CS_376 Programming with Abstract Data Types

Ulrich Berger

Lecture Notes

Department of Computer Science

Swansea University

Autumn 2009

Contents

1 Introduction **4**

I Foundations **7**

2 Formal Methods in Software Design **8**

2.1 The software design process 8

2.2 An Example of Program Development 10

2.3 Programming by transformation 11

2.4 Programming by extraction from proofs 12

3 Logic **13**

3.1 Signatures and algebras 13

3.2 Terms and their semantics 16

3.3 Formulas and their semantics 18

3.4 Logical consequence, logical validity, satisfiability 21

3.5 Substitutions 23

3.6 Other Logics 27

3.7 Summary and Exercises 28

4 Proofs **30**

4.1 Natural Deduction 30

4.2 Equality rules 35

4.3 Soundness and completeness 37

4.4 Axioms and rules for data types 38

4.5 Summary and Exercises 39

II Abstract Data Types **43**

5 Algebraic Theory of Abstract Data Types **44**

5.1 Homomorphisms and abstract data types 44

5.2 The Homomorphism Theorem 49

5.3 Initial algebras 53

5.4 Summary and Exercises 57

6 Specification of Abstract Data Types **59**

6.1 Loose specifications 59

6.2 Initial specifications 66

6.3 Exception handling 75

6.4 Modularisation 77

6.5 Abstraction through Information hiding 78

6.6 Specification languages 79

6.7 Summary and Exercises 81

7 Implementation of Abstract Data Types **85**

7.1 Implementing ADTs in Functional and Object Oriented Style 85

7.2 Efficiency 89

7.3 Persistence 91

7.4 Structural Bootstrapping 94

7.5 Correctness 95

7.6 Summary and Exercises 96

III Advanced Methods of Program Development **99**

8 Term Rewriting and Rapid Prototyping **100**

8.1 Equational logic 100

8.2 Term rewriting systems 103

8.3 Termination 106

8.4 Confluence 111

8.5 Rapid prototyping 115

8.6 Summary and Exercises 119

9 Programs from proofs **122**

9.1 Formulas as data types 123

9.2 A notation system for proofs 125
9.3 Program synthesis from intuitionistic proofs 127
9.4 Program synthesis from classical proofs 130
9.5 Applications 131
9.6 Summary and Exercises 132

1 Introduction

This course gives an introduction to *Abstract Data Types* and their role in current and future methodologies for the development of reliable software.

Before we begin with explaining what Abstract Data Types are and what methodologies we have in mind let us clarify what*reliable software* is. By reliable software we mean computer programs that are

- *adequate* – they solve the customers' problems,
- *correct* – they are free of bugs and thus behave as expected,
- *easy to maintain* – they can be easily modified or extended without introducing new errors.

Conventional programming techniques to a large extent fail to produce software meeting these requirements. It is estimated that about 80% of the total time and money currently invested into software development is spent on finding errors and amending incorrect or poorly designed software. Hence there is an obvious need for better programming methodologies.

In this course we will study *formal methods* for the development of programs that are guaranteed to be adequate and correct and are easy to maintain. These methods will use the concept of an Abstract Data Type and will be fundamentally based on mathematical and logical disciplines such as *mathematical modelling*, *formal specification* and *formal reasoning*.

Now, what are Abstract Data Types and why are they useful for producing better programs? And what does this have to do with mathematics and logics?

In a nutshell, these questions can be answered as follows:

- An *Abstract Data Type* consists of a data structure (a collection of objects of similar "shape") together with operations whose implementation is however *hidden*.
- Abstract Data Types can be seen as small independent program units. As such they support *modularisation*, that is, the breaking down of complex program systems into small manageable parts, and *abstraction*, that is, the omission of unnecessary details from programs and the guarantee that a change of one unit does not affect other units.
- *Mathematics* is used to build *models* of Abstract Data Types called *algebras*. The study of algebras, relations between algebras and mathematical operations on algebras is essential for a thorough understanding of many important programming concepts.
- *Logic* is used to formally *specify* (describe) Abstract Data Types and to prove properties about them, that is, to *prove the correctness* of program units. Logic can also be used to *synthesise correct programs* automatically from a formal specification or a formal proof of a specification.

These notes are organised as follows:

Part I lays the mathematical and logical foundation necessary for a thorough understanding of Abstract Data Types and their use in programming. In Chapter 2 we motivate the use of formal methods in the design of software and give an overview of the things to come. We compare the conventional software design process with a more structured approach involving modularisation and abstraction and discuss the advantages of the latter method. By means of a simple case study we demonstrate how logic can be used to synthesise correct programs automatically. In *Chapter 3* we then introduce the fundamental concepts of formal logic: *Signatures*, *algebras*, *formulas* and the notion of *logical truth*. In *Chapter 4* we study a formal notion of *proof* and discuss Gödel's famous *Completeness Theorem* and some of its consequences.

Part II is about Abstract Data Types as a well-established tool in modern high-level programming. *Chapter 5* presents Abstract Data Types from an algebraic perspective and studies the structural properties of Abstract Data Types, using basic notions of *category theory*. *Chapter 6* is concerned with the *formal specification* of Abstract Data Types. A particularly important role will be played by *initial specifications* which are particularly simple, but also very concise since they allow to pin down Algebraic Data Types up to isomorphism. Besides studying the theory of specifications we also look at various systems supporting the specification of Abstract Data Types and at some industrially applied specification languages. In *Chapter 7* we discuss how Abstract Data Types can be implemented in functional and object-oriented programming languages and how they can be used for structuring large software systems. We also look at some techniques for the efficient implementation of common data structures like trees and queues.

Part III, finally, presents two advanced logical methods for program synthesis. *Chapter 8* shows how to automatically generate implementations of Abstract Data Types from an algorithmic interpretation of equational specifications (term rewriting, rapid prototyping) and *Chapter 9* discusses the *proofs-as-programs* paradigm as a methodology for synthesising correct programs form formal proofs.

This course is mainly based on the following literature (see the List of References at the end of these notes):

- Van Dalen's textbook [Dal] and the monograph [TS], by Troelstra and Schwichtenberg give introductions into formal logic with a focus on constructive (or intuitionistic) logic. Both books will mainly be used in the Chapters 3, 4 and 9.
- The book [LEW], by Loecks, Ehrich and Wolf, covers the theoretical foundations of ADTs and their specification. It will be an important source for the Chapters 5 and 6.
- Meinke and Tucker's Chapter in the Handbook of Logic in Computer Science, [MeTu], focuses on the model theory of ADTs from an algebraic point of view. It will mainly be used in the Chapters 3 and 5.
- The textbook [BaNi], by Baader and Nipkow, treats term rewriting, that is, the algorithmic aspects of equational specifications. It is the basis for Chapter 8.

- Okasaki's monograph [Oka] studies efficient functional implementations of ADTs. The examples in Chapter 7 are mainly taken from this book.
- Elien's book [Eli], discusses the role of ADTs in Functional and Object Oriented Programming. It is the source of Chapter 7.1.

Further references are given in the text. These notes are self contained as reading material for this course. However, the course can only give a hint at the deep and beautiful ideas underlying contemporary theoretical computer science in general and the theory of abstract data types in particular. The interested reader may find it useful to consult the original sources, in order to get more background information and to study more thoroughly some of the results (for example, Gödel's Completeness Theorem) the proofs of which are beyond the scope of this course.

The photographs are taken from the web page,

`http://www-history.mcs.st-andrews.ac.uk/history/index.html`,

School of Mathematics, University of St Andrews.

Part I

Foundations

2 Formal Methods in Software Design

2.1 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 pup 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, [LEW]). It has at least two deficiencies. First, being based on tests, it can only confirm the

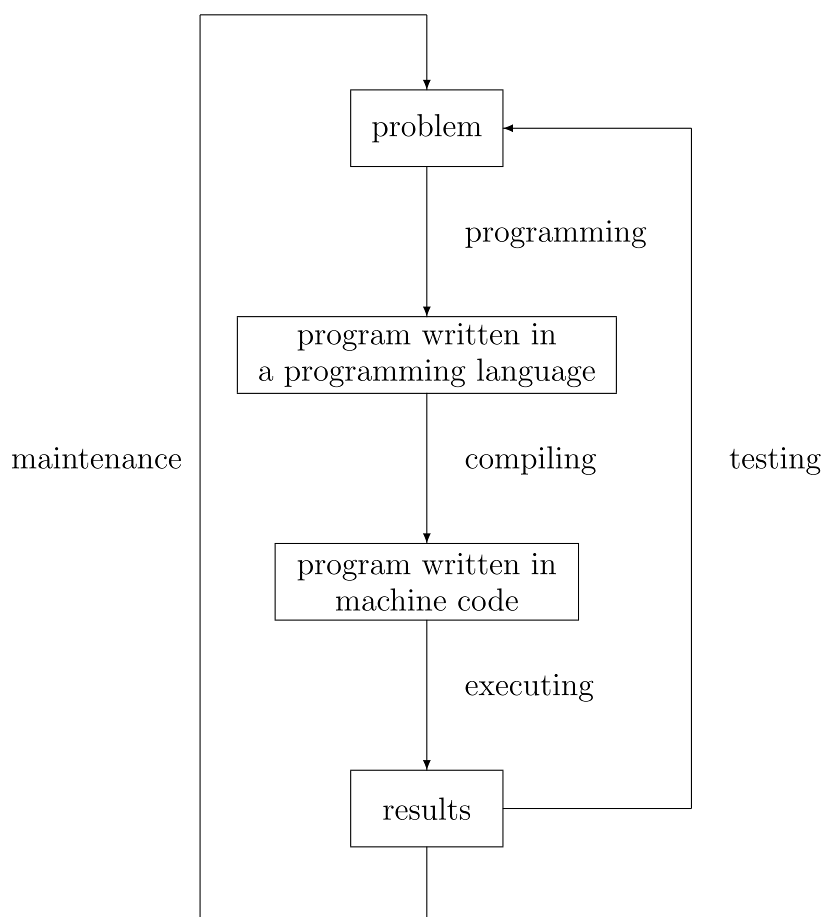


Figure 1: A software life-cycle model illustrating conventional software design

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 [LEW].

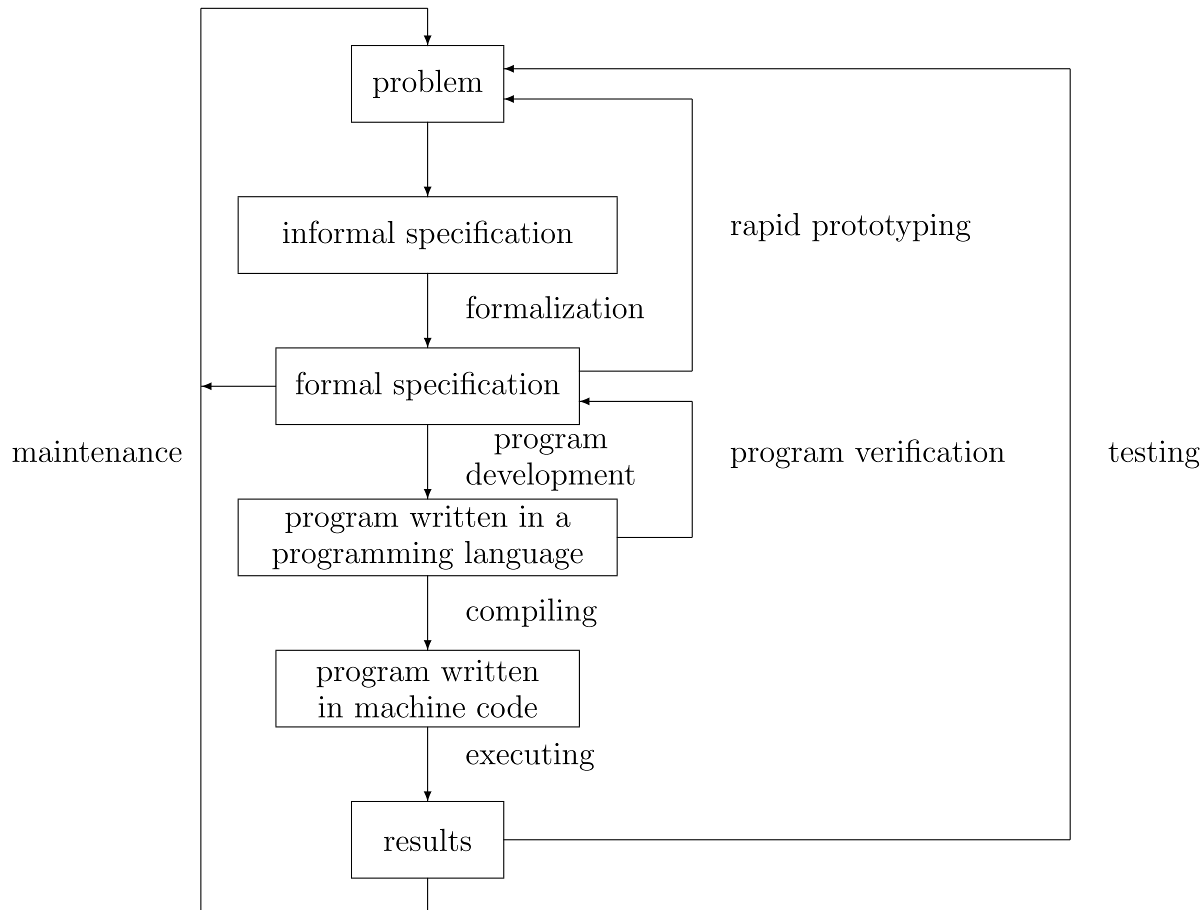


Figure 2: A software life-cycle model illustrating abstraction

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. **Programming with Abstract Data Types:** From the formal specification (that is, the specification of an Abstract Data Type) a certain method of program development leads to a program that is provably correct. This means it can be *proven* that the program meets the specification (program verification). In the course we will discuss different such methods, two of them are illustrated in the example in section 2.2.

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

2.2 An Example of Program Development

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 also dividing m and n, then l divides k.

Formal specification (of an ADT)

$$k = gcd(m,n) \leftrightarrow k \mid m \wedge k \mid n \wedge \forall l(l \mid m \wedge l \mid n \rightarrow l \mid k)$$
$$k \mid m \leftrightarrow \exists q\, k * q = m$$

Transformation

Formal specification'

$\exists r\,[\,r < n \wedge \exists q\, m = q * n + r \wedge$
$\quad r = 0 \rightarrow gcd(m,n) = n \wedge$
$\quad r > 0 \rightarrow gcd(m,n) = gcd(n,r)\,]$

Formal specification"

$mod(m,n) < n \wedge \exists q[m = q * n + mod(m,n)] \wedge$
$mod(m,n) = 0 \rightarrow gcd(m,n) = n \wedge$
$mod(m,n) > 0 \rightarrow gcd(m,n) = gcd(n, mod(m,n))$

Formal specification"'

$m < n \rightarrow mod(m,n) = m \wedge$
$m \geq n \rightarrow mod(m,n) = mod(m-n,n) \wedge$
$mod(m,n) = 0 \rightarrow gcd(m,n) = n \wedge$
$mod(m,n) > 0 \rightarrow gcd(m,n) = gcd(n, mod(m,n))$

Program extraction

Prove the formula
$\forall m > 0\, \forall n > 0\, \exists k$
$\quad k \mid m \wedge k \mid n \wedge$
$\quad \forall l(l \mid m \wedge l \mid n \rightarrow l \mid k)$

From a formal proof extract a program *gcd* provably satisfying the specification, that is, the formula
$\forall m > 0\, \forall n > 0$
$\quad gcd(m,n) \mid m \wedge gcd(m,n) \mid n \wedge$
$\quad \forall l(l \mid m \wedge l \mid n \rightarrow l \mid gcd(m,n))$
is provable

Program

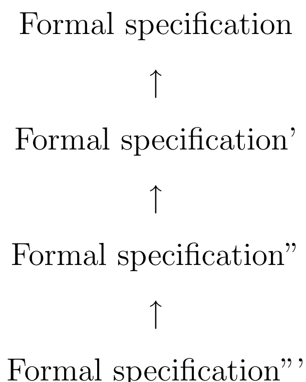
```
function mod (m,n:integer, m,n>0) : integer;
          begin
                 if m < n then mod := m
                            else  mod := mod(m-n,n)
          end

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
```

2.3 Programming by transformation

In Example 2.2 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 *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



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 confused.

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,
- **modularisation**, i.e. the programming task is divided into small manageable pieces that can be solved independently.

In this example modularisation took place by dividing the development into four relatively small steps and separating the problem of computing the modulus as an independent programming task. Furthermore, we abstracted from a concrete representation of natural numbers and the arithmetical operations of addition, subtraction and multiplication.

2.4 Programming by extraction from proofs

The right hand side of example 2.2 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 gcd) is extracted (fully automatically) that provable meets the specification, that is,

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

   is provable (in our example, $\forall m, n\, (gcd(m, n)$ 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. [Dal] 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

formula	$\equiv$	data type
constructive proof of formula A	$\equiv$	program of data type A

The constructive proof calculus studied in this course will be *natural deduction.* We will mainly follow the books [Dal] and [TS] as well as the article [Sch].

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.

3 Logic

In this chapter we study the syntax and semantics of many-sorted first-order predicate logic, which is the foundation for the specification, modelling and implementation of Abstract Data Types.

3.1 Signatures and algebras

The purpose of a signature is to provide names for objects and operations and fix their format. Hence a signature is very similar to the programming concept of an interface.

3.1.1 Definition

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

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

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

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

  We require that for every sort there is at least one constant of that sort.

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.

3.1.2 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 *nonempty* set A_s, called the **carrier set** of the sort s.
- For each constant $c\colon s$ in Ω an element $c^A \in A_s$.
- For each operation $f\colon s_1 \times \ldots \times s_n \to s$ in Ω a function

  $$f^A\colon A_{s_1} \times \ldots \times A_{s_n} \to A_s$$

3.1.3 Remarks

1. In the definition of an algebra (3.1.1) the expression $f\colon s_1 \times \ldots \times s_n \to s$ is meant *symbolically*, i.e. '$\times$' and '$\to$' are to be read as uninterpreted symbols. In the definition of an algebra (3.1.2), however, we used the familiar mathematical notation for set-theoretic functions to communicate by $f^A\colon A_{s_1} \times \ldots \times A_{s_n} \to A_s$ 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 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 Σ-*algebra* we mean an algebra for the signature Σ.

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*.

3.1.4 Example

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

$$\begin{aligned} S &= \{\mathsf{nat}, \mathsf{boole}\} \\ \Omega &= \{0\colon \mathsf{nat},\ \mathrm{T}\colon \mathsf{boole},\ \mathrm{F}\colon \mathsf{boole},\ \mathrm{add}\colon \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat},\ \mathrm{le}\colon \mathsf{nat} \times \mathsf{nat} \to \mathsf{boole}\} \end{aligned}$$

We follow [Tuc] and display signatures in a box:

Signature	Σ
Sorts	$\mathsf{nat},\ \mathsf{boole}$
Constants	$0\colon \mathsf{nat},\ \mathrm{T}\colon \mathsf{boole}, \mathrm{F}\colon \mathsf{boole}$
Operations	$\mathrm{add}\colon \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$
	$\leq\colon \mathsf{nat} \times \mathsf{nat} \to \mathsf{boole}$

The Σ-algebra A of natural numbers with 0, and addition and the relation $\leq$ is given by

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

$$A_{\mathsf{nat}} = \mathbf{N}, \quad A_{\mathsf{boole}} = \mathbf{B},$$

the constants 0, T, F, i.e.

$$0^A = 0, \quad \mathrm{T}^A = \mathrm{T}, \quad \mathrm{F}^A = \mathrm{F},$$

the operations of addition on $\mathbf{N}$ and the comparison relation $\leq$ viewed as a boolean function, i.e. for all $n, m \in \mathbf{N}$,

$$\mathrm{add}^A(n, m) = n + m,$$

$$\leq^A (n,m) = \begin{cases} \mathrm{T} & \text{if } n \leq m \\ \mathrm{F} & \text{otherwise} \end{cases}$$

Again we use the more readable box notation [Tuc]:

Algebra	A
Carriers	$\mathbf{N},\ \mathbf{B}$
Constants	$0,\ \mathrm{T},\ \mathrm{F}$
Operations	$+\colon \mathbf{N} \times \mathbf{N} \to \mathbf{N}$
	$\leq\colon \mathbf{N} \times \mathbf{N} \to \mathbf{B}$

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

$$B_{\mathsf{nat}} = \mathbf{N}^+, \quad B_{\mathsf{boole}} = \mathbf{B},$$

$$0^B = 1, \quad \mathrm{T}^B = \mathrm{T}, \quad \mathrm{F}^B = \mathrm{F},$$

$$\mathrm{add}^B(n,m) = n * m \text{ for all } n, m \in \mathbf{N}^+.$$

$$\leq^A (n,m) = \begin{cases} \mathrm{T} & \text{if } n \mid m \\ \mathrm{F} & \text{otherwise} \end{cases}$$

Written in a box

Algebra	B
Carriers	$\mathbf{N}^+, \mathbf{B}$
Constants	$1, \mathrm{T}, \mathrm{F}$
Operations	$*\colon \mathbf{N}^+ \times \mathbf{N}^+ \to \mathbf{N}^+$
	$\mid\colon \mathbf{N}^+ \times \mathbf{N}^+ \to \mathbf{B}$

3.1.5 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 Σ in example 3.1.4 we would have exchanged the order of the sorts nat and 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 boole would be associated with the set $\mathbf{N}$ and nat with $\mathbf{B}$, and consequently the arities of the operations of Σ would not fit with the operations of the algebra A.

Therefore: When displaying signatures and algebras in boxes *order matters.*

2. Operations with target sort boole are often called *predicates*. In many logic text books predicates are not represented by boolean functions, but treated as separate entities.

3.2 Terms and their semantics

The constants and operations of a signature Σ can be used to build formal expressions, called *terms*, which denote elements of a given Σ-algebra.

3.2.1 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 Σ of sort s is a term of sort s.

(iii) If $f \colon s_1 \times \ldots \times s_n \to s$ is an operation in Σ, 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 $\mathrm{T}(\Sigma, X)_s$.

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 $\mathrm{T}(\Sigma)_s$. Clearly $\mathrm{T}(\Sigma)_s = \mathrm{T}(\Sigma, \emptyset)_s$.

3.2.2 Example

For the signature Σ of example 3.1.4 and the set of variables $X := \{x, y\}$ the following are examples of terms in $\mathrm{T}(\Sigma, X)$:

x

0

$\mathrm{add}(0, y)$

$\mathrm{add}(\mathrm{add}(0, x), y)$

$\mathrm{add}(\mathrm{add}(0, 0), \mathrm{add}(x, x))$

$\mathrm{add}(0, \mathrm{add}(0, \mathrm{add}(0, 0)))$

The second and the last of these terms are closed.

In order to declare the *semantics* of terms in a Σ-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.

3.2.3 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** $\alpha \colon X \to A$ is a function assigning to every variable $x \in X_s$ an element $\alpha(x) \in A_s$.

Given a variable assignment $\alpha \colon X \to A$ we define for each term $t \in \mathrm{T}(\Sigma, X)_s$ 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. $t \in \mathrm{T}(\Sigma)$ $(= \mathrm{T}(\Sigma, \emptyset))$ the variable assignment α and rule (i) are obsolete and we write t^A instead of $t^{A,\alpha}$.

3.2.4 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 **composite term** $f(t_1, \ldots, t_n)$ using the values of the function at the components $t_1, \ldots, t_n$.

3.2.5 Exercise

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

3.2.6 Example

Let us calculate the values of the terms in example 3.1.4 in the Σ-algebra A under the variable assignment $\alpha \colon \{x, y\} \to \mathbf{N}$, $\alpha(x) := 3$ and $\alpha(y) := 5$.

$$x^{A,\alpha} = \alpha(x) = 3$$

$$0^{A,\alpha} = 0^A = 0$$

$$\mathrm{add}(0, y)^{A,\alpha} =$$

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

$$\mathrm{add}(\mathrm{add}(0, 0), \mathrm{add}(x, x))^{A,\alpha} =$$

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

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

3.2.7 Definition

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

Algebra	$\mathrm{T}(\Sigma, X)$
Carriers	$\mathrm{T}(\Sigma, X)_s \quad (s \in S)$
Constants	$c^{\mathrm{T}(\Sigma,X)} := c$
Operations	$f^{\mathrm{T}(\Sigma,X)}(t_1, \ldots, t_n) := f(t_1, \ldots, t_n)$

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

3.3 Formulas and their semantics

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

3.3.1 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) $\perp$ is a formula, called **absurdity**.

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

(iii) If P and Q are formulas then $P \to Q$, $P \land Q$, and $P \lor Q$ are formulas, called **implication** ('if then'), **conjunction** ('and') and **disjunction** ('or'), respectively.

(iv) If P is a formula then $\forall x\, P$ and $\exists x\, P$ are formulas for every variable $x \in X$, called **universal quantification** ('for all') and **existential quantification** ('exists'), respectively.

Formulas over a signature Σ are also called **Σ-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 $\mathsf{FV}(P)$ 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 $\mathsf{FV}(P) = \emptyset$.

We set

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

and use the abbreviation

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

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

3.3.2 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. A formula is **universal** if it is of the form $\forall x_1 \ldots \forall x_n\, P$ where P is quantifier free.

3.3.3 Abbreviations

Formula	Abbreviation
$P \to \bot$	$\neg P$ (negation)
$\forall x_1 \forall x_2 \ldots \forall x_n\, P$	$\forall x_1, x_2, \ldots, x_n P$
$\exists x_1 \exists x_2 \ldots \exists x_n\, P$	$\exists x_1, x_2, \ldots, x_n\, P$
$\forall x_1, \ldots, x_n P$, where $\{x_1, \ldots, x_n\} = \mathsf{FV}(P)$	$\forall\, P$ (closure of P)
$(P \to Q) \wedge (Q \to P)$	$P \leftrightarrow Q$ (equivalence)
$t = \mathrm{T}$	t (provided t is of sort boole)

3.3.4 Examples

$$P_1 :\equiv \mathrm{T} = \mathrm{F}$$

$$P_2 :\equiv x = 0 \to \mathrm{add}(y, x) = y$$

$$P_3 :\equiv \exists y\, (x = y \to \forall z\, x = z)$$

$$P_4 :\equiv \forall x\, (0 \leq x = \mathrm{T})$$

P_1 and P_2 are quantifier free.

P_1 is an equation. P_4 is universal.

P_1 and P_4 are closed.

$\mathsf{FV}(P_2) = \{x, y\}$, $\mathsf{FV}(P_3) = \{x\}$.

P_4 can be abbreviated $\forall x\,(0 \leq x)$.

3.3.5 Exercise

Let Σ be the signature of example 3.1.4 and A the Σ-algebra of the natural numbers with zero, addition and the 'less-or-equal' relation. Write down Σ-formulas expressing in A the following statements.

(a) x is an even number.

(b) x is greater than y.

(c) x is the average of y and z.

In order to precisely declare the **semantics** of a formula we define what it means for a formula to be true in an algebra.

3.3.6 Definition (Semantics of formulas)

Let $\Sigma = (S, \Omega)$ be a signature, $X = (X_s)_{s \in S}$ a set of variables, A a Σ-algebra $\alpha\colon X \to A$, 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** α', or 'A, α **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) $A, \alpha \models P \wedge Q$ iff $A, \alpha \models P$ and $A, \alpha \models Q$.

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

$A, \alpha \models P \to Q$ 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 $A, \alpha_x^a \models P$ for all $a \in A_s$ (provided x is of sort s).

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

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

For closed Σ-formulas P the variable assignment is obviously redundant and we write

$$A \models P$$

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

$$A \models \Gamma,$$

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

3.4 Logical consequence, logical validity, satisfiability

We may now make precise what it means that a formula P is a logical consequence of a set of formulas.

3.4.1 Definition (Logical consequence)

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

$$\Gamma \models P,$$

if P is true in all models of Γ, that is,

$$A \models \Gamma \quad \text{implies} \quad A \models P, \qquad \text{for all } \Sigma\text{-algebras } A$$

3.4.2 Definition (Logical validity)

A closed Σ-formula P is said to be **(logically) valid**, written

$$\models P,$$

if P is rue in all Σ-algebras, that is $A \models P$ for all Σ-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.

3.4.3 Definition (Satisfiability)

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

3.4.4 Exercise

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

$$\begin{array}{rcl} P \text{ valid} & \Leftrightarrow & \{\neg P\} \text{ unsatisfiable (that is, not satisfiable)} \\ P \text{ satisfiable} & \Leftrightarrow & \{\neg P\} \text{ not valid} \end{array}$$

3.4.5 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, by Church's Theorem, 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.

3.4.6 Examples

Consider a signature with the sorts nat and boole and the operation $<\colon \mathsf{nat} \times \mathsf{nat} \to \mathsf{boole}$. Then the formula

$$\exists x\, \forall y\, (x < y) \to \forall y\, \exists x\, (x < y)$$

is a tautology. The formula

$$\forall x, y\, (x < y \to \exists z\, (x < z \wedge z < y))$$

is satisfiable, but not a tautology (why?). Set

$$\Gamma := \{\forall x\, \neg(x < x),\ \forall x.y.z\, (x < y \wedge y < z \to x < z)\}$$

Then the formula

$$P :\equiv \forall x, y\, (x < y \to \neg y < x)$$

is a logical consequence of Γ, that is, $\Gamma \models P$.

3.5 Substitutions

Now we study the operation of replacing the free variables occurring in a term or formula by terms.

3.5.1 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 $\theta\colon X \to \mathrm{T}(\Sigma, Y)$ 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 θ we define for every $t \in \mathrm{T}(\Sigma, X)$ a term $t\theta \in \mathrm{T}(\Sigma, Y)$ by

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

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 θ 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_i and t_i have the same sort, and the variables x_i are all distinct.

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

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

It can be easily proved that

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

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

(c) If $\theta\colon X \to \mathrm{T}(\Sigma, Y)$ is a substitution and $\alpha\colon Y \to A$ is a variable assignment, then the variable assignments $\theta^{A,\alpha}\colon X \to A$ 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 $P(t) :\Leftrightarrow t(\theta\sigma) = (t\theta)\sigma$). 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).* $x(\theta\sigma) = (\theta\sigma)(x) = (x\theta)\sigma$

(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 $i = 1, \ldots, n$.

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

3.5.2 Exercise

Let $\theta\colon Y \to \mathrm{T}(\Sigma, X)$ be a substitution. Note that θ can also be viewed as a variable assignment in the term algebra $\mathrm{T}(\Sigma, X)$ (see definition 3.2.7). Prove by induction on terms that for any term $t \in \mathrm{T}(\Sigma, Y)$

$$t^{\mathrm{T}(\Sigma,X),\theta} = t\theta$$

3.5.3 Example

Let Σ and A be as in example 3.1.4 and $X := \{x, y\}$. Define the substitution $\theta\colon X \to \mathrm{T}(\Sigma, X)$ by $\theta := \{\mathrm{add}(y, y)/x\}$, i.e. $\theta(x) = \mathrm{add}(0, y)$, $\theta(y) = y$. Now we have for example

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

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

$$\begin{array}{rcl} \theta^{A,\alpha}(x) & = & \mathrm{add}(y, y)^{A,\alpha} = 7 + 7 = 14 \\ \theta^{A,\alpha}(y) & = & y^{A,\alpha} = 7 \end{array}$$

Now for example

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

Note that also

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

The following lemma proves that the observation above is not a coincidence.

3.5.4 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 $t \in \mathrm{T}(\Sigma, X)$, substitution $\theta\colon X \to \mathrm{T}(\Sigma, Y)$ and variable assignment $\alpha\colon Y \to A$

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

For a substitution $\{r/x\}$ this means

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

Proof. The equation is proved by induction on terms.

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

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

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

$$\begin{array}{rcll} (f(t_1,\ldots,t_n)\theta)^{A,\alpha} & = & f(t_1\theta,\ldots,t_n\theta)^{A,\alpha} & \\ & = & f^A((t_1\theta)^{A,\alpha},\ldots,(t_n\theta)^{A,\alpha}) & \\ & = & f^A(t_1^{A,(\theta^{A,\alpha})},\ldots,t_n^{A,(\theta^{A,\alpha})}) & \text{by induction hypothesis} \\ & = & f(t_1,\ldots,t_n)^{A,(\theta^{A,\alpha})} & \end{array}$$

3.5.5 Definition (Applying substitutions to formulas)

Let $\theta\colon X \to \mathrm{T}(\Sigma, Y)$ be a substitution and P a formula with $\mathsf{FV}(P) \subseteq X$. The intuitive definition of applying the θ to A is

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

So, for example

$$(\exists y(x + y + 1 = 0))\{y * y/x\}$$

is *not*

$$\exists y(y * y + y + 1 = 0).$$

but

$$\exists z(y * y + z + 1 = 0)$$

$P\theta$ can be defined more precisely by recursion on P

Exercise: Carry this out.

3.5.6 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 $\mathsf{FV}(P) \subseteq X$, substitution $\theta\colon X \to \mathrm{T}(\Sigma, Y)$ and variable assignment $\alpha\colon Y \to A$

$$A, \alpha \models P\theta \quad \Leftrightarrow \quad A, \theta^{A,\alpha} \models P$$

For a substitution $\{r/x\}$ this means

$$A, \alpha \models P\{r/x\} \quad \Leftrightarrow \quad A, \alpha_x^{r^{A,\alpha}} \models P$$

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

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

for $i = 1, 2$. Therefore

$$\begin{aligned} A, \alpha \models (t_1 = t_2)\theta &\Leftrightarrow A, \alpha \models t_1\theta = t_2\theta \\ &\Leftrightarrow (t_1\theta)^{A,\alpha} = (t_2\theta)^{A,\alpha} \\ &\Leftrightarrow t_1^{A,(\theta^{A,\alpha})} = t_2^{A,(\theta^{A,\alpha})} \\ &\Leftrightarrow A, \theta^{A,\alpha} \models t_1 = t_2 \end{aligned}$$

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

3.5.7 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 α a variable assignment. If

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

then for every term t and every formula P

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

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

Proof. Assume $r^{A,\alpha} = r'^{A,\alpha}$. By the substitution lemma for formulas, 3.5.4, we have

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

Similarly, by the substitution lemma for formulas, 3.5.4, we have

$$A, \alpha \models P\{r/x\} \Leftrightarrow A, \alpha_x^{r^{A,\alpha}} \models P \Leftrightarrow A, \alpha_x^{r'^{A,\alpha}} \models P \Leftrightarrow A, \alpha \models P\{r'/x\}$$

3.6 Other Logics

First-order predicate logic is a general purpose logic. It is used to

- formulate and answer questions concerning the foundations of mathematics,
- specify and verify programs written in all kinds of programming languages.

There exist many other, more specialized, logics which are tailored for specific kinds of problems in computer science. For example:

- Hoare logic – imperative programs
- higher-order logic – functional programs
- clausal logic – logic programming, AI
- modal/temporal/process logic – distributed processes
- bounded/linear logic – complexity analysis
- equational logic – hardware, rapid prototyping

We will study equational logic in Chapter 8.

3.7 Summary and Exercises

The following notions where central in this chapter.

- Signatures and algebras.
- Terms and their value in an algebra, $t^{A,\alpha}$.
- Formulas and their meaning in an algebra, $A, \alpha \models P$.
- Induction on terms.
- Logical consequence

Exercises.

1. In this and the next exercise we consider terms of an arbitrary signature. Define the *size* of a term t, that is, the number of occurrences of variables, constants and operations in t, by recursion on t.

2. We define the *depth* of a term t as the length of the longest branch in the syntax tree of t, that is,

$\mathsf{depth}(t) = 0$, if t is a constant or a variable.

$\mathsf{depth}(f(t_1, \ldots, t_n)) = 1 + \mathsf{maximum}\{\mathsf{depth}(t_1), \ldots, \mathsf{depth}(t_1)\}$

Show that $\mathsf{size}(t) \leq (1 + \mathsf{arity}(t))^{\mathsf{depth}(t)}$, where $\mathsf{arity}(t)$ is 0 if t is a constant or a variable, and otherwise the largest arity of an operation occurring in t.

Hint: First, give a recursive definition of $\mathsf{arity}(t)$.

3. Compute the value of the term $t := f(x, f(x, x))$ in the algebra C with carrier $\mathbf{N}$ and $f^C(n, m) := n + 2 * m$ under the valuation α with $\alpha(x) := 3$.

4. Let Σ be the signature with one sort, one constant, 0, and two binary operations, + and $*$. Let R be the Σ-algebra with the real numbers as carrier, the constant 0, and the usual addition and multiplication of real numbers. Write down formulas that express in R the following statements:

(a) $x \leq y$.

(b) $x = 1$.

4 Proofs

In this chapter we study a system of simple proof rules for deriving tautologies, that is, logically valid formulas. The famous *Completeness Theorem*, by the Austrian logician *Kurt Gödel*, states that this system of rules suffices to derive in fact *all* tautologies.

K Gödel (1906-1978)

4.1 Natural Deduction

The proof calculus of *Natural Deduction* was first introduced by Gentzen and further developed by Prawitz.

Compared with other proof calculi, e.g. Sequent Calculi, or Hilbert Calculi, Natural Deduction has the advantage of being

- close to the natural way of human reasoning, and thus easy to learn;
- closely related to functional programming, and thus is particularly well suited for program synthesis from proofs, which we will study in the next chapter.

We will first study the rules for the logical connectives and quantifiers. The rules for equality will be dealt with in section 4.2.

4.1.1 Definition (Derivation)

A *derivation* is a finite tree (drawn correctly, that is, leaves on top and root at the bottom), where each node is labelled by a formula and a rule according to figure 3 on page 32. In order to understand these rules, one needs to know the following:

1. An application of the rule $\to^+$, deriving $P \to Q$ from Q, *binds* every (unlabelled) occurrence of P at a leaf above that rule. We mark such a binding by attaching to the leaf P and the rule $\to^+$ a fresh label u.

2. The *free assumptions* of a derivation d, written $\mathsf{FA}(d)$, are those formulas P occurring *unlabelled* at a leaf of d (that is, those P that are not bound by a rule $\to^+$).

3. In the $\forall^+$ rule, the label (*) means the so-called *variable condition*, that is, the requirement that x must not occur free in any free assumption above that rule.

4. In the $\exists^-$ rule, the label (**) means the restriction that x must not be free in Q.

In the following, $P(x)$ stands for a formula possibly containing x free, and $P(t)$ stands for the formula $P(x)\{t/x\}$. For each logical connective there are two kinds of rules:

Introduction rules, describing how to *obtain* a formula built from that connective;

Elimination rules, describing how to *use* a formula built from that connective.

The formula at the root of a derivation d is called the *end formula* of d. If d is a derivation with $\mathsf{FA}(d) \subseteq \Gamma$ and end formula P, we say *d is derivation of P from* Γ and write

$$\Gamma \vdash d\colon P.$$

4.1.2 Examples (propositional connectives)

1. We begin with a derivation involving the *conjunction introduction rule*, $\wedge^+$ and the *conjunction elimination rules*, $\wedge^-_{\mathrm{l}}$ and $\wedge^-_{\mathrm{r}}$. We derive from the assumption $P \wedge Q$ the formula $Q \wedge P$:

$$\dfrac{\dfrac{P \wedge Q}{Q}\,\wedge^-_{\mathrm{r}} \qquad \dfrac{P \wedge Q}{P}\,\wedge^-_{\mathrm{l}}}{Q \wedge P}\,\wedge^+$$

2. If we add to the derivation in example 1 an application of the *implication introduction rule*, $\to^+$, we obtain a derivation of $P \wedge Q \to Q \wedge P$ that does not contain free assumptions:

$$\dfrac{\dfrac{\dfrac{u : P \wedge Q}{Q}\,\wedge^-_{\mathrm{r}} \qquad \dfrac{u : P \wedge Q}{P}\,\wedge^-_{\mathrm{l}}}{Q \wedge P}\,\wedge^+}{P \wedge Q \to Q \wedge P}\,\to^+ u$$

3. In the following derivation of the formula $P \to (Q \to P)$ we use the rule $\to^+$ twice. The upper instance of this rule is used with the formula Q, without Q actually occurring as an open assumption.

	Introduction rules	Elimination rules
$\wedge$	$\dfrac{P \qquad Q}{P \wedge Q}\, \wedge^+$	$\dfrac{P \wedge Q}{P}\, \wedge^-_{\mathrm{l}} \qquad \dfrac{P \wedge Q}{Q}\, \wedge^-_{\mathrm{r}}$
$\to$	$\begin{array}{c} u : P \\ \vdots \\ \dfrac{\mathrm{Q}}{P \to Q}\, \to^+ u \end{array}$	$\dfrac{P \to Q \qquad P}{Q}\, \to^-$
$\vee$	$\dfrac{P}{P \vee Q}\, \vee^+_{\mathrm{l}} \qquad \dfrac{Q}{P \vee Q}\, \vee^+_{\mathrm{r}}$	$\dfrac{P \vee Q \qquad P \to R \qquad Q \to R}{R}\, \vee^-$
$\bot$		$\dfrac{\bot}{P}\, \mathrm{efq} \qquad \dfrac{\neg\neg P}{P}\, \mathrm{raa}$
$\forall$	$\dfrac{P(x)}{\forall x\, P(x)}\, \forall^+ \qquad (*)$	$\dfrac{\forall x\, P(x)}{P(t)}\, \forall^-$
$\exists$	$\dfrac{P(t)}{\exists x\, P(x)}\, \exists^+$	$\dfrac{\exists x\, P(x) \qquad \forall x\, (P(x) \to Q)}{Q}\, \exists^- \qquad (**)$

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

$$\dfrac{\dfrac{u : P}{Q \to P}\, \to^+ v}{P \to (Q \to P)}\, \to^+ u$$

4. Next let us derive $P \to (Q \to R)$ from $P \wedge Q \to R$. Here we use for the first time the *implication elimination rule*, $\to^-$, also called *modus ponens*. The easiest way to find the derivations is to construct it "bottom up".

$$\dfrac{\dfrac{\dfrac{P \wedge Q \to R \qquad \dfrac{u : P \qquad v : Q}{P \wedge Q}\, \wedge^+}{R}\, \to^-}{Q \to R}\, \to^+ v}{P \to (Q \to R)}\, \to^+ u$$

5. In order to familiarise ourselves with the *disjunction introduction rules*, $\vee^+_{\mathrm{l}}$, $\vee^+_{\mathrm{r}}$, and the *disjunction elimination rule*, $\vee^-$, we derive $Q \vee P$ from $P \vee Q$.

$$\dfrac{P \lor Q \qquad \dfrac{\dfrac{u : P}{Q \lor P}\,\lor^+_{\mathrm{r}}}{P \to Q \lor P}\,\to^+ u \qquad \dfrac{\dfrac{v : Q}{Q \lor P}\,\lor^+_{\mathrm{l}}}{Q \to Q \lor P}\,\to^+ v}{Q \lor P}\,\lor^-$$

6. As a slightly more complicated example 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.

$$\dfrac{\dfrac{\dfrac{u : P \land (Q \lor R)}{Q \lor R}\,\land^-_{\mathrm{r}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{u : P \land (Q \lor R)}{P}\,\land^-_{\mathrm{l}} \qquad v : Q}{P \land Q}\,\land^+}{(P \land Q) \lor (P \land R)}\,\lor^+_{\mathrm{l}}}{Q \to (P \land Q) \lor (P \land R)}\,\to^+ v \qquad \dfrac{\dfrac{\dfrac{\dfrac{u : P \land (Q \lor R)}{P}\,\land^-_{\mathrm{l}} \qquad w : R}{P \land R}\,\land^+}{(P \land Q) \lor (P \land R)}\,\lor^+_{\mathrm{r}}}{R \to (P \land Q) \lor (P \land R)}\,\to^+ w}{(P \land Q) \lor (P \land R)}\,\lor^-}{P \land (Q \lor R) \to (P \land Q) \lor (P \land R)}\,\to^+ u$$

7. Finally, we turn our attention to the rules concerning absurdity, $\bot$, namely *ex-falso-quodlibet*, efq, and *reductio-ad-absurdum*, raa. Recall that $\neg P$ is shorthand for $P \to \bot$, and therefore $\neg\neg P$ stands for $(P \to \bot) \to \bot$. We derive *Peirce's law* $(P \to Q) \to P \vdash P$:

$$\dfrac{\dfrac{\dfrac{u : \neg P \qquad \dfrac{(P \to Q) \to P \qquad \dfrac{\dfrac{\dfrac{u : \neg P \qquad v : P}{\bot}\,\to^-}{Q}\,\mathrm{efq}}{P \to Q}\,\to^+ v}{P}\,\to^-}{\bot}\,\to^-}{\neg\neg P}\,\to^+ u}{P}\,\mathrm{raa}$$

8. The rule ex-falso-quodlibet is weaker than reductio-ad-absurdum in the sense that the former can be obtained from the latter: From the assumption $\bot$ we can derive any formula P without using ex-falso-quodlibet (but using reductio-ad-absurdum instead):

$$\dfrac{\dfrac{\bot}{(P \to \bot) \to \bot}\,\to^+ u}{P}\,\mathrm{raa}$$

4.1.3 Examples (quantifier rules)

1. In the following derivation of $\forall y\, P(y+1)$ from $\forall x\, P(x)$ we use the *for-all introduction rule*, $\forall^+$, and the *for-all elimination rule*, $\forall^-$:

$$\dfrac{\dfrac{\forall x\, P(x)}{P(x+1)}\;\forall^-}{\forall x\, P(x+1)}\;\forall^+$$

In the application of $\forall^+$ the variable condition is satisfied, because x is not free in $\forall x\, P(x)$.

2. Find out what's wrong with the following 'derivations'.

$$\dfrac{\dfrac{\forall y(x < 1 + y)}{x < 1 + 0}\;\forall^-}{\forall x(x < 1 + 0)}\;\forall^+$$

$$\dfrac{\dfrac{\dfrac{\forall x(\forall y(x < y + 1) \to x = 0)}{\forall y(y < y + 1) \to y = 0}\;\forall^- \qquad \forall y(y < y + 1)}{y = 0}\;\to^-}{\forall y(y = 0)}\;\forall^+$$

3. The *exists introduction rule*, $\exists^+$, and the *exists elimination rule*, $\exists^-$ are used in the following derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{u : \forall x\, (x - 1) + 1 = x}{(x - 1) + 1 = x}\;\forall^-}{\exists y(y + 1 = x)}\;\exists^+}{\forall x\, \exists y(y + 1 = x)}\;\forall^+}{\forall x\, ((x - 1) + 1 = x) \to \forall x\, \exists y(y + 1 = x)}\;\to^+ u$$

4. Let us derive from the assumptions $\exists x\, P(x)$ and $\forall x(P(x) \to Q(f(x)))$ the formula $\exists y\, Q(y)$:

$$\dfrac{\exists x\, P(x) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x(P(x) \to Q(f(x)))}{P(x) \to Q(f(x))}\;\forall^- \qquad u : P(x)}{Q(f(x))}\;\to^-}{\exists y\, Q(y)}\;\exists^+}{P(x) \to \exists y\, Q(y)}\;\to^+ u}{\forall x\, (P(x) \to \exists y\, Q(y))}\;\forall^+}{\exists y\, Q(y)}\;\exists^-$$

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

4.2 Equality rules

So far we only considered the Natural Deduction rules for logic without equality. Here are the rules for equality:

Reflexivity $$\frac{}{t = t}\,\text{refl}$$

Symmetry $$\frac{s = t}{t = s}\,\text{sym}$$

Transitivity $$\frac{r = s \qquad s = t}{r = t}\,\text{trans}$$

Compatibility $$\frac{s_1 = t_1 \qquad \ldots \qquad s_n = t_n}{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)}\,\text{comp}$$

for every operation f of n arguments.

4.2.1 Example

Let us derive from the assumptions $\forall x, y\,(x + y = y + x)$ and $\forall x\,(x + 0 = x)$ the formula $\forall x, y\,((0 + x) * y = x * y)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x\,\forall y\,(x + y = y + x)}{\forall y\,(0 + y = y + 0)}\,\forall^-}{0 + x = x + 0}\,\forall^- \qquad \dfrac{\forall x\,(x + 0 = x)}{x + 0 = x}\,\forall^-}{0 + x = x}\,\text{trans} \qquad \dfrac{}{y = y}\,\text{refl}}{(0 + x) * y = x * y}\,\text{comp}}{\forall y\,((0 + x) * y = x * y)}\,\forall^+}{\forall x\,\forall y\,((0 + x) * y = x * y)}\,\forall^+$$

4.2.2 Definition

$\Gamma \vdash_{\mathrm{c}} P \quad :\Leftrightarrow \quad \Gamma \vdash d\colon P$ for some derivation d.

(P is derivable from Γ in *classical logic*)

$\Gamma \vdash_{\mathrm{i}} P \quad :\Leftrightarrow \quad \Gamma \vdash d \colon P$ for some derivation d not using the rule reductio-ad-absurdum.
(P is derivable from Γ in *intuitionistic logic*)

$\Gamma \vdash_{\mathrm{m}} P \quad :\Leftrightarrow \quad \Gamma \vdash d \colon P$ for some derivation d using neither the rule reductio-ad-absurdum nor the rule ex-falso-quodlibet.

(P is derivable from Γ in *minimal logic*)

4.2.3 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

$$r = s \vdash_{\mathrm{m}} t(r) = t(s)$$

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 $r = s \vdash_{\mathrm{m}} t = t$ 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 $r = s \vdash_{\mathrm{m}} r = s$ 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 $r = s \vdash_{\mathrm{m}} t_i(r) = t_i(s)$ for $i = 1, \ldots, n$. One application of the compatibility rule yields the required sequent.

4.2.4 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:

$$r = s \vdash_{\mathrm{m}} P(r) \leftrightarrow P(s)$$

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

$$r = s \vdash_{\mathrm{m}} t_1(r) = t_2(r) \leftrightarrow t_1(s) = t_2(s)$$

By Lemma 4.2.3 we have already derivations of

$$r = s \vdash_{\mathrm{m}} t_1(r) = t_1(s) \qquad \text{and} \qquad r = s \vdash_{\mathrm{m}} t_2(r) = t_2(s)$$

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.

4.3 Soundness and completeness

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

4.3.1 Soundness Theorem

If $\Gamma \vdash_{\mathrm{c}} P$ 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 Γ and every formula P,

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

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

4.3.2 Completeness Theorem (Gödel)

If $\Gamma \models P$ then $\Gamma \vdash_{\mathrm{c}} P$.

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

The proof of this theorem is beyond the scope of this course. Detailed expositions of the proof can be found in any textbook on Mathematical Logic, for example [Sho].

The following consequence of the Completeness Theorem refers to the notion of consistency.

4.3.3 Definition (Consistency)

A (possibly infinite) set of formulas Γ is called **consistent** if $\Gamma \not\vdash_{\mathrm{c}} \bot$, that is there is no (classical) derivation of $\bot$ from assumptions in Γ.

In other words: A set of formulas Γ is consistent if and only if no contradiction can be derived from Γ.

4.3.4 Satisfiability Theorem

Every consistent set of formulas has a model.

Proof. Exercise.

Another important consequence of Gödel's Completeness Theorem is the fact that all logically valid formulas can be effectively enumerated.

4.3.5 Satisfiability Theorem

The set of all logically valid formulas is recursively enumerable.

Proof. Exercise.

4.4 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. Simar axioms could be stated for binary number, lists, finite trees etc., more generally for freely generated data types.

4.4.1 Axioms for the booleans

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

Boole 1 $$\frac{}{\mathrm{T} \neq \mathrm{F}}\ \text{boole1}$$

Boole 2 $$\frac{}{\forall x\,(x = \mathrm{T} \lor x = \mathrm{F})}\ \text{boole2}$$

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

4.4.2 Lemma

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

We leave the proof as an exercise to the reader.

Hint: We have to derive $\forall x\,((x = \mathrm{T} \to \bot) \leftrightarrow x = \mathrm{F})$. For the implication from left to right use the second boolean axiom, for the converse implication use the first boolean axiom.

4.4.3 Peano Axioms

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

G Peano (1858 - 1932)

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

Peano 1 $\dfrac{}{0 \neq t+1}\,\text{peano1}$

Peano 2 $\dfrac{}{s+1 = t+1 \to s = t}\,\text{peano2}$

Induction $\dfrac{P(0) \qquad \forall x\,(P(x) \to P(x+1))}{\forall x\,P(x)}\,\text{ind}$

4.4.4 Remarks

1. 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.

2. Similar axioms can be introduced for data types such as lists or trees.

4.5 Summary and Exercises

- Derivations: minimal, intuitionistic and classical.
- The Soundness and Completeness Theorem for First-Order Logic.
- Axioms and rules for equality and data types.

Exercises.

Derive the following formulas

Propositional logic

1. Minimal logic

(a) $(P \to \neg Q) \to (Q \to \neg P)$

(b) $(P \to (Q \to R)) \to ((P \to Q) \to (P \to R))$

(c) $P \to \neg\neg P$

(d) $\neg(P \wedge \neg P)$

(e) $(P \wedge (Q \vee R)) \leftrightarrow ((P \wedge Q) \vee (P \wedge R))$

(f) $(P \vee Q) \to \neg(\neg P \wedge \neg Q)$

(g) $\neg(P \leftrightarrow \neg P)$

2. Intuitionistic logic

(a) $(P \wedge \neg P) \to R$

(b) $(\neg P \vee Q) \to (P \to Q)$

(c) $(\neg\neg P \to P) \leftrightarrow ((\neg P \to P) \to P)$

(d) $(P \vee Q) \to (\neg P \to Q)$

3. Classical logic

(a) $\neg\neg P \to P$

(b) $(\neg P \to P) \to P$

(c) $P \vee \neg P$

(d) $\neg(\neg P \wedge \neg Q) \to (P \vee Q)$

(e) $\neg(\neg P \vee \neg Q) \to (P \wedge Q)$

(f) $\neg(P \to Q) \to P \wedge \neg Q$

Quantifier logic

1. Minimal logic

(a) $\forall x\,(P(x) \to Q(x)) \to (\forall x\,P(x) \to \forall x\,Q(x))$

(b) $\forall x\,(P(x) \to \exists y\,P(y))$

(c) $\forall x\,P(x) \to \exists x\,P(x)$

(d) $\exists x\,(P(x) \vee Q(x)) \leftrightarrow \exists x\,P(x) \vee \exists x\,Q(x)$

(e) $\exists x\,(P(x) \wedge Q(x)) \to \exists x\,P(x) \wedge \exists x\,Q(x)$

(f) $\exists x\,\neg P(x) \to \neg\forall x\,P(x)$

(g) $\neg\neg\forall x\,P(x) \to \forall x\,\neg\neg P(x)$

2. Intuitionistic logic

(a) $\exists x\,(P(x) \wedge \neg P(x)) \to \forall x\,(P(x) \wedge \neg P(x))$

(b) $\forall x\,(\neg f(x) < 0) \to f(0) < 0 \to f(0) = 0$

3. Classical logic

(a) $\forall x\,\neg\neg P(x) \to \neg\neg\forall x\,P(x)$

(b) $\neg\forall x\,P(x) \to \exists x \neg P(x)$

(c) $\exists x\,(P(x) \to \forall y\,P(y))$

4. Assume "$P(x)$" means "it is raining in Swansea at day x". Write out 2. (c) as an English sentence (the 100% reliable weather forecast for Swansea).

5. Prove the Satisfiability Theorem from the Completeness Theorem and vice versa.

Part II

Abstract Data Types

5 Algebraic Theory of Abstract Data Types

An Abstract Data Type (ADT) is a collection of objects and functions, that is, an algebra, where one ignores how the objects are constructed and how the functions are implemented. More precisely, if A and B are *isomorphic* algebras, that is, there exists a bijection between A and B that respects the operations, then A and B are regarded as identical. In this chapter we study the algebraic theory of ADTs.

5.1 Homomorphisms and abstract data types

In order to understand what it means for two algebras to be isomorphic, we first study the more general notion of a *homomorphism*, that is, a "structure preserving mapping" between algebras.

5.1.1 Definition

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

$$\varphi_s\colon A_s \to B_s$$

such that

- $\varphi_s(c^A) = c^B$ for each constant $c\colon s$ in Ω,
- $\varphi_s(f^A(a_1, \ldots, a_n)) = f^B(\varphi_{s_1}(a_1), \ldots, \varphi_{s_n}(a_n))$ for each operation $f\colon s_1 \times \ldots \times s_n \to s$ 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*:

$$\begin{array}{ccccccc} A_{s_1} & \times & \cdots & \times & A_{s_n} & \xrightarrow{f^A} & A_s \\ \varphi_{s_1} \downarrow & & \cdots & & \varphi_{s_n} \downarrow & & \varphi_s \downarrow \\ B_{s_1} & \times & \cdots & \times & B_{s_n} & \xrightarrow{f^B} & B_s \end{array}$$

A homomorphism $\varphi\colon A \to B$ is called

monomorphism	if all $\varphi_s\colon A_s \to B_s$ are	**injective**
epimorphism	if all $\varphi_s\colon A_s \to B_s$ are	**surjective**
isomorphism	if all $\varphi_s\colon A_s \to B_s$ are	**bijective**

A homomorphism (isomorphism) from an algebra to itself is called **endomorphism (automorphism)**.

5.1.2 Example

Consider the following signature

Signature	Σ
Sorts	nat
Constants	$0\colon$ nat
Operations	add: nat $\times$ nat $\to$ nat

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

Algebra	A
Carriers	$\mathbf{N}$
Constants	0
Operations	$+\colon \mathbf{N} \times \mathbf{N} \to \mathbf{N}$

For the same signature Σ 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

Algebra	B
Carriers	M
Constants	1
Operations	$*\colon M \times M \to M$

Define

$$\varphi\colon \mathbf{N} \to M \ , \quad \varphi(n) := 2^n$$

We show that φ is an isomorphism from the algebra A to the algebra B (note that φ consists of just one function, since Σ contains only one sort). In order to check that φ is a homomorphism we calculate

- $\varphi(0^{\mathrm{A}}) = \varphi(0) = 1 = 0^{\mathrm{B}}$.
- $\varphi(\mathrm{add}^{\mathrm{A}}(m, n)) = \varphi(m + n) = 2^{m+n} = 2^m * 2^n = \varphi(m) * \varphi(n) = \mathrm{add}^{\mathrm{B}}(\varphi(m), \varphi(n))$.

Since obviously φ is bijective, it is an isomorphism.

5.1.3 Example

Signature	**STACK**
Sorts	elts, stack
Constants	$0\colon$ elts
	emptystack: stack
Operations	push: elts $\times$ stack $\to$ stack
	pop: stack $\to$ stack
	top: stack $\to$ elts

The following is an algebra for the signature **STACK**.

Algebra	SeqN
Carriers	$\mathbf{N}$, $\mathbf{N}^*$ (the set of finite sequences of natural numbers)
Constants	0, $\langle\rangle$ (the empty sequence)
Operations	cons: $\mathbf{N} \times \mathbf{N}^* \to \mathbf{N}^*$ (insert a number in front of a sequence)
	tail: $\mathbf{N}^* \to \mathbf{N}^*$ (remove first element if nonempty, o.w. return $\langle\rangle$)
	head: $\mathbf{N}^* \to \mathbf{N}$ (take first element if nonempty, otherwise return 0)

Consider also the following algebra for the signature **STACK**:

Algebra	Stack0
Carriers	$\{0\}$, $\mathbf{N}$
Constants	0, 0
Operations	$\mathrm{push}^{\mathrm{Stack0}}\colon \{0\} \times \mathbf{N} \to \mathbf{N}$, $\mathrm{push}^{\mathrm{Stack0}}(0, n) := n + 1$
	$\mathrm{pop}^{\mathrm{Stack0}}\colon \mathbf{N} \to \mathbf{N}$, $\mathrm{pop}^{\mathrm{Stack0}}(n + 1) := n$, $\mathrm{pop}^{\mathrm{Stack0}}(0) := 0$
	$\mathrm{top}^{\mathrm{Stack0}}\colon \mathbf{N} \to \{0\}$, $\mathrm{top}^{\mathrm{Stack0}}(n) := 0$

Let us define a homomorphism $\varphi\colon \mathrm{SeqN} \to \mathrm{Stack0}$

Note that φ must be a pair of functions $\varphi = (\varphi_{\mathrm{elts}}, \varphi_{\mathrm{stack}})$ where

$$\varphi_{\mathrm{elts}}\colon \mathrm{SeqN}_{\mathrm{elts}} \to \mathrm{Stack0}_{\mathrm{elts}}\ , \quad \varphi_{\mathrm{stack}}\colon \mathrm{SeqN}_{\mathrm{stack}} \to \mathrm{Stack0}_{\mathrm{stack}}.$$

Since $\mathrm{SeqN}_{\mathrm{elts}} = \mathbf{N}$, $\mathrm{Stack0}_{\mathrm{elts}} = \{0\}$, $\mathrm{SeqN}_{\mathrm{stack}} = \mathbf{N}^*$, and $\mathrm{Stack0}_{\mathrm{stack}} = \mathbf{N}$, this means

$$\varphi_{\mathrm{elts}}\colon \mathbf{N} \to \{0\}\ , \quad \varphi_{\mathrm{stack}}\colon \mathbf{N}^* \to \mathbf{N}.$$

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

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

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

$$\begin{aligned}
\varphi_{\text{stack}}(\text{push}^{\text{SeqN}}(n,\alpha)) &= \varphi_{\text{stack}}(\text{cons}(n,\alpha)) \\
&= \text{length}(\text{cons}(n,\alpha)) \\
&= \text{length}(\alpha)+1 \\
&= \text{push}^{\text{Stack0}}(0,\text{length}(\alpha)) \\
&= \text{push}^{\text{Stack0}}(\varphi_{\text{elts}}(n),\varphi_{\text{stack}}(\alpha))
\end{aligned}$$

Obviously φ_{elts} and φ_{stack} are both surjective, hence φ is an epimorphism. But clearly φ is not a monomorphism.

Remark. This example exhibits a typical feature of epimorphisms: they simplify. In our example φ 'forgets' the natural numbers and replaces them by 0.

5.1.4 Definition

Let $\Sigma = (S,\Omega)$ be a signature and A, B, C Σ-algebras. For homomorphisms $\varphi\colon A \to B$, $\psi\colon B \to C$ its **composition** $\psi \circ \varphi\colon A \to C$ 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 φ is a homomorphism from A to B and ψ 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 φ 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 $\varphi_s^{-1}\colon B_s \to A_s$ is a bijective function for each $s \in S$, it suffices to show that φ^{-1} is a homomorphism. For each constant $c\colon s$ we have

$$\begin{aligned}
\varphi_s^{-1}(c^B) &= \varphi_s^{-1}(\varphi_s(c^A)) \\
&= c^A
\end{aligned}$$

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

$$\varphi_s^{-1}(f^B(b_1,\ldots,b_n)) = f^A(\varphi_{s_1}^{-1}(a_1),\ldots,\varphi_{s_n}^{-1}(a_n)).$$

We have

$$\begin{aligned}
\varphi_s^{-1}(f^B(b_1,\ldots,b_n)) &= \varphi_s^{-1}(f^B(\varphi_{s_1}(\varphi_{s_1}^{-1}(b_1)),\ldots,\varphi_{s_n}(\varphi_{s_n}^{-1}(b_1)))) \\
&= \varphi_s^{-1}(\varphi_s(f^A(\varphi_{s_1}^{-1}(b_1),\ldots,\varphi_{s_n}^{-1}(b_1)))) \\
&= f^A(\varphi_{s_1}^{-1}(b_1),\ldots,\varphi_{s_n}^{-1}(b_1)))
\end{aligned}$$

□

5.1.7 Definition

For two Σ-algebras A and B we set

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

5.1.8 Theorem

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

Proof. Let A, B, C be Σ-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 $\varphi\colon A \to B$. By theorem 5.1.6 $\varphi^{-1}\colon B \to A$ is an isomorphism, hence $B \simeq A$.

(ii) *Transitivity.* Assume $A \simeq B$ and $B \simeq C$, i.e. there are isomorphisms $\varphi\colon A \to B$ and $\psi\colon B \to C$. By theorem 5.1.5 $\psi \circ \varphi\colon A \to C$ 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 Σ we let Alg(Σ) denote the class of all Σ-algebras.

In this definition we had to use the word 'class' instead of 'set', because in general Alg(Σ) 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 $\mathrm{Alg}(\{s\}, \emptyset)$ is a proper class, i.e. not a set.

5.1.10 Definition

An **Abstract Data Type** (**ADT**) for a signature Σ is a *nonempty* class $\mathcal{C} \subseteq \mathrm{Alg}(\Sigma)$ of Σ-algebras which is closed under isomorphisms, i.e.

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

An ADT $\mathcal{C}$ is called **monomorphic** if all its elements are isomorphic, i.e. if

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

Otherwise $\mathcal{C}$ is called **polymorphic**.

5.1.11 Example

Let Σ be a signature.

(a) $\mathrm{Alg}(\Sigma)$ is a polymorphic ADT (a very uninteresting ADT though).

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

Remark. Another way of looking at ADTs is to view them as *abstract properties* of algebras[1]. 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 $\mathbf{N}$ of natural numbers as carrier set. By referring to the concrete set $\mathbf{N}$ the property of being invariant under isomorphism is lost. Hence the class

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

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 should be avoided since they make a software development inflexible and difficult to maintain.

In Chapter 6 we will show that any property described by a formula in the language of a given signature Σ is invariant under homomorphism and hence defines an ADT for Σ.

5.2 The Homomorphism Theorem

We now discuss three fundamental methods of constructing from a given algebra another one 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).

[1]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.

5.2.1 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 Σ is also a sort in Σ', and every operation or constant in Σ is also an operation or constant in Σ'.

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

If Σ is a subsignature of Σ' we also say that Σ' is an **expansion** of Σ.

5.2.2 Definition

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

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

We call B the **Σ-reduct** of A and denote it by $A_{|\Sigma}$.

If B is the Σ-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 $\mathcal{C}$ for a signature Σ' and a subsignature Σ of Σ' we can define the Σ-reduct of $\mathcal{C}$ by

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

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 Σ-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 $c \in \Omega$.
- $f^A(a_1, \dots, a_n) = f^B(a_1, \dots, a_n)$ for all operations $f \in \Omega$ and all $(a_1, \dots, a_n) \in A_{s_1} \times \dots \times A_{s_n}$.

5.2.4 Remarks

1. Obviously, the relation 'A is a subalgebra of B' defines a partial order on the class $\mathrm{Alg}(\Sigma)$ of all Σ-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 c^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 Σ-algebras and let $\varphi\colon A \to B$ be a homomorphism. For each sort $s \in S$ we define the set

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

From the properties of a homomorphism it follows that the sets $\varphi(A_s)$ contain all constants c^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 φ*.

5.2.6 Definition

Let $\Sigma = (S, \Omega)$ be a signature and A a Σ-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 $f\colon s_1 \times \ldots \times s_n \to s$ and any $a_i, b_i \in A_{s_i}$

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

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 Σ-algebra $A/{\sim}$ defined as follows.

- $(A/{\sim})_s := \{[a]_{\sim_s} \mid a \in A_s\}$, for every sort $s \in \Sigma$.
- $c^{A/\sim} := [c^A]_{\sim_s}$, for every constant $c\colon s$.
- $f^{A/\sim}([a_1]_{\sim_{s_1}}, \ldots, [a_n]_{\sim_{s_n}}) := [f^A(a_1, \ldots, a_n)]_{\sim_s}$, for each operation $f\colon s_1 \times \ldots \times s_n \to s$ 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 $[a_i]_{\sim_{s_i}}$.

5.2.7 Example

Consider the signature

Signature	Σ
Sorts	nat
Constants	$0\colon$ nat
Operations	add: nat $\times$ nat $\to$ nat

and the Σ-algebra A of natural numbers with 0 and addition. We define a binary relation $\sim$ on $A's$ carrier $\mathbf{N}$ by

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

Clearly $\sim$ is an equivalence relation (prove this as an exercise). To prove that it is a congruence for A, we have to show that $\sim$ is preserved by the operation $\mathrm{add}^{\mathrm{A}}$, 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 $a_1, a_2, b_1, b_2 \in \mathbf{N}$. We leave the verification of this implication as an exercise.

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

$\mathrm{add}^{\mathrm{A}/\sim}$	EVEN	ODD
EVEN	EVEN	ODD
ODD	ODD	EVEN

5.2.8 Homomorphism Theorem

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

$$a \sim_{\varphi,s} b \quad :\Leftrightarrow \quad \varphi_s(a) = \varphi_s(b).$$

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 φ, i.e.

$$A/\!\sim_\varphi \quad \simeq \quad \varphi(A),$$

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

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

for $s \in S$ and $a \in A_s$.

Proof. The easy proof that $\sim_\varphi$ is a congruence on A is left as an exercise (5.4). 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. $f\colon s_1 \to s$. 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:

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

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 $\varphi\colon A \to B$ naturally induces a congruence $\sim_\varphi$ on A. In fact *every* congruence $\sim$ on A can be obtained in that way, since the mappings $[\cdot]_{\sim_s}\colon A_s \to A_s/{\sim_s}$ obviously constitute a homomorphism $[\cdot]_\sim\colon A \to A/{\sim}$ 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 Σ-algebra and $\mathcal{C}$ a class of Σ-algebras.

A is **initial** for $\mathcal{C}$ if for every $B \in \mathcal{C}$ there exists exactly one homomorphism $\varphi\colon A \to B$.

We say A is **initial in** $\mathcal{C}$ if A is initial for $\mathcal{C}$ and in addition $A \in \mathcal{C}$. We say that A is **initial** if A is initial in $\mathrm{Alg}(\Sigma)$.

Remarks. 1. By replacing in the definition above '$\varphi\colon A \to B$' by '$\varphi\colon B \to A$' we obtain the notion of a *final algebra*.

2. Initiality is a concept coming from *Category Theory* [McL], a mathematical discipline which is pervasive in Computer Science. Also the notions of signature, algebra and homomorphism e.t.c. have category theoretic generalisations. By the category theoretic principle of *dualisation* one obtains form the theory of algebras a theory of *coalgebras* and the proof principle of *coinductioon*. Coagebras and coinduction are important for modelling infinite data (for example infinite streams) as well as interactive programs, processes and games. A good introduction into the category theoretic approach to algebras and coalgebras is given in [JR].

5.3.2 Definition

For any Σ-algebra A and any variable assignment $\alpha\colon X \to A$ the family of functions $\mathrm{eval}^{A,\alpha} = (\mathrm{eval}^{A,\alpha}_s)_{s\in S}$ defined by

$$\mathrm{eval}^{A,\alpha}_s\colon \mathrm{T}(\Sigma, X) \to A \quad , \qquad \mathrm{eval}^{A,\alpha}_s(t) := t^{A,\alpha}$$

is a homomorphism $\mathrm{eval}^{A,\alpha}\colon \mathrm{T}(\Sigma, X) \to A$ which is called **evaluation homomorphism**.

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

5.3.3 Definition

A Σ-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 $t \in \mathrm{T}(\Sigma)$ such that $t^A = a$. Note that this is equivalent to saying that $\mathrm{eval}\colon \mathrm{T}(\Sigma) \to A$ is an epimorphism (isomorphism).

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

5.3.4 Examples

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

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

3. Let Σ and A be the signature and algebra of the examples 5.1.2 and 5.2.7 with 0 and addition. The Σ-algebra A is not term generated, since 0 is the only natural number which is the value of a closed Σ-term.

4. Consider the following signature.

Signature	**BOOLE**
Sorts	boole
Constants	$\mathrm{T}\colon$ boole $\mathrm{F}\colon$ boole
Operations	$\mathrm{not}\colon$ boole $\to$ boole $\mathrm{and}\colon$ boole $\times$ boole $\to$ boole $\mathrm{or}\colon$ boole $\times$ boole $\to$ boole

and the following **BOOLE**-algebra

Algebra	Boole
Carriers	$\mathbf{B} := \{\#\mathrm{t}, \#\mathrm{f}\}$
Constants	#t, #f
Operations	$\neg\colon \mathbf{B} \to \mathbf{B}$ (negation)
	$\wedge\colon \mathbf{B} \times \mathbf{B} \to \mathbf{B}$ (conjunction)
	$\vee\colon \mathbf{B} \times \mathbf{B} \to \mathbf{B}$ (disjunction)

The algebra Boole of boolean values is generated, since, for example, $\mathrm{T}^{\mathrm{Boole}} = \#\mathrm{t}$ and $\mathrm{F}^{\mathrm{Boole}} = \#\mathrm{f}$. However, Boole is not freely generated since, for example, $\#\mathrm{t} = \mathrm{T}^{\mathrm{Boole}} = \mathrm{not}(\mathrm{T})^{\mathrm{Boole}}$.

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

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

5.3.5 Theorem

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

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

Base.

$$\begin{aligned} \varphi(c) &= \varphi(c^{\mathrm{T}(\Sigma)}) \\ &= c^A \qquad \text{since } \varphi \text{ is a homomorphism} \\ &= \mathrm{eval}^A(c). \end{aligned}$$

Step.

$$\begin{aligned} \varphi(f(t_1, \ldots, t_n)) &= \varphi(f^{\mathrm{T}(\Sigma)}(t_1, \ldots, t_n)) \\ &= f^A(\varphi(t_1), \ldots, \varphi(t_n)) \qquad \text{since } \varphi \text{ is a homomorphism} \\ &= f^A(\mathrm{eval}^A(t_1), \ldots, \mathrm{eval}^A(t_n)) \qquad \text{by induction hypothesis} \\ &= \mathrm{eval}^A(f(t_1, \ldots, t_n)) \qquad \text{by definition of } \mathrm{eval}^A \end{aligned}$$

5.3.6 Theorem

For a Σ-algebra A the following statements are equivalent.

(i) A is initial.

(ii) A is freely generated.

(iii) $A \simeq \mathrm{T}(\Sigma)$.

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

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

(ii)⇒(iii) If A is freely generated then $\mathrm{eval}^A\colon \mathrm{T}(\Sigma) \to A$ is an isomorphism. Hence $A \simeq \mathrm{T}(\Sigma)$.

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

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

For example let $\mathcal{C}$ 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(**BOOLE**) does certainly not belong to $\mathcal{C}$ because e.g. the law $\mathrm{not}(\mathrm{T}) = \mathrm{F}$ does not hold in T(**BOOLE**) (the terms not(T) and F are not equal).

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

5.3.7 Theorem

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

$$t_1 \sim_{\mathcal{C},s} t_2 \quad :\Longleftrightarrow \quad \text{for all } A \in \mathcal{C}\ t_1^A = t_2^A$$

Then $\sim_{\mathcal{C}} := (\sim_{\mathcal{C},s})_{s \in S}$ is a congruence on $\mathrm{T}(\Sigma)$ and the quotient algebra

$$\mathrm{T}_{\mathcal{C}}(\Sigma) \quad := \quad \mathrm{T}(\Sigma)/{\sim_{\mathcal{C}}}$$

is initial for $\mathcal{C}$. For every $A \in \mathcal{C}$ the unique homomorphism from $\varphi\colon \mathrm{T}_{\mathcal{C}}(\Sigma) \to A$ is given by

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

for each sort s and each $t \in \mathrm{T}(\Sigma)$ of sort s.

Proof. By definition $\sim_{\mathcal{C}}$ is the intersection of the congruences $\sim_{\mathrm{eval}^A}$ $(A \in \mathcal{C})$ (cf. the Homomorphism Theorem 5.2.8). Since congruences are closed under intersections (the easy proof is left as an exercise) it follows that $\sim_{\mathcal{C}}$ is a congruence on $\mathrm{T}(\Sigma)$. It is easy to see that φ above is a well-defined homomorphism. The proof that φ is unique is similar to the proof of theorem 5.3.5 and is left as an exercise. Hence $\mathrm{T}_{\mathcal{C}}(\Sigma)$ is initial for $\mathcal{C}$.

5.4 Summary and Exercises

- Homomorphisms, epi-, mono-, isomorphisms.
- Homomorphic and polymorphic Abstract data types.
- Congruence, quotient algebra.
- Initial algebras, uniqueness up to isomorphism of initial algebras.
- Term generated and freely generated algebras.

Exercises.

1. Consider again the Σ-algebra of natural numbers with 0 and addition from exercise 5.2.7. Show that a function $\varphi\colon \mathbf{N} \to \mathbf{N}$ is an endomorphism on A if and only if there is a number $k \in \mathbf{N}$ such that $\varphi(n) = k * n$ for all $n \in \mathbf{N}$. How are the automorphisms on A characterised?

2. Extend the signature of Example 1 by a unary successor operation and let B be the extension of A by the usual successor function on $\mathbf{N}$. Show that the only homomorphism on B is the identity.

3. Let Σ be the signature of Example 1. Consider the Σ-algebra C of finite lists of natural numbers where 0 is interpreted by the empty list and the binary operation is interpreted by concatenation of lists. Is the operation of reversing a list a homomorphism on C?

4. Define an epimorphism from C to A and a monomorphism from A to C.

5. Which of the algebras A, B and C are generated respectively freely generated?

6. Prove that homomorphisms are closed under composition.

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

$$a \sim_{\varphi,s} b \quad :\Leftrightarrow \quad \varphi_s(a) = \varphi_s(b).$$

Show that the family $\sim_\varphi := (\sim_{\varphi,s})_{s\in S}$ is a congruence on A (see Theorem 5.2.8).

6 Specification of Abstract Data Types

We now study *formal specifications* of abstract data types, also called *algebraic specifications*. First, we will consider arbitrary *first-order specification*, but will later concentrate on *equational specifications* which, as we will see in Chapter 8, can be used to automatically generate provably correct "prototypes" of programs.

6.1 Loose specifications

6.1.1 Definition

A **loose specification** is a pair (Σ, Φ) where Σ is a signature and Φ is a set of closed Σ-formulas. The formulas in Φ are called the **axioms** of the specification.

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

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

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

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

6.1.2 Lemma

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

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

Proof. Structural induction on t.

Base: variables.

$$\begin{aligned} \varphi(x^{A,\alpha}) &= \varphi(\alpha(x)) \\ &= (\varphi \circ \alpha)(x) \\ &= x^{B,\varphi\circ\alpha} \end{aligned}$$

Base: constants.

$$\begin{aligned} \varphi(c^{A,\alpha}) &= \varphi(c^{A}) \\ &= \varphi(c^{B}) \\ &= c^{B,\varphi\circ\alpha} \end{aligned}$$

Step.

$$
\begin{aligned}
\varphi(f(t_1,\ldots,t_n)^{A,\alpha}) &= \varphi(f^A(t_1^{A,\alpha},\ldots,t_n^{A,\alpha})) \\
&= f^B(\varphi(t_1^{A,\alpha}),\ldots,\varphi(t_n^{A,\alpha})) \\
&= f^B(t_1^{B,\varphi\circ\alpha},\ldots,t_n^{B,\varphi\circ\alpha}) \qquad \text{(by i.h.)} \\
&= f(t_1,\ldots,t_n)^{B,\varphi\circ\alpha}
\end{aligned}
$$

6.1.3 Theorem

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

$$A, \alpha \models P \qquad \text{iff} \qquad B, \varphi\circ\alpha \models P$$

In particular when P is closed we have

$$A \models P \qquad \text{iff} \qquad B \models P$$

Proof. Structural induction on the formula P.

(i) *Base.*

$$
\begin{aligned}
A, \alpha \models t_1 = t_2 \quad &\text{iff} \quad t_1^{A,\alpha} = t_2^{A,\alpha} \\
&\text{iff} \quad \varphi(t_1^{A,\alpha}) = \varphi(t_2^{A,\alpha}) \qquad (\varphi \text{ is injective}) \\
&\text{iff} \quad t_1^{B,\varphi\circ\alpha} = t_2^{B,\varphi\circ\alpha} \qquad \text{(Lemma 6.1.2)} \\
&\text{iff} \quad B, \varphi\circ\alpha \models t_1 = t_2
\end{aligned}
$$

(ii) *Step: propositional connectives.*

$$
\begin{aligned}
A, \alpha \models P \wedge Q \quad &\text{iff} \quad A, \alpha \models P \text{ and } A, \alpha \models Q \\
&\text{iff} \quad B, \varphi\circ\alpha \models P \text{ and } B, \varphi\circ\alpha \models Q \qquad \text{(i.h.)} \\
&\text{iff} \quad B, \varphi\circ\alpha \models P \wedge Q
\end{aligned}
$$

$$P \vee Q,\ P \to Q \qquad \text{similar}$$

$$
\begin{aligned}
A, \alpha \models \neg P \quad &\text{iff} \quad A, \alpha \not\models P \\
&\text{iff} \quad B, \varphi\circ\alpha \not\models P \qquad \text{(i.h.)} \\
&\text{iff} \quad B, \varphi\circ\alpha \models \neg P
\end{aligned}
$$

(iii) *Step: quantifiers.*

$$\begin{array}{lll} A,\alpha \models \forall x\, P & \text{iff} & A,\alpha^a_x \models P \text{ for all } a \in A_s \\ & \text{iff} & B,\varphi\circ(\alpha^a_x) \models P \text{ for all } a \in A_s \qquad \text{(i.h.)} \\ & \text{iff} & B,(\varphi\circ\alpha)^{\varphi(a)}_x \models P \text{ for all } a \in A_s \qquad (\varphi\circ(\alpha^a_x) = (\varphi\circ\alpha)^{\varphi(a)}_x) \\ & \text{iff} & B,(\varphi\circ\alpha)^b_x \models P \text{ for all } b \in B_s \qquad (\varphi \text{ is surjective}) \\ & \text{iff} & B,\varphi\circ\alpha \models \forall x\, P \end{array}$$

$$\begin{array}{lll} A,\alpha \models \exists x\, P & \text{iff} & A,\alpha^a_x \models P \text{ for at least one } a \in A_s \\ & \text{iff} & B,\varphi\circ(\alpha^a_x) \models P \text{ for at least one } a \in A_s \qquad \text{(i.h.)} \\ & \text{iff} & B,(\varphi\circ\alpha)^{\varphi(a)}_x \models P \text{ for at least one } a \in A_s \qquad (\varphi\circ(\alpha^a_x) = (\varphi\circ\alpha)^{\varphi(a)}_x) \\ & \text{iff} & B,(\varphi\circ\alpha)^b_x \models P \text{ for at least one } b \in B_s \qquad (\varphi \text{ is surjective}) \\ & \text{iff} & B,\varphi\circ\alpha \models \exists x\, P \end{array}$$

6.1.4 Theorem

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

Proof. Let A, B be Σ-algebras such that $A \in \mathrm{Mod}_\Sigma(\Phi)$ and $A \simeq B$. We have to show $B \in \mathrm{Mod}_\Sigma(\Phi)$. Since $A \in \mathrm{Mod}_\Sigma(\Phi)$ 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. $B \in \mathrm{Mod}_\Sigma(\Phi)$.

6.1.5 Example

Consider the loose specification (Σ, Φ), where

Signature	Σ
Sorts	boole
Constants	$\mathrm{T}, \mathrm{F} \colon \mathsf{boole}$
Operations	$\mathrm{not} \colon \mathsf{boole} \to \mathsf{boole}$
	$\mathrm{and}, \mathrm{or} \colon \mathsf{boole} \times \mathsf{boole} \to \mathsf{boole}$

and $\Phi = \{P_1, \ldots, P_6\}$, with

$$P_1 \quad :\equiv \quad \mathrm{not}(\mathrm{T}) = \mathrm{F}$$

$$P_2 \quad :\equiv \quad \mathrm{not}(\mathrm{F}) = \mathrm{T}$$

$$P_3 \quad :\equiv \quad \mathrm{and}(\mathrm{T},\mathrm{T}) = \mathrm{T}$$

$$P_4 \quad :\equiv \quad \forall x\,(\mathrm{and}(\mathrm{F}, x) = \mathrm{F})$$

$$P_5 \quad :\equiv \quad \forall x\,(\mathrm{and}(x, \mathrm{F}) = \mathrm{F})$$

$$P_6 \quad :\equiv \quad \forall x, y\,(\mathrm{or}(x, y) = \mathrm{not}(\mathrm{and}(\mathrm{not}(x), \mathrm{not}(y))))$$

Consider the following Σ-algebras

Algebra	Boole
Carriers	$\mathbf{B} := \{\#\mathrm{t}, \#\mathrm{f}\}$
Constants	#t, #f
Operations	$\neg\colon \mathbf{B} \to \mathbf{B}$ (negation)
	$\wedge\colon \mathbf{B} \times \mathbf{B} \to \mathbf{B}$ (conjunction)
	$\vee\colon \mathbf{B} \times \mathbf{B} \to \mathbf{B}$ (disjunction)

Algebra	Pow(**N**)
Carriers	$\mathcal{P}(\mathbf{N}) := \{A \mid A \subseteq \mathbf{N}\}$, the powerset of **N**
Constants	$\mathbf{N}$, $\emptyset$
Operations	$\mathbf{N} \setminus \cdot\colon \mathcal{P}(\mathbf{N}) \to \mathcal{P}(\mathbf{N})$ (complement)
	$\cap\colon \mathcal{P}(\mathbf{N}) \times \mathcal{P}(\mathbf{N}) \to \mathcal{P}(\mathbf{N})$ (intersection)
	$\cup\colon \mathcal{P}(\mathbf{N}) \times \mathcal{P}(\mathbf{N}) \to \mathcal{P}(\mathbf{N})$ (union)

which are clearly models of the loose specification (Σ, Φ), that is

$$\mathrm{Boole} \models \Phi \quad \text{and} \quad \mathrm{Pow(N)} \models \Phi$$

or

$$\mathrm{Boole}, \mathrm{Pow(N)} \in \mathrm{Mod}_\Sigma(\Phi)$$

The ADT $\mathrm{Mod}_\Sigma(\Phi)$ is polymorphic since it contains the non-isomorphic algebras Boole and 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 Pow(N)).

$$P_7 \quad :\equiv$$

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

$$P_8 \quad :\equiv$$

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

Signature	Σ_{0S+}
Sorts	nat
Constants	$0\colon \mathsf{nat}$
Operations	$\mathsf{succ}\colon \mathsf{nat} \to \mathsf{nat}$
	$+\colon \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$

As the names suggest the intended algebra for this signature is the algebra N_{0S+} of natural numbers with the constant 0, the usual successor function $\mathsf{succ}\colon \mathbf{N} \to \mathbf{N}$, $\mathsf{succ}(n) := n+1$, and addition $+\colon \mathbf{N} \times \mathbf{N} \to \mathbf{N}$.

Let us try to characterise N_{0S+} up to isomorphism by a loose specification (Σ_{0S+}, Φ) with a suitable set of axioms Φ. Let us put into Φ formulas $P_1, \ldots, P_5$ expressing that

1. 0 is not a successor,
2. succ is injective (one-to-one)
3. every number is either 0 or a successor,

4/5. addition can be defined from 0 and succ by primitive recursion in the usual way.

$$P_1 \quad :\equiv \quad \forall x\, (0 \neq \mathsf{succ}(x))$$

$$P_2 \quad :\equiv \quad \forall x, y\, (\mathsf{succ}(x) = \mathsf{succ}(y) \to x = y)$$

$$P_3 \quad :\equiv \quad \forall x\, (x = 0 \vee \exists y\, (x = \mathsf{succ}(y)))$$

$$P_4 \quad :\equiv \quad \forall x\, (x + 0 = x)$$

$$P_5 \quad :\equiv \quad \forall x, y\, (x + \mathsf{succ}(y) = \mathsf{succ}(x + y))$$

Clearly the algebra N_{0S+} is a model of $\{P_1, \ldots, P_5\}$.

But there are still unwanted models. For example the Σ_{0S+}-algebra A with $A_{\mathsf{nat}} := \{0\} \times \mathbf{N} \cup \{1\} \times \mathbf{Z}$ with $0^A := (0,0)$, $\mathsf{succ}^A((i,n)) := (i, n+1)$, 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

$$P_6 \quad :\equiv \quad \forall x\, (x + x = x \to x = 0)$$

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 (Σ, Φ) 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 $\mathrm{Mod}_\Sigma(\Phi)$ 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.

6.1.8 Example

The loose specification $(\Sigma, \{P_1, \ldots, P_5\})$ of Example 6.1.6 is displayed as follows.

Loose Spec	
Sorts	nat
Constants	$0 \colon \mathsf{nat}$
Operations	$\mathsf{succ} \colon \mathsf{nat} \to \mathsf{nat}$ $+ \colon \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$
Variables	$x, y \colon \mathsf{nat}$
Axioms	$0 \neq \mathsf{succ}(x)$ $\mathsf{succ}(x) = \mathsf{succ}(y) \to x = y$ $x = 0 \lor \exists y\, (x = \mathsf{succ}(y))$ $x + 0 = x$ $x + \mathsf{succ}(y) = \mathsf{succ}(x + y)$

6.1.9 Definition

A loose specification (Σ, Φ) is called **adequate** for a Σ-algebra A if $A \in \mathrm{Mod}_\Sigma(\Phi)$.

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

$$B \in \mathrm{Mod}_\Sigma(A) \qquad \text{iff} \qquad B \simeq A.$$

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 N_{0S+}, but not strictly adequate.

6.1.10 Definition

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

(Σ', Φ') is called **persistent extension** of (Σ, Φ) if (Σ', Φ') is an extension of (Σ, Φ) and addition for every closed Σ-formula P it holds that $\Phi' \models P$ if and only of $\Phi \models P$.

Remark. Persistence is an important property of an extension of a specification: It says that the old operations are not affected by the extension, that is, no new facts about the old operations follow from the new axioms.

6.1.11 Lemma

Let the loose specification (Σ', Φ') be an extension of the loose specification (Σ, Φ) such that very Σ-algebra A satisfying Φ can be expanded to a Σ'-algebra A' satisfying Φ'.

Then (Σ', Φ') is a persistent extension of (Σ, Φ).

Proof. Let P be a closed Σ-formula such that $\Phi' \models P$. We have to show $\Phi \models P$. To this end we take an arbitrary Σ-algebra A satisfying Φ and have to show that A satisfies P. By assumption there is an expansion A' of A such that A' satisfies Φ'. 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 (Exercise 6 at the end of this Chapter).

6.1.12 Example

Let (Σ, Φ) be the specification in Example 6.1.8. Extend (Σ, Φ) by the operation

$$\mathrm{pred}\colon \mathsf{nat} \to \mathsf{nat}$$

and the equation

$$\mathrm{pred}(\mathsf{succ}(x)) = x$$

to obtain (Σ', Φ'). We use Lemma 6.1.11 to show that (Σ', Φ') is persistent: Let A be any model of (Σ, Φ). Thanks to the axiom $\mathsf{succ}(x) = \mathsf{succ}(y) \to x = y$ the function succ^A is injective. Hence we can define $\mathrm{pred}^A(a) := b$ if $\mathsf{succ}^A(b) = a$ (the b is unique if it exists) and $\mathrm{pred}^A(a) := a$ otherwise. This defines an expansion of A which is a model of (Σ', Φ').

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

Define two Σ-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 $a = b \vee a = c$ is true, i.e. $B, C \in \mathrm{Mod}_\Sigma(\Phi)$. Since we assumed A to be initial for $\mathrm{Mod}_\Sigma(\Phi)$, we have homomorphisms

$$\varphi\colon A \to B, \quad \psi\colon A \to C$$

Using the homomorphic property of φ and ψ we see

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

$$\psi(a^A) = a^C \neq b^C = \varphi(b^A), \text{ hence } a^A \neq b^A$$

Hence the formula $a = b \vee a = c$ 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 Σ is a formulas of the form

$$t_1 = t_2$$

where t_1, t_2 are Σ-terms of the same sort.

If E is a set of equations over Σ 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 Σ. We define

$$\mathrm{T}_E(\Sigma) := \mathrm{T}_{\mathrm{Mod}_\Sigma(\forall E\,)}(\Sigma)$$

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

$$t_1 \sim_E t_2 \quad \Leftrightarrow \quad \forall E \models t_1 = t_2$$

Hence the elements of $\mathrm{T}_E(\Sigma)$ 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 Σ. Then the Σ-algebra $\mathrm{T}_E(\Sigma)$ is initial in $\mathrm{Mod}_\Sigma(\forall E\,)$.

For every $A \in \mathrm{Mod}_\Sigma(\forall E\,)$ the unique homomorphism $\varphi_A \colon \mathrm{T}_E(\Sigma) \to A$ is given by

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

for each $t \in \mathrm{T}(\Sigma)$.

Proof. In theorem 6.2.3 it was proved that $\mathrm{T}_E(\Sigma)$ is initial for $\mathrm{Mod}_\Sigma(\forall E\,)$, and that φ_A is the unique homomorphism from $\mathrm{T}_E(\Sigma)$ to A. Hence it only remains to show that $\mathrm{T}_E(\Sigma)$ 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 $\mathrm{T}_E(\Sigma)$.

In preparation of proving this we first show

$$t_1\theta \sim_E t_2\theta \quad \text{for all substitutions } \theta \colon X \to \mathrm{T}(\Sigma) \tag{+}$$

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

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:

$$(t_1\theta)^A \quad \overset{3.5.4}{=} \quad t_1^{A,\theta^A} \quad \overset{A\models\forall(t_1=t_2)}{=} \quad t_2^{A,\theta^A} \quad \overset{3.5.4}{=} \quad (t_1\theta)^A$$

Having proved (+) it is now easy to prove that the formula $\forall(t_1 = t_2)$ is true in $\mathrm{T}_E(\Sigma)$. Let $\alpha \colon X \to \mathrm{T}_E(\Sigma)$ be a variable assignment. We have to prove

$$t_1^{\mathrm{T}_E(\Sigma),\alpha} = t_2^{\mathrm{T}_E(\Sigma),\alpha} \tag{++}$$

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 $\theta\colon X \to \mathrm{T}(\Sigma)$. 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 $[\cdot]_{\sim_E}\colon \mathrm{T}(\Sigma) \to \mathrm{T}_E(\Sigma)$ $(=\mathrm{T}(\Sigma)/{\sim_E})$ is a homomorphism. Finally note that the substitution θ can also be viewed as a variable assignment for the closed term algebra $\mathrm{T}(\Sigma)$. Baring all this in mind we can now prove (++). We have

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

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

6.2.4 Definition

Let Σ be a signature and E a set of equations over Σ. Then

$$\text{Init–Spec}(\Sigma, E)$$

is called an **initial specification**.

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

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

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

6.2.5 Theorem

Let $\mathrm{T}_E(\Sigma)$ be an initial specification. Then for any Σ-algebra A the following conditions are equivalent:

(i) A is a model of Init–Spec(Σ, E).

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

(iii) $A \simeq \mathrm{T}_E(\Sigma)$.

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

$$A \models t_1 = t_2 \quad \Leftrightarrow \quad \forall E\, \models t_1 = t_2$$

(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 Σ-terms t_1, t_2 of the same sort we have

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

In particular Init–Mod$_\Sigma(E)$ is a monomorphic ADT containing $\mathrm{T}_E(\Sigma)$.

Proof. '(i)⇔(ii)' is just a repetition of definition 6.2.4 above.

'(ii)⇔(iii)' follows from theorem 6.2.3 and the fact that initial algebras are unique up to isomorphism (coursework 2).

'(iii)⇒(iv)'. Let $\varphi\colon \mathrm{T}_E(\Sigma) \to A$ be an isomorphism. By lemma 6.1.2 and the fact that $t^{\mathrm{T}_E(\Sigma)} = [t]$ we have $t^A = \varphi([t])$ for all closed Σ-terms. This clearly implies (iv).

'(iv)⇒(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 $\alpha\colon X \to A$ a variable assignment, where $X := \mathsf{FV}(t_1 = t_2)$. 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 3.5.6 we have

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

and similarly $t_1^{A,\alpha} = (t_1\theta)^A$. Since $t_1 = t_2$ is an equation in E we have $\forall E \models t_1\theta = t_2\theta$ (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)⇒(iii)'. Assume that (v) holds. Since by assumption $A \in \mathrm{Mod}_\Sigma(\forall E)$ we now by initiality of $\mathrm{T}_E(\Sigma)$ that there is unique homomorphism $\varphi\colon \mathrm{T}_E(\Sigma) \to A$. Using once more the fact that $t^A = \varphi([t])$ for all closed Σ-terms (see '(iii)⇒(iv)' above) it is plain that our assumption (v) implies that φ is bijective.

The following theorem characterises equality between *open* terms in initial models.

6.2.6 Theorem

Let A be a model of the initial specification Init–Spec(Σ, 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 $\theta\colon X \to \mathrm{T}(\Sigma)$, where $X := \mathsf{FV}(t_1 = t_2)$.

Proof. '(i)⇒(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 $\varphi\colon A \to B$. Since A and B are both generated φ is surjective. Hence clearly $B \models \forall(t_1 = t_2)$.

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

‘(i)⇔(iii)’. Since A is generated $A \models \forall(t_1 = t_2)$ is equivalent to $A \models t_1\theta = t_2\theta$ for all substitutions $\theta\colon X \to \mathrm{T}(\Sigma)$, and by theorem 6.2.5 (iv) the latter is equivalent to (iii).

6.2.7 Example

Let us give an adequate initial specification of the algebra Boole defined in example 6.1.5.

Init Spec	BOOLE
Sorts	boole
Constants	$\mathsf{T}, \mathsf{F}\colon \mathsf{boole}$
Operations	$\neg\colon \mathsf{boole} \to \mathsf{boole}$ $\mathsf{and}, \mathsf{or}\colon \mathsf{boole} \times \mathsf{boole} \to \mathsf{boole}$
Variables	$x, y\colon \mathsf{boole}$
Equations	$\neg\mathsf{T} = \mathsf{F}$ $\neg\mathsf{F} = \mathsf{T}$ $\mathsf{and}(\mathsf{T}, \mathsf{T}) = \mathsf{T}$ $\mathsf{and}(\mathsf{F}, x) = \mathsf{F}$ $\mathsf{and}(x, \mathsf{F}) = \mathsf{F}$ $\mathsf{or}(x, y) = \neg(\neg x, \neg y)$

Note that it is no longer necessary to specify that T and F are different and the only elements of the carrier set.

How do we show that this specification is adequate? We use Theorem 6.2.5 (v). Clearly, Boole is a generated model of $\forall E$ where E is the set of six equations of our specification above. Furthermore, by induction on closed terms t, we show:

(*) If $t^{\mathrm{Boole}} = b$ where $b \in \{\mathsf{T}, \mathsf{F}\}$, then $\forall E \models t = b$.

Now, if for two closed terms t_1, t_2 of the same sort we have $A \models t_1 = t_2$, say, $t_i^{\mathrm{Boole}} = \mathsf{T}$ for $i = 1, 2$, then, by (*), it follows $\forall E \models t_1 = \mathsf{T} = t_2$. Hence, by Theorem 6.2.5 “(v)⇒(i)” it follows that that the specification is adequate.

6.2.8 Example

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

We display the initial specification Init–Spec(Σ, E) by

Init Spec	NAT
Sorts	nat
Constants	$0\colon$ nat
Operations	succ: nat $\to$ nat $+\colon$ nat $\times$ nat $\to$ nat
Variables	$x, y\colon$ nat
Equations	$x + 0 = x$ $x + \mathsf{succ}(y) = \mathsf{succ}(x + y)$

We can show that this specification is adequate for the standard algebra $\mathrm{N}_{0\mathrm{S}+}$ of natural numbers with 0 and addition (see Example 5.2.7), with a similar method as in Example 6.1.5: It suffices to show that if a closed term t has value n in the standard algebra, then $\forall E \models t = \mathsf{succ}^n(0)$. Again, this can easily proven by induction on t.

From this, it also follows that the elements of the carrier set of $\mathrm{T}_E(\Sigma)$ are the equivalence classes $[0]$, $[\mathsf{succ}(0)]$, $[\mathsf{succ}(\mathsf{succ}(0))]$, ... One has for instance

$$\begin{aligned}[0] &= \{0,\ 0+0,\ (0+0)+0, \ldots\} \\ &= \text{the set of closed terms built from 0 and +}\end{aligned}$$

$$\begin{aligned}[\mathsf{succ}(0)] &= \{\mathsf{succ}(0),\ \mathsf{succ}(0)+0,\ 0+\mathsf{succ}(0), \ldots\} \\ &= \text{the set of closed terms built from 0 and + and exactly one occurrence of } \mathsf{succ}\end{aligned}$$

and in general

$$\begin{aligned}[n] &= \text{the set of closed terms built from 0 and + and exactly } n\text{-times occurrences of } \mathsf{succ} \\ &= \text{the set of closed terms } t \text{ such that } t^A = n\end{aligned}$$

In the next chapter we will develop tools that will facilitate similar proofs for a large class of initial specifications.

The constant 0 and the f operation succ form a system of generators for $\mathrm{N}_{0\mathrm{S}+}$, since all elements are generated by terms built from 0 and succ. NAT is freely generated by 0 and succ, because different terms built from 0 and succ denote different numbers.

6.2.9 Example

Let us modify example 6.2.8 as follows. We extend the initial specification by an operation $-\colon$ nat $\times$ nat $\to$ nat and add the two equations

$x - 0 = x$

$\mathsf{succ}(x) - \mathsf{succ}(y) = x - y$

Let E be this extended set of equations. We also expand the standard algebra of natural numbers in 6.2.8 by interpreting the new operation, $-$, by the operation $\dot{-} : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$

$$n \dot{-} m \quad := \quad \begin{cases} n - m & \text{if } n \geq m \\ 0 & \text{otherwise} \end{cases}$$

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 Init–Spec(Σ, E), since, for example in A the equation

$0 - \mathsf{succ}(0) = 0$

holds, whereas clearly

$\forall E \not\models 0 - \mathsf{succ}(0) = 0$

Therefore 6.2.5 (v) does not hold.

6.2.10 Exercises

Let the set of equations E and the algebra A be as in example 6.2.9.

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

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

(c) Give an informal description of 'the' model of the initial specification Init–Spec(Σ, E').

6.2.11 Example

Init Spec	NATSET
Sorts	boole, nat, set
Constants	T: boole F: boole 0: nat emptyset: set
Operations	succ: nat $\longrightarrow$ nat isempty: set $\longrightarrow$ boole insert: set $\times$ nat $\longrightarrow$ set
Variables	x, y: nat, s: set
Equations	$\mathsf{insert}(\mathsf{insert}(s, x), x) = \mathsf{insert}(s, x)$ $\mathsf{insert}(\mathsf{insert}(s, x), y) = \mathsf{insert}(\mathsf{insert}(s, y), x)$ $\mathsf{isempty}(\mathsf{emptyset}) = \mathsf{T}$ $\mathsf{isempty}(\mathsf{insert}(s, x)) = \mathsf{F}$

Let A be the classical algebra of finite set of natural numbers with the obvious interpretation of the constants and operation. In order to show that NATSET is an adequate specification of A we use again Theorem 6.2.5. Clearly A is a generated model of the equations E of NATSET. By induction closed terms t one can also show that $\forall E \models t = t'$ where

$$t' = \mathsf{insert}(...\mathsf{insert}(\mathsf{emptyset}, \mathsf{succ}^{n_1}(0)), \ldots \mathsf{succ}^{n_k}(0))$$

where $t^A = \{n_1, \ldots, n_k\}$ with $n_1 < \ldots < n_k$. With a similar argument as in the previous examples it follows that if $A \models t_1 = t_2$, then $\forall E \models t_1 = t_2$.

6.2.12 Example

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

- $\mathsf{write}(x)$: insert the character x immediately to the left of the cursor;
- $\rhd$: move the cursor one position to the right;
- $\lhd$: move the cursor one position to the left;
- del: delete the character immediately to the right of the cursor.

Consider for example the file

$edir|or$

where the '|' represents the cursor. After entering $\triangleleft$ we get

$edi|ror$

and entering del thereafter yields

$edi|or$

Finally we write the character t and obtain

$edit|or$

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:

$([\mathsf{r}, \mathsf{i}, \mathsf{d}, \mathsf{e}], [\mathsf{o}, \mathsf{r}])$

$([\mathsf{i}, \mathsf{d}, \mathsf{e}], [\mathsf{r}, \mathsf{o}, \mathsf{r}])$

$([\mathsf{i}, \mathsf{d}, \mathsf{e}], [\mathsf{o}, \mathsf{r}])$

$([\mathsf{t}, \mathsf{i}, \mathsf{d}, \mathsf{e}], [\mathsf{o}, \mathsf{r}])$

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 implement all possible actions of the editor.

To create a file we will use the generator $\mathsf{cf}\colon \mathsf{charlist} \times \mathsf{charlist} \to \mathsf{file}$ and for creating lists of characters the usual generators $\mathsf{nil}\colon \mathsf{charlist}$ and $\mathsf{cons}\colon \mathsf{char} \times \mathsf{charlist} \to \mathsf{charlist}$.

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 formalises our ideas:

Init Spec	EDITOR
Sorts	char, charlist, file
Constants	newfile: file $a, \ldots, z, _$: char nil: charlist
Operations	cons: char $\times$ charlist $\longrightarrow$ charlist cf: charlist $\times$ charlist $\longrightarrow$ file write: char $\times$ file $\longrightarrow$ file $\triangleleft$: file $\longrightarrow$ file $\triangleright$: file $\longrightarrow$ file del: file $\longrightarrow$ file
Variables	x: char, l, r: charlist
Equations	newfile = cf(nil, nil) write(x, cf(l, r)) = cf(cons(x, l), r) $\triangleleft$ (cf(nil, r)) = cf(nil, r) $\triangleleft$ (cf(cons(x, l), r)) = cf(l, cons(x, r)) $\triangleright$ (cf(l, nil)) = cf(l, nil) $\triangleright$ (cf(l, cons(x, r))) = cf(cons(x, l), r) del(cf(l, nil)) = cf(l, nil) del(cf(l, cons(x, r))) = cf(l, r)

6.2.13 Exercises

(a) Extend the editor specified in example 6.2.12 by a command backspace that deletes the character immediately to the left of the cursor.d

(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.

6.3 Exception handling

In section 5.1 we considered the algebra SeqN of finite sequences of natural numbers. We defined that the head and tail of an empty list are respectively zero and the empty list. It

would however been more natural to raise an exception in this situation, reflecting the fact that the operations head and tail should not be performed on an empty list.

There are essentially four ways to handle exceptions (see [LEW]):

(i) *Loose specification*: We do not specify what the result in an exceptional situation is. This forces us to accept polymorphic abstract data types as models of data.

(ii) *Partial algebras*: We interpret operations like head and tail as *partial functions*. This means one has to develop a logic (syntax and semantics) for partial algebras.

(iii) *Subsorts*: One introduces, for example, the subsort of nonempty lists, and defines head and tail on this subsort only. Algebras with subsorts are also called *order-sorted algebras*.

(iv) *Algebras with error elements*: One requires that every carrier set contains a special element called error, and defines all operations such that they propagate errors. For example $\mathsf{error} + n = n + \mathsf{error} = \mathsf{error}$ for all $n \in \mathbf{N} \cup \{\mathsf{error}\}$.

All four approaches have advantages and disadvantages. In this course we will pursue approach no. (iv) because it can be easily combined with the initial algebra semantics. As an example we consider the specification NAT of example 6.2.8 extended by subtraction, such that $0 - \mathsf{succ}(m)$ raises an exception.

Init Spec	NAT
Sorts	nat
Constants	$0\colon \mathsf{nat}$
Operations	$\mathsf{succ}\colon \mathsf{nat} \to \mathsf{nat}$ $+\colon \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$ $-\colon \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$
Variables	$x, y\colon \mathsf{nat}$
Equations	$x + 0 = x$ $x + \mathsf{succ}(y) = \mathsf{succ}(x + y)$ $x - 0 = x$ $0 - 0 = 0$ $\mathsf{succ}(x) - \mathsf{succ}(y) = x - y$ $0 - \mathsf{succ}(x) = \mathsf{error}$

This specification is shorthand for the initial specification containing an extra constant error and equations specifying that an exception is propagated by all operations. Therefore the full specification (which we however usually do not write out) reads:

Init Spec	NAT
Sorts	nat
Constants	$0\colon \mathsf{nat}$, $\mathsf{error}\colon \mathsf{nat}$
Operations	$\mathsf{succ}\colon \mathsf{nat} \to \mathsf{nat}$ $+\colon \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$ $-\colon \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$
Variables	$x, y\colon \mathsf{nat}$
Equations	$x + 0 = x$ $x + \mathsf{succ}(y) = \mathsf{succ}(x + y)$ $x - 0 = x$ $0 - 0 = 0$ $\mathsf{succ}(x) - \mathsf{succ}(y) = x - y$ $0 - \mathsf{succ}(x) = \mathsf{error}$ $\mathsf{succ}(\mathsf{error}) = \mathsf{error}$ $\mathsf{error} + x = \mathsf{error}$ $x + \mathsf{error} = \mathsf{error}$ $x - \mathsf{error} = \mathsf{error}$ $\mathsf{error} - x = \mathsf{error}$

6.4 Modularisation

In example 6.2.11 we considered an initial specification of the algebra of finite sets of natural numbers. There are two obvious way to improve this specification using modularisation techniques:

Firstly, it seems unnecessary to repeat the signature of the booleans. Instead it would be better to **import** these from the ADT of booleans specified in example 6.1.5. This would give us the extra advantage of having available the usual operations on boolean values.

Secondly, there is nothing special about the natural numbers as being the type of elements of sets. Instead we could have specified finite sets of an arbitrary element type $\mathsf{element}$. Abstracting from a concrete type of elements gives us a **parametric** or **polymorphic** specification.

Applying both modularisation techniques we arrive at the following specification:

Init Spec	SET(element)
import	BOOLE
Sorts	set
Constants	emptyset : set
Operations	isempty : set $\longrightarrow$ boole insert : set $\times$ element $\longrightarrow$ set
Variables	x, y : element, s : set
Equations	insert(insert(s, x), x) = insert(s, x) insert(insert(s, x), y) = insert(insert(s, y), x) isempty(emptyset = T isempty(insert(s, x)) = F

The specification NATSET of example 6.2.11 con now simply obtained as SET(nat).

Combining specifications in the way described above requires some care. For example, one has to make sure that signatures do not overlap (e.g. in the example above nat must occur in no other signature than the one of the specification NAT). If there is an overlap, then appropriate renaming mechanisms have to resolve conflicts.

Our specification of the editor (example 6.2.12 can also be improved through modularisation as it contains the signature of finite lists (of characters). It would be better to *import* the specification of finite lists rather then mix it up with operations that concern the editor.

6.5 Abstraction through Information hiding

Another important aspect of algebraic specifications of abstract data types is *information hiding*. For example, in our specification of the editor we used the constructors for lists and also the constructor cf for creating a file from two lists. The constructor cf was a detail of a possible *implementation* of the data type of files, it was not present in the original informal description of an editor. Since we may later decide to change the implementation it is important that the user of the editor does not have access to such details. Otherwise a small change of an implementation detail of our editor my have a disastrous effect on a software system using this editor. All modern specification languages (see next chapter) and also most modern high-level programming languages have mechanisms for hiding operations, for example by making only those operations visible that are explicitely exported. Our specification of the editor might then, for example, look as follows (we also import lists):

Init Spec	EDITOR
Export	$\mathsf{file}, \mathsf{newfile}, \triangleright, \triangleleft, \mathsf{del}, \mathsf{write}$
Import	LIST(char)
Sorts	char, charlist, file
Constants	$\mathsf{newfile} \colon \mathsf{file}$ $a, \ldots, z, _ \colon \mathsf{char}$ $\mathsf{nil} \colon \mathsf{charlist}$
Operations	$\mathsf{cons} \colon \mathsf{char} \times \mathsf{charlist} \to \mathsf{charlist}$ $\mathsf{cf} \colon \mathsf{charlist} \times \mathsf{charlist} \to \mathsf{file}$ $\mathsf{write} \colon \mathsf{char} \times \mathsf{file} \to \mathsf{file}$ $\triangleleft \colon \mathsf{file} \to \mathsf{file}$ $\triangleright \colon \mathsf{file} \to \mathsf{file}$ $\mathsf{del} \colon \mathsf{file} \to \mathsf{file}$
Variables	$x \colon \mathsf{char}, \quad l, r \colon \mathsf{charlist}$
Equations	$\mathsf{newfile} = \mathsf{cf}(\mathsf{nil}, \mathsf{nil})$ $\mathsf{write}(x, \mathsf{cf}(l, r)) = \mathsf{cf}(\mathsf{cons}(x, l), r)$ $\triangleleft\ (\mathsf{cf}(\mathsf{nil}, r)) = \mathsf{cf}(\mathsf{nil}, r)$ $\triangleleft\ (\mathsf{cf}(\mathsf{cons}(x, l), r)) = \mathsf{cf}(l, \mathsf{cons}(x, r))$ $\triangleright\ (\mathsf{cf}(l, \mathsf{nil})) = \mathsf{cf}(l, \mathsf{nil})$ $\triangleright\ (\mathsf{cf}(l, \mathsf{cons}(x, r))) = \mathsf{cf}(\mathsf{cons}(x, l), r)$ $\mathsf{del}(\mathsf{cf}(l, \mathsf{nil})) = \mathsf{cf}(l, \mathsf{nil})$ $\mathsf{del}(\mathsf{cf}(l, \mathsf{cons}(x, r))) = \mathsf{cf}(l, r)$

6.6 Specification languages

Stand alone specifications (loose or initial) that are not combined from other specifications (e.g. BOOLE, NAT) are called **atomic specifications**. Starting from these atomic specifications one can build more complex specifications by certain operations. The operations **import** and **parametrisation** were discussed in the previous section. Other important operations are:

Union If Spec_1 and Spec_2 are specifications then $\mathsf{Spec}_1 + \mathsf{Spec}_2$ is a specification.

The signature of $\mathsf{Spec}_1 + \mathsf{Spec}_2$ is $\Sigma_1 \cup \Sigma_2$, where Σ_i is the signature of Spec_i (assuming that Σ_1 and Σ_2 are 'compatible').

A $\Sigma_1 \cup \Sigma_2$-algebra A is a model of $\mathsf{Spec}_1 + \mathsf{Spec}_2$ if and only if $A_{|\Sigma_1}$ is a model of Spec_1 and $A_{|\Sigma_2}$ is a model of Spec_2.

Restriction If Spec is a specification with signature Σ and Σ_0 is a subsignature of Σ then $\mathsf{Spec}_{|\Sigma_0}$ is a specification with signature Σ_0.

A Σ_0-algebra A is a model of $\mathsf{Spec}_{|\Sigma_0}$ if and only if $A = B_{|\Sigma_0}$ for some model B of Spec.

Further fundamental construction principles for specifications are **renaming**, **inheritance** and **quotients**.

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**.

6.6.1 Example

In example 6.2.8 we produced an initial specification of the algebra of natural numbers with 0, successor and addition. In example 6.2.9 and exercise 6.2.10 we extended this by 'cut-off' subtraction, $n \dot{-} m$, 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 (thus pursuing approach no. (iii) of section 6.3.

Let Init–Spec$(\Sigma, E,)$ be the initial specification of example 6.2.8 (specifying the natural numbers with 0 and addition). Let (Σ', E') be the loose specification, where Σ' is Σ expanded by the operation $-$ (minus), and E' consists of the equations

$$\begin{aligned} x - 0 &= x \\ \mathsf{succ}(x) - \mathsf{succ}(y) &= x - y \end{aligned}$$

Then the models of the specification

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

are, up to isomorphism, exactly those Σ'-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 [Daw, Jac].

ASL A kernel language for algebraic specifications [SW].

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

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

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

CCS, CSP Formal languages for specifying concurrent processes [Mil, Hoa].

UML A design and modelling language for object oriented programming [BRJ].

CASL Common Algebraic Specification Language. Machine support by the interactive Theorem Prover Isabelle/HOL. Integration of Process Algebra [Ast, Rog].

6.6.2 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 modelled 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). [2] For example if our specification of an editor (6.2.12) 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).

6.7 Summary and Exercises

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

- *Loose specifications*: Arbitrary axioms are allowed. Every algebra satisfying the axioms is a model. The class of all models of a consistent loose specification forms an ADT. By the Loewenheim-Skolem Theorem, loose specifications usually cannot pin down ADTs up to isomorphism, that is, the model class is a *polymorphic* ADT. Persistent extensions.

- *Initial Specifications*: Only *equations* are allowed as axioms. Every algebra which is initial in the class of all loose models of a specification is an initial model. The class of all models of an initial specification forms an ADT. Initial specifications do pin down ADTs up to isomorphism, that is, the model class is a *monomorphic* ADT. A model of an initial specification can be constructed as a quotient of the term algebra (see Theorem 6.2.5).

- Generators and observers, exception handling,

- *Modularisation*: Structuring specifications using *import* declarations and *polymorphic parametrisation*. *Abstraction*: *Information Hiding* via *export* declarations.

[2] Algebras with state are often called **evolving algebras** (Börger), or **abstract state machines** (Gurevich).

- Specification Languages.

Exercises. 1. Consider the following loose specification $\mathsf{LIST}(\mathsf{BOOLE})$:

Loose Spec	
Sorts	boole, list
Constants	$\mathrm{T}\colon \mathsf{boole}$, $\mathrm{F}\colon \mathsf{boole}$, $\mathrm{nil}\colon \mathrm{list}$
Operations	$\mathrm{cons}\colon \mathsf{boole} \times \mathrm{list} \to \mathrm{list}$ $\mathrm{first}\colon \mathrm{list} \to \mathsf{boole}$ $\mathrm{rest}\colon \mathrm{list} \to \mathrm{list}$
Variables	$x\colon \mathsf{boole}$, $l\colon \mathrm{list}$
Axioms	$\mathrm{first}(\mathrm{cons}(x,l)) = x$ $\mathrm{rest}(\mathrm{cons}(x,l)) = l$

Let Σ be the signature of $\mathsf{LIST}(\mathsf{BOOLE})$:

Show that the following Σ-algebra A is a model of $\mathsf{LIST}(\mathsf{BOOLE})$:

$A_{\mathsf{boole}} := \{\#\mathrm{t},\ \#\mathrm{f}\}$,

$A_{\mathrm{list}} :=$ the set of finite lists of boolean values.

$\mathrm{T}^A := \#\mathrm{t}$,

$\mathrm{F}^A := \#\mathrm{f}$,

$\mathrm{nil}^A := []$ (the empty list).

$\mathrm{cons}^A(a, [a_1, \ldots, a_n]) := [a, a_1, \ldots, a_n]$

$$\mathrm{first}^A(l) := \begin{cases} \#\mathrm{f} & \text{if } l = [] \\ \text{the first element of } l & \text{otherwise} \end{cases}$$

$$\mathrm{rest}^A(l) := \begin{cases} [] & \text{if } l = [] \\ \text{the result of removing the first element from } l & \text{otherwise} \end{cases}$$

2. Is the Σ-algebra A, defined in the previous exercise, initial in the class of all models of $\mathsf{LIST}(\mathsf{BOOLE})$? Justify your answer.

3. Determine for the specifications BOOLE, SET, EDITOR and TREE generators and observers. Are the corresponding algebras freely generated by the generators?

4. In the exercises 5.4 (a) and (c) we discussed the algebra A of natural numbers and also the algebra C of lists of natural numbers with the empty list and concatenation of lists (so A and C are algebras over the same signature). Show that there is a bijection between A and C, but no isomorphism.

Hint: For showing that A and C are not isomorphic use Theorem 6.1.3.

5. Let (Σ', Φ') be an extension of (Σ, Φ) such that for every closed Σ-formula P it holds that if $\Phi' \models P$ then $\Phi \models P$.

Show that (Σ', Φ') is a persistent extension of (Σ, Φ).

6. Show that if A is a Σ-algebra and A' is an expansion of A, then for every Σ-formula P it holds that $A \models P$ if and only if $A' \models P$.

Hint: Structural induction on P.

7. Extend the specification NATSET in Example 6.2.11 by an operation $\mathsf{member} : \mathsf{nat} \to \mathsf{set} \to \mathsf{boole}$ and equations that specify the new operation as a test for membership. Show that the extension is persistent.

8. Extend the specification NATSET in Example 6.2.11 by an operation $\mathsf{select} : \mathsf{set} \to \mathsf{nat}$ and the equations

$$\mathsf{select}(\mathsf{nil}) = 0$$
$$\mathsf{select}(\mathsf{insert}(s, x)) = x$$

Is this extension persistent?

9. Give an initial specification of FIFO (first-in-first-out) queues of natural numbers. The signature should contain (among other things) the operations

- snoc : queue → nat → queue, inserting an element at the end of a queue,
- head : queue → nat, computing the first element of a nonempty queue,
- tail : queue → queue, computing the tail of a nonempty queue (first element removed),
- member : nat → queue → boole, testing membership,
- length : queue → nat, computing the length of a queue,
- isempty : queue → boole, testing whether a queue is empty.

Determine constructors and observers. Are the constructors free?

10. Let Σ be the signature with one sort, one constant, 0, and one binary operation, $+$. Let A be the Σ-algebra of real numbers with 0 and addition. For any Σ-formula P with exactly one free variable x and any real number $r \in A$, we let "$A \models P(r)$" mean "$A, \alpha \models P$ for some (or any) variable assignment α such that $\alpha(x) = r$".

Show that if $A \models P(r)$ for *some* $r \neq 0$, then $A \models P(s)$ for *all* $s \neq 0$.

Hint: Show that for every real number $c \neq 0$ the function $\varphi \colon A \to A$, defined by $\varphi(a) := c * a$, is an automorphism. Now use Theorem 6.1.3.

Remark: From this exercise it follows that no non nontrivial properties of real numbers can be expressed by a formula built from 0 and + only. The only exception are "$x = 0$" and "$x \neq 0$". We cannot express, for example, "$x > 0$", or "$x = 1$".

11. Let Σ be the signature with one sort, two constants, 0 and 1, and two binary operations, $+$ and $*$. Let R be the Σ-algebra of real numbers with 0, 1, addition and multiplication.

(a) Show that "$x < y$" is expressible by a Σ-formula.

(b) Let $\varphi \colon R \to R$ be an automorphism. Show that $\varphi(q) = q$ for all rational numbers q.

(c) Show that the only automorphism on R is the identity.

Hint for (c). Use parts (a) and (b) as well as Theorem 6.1.3 to show that for all rationals q and all reals r we have $q < \varphi(r)$ if and only if $q < r$. This clearly implies $\varphi(r) = r$ for all reals r.

Remark: On the Σ-algebra C of complex numbers there exist exactly two automorphisms: The identity, and the mapping sending a complex number $z = x + iy$ to its conjugate complex $\overline{z} = x - iy$.

7 Implementation of Abstract Data Types

Abstract data types given by an initial specification can be implemented in most modern programming languages. By means of some simple examples, we describe and compare the implementation of abstract data types in a functional and an object-oriented style. We discuss the dangers of breaking abstraction barriers in software development, or using non-persistent, that is, destructive operations (which are typical in imperative programming). We also describe some simple techniques of improving the efficiency of the implementation of ADTs.

7.1 Implementing ADTs in Functional and Object Oriented Style

Consider the following initial specification of an abstract data type of binary trees with natural numbers attached to each node.

Init Spec	TREE
import	NAT, BOOLE
Sorts	tree
Operations	$\mathsf{leaf}\colon \mathsf{nat} \to \mathsf{tree}$ $\mathsf{branch}\colon \mathsf{tree} \times \mathsf{nat} \times \mathsf{tree} \to \mathsf{tree}$ $\mathsf{isleaf}\colon \mathsf{tree} \to \mathsf{boole}$ $\mathsf{root}\colon \mathsf{tree} \to \mathsf{nat}$ $\mathsf{left}\colon \mathsf{tree} \to \mathsf{tree}$ $\mathsf{right}\colon \mathsf{tree} \to \mathsf{tree}$
Variables	$x\colon \mathsf{nat}, \quad s, t\colon \mathsf{tree}$
Equations	$\mathsf{isleaf}(\mathsf{leaf}(x)) = \mathsf{T}$ $\mathsf{isleaf}(\mathsf{branch}(s,x,t)) = \mathsf{F}$ $\mathsf{root}(\mathsf{leaf}(x)) = x$ $\mathsf{root}(\mathsf{branch}(s,x,t)) = x$ $\mathsf{left}(\mathsf{leaf}(x)) = \mathsf{error}$ $\mathsf{left}(\mathsf{branch}(s,x,t)) = s$ $\mathsf{right}(\mathsf{leaf}(x)) = \mathsf{error}$ $\mathsf{right}(\mathsf{branch}(s,x,t)) = t$

We first implement the Abstract Data Type TREE in the functional programming language *Haskell*. Obviously, the type of integers in TREE could be replaced by any other data type. Therefore we define, more generally, a polymorphic data type of trees with labels from some unspecified type a.

Using Haskell's `data` construct the definition of the data type `Tree` is very easy, and, using pattern matching, the equations of the specifications literally translate into a Haskell program. In order to hide the the way the data type is defined the constructors are not exported directly, but only aliases of them.

```
module Tree (Tree,leaf,branch,isleaf,root,left,right) where

data Tree a = Leaf a | Branch (Tree a) a (Tree a)

leaf :: a -> Tree a
leaf x = Leaf x

branch :: Tree a -> a -> Tree a -> Tree a
branch s x t = Branch s x t

isleaf :: Tree a -> Bool
isleaf (Leaf x)       = True
isleaf (Branch s x t) = False

root :: Tree a -> a
root (Leaf x)       = x
root (Branch s x t) = x

left   :: Tree a -> Tree a
left (Leaf x)       = error "left of Leaf"
left (Branch s x t) = s

right  :: Tree a -> Tree a
right (Leaf x)       = error "right of Leaf"
right (Branch s x t) = t
```

Now we implement TREE in the object-oriented programming language *Java*. As trees come in two shapes, either $\mathsf{leaf}(x)$, or $\mathsf{branch}(s,x,t)$, we cannot implement trees as objects of one class, because, roughly speaking, one class can contain objects of one shape only. One way around this problem is to define an *abstract class* of trees with abstract methods `getRoot` and `isLeaf` and two subclasses, one for trees of the shape $\mathsf{leaf}(x)$ and one for trees of the shape $\mathsf{branch}(s,x,t)$:

```
public abstract class Tree {
 public abstract int getRoot () ;
 public abstract boolean isLeaf () ;
}

public class Leaf extends Tree {
 private int label ;
 public Leaf (int x) {
```

```
  label = x ;
 }
 public int getRoot () {
  return label ;
 }
 public boolean isLeaf () {
  return true ;
 }
}

public class Branch extends Tree {
 private Tree left ;
 private int label ;
 private Tree right ;
 public Branch (Tree s, int x, Tree t) {
  left = s ;
  label = x ;
  right = t ;
 }
 public int getRoot () {
  return label ;
 }
 public boolean isLeaf () {
  return false ;
 }
 public Tree getLeft () {
  return left ;
 }
 public Tree getRight () {
  return right ;
 }
}
```

The following should be noted about the Java implementation:

- The generators of trees are given by the constructors `Leaf` and `Branch`. Strictly speaking, this violates the principle of abstractness since a detail of the implementation is revealed to the user.
- The observers `getLeft` and `getRight` are defined only for objects of the class `Branch`, but not for `Tree` in general.
- Java does not support polymorphic data types (as Haskell does). There are however extensions of Java (Generic Java, Poly Java, Pizza) supporting parametric polymorphism (and more).

In order to compare the functional and the object-oriented implementation according to flexibility w.r.t. modifications, we enrich our abstract data type TREE by an observer

$$\mathsf{depth}\colon \mathsf{tree} \to \mathsf{nat}$$

computing the depth of a tree.

In Haskell, we simply add to the module `Tree` the lines

```
depth  :: Tree a -> Int
depth (Leaf x)       = 0
depth (Branch s x t) = 1 + max (depth s) (depth t)
```

In Java, however, we have to enlarge the abstract class `Tree` as well as both subclasses, `Leaf` and `Branch`, by suitable methods:

```
public abstract class Tree {
 public abstract int getRoot () ;
 public abstract boolean isLeaf () ;
 public abstract int depth () ;
}

public class Leaf extends Tree {
 private int label ;
 public Leaf (int x) {
  label = x ;
 }
 public int getRoot () {
  return label ;
 }
 public boolean isLeaf () {
  return true ;
 }
 public int depth () {
  return 0 ;
 }
}

public class Branch extends Tree {
 private Tree left ;
 private int label ;
 private Tree right ;
 public Branch (Tree s, int x, Tree t) {
  left = s ;
  label = x ;
  right = t ;
```

```
 }
 public int getRoot () {
  return label ;
 }
 public boolean isLeaf () {
  return false ;
 }
 public Tree getLeft () {
  return left ;
 }
 public Tree getRight () {
  return right ;
 }
 public int depth () {
  return (1 + max(left.depth,right.depth)) ;
 }
}
```

In general, modifications like this, scattered through the program code, are extremely error prone if not totally infeasible. In order to minimise the risk of introducing errors it that way, it therefore is advisable to keep abstract data types rather small in terms of the number of operations.

Next we modify our abstract data type TREE by a generator

$$\mathsf{treesucc}\colon \mathsf{nat} \times \mathsf{tree}$$

that generates from a number and *one* tree a new tree.

This time the extension in Haskell is more awkward because we have to extend the definition of each observer by a new clause for the new generator. On the other hand the corresponding extension in Java is straightforward: We just have to add a new (sub)class.

Exercise: Carry out the extensions of the abstract data type TREE by the generator treesucc in Haskell as well as in Java.

7.2 Efficiency

In our Haskell implementation of the abstract data type TREE, all operations clearly run in time $O(1)$, that is, in constant time, except for the operation `depth`. The latter has time complexity $O(n)$ (where n is the number of labels of a tree) since `depth` has to run through all nodes of a tree in order to determine its depth (the Java implementation has the same complexities). There is a simple way to improve the implementation such that `depth` runs in constant time: Just add an extra argument to the constructor `Branch` recording the depth of the tree:

```
module Tree (Tree,leaf,branch,isleaf,root,left,right,depth) where

data Tree a = Leaf a | Branch Int (Tree a) a (Tree a)

depth :: Tree a -> Int
depth (Leaf x)         = 0
depth (Branch d s x t) = d

leaf :: a -> Tree a
leaf x = Leaf x

branch :: Tree a -> a -> Tree a -> Tree a
branch s x t = Branch (1 + max (depth s) (depth t)) s x t

isleaf :: Tree a -> Bool
isleaf (Leaf x)         = True
isleaf (Branch d s x t) = False

root :: Tree a -> a
root (Leaf x)         = x
root (Branch d s x t) = x

left   :: Tree a -> Tree a
left (Leaf x)         = error "left of Leaf"
left (Branch d s x t) = s

right  :: Tree a -> Tree a
right (Leaf d)         = error "right of Leaf"
right (Branch d s x t) = t
```

Remarks.

1. Note that now no longer all objects of the form `Branch d s x t` represent legal trees, but only those where `d` is actually the height of the tree given by `s`, `x` and `t`. However, this is unproblematic because using the exported operations only legal trees can be generated. The situation that not all elements of a type are legal representatives of the implemented ADT is very common. Ordered lists or balanced trees representing sets or finite maps are examples.

2. How do we know that our implementation is *correct*? In this example, the correctness proof is rather trivial: We show that the property "`depth(t)` is the depth of `t`" holds for terms of the form `leaf x` and is preserved by the operation `branch`. Both facts are obvious.

3. Consider what would happen if in Section 7.1 we hadn't made the function `branch` abstract, but had used the constructor `Branch` directly: In that case, all expressions of the form `Branch s x t` would have to be modified to `Branch d s x t`. Needless to say that such kinds of editings throughout a program (not just within one ADT) are very dangerous.

4. The dramatic effect of this modification of the implementation on the efficiency of `depth` can be seen by generating large trees using the function

```
mkTree :: Int -> TreeInt
mkTree n | n <= 0    = leaf 0
         | otherwise = let t = mkTree (n-1) in branch t n t
```

and evaluating the expression `depth (mkTree 100)` under both implementations.

5. Finally, it should be stressed that the new implementation is, for the user, indistinguishable from the old one (except for efficiency, of course).

7.3 Persistence

Suppose we wish to implement in Haskell the ADT of queues as described informally in Exercise 9 in Section 6.7. A first solution implements queues as lists and the operation $\mathsf{snoc}(q, x)$ as $q +\!\!+ [x]$ (appending the singleton list $[x]$ to q). For brevity we do not implement all operation of Exercise 9. We do also make the implementation polymorphic in the type of elements of a queue.

```
module Queue (Queue,emptyQ,snoc,head,tail) where

import Prelude hiding (head,tail)

type Queue a = [a]

emptyQ :: Queue a
emptyQ = []

snoc :: Queue a -> a -> Queue a
snoc q x = q ++ [x]

head :: Queue a -> a
head []     = error "head of empty queue"
head (x:xs) = x

tail :: Queue a -> Queue a
tail []     = error "tail of empty queue"
tail (x:q) = q
```

In this implementation the runtime of `snoc` is $O(n)$ because the append function is defined by recursion on its first argument:

```
(++)              :: [a] -> [a] -> [a]
[]     ++ ys       = ys
(x:xs) ++ ys       = x : (xs ++ ys)
```

Hence $xs ++ ys$ takes $\mathsf{length}(xs)$ many steps and consequently $\mathsf{snoc}(q, x)$ takes $\mathsf{length}(q)$ many steps.

In an imperative programming language a better runtime of the append function can be easily achieved by implementing a list as a linked structure with a pointer to the head and the end of the list. Then, appending two lists boils down to some simple pointer manipulations, as illustrated in figure 4 on page 92. Clearly, the runtime is independent of the lengths of the lists, that is, $O(1)$. Note, however, that the arguments to ++ are *destroyed* when executing

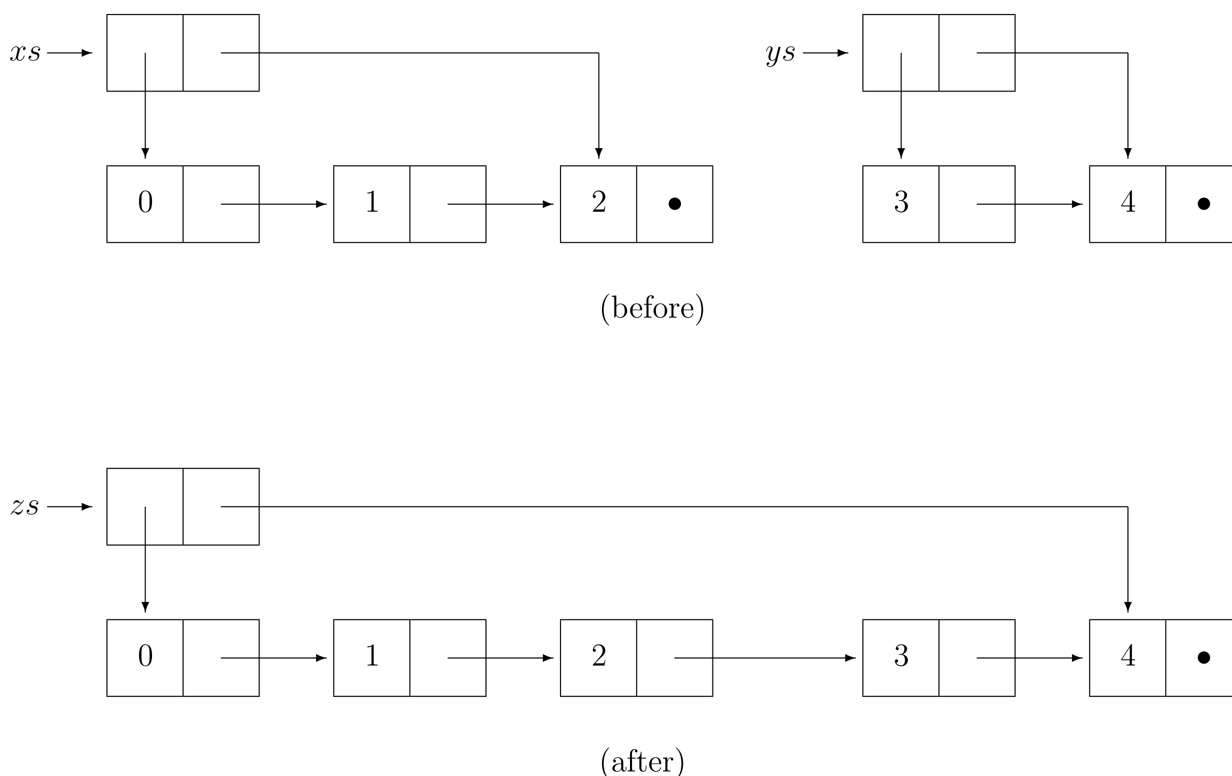


Figure 4: Executing $zs = xs ++ ys$ in an imperative setting. This operation destroys the argument lists xs and ys [Oka].

the operation. In other words, the imperative implementation of ++ is not *persistent*.

In contrast, in a functional language, arguments of an operations are never destroyed. They can still be freely used after executing the operation. In the case of the append operation, the first argument is *copied* to be used as a part of the result, as illustrated in figure 5 on page 93. At first glance it might seem that in this example (and many others) persistence is incompatible with (time and space) efficiency. There are, however, possibilities to reconcile persistence and efficiency in a purely functional setting. For example, we can implement a queue by two lists, Qfr (Q is the constructor for queues), where f represents the *front* and r the *reversed rear* of the queue, maintaining the invariant that whenever f is empty, so is r:

```
module Queue (Queue,emptyQ,snoc,head,tail) where

import Prelude hiding (head,tail)
```

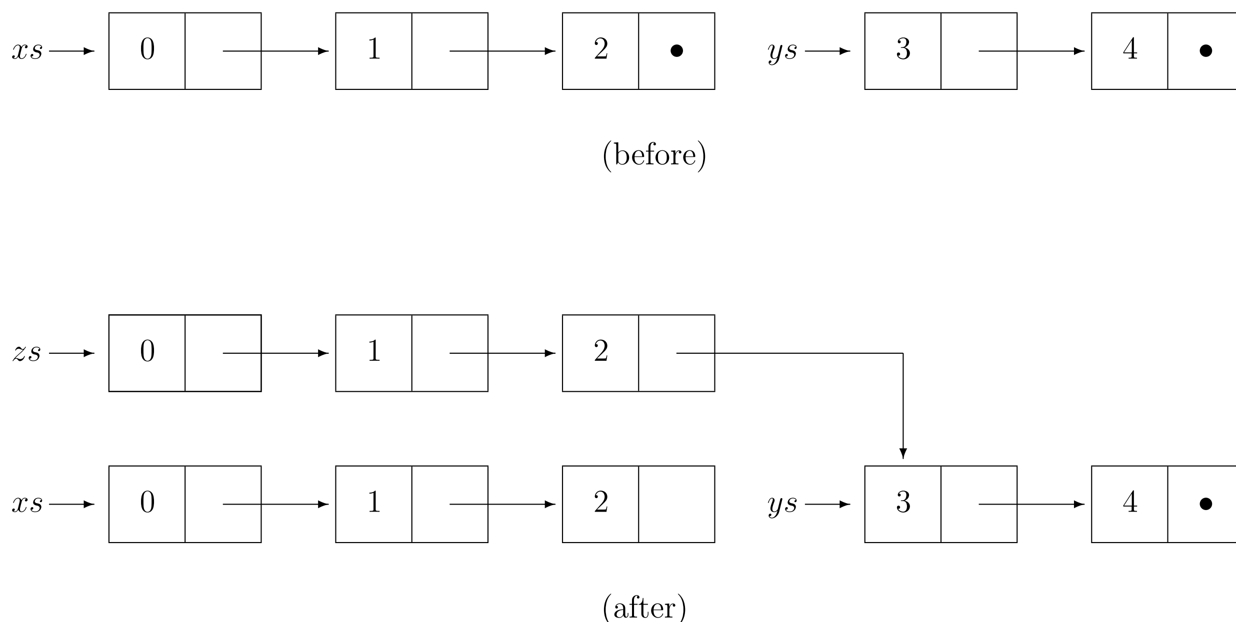


Figure 5: Executing $zs = xs ++ ys$ in a functional setting. The arguments xs and ys are unaffected by the operation [Oka].

```
data Queue a = Q [a] [a] deriving Show

emptyQ :: Queue a
emptyQ = Q [] []

snoc :: Queue a -> a -> Queue a
snoc (Q [] r) x = Q [x] []
snoc (Q f r) x  = Q f (x:r)

head :: Queue a -> a
head (Q [] r)    = error "head of empty queue"
head (Q (x:f) r) = x

tail :: Queue a -> Queue a
tail (Q [] r)    = error "tail of empty queue"
tail (Q [x] r)   = Q (reverse r) []
tail (Q (x:f) r) = Q f r
```

Clearly, the operations snoc and head run in constant time. Furthermore, tail(Qfr) runs in constant time except when f happens to be a singleton. Since the runtime of of tail(Qfr) is bounded by the number of snoc operations needed to build up r one says that tail runs in *constant amortised time*. Practically, this means that when we do not make use of persistence

(that is, we use the queue in a single threaded way), then, when viewed as part of a sequence of operation, the operation head behaves as if it ran in constant time (see [Oka] for details).

7.4 Structural Bootstrapping

The implementation of queues in the previous section can be slightly improved by maintaining the stronger invariant $\mathsf{length}(f) \geq \mathsf{length}(r)$ (Exercise 3 in Section 7.6). A more substantial performance improvement can be achieved by a technique called *structural bootstrapping*. The general idea of bootstrapping ("pulling yourself up by your bootstraps") is to obtain a solution to a problem from another (simpler, incomplete, or inefficient) instance of the same problem. In the case of queues, the idea is to split the front part of a queue into a list f and a queue $m_1, \ldots, m_k$ of reversed rear parts, maintaining the invariant

$$\mathsf{length}(f) + \mathsf{length}(m_1) + \ldots + \mathsf{length}(m_k) \geq \mathsf{length}(r).$$

In order to make the computation of the lengths efficient we add the numbers $lfm := \mathsf{length}(f) + \mathsf{length}(m_1) + \ldots + \mathsf{length}(m_k)$ and $\mathsf{length}(r)$ as extra arguments the constructor of a queue.

```
data Queue a = E | Q Int [a] (Queue [a]) Int [a] deriving Show

emptyQ :: Queue a
emptyQ = Q 0 [] E 0 []

snoc :: Queue a -> a -> Queue a
snoc E x              = Q 1 [x] E 0 []
snoc (Q lfm f m lr r) x = check lfm f m (lr+1) (x:r)

head :: Queue a -> a
head E       = error "head of empty queue"
head (Q lfm (x:f) m lr r) = x

tail :: Queue a -> Queue a
tail E                    = error "tail of empty queue"
tail (Q lfm (x:f) m lr r) = check (lfm-1) f m lr r

check,checkF :: Int -> [a] -> Queue [a] -> Int -> [a] -> Queue a
check lfm f m lr r =
  if lfm >= lr then checkF lfm f m lr r
  else checkF (lfm+lr) f (snoc m (reverse r)) 0 []

checkF lfm [] E lr r = E
checkF lfm [] m lr r = Q lfm (head m) (tail m) lr r
checkF lfm f m lr r  = Q lfm f m lr r
```

Remarks. 1. In the implementation above, a queue of elements of type a has a middle part consisting of a queue of elements of type [a]. In technical terms: the definition of the data

type `Queue a` above is an instance of *polymorphic recursion*.

2. The efficiency of this implementation relies to a large extent that Haskell is a *lazy language*, that is, the execution of operations is suspended until the result is actually needed. The gain in efficiency is greatest in applications that make heavy use of persistence. This is explained in detail in [Oka].

3. Although all operations of this implementation run in constant amortised time, it can happen that an application of a head or tail operation takes time proportional to the length of the queue. The reason is that sometimes rather long lists need to be reversed and reversion is a *batched* operation, that is, it needs to be fully executed when needed.

4. In applications where predictability is more important than raw speed (for example, one might prefer to have 1000 times 0.2 seconds response time rather than having 999 times 0.1 seconds, but once 20 seconds), then one is more interested in worst case complexity, but not in amortised complexity. To achieve good worst time behaviour for queues one needs to replace the operation of reversing by a more complex operation whose execution can be scheduled (see [Oka] for details).

7.5 Correctness

We have seen a few implementations of abstract data types. How can we prove that these implementations are correct? Before we can answer this question we need to know what it means for an implementation to be correct, end, first of all, we need to clarify what it means to implement an abstract data type.

The implementation of queues by two lists (front and rear), at the end of Section 7.3, gives us a good guideline for answering these questions.

(1) The data type

```
data Queue a = Q [a] [a]
```

together with the operations snoc, head, tail defines an algebra.

(2) Not all elements of this algebra are legal queues: For `Q f r` to be legal, we require that if `f` is empty, then so is `r`. The legal elements of `Queue a` form a *subalgebra* (this requires a proof that the operations preserve legality).

(3) Different legal elements of `Queue a` may denote the same queue. For example, `Q [1,2] [4,3]` denotes the same queue as `Q [1,2,3] [4]`. The relation of denoting the same queue is a *congruence* on the subalgebra defined in (2) (this requires a proof that the operations respect this relation).

(4) Summing up, we see that the abstract data type of queues is implemented as a *quotient of a subalgebra* of the algebra `Queue a`.

It remains to be shown that this quotient is indeed a model of our ADT of queues. This can be done in different ways:

(a) We can prove directly that this quotient is a model of a given specification. In our case we could use the initial specification to be found in Exercise 9 in Section 6.7 and use Theorem 6.2.5.

(b) We can use a “canonical model” of queues and prove that our implementation is *isomorphic* to the canonical model. According to the *Homomorphism Theorem* 5.2.8 it suffices to define an epimorphism from the subalgebra in (2) to the canonical model such that the congruence in (3) coincides with the congruence induced by the epimorphism. The latter means that two legal elements in `Queue a` denote the same queue if and only if the homomorphism maps them to the same element in the canonical model.

In our example, the canonical model would be the data type of lists, `[a]`, with the implementations of snoc, head, tail as defined at the beginning of Section 7.3. The homomorphism maps `Q f r` to `f ++ reverse r` (it has to be checked that this is indeed a homomorphism). The congruence in (3) is defined exactly such that (b) is fulfilled. Hence we know that our implementation of queues is correct.

7.6 Summary and Exercises

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

- Implementing abstract data types in *functional* and *object-oriented* style. Abstraction by hiding constructors of the data type. Negative effects of breaking abstraction barriers.
- Gaining efficiency through adding information to the constructors of a data type.
- Persistence: The application of an operation does not destroy the arguments. Always satisfied in a functional, but not necessarily in an imperative setting.
- Further gain of efficiency through structural bootstrapping. Polymorphic recursion.
- Implementing an ADT as a quotient of a subalgebra of a concrete data type. Using the Homomorphism Theorem to prove correctness.

Exercises. 1. Extend the abstract data type TREE of Section 7.1 by a generator

$\mathsf{treesucc}\colon \mathsf{nat} \times \mathsf{tree}$

that generates from a number and *one* tree a new tree. Carry out the extensions of the abstract data type TREE by the generator treesucc in Haskell as well as in Java.

2. Implement in Haskell an ADT of integer labelled trees, similar to the trees in Section 7.1, but with an extra operation computing the sum of the labels in a tree. Make sure that all operations run in constant time.

3. Implement in Haskell a variant of the queues in Section 7.3 that maintains the stronger invariant $\mathsf{length}(f) \geq \mathsf{length}(r)$.

4. Extend the data type `Queue a` defined at the beginning of Section 7.3 by all the operation specified in Exercise 9 of Section 6.7.

5. Prove that the legal elements on `Queue a`, as defined in Section 7.5, form a subalgebra of `Queue a`. Furthermore, prove that the map that sends a legal `Q f r` to `f ++ reverse r` is an epimorphism.

Part III

Advanced Methods of Program Development

8 Term Rewriting and Rapid Prototyping

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, 4.3.1, 4.3.2, 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 and equality rules 4.2 are needed (Birkhoff's Theorem 8.1.5). 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 8.5).

Assumption. From now on we will always assume that all signatures Σ 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 Σ-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 Σ-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.

8.1 Equational logic

8.1.1 Definition

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

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

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

Transitivity $$\frac{t_1 = t_2 \qquad t_2 = t_3}{t_1 = t_3}$$

Compatibility $$\frac{t_1 = t_1' \qquad \dots \qquad t_n = t_n'}{f(t_1, \dots, t_n) = f(t_1', \dots, t_n')}$$

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

In these rules $t, t_i, t_i' \in \mathrm{T}(\Sigma, X)$, $\theta\colon X \to \mathrm{T}(\Sigma, Y)$ is a substitution, and f is an operation of the signature Σ. 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).

8.1.2 Example

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

$$\dfrac{\dfrac{x + \mathsf{succ}(y) = \mathsf{succ}(x + y)}{0 + \mathsf{succ}(0) = \mathsf{succ}(0 + 0)}\,\text{Inst} \qquad \dfrac{\dfrac{x + 0 = x}{0 + 0 = 0}\,\text{Inst}}{\mathsf{succ}(0 + 0) = \mathsf{succ}(0)}\,\text{Comp}}{0 + \mathsf{succ}(0) = \mathsf{succ}(0)}\,\text{Trans}$$

8.1.3 Lemma

If $\vdash_E t_1 = t_2$ then $\forall E \vdash_{\mathrm{m}} t_1 = t_2$.

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

8.1.4 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 8.1.3 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 4.3.1, it follows $\forall E \models \forall(t_1 = t_2)$.

8.1.5 Completeness Theorem (Birkhoff)

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

Proof. Let Σ be the signature of $E \cup \{t_1 = t_2\}$ and let V be the set of all variables for the sorts of Σ. Then $\mathrm{T}(\Sigma, V)$ is the set of all Σ-terms which, according to definition 3.2.7, can be viewed as a Σ-algebra. For Σ-terms t_1, t_2 of the same sort we define

$$t_1 =_E t_2 \quad :\Leftrightarrow \quad \vdash_E t_1 = t_2$$

Because of the rules *Reflexivity*, *Symmetry*, *Transitivity*, and *Compatibility* this defines a congruence $=_E$ on $\mathrm{T}(\Sigma, V)$. Set

$$A := \mathrm{T}(\Sigma, V)/{=_E}$$

The following fact will be crucial in the rest of the proof. Let t be a Σ-term and $\alpha\colon X \to A$ a variable assignment, where $X \supseteq \mathsf{FV}(t)$. Then

$$t^{A,\alpha} \quad = \quad [t\theta]_{=_E} \qquad (+)$$

for any substitution $\theta\colon X \to \mathrm{T}(\Sigma, V)$, such that $\theta(x) \in \alpha(x)$ for all $x \in X$.

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

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

We now use (+) to show the following.

$$A \models \forall(t_1 = t_2) \quad \Leftrightarrow \quad \vdash_E t_1 = t_2 \qquad (++)$$

for all Σ-terms t_1, t_2 of the same sort.

Proof of (++).

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

$$[t_1]_{=_E} \quad = \quad [t_1\theta]_{=_E} \quad \overset{(+)}{=} \quad t_1^{A,\alpha} \quad \overset{A\models\forall(t_1=t_2)}{=} \quad t_2^{A,\alpha} \quad = \quad \ldots \quad = \quad [t_1]_{=_E}$$

i.e. $\vdash_E t_1 = t_2$.

'$\Leftarrow$'. Assume $\vdash_E t_1 = t_2$. Let $\alpha\colon X \to A$ be a variable assignment We have to show $t_1^{A,\alpha} = t_2^{A,\alpha}$.

Define a substitution $\theta\colon X \to \mathrm{T}(\Sigma, V)$ 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 (+) a second time we get $t_i^{A,\alpha} = [t_i\theta]_{=_E}$ 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 $[t_1\theta]_{=_E} = [t_2\theta]_{=_E}$ and therefore $t_1^{A,\alpha} = t_2^{A,\alpha}$. Thus (++) is proved.

From (++), '$\Leftarrow$' it follows that A is a model of $\forall E$, since

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

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 (++), $\Rightarrow$.

8.2 Term rewriting systems

Now we show how equational logic can be mechanised.

8.2.1 Definition

A **term rewriting system** over a signature Σ is a finite set R of **rewrite rules** $l \mapsto r$, where r and l are Σ-terms, such that

(i) l is not a variable,

(ii) $\mathsf{FV}(r) \subseteq \mathsf{FV}(l)$.

Given a term rewriting system R we define a binary relation $\to_R$ on the set of Σ-terms by

$$t \to_R t' \quad :\Leftrightarrow \quad t \equiv u\{l\theta/x\} \text{ and } t' \equiv u\{r\theta/x\}$$

for some rewrite rule $l \mapsto r \in R$,
some Σ-term u with exactly one occurrence of some variable x, and
some substitution $\theta\colon X \to \mathrm{T}(\Sigma, Y)$

In other words, $t \to_R t'$ 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 θ is a substitution. We also say that the subterm $l\theta$ **matches** l, or is an **instance** of l.

If $t \to_R^* t'$ we say **rewrites to** t',

We call $\to_R$ the **term rewriting relation generated by** R.

Furthermore we define

$$t \to_R^* t' \quad :\Leftrightarrow \quad t \equiv t_0 \to_R \ldots \to_R t_n \equiv t' \text{ for some } \Sigma\text{-terms } t_0, \ldots, t_n$$

$$t \leftrightarrow_R t' \quad :\Leftrightarrow \quad t \to_R t' \text{ or } t' \to_R t$$

$$t \simeq_R t' \quad :\Leftrightarrow \quad t \equiv t_0 \leftrightarrow_R \ldots \leftrightarrow_R t_n \equiv t' \text{ for some } \Sigma\text{-terms } t_0, \ldots, t_n$$

Any finite or infinite sequence $t_0 \to_R t_1 \to_R \ldots$ is called a **reduction sequence**.

A term t is in **normal form** w.r.t. R if it cannot be rewritten, i.e. $t \not\to_R t'$ for any t'.

We say that t' **is a normal form of** t, or t **normalises to** t' if $t \to_R^* t'$ and t' is in normal form.

8.2.2 Definition

Let E be a set of equations over the signature Σ. If the set $R := \{l \mapsto r \mid l = r \in E\}$ is a term rewriting system (i.e. conditions (i) and (ii) in definition 8.2.1 are met), we call R the **term rewriting system defined by** E. In this case we will write $t \to_E t'$ instead of $t \to_R t'$ 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 $r \mapsto l$.

8.2.3 Example

Let again $E := \{\ x + 0 = x,\ x + \mathsf{succ}(y) = \mathsf{succ}(x + y)\ \}$. Clearly E is a term rewriting system. Let us rewrite the term $t :\equiv 0 + (0 + \mathsf{succ}(0))$. We have to find a subterm of t that matches the left hand side of a rule in E. $0 + \mathsf{succ}(0)$ is such a subterm, since $0 + \mathsf{succ}(0) \equiv (x + \mathsf{succ}(y))\{0/x,\, 0/y\}$. We mark the occurrence of this subterm in t by underlining it: $0 + (\underline{0 + \mathsf{succ}(0)})$. Now we replace this instance of $x + \mathsf{succ}(y)$ by the corresponding instance of $\mathsf{succ}(x + y)$, i.e. by $\mathsf{succ}(x + y)\{0/x,\, 0/y\} \equiv \mathsf{succ}(0 + 0)$, and obtain $0 + \mathsf{succ}(0 + 0)$. Hence

$$0 + (\underline{0 + \mathsf{succ}(0)}) \to_E 0 + \mathsf{succ}(0 + 0)$$

The term $0 + \mathsf{succ}(0 + 0)$ can again be rewritten by replacing the subterm $0 + 0$, which matches the left hand side of the rule $x + 0 \mapsto 0$:

$$0 + \mathsf{succ}(\underline{0 + 0}) \to_E 0 + \mathsf{succ}(0)$$

Furthermore $0 + \mathsf{succ}(0)$ rewrites to $\mathsf{succ}(0 + 0)$ and the latter to $\mathsf{succ}(0)$

$$\underline{0 + \mathsf{succ}(0)} \to_E \underline{\mathsf{succ}(0 + 0)} \to_E \mathsf{succ}(0)$$

Hence we have $0 + (0 + \mathsf{succ}(0)) \to_E^* \mathsf{succ}(0)$.

Exercise: write a reduction sequence showing that $\mathsf{succ}(0 + 0) + 0 \to_E^* \mathsf{succ}(0)$.

Obviously, the terms in normal form w.r.t. E are precisely the terms $\mathsf{succ}^n(0)$ ($n \in \mathbf{N}$). In particular $\mathsf{succ}(0)$ is in normal form.

We saw that $0 + (0 + \mathsf{succ}(0))$ and $\mathsf{succ}(0 + 0) + 0$ both formalize to $\mathsf{succ}(0)$. Hence we have $0 + (0 + \mathsf{succ}(0)) \simeq_E \mathsf{succ}(0 + 0) + 0$.

Exercise: normalize $0 + (0 + \mathsf{succ}(0))$ to $\mathsf{succ}(0)$ using a different reduction sequence.

8.2.4 Lemma

The relation $\simeq_E$ is a congruence on the Σ-algebra $\mathrm{T}(\Sigma, X)$ which in addition is closed under substitutions, i.e. if $t \simeq_E t'$ then $t\theta \simeq_E t'\theta$ for every substitution θ.

Proof. Obviously $\simeq_E$ is an equivalence relation. In order to show that $\simeq_E$ is compatible with the operation in Σ, (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

$$t_i \to_E t_i' \quad \Rightarrow \quad f(t_1, \ldots, t_i, \ldots, t_n) \to_E f(t_1, \ldots, t_i', \ldots, t_n)$$

Similarly in order to show that $\simeq_E$ is closed under substitution it suffices to observe that $\to_E$ is closed under substitution, i.e.

$$t \to_E t' \quad \Rightarrow \quad t\theta \to_E t'\theta$$

8.2.5 Theorem

$$\vdash_E t = t' \qquad \Longleftrightarrow \qquad t \simeq_E t'$$

Proof. '$\Rightarrow$' is proved by induction on the derivation of $\vdash_E t = t'$. Having lemma 8.2.4 at hand the proof is trivial.

'$\Leftarrow$'. Because of the derivation rules *Reflexivity*, *Symmetry*, and *Transitivity*, it obviously suffices to show

$$t \to_E t' \quad \Rightarrow \quad \vdash_E t = t'$$

But this is easy using the rules *Compatibility*, and *Instance*.

8.2.6 Corollary

$$\forall E \models \forall(t = t') \qquad \Longleftrightarrow \qquad t \simeq_E t'$$

Proof. Theorems 8.1.4, 8.1.5, and 8.2.5.

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 (8.1.4, 8.1.5) 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 Σ 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 $\vdash_E t = t' \quad \Leftrightarrow \quad t \simeq_E t'$ proved in Theorem 8.2.5 provides an optimization of this algorithm, by replacing the 'blind' search for $\vdash_E t = t'$ by a 'goal directed' search for $t \simeq_E t'$.

8.3 Termination

8.3.1 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.

8.3.2 Example

Let $R := \{\ x + 0 \mapsto x,\ x + \mathsf{succ}(y) \mapsto \mathsf{succ}(x + y),\ 0 + y \mapsto y + 0\ \}$. 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.

E Noether (1882 - 1935)

8.3.3 Lemma

Let R be a term rewriting system over a signature Σ.

Let μ be a function mapping Σ-terms to natural numbers such that

$$t \to_R t' \quad \Rightarrow \quad \mu(t) > \mu(t')$$

Then R is terminating.

Proof. If we had an infinite reduction sequence $t_0 \to_R t_1 \to_R \ldots$, we would get an infinite decreasing sequence of natural numbers $\mu(t_0) > \mu(t_1) > \ldots$, which is impossible.

8.3.4 Example

$R := \{\ x - 0 \mapsto x,\ \mathsf{succ}(x) - \mathsf{succ}(y) \mapsto x - y\ \}$. Set $\mu(t) :=$ the length of t. Then clearly $\mu(t) > \mu(t')$ whenever $t \to_R t'$. Hence R is terminating according to lemma 8.3.3.

8.3.5 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

$$f(g(x), g(x)) \to_R f(g(x), g(x)) \to_R \ldots$$

The following lemma clarifies the situation.

8.3.6 Lemma

Let R be a term rewriting system over a signature Σ such for every rule $l \mapsto r$ in R

> r is shorter than l,
>
> every variable $x \in \mathsf{FV}(r)$ 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 $t \to_R t'$ then t' is shorter than t.

Unfortunately, the applicability of Lemma 8.3.6 is rather restricted. For example for the term rewriting system $R := \{\ x + 0 \mapsto x,\ x + \mathsf{succ}(y) \mapsto \mathsf{succ}(x + y)\ \}$ 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.

8.3.7 Theorem

Let R be a term rewriting system over a signature Σ.

Let A be a Σ-algebra with $A_s = \mathbf{N}$ for every sort s such that f^A is a strictly monotone function for every operation f in Σ, i.e.

$$n_i > n_i' \quad \Rightarrow \quad f^A(n_1, \ldots, n_i, \ldots, n_k) > f^A(n_1, \ldots, n_i', \ldots, n_k),$$

and such that $l^{A,\alpha} > r^{A,\alpha}$ for every rewrite rule $l \mapsto r \in R$ and every variable assignment α.

Then R is terminating.

Proof. We set $\mu(t) := t^{A,\alpha}$, where α is an arbitrary variable assignment (e.g. $\alpha(x) := 0$ for all variables x). By Lemma 8.3.3 it suffices to show that $\mu(t) > \mu(t')$ whenever $t \to_R t'$.

To this end we first show that for any Σ-term t and $x \in \mathsf{FV}(t)$

$$n > m \quad \Rightarrow \quad t^{A,\alpha_x^n} > t^{A,\alpha_x^m} \tag{+}$$

We prove (+) by induction on t.

Base: $t \equiv x$. $x^{A,\alpha_x^n} = n > m = x^{A,\alpha_x^m}$.

Step: $t = f(t_1, \ldots, t_k)$. Let $n_i := t_i^{A,\alpha_x^n}$, and $m_i := t_i^{A,\alpha_x^m}$. By induction hypothesis $n_i > m_i$ if $x \in \mathsf{FV}(t_i)$, and this the case at least for one $i \in \{1, \ldots, k\}$, and, of course $n_i = m_i$ if $x \notin \mathsf{FV}(t_i)$. Hence, because f^A is strictly monotone,

$$f(t_1,\dots,t_k)^{A,\alpha_x^n} = f^A(n_1,\dots,n_k) > f^A(m_1,\dots,m_k) = f(t_1,\dots,t_k)^{A,\alpha_x^m}$$

Having proved (+), can now easily complete the proof. Assume $t \to_R t'$. 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 $\{l\theta/x\}^{A,\alpha} = \alpha_x^n$ and therefore, using the Substitution Lemma 3.5.4

$$\mu(t) = t^{A,\alpha} = (u\{l\theta/x\})^{A,\alpha} = u^{A,\{l\theta/x\}^{A,\alpha}} = u^{A,\alpha_x^n}$$

Similarly $\mu(t') = u^{A,\alpha_x^m}$, 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 3.5.4 once more and using the assumption on R:

$$n = l^{A,\theta^{A,\alpha}} > r^{A,\theta^{A,\alpha}} = m$$

8.3.8 Example

Let us use Theorem 8.3.7 to prove termination of the term rewriting system $R := \{\ x + 0 \mapsto x,\ x + \mathsf{succ}(y) \mapsto \mathsf{succ}(x + y)\ \}$.

We define the algebra A by setting

$$0^A := 1$$

$$\mathsf{succ}^A(n) := n + 1$$

$$n +^A m := n + 2 * m$$

Then, with $n := \alpha(x)$ and $m := \alpha(y)$, we have

$$(x + 0)^{A,\alpha} = n +^A 0^A = n + 2 * 1 > n = x^{A,\alpha}$$

$$(x + \mathsf{succ}(y))^{A,\alpha} = n +^A \mathsf{succ}^A(m) = n + 2 * (m + 1) > n + 2 * m + 1 = \mathsf{succ}(x + y)^{A,\alpha}$$

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 8.3.3) is complete in the sense that for every terminating term rewriting system such a measure exists.

8.3.9 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 labelled by t,

the children of a node labelled by u are labelled by the terms u' such that $u \to_R u'$.

Clearly the immediate subtrees of the reduction tree of t are precisely the reduction trees of the terms t' with $t \to_R t'$.

8.3.10 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 $t_0 \to_R t'$. Hence there must be a term t_1 such that $t_0 \to_R t_1$ 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 $t_1 \to_R t_2$ and the reduction tree of t_2 is infinite. Continuing in this way we obtain an infinite reduction sequence $t_0 \to_R t_1 \to_R t_2 \to_R \ldots$.

8.3.11 Definition

Let R be a terminating term rewriting system. For every term t we set

$$\begin{aligned} \sharp(t) \quad &:= \quad \text{height of the reduction tree of } t \\ &= \quad \max\{n \in \mathbf{N} \mid t \text{ starts a reduction sequence of length } n\} \end{aligned}$$

$\sharp(t)$ is called the **reduction height** of t. Obviously

$$\sharp(t) = \begin{cases} 0 & \text{if } t \text{ is in normal form} \\ 1 + \max\{\sharp(t') \mid t \to_R t'\} & \text{otherwise} \end{cases}$$

In particular $t \to_R t'$ implies $\sharp(t') < \sharp(t)$.

The tacitly assumed condition on the term rewriting system R to be finite is necessary, as the following example shows.

8.3.12 Example

Consider the *infinite* (!) term rewriting system

$$R := \{\ x + 0 \mapsto x,\ x + \mathsf{succ}(y) \mapsto \mathsf{succ}(x + y)\ \} \cup \{\ c \mapsto 0 + \mathsf{succ}^n(0)\ \mid n \in \mathbf{N}\ \}$$

It is easy to see that R is terminating, but the term c starts arbitrarily long reduction sequences, namely

$$c \to_R 0 + \mathsf{succ}^n(0) \to_R \mathsf{succ}(0) + \mathsf{succ}^{n-1}(0) \to_R \ldots \to_R \mathsf{succ}^n(0) + 0 \to^n_{\mathsf{succ}} (0)$$

for every $n \in \mathbf{N}$.

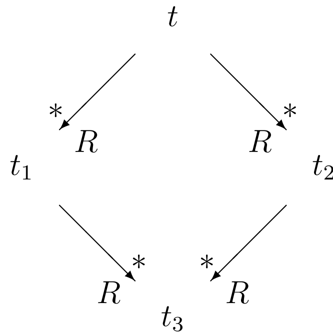
Remark. It is undecidable whether or not a term rewriting system R is terminating. This can be seen, for example, by representing each Turing machine T by a term rewriting systems R_T in such a way that the T halts if and only R_T terminates, thus reducing the halting problem for Turing machines, which is well-known to be undecidable, by the termination problem for term rewriting systems. However, due to König's Lemma 8.3.10, 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 8.3.7 being just one of simplest.

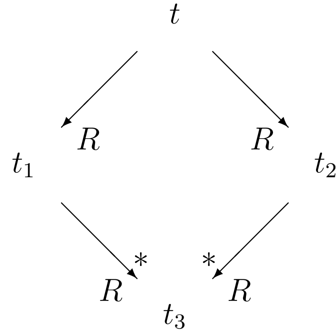
8.4 Confluence

8.4.1 Definition

1. A term rewriting system R over a signature Σ is **confluent** if for all Σ-terms t, t_1, t_2 such that $t \to^*_R t_1$ and $t \to^*_R t_2$ there exists a Σ-term t_3 such that $t_1 \to^*_R t_3$ and $t_2 \to^*_R t_3$.



2. A term rewriting system R over a signature Σ is **locally confluent** if for all Σ-terms t, t_1, t_2 such that $t \to_R t_1$ and $t \to_R t_2$ there exists a Σ-term t_3 such that $t_1 \to_R^* t_3$ and $t_2 \to_R^* t_3$.



8.4.2 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 $a \to_R c$ and $a \to_R^* d$, 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.

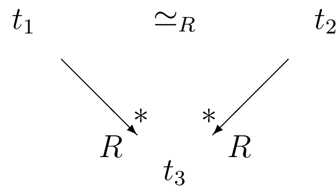
(c) Is R terminating?

8.4.3 Theorem

Let R be a confluent term rewriting system over a signature Σ.

Then R has the following so-called *Church-Rosser property*: For all Σ-terms t_1, t_2

$$t_1 \simeq_R t_2 \quad \Leftrightarrow \quad \text{there exists a } \Sigma\text{-term } t_3 \text{ such that } t_1 \to_R^* t_3 \text{ and } t_2 \to_R^* t_3$$



Proof. '⇒' is obvious.

'⇒' is easily proved by induction on n, where $t_1 \equiv u_0 \leftrightarrow_R \ldots \leftrightarrow_R u_n \equiv t_2$.

8.4.4 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

$$t \to_R^* t_1 \text{ and } t \to_R^* t_2 \quad \Rightarrow \quad \text{there exists a } \Sigma\text{-term } t_3 \text{ such that } t_1 \to_R^* t_3 \text{ and } t_2 \to_R^* t_3$$

by induction on $\sharp(t)$ (cf. definition 8.3.11; $\sharp(t)$ exists because R is assumed to be terminating). So, assume $t \to_R^* t_1$ and $t \to_R^* t_2$. 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 $t \to_R t_i'$ and $t_i' \to_R^* t_i$. (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 $t_1' \to_R^* t_3'$ and $t_2' \to_R^* t_3'$. Furthermore, since $\sharp(t_i') < \sharp(t)$, we know, by induction hypothesis, that there are terms u_1, u_2 with $t_i \to_R^* u_i$ and $t_3' \to_R^* u_i$, for $i = 1, 2$. Using the induction hypothesis once more, now with t_3' (we have $\sharp(t_3') \le \sharp(t_1') < \sharp(t)$) we conclude that there is a term t_3 such that $u_1 \to_R^* t_3$ and $u_2 \to_R^* t_3$.

8.4.5 Lemma

Let R be a confluent and terminating term rewriting system over a signature Σ. Then every Σ-term t has a unique normal form.

Proof. Since R is terminating t has a normal form t', i.e. $t \to_R^* t'$ and t' is in normal form. If also $t \to_R^* t''$ 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 $t' \equiv t''$.

8.4.6 Definition

Let R be a confluent and terminating term rewriting system over a signature Σ. Then every for every Σ-term t we denote by

$$\text{nf}(t)$$

the unique normal form of t.

8.4.7 Theorem

Let E be a system of equations over a signature Σ defining confluent and terminating term rewriting system . Then for all Σ-terms t_1, t_2

$$\forall E \models t_1 = t_2 \quad \Leftrightarrow \quad \text{nf}(t_1) = \text{nf}(t_2)$$

In particular, the relation $\forall E \models t_1 = t_2$ is decidable.

Proof. Clearly $t_1 \simeq_R t_2$ if and only if $\mathrm{nf}(t_1) = \mathrm{nf}(t_2)$. The result follows with Theorem 8.2.5.

Theorem 8.4.7 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 equation 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.

8.4.8 Definition

A **unifier** of two terms t_1, t_2 is a substitution $\theta\colon \mathsf{FV}(t_1) \cup \mathsf{FV}(t_2) \to \mathrm{T}(\Sigma, V)$ (V the set of all variables) such that

$$t_1\theta \equiv t_2\theta$$

A **most general unifier** of t_1, t_2 is a unifier θ of t_1, t_2 with the additional property that for any other unifier θ' of t_1, t_2 there exists a substitution σ 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.

8.4.9 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 $y+\mathsf{succ}(z) \mapsto \mathsf{succ}(y+z)$ is a variant of $x+\mathsf{succ}(y) \mapsto \mathsf{succ}(x+y)$, but $z+\mathsf{succ}(z) \mapsto \mathsf{succ}(z+z)$ is not.

8.4.10 Definition

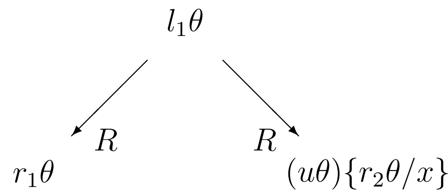
Let R be a term rewriting system over a signature Σ and $l_i \mapsto r_i$, $i = 1, 2$, be variants of rules in R such that $\mathsf{FV}(l_1) \cap \mathsf{FV}(l_2) = \emptyset$.

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 $l_1 \equiv u\{t/x\}$, where x is a fresh variable occurring exactly once in u, and there is a most general unifier θ of t and l_2, i.p. $t\theta \equiv l_2\theta$. 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 $\mathrm{CP}(R)$ denote the set of all critical pairs of R.

Note that $\mathrm{CP}(R)$ is a finite set that can be easily computed from R.



8.4.11 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.

8.4.12 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 8.4.11 and Newman's Lemma 8.4.4.

Remark. For arbitrary term rewriting systems confluence is undecidable (see e.g. Baader/Nipkow).

Remark. Theorem 8.4.12 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 CP(R). If for all $(t_1, t_2) \in$ 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 8.4.12).

2. For any $(t_1, t_2) \in$ 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 **Knuth-Bendix completion algorithm** mentioned earlier.

8.5 Rapid prototyping

Rapid Prototyping is the process of automatically generating an implementation, called *rapid prototype*, of an ADT given by an initial specification whose associated term rewriting is confluent and terminating (if the term rewriting is not confluent, it can often be automatically made so by applying the Knuth-Bendix completion algorithm. Although the automatically generated prototype is usually not efficient it can be very useful for detecting inadequacies of a specification at an earlier stage of a software development.

8.5.1 Definition

The term rewriting system associated with an initial specification Init–Spec(Σ, E) is the term rewriting system defined by E, provided, of course, E defines a term rewriting system (cf. Definition 8.2.2).

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).

8.5.2 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.

Algebra	$\mathrm{NF}_E(\Sigma)$
Carriers	$\mathrm{NF}_E(\Sigma)_s := \{t \in \mathrm{T}(\Sigma) \mid t \text{ in normal form } \}$
Constants	$c^{\mathrm{NF}_E(\Sigma)} := \mathrm{nf}(c)$
Operations	$f^{\mathrm{NF}_E(\Sigma)}(t_1, \ldots, t_n) := \mathrm{nf}(f(t_1, \ldots, t_n))$

8.5.3 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 $\mathrm{NF}_E(\Sigma)$ of closed normal terms is a model of Init–Spec(Σ, E).

Furthermore for every closed Σ-term t

$$t^{\mathrm{NF}_\Sigma(E)} = \mathrm{nf}(t)$$

Proof. In order to show that $\mathrm{NF}_E(\Sigma)$ is a model of Init–Spec(Σ, E) it suffices to prove that $\mathrm{NF}_E(\Sigma)$ is isomorphic to $\mathrm{T}_E(\Sigma)$. Obviously the identical embedding of $\mathrm{NF}_E(\Sigma)$ into $\mathrm{T}_E(\Sigma)$ is an isomorphism.

Furthermore $[\mathrm{nf}]_: \mathrm{T}_E(\Sigma) \to \mathrm{NF}_E(\Sigma)$ defined by $[\mathrm{nf}]_([t]_) := \mathrm{nf}(t)$ clearly is a well-defined homomorphism. Because evaluation of terms defines another homomorphism from $\mathrm{T}_E(\Sigma)$ to $\mathrm{NF}_E(\Sigma)$ (cf. 6.2.3) we have, by initiality of $\mathrm{T}_E(\Sigma)$, that $t^{\mathrm{NF}_\Sigma(E)} = \mathrm{nf}(t)$.

The following theorem provides a useful criterion for testing whether an initial specification is adequate for a given algebra A.

8.5.4 Theorem

Let A be a Σ-algebra and Init–Spec(Σ, E) an initial specification defining a terminating and confluent term rewriting system.

Then Init–Spec(Σ, E) is adequate for A (i.e. A is a model of Init–Spec(Σ, E)) iff

(i) Every element $a \in A_s$ is the value of a unique closed normal Σ-term.

(ii) $f^A(t_1^A, \ldots, t_n^A) = (\mathrm{nf}(f(t_1, \ldots, t_n)))^A$ for every operation $f\colon s_1 \times \ldots \times s_n \to s$ and all closed normal terms t_i of sort s_i, $i = 1, \ldots, n$.

Proof. By (ii) evaluation of closed normal Σ-terms in A is a homomorphism from $\mathrm{NF}_E(\Sigma)$ to A, and by (i) this homomorphism is bijective, i.e. an isomorphism. Since A is isomorphic to $\mathrm{NF}_E(\Sigma)$, and by Theorem 8.5.3, $\mathrm{NF}_E(\Sigma)$ is a model of Init–Spec(Σ, E) it follows that A is a model of Init–Spec(Σ, E), too.

On the other hand if A is a model of Init–Spec(Σ, E), then A must be isomorphic to $\mathrm{NF}_E(\Sigma)$, and hence (i) and (ii) hold.

8.5.5 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 $\mathrm{low}(n, l)$ selects from the list l the list of those elements that are $\leq n$, and $\mathrm{high}(n, l)$ 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.

Init Spec QUICKSORT

Sorts nat, boole, natlist

Constants
$0\colon$ nat
$\mathrm{T}\colon$ boole
$\mathrm{F}\colon$ boole
$\mathrm{nil}\colon$ natlist

Operations
succ: nat $\to$ nat
$\mathrm{cons}\colon$ nat $\times$ natlist $\to$ natlist
$\mathrm{if}\colon$ nat $\times$ natlist $\times$ natlist $\to$ natlist
$<\colon$ nat $\times$ nat $\to$ boole
$@\colon$ natlist $\times$ natlist $\to$ natlist
$\mathrm{low}\colon$ nat $\times$ natlist $\to$ natlist
$\mathrm{high}\colon$ nat $\times$ natlist $\to$ natlist
$\mathrm{sort}\colon$ natlist $\to$ natlist

Variables $x, y\colon$ nat, $l, l_1, l_2\colon$ natlist

Equations
$\mathrm{if}(\mathrm{T}, l_1, l_2) = l_1$
$\mathrm{if}(\mathrm{F}, l_1, l_2) = l_2$

$x < x = \mathrm{F}$
$0 < \mathsf{succ}(x) = \mathrm{T}$
$\mathsf{succ}(x) < 0 = \mathrm{F}$
$\mathsf{succ}(x) < \mathsf{succ}(y) = x < y$

$\mathrm{nil} \,@\, l = l$
$\mathrm{cons}(x, l_1) \,@\, l_2 = \mathrm{cons}(x, l_1 \,@\, l_2)$

$\mathrm{low}(x, \mathrm{nil}) = \mathrm{nil}$
$\mathrm{low}(x, \mathrm{cons}(y, l)) = \mathrm{if}(y < x, \mathrm{low}(x, l), \mathrm{cons}(y, \mathrm{low}(x, l)))$

$\mathrm{high}(x, \mathrm{nil}) = \mathrm{nil}$
$\mathrm{high}(x, \mathrm{cons}(y, l)) = \mathrm{if}(x < y, \mathrm{cons}(y, \mathrm{high}(x, l)), \mathrm{high}(x, l))$

$\mathrm{sort}(\mathrm{nil}) = \mathrm{nil}$
$\mathrm{sort}(\mathrm{cons}(x, l)) = \mathrm{sort}(\mathrm{low}(x, l)) \,@\, \mathrm{cons}(x, \mathrm{sort}(\mathrm{high}(x, l)))$

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 $(\mathrm{F}, x <$

x), which is generated by the first and last rule for $<$. Since $x < x \to_R$ F we conclude with Theorem 8.4.12 that R is confluent.

It follows with Theorem 8.5.3 that QUICKSORT is adequate fort A.

If we are unable to prove that a given initial specification is confluent and terminating, the following theorem might be useful.

8.5.6 Theorem

Let E be a system of equations over a signature Σ defining a term rewriting system such that every Σ-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(Σ, E) is adequate for A.

Proof. By Theorem 6.2.5 (iv) it suffices to show that for any two closed Σ-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 $\vdash_E t_i = u_i$.

8.6 Summary and Exercises

The central notions and results of this section were the following.

- The *deduction rules* of *equational logic* (8.1.1);
- the notion of a *term rewriting system* R and associated with it the relations

  $t \to_R t'$

  $t \to_R^* t'$

  $t \leftrightarrow_R t'$

  $t \simeq_R t'$

  and the notion of a term in *normal form* (8.2.1);
- the term rewriting system associated with a system of equations (8.2.2)
- the *Soundness Theorem* (8.1.4) and *Birkhoff's Completeness Theorem* (8.1.5) for equational logic; together with 8.2.5 they yield the equivalences

  $$\forall E \models \forall(t = t') \quad \Longleftrightarrow \quad \vdash_E t = t' \quad \Longleftrightarrow \quad t \simeq_E t'$$
- the property of *termination* (8.3.1) and some simple techniques for *proving termination* (8.3.3, 8.3.6, 8.3.7);
- the property of *confluence* (8.4.1);

- the *normal form of a term* $\mathrm{nf}(t)$ w.r.t a confluent and terminating term rewriting system (8.4.6);
- *the term rewriting system associated with an initial specification*;
- *rapid prototyping* (8.5.1), 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 Init–Spec(Σ, 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

$$\forall E \models t_1 = t_2 \qquad \Longleftrightarrow \qquad \mathrm{nf}(t_1) \equiv \mathrm{nf}(t_2)$$

  (8.4.7); 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 (8.5.2, 8.5.3)

Exercises.

1. Consider the following initial specification

Init Spec	DH
Sorts	rat
Constants	one: rat
Operations	double: rat $\to$ rat half: rat $\to$ rat
Variables	x: rat
Equations	half(double(x)) $= x$ double(half(x)) $= x$

Let Σ be the signature of DH. Let $\mathbf{Q}^+$ be the Σ-algebra of positive rational numbers with the obvious interpretation of the constant and the operations.

(a) Show that $\mathbf{Q}^+$ is not a model of DH.

(b) Describe a subalgebra of $\mathbf{Q}^+$ that is a model of DH.

(c) Describe the closed Σ-terms that are in normal form with respect to the term rewriting system associated with DH.

(d) Construct a model A of DH with $A_{\text{rat}} = \mathbf{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$ and $\text{one}^A := 0$.

2. Consider the initial specification

Init Spec	I
Sorts	nat, boole
Constants	$0\colon$ nat, $\mathrm{T}\colon$ boole, $\mathrm{F}\colon$ boole
Operations	succ: nat $\to$ nat iszero: nat $\to$ boole
Variables	$x\colon$ nat
Equations	$\mathsf{succ}(\mathsf{succ}(0)) = 0$ $\mathrm{iszero}(0) = \mathrm{T}$ $\mathrm{iszero}(\mathsf{succ}(x)) = \mathrm{F}$

Let Σ be the signature of I, and E the set of equations of I.

(a) Describe the closed Σ-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 $\vdash_E \mathrm{T} = \mathrm{F}$.

(c) Show that $\vdash_E \mathsf{succ}(\mathsf{succ}(x)) = x$ does not hold.

(d) Construct a model of I. How many elements do its carriers contain?

3. Consider the signature $\Sigma := (\{s\}, \{c\colon s,\ f\colon s \to s\})$ and the term rewriting system $R := \{f(f(x)) \mapsto c\}$ over Σ.

Show that R is terminating, but not confluent.

4. Investigate whether the initial specification of queues in Exercise 9 of Section 6.7 can be used for rapid prototyping.

9 Programs from proofs

In Chapter 3 we distinguished between classical, intuitionistic and minimal logic. It seemed that classical logic, which includes the use of the rule reductio-ad-absurdum, is the 'right' logic, because it is with respect to this logic that Gödel's Completeness Theorem 4.3.2, asserting a perfect match between (algebraic) logical validity and formal provability, holds. On the other hand, the Completeness Theorem does not hold for intuitionistic logic. For example, Peirce's law, $((P \to Q) \to P) \to P$, is logically valid, but it is not provable in intuitionistic logic. (other examples of logically valid, but intuitionistically unprovable formulas are $\neg\neg P \to P$, $P \vee \neg P$, $\neg\forall x\, P(x) \to \exists x \neg P(x)$ and many others).

Despite its apparent deficiencies, intuitionistic logic is very important historically and philosophically, and has nowadays great impact on cutting edge developments in Theoretical Computer Science.

Historically, intuitionistic logic can be seen as an attempt to overcome the crisis in the foundation of Mathematics caused by the discovery of set-theoretic paradoxes in the beginning of the last century (e.g. Russell's paradox). Philosophically, intuitionism has emerged from a critique of reasoning methods used in classical mathematics.

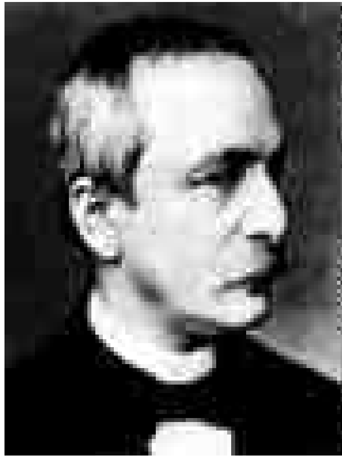
In *classical logic* usually a *Platonistic* view of the mathematical universe is adopted. According to this view, infinite entities, like, for example, the set of natural numbers, are assumed to exist in some 'ideal world', like finite entities exist in the real world. Consequently, any mathematical statement, represented by a closed formula P say, is either true or false in this world. We did adopt this view in the previous chapter when we introduced algebras as (fragments of) the mathematical universe and defined when a formula is true or false in an algebra. Looking at one of the controversial formulas, like $P \vee \neg P$, we see that it is clearly valid classically, since in any algebra either P is true or $\neg P$ is true.

Intuitionists reject the platonistic view of Mathematics as intuitively unjustified. They rather see Mathematics as a system of *mental constructions*. Consequently, in intuitionistic logic only such axioms and proof rules are accepted that can be directly justified as providing mental constructions for evidence of a formula. For example, in order to accept $P \vee \neg P$ as an axiom for every formula P, an intuitionist would demand a uniform construction that for every formula P and any interpretation of the symbols in P yields either a proof of P or its negation. Since no such construction is known, this axiom is rejected. By a similar argument the rule reductio-ad-absurdum is rejected.

Some of the most important historical figures of intuitionism are Kronecker, Brouwer, Heyting, Kolmogorov, and also Gödel. I recommend to read Heyting's book *Intuitionism - An introduction* [Hey] for a very entertaining introduction into the ideas of intuitionism: In its introductory chapter Heyting lets fictive protagonists of classical, intuitionistic, and other schools in logic have a controversial dispute on their standpoints. An excellent in depth introduction to intuitionism provides the book *Constructivism in Mathematics, Vol. I*, by A S Troelstra and D van Dalen [TD]. Also very useful is chapter 5 in van Dalen's text book [Dal].

In this chapter (of this course) we will study how the constructive aspect of intuitionistic logic can be exploited to automatically synthesise correct programs from intuitionistic proofs. 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*.


L Kronecker (1823 - 1891)

L E J Brouwer (1881 - 1966)

A Heyting (1898 - 1980)

A Kolmogorov (1903 - 1987)

9.1 Formulas as data types

Let us, for the moment, identify the notion of a 'data type' with the notion of a 'set'. On data types A, B we have the following familiar constructions:

$$\begin{aligned}
A \times B &:= \{(a,b) \mid a \in A,\ b \in B\} \quad \text{(cartesian product)}\\
A + B &:= \{(0,a) \mid a \in A\} \cup \{(1,b) \mid b \in B\} \quad \text{(disjoint sum)}\\
A \to B &:= \{f \mid f\colon A \to B\} \quad \text{(function type)}
\end{aligned}$$

Figure 6 shows how each constructor for formulas corresponds to a data type construction (assuming the sort s of the variable x corresponds to the data type D).

Formula		Data Type	
conjunction	$P \wedge Q$	$A \times B$	cartesian product
implication	$P \to Q$	$A \to B$	function type
disjunction	$P \vee Q$	$A + B$	disjoint sum
for all	$\forall x\, P(x)$	$D \to A$	function type
exists	$\exists x\, P(x)$	$D \times A$	cartesian product
equations	$s = t$	$\{*\}$	a singleton set
falsity	$\bot$	$\{\}$	the empty set

Figure 6: The formulas-as-types correspondence

9.1.1 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

$$\mathbf{N} \to \mathbf{N} \times (\{*\} + \{*\})$$

Since $\{*\} + \{*\}$ $(= \{(0,*), (1,*)\})$ is a set with two (different) elements we may replace it by the type $\mathbf{B}$ of boolean values. Therefore the formula above corresponds to the data type

$$\mathbf{N} \to \mathbf{N} \times \mathbf{B}$$

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 $\langle y, \mathrm{T}\rangle$, this means that x is even and $x = y + y$, whereas an output $\langle y, \mathrm{F}\rangle$ means that x is odd and $x = y + y + 1$.

9.2 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 9.1.1. The definition of proof terms is given in figure 7. 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.

9.2.1 Example

Consider the proof

$$\dfrac{\dfrac{\dfrac{P \wedge Q}{Q}\wedge_{\mathrm{r}}^- \qquad \dfrac{P \wedge Q}{P}\wedge_{\mathrm{l}}^-}{Q \wedge P}\wedge^+}{(P \wedge Q) \to (Q \wedge P)}\to^+$$

Written with proof terms this reads:

$$\dfrac{\dfrac{\dfrac{u\!:\!P \wedge Q}{\pi_{\mathrm{r}}(u) : Q}\wedge_{\mathrm{r}}^- \qquad \dfrac{u\!:\!P \wedge Q}{\pi_{\mathrm{l}}(u) : P}\wedge_{\mathrm{l}}^-}{\langle \pi_{\mathrm{r}}(u), \pi_{\mathrm{l}}(u)\rangle : Q \wedge P}\wedge^+}{\lambda u\!:\!P \wedge Q.\langle \pi_{\mathrm{r}}(u), \pi_{\mathrm{l}}(u)\rangle : (P \wedge Q) \to (Q \wedge P)}\to^+$$

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

$$\lambda u\!:\!P \wedge Q.\langle \pi_{\mathrm{r}}(u), \pi_{\mathrm{l}}(u)\rangle$$

9.2.2 Exercises

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

$$\dfrac{\dfrac{\dfrac{\dfrac{P \wedge Q \to R \qquad \dfrac{P \qquad Q}{P \wedge Q}\wedge^+}{R}\to^-}{Q \to R}\to^+}{P \to (Q \to R)}\to^+}{(P \wedge Q \to R) \to (P \to (Q \to R))}\to^+$$

assumption	variable	$u\!:\!P$
$\wedge^+$	pairing	$$\frac{d:P \qquad e:Q}{\langle d,e\rangle : P\wedge Q}\ \wedge^+$$
$\wedge^-$	projections	$$\frac{d:P\wedge Q}{\pi_{\mathrm{l}}(d):P}\ \wedge^-_{\mathrm{l}} \qquad \frac{d:P\wedge Q}{\pi_{\mathrm{r}}(d):Q}\ \wedge^-_{\mathrm{r}}$$
$\to^+$	abstraction	$$\frac{d:Q}{\lambda u\!:\!P.d : P\to Q}\ \to^+$$
$\to^-$	procedure call	$$\frac{d:P\to Q \qquad e:P}{(de):Q}\ \to^-$$
$\vee^+$	injections	$$\frac{d:P}{\mathrm{inl}_Q(d):P\vee Q}\ \vee^+_{\mathrm{l}} \qquad \frac{d:Q}{\mathrm{inr}_P(d):P\vee Q}\ \vee^+_{\mathrm{r}}$$
$\vee^-$	case analysis	$$\frac{d:P\vee Q \qquad e_1:P\to R \qquad e_2:Q\to R}{\mathrm{cases}[d,e_1,e_2]:R}\ \vee^-$$
$\forall^+$	abstraction	$$\frac{d:P(x)}{\lambda x.d:\forall x\,P(x)}\ \forall^+ \qquad (*)$$
$\forall^-$	procedure call	$$\frac{d:\forall x\,P(x)}{(dt):P(t)}\ \forall^-$$
$\exists^+$	pairing	$$\frac{d:P(t)}{\langle t,d\rangle:\exists x\,P(x)}\ \exists^+$$
$\exists^-$	matching	$$\frac{d:\exists x\,P(x) \qquad e:\forall x\,(P(x)\to Q)}{\mathrm{match}[d,e]:Q}\ \exists^- \qquad (**)$$
induction	recursion	$$\frac{d:P(0) \qquad e:\forall x\,(P(x)\to P(x+1))}{\mathrm{ind}[d,e]:\forall x\,P(x)}\ \mathrm{ind}$$

Figure 7: Natural deduction with proof terms

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

$$\lambda u{:}P \to (Q \to R)\,.\,\lambda v{:}P \wedge Q\,.\,((u\pi_{\mathrm{l}}(v))\pi_{\mathrm{r}}(v))$$

9.3 Program synthesis from intuitionistic 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, if the proof was intuitionistic, only little modifications and simplification are necessary in order to transform the corresponding 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 summarised in the following theorem:

9.3.1 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 $\vec{x}, \vec{y}$ of variables, and the variables in $\vec{x}$ may be subject to preconditions.

9.3.2 Example (Quotient and remainder)

An example of such a generalised formula is

$$(+) \qquad \forall b\, (b > 0 \to \forall a\, \exists q\, \exists r\, (a = b * q + r \wedge r < b))$$

where the variables range over natural numbers. This formula says that division with remainder 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

$$a = b * q + r \quad \text{and} \quad r < b.$$

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.minlog-system.de`)

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

$$\forall a\, \exists q\, \exists r\, (a = b * q + r \land r < b))$$

by induction on a.

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

Step. We have to prove

$$\forall a\, [\exists q\, \exists r\, (a = b * q + r \land r < b)) \to \exists q_1\, \exists r_1\, (a + 1 = b * q_1 + r_1 \land r_1 < b))]$$

So, let a be given and assume as induction hypothesis:

$$\exists q\, \exists r\, (a = b * q + r \land r < b)$$

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

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

$$u : a = b * q + r \land r < b$$

(formally we use $\exists^-$ backwards followed by $\forall^-$ and $\to^-$ backwards). We need to find q_1 and r_1 such that $a + 1 = b * q_1 + r_1 \land r_1 < b$ (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 $a + 1 = b * q + r + 1$.

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 $a + 1 = b * q + r + 1 = b * q + b = b * (q + 1) + 0$.

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

Intuitively this proof corresponds to the following program:

```
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

$$\frac{d : P(0) \qquad e : \forall x\,(P(x) \rightarrow P(x+1))}{\mathrm{ind}[d,e] : \forall x\, P(x)}\ \mathrm{ind}$$

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 ind$[d,e]$ is a procedure f that is defined from g and h by *primitive recursion*:

$$\begin{aligned} f(0) &= g \\ f(a+1) &= h(a, f(a)) \end{aligned}$$

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

We do not show the proof term, since it would fill several pages. But here is the *program* which is extracted fully automatically from the proof:

```
(define (quotrem-prog n^1)
  ((nat-rec-run (cons 0 0))
   (lambda (n^3)
     (lambda (nat*nat^4)
       (cons (if ((<-run ((plus-run (cdr nat*nat^4)) 1)) n^1)
                 (car nat*nat^4)
                 ((plus-run (car nat*nat^4)) 1))
             (if ((<-run ((plus-run (cdr nat*nat^4)) 1)) n^1)
                 ((plus-run (cdr nat*nat^4)) 1)
                 0))))))
```

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

```
((quotrem-prog 7) 93)
```

```
> (13 . 2)
```

This means

$$93 = 7 * 13 + 2$$

The advantage of program synthesis from proofs over conventional programming can be summarised 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 for syntactical and type correctness, but these checks do not guarantee that the program behaves as it should.

9.4 Program synthesis from classical proofs

We begin with an example showing that we cannot expect theorem 9.3.1 to hold for classical proofs, that is proofs using the rule reductio ad absurdum, in general. We prove classically the following

9.4.1 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 $P \vee \neg P$ where P is the statement "$\sqrt{2}^{\sqrt{2}}$ is rational".

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 $x := \sqrt{2}^{\sqrt{2}}$ and $y := \sqrt{2}$. Now again x and y are irrational and we have

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

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 form of program synthesis from proofs for classical logic.

9.4.2 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.

Proof (sketch) The proof of this theorem proceeds (roughly) in the following steps:

1. The classical proof of $\forall x\, \exists y\, R(x, y)$ is first transformed into a classical proof of $\exists y\, R(x, y)$ (by one application of $\forall^-$) and is then transformed in to a minimal-logical proof of $\neg\neg\exists y\, R(x, y)$, that is

$$(\exists y\, R(x, y) \to \bot) \to \bot$$

("It's impossible that there doesn't exist y with $R(x, y)$"). The second transformation is due to Gödel and is called *negative translation*.

2. Since in minimal logic neither efq nor raa are used, the symbol $\bot$ has no special meaning and can therefore be replaced by any formula (without spoiling the proof). Replacing $\bot$ by the formula $\exists y\, R(x, y)$ we obtain an intuitionistic proof of

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

and from this we trivially obtain an intuitionistic proof of

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

3. Now we apply Theorem 9.3.1 to obtain a program p satisfying $\forall x\, R(x, p(x))$.

Note that theorem 9.4.2 does not apply to the last example, because the statement "x^y is irrational", when formalized does contain quantifiers.

9.5 Applications

Presently research groups at many Universities pursue the approach of program synthesis from proofs and a number of implementations of theorem provers supporting this technique have been developed. For example: *Agda* (Coquand, Gothenburg, [Agd]), *Coq* (Huet, INRIA, [Coq]), *Fred* (Crossley, Melbourne, [Cro]),

Isabelle (Paulson, Cambridge, [Isa]),

Minlog (Schwichtenberg, Munich, [Min]),

PX (Hayashi, Kyoto, [PX]).

Nuprl (Constable, Cornell, [Con]).

Several important programs have been obtained by program synthesis. For example:

- Efficient algorithms in lambda-calculus and term rewriting.
- Algorithms in Computer Algebra (computation of Gröbner bases).
- Graph-theoretic algorithms.
- Algorithms extracted from theorems in infinitary combinatorics.
- Sorting algorithms.

Although not yet industrially applied the method of program synthesis from proofs is a promising technology that in the future might play an important role in the development of reliable and maintainable software.

9.6 Summary and Exercises

- The difference between classical and intuitionistic logic.
- The correspondence between formulas and data types.
- Proof terms; the correspondence between proofs and programs.
- Program extraction from intuitionistic proofs.
- Program extraction from classical proofs.

Exercises

1. Find the proof term for the following proof:

$$\dfrac{P \lor Q \qquad \dfrac{\dfrac{P}{Q \lor P}\,\lor^+_{\mathrm{r}}}{P \to Q \lor P}\,\to^+ \qquad \dfrac{\dfrac{Q}{Q \lor P}\,\lor^+_{\mathrm{l}}}{Q \to Q \lor P}\,\to^+}{Q \lor P}\,\lor^-$$

2. To which proof does the following proof term correspond?

$$\lambda u\!:\!P \to (Q \to R)\,.\,\lambda v\!:\!P \to Q\,.\,\lambda w\!:\!P\,.\,((uw)(vw))$$

3. Why is the following statement intuitionistically not provable?

"Every Turing machine either halts or doesn't halt."

References

[Agd] `http://www.cs.chalmers.se/~catarina/agda/`

[Ast] E Astesiano et al, CASL: The Common Algebraic Specification Language, Theoretical Computer Science, 2002.

[BaNi] A Baader, T Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.

[BRJ] G Booch, J Rumbaugh, I Jacobson. The Unified Modeling Language User Guide, Addison-Wesley, 1999.

[Bro] Broy et. al., The Requirement and Design Specification Language SPECTRUM, Technical Report, TU-München, 1993.

[Con] Constable et al., Implementing Mathematics with the Nuprl Proof Development System, Prentice–Hall, 1986.

[Coq] Y Bertot, P Casteran, Coq'Art: The Calculus of Inductive Constructions, Interactive Theorem Proving and Program Development, Texts in Theoretical Computer Science. An EATCS Series, `http://coq.inria.fr/doc-eng.html`, 2004.

[Cro] Fred: An Approach to Generating Real, Correct, Reusable Programs from Proofs, J Crossley, I Poernomo, Journal of Universal Computer Science, Vol 7(1), 2001.

[Dal] D van Dalen, Logic and Structure, 3rd ed., Springer, 1994.

[Daw] J Dawes. The VDM-SL Reference Guide, Pitman, 1991.

[Eli] A Eliens, Principles of Object-Oriented Software Development, 2nd ed., Addison Wesley, 2000.

[GH] J V Guttag, J J Horning, Larch: Languages and tools for formal specification, Springer, 1993.

[Hey] A Heyting, Intuitionism - An introduction, North-Holland, 1966.

[Hoa] C A R Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[Isa] Isabelle/HOL – A Proof Assistant for Higher-Order Logic, T Nipkow, L Paulson, M Wenzel, LNCS 2283, Springer, 2002.

[Jac] J Jacky, The way of Z - practical programming with formal methods, Cambridge University Press, 1997.

[JR] B Jacobs, J Rutten, A Tutorial on (Co)Algebras and (Co)Induction, EATCS Bulletin 62, pages 222–259, 1997.

[LEW] J Loeckx, H-D Ehrich, M Wolf, Specification of Abstract Data Types, Wiley/Teubner, 1996.

[McL] S Mac Lane. Categories for the Working Mathematician. Springer Verlag, 1971.

[MeTu] K Meinke, J V Tucker, Universal Algebra, pp. 189-411 in Handbook of Logic in Computer Science, Oxford University Press, 1992.

[Mil] R Milner, Communication and Concurrency, Prentice Hall, 1989.

[Min] Proof theory at work: Program development in the Minlog system, Benl et. al., in: Automated Deduction – A Basis for Applications, W Bibel, P H Schmitt, eds., Applied Logic Series, pages 41 – 71, Dordrecht, 1998.

[Oka] C Okasaki, Purely Functional Data Structures, Cambridge University Press, 1998.

[PX] S Hayashi, H Nakano, PX: A Computational Logic, MIT Press, 1988.

[Rog] M Roggenbach, CSP CASL - A new Integration of Process Algebra and Algebraic Specification, In F.Spoto, G.Scollo, A.Nijholt: Proceedings of AMiLP-2003, TWLT 21, Universiteit Twente, 2003.

[Sho] J R Shoenfield, Mathematical Logic, Addison-Wesley, 1967.

[ST] D Sannella, A Tarlecki, Foundations of Algebraic Specifications, Cambridge University Press 1997.

[SW] D Sannella, M Wirsing, A kernel language for algebraic specification and implementation, LNCS 158, Springer, 1983.

[Sch] H Schwichtenberg, Minimal Logic for Computable Functionals, `http://www.mathematik.uni-muenchen.de/~minlog/minlog/`, 2004.

[TD] A S Troelstra, D van Dalen, Constructivism in Mathematics, Vol. I, North-Holland, 1988.

[TS] A S Troelstra, H Schwichtenberg, Basic Proof Theory, Cambridge University Press, 1996.

[Tuc] J V Tucker, Theory of Programming Languages, Course Notes, UWS, 2005.