CS_376 Programming with Abstract Data Types

Ulrich Berger

Lecture Notes

Department of Computer Science

University of Wales Swansea

Autumn 2005
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 40

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 50
## 5.3 Initial algebras

- Page 54

## 5.4 Summary and Exercises

- Page 58

### 6 Specification of Abstract Data Types

- 6.1 Loose specifications
  - Page 59

- 6.2 Initial specifications
  - Page 66

- 6.3 Exception handling
  - Page 77

- 6.4 Modularisation
  - Page 79

- 6.5 Abstraction through Information hiding
  - Page 80

- 6.6 Specification languages
  - Page 81

- 6.7 Summary and Exercises
  - Page 83

### 7 Implementation of Abstract Data Types

- 7.1 Implementing ADTs in Functional and Object Oriented Style
  - Page 87

- 7.2 Efficiency
  - Page 91

- 7.3 Persistence
  - Page 93

- 7.4 Structural Bootstrapping
  - Page 96

- 7.5 Correctness
  - Page 97

- 7.6 Summary and Exercises
  - Page 98

### III Advanced Methods of Program Development

### 8 Term Rewriting and Rapid Prototyping

- 8.1 Equational logic
  - Page 102

- 8.2 Term rewriting systems
  - Page 105

- 8.3 Termination
  - Page 108

- 8.4 Confluence
  - Page 113

- 8.5 Rapid prototyping
  - Page 117

- 8.6 Summary and Exercises
  - Page 121

### 9 Programs from proofs

- 9.1 Formulas as data types
  - Page 125
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.2</td>
<td>A notation system for proofs</td>
<td>127</td>
</tr>
<tr>
<td>9.3</td>
<td>Program synthesis from intuitionistic proofs</td>
<td>129</td>
</tr>
<tr>
<td>9.4</td>
<td>Program synthesis from classical proofs</td>
<td>132</td>
</tr>
<tr>
<td>9.5</td>
<td>Applications</td>
<td>133</td>
</tr>
<tr>
<td>9.6</td>
<td>Summary and Exercises</td>
<td>134</td>
</tr>
</tbody>
</table>
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 from 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 pop up or the program appears to be inadequate. The process of maintaining is then started by repeating the different design steps. This methodology is illustrated by the so-called software life-cycle model (figure 1, [LEW]). It has at least two deficiencies. First, being based on tests, it can only confirm the existence of errors, not their absence. Hence testing fails to prove the correctness of a program. A second deficiency of testing is the fact that results are compared with expectations resulting from one’s own understanding of the problem. Hence testing may fail to unveil inadequacies of the program.

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

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)
\[ k = gcd(m,n) \iff k \mid m, k \mid n, \forall l (l \mid m, l \mid n \implies l \mid k) \]
\[ k \mid m \iff \exists q (k = q \times m) \]

Transformation

<table>
<thead>
<tr>
<th>Formal specification’</th>
<th>Program extraction</th>
</tr>
</thead>
</table>
| $\exists r [ r < n \land \exists q (m = q \times n + r) \land$
  \[
  r = 0 \rightarrow gcd(m,n) = n \land
  r > 0 \rightarrow gcd(m,n) = gcd(n,r)
  \]
| Prove the formula
  $\forall m > 0 \forall n > 0 \exists k$
  \[
  k \mid m, k \mid n \land
  \forall l (l \mid m \land l \mid n \implies l \mid k)
  \]
| From a formal proof
  extract a program $gcd$
  provably satisfying the specification,
  that is, the formula
  $\forall m > 0 \forall n > 0$
  \[
  gcd(m,n) \mid m, gcd(m,n) \mid n \land
  \forall l (l \mid m \land l \mid n \implies l \mid gcd(m,n))
  \]
  is provable

<table>
<thead>
<tr>
<th>Formal specification”</th>
</tr>
</thead>
</table>
| $mod(m,n) < n \land \exists q (m = q \times n + mod(m,n)) \land$
  \[
  mod(m,n) = 0 \rightarrow gcd(m,n) = n \land
  mod(m,n) > 0 \rightarrow gcd(m,n) = gcd(n,mod(m,n))
  \]

<table>
<thead>
<tr>
<th>Formal specification””</th>
</tr>
</thead>
</table>
| $m < n \rightarrow mod(m,n) = m \land$
  \[
  m \geq n \rightarrow mod(m,n) = mod(m-n,n) \land
  mod(m,n) = 0 \rightarrow gcd(m,n) = n \land
  mod(m,n) > 0 \rightarrow gcd(m,n) = gcd(n,mod(m,n))
  \]

Program

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 \( \text{mod} \) for the modulus function computing the remainder in an integer division.

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

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

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

\[
\text{Formal specification} \\
\uparrow \\
\text{Formal specification'} \\
\uparrow \\
\text{Formal specification”} \\
\uparrow \\
\text{Formal specification”’}
\]

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 \( \text{gcd} \)) is extracted (fully automatically) that provable meets the specification, that is,
   \[ \forall x A(x, p(x)) \]
   is provable (in our example, \( \forall m, n \ (\text{gcd}(m, n) \) is the greatest common divisor of \( m \) and \( n \))).

The concept of a constructive proof as an alternative foundation for logic and mathematics has been advocated first by L Kronecker, L E J Brouwer and A Kolmogorov in the beginning of the 20th century, and was formalized by Brouwer’s student A Heyting. The algorithmic interpretation of constructive proofs was formulated first by Brouwer, Heyting and Kolmogorov and is therefore often called BHK-interpretation (cf. [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

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

The constructive proof calculus studied in this course will be natural deduction. We will mainly follow the 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.
- $\Omega$ is a set whose elements are called operations, and which are of the form $f: s_1 \times \ldots \times s_n \to s$,

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

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

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

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 $c: s$ in $\Omega$ an element $c^A \in A_s$.
- For each operation $f: s_1 \times \ldots \times s_n \to s$ in $\Omega$ a function $f^A: A_{s_1} \times \ldots \times A_{s_n} \to A_s$.
3.1.3 Remarks

1. In the definition of an algebra (3.1.1) the expression $f : s_1 \times \ldots \times s_n \rightarrow s$ is meant symbolically, i.e. ‘$\times$’ and ‘$\rightarrow$’ are to be read as uninterpreted symbols. In the definition of an algebra (3.1.2), however, we used the familiar mathematical notation for set-theoretic functions to communicate by $f^A : A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_s$ a semantical object, namely a function $f^A$ whose domain is the cartesian product of the sets $A_{s_i}$ and whose range is $A_s$.

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

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

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

3.1.4 Example

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

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

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

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

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

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

$A_{\text{nat}} = \mathbb{N}$, \hspace{1em} $A_{\text{boole}} = \mathbb{B}$,

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

$0^A = 0$, \hspace{1em} $T^A = T$, \hspace{1em} $F^A = F$, 
the operations of addition on \( \mathbb{N} \) and the comparison relation \( \leq \) viewed as a boolean function, i.e. for all \( n, m \in \mathbb{N} \),

\[
\text{add}^A(n, m) = n + m,
\]

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

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

<table>
<thead>
<tr>
<th>Algebra</th>
<th>( A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carriers</td>
<td>( \mathbb{N}, B )</td>
</tr>
<tr>
<td>Constants</td>
<td>0, T, F</td>
</tr>
<tr>
<td>Operations</td>
<td>( +: \mathbb{N} \times \mathbb{N} \to \mathbb{N} )</td>
</tr>
<tr>
<td></td>
<td>( \leq: \mathbb{N} \times \mathbb{N} \to B )</td>
</tr>
</tbody>
</table>

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

\[
B_{\text{nat}} = \mathbb{N}^+, \quad B_{\text{boole}} = B,
\]

\[
0^B = 1, \quad T^B = T, \quad F^B = F,
\]

\[
\text{add}^B(n, m) = n \ast m \text{ for all } n, m \in \mathbb{N}^+.
\]

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

Written in a box

<table>
<thead>
<tr>
<th>Algebra</th>
<th>( B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carriers</td>
<td>( \mathbb{N}^+, B )</td>
</tr>
<tr>
<td>Constants</td>
<td>1, T, F</td>
</tr>
<tr>
<td>Operations</td>
<td>( \ast: \mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{N}^+ )</td>
</tr>
<tr>
<td></td>
<td>(</td>
</tr>
</tbody>
</table>
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 \( \Sigma \) in example 3.1.4 we would have exchanged the order of the sorts \( \text{nat} \) and \( \text{boole} \), we would have defined the same signature. But then the box displaying the algebra \( A \) would not be well-defined, since then the sort \( \text{boole} \) would be associated with the set \( \mathbb{N} \) and \( \text{nat} \) with \( \mathbb{B} \), and consequently the arities of the operations of \( \Sigma \) 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 \( \text{boole} \) 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 \( \Sigma \) can be used to build formal expressions, called terms, which denote elements of a given \( \Sigma \)-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 \( \Sigma \) of sort \( s \) is a term of sort \( s \).

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

\[
f(t_1, \ldots, t_n)
\]

is a term of sort \( s \).

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

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

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

3.2.2 Example

For the signature \( \Sigma \) of example 3.1.4 and the set of variables \( X := \{x, y\} \) the following are examples of terms in \( T(\Sigma, X) \):
\[ x \]
\[ 0 \]
\[ \text{add}(0, y) \]
\[ \text{add}((\text{add}(0, x), y) \]
\[ \text{add}((\text{add}(0, 0), \text{add}(x, x)) \]
\[ \text{add}(0, \text{add}(0, \text{add}(0, 0))) \]

The second and the last of these terms are closed.

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

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 \(t \in T(\Sigma, X)_s\) 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. \(t \in T(\Sigma) = T(\Sigma, \emptyset)\) the variable assignment \(\alpha\) 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 \( \text{var}(t) \) 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 \( \Sigma \)-algebra \( A \) under the variable assignment \( \alpha : \{x,y\} \rightarrow \mathbb{N}, \alpha(x) := 3 \) and \( \alpha(y) := 5 \).

\[
\begin{align*}
x^{A,\alpha} &= \alpha(x) = 3 \\
0^{A,\alpha} &= 0^A = 0 \\
\text{add}(0,y)^{A,\alpha} &= \\
\text{add}(\text{add}(0,x),y)^{A,\alpha} &= \\
\text{add}(\text{add}(0,0),\text{add}(x,x))^{A,\alpha} &= \\
\text{add}(0,\text{add}(0,\text{add}(0,0)))^{A,\alpha} &= 
\end{align*}
\]

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 \( \Sigma \). We define a \( \Sigma \)-algebra \( T(\Sigma, X) \), called term algebra, as follows.

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

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

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

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

(iii) If $P$ and $Q$ are formulas then $P \rightarrow Q$, $P \land Q$, and $P \lor Q$ are formulas, 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 $\Sigma$ are also called $\Sigma$-formulas

A free occurrence of a variable $x$ in a formula $P$ is an occurrence of $x$ in $P$ which is not in the scope of a quantifier $\forall x$ or $\exists x$. We let $FV(P)$ 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 $FV(P) = \emptyset$.

We set

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

and use the abbreviation

$$L(\Sigma) := L(\Sigma, \emptyset),$$

i.e. $L(\Sigma)$ is the set of closed $\Sigma$-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

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

3.3.4 Examples

$P_1 \quad \equiv \quad T = F$

$P_2 \quad \equiv \quad x = 0 \rightarrow \text{add}(y, x) = y$

$P_3 \quad \equiv \quad \exists y (x = y \rightarrow \forall z x = z)$

$P_4 \quad \equiv \quad \forall x (0 \leq x = T)$

$P_1$ and $P_2$ are quantifier free.

$P_1$ is an equation. $P_4$ is universal.

$P_1$ and $P_4$ are closed.

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

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

3.3.5 Exercise

Let $\Sigma$ be the signature of example 3.1.4 and $A$ the $\Sigma$-algebra of the natural numbers with zero, addition and the ‘less-or-equal’ relation. Write down $\Sigma$-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 $\Sigma$-algebra $\alpha : X \rightarrow A$, and $P \in L(\Sigma, X)$.

We define the relation

$$ A, \alpha \models P $$

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

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

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

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

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

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

(iv) $A, \alpha \models \forall x P$ iff $A, \alpha_x^a \models P$ 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 $\alpha_x^a$ defined by $\alpha_x^a(x) = a$ and $\alpha_x^a(y) = \alpha(y)$ for every variable different from $x$.

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

$$ A \models P $$

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

$$ A \models \Gamma, $$

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

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

\[
\Gamma \models P,
\]

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

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

3.4.2 Definition (Logical validity)

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

\[
\models P,
\]

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

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

3.4.3 Definition (Satisfiability)

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

3.4.4 Exercise

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

\[
P \text{ valid } \iff \{\neg P\} \text{ unsatisfiable (that is, not satisfiable)}
\]

\[
P \text{ satisfiable } \iff \{\neg P\} \text{ not valid}
\]

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 \texttt{nat} and \texttt{boole} and the operation \(<\colon \texttt{nat} \times \texttt{nat} \to \texttt{boole}.

Then the formula

\[ \exists x \forall y \,(x < y) \rightarrow \forall x \exists y \,(x < y) \]

is a tautology. The formula

\[ \forall x, y \,(x < y \rightarrow \exists z \,(x < z \land z < y)) \]

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

\[ \Gamma := \{ \forall x \,\neg(x < x), \, \forall x \, y \, z \,(x < y \land y < z \rightarrow x < z) \} \]

Then the formula

\[ P := \forall x, y \,(x < y \rightarrow \neg y < x) \]

is a logical consequence of \(\Gamma\), 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 \(\theta \colon X \rightarrow T(\Sigma, Y)\) that respects sorts, i.e. the variable \(x\) and the term \(\theta(x)\) have the same sorts for all \(x \in X\).

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

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

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

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

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

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

This also yields a recursive algorithm for computing \(t\theta\).
Notation

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

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

\[
(\theta \sigma)(x) := \theta(x) \sigma
\]

It can be easily proved that

\[
t(\theta \sigma) = (t \theta) \sigma
\]

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

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

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

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

\[
\{t/x\}^{A, \alpha} = \alpha^t_x
\]

Induction

The equation \( t(\theta \sigma) = (t \theta) \sigma \) in (b) above can be proved by induction on terms. By this we mean the following proof principle. Let \( P(t) \) be a statement about terms \( t \) (in (b) above we have for example \( P(t) :\iff t(\theta \sigma) = (t \theta) \sigma \)). In order to prove that \( P(t) \) holds for all terms \( t \) one has to prove the following.

- **Induction base.**
  
  \( P(t) \) holds for all atomic terms, i.e. variables and constants.

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

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

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

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

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

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

\[
= f(t_1 \theta, \ldots, t_n \theta) \sigma
\]

\[
= (f(t_1, \ldots, t_n) \theta) \sigma
\]
3.5.2 Exercise

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

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

3.5.3 Example

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

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

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

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

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

Now for example

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

Note that also

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

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

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 $t \in T(\Sigma, X)$, substitution $\theta: X \to T(\Sigma, Y)$ and variable assignment $\alpha: Y \to A$

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

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

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

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). \((c\theta)^{A,\alpha} = c^{A,\alpha} = c^A = c^{A,\langle\alpha\rangle}\)

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

\[
(f(t_1, \ldots, t_n)\theta)^{A,\alpha} = f(t_1\theta, \ldots, t_n\theta)^{A,\alpha} = f^A((t_1\theta)^{A,\alpha}, \ldots, (t_n\theta)^{A,\alpha}) = f^A(t_1^{A,\langle\alpha\rangle}, \ldots, t_n^{A,\langle\alpha\rangle})\quad\text{by induction hypothesis}
\]

\[
= f(t_1, \ldots, t_n)^{A,\langle\alpha\rangle}
\]

3.5.5 Definition (Applying substitutions to formulas)

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

\[
P\theta := \text{the result of replacing every free occurrence of a variable } x \text{ in } P \text{ by } \theta(x),
\]

possibly renaming the bound variables of \(P\) in order to avoid variable clashes

So, for example

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

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 \(\text{FV}(P) \subseteq X\), substitution \(\theta: X \to T(\Sigma, Y)\) and variable assignment \(\alpha: Y \to A\)

\[
A, \alpha \models P\theta \iff A, \theta^A,\alpha \models P
\]

For a substitution \(\{r/x\}\) this means

\[
A, \alpha \models P\{r/x\} \iff A, \alpha^R_{x,\alpha} \models P
\]

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

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

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

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 $\alpha$ a variable assignment. If

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

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

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

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

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

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

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

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

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,

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

   $$\text{depth}(f(t_1,\ldots,t_n)) = 1 + \max\{\text{depth}(t_1),\ldots,\text{depth}(t_1)\}$$
Show that \( \text{size}(t) \leq (1 + \text{arity}(t))^{\text{depth}(t)} \), where \( \text{arity}(t) \) 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 \( \text{arity}(t) \).

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

4. Let \( \Sigma \) be the signature with one sort, on constant, 0, and two binary operations, + and *. Let \( R \) be the \( \Sigma \)-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.

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 $\rightarrow^+$, deriving $P \rightarrow Q$ 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 $\rightarrow^+$ a fresh label $u$.

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

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 $\text{FA}(d) \subseteq \Gamma$ and end formula $P$, we say $d$ is *derivation of $P$ from $\Gamma$* and write

$$\Gamma \vdash d: P.$$
### Introduction rules

<table>
<thead>
<tr>
<th></th>
<th>Introduction rules</th>
<th>Elimination rules</th>
</tr>
</thead>
</table>
| ∧ | \[
\begin{align*}
\frac{P \land Q}{P \land Q} & \quad ^+ \\
\frac{P \land Q}{P} & \quad ^{-} \\
\frac{P \land Q}{Q} & \quad ^{-}
\end{align*}
\] | \[
\begin{align*}
\frac{P \land Q}{P} & \quad ^{-} \\
\frac{P \land Q}{Q} & \quad ^{-}
\end{align*}
\] |
| → | \[
\begin{align*}
\frac{u : P \\ \vdots}{Q} & \quad ^{+} u \\
\frac{P \rightarrow Q}{Q} & \quad ^{-}
\end{align*}
\] | \[
\begin{align*}
\frac{P \rightarrow Q}{P} & \quad ^{-}
\end{align*}
\] |
| ∨ | \[
\begin{align*}
\frac{P}{P} & \quad ^{+} \\
\frac{Q}{Q} & \quad ^{+}
\end{align*}
\] | \[
\begin{align*}
\frac{P \lor Q}{P} & \quad ^{-} \\
\frac{P \lor Q}{Q} & \quad ^{-}
\end{align*}
\] |
| ⊥ | \[
\begin{align*}
\frac{\bot}{P} & \quad ^{eq} \\
\frac{\bot}{P} & \quad ^{raa}
\end{align*}
\] |
| ∀ | \[
\begin{align*}
\frac{P(x)}{\forall x \, P(x)} & \quad ^{+} \\
\end{align*}
\] | \[
\begin{align*}
\frac{\forall x \, P(x)}{P(t)} & \quad ^{-}
\end{align*}
\] |
| ∃ | \[
\begin{align*}
\frac{P(t)}{\exists x \, P(x)} & \quad ^{+}
\end{align*}
\] | \[
\begin{align*}
\frac{\exists x \, P(x)}{\forall x \, (P(x) \rightarrow Q)} & \quad ^{-} \\
\frac{\forall x \, (P(x) \rightarrow Q)}{Q} & \quad ^{-}
\end{align*}
\] |

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

#### 4.1.2 Examples (propositional connectives)

1. We begin with a derivation involving the *conjunction introduction rule*, \(^{+}\) and the *conjunction elimination rules*, \(^{-}\) and \(^{-}\). We derive from the assumption \(P \land Q\) the formula \(Q \land P\):

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

2. If we add to the derivation in example 1 an application of the *implication introduction rule*, \(^{-}\), we obtain a derivation of \(P \land Q \rightarrow Q \land P\) that does not contain free assumptions:
3. In the following derivation of the formula $P \rightarrow (Q \rightarrow P)$ we use the rule $\rightarrow^+$ twice. The upper instance of this rule is used with the formula $Q$, without $Q$ actually occurring as an open assumption.

$$
\begin{align*}
\frac{u : P \land Q}{Q} & & \land^- \\
\frac{u : P \land Q}{P} & & \land^+
\end{align*}
\frac{Q \land P}{P \land Q \rightarrow Q \land P} & \rightarrow^+_u
$$

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

$$
\begin{align*}
\frac{P \land Q \rightarrow R}{u : P \land Q} & & \land^+ \\
& \frac{v : Q}{P \land Q} & \rightarrow^- \\
\frac{R}{Q \rightarrow R} & \rightarrow^+_v \\
\frac{Q \rightarrow R}{P \rightarrow (Q \rightarrow R)} & \rightarrow^+_u
\end{align*}
$$

5. In order to familiarise ourselves with the *disjunction introduction rules*, $\lor^+_1$, $\lor^+_r$, and the *disjunction elimination rule*, $\lor^-$, we derive $Q \lor P$ from $P \lor Q$.

$$
\begin{align*}
\frac{u : P \lor P}{Q \lor P} & \lor^+_1 \\
\frac{v : Q}{P \lor Q} & \lor^+_r \\
\frac{P \lor Q}{Q \lor P} & \rightarrow^+_u \\
\frac{Q \lor P}{P \rightarrow Q \lor P} & \rightarrow^+_v \\
\frac{Q \lor P}{Q \lor P} & \rightarrow^- \\
\frac{P \lor Q}{Q \lor P \rightarrow Q \lor P} & \rightarrow^+_u
\end{align*}
$$

6. As a slightly more complicated example we derive (one half of) one of de-Morgan’s laws, $P \land (Q \lor R) \rightarrow (P \land Q) \lor (P \land R)$, without assumptions.

$$
\begin{align*}
\frac{u : P \land (Q \lor R)}{(P \land Q) \lor (P \land R)} & \land^- \\
\frac{P}{P \land Q} & \land^+ \\
\frac{v : Q}{P \land Q} & \land^- \\
\frac{P \land R}{(P \land Q) \lor (P \land R)} & \lor^+_1 \\
\frac{w : R}{(P \land Q) \lor (P \land R)} & \lor^+_r \\
\frac{R}{P \land (Q \lor R) \rightarrow (P \land Q) \lor (P \land R)} & \rightarrow^+_w \\
\frac{P \land (Q \lor R)}{(P \land Q) \lor (P \land R)} & \lor^-
\end{align*}
$$
7. Finally, we turn our attention to the rules concerning absurdity. $\bot$, namely \textit{ex-falso-quodlibet}, \text{efq}, and \text{reductio-ad-absurdum}, \text{raa}. Recall that $\neg P$ is shorthand for $P \rightarrow \bot$, and therefore $\neg\neg P$ stands for $(P \rightarrow \bot) \rightarrow \bot$. We derive \textit{Peirce’s law} $(P \rightarrow Q) \rightarrow P \vdash P$:

$$
\begin{array}{c}
\text{u : } \neg P \\
\text{v : } P \\
\hline
\mid \text{efq} \quad \mid \text{raa} \\
\mid (P \rightarrow Q) \rightarrow P \\
\mid P \rightarrow Q \\
\mid \rightarrow v \\
\mid \rightarrow u \\
\mid \bot \\
\mid (P \rightarrow \bot) \rightarrow \bot \\
\mid P \\
\end{array}
$$

8. The rule \textit{ex-falso-quodlibet} is weaker than \textit{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 \textit{ex-falso-quodlibet} (but using \textit{reductio-ad-absurdum} instead):

$$
\begin{array}{c}
\text{u : } \neg P \\
\hline
\mid \text{raa} \\
\mid (P \rightarrow \bot) \rightarrow \bot \\
\mid P
\end{array}
$$

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 \textit{for-all introduction rule}, $\forall^+$, and the \textit{for-all elimination rule}, $\forall^-$:

$$
\begin{array}{c}
\forall x P(x) \\
\hline
\forall x P(x + 1) \\
\end{array}
$$

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’.

$$
\begin{array}{c}
\forall y(x < 1 + y) \\
\hline
\forall x(x < 1 + 0) \\
\end{array}
$$

$$
\begin{array}{c}
\forall x(\forall y(x < y + 1) \rightarrow x = 0) \\
\hline
\forall y(y < y + 1) \rightarrow y = 0 \\
\forall y(y < y + 1) \\
\hline
\forall y(y = 0)
\end{array}
$$
3. The *exists introduction rule*, \( \exists^+ \), and the *exists elimination rule*, \( \exists^- \) are used in the following derivation.

\[
\begin{array}{c}
u : \forall x \ (x - 1) + 1 = x \\
(x - 1) + 1 = x \\
\exists y (y + 1 = x) \\
\forall x \exists y (y + 1 = x) \\
\forall x ((x - 1) + 1 = x) \rightarrow \forall x \exists y (y + 1 = x) \rightarrow^+ u
\end{array}
\]

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

\[
\begin{array}{c}
\forall x (P(x) \rightarrow Q(f(x))) \\
P(x) \rightarrow Q(f(x)) \\
\forall x (P(x) \rightarrow Q(f(x))) \\
\forall x (P(x) \rightarrow \exists y \ Q(y)) \rightarrow^+ u \\
\exists x \ P(x) \\
\forall x (P(x) \rightarrow \exists y \ Q(y)) \rightarrow^- \\
\exists y \ Q(y)
\end{array}
\]

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**  
  \[ t = t \text{ refl} \]

- **Symmetry**  
  \[ s = t \text{ sym} \]

- **Transitivity**  
  \[ r = s \quad s = t \text{ trans} \]

- **Compatibility**  
  \[ s_1 = t_1 \quad \ldots \quad s_n = t_n \text{ comp} \]
  \[ f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) \]
  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 \( \forall x, y \ ((0 + x) \ast y = x \ast y) \):

\[
\frac{\forall x_1, y_1 \ (x_1 + y_1 = y_1 + x_1)}{\forall y_2 \ (0 + y_2 = y_2 + 0)} \ \forall^\ \ \ \ \frac{\forall x_3 \ (x_3 + 0 = x_3)}{0 + x = x} \ \text{trans} \ \frac{0 + x = x}{(0 + x) \ast y = x \ast y} \ \text{refl comp} \\
\frac{\forall y_4 \ ((0 + x) \ast y_4 = x \ast y_4)}{\forall x_5 \forall y_5 \ ((0 + x) \ast y_5 = x \ast y_5)} \ \forall^\ +
\]

4.2.2 Definition

\[\Gamma \vdash \_ P : \Leftrightarrow \Gamma \vdash d : P \text{ for some derivation } d.\]

(P is derivable from \( \Gamma \) in \textit{classical logic})

\[\Gamma \vdash \_ P : \Leftrightarrow \Gamma \vdash d : P \text{ for some derivation } d \text{ not using the rule reductio-ad-absurdum.}\]

(P is derivable from \( \Gamma \) in \textit{intuitionistic logic})

\[\Gamma \vdash \_ P : \Leftrightarrow \Gamma \vdash d : P \text{ for some derivation } d \text{ using neither the rule reductio-ad-absurdum nor the rule ex-falso-quodlibet.}\]

(P is derivable from \( \Gamma \) in \textit{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

\[ r = s \vdash \_ t(r) = t(s) \]

\textbf{Proof.} Induction on \( t(x) \).

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

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

Finally, consider \( t(x) \) of the form \( f(t_1(x), \ldots, t_n(x)) \). By induction hypothesis we may assume that we already have a derivation of \( r = s \vdash \_ t_i(r) = t_i(s) \) for \( i = 1, \ldots, n \). One application of the compatibility rule yields the required sequent.
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:

$$r = s \vdash_m P(r) \leftrightarrow P(s)$$

**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

$$r = s \vdash_m t_1(r) = t_2(r) \leftrightarrow t_1(s) = t_2(s)$$

By Lemma 4.2.3 we have already derivations of

$$r = s \vdash_m t_1(r) = t_1(s) \quad \text{and} \quad r = s \vdash_m t_2(r) = t_2(s)$$

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

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

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

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

Whilst the soundness theorem is not very surprising, because it just states that the inference rules are correct, the following completeness theorem proved by 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 $\Gamma \vdash_c P$.

In words: If $P$ is a logical consequence of $\Gamma$ (i.e. $P$ is true in all models of $\Gamma$), then this can be formally derived by the inference rules of natural deduction.
The 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 \( \Gamma \) is called consistent if \( \Gamma \not\vdash \bot \), that is there is no (classical) derivation of \( \bot \) from assumptions in \( \Gamma \).

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

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. Similar 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 bool.

\[
\text{Boole 1} \quad \begin{array}{c}
T \neq F \\
\text{bool1}
\end{array}
\]

\[
\text{Boole 2} \quad \forall x \ (x = T \lor x = F) \quad \text{bool2}
\]
Recall that $r \neq s$ is an abbreviation for $\neg r = s$ which in turn stands for $r = s \rightarrow \bot$. Recall also that we agreed to abbreviate an equation $t = T$ by $t$.

4.4.2 Lemma

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

We leave the proof as an exercise to the reader.

Hint: We have to derive $\forall x ((x = T \rightarrow \bot) \leftrightarrow x = F)$. 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 were 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 $\frac{0 \neq t + 1}{\text{peano1}}$

Peano 2 $\frac{s + 1 = t + 1 \rightarrow s = t}{\text{peano2}}$

Induction $\frac{P(0) \quad \forall x (P(x) \rightarrow P(x + 1))}{\forall x P(x)}$ $\text{ind}$
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) \((P \rightarrow \neg Q) \rightarrow (Q \rightarrow \neg P)\)

(b) \((P \rightarrow (Q \rightarrow R)) \rightarrow ((P \rightarrow Q) \rightarrow (P \rightarrow R))\)

(c) \(P \rightarrow \neg \neg P\)

(d) \(\neg (P \land \neg P)\)

(e) \((P \land (Q \lor R)) \leftrightarrow ((P \land Q) \lor (P \land R))\)

(f) \((P \lor Q) \rightarrow \neg (\neg P \land \neg Q)\)

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

2. Intuitionistic logic

(a) \((P \land \neg P) \rightarrow R\)

(b) \((\neg P \lor Q) \rightarrow (P \rightarrow Q)\)

(c) \((\neg \neg P \rightarrow P) \leftrightarrow ((\neg P \rightarrow P) \rightarrow P)\)
(d) \((P \lor Q) \rightarrow (\neg P \rightarrow Q)\)

3. Classical logic

(a) \(\neg\neg P \rightarrow P\)
(b) \((\neg P \rightarrow P) \rightarrow P\)
(c) \(P \lor \neg P\)
(d) \((\neg P \land \neg Q) \rightarrow (P \lor Q)\)
(e) \((\neg P \lor \neg Q) \rightarrow (P \land Q)\)
(f) \((P \rightarrow Q) \rightarrow P \land \neg Q\)

Quantifier logic

1. Minimal logic

(a) \(\forall x (P(x) \rightarrow Q(x)) \rightarrow (\forall x P(x) \rightarrow \forall x Q(x))\)
(b) \(\forall x (P(x) \rightarrow \exists y P(y))\)
(c) \(\forall x P(x) \rightarrow \exists x P(x)\)
(d) \(\exists x (P(x) \lor Q(x)) \leftrightarrow \exists x P(x) \lor \exists x Q(x)\)
(e) \(\exists x (P(x) \land Q(x)) \rightarrow \exists x P(x) \land \exists x Q(x)\)
(f) \(\exists x \neg P(x) \rightarrow \neg \forall x P(x)\)
(g) \(\neg\neg \forall x P(x) \rightarrow \forall x \neg\neg P(x)\)

2. Intuitionistic logic

(a) \(\exists x (P(x) \land \neg P(x)) \rightarrow \forall x (P(x) \land \neg P(x))\)
(b) \(\forall x (\neg f(x) < 0) \rightarrow f(0) < 0 \rightarrow f(0) = 0\)

3. Classical logic

(a) \(\forall x \neg \neg P(x) \rightarrow \neg \forall x P(x)\)
(b) \(\neg \forall x P(x) \rightarrow \exists x \neg P(x)\)
(c) \(\exists x (P(x) \rightarrow \forall y P(y))\)

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.