+$</td>
</tr>
<tr>
<td>$\land^-$</td>
<td>$\Gamma \vdash P \land Q \quad \Gamma \vdash P \quad \Gamma \vdash Q$</td>
<td>$\frac{P \land Q}{\Gamma \vdash P} \quad \frac{P \land Q}{\Gamma \vdash Q} \quad \land^-$</td>
</tr>
<tr>
<td>$\rightarrow^+$</td>
<td>$\Gamma, P \vdash Q \quad \Gamma \vdash P \rightarrow Q$</td>
<td>$\frac{P}{\Gamma \vdash P \rightarrow Q} \quad \rightarrow^+ u \vdash P$</td>
</tr>
<tr>
<td>$\rightarrow^-$</td>
<td>$\Gamma \vdash P \rightarrow Q \quad \Gamma \vdash P$</td>
<td>$\frac{P \rightarrow Q}{\Gamma \vdash P} \quad \rightarrow^- P$</td>
</tr>
<tr>
<td>$\forall^+$</td>
<td>$\Gamma \vdash P \quad \forall^+_1 \frac{\Gamma \vdash Q}{\Gamma \vdash P \lor Q} \quad \forall^+_1 \frac{P \lor Q}{\Gamma \vdash P} \quad \forall^+_1$</td>
<td>$\frac{P}{\Gamma \vdash P \lor Q} \quad \frac{Q}{\Gamma \vdash P \lor Q} \quad \forall^+_1$</td>
</tr>
<tr>
<td>$\forall^-$</td>
<td>$\Gamma \vdash P \lor Q \quad \Gamma \vdash P \rightarrow R \quad \Gamma \vdash Q \rightarrow R$</td>
<td>$\frac{P \lor Q}{\Gamma \vdash P \lor Q} \quad \frac{P \lor Q}{\Gamma \vdash P \lor Q} \quad \forall^-$</td>
</tr>
<tr>
<td>$\bot$</td>
<td>$\Gamma \vdash \bot$</td>
<td>$\frac{\bot}{\Gamma \vdash P} \quad \frac{\bot \vdash P}{\Gamma \vdash P} \quad \bot$</td>
</tr>
<tr>
<td>$\forall^+$</td>
<td>$\Gamma \vdash P(x) \quad \forall^+_1 \frac{\forall x P(x)}{\Gamma \vdash P(x)} \quad (*)$</td>
<td>$\frac{P(x)}{\forall x P(x)} \quad (*)$</td>
</tr>
<tr>
<td>$\forall^-$</td>
<td>$\Gamma \vdash \forall x P(x) \quad t \quad \forall^- \frac{\forall x P(x)}{\Gamma \vdash P(t)} \quad (*)$</td>
<td>$\frac{\forall x P(x)}{\Gamma \vdash P(t)} \quad (*)$</td>
</tr>
<tr>
<td>$\exists^+$</td>
<td>$\Gamma \vdash P(t) \quad \exists^+_1 \frac{\exists x P(x)}{\Gamma \vdash P(x)} \quad (*)$</td>
<td>$\frac{P(t)}{\exists x P(x)} \quad (*)$</td>
</tr>
<tr>
<td>$\exists^-$</td>
<td>$\Gamma \vdash \exists x P(x) \quad \forall x (P(x) \rightarrow Q)$</td>
<td>$\frac{\exists x P(x) \quad \forall x (P(x) \rightarrow Q)}{\Gamma \vdash Q} \quad (*)$</td>
</tr>
</tbody>
</table>

Figure 3: The rules of natural deduction (without equality rules)
3.3.2 Lemma

Let \( P(x) \) be a formula possibly containing the variable \( x \), and let \( r, s \) be terms of the same sort as \( x \). Then the following sequent is derivable.

\[
r = s \vdash P(r) \leftrightarrow P(s)
\]

**Proof.** Induction on the formula \( P(x) \).

If \( P(x) \) is an equation, say, \( t_1(x) = t_2(x) \), then we have to derive

\[
r = s \vdash t_1(r) = t_2(r) \leftrightarrow t_1(s) = t_2(s)
\]

By Lemma 3.3.1 we have already derivations of

\[
r = s \vdash t_1(r) = t_1(s) \quad \text{and} \quad r = s \vdash t_2(r) = t_2(s)
\]

It is now easy to obtain the required derivation using the symmetry rule and the transitivity rule. We leave this as an exercise to the reader.

If \( P(x) \) is a compound formula we can use the induction hypothesis in a straightforward way.

3.3.3 Definition

\[
\Gamma \vdash_c P : \iff \Gamma \vdash P \text{ is derivable.}
\]

(\( P \) is derivable from \( \Gamma \) in *classical logic*)

\[
\Gamma \vdash_l P : \iff \Gamma \vdash P \text{ is derivable without using reductio ad absurdum.}
\]

(\( P \) is derivable from \( \Gamma \) in *intuitionistic logic*)

\[
\Gamma \vdash_m P : \iff \Gamma \vdash P \text{ is derivable using neither r.a.a. nor ex-falso-quodlibet.}
\]

(\( P \) is derivable from \( \Gamma \) in *minimal logic*)

If \( \Gamma \) is infinite we define

\[
\Gamma \vdash_c (\vdash_l (\vdash_m)P \quad : \iff \Gamma_0 \vdash_c (\vdash_l (\vdash_m)P \text{ for some finite } \Gamma_0 \subseteq \Gamma
\]

3.4 Soundness and completeness

The soundness and completeness theorems below state that the logical inference rules introduced above precisely capture the notion of logical consequence.
3.4.1 Soundness Theorem

If $\Gamma \vdash_c P$ then $\Gamma \models P$.

**Proof.** The theorem follows immediately from the following statement which can be easily shown by induction on derivations:

For every finite set of (not necessarily closed) formulas $\Gamma$ and every formula $P$,

$$
\text{if } \Gamma \vdash_c P \text{ then } A, \alpha \models P \text{ for all algebras } A \text{ and variable assignments } \alpha \text{ such that } A, \alpha \models \Gamma
$$

Whilst the soundness theorem is not very surprising, because it just states that the inference rules are correct, the following completeness theorem proved by K Gödel in 1937, states that the logical inference rules above in fact capture all possible ways of correct reasoning.

3.4.2 Completeness Theorem (K Gödel)

If $\Gamma \models P$ then $\Gamma \vdash_c P$.

In words: If $P$ is a logical consequence of $\Gamma$ (i.e. $P$ is true in all models of $\Gamma$), then this can be formally derived by the inference rules of natural deduction.

The following reformulation of the completeness theorem refers to the notion of consistency.

3.4.3 Definition (Consistency)

A (possibly infinite) set of formulas $\Gamma$ is called **consistent** if $\Gamma \not\vdash_c \bot$, that is there is no finite subset $\Gamma_0 \subseteq \Gamma$ such that $\Gamma_0 \vdash_c \bot$ is derivable.

3.4.4 Satisfiability Theorem

Every consistent set of formulas has a model.

**Exercise.** Prove the Satisfiability Theorem from the Completeness Theorem and vice versa.

3.5 Axioms and rules for data types

For many common data types we can formulate axioms describing their characteristic features. We will only treat the booleans and the (unary) natural numbers. Similar axioms could be stated for binary number, lists, finite trees etc., more generally for freely generated data types.
3.5.1 Axioms for the booleans

The variable $x$ below is supposed to be of sort boole.

**Boole 1** \[ \Gamma \vdash T \neq F \text{ boole1} \]

**Boole 2** \[ \Gamma \vdash \forall x \ (x = T \lor x = F) \text{ boole2} \]

Recall that $r \neq s$ is an abbreviation for $\neg r = s$ which in turn stands for $r = s \rightarrow \bot$. Recall also that we agreed to abbreviate an equation $t = T$ by $t$.

3.5.2 Lemma

We can derive $\forall x (\neg x \leftrightarrow x = F)$ without assumptions.

**Proof.** We have to derive $\forall x ((x = T \rightarrow \bot) \leftrightarrow x = F)$. Obviously it suffices to derive

\[ \forall x ((x = T \rightarrow \bot) \rightarrow x = F) \quad \text{and} \quad \forall x (x = F \rightarrow (x = T \rightarrow \bot)) \]

We leave these derivations as an exercise to the reader.

3.5.3 Peano Axioms

The following axioms and rules where set up (in a slightly different form) by G Peano to describe the structure of natural numbers with zero and the successor function (we write $t + 1$ for the successor of $t$).

In the following the terms $s, t$ and the variable $x$ are supposed to be of sort nat.

**Peano 1** \[ \Gamma \vdash 0 \neq t + 1 \text{ peano1} \]

**Peano 2** \[ \Gamma \vdash s + 1 = t + 1 \rightarrow s = t \text{ peano2} \]

**Induction** \[ P(0) \quad \forall x (P(x) \rightarrow P(x + 1)) \quad \text{ind} \]

\[ \forall x P(x) \]

3.5.4 Remark

In applications there will be further axioms describing additional operations on the booleans and natural numbers. Examples are, the equations defining addition by primitive recursion from zero and the successor function.
4 Programs from proofs

We now study how proofs can give rise to provably correct programs. We shall follow the idea of the so-called

Curry-Howard Correspondence

according to which

formulas correspond to data types

and

proofs correspond to programs.

4.1 Formulas as data types

The following table show how each constructor for formulas corresponds to a data type construction. We assume that the variable $x$ below is of sort $s$.

<table>
<thead>
<tr>
<th>Formula</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conjunction $P \land Q$</td>
<td>$P \times Q$ cartesian product</td>
</tr>
<tr>
<td>implication $P \rightarrow Q$</td>
<td>$P \rightarrow Q$ function type</td>
</tr>
<tr>
<td>disjunction $P \lor Q$</td>
<td>$P + Q$ disjoint sum</td>
</tr>
<tr>
<td>for all $\forall x P(x)$</td>
<td>$s \rightarrow P$ function type</td>
</tr>
<tr>
<td>exists $\exists x P(x)$</td>
<td>$s \times P$ cartesian product</td>
</tr>
<tr>
<td>equations $s = t$</td>
<td>${s}$ a singleton set</td>
</tr>
<tr>
<td>falsity $\bot$</td>
<td>${}$ the empty set</td>
</tr>
</tbody>
</table>

Figure 4: The formulas-as-types correspondence
4.1.1 Example

Assume the variables $x, y$ are of sort nat. Then the formula

$$P : \equiv \forall x \exists y (x = y + y \vee x = y + y + 1)$$

corresponds to the data type

$$\text{nat} \rightarrow \text{nat} \times \{\ast\}$$

Since $\{\ast\}$ is a set with two (different) elements we may replace it by the type boole. Therefore the formula above corresponds to the data type

$$\text{nat} \rightarrow \text{nat} \times \text{boole}$$

The idea is that a proof of $P$ yields a program of that type (that is, an operation that accepts as input a natural number and outputs a pair consisting of a natural number and a boolean value) that realizes the formula $P$, that is, solves the problem naturally associated with $P$. In our example the problem consists in deciding for every natural number $x$ whether it is even or odd and computing the integer half of $x$ (rounded down). So, if on input $x$ the program outputs a pair $\langle y, T \rangle$ this would mean that $x$ is even and $x = y + y$, whereas an output $\langle y, F \rangle$ means that $x$ is odd and $x = y + y + 1$.

4.2 A notation system for proofs

We now introduce for every proof a proof term that will give us the desired program corresponding to the proof as briefly explained in example 4.1.1. The definition of these proof terms is given in tables 5. The names of the term constructors indicate their intended computational meaning. Proof rules and axioms that are not mentioned in these tables have proof terms with a trivial computational meaning.

Induction corresponds to recursion:

<table>
<thead>
<tr>
<th>induction</th>
<th>recursion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d : P(0)$</td>
<td>$e : \forall x (P(x) \rightarrow P(x + 1))$</td>
</tr>
<tr>
<td>$\text{ind}[d, e] : \forall x P(x)$</td>
<td></td>
</tr>
</tbody>
</table>

All other rules do not correspond to computationally meaningful program constructs:

<table>
<thead>
<tr>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_1 : P_1 \ldots d_n : P_n$</td>
</tr>
<tr>
<td>$\text{rule}[d_1, \ldots, d_n]$</td>
</tr>
<tr>
<td>$\text{rule}$</td>
</tr>
<tr>
<td>use</td>
</tr>
<tr>
<td>-----</td>
</tr>
</tbody>
</table>
| \( \land^+ \) | pairing | \[
\frac{d : P}{(d, e) : P \land Q} \land^+
\]
| \( \land^- \) | projections | \[
\begin{align*}
\frac{d : P \land Q}{\pi_1(d) : P} & \land^- \\
\frac{d : P \land Q}{\pi_r(d) : Q} & \land^-
\end{align*}
\]
| \( \to^+ \) | abstraction | \[
\frac{d : Q}{\lambda u : P.d : P \to Q} \to^+ u : P
\]
| \( \to^- \) | procedure call | \[
\frac{d : P \to Q}{(de) : Q} \to^-
\]
| \( \lor^+ \) | injections | \[
\begin{align*}
\frac{d : P}{\text{inl}_P(d) : P \lor Q} & \lor^+ \\
\frac{d : Q}{\text{inr}_P(d) : P \lor Q} & \lor^-
\end{align*}
\]
| \( \lor^- \) | case analysis | \[
\frac{d : P \lor Q}{\text{cases}[d, e_1, e_2] : R} \lor^-
\]
| efq | exception | \[
\frac{d : \bot}{e_{\text{fq}}(d) : P} \text{efq}
\]
| raa | call/cc | \[
\frac{d : \lnot P}{r_{\text{aa}}(d) : P} \text{raa}
\]
| \( \forall^+ \) | abstraction | \[
\frac{d : P(x)}{\lambda x.d : \forall x P(x)} \forall^+ \ (*)
\]
| \( \forall^- \) | procedure call | \[
\frac{d : \forall x P(x)}{(dt) : P(t)} \forall^-
\]
| \( \exists^+ \) | pairing | \[
\frac{d : P(t)}{(d, t) : \exists x P(x)} \exists^+
\]
| \( \exists^- \) | matching | \[
\frac{d : \exists x P(x) \quad e : \forall x (P(x) \to Q)}{\text{match}[d, e] : Q} \exists^- \ (*)
\]

Figure 5: Natural deduction with proof terms
4.2.1 Example

Consider the proof

\[
\frac{u : P \land Q \land \top}{Q} \quad \frac{u : P \land Q}{P} \quad \lambda^+ \quad \frac{u : P \land Q \land \top}{Q \land P} \quad \frac{u : P \land Q}{(P \land Q) \to (Q \land P)} \to^+ u : P \land Q
\]

Written with proof terms this reads:

\[
\frac{\pi_1(u) : Q \land \top}{\langle \pi_1(u), \pi_1(u) \rangle : Q \land P} \quad \lambda^+ \quad \frac{\pi_2(u) : P \land \top}{\langle \pi_1(u), \pi_1(u) \rangle : (P \land Q) \to (Q \land P)} \to^+ \lambda u : P \land Q \langle \pi_1(u), \pi_1(u) \rangle
\]

The complete information about this proof is contained in the proof term

\[
\lambda u : P \land Q \langle \pi_1(u), \pi_1(u) \rangle
\]

4.2.2 Exercises

(a) Find the proof term for the following proof:

\[
\frac{u : P \land Q \to R}{v : P \quad w : Q} \quad \frac{P \land Q}{\to^-} \quad \frac{R}{\to^+ v : P} \quad \frac{Q \to R}{w : Q} \quad \frac{P \to (Q \to R)}{\to^+ v : P} \quad \frac{(P \land Q \to R) \to (P \to (Q \to R))}{\to^+ u : P \land Q \to R}
\]

(b) To which proof does the following proof term correspond?

\[
\lambda u : P \to (Q \to R) \cdot \lambda v : P \land Q \cdot ((u \pi_1(v)) \pi_1(v))
\]
4.3 Program synthesis from proofs

In the previous section we assigned to each proof a certain proof term written in a language very similar to a functional programming language. Indeed it requires only little modifications and simplification to transform a proof term into an executable functional program. Technically this transformation is done via a so-called formalized realizability interpretation. Its main task is to

- give the constructors of the proof terms a computational interpretation,
- delete all parts of the proof term that are computationally meaningless.

The method of program synthesis from proofs is summarized in the following theorem:

4.3.1 Theorem (Program synthesis from constructive proofs)

From every constructive, that is, intuitionistic proof of a formula

\[ \forall x \exists y R(x, y) \]

one can extract a program \( p \) such that

\[ \forall x R(x, p(x)) \]

is provable, that is, \( p \) is provably correct.

The statement of this theorem is a little bit simplified. So, for example instead of single variables \( x \) and \( y \) one may have lists \( \bar{x}, \bar{y} \) of variables, and the variables in \( \bar{x} \) may be subject to preconditions.

4.3.2 Example (Quotient and remainder)

An example of such a generalized formula is

\[ (+) \quad \forall b \ ((b > 0 \rightarrow \forall a \exists q \exists r \ (a = b \times q + r \land r < b)) \]

where the variables range over natural numbers. This formula says that division with rest by a positive number \( b \) is possible for all \( b \). The numbers \( q \) and \( r \) whose existence is claimed are the quotient and the remainder of this division.

According to the theorem above a constructive proof of \((+)\) should yield a program that for inputs \( b \) and \( a \), where \( b > 0 \), computes numbers \( q \) and \( r \) such that

\[ a = b \times q + r \quad \text{and} \quad r < b. \]
We now sketch a proof of \(+\) and show how two extract a program from it. Then a full formal proof will be given and program extraction will be carried out in the interactive proof system MINLOG (http://www....).

In order to prove \(+\) let \(b > 0\) be given (\(\forall^+\) and \(\rightarrow^+\) backwards). We prove

\[
\forall a \exists q \exists r (a = b \cdot q + r \land r < b)
\]

by induction on \(a\).

**Base.** We need to prove \(\exists q \exists r (0 = b \cdot q + r \land r < b)\). But that is easy: take \(q := 0\) and \(r := 0\).

**Step.** We have to prove

\[
\forall a \left[ \exists q \exists r (a = b \cdot q + r \land r < b) \right] \rightarrow \exists q_1 \exists r_1 (a + 1 = b \cdot q_1 + r_1 \land r_1 < b)]
\]

So, let \(a\) be given and assume

**induction hypothesis:** \(\exists q \exists r (a = b \cdot q + r \land r < b)\)

We have to prove \(\exists q_1 \exists r_1 (a + 1 = b \cdot q_1 + r_1 \land r_1 < b)\).

Using the ind. hyp. we may assume we have \(q\) and \(r\) such that

\[
u : a = b \cdot q + r \land r < b
\]

(formally we use \(\exists^\rightarrow\) backwards followed by \(\forall^\rightarrow\) and \(\rightarrow^\rightarrow\) backwards). We need to find \(q_1\) and \(r_1\) such that \(a + 1 = b \cdot q_1 + r_1 \land r_1 < b\) (in order to apply \(\exists^+\)).

**Case \(r + 1 < b\).** Then we can set \(q_1 := q\) and \(r_1 := r + 1\), because from assumption \(u\) it follows that \(a + 1 = b \cdot q + r + 1\).

**Case \(r + 1 \not< b\).** Then, by assumption \(u\), we must have \(r + 1 = b\). We set \(q_1 := q + 1\) and \(r_1 := 0\). This works, because, using \(u\) once more, we obtain \(a + 1 = b \cdot q + r + 1 = b \cdot q + b = b \cdot (q + 1) + 0\).

This ends the proof of the induction step and completes the proof.

Intuitively this proof corresponds roughly to the following program:

```plaintext
function quotrem (b,a:integer, b>0) : integer × integer
begin
  if a=0 then quotrem := (0,0)
  else let (q,r) := quotrem(b,a-1)
    if r<b then quotrem := (q,r+1)
    else quotrem := (q+1,0)
end
```
The program is recursive because the proof was done by induction. More formally, if we have a proof

\[ \frac{d : P(0) \quad e : \forall x \; (P(x) \rightarrow P(x + 1))}{\text{ind}[d, e] : \forall x \; P(x)} \text{ind} \]

and we assume we have already extracted programs \( g \) and \( h \) from the proof terms \( d \) and \( e \), respectively, then the program extracted from the proof term \( \text{ind}[d, e] \) is a procedure \( f \) that is defined from \( g \) and \( h \) by 

**primitive recursion:**

\[
\begin{align*}
    f(0) &= g \\
    f(a + 1) &= h(a, f(a))
\end{align*}
\]

In the lecture we carried out the formal proof of \((+)\) in the Minlog system.

We do no show the proof term, since it would fill several pages. But here is the program which is extracted fully automatically from the proof:

\[
\text{(define (quotrem-prog n^1)}
\]
\[
\quad ((\text{nat-rec-run (cons 0 0)})
\quad \lambda (n^3)
\quad \lambda (\text{nat*nat^4})
\quad \text{(cons (if ((<-run ((\text{plus-run (cdr nat*nat^4)) 1)) n^1)}
\quad \text{(car nat*nat^4)}
\quad ((\text{plus-run (car nat*nat^4)) 1}))
\quad \text{(if ((<-run ((\text{plus-run (cdr nat*nat^4)) 1)) n^1)}
\quad \text{(plus-run (cdr nat*nat^4)) 1)}
\quad 0))))))
\]

This is a functional program in the programming language Scheme (a Lisp dialect). Let us try it out:

\[
\text{((quotrem-prog 7) 93)}
\]
\[
> (13 . 2)
\]

This means

\[
93 = 7 \times 13 + 2
\]

The advantage of program synthesis from proofs over conventional programming can be summarized as follows:
Programs extracted from proofs are guaranteed to be correct and their correctness can mechanically be checked by simply checking whether the proof is syntactically correct. Such a correctness check is impossible for conventional programs. Conventional programs can only be checked whether they are syntactically correct or type correct, but these checks do not guarantee that the program behaves as it should.

We close this section with an example showing that we cannot expect theorem 4.3.1 to hold for classical proofs, that is proofs using the rule reductio ad absurdum. We prove classically the following

**Theorem.** There are irrational numbers $x$ and $y$ such that $x^y$ is rational.

**Proof.** We do case analysis according to whether or not $\sqrt{2^{\sqrt{2}}}$ is rational. This case analysis makes use of the classically valid formula $P \lor \neg P$ where $P$ is the statement “$\sqrt{2^{\sqrt{2}}}$ is rational” (see first coursework, question 4 (c)).

*Case $\sqrt{2^{\sqrt{2}}}$ is rational.* Then we can take $x := \sqrt{2}$ and $y := \sqrt{2}$, because, as we all know, $\sqrt{2}$ is irrational.

*Case $\sqrt{2^{\sqrt{2}}}$ is irrational.* Then take $x := \sqrt[4]{2^{\sqrt{2}}}$ and $y := \sqrt{2}$. Now again $x$ and $y$ are irrational and we have

$$x^y = (\sqrt{2^{\sqrt{2}}})^{\sqrt{2}} = \sqrt{2^{\sqrt{2}\cdot\sqrt{2}}} = \sqrt{2^2} = 2$$

so $x^y$ is rational.

Although this is a nice and short proof, it is somewhat unsatisfactory since it does not provide examples of irrational numbers $x, y$ such that $x^y$ is rational. A constructive proof would yield such examples.

Nevertheless we have the following restricted theorem for classical logic.

**4.3.3 Theorem (Program synthesis from classical proofs)**

From every classical proof of

$$\forall x \exists y \ R(x, y)$$

where the formula $R(x, y)$ is quantifier free one can extract a program $p$ such that

$$\forall x \ R(x, p(x))$$

is provable, that is, $p$ is provably correct.

Note that theorem 4.3.3 does not apply to the last example, because the statement “$x$ is irrational”, when formalized does contain quantifiers.
5 Abstract data types

In the chapters 3 and 4 we studied the formal notion of proof and how it can be used to generate correct programs. Now we will study abstract data types and their formal specification, that is formal description. Since these specifications describe certain classes of algebras, it is common to call them algebraic specifications. Formal specifications are of fundamental importance in software engineering, but they can also be used - as we will see in the last chapter - to generate provably correct “prototypes” of programs.

5.1 Homomorphisms and abstract data types

Homomorphisms are mappings between algebras that ‘preserve the structure’. They are fundamental for the study of algebras.

5.1.1 Definition

Let \( \Sigma = (S, \Omega) \) be a signature and \( A, B \) two \( \Sigma \)-algebras. A homomorphism \( \varphi : A \to B \) from \( A \) to \( B \) is a family \( \varphi = (\varphi_s)_{s \in S} \) of functions

\[
\varphi_s : A_s \to B_s
\]

such that

- \( \varphi_s(c^A) = c^B \) for each constant \( c : s \) in \( \Omega \),

- \( \varphi_s(f^A(a_1, \ldots, a_n)) = f^B(\varphi_{s_1}(a_1), \ldots, \varphi_{s_n}(a_n)) \) for each operation \( f : s_1 \times \ldots \times s_n \to s \) and all \( (a_1, \ldots, a_n) \in A_{s_1} \times \ldots \times A_{s_n} \).

The second condition can be abbreviated, using the symbol ‘\( \circ \)’ for composition, by

\[
\varphi_s \circ f^A = f^B \circ (\varphi_{s_1}, \ldots, \varphi_{s_n})
\]

and depicted by the following commutative diagram:

\[
\begin{array}{ccc}
A_{s_1} \times \cdots \times A_{s_n} & \overset{f^A}{\longrightarrow} & A_s \\
\downarrow \varphi_{s_1} \quad \cdots \quad \downarrow \varphi_{s_n} & & \downarrow \varphi_s \\
B_{s_1} \times \cdots \times B_{s_n} & \overset{f^B}{\longrightarrow} & B_s
\end{array}
\]

The homomorphism \( \varphi : A \to B \) is called a isomorphism (monomorphism) (epimorphism) from \( A \) to \( B \) if \( \varphi_s : A_s \to B_s \) is bijective (injective) (surjective) for every sort \( s \). A homomorphism (isomorphism) from an algebra to itself is called endomorphism (automorphism).
5.1.2 Example

Consider the following signature

<table>
<thead>
<tr>
<th>Signature</th>
<th>$\Sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>nat</td>
</tr>
<tr>
<td>Constants</td>
<td>$0 : \text{nat}$</td>
</tr>
<tr>
<td>Operations</td>
<td>$\text{add} : \text{nat} \times \text{nat} \rightarrow \text{nat}$</td>
</tr>
</tbody>
</table>

The $\Sigma$-algebra $A$ of natural numbers with 0 and addition is given by

<table>
<thead>
<tr>
<th>Algebra</th>
<th>$A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carriers</td>
<td>$\mathbb{N}$</td>
</tr>
<tr>
<td>Constants</td>
<td>0</td>
</tr>
<tr>
<td>Operations</td>
<td>$+ : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$</td>
</tr>
</tbody>
</table>

For the same signature $\Sigma$ we also consider another algebra with carrier $M := \{1, 2, 4, 8, \ldots\}$, the constant 1 and multiplication restricted to $M$. We call this algebra $B$. Hence we have

<table>
<thead>
<tr>
<th>Algebra</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carriers</td>
<td>$M$</td>
</tr>
<tr>
<td>Constants</td>
<td>1</td>
</tr>
<tr>
<td>Operations</td>
<td>$* : M \times M \rightarrow M$</td>
</tr>
</tbody>
</table>

Define

$$ \varphi : \mathbb{N} \rightarrow M , \quad \varphi(n) := 2^n $$

We show that $\varphi$ is an isomorphism from the algebra $A$ to the algebra $B$ (note that $\varphi$ consists of just one function, since $\Sigma$ contains only one sort). In order to check that $\varphi$ is a homomorphism we calculate
\[ \varphi(0^A) = \varphi(0) = 1 = 0^B. \]
\[ \varphi(\text{add}^A(m, n)) = \varphi(m + n) = 2^{m+n} = 2^m \cdot 2^n = \varphi(m) \cdot \varphi(n) = \text{add}^B(\varphi(m), \varphi(n)). \]

Since obviously \( \varphi \) is bijective, it is an isomorphism.

### 5.1.3 Example

The data type \textit{Stack} is given by an unspecified set \( E \) of \textit{elements}, a set stack of \textit{stacks} and the operations The corresponding signature is

<table>
<thead>
<tr>
<th>Signature</th>
<th>STACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>elts, stack</td>
</tr>
<tr>
<td>Constants</td>
<td>emptystack: stack</td>
</tr>
<tr>
<td>Operations</td>
<td>push: elts \times stack \to stack</td>
</tr>
<tr>
<td></td>
<td>pop: stack \to stack</td>
</tr>
<tr>
<td></td>
<td>top: stack \to elts</td>
</tr>
</tbody>
</table>

The following is an algebra for the signature \textit{STACK}.

<table>
<thead>
<tr>
<th>Algebra</th>
<th>Seq(\mathbb{N})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carriers</td>
<td>(\mathbb{N}, \mathbb{N}^*) (the set of finite sequences of natural numbers)</td>
</tr>
<tr>
<td>Constants</td>
<td>(\epsilon) (the empty sequence)</td>
</tr>
<tr>
<td>Operations</td>
<td>cons: (\mathbb{N} \times \mathbb{N}^* \to \mathbb{N}^*) (insert a number in front of a sequence)</td>
</tr>
<tr>
<td></td>
<td>tail: (\mathbb{N}^* \to \mathbb{N}^*) (remove first element if nonempty, o.w. return (\epsilon))</td>
</tr>
<tr>
<td></td>
<td>head: (\mathbb{N}^* \to \mathbb{N}) (take first element if nonempty, otherwise return 0)</td>
</tr>
</tbody>
</table>

where push adds an element to a stack, pop removes the topmost symbol from a non-empty stack and top returns the topmost element.

Consider the following algebra for the signature \textit{STACK}:
Let us define a homomorphism $\varphi : \text{SeqN} \rightarrow \text{Stack0}$
Note that $\varphi$ must be a pair of functions $\varphi = (\varphi_{\text{elts}}, \varphi_{\text{stack}})$ where

$\varphi_{\text{elts}} : \text{SeqN}_{\text{elts}} \rightarrow \text{Stack0}_{\text{elts}} \quad \varphi_{\text{stack}} : \text{SeqN}_{\text{stack}} \rightarrow \text{Stack0}_{\text{stack}}$

Since $\text{SeqN}_{\text{elts}} = \mathbb{N}$, $\text{Stack0}_{\text{elts}} = \{0\}$, $\text{SeqN}_{\text{stack}} = \mathbb{N}^\ast$, and $\text{Stack0}_{\text{stack}} = \mathbb{N}$, this means

$\varphi_{\text{elts}} : \mathbb{N} \rightarrow \{0\} \quad \varphi_{\text{stack}} : \mathbb{N}^\ast \rightarrow \mathbb{N}$

Hence for $\varphi_{\text{elts}}$ we have no choice; we have to set $\varphi_{\text{elts}}(n) := 0$ for all $n \in \mathbb{N}$. For $\varphi_{\text{stack}}$ we stipulate

$\varphi_{\text{stack}}(\alpha) := \text{length}(\alpha) \quad \text{(the length of the sequence } \alpha)\text{).}$

In order to show that $\varphi$ is a homomorphism, we have to check 4 equations, one for each constant and operation in \textbf{STACK}. We only check the equation for push and leave the rest as an exercise.

$\varphi_{\text{stack}}(\text{push}^{\text{SeqN}}(n, \alpha)) = \varphi_{\text{stack}}(\text{cons}(n, \alpha))$

$= \text{length}(\text{cons}(n, \alpha))$

$= \text{length}(\alpha) + 1$

$= \text{push}^{\text{Stack0}}(0, \text{length}(\alpha))$

$= \text{push}^{\text{Stack0}}(\varphi_{\text{elts}}(n), \varphi_{\text{stack}}(\alpha))$

Obviously $\varphi_{\text{elts}}$ and $\varphi_{\text{stack}}$ are both surjective, hence $\varphi$ is an epimorphism. But clearly $\varphi$ is not a monomorphism.

\textbf{Remark.} This example exhibits a typical feature of epimorphisms: they simplify. In our example $\varphi$ ‘forgets’ the natural numbers and replaces them by 0.

5.1.4 Definition

Let $\Sigma = (S, \Omega)$ be a signature and $A, B, C \in \Sigma$-algebras. For homomorphism $\varphi : A \rightarrow B$, $\psi : B \rightarrow C$ its \textbf{composition} $\psi \circ \varphi : A \rightarrow C$ is defined as the family $\psi \circ \varphi := (\psi_s \circ \varphi_s)_{s \in S}$. 
5.1.5 Theorem

Homomorphisms are closed under composition, that is, if $\varphi$ is a homomorphism from $A$ to $B$ and $\psi$ is a homomorphism from $B$ to $C$, then $\psi \circ \varphi$ is a homomorphism from $A$ to $C$.

Proof. Coursework.

5.1.6 Theorem

Isomorphisms are closed under inverses, that is, if $\varphi$ is an isomorphism from $A$ to $B$, then $\varphi^{-1} := (\varphi_s^{-1})_{s \in S}$ is an isomorphism from $B$ to $A$.

Proof. As $\varphi_s^{-1} : B_s \to A_s$ is a bijective function for each $s \in S$, it suffices to show that $\varphi^{-1}$ is a homomorphism. For each constant $c : s$ we have

$$\varphi_s^{-1}(c^B) = \varphi_s^{-1}(\varphi_s(c^A)) = c^A$$

Now let $f : s_1 \times \ldots \times s_n \to s$ be an operation in $\Omega$. The homomorphism condition is

$$\varphi_s^{-1}(f^B(b_1, \ldots, b_n)) = f^A(\varphi_s^{-1}(a_1), \ldots, \varphi_s^{-1}(a_n)).$$

We have

$$\varphi_s^{-1}(f^B(b_1, \ldots, b_n)) = \varphi_s^{-1}(f^B(\varphi_s^{-1}(a_1), \ldots, \varphi_s^{-1}(a_n))) = f^A(\varphi_s^{-1}(a_1), \ldots, \varphi_s^{-1}(a_n)).$$

\[\square\]

5.1.7 Definition

For two $\Sigma$-algebras $A$ and $B$ we set

$$A \simeq B \iff \text{there exists an isomorphism from } A \text{ to } B$$

5.1.8 Theorem

For every signature $\Sigma$ the relation of isomorphism between $\Sigma$-algebras, $A \simeq B$ is an equivalence relation.

Proof. Let $A, B, C$ be $\Sigma$-algebras, where $\Sigma = (S, \Omega)$.

(i) Reflexivity. $A \simeq A$ holds, since clearly the family of identity functions on the carriers of $A$ is an isomorphism from $A$ to $A$.

(ii) Symmetry. Assume $A \simeq B$, i.e. there is an isomorphism $\varphi : A \to B$. By theorem 5.1.6 $\varphi^{-1} : B \to A$ is an isomorphism, hence $B \simeq A$.

(iii) Transitivity. Assume $A \simeq B$ and $B \simeq C$, i.e. there are isomorphisms $\varphi : A \to B$ and $\psi : B \to C$. By theorem 5.1.5 $\psi \circ \varphi : A \to C$ is an homomorphism. Since obviously $\psi \circ \varphi$ is bijective, it is an isomorphisms. Hence $A \simeq C$.
5.1.9 Definition

For a signature $\Sigma$ we let $\text{Alg}(\Sigma)$ denote the class of all $\Sigma$-algebras.

In this definition we had to use the word `class' instead of `set', because in general $\text{Alg}(\Sigma)$ is too large to be a set. For example, the class of all algebras for the trivial signature $(\{s\}, \emptyset)$ (one sort no constants, no operations) corresponds to the class of all sets, since an algebra for this signature consist of a carrier set for the sort $s$ only, i.e. is a set. But from Russell's Paradox it follows that the class of all sets is not a set (intuitively its too large). Hence we see that $\text{Alg}(\{s\}, \emptyset)$ is a proper class, i.e. not a set.

5.1.10 Definition

An abstract data type (ADT) for a signature $\Sigma$ is a class $C \subseteq \text{Alg}(\Sigma)$ of $\Sigma$-algebras which is closed under isomorphisms, i.e.

$$\text{if } A \in C \text{ and } A \simeq B \text{ then } B \in C.$$  

An ADT $C$ is called monomorphic if all its elements are isomorphic, i.e. if

$$A \in C \text{ and } B \in C \text{ then } A \simeq B.$$  

Otherwise $C$ is called polymorphic.

Example 8. Let $\Sigma$ be a signature.

(a) $\text{Alg}(\Sigma)$ is an ADT (which is usually polymorphic).

(b) For each $\Sigma$-algebra $A$ the class $\{B \in \text{Alg}(\Sigma) \mid B \simeq A\}$ is a monomorphic ADT. In fact every nonempty monomorphic ADT is of this form.

(c) The class of finite $\Sigma$-algebras, more precisely

$$\{A \in \text{Alg}(\Sigma) \mid \text{all carriers of } A \text{ are finite} \}$$

is an ADT (which is usually polymorphic).

Remark. Another way of looking at ADTs is to view them as abstract properties of algebras$^2$. The property defined by an ADT is abstract because it is invariant under isomorphic copies (cf. e.g. the property of having finite carriers in example 8 (c) above). An example of a non-abstract property is the property of having the set of $\mathbb{N}$ of natural numbers as carrier set. By referring to the concrete set $\mathbb{N}$ the property of being invariant under isomorphism is lost. Hence the class

$$\{A \in \text{Alg}(\Sigma) \mid \text{all carriers of } A \text{ are } = \mathbb{N}\}$$

is not an ADT. Referring—like above— to a fixed set in the specification of a data type means on the programming side to fix a concrete implementation of a data type already in the specification of a system. Such premature design decisions usually are disastrous since they make a software development inflexible and difficult to maintain.

---

$^2$In general the concept of a class and of a property are equivalent: each class defines the property of being in the class, and conversely each property defines the class of object having that property.
5.2 Reducts, Subalgebras and Quotients

In this section we discuss three fundamental methods of passing from one algebra to another which is in some sense smaller or simpler. The first method is to throw carrier sets and operations away (reducts), the second is to throw elements away, i.e. make the carrier sets smaller (subalgebras), the third is to ‘forget’ differences between elements, i.e. to identify certain elements (quotients).

5.2.1 Definition

A signature $\Sigma = (S, \Omega)$ is a subsignature of a signature $\Sigma' = (S', \Omega')$ if $S \subseteq S'$ and $\Omega \subseteq \Omega'$, i.e. every sort in $\Sigma$ is also a sort in $\Sigma'$, and every operation or constant in $\Sigma$ is also an operation or constant in $\Sigma'$.

We write $\Sigma \subseteq \Sigma'$ to indicate that $\Sigma$ is a subsignature of $\Sigma'$.

If $\Sigma$ is a subsignature of $\Sigma'$ we also say that $\Sigma'$ is an expansion of $\Sigma$.

5.2.2 Definition

Let $\Sigma$ be a subsignature of $\Sigma'$. To every $\Sigma'$-algebra $A$ we can construct a $\Sigma$-algebra $B$ by ‘throwing away’ all parts of $A$ not named in $\Sigma$, i.e.

- $B_s := A_s$ for all sorts $s$ in $\Sigma$,
- $c^B := c^A$ for all constants $c$ in $\Sigma$,
- $f^B := f^A$ for all operations $f$ in $\Sigma$,

We call $B$ the $\Sigma$-reduct of $A$ and denote it by $A|_\Sigma$.

If $B$ is the $\Sigma$-reduct of $A$ we also say that $A$ is an expansion of $B$.

The notion of a reduct can be easily extended to ADTs. Given an ADT $C$ for a signature $\Sigma'$ and a subsignature $\Sigma$ of $\Sigma'$ we can define the $\Sigma$-reduct of $C$ by

$$C|_\Sigma := \{ A|_\Sigma \mid A \in C \}$$

It is easy to see that this class is an ADT again.

5.2.3 Definition

Let $\Sigma = (S, \Omega)$ be a signature and let $A$ and $B$ be $\Sigma$-algebras. $A$ is called a subalgebra of $B$ if

- $A_s \subseteq B_s$ for all $s \in S$,
- $c^A = c^B$ for all constants $c \in \Omega$,
- $f^A(a_1, \ldots, a_n) = f^B(a_1, \ldots, a_n)$ for all operations $f \in \Omega$ and all $(a_1, \ldots, a_n) \in A_{s_1} \times \ldots \times A_{s_n}$. 
5.2.4 Remarks

1. Obviously, the relation ‘$A$ is a subalgebra of $B$’ defines a partial order on the class $\text{Alg}(\Sigma)$ of all $\Sigma$-algebras.

2. Clearly, a subalgebra $A$ of an algebra $B$ is completely determined by $B$ and the sets $A_s$. However, if we chose arbitrary subsets $A_s$ of $B_s$ for all sorts $s$ these will define a subalgebra of $B$ only if the sets $A_s$ contain all the constants $\sigma_B$ and are ‘closed’ under the operations $f_B$. For example, the set of even numbers defines a subalgebra of the $A$ of example 5.1.2, since 0 is even and the even numbers are closed under addition. However the odd numbers do not define a subalgebra of $A$.

5.2.5 Example

Let $\Sigma = (S, \Omega)$ a signature, let $A$ and $B$ be $\Sigma$-algebras and let $\varphi : A \to B$ be a homomorphism. For each sort $s \in S$ we define the set

$$\varphi(A_s) := \{ \varphi(a) \mid a \in A_s \} \subseteq B_s$$

From the properties of a homomorphism it follows that the sets $\varphi(A_s)$ contain all constants $\sigma_B$ and are ‘closed’ under the operations $f_B$. Hence the family of sets $\varphi(A) := (\varphi(A_s))_{s \in S}$ defines a subalgebra of $B$ called homomorphic image of $A$ under $\varphi$.

5.2.6 Definition

Let $\Sigma = (S, \Omega)$ be a signature and $A$ a $\Sigma$-algebra. A congruence on $A$ is a family $\sim = (\sim_s)_{s \in S}$ of equivalence relations $\sim_s$ on $A_s$, $s \in S$, that is respected by all operations of $A$, i.e. for any operation $f : s_1 \times \ldots \times s_n \to s$ and any $a_i, b_i \in A_{s_i}$

$$a_i \sim_s b_i \ (1 \leq i \leq n) \ \Rightarrow \ f^A(a_1, \ldots, a_n) \sim_s f^A(b_1, \ldots, b_n)$$

For $a \in A_s$ we set

$$[a]_{\sim_s} := \{ b \in A_s \mid a \sim_s b \}$$

The quotient algebra (or quotient) of $A$ by $\sim$ is the $\Sigma$-algebra $A/\sim$ defined as follows.

- $(A/\sim)_s := \{ [a]_{\sim_s} \mid a \in A_s \}$, for every sort $s \in \Sigma$.
- $\sigma^A/\sim := [\sigma^A]_{\sim_s}$, for every constant $\sigma : s$.
- $f^A/\sim([a_1]_{\sim_{s_1}}, \ldots, [a_n]_{\sim_{s_n}}) := [f^A(a_1, \ldots, a_n)]_{\sim_s}$, for each operation $f : s_1 \times \ldots \times s_n \to s$ and all $a_i \in A_{s_i}$.
Note that the condition on $\sim$ of being a congruence is needed to verify that $f^{A/\sim}$ is well-defined, i.e. the right hand side of the defining equation does not depend on the choice of the representatives $a_i$ of the equivalence classes $[a_i]_{\sim s_i}$.

**Example 10.** On the set $\mathbb{N}$ of natural numbers, considered as the carrier of the $\textbf{NZA}$-algebra $A$, we define a relation $\sim$ by

$$a \sim b \iff a + b \text{ is even}.$$ 

Clearly $\sim$ is an equivalence relation. To prove that it is a congruence for $A$, we have to show that $\sim$ is preserved by the operation $\text{add}^A$, which is addition. Hence we have to show

$$a_1 \sim b_1, \ a_2 \sim b_2 \implies a_1 + a_2 \sim b_1 + b_2$$

for all $a_1, a_2, b_1, b_2 \in \mathbb{N}$. We leave the verification of this implication as an exercise.

It is clear that $\sim$ has two equivalence classes, the set $\text{EVEN}$ of even numbers and the set $\text{ODD}$ of odd numbers. Hence the carrier of the quotient algebra $A/\sim$ is the two element set $\{\text{EVEN}, \text{ODD}\}$. For $v, w \in \{\text{EVEN}, \text{ODD}\}$ we have $\text{add}^{A/\sim}(v, w) = \text{EVEN}$ or $\text{ODD}$, depending on whether $v = w$ or $v \neq w$ (the sum of two numbers is even if and only if both are even or both are odd). Hence the table for $\text{add}^{A/\sim}$ is

<table>
<thead>
<tr>
<th>add$^{A/\sim}$</th>
<th>EVEN</th>
<th>ODD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEN</td>
<td>EVEN</td>
<td>ODD</td>
</tr>
<tr>
<td>ODD</td>
<td>ODD</td>
<td>EVEN</td>
</tr>
</tbody>
</table>

### 5.2.7 Homomorphism Theorem

Let $A, B$ be algebras for a signature $\Sigma = (S, \Omega)$, and let $\varphi : A \to B$ be a homomorphism. For each sort $s \in S$ define a binary relation $\sim_{\varphi, s}$ on $A_s$ by

$$a \sim_{\varphi, s} b \iff \varphi_s(a) = \varphi_s(b).$$

Then the family $\sim_{\varphi} := (\sim_{\varphi, s})_{s \in S}$ is a congruence on $A$ and the quotient algebra $A/\sim_{\varphi}$ is isomorphic to the homomorphic image of $A$ under $\varphi$, i.e.

$$A/\sim_{\varphi} \simeq \varphi(A),$$

the canonical isomorphism $[\varphi] : A/\sim_{\varphi} \to \varphi(A)$ being defined by

$$[\varphi]([a]_{\sim_{\varphi, s}}) := \varphi_s(a)$$

for $s \in S$ and $a \in A_s$. 
Proof. The easy proof that $\sim_\varphi$ is a congruence on $A$ is left to the reader. In order to prove that $[\varphi]$ is a homomorphism, we take an operation $f \in \Omega$, which, for simplicity, we assume to be unary, e.g. $f : s_1 \rightarrow s$. Let $a \in A_{s_1}$. We have to show that $[\varphi]_s(f^{A/\sim_\varphi}([a]_{\sim_\varphi,s_1})) = f^B([\varphi]_{s_1}([a]_{\sim_\varphi,s_1}))$, which is verified by the following calculation:

$$
[\varphi]_s(f^{A/\sim_\varphi}([a]_{\sim_\varphi,s_1})) = [\varphi]_s([f^A(a)]_{\sim_\varphi,s}) = \varphi_s(f^A(a)) = f^B(\varphi_{s_1}(a)) = f^B([\varphi]_{s_1}([a]_{\sim_\varphi,s_1}))
$$

Since obviously $[\varphi]_s$ is bijective for each sort $s$, we have shown that $[\varphi]$ is an isomorphism.

Remark. The above theorem tells us that each homomorphism $\varphi : A \rightarrow B$ naturally induces a congruence $\sim_\varphi$ on $A$. In fact every congruence $\sim$ on $A$ can be obtained in that way, since the mappings $[\cdot]_{\sim} : A_s \rightarrow A_s/\sim_s$ obviously constitute a homomorphism $[\cdot]_{\sim} : A \rightarrow A/\sim$ and clearly the congruence induced by $[\cdot]_{\sim}$ coincides with $\sim$, i.e. $\sim_{[\cdot]_{\sim}} = \sim$.

5.3 Initial algebras

5.3.1 Definition

Let $A$ be a $\Sigma$-algebra and $C$ a class of $\Sigma$-algebras.

$A$ is initial for $C$ if for every $B \in C$ there exists exactly one homomorphism $\varphi : A \rightarrow B$.

We say $A$ is initial in $C$ if $A$ is initial for $C$ and in addition $A \in C$. We say that $A$ is initial if $A$ is initial in Alg($\Sigma$).

Remark. By replacing in the definition above $\varphi : A \rightarrow B$ by $\varphi : B \rightarrow A$ we obtain the notion of a final algebra.

5.3.2 Definition

For any $\Sigma$-algebra $A$ and any variable assignment $\alpha : X \rightarrow A$ the family of functions $\text{eval}^{A,\alpha} = (\text{eval}^{A,\alpha}_s)_{s \in S}$ defined by

$$
\text{eval}^{A,\alpha}_s : T(\Sigma, X) \rightarrow A, \quad \text{eval}^{A,\alpha}_s(t) := t^{A,\alpha}
$$

is a homomorphism $\text{eval}^{A,\alpha} : T(\Sigma, X) \rightarrow A$ which is called evaluation homomorphism.

In the special case $X = \emptyset$ the evaluation homomorphism is independent of a variable assignment and is written $\text{eval}^A : T(\Sigma) \rightarrow A$. We sometimes also write eval instead of eval$^A$ if the algebra $A$ is clear from the context.
5.3.3 Definition

A $\Sigma$-algebra $A$ is called **generated (freely generated)** if every $a \in A_s$ is the value of a (unique) closed term, i.e., for every $a \in A_s$ there exists a (unique) term $t \in T(\Sigma)$ such that $t^A = a$. Note that this is equivalent to saying that $\text{eval}: T(\Sigma) \to A$ is an epimorphism (isomorphism).

It is convenient to generalize this definition as follows. Let $\Sigma = (S, \Omega)$ and let $\Omega' \subseteq \Omega$ be a set of constants and operations called **constructors**. We set $\Sigma' := (S, \Omega')$. A $\Sigma$-algebra $A$ is called **generated (freely generated) by $\Omega'$** if every $a \in A_s$ is the (unique) value of a closed $\Sigma'$-term, i.e., for every $a \in A_s$ there exists a (unique) term $t \in T(\Sigma')$ such that $t^A = a$.

5.3.4 Examples

1. The closed term algebra $T(\Sigma)$ is freely generated, since for every closed term $\Sigma$-term $t$ we have $t^{T(\Sigma)} = t$.

2. For a given $\Sigma$-algebra $A$ the subalgebra $\text{eval}(T(\Sigma))$, i.e., the homomorphic image (cf. example 9) of the closed term algebra $T(\Sigma)$ under the evaluation homomorphism $\text{eval}^A: T(\Sigma) \to A$, is generated.

3. The $\text{NZA}$-algebra $\text{N0+}$ is not term generated, since $0$ is the only natural number which is the value of a closed $\text{NZA}$-term.

4. Consider the following signature.

<table>
<thead>
<tr>
<th>Signature</th>
<th>BOOLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>boole</td>
</tr>
<tr>
<td>Constants</td>
<td>$T$ : boole</td>
</tr>
<tr>
<td></td>
<td>$F$ : boole</td>
</tr>
<tr>
<td>Operations</td>
<td>$\text{not}$ : boole $\to$ boole</td>
</tr>
<tr>
<td></td>
<td>$\text{and}$ : boole $\times$ boole $\to$ boole</td>
</tr>
<tr>
<td></td>
<td>$\text{or}$ : boole $\times$ boole $\to$ boole</td>
</tr>
</tbody>
</table>

and the following $\text{BOOLE}$-algebra
The algebra Boole of boolean values is generated, since, for example, $T_{\text{Boole}} = \#t$ and $F_{\text{Boole}} = \#f$. However, Boole is not freely generated since, for example, $\#t = T_{\text{Boole}} = \text{not}(T_{\text{Boole}})$.

Obviously Boole is freely generated by \{T,F\}.

Boole is also generated by \{T,\text{not}\}, since $\text{not}(T_{\text{Boole}}) = \#f$, but not freely, since, for example, $\#t = T_{\text{Boole}} = \text{not}(\text{not}(T))_{\text{Boole}}$.

5.3.5 Theorem

For every signature $\Sigma$ the closed term algebra $T(\Sigma)$ is initial.

**Proof.** For every $\Sigma$-algebra $A$ we have the evaluation homomorphism $\text{eval}^A : T(\Sigma) \rightarrow A$. In order to show that $\text{eval}^A$ is unique, let $\varphi : T(\Sigma) \rightarrow A$ be a further homomorphism. We prove by term induction that $\varphi(t) = \text{eval}^A(t)$ for all $t \in T(\Sigma)$ (here and further on we omit sorts as subscripts as long as this does not lead to ambiguities).

**Base.**

$$\varphi(c) = \varphi(c^{T(\Sigma)})$$
$$= c^A \quad \text{since } \varphi \text{ is a homomorphism}$$
$$= \text{eval}^A(c).$$

**Step.**

$$\varphi(f(t_1, \ldots, t_n)) = \varphi(f^{T(\Sigma)}(t_1, \ldots, t_n))$$
$$= f^A(\varphi(t_1), \ldots, \varphi(t_n)) \quad \text{since } \varphi \text{ is a homomorphism}$$
$$= f^A(\text{eval}^A(t_1), \ldots, \text{eval}^A(t_n)) \quad \text{by induction hypothesis}$$
$$= \text{eval}^A(f(t_1, \ldots, t_n)) \quad \text{by definition of } \text{eval}^A.$$
5.3.6 Theorem

For a $\Sigma$-algebra $A$ the following statements are equivalent.

(i) $A$ is initial.

(ii) $A$ is freely generated.

(iii) $A \simeq T(\Sigma)$.

Proof. Recall that asserting (ii) is equivalent to saying that eval$^A \colon T(\Sigma) \to A$ is an isomorphism.

(i) $\Rightarrow$ (ii) Let $A$ be initial. We have to show that $A$ is freely generated, i.e. eval$^A \colon T(\Sigma) \to A$ is an isomorphism. Since $A$ is initial there is a unique homomorphism $\varphi : A \to T(\Sigma)$. Then $\varphi \circ \text{eval}^A : T(\Sigma) \to T(\Sigma)$ is a homomorphism. Furthermore the identity id$_{T(\Sigma)} : T(\Sigma) \to T(\Sigma)$ is a homomorphism, trivially. Since, by theorem 5.3.5, $T(\Sigma)$ is initial we may conclude that $\varphi \circ \text{eval}^A = \text{id}_{T(\Sigma)}$. We also have the homomorphism eval$^A \circ \varphi : A \to A$, and, using the initiality of $A$, it follows with a similar argument that eval$^A \circ \varphi = \text{id}^A$. Therefore eval$^A$ must be an isomorphism (with inverse $\varphi$).

(ii) $\Rightarrow$ (iii) If $A$ is freely generated then eval$^A : T(\Sigma) \to A$ is an isomorphism. Hence $A \simeq T(\Sigma)$.

(iii) $\Rightarrow$ (i) Assume $A \simeq T(\Sigma)$, i.e. there is an isomorphism $\varphi : A \to T(\Sigma)$. In order to show that $A$ is initial we take an arbitrary $\Sigma$-algebra $B$ and show that there is exactly one homomorphism from $A$ to $B$. Since eval$^B \circ \varphi : A \to B$ is a homomorphism we have to prove that any other homomorphism from $A$ to $B$ coincides with eval$^B \circ \varphi$. So, let $\psi : A \to B$ a homomorphism. Since $\psi \circ \varphi^{-1} : T(\Sigma) \to B$ is a homomorphism we may use the initiality of $T(\Sigma)$ to conclude that $\psi \circ \varphi^{-1} = \text{eval}^B$. Hence $\psi = \text{eval}^B \circ \varphi$.

Given a class $C$ of $\Sigma$-algebras one is often interested in finding a $\Sigma$ algebra which is initial in $C$. Now, since the $\Sigma$-algebra $T(\Sigma)$ is initial it is also initial for $C$, but in general not initial in $C$, since $T(\Sigma)$ might fail to be an element of $C$.

For example let $C$ be the class of all algebras for the signature BOOLE in which the usual laws for a boolean algebra are true (a precise definition of these laws will be presented in the next chapter). Then the closed term algebra $T(\text{BOOLE})$ does certainly not belong to $C$ because e.g. the law not$(T) = F$ does not hold in $T(\text{BOOLE})$ (the terms not$(T)$ and $F$ are not equal).

In the following we describe how to construct from a class $C$ of $\Sigma$-algebras a $\Sigma$-algebra which is always initial for $C$ and, as we will see later, is in many cases an element of $C$ and therefore initial in $C$.

5.3.7 Theorem

Let $\Sigma = (S, \Omega)$ be a signature and $C$ a class of $\Sigma$-algebras. For every sort $s \in S$ we define a binary relation $\sim_{C,s}$ on $T(\Sigma)_s$ by

$t_1 \sim_{C,s} t_2 \iff \text{ for all } A \in C \ t_1^A = t_2^A$
Then \( \sim_C := (\sim_{C,s})_{s \in S} \) is a congruence on \( T(\Sigma) \) and the quotient algebra

\[
T_C(\Sigma) := T(\Sigma)/\sim_C
\]

is initial for \( C \). For every \( A \in C \) the unique homomorphism from \( \varphi: T_C(\Sigma) \to A \) is given by

\[
\varphi_s([t]_{\sim_{C,s}}) = t^A
\]

for each sort \( s \) and each \( t \in T(\Sigma) \) of sort \( s \).

**Proof.** By definition \( \sim_C \) is the intersection of the congruences \( \sim_{\text{eval}^A} (A \in C) \) (cf. the Homomorphism Theorem 5.2.7). Since congruences are closed under intersections (the easy proof is left as an exercise) it follows that \( \sim_C \) is a congruence on \( T(\Sigma) \). It is easy to see that \( \varphi \) above is a well-defined homomorphism. The proof that \( \varphi \) is unique is similar to the proof of theorem 5.3.5 and is left as an exercise. Hence \( T_C(\Sigma) \) is initial for \( C \).
6 Specifications

Having introduced algebras and some basic mathematical constructions for them we now study formal descriptions of algebras, which we call specifications. Initially we will consider arbitrary first-order specification, but will later concentrate on equational specifications which have particularly nice properties.

6.1 Loose specifications

6.1.1 Definition

A loose specification is a pair $(\Sigma, \Phi)$ where $\Sigma$ is a signature and $\Phi$ is a set of closed $\Sigma$-formulas. The formulas in $\Phi$ are called the axioms of the specification.

A $\Sigma$-algebra $A$ is a model of the loose specification $(\Sigma, \Phi)$ if all axioms in $\Phi$ are true in $A$, i.e. $A \models \Phi$.

We let $\text{Mod}_\Sigma(\Phi)$ denote the class of all models of the loose specification $(\Sigma, \Phi)$, i.e.

$$\text{Mod}_\Sigma(\Phi) \equiv \{ A \in \text{Alg}(\Sigma) \mid A \models \Phi \}$$

We will see that $\text{Mod}_\Sigma(\Phi)$ is an abstract data type. The proof of this fundamental facts needs some preparations.

6.1.2 Lemma

Let $\varphi : A \to B$ be a homomorphism between $\Sigma$-algebras and $\alpha : X \to A$ a variable assignment. Then for every term $t \in T(\Sigma, X)$ we have

$$\varphi(t^{A,\alpha}) = t^{B,\varphi\circ\alpha}$$

**Proof.** Structural induction on $t$.

**Base: variables.**

$$\varphi(x^{A,\alpha}) = \varphi(\alpha(x)) = (\varphi \circ \alpha)(x) = x^{B,\varphi\circ\alpha}$$

**Base: constants.**

$$\varphi(c^{A,\alpha}) = \varphi(c^{A}) = \varphi(c^{B}) = c^{B,\varphi\circ\alpha}$$
Step.

\[
\varphi(f(t_1, \ldots, t_n)^{A,\alpha}) = \varphi(f^A(t_1^{A,\alpha}, \ldots, t_n^{A,\alpha})) = f^B(\varphi(t_1^{A,\alpha}), \ldots, \varphi(t_n^{A,\alpha})) = f^B(t_1^{B,\varphi\circ\alpha}, \ldots, t_n^{B,\varphi\circ\alpha}) \quad \text{(by i.h.)} = f(t_1, \ldots, t_n)^{B,\varphi\circ\alpha}
\]

6.1.3 Theorem

Let \( \varphi : A \to B \) be an isomorphism between \( \Sigma \)-algebras. Then for every formula \( P \in \mathcal{L}(\Sigma, X) \) and every assignment \( \alpha : X \to A \) we have

\[
A, \alpha \models P \quad \text{iff} \quad B, \varphi \circ \alpha \models P
\]

In particular when \( P \) is closed we have

\[
A \models P \quad \text{iff} \quad B \models P
\]

Proof. Structural induction on the formula \( P \).

(i) Base.

\[
A, \alpha \models t_1 = t_2 \quad \text{iff} \quad t_1^{A,\alpha} = t_2^{A,\alpha} \quad \text{iff} \quad \varphi(t_1^{A,\alpha}) = \varphi(t_2^{A,\alpha}) \quad (\varphi \text{ is injective}) \quad \text{iff} \quad t_1^{B,\varphi\circ\alpha} = t_2^{B,\varphi\circ\alpha} \quad \text{(Lemma 6.1.2)} \quad \text{iff} \quad B, \varphi \circ \alpha \models t_1 = t_2
\]

(ii) Step: propositional connectives.

\[
A, \alpha \models P \land Q \quad \text{iff} \quad A, \alpha \models P \text{ and } A, \alpha \models Q \quad \text{iff} \quad B, \varphi \circ \alpha \models P \text{ and } B, \varphi \circ \alpha \models Q \quad \text{(i.h.)} \quad \text{iff} \quad B, \varphi \circ \alpha \models P \land Q
\]

\[
P \lor Q, \ P \to Q \quad \text{similar}
\]

\[
A, \alpha \models \neg P \quad \text{iff} \quad A, \alpha \not\models P \quad \text{iff} \quad B, \varphi \circ \alpha \not\models P \quad \text{(i.h.)} \quad \text{iff} \quad B, \varphi \circ \alpha \models \neg P
\]
(iii) **Step: quantifiers.**

\[ A, \alpha \models \forall x P \iff A, \alpha^a_x \models P \text{ for all } a \in A_s \]
\[ B, \varphi \circ (\alpha^a_x) \models P \text{ for all } a \in A_s \quad \text{(i.h.)} \]
\[ B, (\varphi \circ \alpha)^{\varphi(a)}_x \models P \text{ for all } a \in A_s \quad (\varphi \circ \alpha^a_x) = (\varphi \circ \alpha)^{\varphi(a)}_x \]
\[ B, (\varphi \circ \alpha)^b_x \models P \text{ for all } b \in B_s \quad (\varphi \text{ is surjective}) \]
\[ B, \varphi \circ \alpha \models \forall x P \]

\[ A, \alpha \models \exists x P \iff A, \alpha^a_x \models P \text{ for at least one } a \in A_s \]
\[ B, \varphi \circ (\alpha^a_x) \models P \text{ for at least one } a \in A_s \quad \text{(i.h.)} \]
\[ B, (\varphi \circ \alpha)^{\varphi(a)}_x \models P \text{ for at least one } a \in A_s \quad (\varphi \circ (\alpha_x^a)) = (\varphi \circ \alpha)^{\varphi(a)}_x \]
\[ B, (\varphi \circ \alpha)^b_x \models P \text{ for at least one } b \in B_s \quad (\varphi \text{ is surjective}) \]
\[ B, \varphi \circ \alpha \models \exists x P \]

### 6.1.4 Theorem

For every loose specification \((\Sigma, \Phi)\) the class of its models, \(\text{Mod}_\Sigma(\Phi)\), is an abstract data type.

**Proof.** Let \(A, B\) be \(\Sigma\)-algebras such that \(A \in \text{Mod}_\Sigma(\Phi)\) and \(A \cong B\). We have to show \(B \in \text{Mod}_\Sigma(\Phi)\). Since \(A \in \text{Mod}_\Sigma(\Phi)\) we have \(A \models \Phi\), i.e. \(A \models P\) for all \(P \in \Phi\). By theorem 6.1.3 it follows that \(B \models P\) for all \(P \in \Phi\) as well. Hence \(B \models \Phi\), i.e. \(B \in \text{Mod}_\Sigma(\Phi)\).

### 6.1.5 Example

Consider the loose specification \((\Sigma, \Phi)\), where

<table>
<thead>
<tr>
<th><strong>Signature</strong></th>
<th>(\Sigma)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sorts</strong></td>
<td>boole</td>
</tr>
<tr>
<td><strong>Constants</strong></td>
<td>(T, F : \text{boole})</td>
</tr>
<tr>
<td><strong>Operations</strong></td>
<td>not : boole (\rightarrow) boole</td>
</tr>
<tr>
<td></td>
<td>and, or : boole (\times) boole (\rightarrow) boole</td>
</tr>
</tbody>
</table>

and \(\Phi = \{P_1, \ldots, P_6\}\), with

\[ P_1 \equiv \text{not}(T) = F \]
\[ P_2 \equiv \text{not}(F) = T \]
\[ P_3 \equiv \text{and}(T, T) = T \]
\[ P_4 \equiv \forall x \, \text{and}(F, x) = F \]
\[ P_5 \equiv \forall x \, \text{and}(x, F) = F \]
\[ P_6 \equiv \forall x, y \, \text{or}(x, y) = \text{not}(\text{and}(\text{not}(x), \text{not}(y))) \]

Consider the following \( \Sigma \)-algebras

<table>
<thead>
<tr>
<th>Algebra</th>
<th>Boole</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carriers</td>
<td>( B := {#t, #f} )</td>
</tr>
<tr>
<td>Constants</td>
<td>#t, #f</td>
</tr>
</tbody>
</table>
| Operations | \( \neg: B \rightarrow B \) (negation)  
\( \land: B \times B \rightarrow B \) (conjunction)  
\( \lor: B \times B \rightarrow B \) (disjunction) |

<table>
<thead>
<tr>
<th>Algebra</th>
<th>Pow(N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carriers</td>
<td>( \mathcal{P}(N) := {A \mid A \subseteq N} ), the powerset of N</td>
</tr>
<tr>
<td>Constants</td>
<td>N, \emptyset</td>
</tr>
</tbody>
</table>
| Operations | \( N \setminus \cdot: \mathcal{P}(N) \rightarrow \mathcal{P}(N) \) (complement)  
\( \cap: \mathcal{P}(N) \times \mathcal{P}(N) \rightarrow \mathcal{P}(N) \) (conjunction)  
\( \cup: \mathcal{P}(N) \times \mathcal{P}(N) \rightarrow \mathcal{P}(N) \) (disjunction) |

which are clearly models of the loose specification \( (\Sigma, \Phi) \), that is

\[
\text{Boole} \models \Phi \quad \text{and} \quad \text{Pow}(N) \models \Phi
\]

or

\[
\text{Boole}, \text{Pow}(N) \in \text{Mod}_E(\Phi)
\]

The ADT \( \text{Mod}_E(\Phi) \) is polymorphic since it contains the non-isomorphic algebras Boole and \( \text{Pow}(N) \) (why are they non-isomorphic?).

If we want to have the algebra Boole as the ‘only’ model of the loose specification —up to isomorphism of course— we have to add further axioms. Let us add an axiom expressing that every element of the algebra is either true or false (thus ‘killing’ the model \( \text{Pow}(N) \)).
\[ P_7 \equiv \]

The extended loose specification \( \text{Mod}_\Sigma (\Phi \cup \{ P_7 \}) \) still has an unwanted model, namely the one element algebra. To rule this out we further add

\[ P_8 \equiv \]

Now it is easy to see that the loose specification \( (\Sigma, \Phi \cup \{ P_7, P_8 \}) \) characterizes the algebra \( \text{Boole} \) up to isomorphism, i.e. \( \text{Mod}_\Sigma (\Phi \cup \{ P_7, P_8 \}) \) is a monomorphic ADT containing \( \text{Boole} \).

In the previous example we succeeded in specifying an algebra up to isomorphism (which is the best we can get). The next example will show that we just happened to be lucky.

### 6.1.6 Example

Consider the following signature.

<table>
<thead>
<tr>
<th>Signature</th>
<th>( \Sigma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>nat</td>
</tr>
<tr>
<td>Constants</td>
<td>0 : nat</td>
</tr>
<tr>
<td>Operations</td>
<td>succ : nat ( \rightarrow ) nat</td>
</tr>
<tr>
<td></td>
<td>+ : nat ( \times ) nat ( \rightarrow ) nat</td>
</tr>
</tbody>
</table>

As the names suggest the intended algebra for this signature is the algebra \( \mathbb{N}_{0^{\mathbb{S}}} \) of natural numbers with the constant 0, the usual successor function \( \text{succ} : \mathbb{N} \rightarrow \mathbb{N}, \text{succ}(n) := n + 1 \), and addition \( + : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \).

Let us try to characterise \( \mathbb{N}_{0^{\mathbb{S}}} \) up to isomorphism by a loose specification \( (\Sigma, \Phi) \) with a suitable set of axioms \( \Phi \). Let us put into \( \Phi \) formulas \( P_1, \ldots, P_5 \) expressing that

1. 0 is not a successor,
2. \( \text{succ} \) is injective (one-to-one)
3. every number is either 0 or a successor,
4/5. addition can be defined from 0 and \( \text{succ} \) by primitive recursion in the usual way.

\[ P_1 \equiv \ \forall x (0 \neq \text{succ}(x)) \]

\[ P_2 \equiv \ \forall x, y (\text{succ} (x) = \text{succ}(y) \rightarrow x = y) \]
\[ P_3 \equiv \forall x \left( x = 0 \lor \exists y \left( x = \text{succ}(y) \right) \right) \]
\[ P_4 \equiv \forall x \left( x + 0 = x \right) \]
\[ P_5 \equiv \forall x, y \left( x + \text{succ}(y) = \text{succ}(x + y) \right) \]

Clearly the algebra \( N_{0s^+} \) is a model of \( \{ P_1, \ldots, P_5 \} \).

But there are still unwanted models. For example the \( \Sigma \)-algebra \( A \) with \( A_{\text{nat}} := \{0\} \times \mathbb{N} \cup \{1\} \times \mathbb{Z} \) with \( 0^A := (0, 0) \), \( \text{succ}^A((i, n)) := (i, n + 1) \), and \( +^A((i, n), (j, m)) := (\max(i, j), n + m) \). It is easy to check that \( A \) is a model of \( \{ P_1, \ldots, P_5 \} \).

Let us try to find an axiom killing this unwanted model. For example the axiom

\[ P_6 \equiv \forall x \left( x + x = x \rightarrow x = 0 \right) \]

holds in \( N_{0s^+} \), but doesn’t hold in \( A \) (for example \((1, 0) + (1, 0) = (1, 0)\)).

But there are still models of \( \{ P_1, \ldots, P_6 \} \) that are non-isomorphic to \( N_{0s^+} \). We could carry on by adding more and more axioms, but would never succeed in characterising \( N_{0s^+} \) up to isomorphism. This is due to the following theorem.

### 6.1.7 Theorem (Loewenheim-Skolem)

If a loose specification \((\Sigma, \Phi)\) has a countably infinite model \( A \), then it also has an uncountable model \( B \). In particular \( A \) and \( B \) are non-isomorphic, and therefore the abstract data type \( \text{Mod}_\Sigma(\Phi) \) is polymorphic.

**Remark.** The inability to characterise algebras by first-order formulas explains the term ‘loose specification’.

**Notation.** We will display loose specifications in a similar way as we display signatures. The axioms will be displayed without universal quantifier prefix, i.e. instead of

\[ \forall x, y \left( \text{succ}(x) = \text{succ}(y) \rightarrow x = y \right) \]

we write

\[ \text{succ}(x) = \text{succ}(y) \rightarrow x = y. \]

For example the loose specification \((\Sigma, \{ P_1, \ldots, P_3 \})\) of example 18 is displayed as

\[ \forall x, y \left( \text{succ}(x) = \text{succ}(y) \rightarrow x = y \right) \]
6.1.8 Definition

A loose specification \((\Sigma, \Phi)\) is called adequate for a \(\Sigma\)-algebra \(A\) if \(A \in \text{Mod}_\Sigma(\Phi)\).

\((\Sigma, \Phi)\) is called strictly adequate for \(A\) if \(A \in \text{Mod}_\Sigma(\Phi)\) and \(\text{Mod}_\Sigma(\Phi)\) is monomorphic, i.e. for any \(\Sigma\)-algebra \(B\)

\[B \in \text{Mod}_\Sigma(A) \quad \text{iff} \quad B \simeq A.\]

Hence, a strictly adequate loose specification characterises an algebra ‘up to isomorphism’.

For example, the loose specification in example 6.1.5 is strictly adequate for the algebra Boole, whereas the loose specification in example 6.1.6 is only adequate for the algebra \(\mathbb{N}_{\text{os}+}\), but not strictly adequate.

6.1.9 Definition

A loose specification \((\Sigma', \Phi')\) is called extension of a loose specification \((\Sigma, \Phi)\) of the signature \(\Sigma'\) is an expansion if the signature \(\Sigma\) (see definition 5.2.2) and \(\Phi' \supseteq \Phi\).

\((\Sigma', \Phi')\) is called persistent extension of \((\Sigma, \Phi)\) if \((\Sigma', \Phi')\) is an extension of \((\Sigma, \Phi)\) and addition for every closed \(\Sigma\)-formula \(P\) it holds that if \(\Phi' \models P\) then \(\Phi \models P\).
6.1.10 Lemma

Let the loose specification \((\Sigma', \Phi')\) be an extension of the loose specification \((\Sigma, \Phi)\) such that every \(\Sigma\)-algebra \(A\) satisfying \(\Phi\) can be expanded to a \(\Sigma'\)-algebra \(A'\) satisfying \(\Phi'\).

Then \((\Sigma', \Phi')\) is a persistent extension of \((\Sigma, \Phi)\).

Proof. Let \(P\) be a closed \(\Sigma\)-formula such that \(\Phi' \models P\). We have to show \(\Phi \models P\). To this end we take an arbitrary \(\Sigma\)-algebra \(A\) satisfying \(\Phi\) and have to show that \(A\) satisfies \(P\). By assumption there is an expansion \(A'\) of \(A\) such that \(A'\) satisfies \(\Phi'\). Since we assumed that \(\Phi' \models P\) we may conclude that \(A'\) satisfies \(P\). Since \(A'\) is an expansion of \(A\) it follows that \(A\) satisfies \(P\) too (one easily proves: if \(A\) is a \(\Sigma\)-algebra and \(A'\) is an expansion of \(A\), then for every \(\Sigma\)-formula \(P\) it holds that \(A \models P\) if and only if \(A' \models P\); the proof goes by structural induction on \(P\)).

6.2 Initial specifications

In order to increase the expressiveness of loose specifications we restrict their semantics to algebras that are initial in the class of all models of the loose specification. Unfortunately initial models do not exist for arbitrary loose specifications, as shown by the following example.

6.2.1 Example

Let \(\Sigma := \{\{s\}, \{a : s, b : s, c : s\}\}\) and \(\Phi := \{a = b \lor a = c\}\). We will show that the loose specification \((\Sigma, \Phi)\) has no initial model. Let \(A\) be a \(\Sigma\)-algebra that is initial for \(\text{Mod}_{\Sigma}(\Phi)\). We have show \(A \notin \text{Mod}_{\Sigma}(\Phi)\), i.e. the formula \(a = b \lor a = c\) is false in \(A\).

Define two \(\Sigma\)-algebra \(B, C\) by \(B_s = C_s = \{0, 1\}\) and

\[
a^B = b^B := 0, \quad c^B := 1.
\]

\[
a^C = c^B := 0, \quad b^C := 1.
\]

Obviously in both algebras the formula \(a = b \lor a = c\) is true, i.e. \(B, C \in \text{Mod}_{\Sigma}(\Phi)\). Since we assumed \(A\) to be initial for \(\text{Mod}_{\Sigma}(\Phi)\), we have homomorphisms

\[
\varphi: A \rightarrow B, \quad \psi: A \rightarrow C
\]

Using the homomorphic property of \(\varphi\) and \(\psi\) we see

\[
\varphi(a^A) = a^B \neq c^B = \varphi(c^A), \text{ hence } a^A \neq c^A
\]

\[
\psi(a^A) = a^C \neq b^C = \psi(b^A), \text{ hence } a^A \neq b^A
\]
Hence the formula $a = b \lor a = c$ is false in $A$.

In order to guarantee the existence of such initial algebras we now drastically restrict the form of axioms.

**Notation.** Recall that an equation over a signature $\Sigma$ is a formulas of the form

$$t_1 = t_2$$

where $t_1, t_2$ are $\Sigma$-terms of the same sort.

If $E$ is a set of equations over $\Sigma$ we set

$$\forall E := \{ \forall(t_1 = t_2) \mid t_1 = t_2 \in E \}$$

### 6.2.2 Definition

Let $E$ be a set of equations over a signature $\Sigma$. We define

$$T_E(\Sigma) := T_{\text{Mod}_\Sigma(\forall E)}(\Sigma)$$

(cf. the proof of theorem 5.3.7), i.e. $T_E(\Sigma) = T(\Sigma)/\sim_E$ where for closed $\Sigma$-terms $t_1, t_2$

$$t_1 \sim_E t_2 \iff \forall E \models t_1 = t_2$$

Hence the elements of $T_E(\Sigma)$ are equivalence classes of closed terms, where two closed terms are equivalent iff they have the same value in all models of $\forall E$.

### 6.2.3 Theorem

Let $E$ be a set of equations over a signature $\Sigma$. Then the $\Sigma$-algebra $T_E(\Sigma)$ is initial in $\text{Mod}_\Sigma(\forall E)$.

For every $A \in \text{Mod}_\Sigma(\forall E)$ the unique homomorphism $\varphi_A : T_E(\Sigma) \to A$ is given by

$$\varphi_A([t]_{\sim_E}) = t^A$$

for each $t \in T(\Sigma)$.

**Proof.** In theorem 6.2.3 it was proved that $T_E(\Sigma)$ is initial for $\text{Mod}_\Sigma(\forall E)$, and that $\varphi_A$ is the unique homomorphism from $T_E(\Sigma)$ to $A$. Hence it only remains to show that $T_E(\Sigma)$ is a model of $\forall E$. Take an equation $t_1 = t_2 \in E$. We have to prove that the formula $\forall (t_1 = t_2)$ is true in $T_E(\Sigma)$.

In preparation of proving this we first show
\[ t_1 \theta \sim_E t_2 \theta \quad \text{for all substitutions } \theta : X \to T(\Sigma) \quad (+) \]

where the congruence \( \sim_E \) is defined as in definition 6.2.2 above and \( X := \text{FV}(t_1 = t_2) \).

In order to prove (+) we take an arbitrary model \( A \) of \( \forall E \) and show that the equation \( t_1 \theta = t_2 \theta \) is true in \( A \), i.e. \( (t_1 \theta)^A = (t_2 \theta)^A \). This can be seen as follows:

\[ (t_1 \theta)^A \overset{2A}{=} t_1^A, \theta^A \overset{A\vdash \forall(t_1 \theta = t_2)}{=} t_2^A, \theta^A \overset{2A}{=} (t_1 \theta)^A \]

Having proved (+) it is now easy to prove that the formula \( \forall(t_1 = t_2) \) is true in \( T_E(\Sigma) \). Let \( \alpha : X \to T_E(\Sigma) \) be a variable assignment. We have to prove

\[ t_1^{T_E(\Sigma), \alpha} = t_2^{T_E(\Sigma), \alpha} \quad (++) \]

Note that for every variable \( x \in X \), \( \alpha(x) \) is an \( \sim_E \)-equivalence class. For every \( x \in X \) chose a term \( \theta(x) \in \alpha(x) \). This defines a substitution \( \theta : X \to T(\Sigma) \). By definition we have \( \alpha(x) = [\theta(x)]_{\sim_E} \) for every \( x \in X \), i.e. \( \alpha = [\cdot]_{\sim_E} \circ \theta \). Note also that \([\cdot]_{\sim_E} : T(\Sigma) \to T_E(\Sigma) (= T(\Sigma) / \sim_E) \) is a homomorphism. Finally note that the substitution \( \theta \) can also be viewed as a variable assignment for the closed term algebra \( T(\Sigma) \). Barring all this in mind we can now prove (++).

We have

\[ t_1^{T_E(\Sigma), \alpha} = t_1^{T_E(\Sigma), [\cdot]_{\sim_E} \circ \theta} \overset{6.1.2}{=} [t_1^{T(\Sigma), \theta}]_{\sim_E} \overset{\text{coursework 1}}{=} [t_1 \theta]_{\sim_E} \]

and similarly \( t_2^{T_E(\Sigma), \alpha} = [t_2 \theta]_{\sim_E} \). Since by (+) we have \([t_1 \theta]_{\sim_E} = [t_2 \theta]_{\sim_E} \), we have proved (++).

### 6.2.4 Definition

Let \( \Sigma \) be a signature and \( E \) a set of equations over \( \Sigma \). Then

\[ \text{Init–Spec}(\Sigma, E) \]

is called an **initial specification**.

A \( \Sigma \)-algebra \( A \) is a **model** of \( \text{Init–Spec}(\Sigma, E) \) if it is an initial model of the loose specification \((\Sigma, \forall E)\), i.e. \( A \) is initial in \( \text{Mod}_\Sigma(\forall E) \).

We let \( \text{Init–Mod}_\Sigma(E) \) denote the class of all models of \( \text{Init–Spec}(\Sigma, E) \).

We also say that \( \text{Init–Spec}(\Sigma, E) \) is an **adequate initial specification** for the \( \Sigma \)-algebra \( A \) if \( A \in \text{Init–Mod}_\Sigma(E) \).
6.2.5 Theorem

Let $T_E(\Sigma)$ be an initial specification. Then for any $\Sigma$-algebra $A$ the following conditions are equivalent:

(i) $A$ is a model of $\text{Init-Spec}(\Sigma, E)$.

(ii) $A$ is initial in $\text{Mod}_E(\forall E)$ (i.e. $A$ is an initial model of the loose specification $(\Sigma, \forall E)$).

(iii) $A \simeq T_E(\Sigma)$.

(iv) $A$ is generated and for any two closed $\Sigma$-terms $t_1, t_2$ of the same sort we have

$$A \models t_1 = t_2 \iff \forall E \models t_1 = t_2$$

(i.e. $t_1$ and $t_2$ have the same value in $A$ iff they have the same value in all models of $\forall E$).

(v) $A$ is a generated model of $\forall E$ and for any two closed $\Sigma$-terms $t_1, t_2$ of the same sort we have

$$A \models t_1 = t_2 \Rightarrow \forall E \models t_1 = t_2$$

In particular $\text{Init--Mod}_E(\Sigma)$ is a monomorphic ADT containing $T_E(\Sigma)$.

Proof. ‘(i)$\Leftrightarrow$(ii)’ is just a repetition of definition 6.2.4 above.

‘(ii)$\Leftrightarrow$(iii)’ follows from theorem 6.2.3 and the fact that initial algebras are unique up to isomorphism (coursework 2).

‘(iii)$\Rightarrow$(iv)’. Let $\varphi: T_E(\Sigma) \to A$ be an isomorphism. By lemma 6.1.2 and the fact that $t^{T_E(\Sigma)} = [t]$ we have $t^A = \varphi([t])$ for all closed $\Sigma$-terms. This clearly implies (iv).

‘(iv)$\Rightarrow$(v)’. Assume that (iv) holds. We have to show that $A$ is a model of $\forall E$.

Let $t_1 = t_2$ be an equation in $E$ and $\alpha: X \to A$ a variable assignment, where $X := \text{FV}(t_1 = t_2)$. We have to show $t_1^{A, \alpha} = t_2^{A, \alpha}$. Since $A$ is generated we have for every variable $x \in X$ a closed term $\theta(x)$ with $\alpha(x) = \theta(x)^A$, i.e. $\alpha = \theta^A$. According to the substitution theorem 2.4.6 we have

$$t_1^{A, \alpha} = t_1^{A, \theta^A} = (t_1, \theta)^A$$

and similarly $t_2^{A, \alpha} = (t_2, \theta)^A$. Since $t_1 = t_2$ is an equation in $E$ we have $\forall E \models t_1 = t_2 = t_2 \theta$ (we showed this in detail in the proof of theorem 6.2.3). Hence $(t_1, \theta)^A = (t_2, \theta)^A$ by assumption (iv). Therefore $t_1^{A, \alpha} = t_2^{A, \alpha}$.

‘(v)$\Rightarrow$(iii)’. Assume that (v) holds. Since by assumption $A \in \text{Mod}_E(\forall E)$ we now by initiality of $T_E(\Sigma)$ that there is unique homomorphism $\varphi: T_E(\Sigma) \to A$. Using once more the fact that $t^A = \varphi([t])$ for all closed $\Sigma$-terms (see ‘(iii)$\Rightarrow$(iv) above) it is plain that our assumption (v) implies that $\varphi$ is bijective.
6.2.6 Example

Let $\Sigma := \{\{\text{nat}\}, \{0 : \text{nat}, \text{succ} : \text{nat} \to \text{nat}, + : \text{nat} \times \text{nat} \to \text{nat}\}\}$ and $E := \{x + 0 = x, \ x + \text{succ}(y) = \text{succ}(x + y)\}$.

We display the initial specification $\text{Init-Spec}(\Sigma, E)$ by

<table>
<thead>
<tr>
<th>Init Spec</th>
<th>NAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>nat</td>
</tr>
<tr>
<td>Constants</td>
<td>0 : nat</td>
</tr>
<tr>
<td>Operations</td>
<td>succ : nat \to nat</td>
</tr>
<tr>
<td></td>
<td>+ : nat \times nat \to nat</td>
</tr>
<tr>
<td>Variables</td>
<td>$x, y : \text{nat}$</td>
</tr>
<tr>
<td>Equations</td>
<td>$x + 0 = x$</td>
</tr>
<tr>
<td></td>
<td>$x + \text{succ}(y) = \text{succ}(x + y)$</td>
</tr>
</tbody>
</table>

The elements of the carrier set of $T_E(\Sigma)$ are the equivalence classes $[0], [\text{succ}(0)], [\text{succ}(\text{succ}(0))], \ldots$. One has for instance

\[
[0] = \{0, 0 + 0, (0 + 0) + 0, \ldots\}
\]

= the set of closed terms built from 0 and +

\[
[\text{succ}(0)] = \{\text{succ}(0), \text{succ}(0) + 0, 0 + \text{succ}(0), \ldots\}
\]

= the set of closed terms built from 0 and + and exactly one occurrence of succ

In the next section we will develop tools for proving these facts easily.

We will also see that the initial specification $\text{Init-Spec}(\Sigma, E)$ is adequate for the standard algebra of natural numbers with 0, successor operation and addition, i.e. this algebra is a model of the initial specification.

6.2.7 Example

Let us modify example 6.2.6 as follows. We extend the initial specification by an operation $- : \text{nat} \times \text{nat} \to \text{nat}$ and add the two equations

$x - 0 = x$
\[ \text{succ}(x) - \text{succ}(y) = x - y \]

Let \( E \) be this extended set of equations. We also expand the standard algebra of natural numbers discussed in 6.2.6 by interpreting the new operation, \(-\), by the operation \(-\) : \( \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \)

\[
n - m := \begin{cases} n - m & \text{if } n \geq m \\ 0 & \text{otherwise} \end{cases}
\]

Let \( A \) be this expanded algebra. Clearly the new equations are valid under this interpretation of \(-\), i.e. \( A \) is a model of the loose specification \( (\Sigma, \forall E) \), but not an initial model, that is \( A \) is not a model of the initial specification \( \text{Init-Spec}(\Sigma, E) \), since, for example in \( A \) the equation

\[ 0 - \text{succ}(0) = 0 \]

holds, whereas clearly

\[ \forall E \not\models 0 - \text{succ}(0) = 0 \]

Therefore 6.2.5 (v) does not hold.

6.2.8 Exercises

(a) Find a model of \( \forall E \) where the equation \( 0 - \text{succ}(0) = 0 \) does not hold.

(b) Find terms \( t, t' \) such that the algebra \( A \) is a model of the initial specification \( \text{Init-Spec}(\Sigma, E') \), where \( E' := E \cup \{ t = t' \} \).

(c) Give an informal description of ‘the’ model of the initial specification \( \text{Init-Spec}(\Sigma, E') \).

6.2.9 Example

<table>
<thead>
<tr>
<th>Init Spec</th>
<th>SET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>nat, set</td>
</tr>
<tr>
<td>Constants</td>
<td>0 : nat</td>
</tr>
<tr>
<td></td>
<td>empty : set</td>
</tr>
<tr>
<td>Operations</td>
<td>succ : nat ( \rightarrow ) nat</td>
</tr>
<tr>
<td></td>
<td>insert : set ( \times ) nat ( \rightarrow ) set</td>
</tr>
<tr>
<td>Variables</td>
<td>( x, y : ) nat, ( s : ) set</td>
</tr>
<tr>
<td>Equations</td>
<td>insert(insert(s, x), x) = insert(s, x)</td>
</tr>
<tr>
<td></td>
<td>insert(insert(s, x), y) = insert(insert(s, y), x)</td>
</tr>
</tbody>
</table>
Let \( A \) be the classical algebra of finite set of natural numbers with the obvious interpretation of the constants and operation. Clearly \( A \) is a generated model of the (generalised) equations \( SET \). It is also not hard to see that if two closed terms have the same value in \( A \), then they have the same value in all models of the equations. Hence, by theorem 6.2.5 \((v)\Rightarrow (i)\), \( A \) is a model of \( SET \).

### 6.2.10 Example

We wish to specify a simple editor. The editor should be able to edit a file by performing the following possible actions:

- **write**\((x)\): insert the character \( x \) immediately to the left of the cursor;
- **\( \rightarrow \)**: move the cursor one position to the right;
- **\( \leftarrow \)**: move the cursor one position to the left;
- **del**: delete the character immediately to the right of the cursor.

Consider for example the file

\[ \textit{edit}\mid\textit{or} \]

where the \( \mid \) representing the cursor. After entering \( \leftarrow \) we get

\[ \textit{edi}\mid\textit{or} \]

and entering **del** thereafter yields

\[ \textit{edi}\mid\textit{or} \]

Finally we write the character \( t \) and obtain

\[ \textit{edit}\mid\textit{or} \]

It is convenient to represent a file with a cursor by a pair of lists of characters representing the part of the file left and right to the cursor, where the left part is represented in reverse order. Then the actions in the example above create the following sequence of representations of files:

\[ ([r, i, d, e], [o, r]) \]
\[ ([i, d, e], [r, o, r]) \]
\[ ([i, d, e], [o, r]) \]
\[ ([t, i, d, e], [o, r]) \]
We see that only the elementary operations of adding an element in front of a list, or removing the first element of a list are needed.

To create a file we will use the constructor \(^3\) cf: charlist \(\times\) charlist \(\rightarrow\) file and for creating lists of characters the usual constructors nil: charlist and cons: char \(\times\) charlist \(\rightarrow\) charlist.

Typing a character or a command will be modeled by an operation type: input \(\times\) file \(\rightarrow\) file, where the elements of the sort input are commands such as \(\triangleright\), or write\((x)\), where \(x\) is a character (so, \(\triangleright\) is a constant and write is a unary operation).

In order to keep things simple we stipulate that typing the command \(\triangleright\) whilst the cursor is at the right end of the file will not modify the file (similarly del and for the right end).

The following initial specification formalizes our ideas:

<table>
<thead>
<tr>
<th>Init Spec</th>
<th>EDITOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>char, charlist, input, file</td>
</tr>
<tr>
<td>Constants</td>
<td>newfile: file</td>
</tr>
<tr>
<td></td>
<td>(a, \ldots, z, \sim: ) char</td>
</tr>
<tr>
<td></td>
<td>nil: charlist</td>
</tr>
<tr>
<td></td>
<td>(\triangleright, &lt;), del: input</td>
</tr>
<tr>
<td>Operations</td>
<td>cons: char (\times) charlist (\rightarrow) charlist</td>
</tr>
<tr>
<td></td>
<td>cf: charlist (\times) charlist (\rightarrow) file</td>
</tr>
<tr>
<td></td>
<td>write: char (\rightarrow) input</td>
</tr>
<tr>
<td></td>
<td>type: input (\times) file (\rightarrow) file</td>
</tr>
<tr>
<td>Variables</td>
<td>(x:) char, (l, r:) charlist</td>
</tr>
<tr>
<td>Equations</td>
<td>newfile = cf(nil, nil)</td>
</tr>
<tr>
<td></td>
<td>type(write((x)), cf((l, r))) = cf((cons(x, l), r))</td>
</tr>
<tr>
<td></td>
<td>type((\triangleright), cf((l, nil))) = cf((l, nil))</td>
</tr>
<tr>
<td></td>
<td>type((\triangleright), cf((l, cons(x, r)))) = cf((cons(x, l), r))</td>
</tr>
<tr>
<td></td>
<td>type(&lt;, cf((nil, r))) = cf((nil, r))</td>
</tr>
<tr>
<td></td>
<td>type(&lt;, cf((cons(x, l), r))) = cf((l, cons(x, r)))</td>
</tr>
<tr>
<td></td>
<td>type(del, cf((l, nil)) = cf((l, nil))</td>
</tr>
<tr>
<td></td>
<td>type(del, cf((l, cons(x, r))) = cf((l, r))</td>
</tr>
</tbody>
</table>

\(^3\)An operation \(f\) for which in a given initial specification no equation of the form \(f(\ldots) = \ldots\) exists is called constructors. Constructors do not really compute, but rather construct data. Typical examples are the successor succ for natural numbers and the constructor cons for lists.
6.2.11 Exercises

(a) Extend the editor specified in example 6.2.10 by a command backspace that deletes the character immediately to the left of the cursor.

(b) Extend the editor by a character for a new line and a command that deletes all text in one line to the right of the cursor.

(c) Improve (b) by putting the deleted text into a buffer and providing a command for yanking back to the right of the cursor the text currently in the buffer.

We close this section with a theorem characterising equality between open terms in initial models.

6.2.12 Theorem

Let \( A \) be a model of the initial specification \( \text{Init–Spec}(\Sigma, E) \). Then for two terms \( t_1, t_2 \) of the same sort the following statements are equivalent:

(i) \( A \models \forall(t_1 = t_2) \).

(ii) \( B \models \forall(t_1 = t_2) \) for all generated models \( B \) of \( \forall E \).

(iii) \( \forall E \models t_1 \theta = t_2 \theta \) for all substitutions \( \theta: X \to T(\Sigma) \), where \( X := \text{FV}(t_1 = t_2) \).

Proof. ‘(i)\( \Rightarrow \) (ii)’. Assume \( A \models \forall(t_1 = t_2) \), and let \( B \) be a generated model of \( \forall E \). By initiality of \( A \) there is a homomorphism \( \varphi: A \to B \). Since \( A \) and \( B \) are both generated \( \varphi \) is surjective. Hence clearly \( B \models \forall(t_1 = t_2) \).

‘(ii)\( \Rightarrow \) (i)’. Obvious, since \( A \) is generated.

‘(i)\( \Leftrightarrow \) (iii)’. Since \( A \) is generated \( A \models \forall(t_1 = t_2) \) is equivalent to \( \forall E \models t_1 \theta = t_2 \theta \) for all substitutions \( \theta: X \to T(\Sigma) \), and by theorem 6.2.5 (iv) the latter is equivalent to (iii).

6.3 Specification languages

The loose and initial specifications we studied so far are called \textbf{atomic specifications}. Starting from these atomic specifications one can build more complex specifications by certain operations. Examples of such operations are:

**Union** If \( \text{Spec}_1 \) and \( \text{Spec}_2 \) are specifications then \( \text{Spec}_1 + \text{Spec}_2 \) is a specification.

The signature of \( \text{Spec}_1 + \text{Spec}_2 \) is \( \Sigma_1 \cup \Sigma_2 \), where \( \Sigma_i \) is the signature of \( \text{Spec}_i \) (assuming that \( \Sigma_1 \) and \( \Sigma_2 \) are ‘compatible’).

A \( \Sigma_1 \cup \Sigma_2 \)-algebra \( A \) is a model of \( \text{Spec}_1 + \text{Spec}_2 \) if and only if \( A |_{\Sigma_1} \) is a model of \( \text{Spec}_1 \) and \( A |_{\Sigma_2} \) is a model of \( \text{Spec}_2 \).
Restriction If Spec is a specification with signature \( \Sigma \) and \( \Sigma_0 \) is a subsignature of \( \Sigma \) then \( \text{Spec}|_{\Sigma_0} \) is a specification with signature \( \Sigma_0 \).

A \( \Sigma_0 \)-algebra \( A \) is a model of \( \text{Spec}|_{\Sigma_0} \) if and only if \( A = B|_{\Sigma_0} \) for some model \( B \) of Spec.

Other fundamental construction principles for specifications are renaming, parametrization, modularization, and inheritance.

Describing abstract data types by atomic specifications is called specification-in-the-small, whereas describing them by complex specifications is called specification-in-the-large.

6.3.1 Example

In example 6.2.6 we produced an initial specification of the aldbra of natural numbers with 0, successor and addition. In example 6.2.7 and exercise 6.2.8 we extended this by ‘cut-off’ subtraction, \( n - m \), which, somewhat unnaturally, returns 0 if \( n < m \).

We will now use the specification construct “\( + \)” (union) to provide a specification of the algebra of natural numbers with 0, successor, addition and subtraction, but leaving it open what the the result of \( n - m \) for \( n < m \) is.

Let \( \text{Init–Spec}(\Sigma, E) \) be the initial specification of example 6.2.6 (specifying the natural numbers with 0 and addition). Let \( (\Sigma', E') \) be the loose specification, where \( \Sigma' \) is \( \Sigma \) expanded by the operation \( - \) (minus), and \( E' \) consists of the equations

\[
\begin{align*}
x - 0 &= x \\
\text{succ}(x) - \text{succ}(y) &= x - y
\end{align*}
\]

Then the models of the specification

\[
\text{Init–Spec}(\Sigma, E) + (\Sigma', E')
\]

are, up to isomorphism, exactly those \( \Sigma' \)-algebras, where the natural numbers, addition and \( n - m \) for \( n \geq m \) have their standard meaning, but the result of \( n - m \) for \( n < m \) can be any natural number.

A specification language is a (formal or informal) language to denote atomic and complex specifications. Here is a selection of some of the most important specification languages currently in use:

VDM, Z Specification languages using set-theoretic notations. VDM and Z are the most widely used specification languages in industry.


ASL A kernel language for algebraic specifications.

Extended ML A specification language for functional programming languages, in particular ML.

Spectrum A very general specification language based on partial algebras, higher order constructs and polymorphism.

Larch A State oriented specification language. Contains an elaborate proof checker.

CCS, CSP Formal languages for specifying concurrent processes.

UML A design and modeling language for object oriented programming.

6.3.2 Remark

As already mentioned in the introduction to chapter 5 specifications as discussed in this course are usually called algebraic or axiomatic specifications. Sometimes they are also called functional specifications, because operations are modeled as functions on data, and they match well with functional programming languages (LISP, SCHEME, ML, HASKELL, e.t.c.). However in (industrially) applied specification languages (VDM, Z) it is common to write specifications in an imperative or state oriented style. In such specifications the execution of an operation may change the state of an algebra (our algebras don’t have a state). For example if our specification of an editor (6.2.10) were rewritten in imperative style the sort file could be suppressed instead one would speak about the current state of the editor. The state oriented style leads in some cases to shorter specifications which also seem to be closer to implementations, however the model theory of state oriented specifications is more complicated (and consequently often omitted in the literature).

6.4 Summary and Exercises

In this section the following notions and results were most important.

- Theorem 6.1.3 stating that isomorphic algebras have the same properties, i.e. the same closed formulas are true;

\(^4\)Algebras with state are often called evolving algebras (Börger), or abstract state machines (Gurevich).
• the notion of a *loose specification* \((\Sigma, \Phi)\) and its class of models \(\text{Mod}_\Sigma(\Phi)\) (6.1.1), which is an ADT (6.1.4);

• *initial specifications* \(\text{Init–Spec}(\Sigma, E)\) and their model class \(\text{Init–Mod}_\Sigma(E)\) (6.2.4); in coursework 2 we proved that \(\text{Init–Mod}_\Sigma(E)\) is a monomorphic ADT; an initial specification \(\text{Init–Spec}(\Sigma, E)\) is called *adequate* for a \(\Sigma\)-algebra \(A\) if \(A \in \text{Init–Mod}_\Sigma(E)\);

• the algebra \(T_E(\Sigma)\) (6.2.2) and the fact that it is a model of \(\text{Init–Spec}(\Sigma, E)\) (6.2.3);

• Theorem 6.2.5 characterising models of initial specifications; particularly useful is (v) which can be rephrased as follows:

Let \(A\) be a generated \(\Sigma\)-algebra. Then an initial specification \(\text{Init–Spec}(\Sigma, E)\) is adequate for \(A\) if and only if

(i) \(A \models \forall E\), i.e. all equations of \(E\) are true in \(A\), and

(ii) for any closed \(\Sigma\)-terms \(t_1, t_2\), if \(t_1^A = t_2^A\) then \(E \models t_1 = t_2\).

Exercises.

1. Consider the following loose specification \(\text{LIST(BOOLE)}::\)

<table>
<thead>
<tr>
<th>Loose Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sorts</strong></td>
</tr>
<tr>
<td><strong>Constants</strong></td>
</tr>
<tr>
<td><strong>Operations</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Variables</strong></td>
</tr>
<tr>
<td><strong>Axioms</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Let \(\Sigma\) be the signature of \(\text{LIST(BOOLE)}::\)

Show that the following \(\Sigma\)-algebra \(A\) is a model of \(\text{LIST(BOOLE)}::\):

\[
A_{\text{boole}} := \{ \#t, \#f \},
\]

\(A_{\text{list}} := \) the set of finite lists of boolean values.
\[ T^A := \#t, \]
\[ F^A := \#f, \]
\[ \text{nil}^A := [] \text{ (the empty list).} \]
\[ \text{cons}^A(a, [a_1, \ldots, a_n]) := [a, a_1, \ldots, a_n] \]
\[ \text{first}^A(l) := \begin{cases} 
\#f & \text{if } l = [] \\
\text{the first element of } l & \text{otherwise}
\end{cases} \]
\[ \text{rest}^A(l) := \begin{cases} 
[] & \text{if } l = [] \\
\text{the result of removing the first element from } l & \text{otherwise}
\end{cases} \]

2. Is the \(\Sigma\)-algebra \(A\), defined in the previous exercise, initial in the class of all models of \(\text{LIST(BOOLE)}\)? Justify your answer.
7 Term rewriting

In section 6.2 we studied initial specifications Init–Spec(Σ, E), where E is a set of equations. One of the main theoretical results was that in a model A of Init–Spec(Σ, E) two closed terms $t_1, t_2$ have the same value if and only if the equation $t_1 = t_2$ is a logical consequence of $\forall E$ (theorem 6.2.5 (iv)). Since, by Gödel’s soundness and completeness theorem, 3.4.1, 3.4.2, logical consequence is equivalent to provability this means

$$A \models t_1 = t_2 \quad \text{if and only if} \quad \forall E \vdash t_1 = t_2$$

In this chapter we will see that in order to derive a sequent $\forall E \vdash t_1 = t_2$ only the rules for universal quantification 3.2.1 and the quality rules 3.3 are needed (Birkhoff’s Theorem 7.1.5). An important consequence of this is the fact that in many practically relevant cases derivations can be mechanised using term rewriting yielding a procedure to decide whether or not $\forall E \vdash t_1 = t_2$. Term rewriting also automatically yields correct implementations of many initial specification of practical interest (rapid prototyping 7.5).

**Assumption.** From now on we will always assume that all signatures $\Sigma$ considered are finite and are such that for every sort $s$ there is at least one closed term of sort $s$. This in particular will imply that for every $\Sigma$-algebra $A$ the carrier sets $A_s$ are all nonempty (since $t^A \in A_s$, where $t$ is a closed term of sort $s$).

This assumption is not a severe restriction, because it is fulfilled for any signature of interest, but it avoids certain strange phenomena. For example, if in a $\Sigma$-algebra $A$ we have $a^A \neq b^A$, where $a, b$ are constants, and $x$ is a variable of sort $s$ where $A_s = \emptyset$, then in $A$ the formula $\forall x (a = b)$ would be true although the equation $a = b$ is false. Empty sorts would also cause some complications in the formulation of the derivation calculus we are going to study now.

7.1 Equational logic

7.1.1 Definition

The **deduction rules** of **equational logic** with respect to a given signature $\Sigma$ are the following.

**Reflexivity**

$$
\frac{t = t}{t = t}
$$

**Symmetry**

$$
\frac{t_1 = t_2}{t_2 = t_1}
$$

**Transitivity**

$$
\frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3}
$$

**Compatibility**

$$
\frac{t_1 = t'_1 \quad \cdots \quad t_n = t'_n}{f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)}
$$

**Instance**

$$
\frac{t_1 = t_2}{t_1 \theta = t_2 \theta}
$$
In these rules \( t, t', t'_1 \in T(\Sigma, X) \), \( \theta : X \rightarrow T(\Sigma, Y) \) is a substitution, and \( f \) is an operation of the signature \( \Sigma \). We write
\[
\vdash_E t_1 = t_2
\]
if the equation \( t_1 = t_2 \) can be derived starting with equations in \( E \) and using the rules above. In this case we also say that the equation \( t_1 = t_2 \) is **derivable from** \( E \) (in equational logic).

### 7.1.2 Example

Let \( E := \{ x + 0 = x, \ x + \text{succ}(y) = \text{succ}(x + y) \} \). Then \( \vdash_E 0 + \text{succ}(0) = \text{succ}(0) \) as the following derivation shows.

\[
\frac{x + \text{succ}(y) = \text{succ}(x + y)}{0 + \text{succ}(0) = \text{succ}(0 + 0)} \quad \text{Inst} \quad \frac{x + 0 = x}{0 + 0 = 0} \quad \text{Inst} \quad \frac{\text{succ}(0 + 0) = \text{succ}(0)}{\text{Comp}} \quad \frac{0 + \text{succ}(0) = \text{succ}(0)}{\text{Trans}}
\]

### 7.1.3 Lemma

If \( \vdash_E t_1 = t_2 \) then \( \forall E \vdash_m t_1 = t_2 \).

**Proof.** Trivial induction on the built-up of equational derivations. Let us verify, for example, the rule **Instance**:

\[
\frac{t_1 = t_2}{t_1 \theta = t_2 \theta}
\]

where, say, \( \theta = \{r_1/x_1, \ldots, r_n/x_n\} \). By induction hypothesis we have \( \forall E \vdash_m t_1 = t_2 \). By \( n \)-fold application of the rule \( \forall^+ \) we obtain \( \forall E \vdash_m \forall x_1, \ldots, x_n (t_1 = t_2) \), and applying \( n \)-times the rule \( \forall^- \) yields \( \forall E \vdash_m (t_1 = t_2)\{r_1/x_1, \ldots, r_n/x_n\} \), that is, \( \forall E \vdash_m t_1 \theta = t_2 \theta \).

### 7.1.4 Soundness Theorem

If \( \vdash_E t_1 = t_2 \) then \( \forall E \models \forall (t_1 = t_2) \).

**Proof.** Assume \( \vdash_E t_1 = t_2 \). Then, by lemma 7.1.3 above, \( \forall E \vdash t_1 = t_2 \), and therefore, using the rule \( \forall^+ \) repeatedly, \( \forall E \vdash \forall (t_1 = t_2) \). From the soundness theorem for natural deduction, theorem 3.4.1, it follows \( \forall E \models \forall (t_1 = t_2) \).
7.1.5 Completeness Theorem (Birkhoff)

If \( \forall E \models \forall (t_1 = t_2) \) then \( \vdash_E t_1 = t_2 \).

**Proof.** Let \( \Sigma \) be the signature of \( E \cup \{ t_1 = t_2 \} \) and let \( V \) be the set of all variables for the sorts of \( \Sigma \). Then \( T(\Sigma, V) \) is the set of all \( \Sigma \)-terms which, according to definition 2.2.7, can be viewed as a \( \Sigma \)-algebra. For \( \Sigma \)-terms \( t_1, t_2 \) of the same sort we define

\[ t_1 =_E t_2 \quad \iff \quad \vdash_E t_1 = t_2 \]

Because of the rules Reflection, Symmetry, Transitivity, and Compatibility this defines a congruence \( =_E \) on \( T(\Sigma, V) \). Set

\[ A := T(\Sigma, V)/=_E \]

The following fact will be crucial in the rest of the proof. Let \( t \) be a \( \Sigma \)-term and \( \alpha : X \rightarrow A \) a variable assignment, where \( X \supseteq \text{FV}(t) \). Then

\[ t^{A, \alpha} = [\alpha t] =_E \quad \text{(+) for any substitution } \alpha : X \rightarrow T(\Sigma, V), \text{ such that } \alpha(x) \in \alpha(x) \text{ for all } x \in X. \]

Proof of (\text{+}). Note that the condition on \( \alpha \) means that \( \alpha(x) = [\alpha(x)] =_E \) for all \( x \in X \), i.e.

\( \alpha = [\cdot] =_E \circ \alpha \). Hence

\[ t^{A, \alpha} = t^{A, [\cdot] =_E \circ \alpha} \overset{\text{6.1.2}}{=} [t^{T(\Sigma, V), \theta}] =_E \overset{\text{coursework 2}}{=} [\theta] =_E \]

We now use (\text{+}) to show the following.

\[ A \models \forall (t_1 = t_2) \iff \vdash_E t_1 = t_2 \text{ (++) for all } \Sigma \text{-terms } t_1, t_2 \text{ of the same sort.} \]

Proof of (\text{++}).

\( \Rightarrow \). Assume \( A \models \forall (t_1 = t_2) \). Using (\text{+}) with \( X := \text{FV}(t_1 = t_2) \), \( \alpha(x) := [x] =_E \), and \( \theta(x) := x \) we get

\[ \vdash_E t_1 =_E t_2 \]

\( \Leftarrow \). Assume \( \vdash_E t_1 = t_2 \). Let \( \alpha : X \rightarrow A \) be a variable assignment. We have to show \( t_1^{A, \alpha} = t_2^{A, \alpha} \).

Define a substitution \( \theta : X \rightarrow T(\Sigma, V) \) by selecting from every \( =_E \)-equivalence class \( \alpha(x) \) (\( x \in X \)) an element \( \theta(x) \in \alpha(x) \) (we did a similar thing in the proof of Theorem 6.2.3). Using (\text{+}) a second time we get \( t_i^{A, \alpha} = [t_i] =_E \) for \( i = 1, 2 \). From the assumption \( \vdash_E t_1 = t_2 \) we may infer \( \vdash_E t_1 \theta = t_2 \theta \) using the rule Instance. Hence \([t_1] =_E \) and therefore \( t_1^{A, \alpha} = t_2^{A, \alpha} \). Thus (\text{++}) is proved.

From (\text{++}), \( \Leftarrow \) it follows that \( A \) is a model of \( \forall E \), since

\[ \forall (t_1 = t_2) \in \forall E \quad \Rightarrow \quad t_1 = t_2 \in E \quad \Rightarrow \quad \vdash_E t_1 = t_2 \overset{\text{++}}{=} A \models \forall (t_1 = t_2) \]

It is now easy to prove the theorem. Assume \( \forall E \models \forall (t_1 = t_2) \). Then \( A \models t_1 = t_2 \), since \( A \) is a model of \( \forall E \). Consequently \( \vdash_E t_1 = t_2 \), by (\text{++}), \( \Rightarrow \).
7.2 Term rewriting systems

Now we show how equational logic can be mechanized.

7.2.1 Definition

A term rewriting system over a signature $\Sigma$ is a finite set $R$ of rewrite rules $r \mapsto l$, where $r$ and $l$ are $\Sigma$-terms, such that

(i) $l$ is not a variable,

(ii) $\text{FV}(r) \subseteq \text{FV}(l)$.

Given a term rewriting system $R$ we define a binary relation $\rightarrow_R$ on the set of $\Sigma$-terms by

$t \rightarrow_R t' \iff t \equiv u[l\theta/x] \text{ and } t' \equiv u[r\theta/x]$ for some rewrite rule $l \mapsto r \in R$, some $\Sigma$-term $u$ with exactly one occurrence of some variable $x$, and some substitution $\theta: X \rightarrow T(\Sigma, Y)$

In other words, $t \rightarrow_R t'$ holds iff $t'$ can be obtained from $t$ by replacing some subterm of the form $l\theta$ by $r\theta$, where $l \mapsto r \in R$ and $\theta$ is a substitution. We also say that the subterm $l\theta$ matches $l$, or is an instance of $l$.

If $t \rightarrow^*_R t'$ we say rewrites to $t'$.

We call $\rightarrow_R$ the term rewriting relation generated by $R$.

Furthermore we define

$t \rightarrow^*_R t' \iff t \equiv t_0 \rightarrow_R \ldots \rightarrow_R t_n \equiv t'$ for some $\Sigma$-terms $t_0, \ldots, t_n$

$t \leftrightarrow_R t' \iff t \rightarrow_R t' \or t' \rightarrow_R t$

$t \simeq_R t' \iff t \equiv t_0 \leftrightarrow_R \ldots \leftrightarrow_R t_n \equiv t'$ for some $\Sigma$-terms $t_0, \ldots, t_n$

Any finite or infinite sequence $t_0 \rightarrow_R t_1 \rightarrow_R \ldots$ is called a reduction sequence.

A term $t$ is in normal form w.r.t. $R$ if it cannot be rewritten, i.e. $t \not\rightarrow_R t'$ for any $t'$.

We say that $t'$ is a normal form of $t$, or $t$ normalizes to $t'$ if $t \rightarrow^*_R t'$ and $t'$ is in normal form.

7.2.2 Definition

Let $E$ be a set of equations over the signature $\Sigma$. If the set $R := \{ l \mapsto r \mid l \equiv r \in E \}$ is a term rewriting system (i.e. conditions (i) and (ii) in definition 7.2.1 are met), we call $R$ the term rewriting system defined by $E$. In this case we will write $t \rightarrow_E t'$ instead of $t \rightarrow_R t'$ and simply say that $E$ is a term rewriting system. The equations $r = l$ in $E$ will then be called rewrite rules and will be written $r \mapsto l$. 
7.2.3 Example

Let again \( E := \{ x + 0 = x, x + \text{succ}(y) = \text{succ}(x + y) \} \). Clearly \( E \) is a term rewriting system. Let us rewrite the term \( t := 0 + \text{succ}(0) \). We have to find a subterm of \( t \) that matches the left hand side of a rule in \( E \). \( 0 + \text{succ}(0) \) is such a subterm, since \( 0 + \text{succ}(0) \equiv (x + \text{succ}(y)) \{ 0/x, 0/y \} \). We mark the occurrence of this subterm in \( t \) by underlining it: \( 0 + (0 + \text{succ}(0)) \). Now we replace this instance of \( x + \text{succ}(y) \) by the corresponding instance of \( \text{succ}(x + y) \), i.e. by \( \text{succ}(x + y) \{ 0/x, 0/y \} \equiv \text{succ}(0 + 0) \), and obtain \( 0 + \text{succ}(0 + 0) \). Hence

\[
0 + (0 + \text{succ}(0)) \rightarrow_E 0 + \text{succ}(0 + 0)
\]

The term \( 0 + \text{succ}(0 + 0) \) can again be rewritten by replacing the subterm \( 0 + 0 \), which matches the left hand side of the rule \( x + 0 \rightarrow 0 \):

\[
0 + \text{succ}(0 + 0) \rightarrow_E 0 + \text{succ}(0)
\]

Furthermore \( 0 + \text{succ}(0) \) rewrites to \( \text{succ}(0 + 0) \) and the latter to \( \text{succ}(0) \)

\[
\begin{align*}
0 + \text{succ}(0) & \rightarrow_E \text{succ}(0 + 0) \\
& \rightarrow_E \text{succ}(0)
\end{align*}
\]

Hence we have \( 0 + (0 + \text{succ}(0)) \rightarrow_E \text{succ}(0) \).

Exercise: write a reduction sequence showing that \( \text{succ}(0 + 0) + 0 \rightarrow E \text{succ}(0) \).

Obviously, the terms in normal form w.r.t. \( E \) are precisely the terms \( \text{succ}^n(0) \) \((n \in \mathbb{N})\). In particular \( \text{succ}(0) \) is in normal form.

We saw that \( 0 + (0 + \text{succ}(0)) \) and \( \text{succ}(0 + 0) + 0 \) both normalize to \( \text{succ}(0) \). Hence we have

\[
0 + (0 + \text{succ}(0)) \simeq_E \text{succ}(0 + 0) + 0.
\]

Exercise: normalize \( 0 + (0 + \text{succ}(0)) \) to \( \text{succ}(0) \) using a different reduction sequence.

7.2.4 Lemma

The relation \( \simeq_E \) is a congruence on the \( \Sigma \)-algebra \( T(\Sigma, X) \) which in addition is closed under substitutions, i.e. if \( t \simeq_E t' \) then \( t\theta \simeq_E t'\theta \) for every substitution \( \theta \).

**Proof.** Obviously \( \simeq_E \) is an equivalence relation. In order to show that \( \simeq_E \) is compatible with the operation in \( \Sigma \), (i.e \( t_1 \simeq_E t'_1, \ldots, t_n \simeq_E t'_n \Rightarrow f(t_1, \ldots, t_n) \simeq_E f(t'_1, \ldots, t'_n) \)) it suffices to observe that obviously

\[
t_i \rightarrow_E t'_i \quad \Rightarrow \quad f(t_1, \ldots, t_i, \ldots, t_n) \rightarrow_E f(t_1, \ldots, t'_i, \ldots, t_n)
\]

Similarly in order to show that \( \simeq_E \) is closed under substitution it suffices to observe that \( \rightarrow_E \) is closed under substitution, i.e.

\[
t \rightarrow_E t' \quad \Rightarrow \quad t\theta \rightarrow_E t'\theta
\]
7.2.5 Theorem

\[ \vdash_E t = t' \iff t \simeq_E t' \]

Proof. ‘\( \Rightarrow \)’ is proved by induction on the derivation of \( \vdash_E t = t' \). Having lemma 7.2.4 at hand the proof is trivial.

‘\( \Leftarrow \)’. Because of the derivation rules Reflexivity, Symmetry, and Transitivity, it obviously suffices to show

\[ t \rightarrow_E t' \Rightarrow \vdash_E t = t' \]

But this is easy using the rules Compatibility, and Instance.

7.2.6 Corollary

\[ \forall E \models \forall (t = t') \iff t \simeq_E t' \]

Proof. Theorems 7.1.4, 7.1.5, and 7.2.5.

Remark. The relation \( \forall E \models \forall (t = t') \) is undecidable, i.e., there is no algorithm deciding for an arbitrary system \( E \) of equations and terms \( t_1, t_2 \) whether or not \( \forall E \models \forall (t = t') \) holds. However, due to the Soundness and Completeness Theorem of equational logic (7.1.4, 7.1.5) there exists an algorithm that terminates if and only if \( \forall E \models \forall (t = t') \) holds: just generate systematically all equational derivations with axioms in \( E \) and wait until the equation \( t = t' \) appears as end formula. In mathematical terminology: for every finite signature \( \Sigma \) the set

\[ \{(E, t_1, t_2) \mid E \text{ a finite system of equations over } E, \ t_1, t_2 \Sigma\text{-terms, } \forall E \models \forall (t = t')\} \]

is recursively enumerable. The equivalence \( \vdash_E t = t' \iff t \simeq_E t' \) proved in Theorem 7.2.5 provides an optimisation of this algorithm, by replacing the ‘blind’ search for \( \vdash_E t = t' \) by a ‘goal directed’ search for \( t \simeq_E t' \).

7.3 Termination

7.3.1 Definition

A term rewriting system \( R \) is terminating if there is no infinite reduction sequence \( t_0 \rightarrow_R t_1 \rightarrow_R \ldots \)

Remark. Terminating term rewriting systems are often also called Noetherian.

In a terminating term rewriting system every term has a normal form, but the converse is not true as the following example shows.
7.3.2 Example

Let \( R := \{ x + 0 \mapsto x, \ succ(y) \mapsto succ(x + y), \ 0 + y \mapsto y + 0 \} \). \( R \) is not terminating, since for example \( 0 + 0 \rightarrow_R 0 + 0 \rightarrow_R \ldots \). Nevertheless every term has a normal form. By removing the last rule the term rewriting system becomes terminating.

The following lemma provides a general strategy for proving termination of a term rewriting system.

7.3.3 Lemma

Let \( R \) be a term rewriting system over a signature \( \Sigma \).

Let \( \mu \) be a function mapping \( \Sigma \)-terms to natural numbers such that

\[ t \rightarrow_R t' \quad \Rightarrow \quad \mu(t) > \mu(t') \]

Then \( R \) is terminating.

Proof. If we had an infinite reduction sequence \( t_0 \rightarrow_R t_1 \rightarrow_R \ldots \), we would get an infinite decreasing sequence of natural numbers \( \mu(t_0) > \mu(t_1) > \ldots \), which is impossible.

7.3.4 Example

\( R := \{ x - 0 \mapsto x, \ succ(x) - succ(y) \mapsto x - y \} \). Set \( \mu(t) := \) the length of \( t \). Then clearly \( \mu(t) > \mu(t') \) whenever \( t \rightarrow_R t' \). Hence \( R \) is terminating according to lemma 7.3.3.

7.3.5 Example

Consider \( R := \{ f(g(x), y) \mapsto f(y, y) \} \). The right hand side of the only rule in \( R \) is shorter than the left hand side. So, we might expect \( R \) to be terminating. But in fact it is not, since

\[ f(g(x), g(x)) \rightarrow_R f(g(x), g(x)) \rightarrow_R \ldots \]

The following lemma clarifies the situation.
7.3.6 Lemma

Let \( R \) be a term rewriting system over a signature \( \Sigma \) such for every rule \( l \rightarrow r \) in \( R \)

\[ r \text{ is shorter than } l, \]

every variable \( x \in \text{FV}(r) \) occurs in \( r \) at most as often as it occurs in \( l \).

Then \( R \) is terminating.

**Proof.** Obviously the assumptions imply that the length of terms provides a termination measure, i.e., if \( t \rightarrow_R t' \) then \( t' \) is shorter than \( t \).

Unfortunately, the applicability of Lemma 7.3.6 is rather restricted. For example for the term rewriting system \( R := \{ \ x+0 \rightarrow x, \ x+\text{succ}(y) \rightarrow \text{succ}(x+y) \ \} \) any application of the second rule will not decrease the length of a term. Nevertheless \( R \) is terminating (as we will show later).

The following theorem provides a somewhat more sophisticated method for proving termination.

7.3.7 Theorem

Let \( R \) be a term rewriting system over a signature \( \Sigma \).

Let \( A \) be a \( \Sigma \)-algebra with \( A_s = \mathbb{N} \) for every sort \( s \) such that \( f^A \) is a strictly monotone function for every operation \( f \) in \( \Sigma \), i.e.

\[ n_i > n'_i \Rightarrow f^A(n_1, \ldots, n_i, \ldots, n_k) > f^A(n_1, \ldots, n'_i, \ldots, n_k), \]

and such that \( t^A,\alpha > r^A,\alpha \) for every rewrite rule \( l \rightarrow r \in R \) and every variable assignment \( \alpha \).

Then \( R \) is terminating.

**Proof.** We set \( \mu(t) := t^A,\alpha \), where \( \alpha \) is an arbitrary variable assignment (e.g. \( \alpha(x) := 0 \) for all variables \( x \)). By Lemma 7.3.3 it suffices to show that \( \mu(t) > \mu(t') \) whenever \( t \rightarrow_R t' \).

To this end we first show that for any \( \Sigma \)-term \( t \) and \( x \in \text{FV}(t) \)

\[ n > m \Rightarrow t^A,\alpha^x > t^A,\alpha^x \quad \text{ (+)} \]

We prove \((+)\) by induction on \( t \).

**Base:** \( t \equiv x. \ x^A,\alpha^x = n > m = x^A,\alpha^x. \)

**Step:** \( t = f(t_1, \ldots, t_k) \). Let \( n_i := t_i^A,\alpha^x \), and \( m_i := t_i^A,\alpha^x \). By induction hypothesis \( n_i > m_i \) if \( x \in \text{FV}(t_i) \), and this the case at least for one \( i \in \{1, \ldots, k\} \), and, of course \( n_i = m_i \) if \( x \notin \text{FV}(t_i) \). Hence, because \( f^A \) is strictly monotone,
\[ f(t_1, \ldots, t_k)^{A, \alpha_n} = f^A(n_1, \ldots, n_k) > f^A(m_1, \ldots, m_k) = f(t_1, \ldots, t_k)^{A, \alpha_n} \]

Having proved (+), can now easily complete the proof. Assume \( t \rightarrow_R t' \). Then \( t \equiv u\{l\theta/x\} \) and \( t' \equiv u\{r\theta/x\} \), where \( l \mapsto r \in R \) and \( x \) occurs exactly once in \( u \). With \( n := (l\theta)^{A, \alpha} \) we clearly have \( \{l\theta/x\}^{A, \alpha} = \alpha_n^\alpha \) and therefore, using the Substitution Lemma 2.4.4

\[ \mu(t) = t^{A, \alpha} = (u\{l\theta/x\})^{A, \alpha} = u^{A, \{l\theta/x\}^{A, \alpha}} = u^{A, \alpha_n^\alpha} \]

Similarly \( \mu(t') = u^{A, \alpha_n^\alpha} \), with \( m := (r\theta)^{A, \alpha} \). Hence, by virtue of (+) it suffices to show that \( n > m \). But this follows easily by applying the Substitution Lemma 2.4.4 once more and using the assumption on \( R \):

\[ n = l^{A, \theta^{A, \alpha}} > r^{A, \theta^{A, \alpha}} = m \]

### 7.3.8 Example

Let us use Theorem 7.3.7 to prove termination of the term rewriting system \( R := \{ x + 0 \mapsto x, \ x + \text{succ}(y) \mapsto \text{succ}(x + y) \} \).

We define the algebra \( A \) by setting

\[ 0^A := 1 \]

\[ \text{succ}^A(n) := n + 1 \]

\[ n +^A m := n + 2 \times m \]

Then with \( n := \alpha(x) \) and \( m := \alpha(y) \) we have

\[ (x + 0)^{A, \alpha} = n +^A 0^A = n + 2 \times 1 > n = x^{A, \alpha} \]

\[ (x + \text{succ}(y))^{A, \alpha} = n +^A \text{succ}^A(m) = n + 2 \times (m + 1) > n + 2 \times m + 1 = \text{succ}(x + y)^{A, \alpha} \]

Now we show that the strategy of proving termination by assigning a measure \( \mu(t) \) to each term \( t \) such that \( \mu(t) \) decreases when \( t \) is rewritten (lemma 7.3.3) is complete in the sense that for every terminating term rewriting system such a measure exists.
7.3.9 Definition

Let $R$ be a term rewriting system. For every term $t$ the **reduction tree of** $t$ is the tree of all reduction sequences starting with $t$, that is,

the root of this tree is labeled by $t$,

the children of a node labeled by $u$ are labeled by the terms $u'$ such that $u \rightarrow_R u'$.

Clearly the immediate subtrees of the reduction tree of $t$ are precisely the reduction trees of the terms $t'$ with $t \rightarrow_R t'$.

7.3.10 König’s Lemma

Let $R$ be a terminating term rewriting system. Then every term has a finite reduction tree.

**Proof.** Assume for contradiction that there is a term $t_0$ with an infinite reduction tree. Since $R$ is finite $t_0$ can be rewritten in finite many ways only, i.e. there are only finitely many terms $t'$ such that $t_0 \rightarrow_R t'$. Hence there must be a term $t_1$ such that $t_0 \rightarrow_R t_1$ and the reduction tree of $t_1$ is infinite. Proceeding with $t_1$ in the same way as we did with $t_0$ we obtain a term $t_2$ such that $t_1 \rightarrow_R t_2$ and the reduction tree of $t_2$ is infinite. Continuing in this way we obtain an infinite reduction sequence $t_0 \rightarrow_R t_1 \rightarrow_R t_2 \rightarrow_R \ldots$.

7.3.11 Definition

Let $R$ be a terminating term rewriting system. For every term $t$ we set

$$\bar{z}(t) := \text{height of the reduction tree of } t$$

$$= \max\{n \in \mathbb{N} \mid t \text{ starts a reduction sequence of length } n\}$$

$\bar{z}(t)$ is called the **reduction height** of $t$. Obviously

$$\bar{z}(t) = \begin{cases} 0 & \text{if } t \text{ is in normal form} \\ 1 + \max\{\bar{z}(t') \mid t \rightarrow_R t'\} & \text{otherwise} \end{cases}$$

In particular $t \rightarrow_R t'$ implies $\bar{z}(t') < \bar{z}(t)$.

The tacitly assumed condition on the term rewriting system $R$ to be finite is necessary, as the following example shows.
7.3.12 Example

Consider the \textit{infinite} (!) term rewriting system

\[ R := \{ x + 0 \mapsto x, \ x + \text{succ}(y) \mapsto \text{succ}(x + y) \} \cup \{ c \mapsto 0 + \text{succ}^n(0) \mid n \in \mathbb{N} \} \]

It is easy to see that \( R \) is terminating, but the term \( c \) starts arbitrarily long reduction sequences, namely

\[ c \rightarrow_R 0 + \text{succ}^n(0) \rightarrow_R \text{succ}(0) + \text{succ}^{n-1}(0) \rightarrow_R \ldots \rightarrow_R \text{succ}^n(0) + 0 \rightarrow_{\text{succ}}^n(0) \]

for every \( n \in \mathbb{N} \).

**Remark.** It is undecidable whether or not a term rewriting system \( R \) is terminating. This can be seen, for example, by representing each Turing machine \( T \) by a term rewriting systems \( R_T \) in such a way that the \( T \) halts if and only \( R_T \) terminates, thus reducing the halting problem for Turing machines, which is well-known to be undecidable, by the termination problem for term rewriting systems. However, due to König's Lemma 7.3.10, this problem is recursively enumerable.

Although being undecidable in general the termination problem can be solved in many interesting cases. In fact it is one of the most important and largest research area in term rewriting theory. There are far reaching and powerful mathematical methods available for proving termination (see e.g. the book of Baader and Nipkow mentioned in the introduction), Theorem 7.3.7 being just one of simplest.

7.4 Confluence

7.4.1 Definition

1. A term rewriting system \( R \) over a signature \( \Sigma \) is \textbf{confluent} if for all \( \Sigma \)-terms \( t, t_1, t_2 \) such that \( t \rightarrow^*_R t_1 \) and \( t \rightarrow^*_R t_2 \) there exists a \( \Sigma \)-term \( t_3 \) such that \( t_1 \rightarrow^*_R t_3 \) and \( t_2 \rightarrow^*_R t_3 \).

\[ t \quad \begin{array}{c} \text{R} \\ \text{R} \end{array} \quad t_1 \quad \begin{array}{c} \text{R} \\ \text{R} \end{array} \quad t_2 \]

2. A term rewriting system \( R \) over a signature \( \Sigma \) is \textbf{locally confluent} if for all \( \Sigma \)-terms \( t, t_1, t_2 \) such that \( t \rightarrow_R t_1 \) and \( t \rightarrow_R t_2 \) there exists a \( \Sigma \)-term \( t_3 \) such that \( t_1 \rightarrow^*_R t_3 \) and \( t_2 \rightarrow^*_R t_3 \).
### 7.4.2 Example

The term rewriting system $R := \{ a \mapsto b, a \mapsto c, b \mapsto a, b \mapsto d \}$ is locally confluent. However, $R$ is not confluent, since $a \rightarrow^*_R c$ and $a \rightarrow^*_R d$, but $c$ and $d$ cannot reduced to a common term.

**Exercises.**

(a) Add one rewrite rule that makes $R$ confluent.

(b) Remove one rewrite rule such that $R$ becomes confluent.

(c) Is $R$ terminating?

### 7.4.3 Theorem

Let $R$ be a confluent term rewriting system over a signature $\Sigma$.

Then $R$ has the following so-called *Church-Rosser property*: For all $\Sigma$-terms $t_1, t_2$

$$t_1 \simeq_R t_2 \iff \text{there exists a } \Sigma\text{-term } t_3 \text{ such that } t_1 \rightarrow^*_R t_3 \text{ and } t_2 \rightarrow^*_R t_3$$

\[
\begin{array}{c}
t_1 \\
\uparrow \quad \uparrow \quad \uparrow \\
R \quad R \\
\downarrow \quad \downarrow \\
* \quad * \\
R \\
\end{array}
\]

\[
\begin{array}{c}
t_1 \quad \simeq_R \\
\downarrow \quad \downarrow \\
R \\
\end{array}
\]

**Proof.**

‘$\Rightarrow$’ is obvious.

‘$\Rightarrow$’ is easily proved by induction on $n$, where $t_1 \equiv u_0 \leftrightarrow_R \ldots \leftrightarrow_R u_n \equiv t_2$. 
7.4.4 Newman's Lemma

Every terminating and locally confluent term rewriting system is confluent.

Proof. Let $R$ be terminating and locally confluent. We prove the implication

$$ t \rightarrow^* R t_1 \text{ and } t \rightarrow^* R t_2 \Rightarrow \text{there exists a } \Sigma\text{-term } t_3 \text{ such that } t_1 \rightarrow^* R t_3 \text{ and } t_2 \rightarrow^* R t_3 $$

by induction on $\sharp(t)$ (cf. definition 7.3.11; $\sharp(t)$ exists because $R$ is assumed to be terminating). So, assume $t \rightarrow^* R t_1$ and $t \rightarrow^* R t_2$. If $t \equiv t_1$ then we may simply chose $t_3 \equiv t_2$, and if $t \equiv t_2$ we chose $t_3 \equiv t_1$. Otherwise there are terms $t'_1$ and $t'_2$ such that for $i = 1, 2$ we have $t \rightarrow^* R t'_i$ and $t'_i \rightarrow^* R t_i$. (It is recommended to draw a picture when reading through rest of the argument.) Since by assumption $R$ is locally confluent there is some term $t'_3$ such that $t'_1 \rightarrow^* R t'_3$ and $t'_2 \rightarrow^* R t'_3$. Furthermore, since $\sharp(t'_i) < \sharp(t)$, we know, by induction hypothesis, that there are terms $u_1, u_2$ with $t_i \rightarrow^* R u_i$ and $t'_3 \rightarrow^* R u_i$, for $i = 1, 2$. Using the induction hypothesis once more, now with $t'_3$ (we have $\sharp(t'_3) \leq \sharp(t'_1) < \sharp(t)$) we conclude that there is a term $t_3$ such that $u_1 \rightarrow^* R t_3$ and $u_2 \rightarrow^* R t_3$.

7.4.5 Lemma

Let $R$ be a confluent and terminating term rewriting system over a signature $\Sigma$. Then every $\Sigma$-term $t$ has a unique normal form.

Proof. Since $R$ is terminating $t$ has a normal form $t'$, i.e. $t \rightarrow^* R t'$ and $t'$ is in normal form. If also $t \rightarrow^* R t''$ with $t'$ in normal form, then, by confluence, $t'$ and $t''$ reduce to the same term, but, since $t'$ and $t''$ are normal, this can only be the case if $t' \equiv t''$.

7.4.6 Definition

Let $R$ be a confluent and terminating term rewriting system over a signature $\Sigma$. Then every for every $\Sigma$-term $t$ we denote by

$$ \text{nf}(t) $$

the unique normal form of $t$.

7.4.7 Theorem

Let $E$ be a system of equations over a signature $\Sigma$ defining confluent and terminating term rewriting system. Then for all $\Sigma$-terms $t_1, t_2$

$$ \forall E \models t_1 = t_2 \Leftrightarrow \text{nf}(t_1) = \text{nf}(t_2) $$
In particular, the relation $\forall E \models t_1 = t_2$ is decidable.

**Proof.** Clearly $t_1 \equiv_R t_2$ if and only if $\text{nf}(t_1) = \text{nf}(t_2)$. The result follows with Theorem 7.2.5.

Theorem 7.4.7 gives us a simple (and often also efficient) method for deciding whether an equation $t_1 = t_2$ follows logically from a set of equations. It is therefore highly desirable to transform a given system of equations into an equivalent one that defines a confluent and terminating term rewriting system. The famous Knuth-Bendix completion algorithm which we discuss next provides a method for doing this in many cases.

### 7.4.8 Definition

A **unifier** of two terms $t_1$, $t_2$ is a substitution $\theta : \text{FV}(t_1) \cup \text{FV}(t_2) \rightarrow T(\Sigma, V)$ ($V$ the set of all variables) such that

$$t_1 \theta = t_2 \theta$$

A **most general unifier** of $t_1$, $t_2$ is a unifier $\theta$ of $t_1$, $t_2$ with the additional property that for any other unifier $\theta'$ of $t_1$, $t_2$ there exists a substitution $\sigma$ with $\theta' = \sigma \circ \theta$.

**Remark.** It can be (efficiently) decided whether two terms $t_1$, $t_2$ are unifiable (Robinson), and if the terms are unifiable a most general unifier can be efficiently computed.

### 7.4.9 Definition

A rule $l' \mapsto r'$ is a **variant** of a rule $l \mapsto r$ if $l' \mapsto r'$ is obtained from $l \mapsto r$ by a consistent variable renaming.

For example $y + \text{succ}(z) \mapsto \text{succ}(y + z)$ is a variant of $x + \text{succ}(y) \mapsto \text{succ}(x + y)$, but $z + \text{succ}(z) \mapsto \text{succ}(z + z)$ is not.

### 7.4.10 Definition

Let $R$ be a term rewriting system over a signature $\Sigma$ and $l_i \mapsto r_i$, $i = 1, 2$, be variants of rules in $R$ such that $\text{FV}(l_1) \cap \text{FV}(l_2) = \emptyset$.

Let $t$ be a subterm of $l_1$ which is not a variable and which is unifiable with $l_2$. I.e. $l_1$ is of the form $l_1 = u\{t/x\}$, where $x$ is a fresh variable occurring exactly once in $u$, and there is a most general unifier $\theta$ of $t$ and $l_2$, i.p. $t\theta = l_2\theta$.

Note that $l_1\theta \equiv (u\theta)\{l_2\theta/x\}$ and therefore

Then $(r_1\theta, (u\theta)\{r_2\theta/x\})$ is called a **critical pair** of $R$.

We let $\text{CP}(R)$ denote the set of all critical pairs of $R$.

Note that $\text{CP}(R)$ is a finite set that can be easily computed from $R$. 

7.4.11 Lemma

A term rewriting system $R$ is locally confluent iff for all critical pairs $(t_1, t_2)$ of $R$ there exists a term $t$ such that $t_1 \rightarrow_R^* t$ and $t_2 \rightarrow_R^* t$.

**Proof.** See e.g. Baader/Nipkow.

7.4.12 Theorem

A terminating term rewriting system $R$ is confluent iff for all critical pairs $(t_1, t_2)$ of $R$ there exists a term $t$ such that $t_1 \rightarrow_R^* t$ and $t_2 \rightarrow_R^* t$.

In particular it is decidable whether a terminating term rewriting system is confluent.

**Proof.** Lemma 7.4.11 and Newman’s Lemma 7.4.4.

**Remark.** For arbitrary term rewriting systems confluence is undecidable (see e.g. Baader/Nipkow).

**Remark.** Theorem ?? suggests an obvious method of how to try to transform a terminating term rewriting system $R$ into an equivalent one that is confluent:

1. Compute $\text{CP}(R)$. If for all $(t_1, t_2) \in \text{CP}(R)$ there is a $t$ with $t_1 \rightarrow_R^* t$ and $t_2 \rightarrow_R^* t$ then stop (in this case $R$ is confluent according to Theorem 7.4.12).

2. For any $(t_1, t_2) \in \text{CP}(R)$ such that there is no $t$ with $t_1 \rightarrow_R^* t$ and $t_2 \rightarrow_R^* t$ either add the rule $t_1 \mapsto t_2$ or the rule $t_2 \mapsto t_1$ to $R$ such that the extended term rewriting system remains terminating (that’s the tricky part and not always possible, i.e. the method may fail here). Set $R$ to be the extended system and go to 1.

A refinement of this algorithm is the above mentioned Knuth-Bendix completion algorithm.

7.5 Rapid prototyping

7.5.1 Definition

The term rewriting system associated with an initial specification $\text{Init-Spec}(\Sigma, E)$ is the term rewriting system defined by $E$, provided, of course, $E$ defines a term rewriting system (cf. Definition 7.2.2).
Rapid prototyping for an initial specification Init–Spec(Σ, E) consists in computing the normal form of a closed Σ-term. This, of course presupposes that the term rewriting system associated with Init–Spec(Σ, E) is confluent and terminating.

Now we show that rapid prototyping for an initial specification Init–Spec(Σ, E) may be viewed as the calculation of the value of a closed term in a particularly perspicuous model of Init–Spec(Σ, E).

7.5.2 Definition

Let Init–Spec(Σ, E) be an initial specification such that the term rewriting system associated with it is terminating and confluent. The algebra of closed normal terms is defined as follows.

<table>
<thead>
<tr>
<th>Algebra</th>
<th>NF_E(Σ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carriers</td>
<td>NF_E(Σ)_n := { t ∈ T(Σ)</td>
</tr>
<tr>
<td>Constants</td>
<td>c^{NF_E(Σ)} := nf(c)</td>
</tr>
<tr>
<td>Operations</td>
<td>f^{NF_E(Σ)}(t_1, \ldots, t_n) := nf(f(t_1, \ldots, t_n))</td>
</tr>
</tbody>
</table>

7.5.3 Theorem

Let Init–Spec(Σ, E) be an initial specification such that the term rewriting system associated with it is terminating and confluent. Then the algebra NF_E(Σ) of closed normal terms is a model of Init–Spec(Σ, E).

Furthermore for every closed Σ-term t

\[ t^{NF_E(Σ)} = nf(t) \]

Proof. In order to show that NF_E(Σ) is a model of Init–Spec(Σ, E) it suffices to prove that NF_E(Σ) is isomorphic to T_E(Σ). Obviously the identical embedding of NF_E(Σ) into T_E(Σ) is an isomorphism.

Furthermore [nf]: T_E(Σ) → NF_E(Σ) defined by [nf][t] := nf(t) clearly is a well-defined homomorphism. Because evaluation of terms defines another homomorphism from T_E(Σ) to NF_E(Σ) (cf. 6.2.3) we have, by initiality of T_E(Σ), that t^{NF_E(Σ)} = nf(t).

The following theorem provides a useful criterion for testing whether an initial specification is adequate for a given algebra A.
7.5.4 Theorem

Let $A$ be a $\Sigma$-algebra and $\text{Init-Spec}(\Sigma, E)$ an initial specification defining a terminating and confluent term rewriting system.

Then $\text{Init-Spec}(\Sigma, E)$ is adequate for $A$ (i.e. $A$ is a model of $\text{Init-Spec}(\Sigma, E)$) iff

(i) Every element $a \in A_s$ is the value of a unique closed normal $\Sigma$-term.

(ii) $f^A(t_1^A, \ldots, t_n^A) = (\text{nf}(f(t_1, \ldots, t_n)))^A$ for every operation $f : s_1 \times \ldots \times s_n \to s$ and all closed normal terms $t_i$ of sort $s_i$, $i = 1, \ldots, n$.

Proof. By (ii) evaluation of closed normal $\Sigma$-terms in $A$ is a homomorphism from $\text{NF}_E(\Sigma)$ to $A$, and by (i) this homomorphism is bijective, i.e. an isomorphism. Since $A$ is isomorphic to $\text{NF}_E(\Sigma)$, and by Theorem 7.5.3, $\text{NF}_E(\Sigma)$ is a model of $\text{Init-Spec}(\Sigma, E)$ it follows that $A$ is a model of $\text{Init-Spec}(\Sigma, E)$, too.

On the other hand if $A$ is a model of $\text{Init-Spec}(\Sigma, E)$, then $A$ must be isomorphic to $\text{NF}_E(\Sigma)$, and hence (i) and (ii) hold.

7.5.5 Example

Let $A$ be the algebra of the quicksort algorithm with all its auxiliary sorts and functions. Hence $A$ has as carrier sets the set of natural numbers the set of lists of natural numbers and the set of Boolean values. It has the usual constants and operations on natural numbers, lists of natural numbers and the Booleans, it has a less-than predicate $<$ on natural numbers, operations low and high such that $\text{low}(n, l)$ selects from the list $l$ the list of those elements that are $\leq n$, and $\text{high}(n, l)$ selects from $l$ the list of those elements that are $> n$. Finally, $A$ has the main operation sort that sorts list using the auxiliary operations low and high. Since low and high use case analysis in their definitions, $A$ also needs an if-then-else operation.

Our goal is to design an initial specification QUICKSORT that is adequate for $A$, i.e. $A$ shall be a model of QUICKSORT. We also aim for rapid prototyping. Hence we have to ensure that the term rewriting system associated with QUICKSORT is confluent and terminating.
<table>
<thead>
<tr>
<th><strong>Init Spec</strong></th>
<th>QUICKSORT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sorts</strong></td>
<td>nat, boole, list</td>
</tr>
<tr>
<td><strong>Constants</strong></td>
<td>0 : nat</td>
</tr>
<tr>
<td></td>
<td>T : boole</td>
</tr>
<tr>
<td></td>
<td>F : boole</td>
</tr>
<tr>
<td></td>
<td>nil : list</td>
</tr>
<tr>
<td><strong>Operations</strong></td>
<td>succ : nat → nat</td>
</tr>
<tr>
<td></td>
<td>cons: nat × list → list</td>
</tr>
<tr>
<td></td>
<td>if : nat × list × list → list</td>
</tr>
<tr>
<td></td>
<td>≤ : nat × nat → boole</td>
</tr>
<tr>
<td></td>
<td>@ : list × list → list</td>
</tr>
<tr>
<td></td>
<td>low : nat × list → list</td>
</tr>
<tr>
<td></td>
<td>high: nat × list → list</td>
</tr>
<tr>
<td></td>
<td>sort : list → list</td>
</tr>
<tr>
<td><strong>Variables</strong></td>
<td>x, y : nat, l, l₁, l₂ : list</td>
</tr>
<tr>
<td><strong>Equations</strong></td>
<td>if(T, l₁, l₂) = l₁</td>
</tr>
<tr>
<td></td>
<td>if(F, l₁, l₂) = l₂</td>
</tr>
<tr>
<td></td>
<td>x &lt; x = F</td>
</tr>
<tr>
<td></td>
<td>0 &lt; succ(x) = T</td>
</tr>
<tr>
<td></td>
<td>succ(x) &lt; 0 = F</td>
</tr>
<tr>
<td></td>
<td>succ(x) &lt; succ(y) = x &lt; y</td>
</tr>
<tr>
<td></td>
<td>nil @ l = l</td>
</tr>
<tr>
<td></td>
<td>cons(x, l₁) @ l₂ = cons(x, l₁ @ l₂)</td>
</tr>
<tr>
<td></td>
<td>low(x, nil) = nil</td>
</tr>
<tr>
<td></td>
<td>low(x, cons(y, l)) = if(y &lt; x, low(x, l), cons(y, low(x, l)))</td>
</tr>
<tr>
<td></td>
<td>high(x, nil) = nil</td>
</tr>
<tr>
<td></td>
<td>high(x, cons(y, l)) = if(x &lt; y, cons(y, high(x, l)), high(x, l))</td>
</tr>
<tr>
<td></td>
<td>sort(nil) = nil</td>
</tr>
<tr>
<td></td>
<td>sort(cons(x, l)) = sort(low(x, l)) @ cons(x, sort(high(x, l)))</td>
</tr>
</tbody>
</table>
The term rewriting system $R$ associated with QUICKSORT is terminating, although we are not in the position to prove this easily with the methods developed so far. In order to check confluence we compute the critical pairs. The only one critical pair is $(F, x < x)$, which is generated by the first and last rule for $<$. Since $x < x \rightarrow_R F$ we conclude with Theorem 7.4.12 that $R$ is confluent.

It follows with Theorem 7.5.3 that QUICKSORT is adequate for $A$.

If we are unable to prove that a given initial specification is confluent and terminating, the following theorem might be useful.

### 7.5.6 Theorem

Let $E$ be a system of equations over a signature $\Sigma$ defining a term rewriting system such that every $\Sigma$-term has a normal form. Let $A$ be a generated model of $\forall E$ such that $t_1^A \neq t_2^A$ for every two different terms $t_1, t_2$ in normal form. Then Init–Spec$(\Sigma, E)$ is adequate for $A$.

**Proof.** By Theorem 6.2.5 (iv) it suffices to show that for any two closed $\Sigma$-terms $t_1, t_2$ with $t_1^A = t_2^A$ we have $\vdash_E t_1 = t_2$. Let $u_1, u_2$ be normal forms of $t_1, t_2$ respectively. Then $t_i^A = u_i^A$ and hence $u_1^A = u_2^A$. By the assumption of the theorem we get $\vdash_E u_1 = u_2$ and hence $\vdash_E t_1 = t_2$, because $\vdash_E t_i = u_i$.

### 7.6 Summary and Exercises

The central notions and results of this section were the following.

- The deduction rules of equational logic (7.1.1);
- the notion of a term rewriting system $R$ and associated with it the relations
  
  \[ t \rightarrow_R t' \]

  \[ t \rightarrow^*_R t' \]

  \[ t \leftrightarrow_R t' \]

  \[ t \simeq_R t' \]

  and the notion of a term in normal form (7.2.1);
- the term rewriting system associated with a system of equations (7.2.2)
- the Soundness Theorem (7.1.4) and Birkhoff’s Completeness Theorem (7.1.5) for equational logic; together with 7.2.5 they yield the equivalences
  
  \[ \forall E \models \forall (t = t') \iff \vdash_E t = t' \iff t \simeq_E t' \]

- the property of termination (7.3.1) and some simple techniques for proving termination (7.3.3, 7.3.6, 7.3.7);
the property of confluence (7.4.1);
the normal form of a term \( \text{nf}(t) \) w.r.t a confluent and terminating term rewriting system (7.4.6);
the term rewriting system associated with an initial specification;
rapid prototyping (7.5.1), which consists in computing the normal forms of closed terms with respect to the term rewriting system associated with the equations of the initial specification \( \text{Init-Spec}(\Sigma, E) \); it can be applied if \( E \) is confluent and terminating, which ensures that every term has a unique normal form; rapid Prototyping can be used to mechanically decide equations \( t_1 = t_2 \) between closed terms by checking whether their normal forms are the same, because of the equivalence

\[
\forall E \models t_1 = t_2 \iff \text{nf}(t_1) \equiv \text{nf}(t_2)
\]

(7.4.7); it can also be used to implement the model of closed normal forms of the initial specification (see below);
the model of closed normal forms of an initial specification whose associated term rewriting system is confluent and terminating (7.5.2, 7.5.3)

Exercises.

1. Consider the following initial specification

<table>
<thead>
<tr>
<th>Init Spec</th>
<th>( DH )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>rat</td>
</tr>
<tr>
<td>Constants</td>
<td>( \text{one} : \text{rat} )</td>
</tr>
<tr>
<td>Operations</td>
<td>( \text{double} : \text{rat} \to \text{rat} )</td>
</tr>
<tr>
<td></td>
<td>( \text{half} : \text{rat} \to \text{rat} )</td>
</tr>
<tr>
<td>Variables</td>
<td>( x : \text{rat} )</td>
</tr>
<tr>
<td>Equations</td>
<td>( \text{half}(\text{double}(x)) = x )</td>
</tr>
<tr>
<td></td>
<td>( \text{double}(\text{half}(x)) = x )</td>
</tr>
</tbody>
</table>

Let \( \Sigma \) be the signature of \( DH \). Let \( Q^+ \) be the \( \Sigma \)-algebra of positive rational numbers with the obvious interpretation of the constant and the operations.

(a) Show that \( Q^+ \) is not a model of \( DH \).
(b) Describe a subalgebra of $\mathbb{Q}^+$ that is a model of $DH$.

(c) Describe the closed $\Sigma$-terms that are in normal form with respect to the term rewriting system associated with $DH$.

(d) Construct a model $A$ of $DH$ with $A_{\text{rat}} = \mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$ and $\text{one}^A := 0$.

2. Consider the initial specification

<table>
<thead>
<tr>
<th>Init Spec</th>
<th>$I$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>nat, boole</td>
</tr>
<tr>
<td>Constants</td>
<td>$0: \text{nat}, T: \text{boole}, F: \text{boole}$</td>
</tr>
</tbody>
</table>
| Operations| $\text{succ}: \text{nat} \rightarrow \text{nat}$  
            | $\text{iszero}: \text{nat} \rightarrow \text{boole}$ |
| Variables | $x: \text{nat}$ |
| Equations | $\text{succ}(\text{succ}(0)) = 0$  
            | $\text{iszero}(0) = T$  
            | $\text{iszero}(\text{succ}(x)) = F$ |

Let $\Sigma$ be the signature of $I$, and $E$ the set of equations of $I$.

(a) Describe the closed $\Sigma$-terms that are in normal form with respect to the term rewriting system $R$ associated with $I$. Show that $R$ is terminating, but not confluent.

(b) Show that $\vdash_E T = F$.

(c) Show that $\vdash_E \text{succ}(\text{succ}(x)) = x$ does not hold.

(d) Construct a model of $I$. How many elements do its carriers contain?

3. Consider the signature $\Sigma := (\{s\}, \{c: s, f : s \rightarrow s\})$ and the term rewriting system $R := \{f(f(x)) \leftrightarrow c\}$ over $\Sigma$.

Show that $R$ is terminating, but not confluent.