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 $\Sigma$-algebras. A homomorphism $\varphi: A \rightarrow B$ from $A$ to $B$ is a family $\varphi = (\varphi_s)_{s \in S}$ of functions

$$\varphi_s: A_s \rightarrow B_s$$

such that

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

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

A homomorphism $\varphi: A \rightarrow B$ is called
monomorphism if all $\varphi_s: A_s \to B_s$ are injective
epimorphism if all $\varphi_s: A_s \to B_s$ are surjective
isomorphism if all $\varphi_s: A_s \to B_s$ are bijective

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

5.1.2 Example

Consider the following signature

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

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

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

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

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

Define

$$\varphi: N \to M, \quad \varphi(n) := 2^n$$

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

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

Since obviously $\varphi$ is bijective, it is an isomorphism.
5.1.3 Example

<table>
<thead>
<tr>
<th>Signature</th>
<th>STACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>elts, stack</td>
</tr>
<tr>
<td>Constants</td>
<td>0: elts emptystack: stack</td>
</tr>
<tr>
<td>Operations</td>
<td>push: elts × stack → stack pop: stack → stack top: stack → elts</td>
</tr>
</tbody>
</table>

The following is an algebra for the signature STACK.

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

Consider also the following algebra for the signature STACK:

<table>
<thead>
<tr>
<th>Algebra</th>
<th>Stack0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carriers</td>
<td>({0}, \mathbb{N})</td>
</tr>
<tr>
<td>Constants</td>
<td>0, 0</td>
</tr>
<tr>
<td>Operations</td>
<td>push(^{\text{Stack0}}): ({0} \times \mathbb{N} \to \mathbb{N}), push(^{\text{Stack0}})(0, (n)) := (n + 1) pop(^{\text{Stack0}}): (\mathbb{N} \to \mathbb{N}), pop(^{\text{Stack0}})((n + 1)) := (n), pop(^{\text{Stack0}})(0) := 0 top(^{\text{Stack0}}): (\mathbb{N} \to {0}), top(^{\text{Stack0}})((n)) := 0</td>
</tr>
</tbody>
</table>

Let us define a homomorphism \(\varphi: \text{SeqN} \to \text{Stack0}\).

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

\[
\varphi_{\text{elts}}: \text{SeqN}_{\text{elts}} \to \text{Stack0}_{\text{elts}}, \quad \varphi_{\text{stack}}: \text{SeqN}_{\text{stack}} \to \text{Stack0}_{\text{stack}}.
\]

Since \(\text{SeqN}_{\text{elts}} = \mathbb{N}\), \(\text{Stack0}_{\text{elts}} = \{0\}\), \(\text{SeqN}_{\text{stack}} = \mathbb{N}^*\), and \(\text{Stack0}_{\text{stack}} = \mathbb{N}\), this means

\[
\varphi_{\text{elts}}: \mathbb{N} \to \{0\}, \quad \varphi_{\text{stack}}: \mathbb{N}^* \to \mathbb{N}.
\]

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

\[
\varphi_{\text{stack}}(\alpha) := \text{length}(\alpha) \quad (\text{the length of the sequence } \alpha).
\]
In order to show that $\varphi$ is a homomorphism, we have to check 4 equations, one for each constant and operation in $\text{STACK}$. We only check the equation for push and leave the rest as an exercise.

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

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

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

### 5.1.4 Definition

Let $\Sigma = (S, \Omega)$ be a signature and $A, B, C$ $\Sigma$-algebras. For homomorphisms $\varphi: A \to B$, $\psi: B \to C$ its **composition** $\psi \circ \varphi: A \to C$ 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 $\varphi$ is a homomorphism from $A$ to $B$ and $\psi$ is a homomorphism from $B$ to $C$, then $\psi \circ \varphi$ is a homomorphism from $A$ to $C$.

**Proof.** Coursework.

### 5.1.6 Theorem

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

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

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

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

$$\varphi^{-1}_s(f^B(b_1, \ldots, b_n)) = f^A(\varphi^{-1}_{s_1}(a_1), \ldots, \varphi^{-1}_{s_n}(a_n)).$$
We have
\[
\varphi^{-1}_s(f^B(b_1, \ldots, b_n)) = \varphi^{-1}_s(f^B(\varphi_{s_1}^{-1}(b_1)), \ldots, \varphi_{s_n}^{-1}(b_1))) = f^A(\varphi_{s_1}^{-1}(b_1), \ldots, \varphi_{s_n}^{-1}(b_1)))
\]

5.1.7 Definition

For two \(\Sigma\)-algebras \(A\) and \(B\) we set
\[
A \simeq B \iff \text{there exists an isomorphism from } A \text{ to } B
\]

5.1.8 Theorem

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

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

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

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

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

5.1.9 Definition

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

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

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

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

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

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

Otherwise $C$ is called polymorphic.

5.1.11 Example

Let $\Sigma$ be a signature.

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

(b) For each $\Sigma$-algebra $A$ the class $\{ B \in \text{Alg}(\Sigma) \mid B \simeq A \}$ 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\textsuperscript{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 $\mathbb{N}$ of natural numbers as carrier set. By referring to the concrete set $\mathbb{N}$ the property of being invariant under isomorphism is lost. Hence the class

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

is not an ADT. Referring –like above– to a fixed set in the specification of a data type means on the programming side to fix a concrete implementation of a data type already in the specification of a system. Such premature design decisions 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 $\Sigma$ is invariant under homomorphism and hence defines an ADT for $\Sigma$.

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).

\textsuperscript{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 $\Sigma$ is also a sort in $\Sigma'$, and every operation or constant in $\Sigma$ is also an operation or constant in $\Sigma'$.

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

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

5.2.2 Definition

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

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

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

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

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

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

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

5.2.3 Definition

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

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

5.2.4 Remarks

1. Obviously, the relation ‘$A$ is a subalgebra of $B$’ defines a partial order on the class $\text{Alg}(\Sigma)$ of all $\Sigma$-algebras.
2. Clearly, a subalgebra $A$ of an algebra $B$ is completely determined by $B$ and the sets $A_s$. However, if we chose arbitrary subsets $A_s$ of $B_s$ for all sorts $s$ these will define a subalgebra of $B$ only if the sets $A_s$ contain all the constants $c^B$ 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 $\Sigma$-algebras and let $\varphi: A \to B$ be a homomorphism. For each sort $s \in S$ we define the set

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

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 $\varphi$.

5.2.6 Definition

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

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

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 $\Sigma$-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 $c: s$.
- $f^{A/\sim}([a_1]_{\sim_{s_1}}, \ldots, [a_n]_{\sim_{s_n}}) := [f^A(a_1, \ldots, a_n)]_{\sim_s}$, for each operation $f: s_1 \times \ldots \times s_n \to s$ and all $a_i \in A_{s_i}$.

Note that the condition on $\sim$ of being a congruence is needed to verify that $f^{A/\sim}$ is well-defined, i.e. the right hand side of the defining equation does not depend on the choice of the representatives $a_i$ of the equivalence classes $[a_i]_{\sim_{s_i}}$. 
5.2.7 Example

Consider the signature

<table>
<thead>
<tr>
<th>Signature</th>
<th>Σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>nat</td>
</tr>
<tr>
<td>Constants</td>
<td>0: nat</td>
</tr>
<tr>
<td>Operations</td>
<td>add: nat × nat → nat</td>
</tr>
</tbody>
</table>

and the Σ-algebra A of natural numbers with 0 and addition. We define a binary relation ∼ on A’s carrier N by

\[ a \sim b \Leftrightarrow a + b \text{ is even} \]

Clearly ∼ is an equivalence relation (prove this as an exercise). To prove that it is a congruence for A, we have to show that ∼ is preserved by the operation add^A, which is addition. Hence we have to show

\[ a_1 \sim b_1, \ a_2 \sim b_2 \implies a_1 + a_2 \sim b_1 + b_2 \]

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

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

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

5.2.8 Homomorphism Theorem

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

\[ a \sim_{\varphi, s} b \Leftrightarrow \varphi_s(a) = \varphi_s(b) \]

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

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

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

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. $f: s_1 \to s$. 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:

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

$$= \varphi_s(f^A(a))$$

$$= f^B(\varphi_{s_1}(a))$$

$$= f^B([\varphi]_{s_1}([a]_{\sim_\varphi,s_1}))$$

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

**Remark.** The above theorem tells us that each homomorphism $\varphi: A \to B$ naturally induces a congruence $\sim_\varphi$ on $A$. In fact *every* congruence $\sim$ on $A$ can be obtained in that way, since the mappings $[\cdot]_{\sim} : A_s \to A_s/\sim_s$ obviously constitute a homomorphism $[\cdot]_{\sim} : A \to A/\sim$ 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 $\Sigma$-algebra and $C$ a class of $\Sigma$-algebras.

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

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

**Remarks.** 1. By replacing in the definition above \(\varphi: A \to B\) by \(\varphi: B \to A\) 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 *coinduction*. 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 \( \Sigma \)-algebra \( A \) and any variable assignment \( \alpha: X \rightarrow A \) the family of functions \( \text{eval}^{A,\alpha} = (\text{eval}^{A,\alpha}_s)_{s \in S} \) defined by

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

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

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

5.3.3 Definition

A \( \Sigma \)-algebra \( A \) is called \textit{generated} (freely generated) if every \( a \in A_s \) is the value of a (unique) closed term, i.e. for every \( a \in A_s \) there exists a (unique) term \( t \in T(\Sigma) \) such that \( t^A = a \). Note that this is equivalent to saying that \( \text{eval}: T(\Sigma) \rightarrow A \) 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 \textit{constructors}, or \textit{generators}. All other operations are called \textit{observers}. We set \( \Sigma' := (S, \Omega') \). A \( \Sigma \)-algebra \( A \) is called \textit{generated} (freely generated) \textit{by} \( \Omega' \) if every \( a \in A_s \) is the (unique) value of a closed \( \Sigma' \)-term, i.e. for every \( a \in A_s \) there exists a (unique) term \( t \in T(\Sigma') \) such that \( t^A = a \).

5.3.4 Examples

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

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

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

4. Consider the following signature.

<table>
<thead>
<tr>
<th>Signature</th>
<th>BOOLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>boole</td>
</tr>
<tr>
<td>Constants</td>
<td>( T: ) boole</td>
</tr>
<tr>
<td></td>
<td>( F: ) boole</td>
</tr>
<tr>
<td>Operations</td>
<td>( \text{not: boole} \rightarrow \text{boole} )</td>
</tr>
<tr>
<td></td>
<td>( \text{and: boole} \times \text{boole} \rightarrow \text{boole} )</td>
</tr>
<tr>
<td></td>
<td>( \text{or: boole} \times \text{boole} \rightarrow \text{boole} )</td>
</tr>
</tbody>
</table>
and the following **BOOLE**-algebra

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

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

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

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

### 5.3.5 Theorem

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

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

**Base.**

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

**Step.**

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

### 5.3.6 Theorem

For a \( \Sigma \)-algebra \( A \) the following statements are equivalent.
(i) $A$ is initial.

(ii) $A$ is freely generated.

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

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

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

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

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

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

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

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

5.3.7 Theorem

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

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

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

$$T_C(\Sigma) := T(\Sigma)/\sim_C$$
is initial for $\mathcal{C}$. For every $A \in \mathcal{C}$ the unique homomorphism from $\varphi: T_{\mathcal{C}}(\Sigma) \to A$ is given by

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

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

**Proof.** By definition $\sim_{\mathcal{C}}$ is the intersection of the congruences $\sim_{\text{eval}^A}$ ($A \in \mathcal{C}$) (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 $T(\Sigma)$. It is easy to see that $\varphi$ above is a well-defined homomorphism. The proof that $\varphi$ is unique is similar to the proof of theorem 5.3.5 and is left as an exercise. Hence $T_{\mathcal{C}}(\Sigma)$ 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 $\Sigma$-algebra of natural numbers with 0 and addition from exercise 5.2.7. Show that a function $\varphi: \mathbb{N} \to \mathbb{N}$ is an endomorphism on $A$ if and only if there is a number $k \in \mathbb{N}$ such that $\varphi(n) = k \ast n$ for all $n \in \mathbb{N}$. 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 $\mathbb{N}$. Show that the only homomorphism on $B$ is the identity.

3. Let $\Sigma$ be the signature of Example 1. Consider the $\Sigma$-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 $\varphi: A \to B$ be a homomorphism. For each sort $s \in S$ define a binary relation $\sim_{\varphi,s}$ on $A_s$ by

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

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

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

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

\[
\text{Mod}_\Sigma(\Phi) \, := \, \{ A \in \text{Alg}(\Sigma) \mid A \models \Phi \}
\]

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

6.1.2 Lemma

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

\[
\varphi(t^{A,\alpha}) = t^{B,\varphi\alpha}
\]

Proof. Structural induction on \( t \).

Base: variables.

\[
\varphi(x^{A,\alpha}) = \varphi(\alpha(x)) = (\varphi \circ \alpha)(x) = t^{B,\varphi\alpha}
\]

Base: constants.

\[
\varphi(c^{A,\alpha}) = \varphi(c^A) = \varphi(c^B) = c^{B,\varphi\alpha}
\]
Step.

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

\[= f(t_1, \ldots, t_n)^{B,\varphi\alpha}\]

6.1.3 Theorem

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

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

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

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

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

(i) Base.

\[A, \alpha \models t_1 = t_2 \quad \text{iff} \quad t_1^{A,\alpha} = t_2^{A,\alpha}\]

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

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

\[\text{iff} \quad B, \varphi \circ \alpha \models t_1 = t_2\]

(ii) Step: propositional connectives.

\[A, \alpha \models P \land Q \quad \text{iff} \quad A, \alpha \models P \text{ and } A, \alpha \models Q\]

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

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

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

\[A, \alpha \models \neg P \quad \text{iff} \quad A, \alpha \not\models P\]

\[\text{iff} \quad B, \varphi \circ \alpha \not\models P \quad \text{(i.h.)}\]

\[\text{iff} \quad B, \varphi \circ \alpha \models \neg P\]
(iii) \textit{Step: quantifiers.}

\[ A, \alpha \models \forall x P \quad \text{iff} \quad A, \alpha^a_x \models P \text{ for all } a \in A_s \]

\[ B, \varphi \circ (\alpha^a_x) \models P \text{ for all } a \in A_s \quad \text{(i.h.)} \]

\[ B, (\varphi \circ \alpha)^{\varphi(a)}_x \models P \text{ for all } a \in A_s \quad (\varphi \circ (\alpha^a_x) = (\varphi \circ \alpha)^{\varphi(a)}_x) \]

\[ B, (\varphi \circ \alpha)^b_x \models P \text{ for all } b \in B_s \quad (\varphi \text{ is surjective}) \]

\[ B, \varphi \circ \alpha \models \forall x P \]

\[ A, \alpha \models \exists x P \quad \text{iff} \quad A, \alpha^a_x \models P \text{ for at least one } a \in A_s \]

\[ B, \varphi \circ (\alpha^a_x) \models P \text{ for at least one } a \in A_s \quad \text{(i.h.)} \]

\[ B, (\varphi \circ \alpha)^{\varphi(a)}_x \models P \text{ for at least one } a \in A_s \quad (\varphi \circ (\alpha^a_x) = (\varphi \circ \alpha)^{\varphi(a)}_x) \]

\[ B, (\varphi \circ \alpha)^b_x \models P \text{ for at least one } b \in B_s \quad (\varphi \text{ is surjective}) \]

\[ B, \varphi \circ \alpha \models \exists x P \]

\section*{6.1.4 Theorem}

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

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

\section*{6.1.5 Example}

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

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{Signature} & \(\Sigma\) \\
\hline
\textbf{Sorts} & \text{boole} \\
\hline
\textbf{Constants} & \(T, F: \text{boole}\) \\
\hline
\textbf{Operations} & \(\neg: \text{boole} \to \text{boole}\) \\
& \(\land, \lor: \text{boole} \times \text{boole} \to \text{boole}\) \\
\hline
\end{tabular}
\end{center}

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

\[ P_1 \equiv \neg(T) = F \]

\[ P_2 \equiv \neg(F) = T \]

\[ P_3 \equiv \land(T, T) = T \]

\[ P_4 \equiv \forall x (\land(F, x) = F) \]
\[ P_5 \equiv \forall x \ (\text{and}(x,F) = F) \]
\[ P_6 \equiv \forall x, y \ (\text{or}(x,y) = \neg(\text{and}(\neg(x), \neg(y)))) \]

Consider the following $\Sigma$-algebras

<table>
<thead>
<tr>
<th>Algebra</th>
<th>Boole</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carriers</td>
<td>$B := {\text{#t}, \text{#f}}$</td>
</tr>
<tr>
<td>Constants</td>
<td>#t, #f</td>
</tr>
<tr>
<td>Operations</td>
<td>$\neg : B \to B$ (negation)</td>
</tr>
<tr>
<td></td>
<td>$\wedge : B \times B \to B$ (conjunction)</td>
</tr>
<tr>
<td></td>
<td>$\vee : B \times B \to B$ (disjunction)</td>
</tr>
</tbody>
</table>

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

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

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

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

The ADT $\text{Mod}_\Sigma(\Phi)$ 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($N$)).

\[ P_7 \equiv \]

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

\[ P_8 \equiv \]

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

<table>
<thead>
<tr>
<th>Signature</th>
<th>$\Sigma_{0S+}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>nat</td>
</tr>
<tr>
<td>Constants</td>
<td>$0 : \text{nat}$</td>
</tr>
</tbody>
</table>
| Operations | $\text{succ} : \text{nat} \to \text{nat}$  
|            | $+: \text{nat} \times \text{nat} \to \text{nat}$ |

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 $\text{succ} : \mathbb{N} \to \mathbb{N}$, $\text{succ} (n) := n + 1$, and addition $+: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

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

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

$$
\begin{align*}
P_1 & : \equiv \forall x \left(0 \neq \text{succ}(x)\right) \\
P_2 & : \equiv \forall x, y \left(\text{succ}(x) = \text{succ}(y) \rightarrow x = y\right) \\
P_3 & : \equiv \forall x \left(x = 0 \lor \exists y \left(x = \text{succ}(y)\right)\right) \\
P_4 & : \equiv \forall x \left(x + 0 = x\right) \\
P_5 & : \equiv \forall x, y \left(x + \text{succ}(y) = \text{succ}(x + y)\right)
\end{align*}
$$

Clearly the algebra $N_{0S+}$ is a model of $\{P_1, \ldots, P_3\}$.

But there are still unwanted models. For example the $\Sigma_{0S+}$-algebra $A$ with $A_{\text{nat}} := \{0\} \times \mathbb{N} \cup \{1\} \times \mathbb{Z}$ with $0^A := (0, 0)$, $\text{succ}^A((i, n)) := (i, n + 1)$, and $+^A((i, n), (j, m)) := (\max(i, j), n + m)$.

It is easy to check that $A$ is a model of $\{P_1, \ldots, P_3\}$.

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

$$
P_6 : \equiv \forall x \left(x + x = x \rightarrow x = 0\right)
$$

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

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

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

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

Notation. We will display loose specifications in a similar way as we display signatures. The axioms will be displayed without universal quantifier prefix.

6.1.8 Example

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

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

6.1.9 Definition

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

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

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

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

$(\Sigma', \Phi')$ is called **persistent extension** of $(\Sigma, \Phi)$ if $(\Sigma', \Phi')$ is an extension of $(\Sigma, \Phi)$ and addition for every closed $\Sigma$-formula $P$ it holds that $\Phi' \models P$ 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 $(\Sigma', \Phi')$ be an extension of the loose specification $(\Sigma, \Phi)$ such that every $\Sigma$-algebra $A$ satisfying $\Phi$ can be expanded to a $\Sigma'$-algebra $A'$ satisfying $\Phi'$.

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

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

### 6.1.12 Example

Let $(\Sigma, \Phi)$ be the specification in Example 6.1.8. Extend $(\Sigma, \Phi)$ by the operation

\[
\text{pred} : \text{nat} \rightarrow \text{nat}
\]

and the equation

\[
\text{pred}(\text{succ}(x)) = x
\]

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

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

6.2.1 Example

Let $\Sigma := (\{s\}, \{a : s, \ b : s, \ c : s\})$ and $\Phi := \{a = b \lor a = c\}$. We will show that the loose specification $(\Sigma, \Phi)$ has no initial model. Let $A$ be a $\Sigma$-algebra that is initial for $\text{Mod}_\Sigma(\Phi)$. We have show $A \not\in \text{Mod}_\Sigma(\Phi)$, i.e. the formula $a = b \lor a = c$ is false in $A$.

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

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

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

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

$$
\varphi : A \rightarrow B, \quad \psi : A \rightarrow C
$$

Using the homomorphic property of $\varphi$ and $\psi$ we see

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

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

Hence the formula $a = b \lor a = c$ is false in $A$.

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

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

$$
t_1 = t_2
$$

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

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

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

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

$$T_E(\Sigma) := T_{\text{Mod}_E}(\Sigma)$$

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

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

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

6.2.3 Theorem

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

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

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

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

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

In preparation of proving this we first show

$$t_1 \theta \sim_E t_2 \theta \quad \text{for all substitutions } \theta: X \rightarrow T(\Sigma) \quad (+)$$

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

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

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

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

$$t_1^{T_E(\Sigma), \alpha} = t_2^{T_E(\Sigma), \alpha} \quad (++)$$
Note that for every variable \( x \in X \), \( \alpha(x) \) is an \( \sim_E \)-equivalence class. For every \( x \in X \) chose a term \( \theta(x) \in \alpha(x) \). This defines a substitution \( \theta: X \rightarrow T(\Sigma) \). By definition we have \( \alpha(x) = [\theta(x)]_{\sim_E} \) for every \( x \in X \), i.e. \( \alpha = [\cdot]_{\sim_E} \circ \theta \). Note also that \( [\cdot]_{\sim_E}: T(\Sigma) \rightarrow T_E(\Sigma) \) \( (=T(\Sigma)/\sim_E) \) is a homomorphism. Finally note that the substitution \( \theta \) can also be viewed as a variable assignment for the closed term algebra \( T(\Sigma) \). Baring all this in mind we can now prove \((++\)). We have

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

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

### 6.2.4 Definition

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

\[
\text{Init--Spec}(\Sigma, E)
\]

is called an **initial specification**.

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

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

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

### 6.2.5 Theorem

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

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

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

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

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

\[
A \models t_1 = t_2 \quad \Leftrightarrow \quad \forall E \models t_1 = t_2
\]

(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 \( \Sigma \)-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 \( \text{Init--Mod}_{\Sigma}(E) \) is a monomorphic ADT containing \( T_E(\Sigma) \).

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

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

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

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

Let \( t_1 = t_2 \) be an equation in \( E \) and \( \alpha : X \to A \) a variable assignment, where \( X := \text{FV}(t_1 = t_2) \). We have to show \( t_1^{A,\alpha} = t_2^{A,\alpha} \). Since \( A \) is generated we have for every variable \( x \in X \) a closed term \( \alpha(x) = \theta(x)^A \), i.e. \( \alpha = \theta^A \). According to the substitution theorem 3.5.6 we have

\[ t_1^{A,\alpha} = t_2^{A,\alpha} = (t_1\theta)^A \]

and similarly \( t_1^{A,\alpha} = (t_2\theta)^A \). 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)\( \Rightarrow \) (iii)’. Assume that (v) holds. Since by assumption \( A \in \text{Mod}_{\Sigma}(\forall E) \) we now by initiality of \( T_E(\Sigma) \) that there is unique homomorphism \( \varphi : T_E(\Sigma) \to A \). Using once more the fact that \( t^A = \varphi([t]) \) for all closed \( \Sigma \)-terms (see ‘(iii)\( \Rightarrow \) (iv)’ above) it is plain that our assumption (v) implies that \( \varphi \) is bijective.

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

**6.2.6 Theorem**

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

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

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

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

**Proof.** ‘(i)\( \Rightarrow \) (ii)’. Assume \( A \models \forall(t_1 = t_2) \), and let \( B \) be a generated model of \( \forall E \). By initiality of \( A \) there is a homomorphism \( \varphi : A \to B \). Since \( A \) and \( B \) are both generated \( \varphi \) is surjective. Hence clearly \( B \models \forall(t_1 = t_2) \).
70

(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 \( \theta: X \to T(\Sigma) \), 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.

<table>
<thead>
<tr>
<th>Init Spec</th>
<th>BOOLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>boole</td>
</tr>
<tr>
<td>Constants</td>
<td>T, F : boole</td>
</tr>
<tr>
<td>Operations</td>
<td>( \neg : \text{boole} \to \text{boole} ) ( \text{and, or} : \text{boole} \times \text{boole} \to \text{boole} )</td>
</tr>
<tr>
<td>Variables</td>
<td>( x, y : \text{boole} )</td>
</tr>
<tr>
<td>Equations</td>
<td>( \neg T = F ) ( \neg F = T ) ( \text{and}(T, T) = T ) ( \text{and}(F, x) = F ) ( \text{and}(x, F) = F ) ( \text{or}(x, y) = \neg(\neg x, \neg y) )</td>
</tr>
</tbody>
</table>

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 \( t^{\text{Boole}} = b \) where \( b \in \{T, F\} \), 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, \( t_i^{\text{Boole}} = T \) for \( i = 1, 2 \), then, by \((*)\), it follows \( \forall E \models t_1 = T = t_2 \). Hence, by Theorem 6.2.5 “(v)⇒(i)” it follows that the specification is adequate.

6.2.8 Example

Let \( \Sigma := (\{\text{nat}\}, \{0 : \text{nat, succ: nat} \to \text{nat}, + : \text{nat} \times \text{nat} \to \text{nat}\}) \) and \( E := \{x + 0 = x, x + \text{succ}(y) = \text{succ}(x + y)\} \).
We display the initial specification \( \text{Init-Spec}(\Sigma, E) \) by

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

We can show that this specification is adequate for the standard algebra \( N_{0S^+} \) 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 \( \forall E \models t = \text{succ}^n(0) \). Again, this can easily proven by induction on \( t \).

From this, it also follows that the elements of the carrier set of \( T_E(\Sigma) \) are the equivalence classes \([0], [\text{succ}(0)], [\text{succ}([\text{succ}(0)])], \ldots\) . One has for instance

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

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

and in general

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

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 operation \( \text{succ} \) form a system of generators for \( N_{0S^+} \), since all elements are generated by terms built from 0 and \( \text{succ} \). NAT is freely generated by 0 and \( \text{succ} \), because different terms built from 0 and \( \text{succ} \) denote different numbers.

6.2.9 Example

Let us modify example 6.2.8 as follows. We extend the initial specification by an operation \( - : \text{nat} \times \text{nat} \rightarrow \text{nat} \) and add the two equations
\( x - 0 = x \)

\( \text{succ}(x) - \text{succ}(y) = x - y \)

Let \( E \) be this extended set of equations. We also expand the standard algebra of natural numbers in 6.2.8 by interpreting the new operation, \(-\), by the operation \( - : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \)

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

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

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

holds, whereas clearly

\[ \forall E \not
\]

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 \( 0 - \text{succ}(0) = 0 \) does not hold.

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

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

<table>
<thead>
<tr>
<th>Init Spec</th>
<th>NATSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>boole, nat, set</td>
</tr>
<tr>
<td>Constants</td>
<td>T: boole</td>
</tr>
<tr>
<td></td>
<td>F: boole</td>
</tr>
<tr>
<td></td>
<td>0: nat</td>
</tr>
<tr>
<td></td>
<td>emptyset: set</td>
</tr>
<tr>
<td>Operations</td>
<td>succ: nat → nat</td>
</tr>
<tr>
<td></td>
<td>isempty: set → boole</td>
</tr>
<tr>
<td></td>
<td>insert: set × nat → set</td>
</tr>
<tr>
<td>Variables</td>
<td>x, y: nat,  s: set</td>
</tr>
<tr>
<td>Equations</td>
<td>insert(insert(s, x), x) = insert(s, x)</td>
</tr>
<tr>
<td></td>
<td>insert(insert(s, x), y) = insert(insert(s, y), x)</td>
</tr>
<tr>
<td></td>
<td>isempty(emptyset) = T</td>
</tr>
<tr>
<td></td>
<td>isempty(insert(s, x)) = F</td>
</tr>
</tbody>
</table>

Let $A$ be the classical algebra of finite set of natural numbers with the obvious interpretation of the constants and operation. 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

$$t' = \text{insert}(\ldots\text{insert}(\text{emptyset}, \text{succ}^n(0)), \ldots \text{succ}^n(0))$$

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:

- **write**(x): insert the character $x$ immediately to the left of the cursor;
- **>`: move the cursor one position to the right;
- **<`: move the cursor one position to the left;
- **del**: delete the character immediately to the right of the cursor.
Consider for example the file

`edi\|or`

where the `|` represents the cursor. After entering `<` 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:

\[
([r, i, d, e], [o, r])
\]

\[
([i, d, e], [r, o, r])
\]

\[
([i, d, e], [o, r])
\]

\[
([t, i, d, e], [o, r])
\]

We see that only the elementary operations of adding an element in front of a list, or removing the first element of a list are needed to implement all possible actions of the editor.

To create a file we will use the generator `cf: charlist \times charlist \rightarrow file` and for creating lists of characters the usual generators `nil: charlist` and `cons: char \times charlist \rightarrow charlist`.

In order to keep things simple we stipulate that typing the command `>` 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, ..., z, \omega: char
nil: charlist

Operations
cons: char \times charlist \rightarrow charlist
cf: charlist \times charlist \rightarrow file
write: char \times file \rightarrow file
\langle: file \rightarrow file
\rangle: file \rightarrow file
del: file \rightarrow file

Variables
x: char, l, r: charlist

Equations
newfile = cf(nil, nil)
write(x, cf(l, r)) = cf(cons(x, l), r)
\langle(cf(nil, r)) = cf(nil, r)
\langle(cf(cons(x, l), r)) = cf(l, cons(x, r))
\rangle(cf(l, nil)) = cf(l, nil)
\rangle(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.

(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 \texttt{head} and \texttt{tail} should not be performed on an empty list.

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

(i) \textit{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) \textit{Partial algebras}: We interpret operations like \texttt{head} and \texttt{tail} as \textit{partial functions}. This
means one has to develop a logic (syntax and semantics) for partial algebras.

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

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

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 \texttt{NAT} of example 6.2.8 extended by subtraction, such that \(0 - \texttt{succ}(m)\)
raises an exception.

<table>
<thead>
<tr>
<th>Init Spec</th>
<th>NAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>nat</td>
</tr>
<tr>
<td>Constants</td>
<td>0: nat</td>
</tr>
<tr>
<td>Operations</td>
<td>\texttt{succ}: nat \rightarrow nat</td>
</tr>
<tr>
<td></td>
<td>+: nat x nat \rightarrow nat</td>
</tr>
<tr>
<td></td>
<td>-: nat x nat \rightarrow nat</td>
</tr>
<tr>
<td>Variables</td>
<td>x, y: nat</td>
</tr>
<tr>
<td>Equations</td>
<td>(x + 0 = x)</td>
</tr>
<tr>
<td></td>
<td>(x + \texttt{succ}(y) = \texttt{succ}(x + y))</td>
</tr>
<tr>
<td></td>
<td>(x - 0 = x)</td>
</tr>
<tr>
<td></td>
<td>(0 - 0 = 0)</td>
</tr>
<tr>
<td></td>
<td>(\texttt{succ}(x) - \texttt{succ}(y) = x - y)</td>
</tr>
<tr>
<td></td>
<td>(0 - \texttt{succ}(x) = \texttt{error})</td>
</tr>
</tbody>
</table>

This specification is shorthand for the initial specification containing an extra constant \texttt{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:
### 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 `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:

<table>
<thead>
<tr>
<th>Init Spec</th>
<th>NAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>nat</td>
</tr>
<tr>
<td>Constants</td>
<td>0: nat, error: nat</td>
</tr>
<tr>
<td>Operations</td>
<td>succ: nat → nat</td>
</tr>
<tr>
<td></td>
<td>+: nat × nat → nat</td>
</tr>
<tr>
<td></td>
<td>-: nat × nat → nat</td>
</tr>
<tr>
<td>Variables</td>
<td>x, y: nat</td>
</tr>
<tr>
<td>Equations</td>
<td>x + 0 = x</td>
</tr>
<tr>
<td></td>
<td>x + succ(y) = succ(x + y)</td>
</tr>
<tr>
<td></td>
<td>x - 0 = x</td>
</tr>
<tr>
<td></td>
<td>0 - 0 = 0</td>
</tr>
<tr>
<td></td>
<td>succ(x) − succ(y) = x − y</td>
</tr>
<tr>
<td></td>
<td>0 − succ(x) = error</td>
</tr>
<tr>
<td></td>
<td>succ(error) = error</td>
</tr>
<tr>
<td></td>
<td>error + x = error</td>
</tr>
<tr>
<td></td>
<td>x + error = error</td>
</tr>
<tr>
<td></td>
<td>x − error = error</td>
</tr>
<tr>
<td></td>
<td>error − x = error</td>
</tr>
</tbody>
</table>
The speciﬁcation NATSET of example 6.2.11 can 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):

<table>
<thead>
<tr>
<th>Init Spec</th>
<th>SET(element)</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>BOOLE</td>
</tr>
<tr>
<td>Sorts</td>
<td>set</td>
</tr>
<tr>
<td>Constants</td>
<td>emptyset: set</td>
</tr>
<tr>
<td>Operations</td>
<td>isempty: set → boole</td>
</tr>
<tr>
<td></td>
<td>insert: set × element → set</td>
</tr>
<tr>
<td>Variables</td>
<td>x, y: element, s: set</td>
</tr>
<tr>
<td>Equations</td>
<td>insert(insert(s, x), x) = insert(s, x)</td>
</tr>
<tr>
<td></td>
<td>insert(insert(s, x), y) = insert(insert(s, y), x)</td>
</tr>
<tr>
<td></td>
<td>isempty(emptyset = T)</td>
</tr>
<tr>
<td></td>
<td>isempty(insert(s, x)) = F</td>
</tr>
</tbody>
</table>

The speciﬁcation NATSET of example 6.2.11 can now simply obtained as SET(nat).
Init Spec | EDITOR  
Export | file, newfile, \(\triangleright\), \(\triangleleft\), del, write  
Import | LIST(char)  
Sorts | char, charlist, file  
Constants | newfile: file  
| \(a, \ldots, z, \omega\): char  
| nil: charlist  
Operations | cons: char \times charlist \rightarrow charlist  
| cf: charlist \times charlist \rightarrow file  
| write: char \times file \rightarrow file  
| \(\triangleleft\): file \rightarrow file  
| \(\triangleright\): file \rightarrow file  
| del: file \rightarrow 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)  

6.6 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 Spec\(_1 +\) Spec\(_2\) is a specification.

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

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

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

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

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

Then the models of the specification

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

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

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

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


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). 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.

---

2Algebras with state are often called evolving algebras (Börger), or abstract state machines (Gurevich).
• Specification Languages.

Exercises. 1. Consider the following loose specification LIST(BOOLE):

<table>
<thead>
<tr>
<th>Loose Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
</tr>
<tr>
<td>boole, list</td>
</tr>
<tr>
<td>Constants</td>
</tr>
<tr>
<td>T: boole, F: boole, nil: list</td>
</tr>
<tr>
<td>Operations</td>
</tr>
<tr>
<td>cons: boole × list → list</td>
</tr>
<tr>
<td>first: list → boole</td>
</tr>
<tr>
<td>rest: list → list</td>
</tr>
<tr>
<td>Variables</td>
</tr>
<tr>
<td>x: boole, l: list</td>
</tr>
<tr>
<td>Axioms</td>
</tr>
<tr>
<td>first(cons(x, l)) = x</td>
</tr>
<tr>
<td>rest(cons(x, l)) = l</td>
</tr>
</tbody>
</table>

Let $\Sigma$ be the signature of LIST(BOOLE):

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

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

2. Is the $\Sigma$-algebra $A$, defined in the previous exercise, initial in the class of all models of LIST(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 $(\Sigma', \Phi')$ be an extension of $(\Sigma, \Phi)$ such that for every closed $\Sigma$-formula $P$ it holds that if $\Phi' \models P$ then $\Phi \models P$.

Show that $(\Sigma', \Phi')$ is a persistent extension of $(\Sigma, \Phi)$.

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

Hint: Structural induction on $P$.

7. Extend the specification NATSET in Example 6.2.11 by an operation $\text{member} : \text{nat} \to \text{set} \to \text{boole}$ 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 $\text{select} : \text{set} \to \text{nat}$ and the equations

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

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

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

Hint: Show that for every real number $c \neq 0$ the function $\varphi : A \to A$, defined by $\varphi(a) := c \cdot a$, 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 $\oplus$ only. The only exception are “$x = 0$” and “$x \neq 0$”. We cannot express, for example, “$x > 0$”, or “$x = 1$”.

11. Let $\Sigma$ be the signature with one sort, two constants, 0 and 1, and two binary operations, $\oplus$ and $\cdot$. Let $R$ be the $\Sigma$-algebra of real numbers with 0, 1, addition and multiplication.

(a) Show that “$x < y$” is expressible by a $\Sigma$-formula.

(b) Let $\varphi : R \to R$ 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 $\Sigma$-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.

<table>
<thead>
<tr>
<th>Init Spec</th>
<th>TREE</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>NAT, BOOLE</td>
</tr>
<tr>
<td>Sorts</td>
<td>tree</td>
</tr>
<tr>
<td>Operations</td>
<td>leaf: nat → tree</td>
</tr>
<tr>
<td></td>
<td>branch: tree × nat × tree → tree</td>
</tr>
<tr>
<td></td>
<td>isleaf: tree → boole</td>
</tr>
<tr>
<td></td>
<td>root: tree → nat</td>
</tr>
<tr>
<td></td>
<td>left: tree → tree</td>
</tr>
<tr>
<td></td>
<td>right: tree → tree</td>
</tr>
<tr>
<td>Variables</td>
<td>x: nat, s, t: tree</td>
</tr>
<tr>
<td>Equations</td>
<td>isleaf(leaf(x)) = T</td>
</tr>
<tr>
<td></td>
<td>isleaf(branch(s, x, t)) = F</td>
</tr>
<tr>
<td></td>
<td>root(leaf(x)) = x</td>
</tr>
<tr>
<td></td>
<td>root(branch(s, x, t)) = x</td>
</tr>
<tr>
<td></td>
<td>left(leaf(x)) = error</td>
</tr>
<tr>
<td></td>
<td>left(branch(s, x, t)) = s</td>
</tr>
<tr>
<td></td>
<td>right(leaf(x)) = error</td>
</tr>
<tr>
<td></td>
<td>right(branch(s, x, t)) = t</td>
</tr>
</tbody>
</table>

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.

```haskell
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 `leaf(x)`, or `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 `leaf(x)` and one for trees of the shape `branch(s, x, t)):

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

public class Leaf extends Tree {
    private int label ;
    public Leaf (int x) {
```
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 computing the depth of a tree.

\[
\text{depth: tree} \to \text{nat}
\]

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

```haskell
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:

```java
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
treesucc: nat × 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 \texttt{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 “\texttt{depth(t)} is the depth of \(t\)” holds for terms of the form \texttt{leaf \ x} and is preserved by the operation \texttt{branch}. Both facts are obvious.

3. Consider what would happen if in Section 7.1 we hadn’t made the function \texttt{branch} abstract, but had used the constructor \texttt{Branch} directly: In that case, all expressions of the form \texttt{Branch \ s \ x \ t} would have to be modified to \texttt{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

```haskell
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 `snoc(q, x)` as `q ++ [x]` (appending the singleton list `[x]` to `q`). For brevity we do not implement all operations of Exercise 9. We do also make the implementation polymorphic in the type of elements of a queue.

```haskell
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:

```haskell
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```
Hence $xs ++ ys$ takes $\text{length}(xs)$ many steps and consequently $\text{snoc}(q, x)$ takes $\text{length}(q)$ 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 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 $r$:

\begin{verbatim}
module Queue (Queue,emptyQ,snoc,head,tail) where
import Prelude hiding (head,tail)
\end{verbatim}
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 expect 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 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 \texttt{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 $$\text{length}(f) \geq \text{length}(r)$$ (Exercise 3 in Section 7.6). A more substantial performance improvement can be achieved by a technique called \textit{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

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

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

```haskell
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 \texttt{Queue} \( a \) above is an instance of \textit{polymorphic recursion}.

2. The efficiency of this implementation relies to a large extent that Haskell is a \textit{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 \textit{batched} operation, that is, it needs to be fully executed when needed.

4. In applications were 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

\[
\texttt{data Queue } a = Q \ [a] \ [a]
\]

together with the operations \texttt{snoc}, \texttt{head}, \texttt{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 \texttt{Queue} \( a \) form a \textit{subalgebra} (this requires a proof that the operations preserve legality).

(3) Different legal elements of \texttt{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 \textit{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 \textit{quotient of a subalgebra} of the algebra \texttt{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.

- 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

\[ \text{treesucc} : \text{nat} \times \text{tree} \]

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 \( \text{length}(f) \geq \text{length}(r) \).

4. Extend the data type \texttt{Queue a} defined at the beginning of Section 7.3 by all the operations specified in Exercise 9 of Section 6.7.

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