Contents

Week 1: Definition and Examples of Algebras

- Examples of Algebras
- Signatures, Terms and Σ-structures

1.1: Examples of Algebras

An algebra is just a set with some functions on it. More precisely

Definition 1.1 A *homogenous algebra* A is a pair $\langle |A|; \Omega \rangle$, where

- $|A|$ is a non-empty set, called *carrier*, and
- $\Omega = \{f_i\}_{i \in I}$ is a family of functions, each with a given arity n_i, i.e., $f_i : \underbrace{|A| \times \ldots \times |A|}_{n_i} \to |A|$.

A *many-sorted* or $\mathcal{S}$*-sorted algebra* A is a pair $\langle |A|; \Omega \rangle$ where

- a carrier is an $\mathcal{S}$-sorted family of non-empty sets $|A| = \{A_s\}_{s \in \mathcal{S}}$, and
- Ω is a family of functions $\{f_i\}_{i \in I}$, each with a *profile*, i.e., $f_i : A_1 \times \ldots \times A_{n_i} \to A_{m_i}$, where each $A_k \in |A|$.

Functions with arity 0 are called *constants*. We require the carriers to be non-empty because it will simplify some technical details. The general theory, however, does not require such a restriction and can be developed for algebras with possibly empty carriers.

Example 1.2 [Boolean Algebras]

The two element set $\mathsf{B} = \{\mathtt{tt}, \mathtt{ff}\}$ of truth values can be equipped with many useful functions (Boolean connectives). For instance

$$
\begin{array}{rrcl}
\mathit{not}: & \mathsf{B} & \to & \mathsf{B} \\
\mathit{or}: & \mathsf{B} \times \mathsf{B} & \to & \mathsf{B} \\
\mathit{and}: & \mathsf{B} \times \mathsf{B} & \to & \mathsf{B} \\
\mathit{impl}: & \mathsf{B} \times \mathsf{B} & \to & \mathsf{B} \\
\mathit{eqv}: & \mathsf{B} \times \mathsf{B} & \to & \mathsf{B} \\
\mathtt{ff}: & & \to & \mathsf{B} \\
\mathtt{tt}: & & \to & \mathsf{B}
\end{array}
$$

These are, of course, supposed to have the standard meaning given by the truth tables. Selecting different functions, we can form various algebras over the set B. For instance:

$$B_1 = \langle \mathsf{B}; \mathit{not}, \mathit{or} \rangle \qquad B_2 = \langle \mathsf{B}; \mathtt{ff}, \mathtt{tt}, \mathit{not}, \mathit{and} \rangle \qquad B_3 = \langle \mathsf{B}; \mathtt{ff}, \mathtt{tt}, \mathit{not}, \mathit{or}, \mathit{and}, \mathit{eqv} \rangle \quad \text{etc.}$$

Notice that B_1 has no constants – although $\mathtt{ff}$ and $\mathtt{tt}$ are elements of its carrier, they are not, properly speaking, a part of the algebra.

Similarly, since $\{\mathit{not}, \mathit{and}\}$ is an adequate set of connectives, there isn't really much difference between the algebras B_2 and B_3 except that the latter has more "ready made" operations. Thus the function *impl*, although possible to express in B_2, is not a part of this algebra as a separate operation.

Of course, if instead of the set B we take another two element set, e.g., $\{1, 0\}$, we can interpret the above operations analogically. The algebras we will obtain will be essentially the same as the ones above. We will later define the notion of this "essentially the same" in a precise way.

Example 1.3 [Natural numbers]

Let $\mathsf{N} = \{0, 1, 2, 3, \ldots\}$ be the set of natural numbers. Again, various operations may be added to this set to form various algebras

$$
\begin{array}{rrcll}
0: & & \to & \mathsf{N} & \\
\mathsf{s}: & \mathsf{N} & \to & \mathsf{N} & \mathsf{s}(x) \stackrel{\text{def}}{=} x+1 \\
\mathsf{p}: & \mathsf{N} & \to & \mathsf{N} & \mathsf{p}(x) \stackrel{\text{def}}{=} \begin{cases} x-1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \\
\mathit{add}: & \mathsf{N} \times \mathsf{N} & \to & \mathsf{N} & \mathit{add}(x,y) \stackrel{\text{def}}{=} x+y \\
\mathit{mlt}: & \mathsf{N} \times \mathsf{N} & \to & \mathsf{N} & \mathit{mlt}(x,y) \stackrel{\text{def}}{=} x*y
\end{array}
$$

$N_0 = \langle \mathsf{N}; 0, \mathsf{s} \rangle$ is an example of an algebra over the set of natural numbers with the constant 0 and successor operation.

$N_1 = \langle \mathsf{N}; 0, \mathsf{s}, \mathit{add} \rangle$ is another algebra over the set of natural numbers with the constant 0, and successor and addition functions.

Note that we define these algebras from something "we know to exist". Natural numbers are given with all possible operations on them – in describing an algebra, like N_1, we only choose some of such "existing" operations, or else define new ones on their basis.

Example 1.4 [1.2 continued]

Given an algebra, say B_2 over B, we may extend its carrier with additional elements. For instance, we may take $\mathsf{B} \cup \{\bullet\}$. This puts an obligation to define all the operations on these new elements. One possibility would be to set $\bullet \ \mathit{and}\ x = x \ \mathit{and}\ \bullet = \bullet$ for all x in the carrier.

Alternatively, we may argue that if one of the operands of *and* is $\mathtt{ff}$, then the whole expression is $\mathtt{ff}$, no matter what the second operand is. This leads to another connective, known as *parallel* or *concurrent and* : $\mathit{and}(\bullet, x) = \mathit{and}(x, \bullet) \stackrel{\text{def}}{=} \begin{cases} \mathtt{ff} & \text{if } x = \mathtt{ff} \\ \bullet & \text{otherwise} \end{cases}$. Defining, in addition $\mathit{not}(\bullet) = \bullet$, we obtain the algebra

$$B3 = \langle \{\mathtt{tt}, \mathtt{ff}, \bullet\}; \mathtt{tt}, \mathtt{ff}, \mathit{not}, \mathit{and} \rangle.$$

Such algebras are counterparts of *three-valued* logics, very common in computing where the additional element is used to represent the "undefined" or "unknown" value.

Example 1.5 [Combining algebras]

If we want to add, for instance, a test operation $\mathit{if}(b, x, y) = \begin{cases} x & \text{if } b = \mathtt{tt} \\ y & \text{otherwise} \end{cases}$ to a natural number algebra, say N_1, we need to extend it with the Boolean sort.

Alternatively, we may combine it with an appropriate algebra over B. This leads to the notion of combining algebras. We can obtain, for instance, the following two-sorted algebra by combining B_1 with N_1 and adding $\mathit{if} : \langle \mathsf{N}, \mathsf{B}; 0, \mathsf{s}, \mathit{add}, \mathit{not}, \mathit{and}, \mathit{if} \rangle$.

Example 1.6 [Peano Arithmetics with Booleans]

This is the classical algebra formalizing the natural numbers. It has two sorts with carriers N and B, and can be obtained by combining N_1 with B_2 and adding the four last operations in the following list:

$$
\begin{array}{rrrl}
PA = & \mathsf{N}, \mathsf{B}; & & \\
\Omega: & 0: & \to \mathsf{N} & \\
& \mathtt{tt}: & \to \mathsf{B} & \\
& \mathtt{ff}: & \to \mathsf{B} & \\
& \mathit{not}: & \mathsf{B} \to \mathsf{B} & \\
& \mathit{and}: & \mathsf{B} \to \mathsf{B} & \\
& \mathsf{s}: & \mathsf{N} \to \mathsf{N} & \mathsf{s}(x) \stackrel{\text{def}}{=} x+1 \\
& \mathit{add}: & \mathsf{N} \times \mathsf{N} \to \mathsf{N} & \mathit{add}(x,y) \stackrel{\text{def}}{=} x+y \\
& \mathit{mlt}: & \mathsf{N} \times \mathsf{N} \to \mathsf{N} & \mathit{mlt}(x,y) \stackrel{\text{def}}{=} x*y \\
& \mathit{if}: & \mathsf{B} \times \mathsf{N} \times \mathsf{N} \to \mathsf{N} & \mathit{if}(b,x,y) \stackrel{\text{def}}{=} \begin{cases} x & \text{if } b = \mathtt{tt} \\ y & \text{otherwise} \end{cases} \\
& \mathit{eq}: & \mathsf{N} \times \mathsf{N} \to \mathsf{B} & \mathit{eq}(x,y) \stackrel{\text{def}}{=} \begin{cases} \mathtt{tt} & \text{if } x = y \\ \mathtt{ff} & \text{otherwise} \end{cases} \\
& \mathit{lt}: & \mathsf{N} \times \mathsf{N} \to \mathsf{B} & \mathit{lt}(x,y) \stackrel{\text{def}}{=} \begin{cases} \mathtt{tt} & \text{if } x < y \\ \mathtt{ff} & \text{otherwise} \end{cases}
\end{array}
$$

Example 1.7 [Integers]

A *semigroup* is any set with a binary associative operation. Let $Z = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ be the set of integers. The algebra $\langle Z; +\rangle$ – integers with addition – is an example of a semigroup. Taking, instead, $Z_* = \langle Z; *\rangle$ yields also a semigroup of integers but with multiplication instead of addition.

Since Z contains the elements 0 and 1, we may select these explicitly as constants. $Z_+ = \langle Z; +, 0\rangle$ is a *monoid* (a semigroup with a neutral, or identity, element : $0+x = x+0 = x$). Similarly, $Z_* = \langle Z; *, 1\rangle$, is another monoid structure over integers in which 1 is the neutral element wrt. multiplication.

Finally, adding to the monoid Z_+ the inverse operation $- : Z \to Z$ (i.e., such that $x + (-x) = -x + x = 0$), we obtain $\langle Z; +, 0, -\rangle$ – a well known example of a *group*.

Although "built from the same integers" all these structures are different algebras.

Example 1.8 [Product sorts]

All functions in an algebra have a single target sort. This is so because composite sorts, i.e., sorts built from other sorts, may be expressed as single sorts with appropriate operations. The sort of pairs $\langle a; b\rangle$, where $a \in A$, $b \in B$, i.e., the cartesian product of A and B, can be obtained in an algebra by taking as the carrier the three sorts A, B, $[A \times B]$ and operations:

$$\begin{array}{lll} pr: & A \times B \to [A \times B] & \\ \pi_1: & [A \times B] \to A & \pi_1(pr(a,b)) \stackrel{\text{def}}{=} a \\ \pi_2: & [A \times B] \to B & \pi_2(pr(a,b)) \stackrel{\text{def}}{=} b \end{array}$$

In the programming context such sorts correspond to `records`. E.g., an instance p of

```
record P
  integer year ;
  text name ;
end
```

can be generated by something like `p := new P(1920, John)` (which corresponds to $pr(1920, John)$). The selectors, corresponding to the projection functions, allow one then to access the stored values, e.g., `p.year` will return `1920`.

Example 1.9 [Infinite arrays]

Another composite sort is a functional sort, i.e., a sort whose elements are functions. An example of this cane be a sort of infinite arrays. An infinite array of A-elements is a function $a : N \to A$, where the set N is used as a set of addresses, and the value $a(i)$ is thought of as the value stored at the address i.

Thus, we will have carrier with three sorts: A, N and A^N – the set of functions from N to A! We can then define following functions:

$$\begin{array}{rll} null: & \to A^N & \\ ev: & A^N \times N \to A & ev(a,i) \stackrel{\text{def}}{=} a(i) \\ ins: & A \times N \times A^N \to A^N & ins(x,i,a) \stackrel{\text{def}}{=} b : b(j) = \begin{cases} a(j) & \text{if } j \neq i \\ x & \text{if } j = i \end{cases} \end{array}$$

We can now add more operations on arrays, for instance, the following operation merging two infinite arrays into one:

$$mrg: A^N \times A^N \to A^N \qquad mrg(a,b) \stackrel{\text{def}}{=} c : c(j) = \begin{cases} a(i) & \text{if } j = 2*i \\ b(i) & \text{if } j = 2*i+1 \end{cases}$$

The sort A^N has, possibly, infinitely many elements where each of this elements is itself an infinite structure (an infinite array). Would it be equally easy to form an algebra of finite arrays? Yes, it would, if we had a constant fixed upper bound for each array. But we would like to have a sort of finite arrays of various, potentially unbounded but finite size. In general, modeling such finite structures requires special treatement of *partiality*. This introduces a lot of complications and various modeling techniques, like partial algebras, error algebras, guarded algebras, which we will not address in detail. We will not address partiality and, merely as an indication of a possible treatement, give the example of an algebra of finite arrays.

Example 1.10 [Finite arrays]

Finite arrays can be modeled analogously to the infinite ones but with an additional argument – the upper bound l. The idea here is that for all $i > l$ the value stored at i does not belong to the set of values A but is another – new – element $\bullet$. This element represents "error" situation.

A finite array is then a pair $\langle a; l\rangle$, where $a \in (A \cup \{\bullet\})^N$ and, furthermore $a(i) = \bullet$ for all $i > l$. The carrier of finite arrays of A-elements is

$$A^{\lceil N \rceil} = \{\langle a; l\rangle \in (A^\bullet)^N \times N : \forall i > l : a(i) = \bullet\}$$

Given an algebra with a carrier set A, we can construct the algebra of finite arrays over A as follows. First we form the algebra $A^\bullet$ by augmenting $|A|$ with a new special (error) element $\bullet$ and the corresponding constant. All operations in this new algebra have to be defined on this new element – typically, one extends all the operations in A to $A^\bullet$ by the *strictness assumption*, i.e., whenever any argument of some operation happens to be $\bullet$ then so is the result.

We then add N and $A^{\lceil N \rceil}$ to the carrier. We also need some operations on the arrays, for instance, a constant generating a new, empty array of a given size, and operations for reading and updating arrays:

$$\begin{array}{rlll} null: & N \to A^{\lceil N \rceil} & null(l) \stackrel{\text{def}}{=} \langle a; l\rangle : & a(n) = \bullet \text{ for all } n \in N \\ ev: & A^{\lceil N \rceil} \times N \to A^\bullet & ev(\langle a; l\rangle, n) \stackrel{\text{def}}{=} a(n) & \\ ins: & A^{\lceil N \rceil} \times N \times A^\bullet \to A^{\lceil N \rceil} & ins(\langle a; l\rangle, n, x) \stackrel{\text{def}}{=} \langle b; l\rangle : & b(j) = \begin{cases} a(j) & \text{if } j < l \text{ and } j \neq n \\ x & \text{if } j = n < l \\ \bullet & \text{otherwise} \end{cases} \end{array}$$

1.2: Signatures and Σ-algebras

In the previous section, a confusion might have arisen as to what is the difference between *add* and +, between *mlt* and $*$. We said that carriers and operations "exist" (+, −, 0, 1, 2, etc.) and that designing an algebra we merely choose some operations for some carriers (0, *add*). In universal algebra the procedure is actually the opposite – we hardly ever start with a ready structure which we try to describe. On the contrary, we start with a language, operations and properties of interest and look at the structures which will satisfy these properties. A language is determined by a signature.

Definition 1.11 A (many-sorted) *signature* Σ is a pair $\langle \mathcal{S}; \Omega\rangle$, where

- $\mathcal{S}$ is a non-empty set of *sort symbols*, and
- Ω is an $\mathcal{S}^* \times \mathcal{S}$-indexed family $\{\Sigma_{w,s}\}_{w,s \in \mathcal{S}^* \times \mathcal{S}}$ of sets, where for each w, s, the set $\Sigma_{w,s}$ is the set of *function symbols* with the profile $w \to s$.

Σ is a *subsignature* of Σ' (or the latter is an *expansion* of the former) iff $\mathcal{S} \subseteq \mathcal{S}'$ and for each $w \in \mathcal{S}^*$, $s \in \mathcal{S} : \Sigma_{w,s} \subseteq \Sigma'_{w,s}$.

Since $w \in \mathcal{S}^*$, i.e., it is a string of sort symbols $s_1, s_2, \ldots, s_n$ (possibly with $n = 0$), the profile $\Sigma_{w,s}$ means the profile $s_1 \times s_2 \times \ldots \times s_n \to s$. When $n = 0$, i.e., $w = \varepsilon$, it is a profile of a constant. We will denote the set of all Σ constants by Σ_ε.

Σ is finite iff only finitely many sets $\Sigma_{w,s}$ are non-empty and all such sets are finite. The idea of "choosing" some operations from a structure, for instance, restricting the natural numbers with the inifity of possible operations to, say algebra N_1, comes from the need to work with finite signatures which can address only such "restricted" parts of potentially infinitely rich structures.

Example 1.12 [Signatures]

$\Sigma_N = \langle \underline{N}; \underline{0} : \to \underline{N}, \underline{s} : \underline{N} \to \underline{N}\rangle$ is a possible signature "for" algebra N_0 from example 1.3.

$\Sigma = \langle \underline{B}; \neg : \underline{B} \to \underline{B}, \vee : \underline{B} \times \underline{B} \to \underline{B}\rangle$ is a possible signature "for" algebra B_1 from example 1.2. It is a subsignature of $\Sigma' = \langle \underline{B}, \top : \to \underline{B}, \bot : \to \underline{B}, \neg : \underline{B} \to \underline{B}, \vee : \underline{B} \times \underline{B} \to \underline{B}\rangle$.

$\Sigma_1 = \Sigma_N \cup \langle \emptyset; + : \underline{N} \times \underline{N} \to \underline{N}\rangle$ is a possible signature "for" algebra N_1.

Signature is a purely syntactic entity. In the above example we used underlined symbols in the signature to emphasize that these are only *symbols*: $\underline{\mathsf{N}}$ is a sort symbol in the signature, with which we may associate an actual carrier set, e.g., the natural numbers N. (We won't always be so precise in notation – whether a symbol or an entity in some algebra is meant, should be clear from the context.) What it means that a signature may be a signature "for" a given algebra and, more genrally, how the semantic structures – algebras – are related to the syntax determined by signatures is given in the following definition.

Definition 1.13 Given a $\Sigma = \langle \mathcal{S}; \Omega \rangle$, a Σ-*algebra* A consists of

- an $\mathcal{S}$-indexed family of non-empty sets: for each $s \in \mathcal{S} : A_s$ is the carrier of sort s; and
- an $\mathcal{S}^* \times \mathcal{S}$-indexed family of functions $\{\Sigma^A_{w,s}\}_{w,s \in \mathcal{S}^* \times \mathcal{S}}$: for each $f \in \Sigma_{w,s}$ with $w = s_1 \ldots s_n$, a function $f^A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$.

$\mathsf{Alg}(\Sigma)$ denotes the class of all Σ-algebras.

Notice that the signature itself does *not* put any restrictions on the properties of the operations in the algebra. The only requirement concerns "typing" of these operations which has to conform to their profiles "declared" in the signature. The operation $\neg$ from Σ in example 1.12 can be *any* unary operation on the carrier of $\underline{\mathsf{B}}$. And in a Σ-algebra the carrier of $\underline{\mathsf{B}}$ can be *any* non-empty set.

We will often use A ambiguously for an algebra or its carrier. Also, we will often drop the sort annotations on terms and carrier sets – always assuming that things are correctly sorted. Notice that a Σ-algebra A *must* contain interpretation of all symbols from Σ; but in addition it *may* contain other operations or sorts. Thus, both algebras B_1 and B_3 will be Σ-algebras for Σ from example 1.12. Even the three-valued algebra from example 1.4 will be a Σ-algebra. We say that Σ *is a signature for* a given algebra A if A is a Σ-algebra but is not a Σ'-algebra for any signature $\Sigma' \supset \Sigma$.

Also, each carrier of an algebra must be non-empty and all functions are total. Although it is possible to relax any of these two conditions, we will not do it here.

Remark 1.14 [Unit Algebra]

For any signature Σ there is a special algebra, called the *unit algebra*, with the carrier (for each sort) being a single element, and all functions returning this unique element (of the appropriate sort).

For instance, the unit algebra for the signature Σ_{N} from example 1.12, will have the carrier $\{\bullet\}$ – its element is the interpretation of $\underline{0}$ and also $s(\bullet) = \bullet$.

The unit algebra for Σ' from the same example will also have a single element $\bullet$ which will be the interpretation of both constants $\top$ and $\bot$ and the result of all operations, i.e., $\neg(\bullet) = \bullet$ and $\bullet \vee \bullet = \bullet$.

1.2.1: Σ-TERMS

A signature gives us a syntactic grasp on the algebras. By means of its expressions – terms – we can refer to various elements of the corresponding algebras.

Definition 1.15 Given a Σ and an ($\mathcal{S}$-sorted) set X of variables, the ($\mathcal{S}$-sorted) set of Σ-terms, $\mathcal{T}(\Sigma, X)$, is defined inductively:

1. $X_s \subseteq \mathcal{T}(\Sigma, X)_s$ for each $s \in \mathcal{S}$
2. $\Sigma_{\varepsilon,s} \subseteq \mathcal{T}(\Sigma, X)_s$ for each $s \in \mathcal{S}$
3. if for $1 \le i \le n : t_i \in \mathcal{T}(\Sigma, X)_{s_i}$ and $f \in \Sigma_{s_1 \ldots s_n s}$ then $f(t_1 \ldots t_n) \in \mathcal{T}(\Sigma, X)_s$
4. nothing else belongs to $\mathcal{T}(\Sigma, X)$.

Ground terms, $\mathcal{T}(\Sigma) = \mathcal{T}(\Sigma, \emptyset)$, are terms without any variables, i.e., defined as above with $X = \emptyset$.

Remark 1.16

Note that we did not require the various $\Sigma_{w,s}$ to be disjoint. Thus, the same term may denote operations with different profiles. For instance, having a signature with sorts N and Z, we may have $0 :\to \mathsf{N}$ and $0 :\to \mathsf{Z}$. However, annotating the operation symbols with their profiles disambiguates overloading of the symbols, i.e., we have here 0_{N} and 0_{Z}. Usually no ambiguity can arise, and then we omit such annotations.

Definition 1.17 Given sets of variables X, Y, a *substitution* is any function $\tau : X \to \mathcal{T}(\Sigma, Y)$. The result of applying a substitution τ to any term t, $t\tau$, is defined inductively:

1. $x\tau \stackrel{\text{def}}{=} \tau(x)$, for any $x \in X$;
2. $z\tau \stackrel{\text{def}}{=} z$, for any variable $z \notin X$;
3. $c\tau \stackrel{\text{def}}{=} c$, for all constants;
4. $f(t_1 \ldots t_n)\tau \stackrel{\text{def}}{=} f(t_1\tau ... t_n\tau)$.

Typically, given a substitution $\tau : X \to \mathcal{T}(\Sigma, Y)$, its application is considered as a function $\_\tau : \mathcal{T}(\Sigma, X) \to \mathcal{T}(\Sigma, Y)$ which applies τ to a term $t \in \mathcal{T}(\Sigma, X)$ and all its subterms.

Example 1.18

Consider the signature $\Sigma = \langle \mathsf{N}; 0 :\to \mathsf{N}, \mathsf{s} : \mathsf{N} \to \mathsf{N}, + : \mathsf{N} \times \mathsf{N} \to \mathsf{N} \rangle$, the sets of variables $X = \{x, x_1\}$, $Y = \{x, y\}$, the term $t = x + (\mathsf{s}x_1 + 0) \in \mathcal{T}(\Sigma, X)$, and two substitutions

$$\sigma_1 = \{x \mapsto \mathsf{ss}0, x_1 \mapsto 0\} \ : \ X \to \mathcal{T}(\Sigma) \text{ and}$$

$$\sigma_2 = \{x \mapsto \mathsf{s}y, x_1 \mapsto \mathsf{s}x\} \ : \ X \to \mathcal{T}(\Sigma, Y).$$

The results of applying these substitutions to t will be

$$t\sigma_1 = \mathsf{ss}0 + (\mathsf{s}0 + 0)$$

$$t\sigma_2 = \mathsf{s}y + (\mathsf{ss}x + 0).$$

The substitution σ_1 is ground and so the result of its application is a ground term. Notice that $X \cap Y \neq \emptyset$ – σ_2 substitutes $\mathsf{s}y$ for the variable x but, at the same time, reintroduces this variable in substitution for x_1.

Substitution is a syntactic operation of replacing occurrences of variables by *terms*. There is an analogous semantic operation called *assignment*.

Definition 1.19 Let A be a Σ-algebra and X be a set of variables. An *assignment* is any function $\alpha : X \to |A|$.

Interpretation of ground terms in an algebra A is determined as in definition 1.13. Non-ground terms, containing variables, do not denote any specific elements of the carrier. However, any assignment $\alpha : X \to |A|$, determines an interpretation of all terms $\mathcal{T}(\Sigma, X)$.

Definition 1.20 Given a Σ-algebra A, a set of variables X, and an assignment $\alpha : X \to |A|$, the interpretation of each term $t \in \mathcal{T}(\Sigma, X)$ under α, t^A_α, is given by:

1. $(x)^A_\alpha \stackrel{\text{def}}{=} \alpha(x)$, for all $x \in X$;
2. $c^A_\alpha \stackrel{\text{def}}{=} c^A$, for all constants;
3. $f(t_1 \ldots t_n)^A_\alpha \stackrel{\text{def}}{=} f^A((t_1)^A_\alpha ... (t_n)^A_\alpha)$.

Notice that such an interpretation of terms $\mathcal{T}(\Sigma, X)$ can be seen as an extension of the assignment α to a function $\overline{\alpha} : \mathcal{T}(\Sigma, X) \to A$.

Later, we will study algebras described by sets of formulae in some language. The basis for the languages will be the equality relation interpreted as identity on the algebras.

Definition 1.21 A Σ-equation is a pair $s = t$ where $s, t \in \mathcal{T}(\Sigma, X)$. Let A be a Σ-algebra and $\alpha : X \to |A|$ an assignment:

A satsifies the equation *under assignment* α, $A \models_\alpha s = t$, iff $s^A_\alpha = t^A_\alpha$. [1]

A *satisfies the equation*, $A \models s = t$, iff $A \models_\alpha s = t$ for all assignments $\alpha : X \to |A|$.

Example 1.22

Take the signature Σ' from example 1.12, i.e., $\Sigma' = \langle \underline{\mathsf{B}}; \top, \bot, \neg, \vee \rangle$. Algebra B_2 from example 1.2 is a Σ'-algebra under the interpretation

$$\underline{\mathsf{B}} \mapsto \mathsf{B} \quad \top \mapsto \mathtt{tt} \quad \bot \mapsto \mathtt{ff} \quad \neg \mapsto not \quad \vee \mapsto and \tag{1.1}$$

We have, for instance, $B_2 \models (x \vee \bot) = \bot$. For x can be assigned either $\alpha_t(x) = \mathtt{tt}$ or $\alpha_f(x) = \mathtt{ff}$. In the first case, we obtain $(x \vee \bot)^{B_2}_{\alpha_t} = and(\alpha_t(x), \mathtt{ff}) = and(\mathtt{tt}, \mathtt{ff}) = \mathtt{ff} = \bot^{B_2}$, and in the second $(x \vee \bot)^{B_2}_{\alpha_f} = and(\alpha_f(x), \mathtt{ff}) = and(\mathtt{ff}, \mathtt{ff}) = \mathtt{ff} = \bot^{B_2}$. (Since $\bot$ is a constant its interpretation does not depend on the assignment.)

On the other hand, $B_2 \not\models \neg x = \bot$, for under the assignment $\alpha_f(x) = \mathtt{ff}$, we get $(\neg x)^{B_2}_{\alpha_f} = not(\mathtt{ff}) = \mathtt{tt} \neq \mathtt{ff} = \bot^{B_2}$.

Now, take the algebra $B3$ from example 1.4 – using the same interpretation schema as in (1.1), we make $B3$ into a Σ'-algebra (the carrier is now the set $\mathsf{B} \cup \{\bullet\}$, while *and* is now the parallel and). To check whether $B3 \models (x \vee \bot) = \bot$ we have to consider all possible assignments to x. For $\mathtt{ff}$ and $\mathtt{tt}$ we obtain the same results as in B_2. For $\alpha(x) = \bullet$, we get $(x \vee \bot)^{B3}_\alpha = and(\alpha(x), \mathtt{ff}) = and(\bullet, \mathtt{ff}) = \mathtt{ff} = \bot^{B3}$. So $B3$ satisfies this equation as well.

Since B_2 did not satisfy $\neg x = \bot$, and interpretation of all operations coincides in $B3$ and B_2 on the common part of the carrier, we can immediately conclude that $B3 \not\models \neg x = \bot$.

Notice that, although $\bullet$ is an element of the carrier of $B3$, we cannot write, for instance, the equation $\neg(\bullet) = \bot$, because $\bullet$ does not belong to Σ', i.e., there are no Σ'-terms which would contain something like $\bullet$ – it is a "junk" element wrt. the signature Σ' which has to be taken into account only when considering the possible assignments to the variables.

Equations are used for specifying some desired properties of the algebras one wants to consider.

Example 1.23 [1.22 continued]

The signature, like Σ', specifies only the arities of the operations. Thus we could interpret the symbol $\vee$ as any binary operation on the carrier. E.g., in the example 1.22, we interpreted it as the operation *and* in the algebras B_2 and $B3$. But if we wanted to ensure that $\vee$ corresponds to Boolean *or*, we could require that it satisfies the equations $x \vee \bot = x$ and $x \vee \top = \top$. Neither B_2 nor $B3$ satisfied these equations under the interpretation (1.1), so they wouldn't be valid models of these axioms.

Example 1.24 [1.7 continued]

Sets with one binary operation are very common structures in algebra of which we have already seen several examples. Putting various requirements on this operation by means of equations one obtains different classes of algebras well know in classical mathematics. We have the following hierarchy of algebras with one binary operation $\circ$:

	$\_ \circ \_$ is	$\Sigma = \langle S; ...$	satisfying the axioms
groupoid	binary operation	$\circ : S \times S \to S \rangle$	
semigroup	binary associative	$\circ : S \times S \to S \rangle$	$x \circ (y \circ z) = (x \circ y) \circ z$
monoid	binary associative	$\circ : S \times S \to S$	$x \circ (y \circ z) = (x \circ y) \circ z$
	with identity	$0 : \ \to S \rangle$	$0 \circ x = x \circ 0 = x$
group	binary associative	$\circ : S \times S \to S$	$x \circ (y \circ z) = (x \circ y) \circ z$
	with identity	$0 : \ \to S$	$0 \circ x = x \circ 0 = x$
	and inverse	$\_^- : S \to S \rangle$	$x \circ (x^-) = (x^-) \circ x = 0$

It is obvious that any group is (or may be seen as) a monoid is a semigroup is a groupoid.

[1] Actually, we should use two different symbols: in an equation $s = t$, the equality $=$ is a syntactic operation which *is interpreted* as identity on the respective elements of the carrier $s^A_\alpha = t^A_\alpha$. We hope that the meaning of the same symbol can be disambiguated from the context.

1.2.2: Term Algebras and "Junk"

Observe that if there are no constants in Σ, the set $\mathcal{T}(\Sigma)$ will be empty. When this is not the case, or more precisely, when for each $s \in \mathcal{S}$ there is a $t \in \mathcal{T}(\Sigma)_s$, we can construct a very special and important Σ-algebra.

Definition 1.25 Let Σ be such that for each $s \in \mathcal{S} : \mathcal{T}(\Sigma)_s \neq \emptyset$. Then the Σ-*term algebra*, T_Σ, is given by:

1. for each $s \in \mathcal{S}$, the carrier $(T_\Sigma)_s$ is $\mathcal{T}(\Sigma)_s$
2. if $f \in \Sigma_{s_1 \ldots s_n s}$ and for $1 \leq i \leq n : t_i \in (T_\Sigma)_{s_i}$, then $f^{T_\Sigma}(t_1 \ldots t_n) \stackrel{\text{def}}{=} f(t_1 \ldots t_n)$.

We will always assume that signatures we are considering satisfy the condition of this definition and will call them *non-void*.

Example 1.26 [Term Algebras]

Let $\Sigma_{\mathsf{B}} = \langle \underline{\mathsf{B}}; \top, \bot : \to \underline{\mathsf{B}},\ \neg : \underline{\mathsf{B}} \to \underline{\mathsf{B}},\ \wedge : \underline{\mathsf{B}} \times \underline{\mathsf{B}} \to \underline{\mathsf{B}} \rangle$. The term algebra $T_{\Sigma_{\mathsf{B}}}$ will have the carrier with (inifinitely many) elements: $\top, \bot, \neg\bot, \neg\top, \neg\neg\bot, ..., \top \wedge \bot, \top \wedge \neg\bot, \neg(\top \wedge \bot), ...$

The term algebra for the signature Σ_{N} from example 1.12 will have the carrier $\{\underline{0}, \underline{s0}, \underline{ss0}, \underline{sss0}, ...\}$. Notice that, though this set is bijective with, it is *not the same* as the set of natural numbers $\mathsf{N} = \{0, 1, 2, 3, ...\}$.

The term algebra for Σ_1 from the same example will have a carrier containing, in addition to these elements, all the elements like $+(\underline{s0}, \underline{0})$, $\underline{s}(+(\underline{s0}, \underline{0}))$, $+(+(\underline{s0}, \underline{0}), \underline{ss0})$, etc.

The term algebra is one special algebra whose carrier elements are terms. As the example of Σ_{N} shows, this carrier may correspond very closely to the "intended carrier". This correspondance will often amount to the fact that each element of the intended carrier can be denoted by some ground term from the signature – the carrier contains "no junk".[2]

Example 1.27 [1.4 continued]

The ground terms from the signature Σ_{B}, in fact, merely the constants $\top$ and $\bot$, make it possible to refer to any particular element of the carrier of the algebra B_2 from example 1.2.

The algebra $B3$ from example 1.4, is also a Σ_{B}-algebra but, unlike B_2, it contains element $\bullet$ for which there is no ground term in Σ_{B}. This undenotable element represents a kind of "junk" in relation to the signature: using signature as a kind of a "window" through which we can look at the algebra, there is no way we can see $\bullet$ through Σ_{B}.

Example 1.28 [1.7 continued]

Let $\Sigma_{\mathsf{Z}} = \langle \underline{\mathsf{Z}}; \underline{0} : \to \underline{\mathsf{Z}},\ \underline{s}, \underline{p} : \underline{\mathsf{Z}} \to \underline{\mathsf{Z}} \rangle$ be a signature for integers with $\underline{s}$ and $\underline{p}$ interpreted, respectively, as $\_ + 1$ and $\_ - 1$. It is easy to see that for any $z \in \mathsf{Z}$, there is a ground term over this signature denoting it: $\underline{0}$ for 0, and, for instance, $\underline{s}^n(\underline{0})$ for $n > 0$ and $\underline{p}^n(\underline{0})$ for $n < 0$. (Notice that there will be several other ground terms for each integer.)

Example 1.29 [1.10 continued]

Given Σ_{Z} and the coresponding algebra Z of integers, we can build the algebra $Z^{\lceil \mathsf{N} \rceil}$ of finite arrays of integers. In order to "talk" about its operations, we will add to the signature the sorts for N and $\mathsf{Z}^{\lceil \mathsf{N} \rceil}$, as well as operation symbols, e.g., $new[\_]$ for $null$, $\_(\_)$ for ev and $\_(\_) := \_$ for ins. (I.e., $a(n)$ denotes $ev(a, n)$ and $a(n) := x$ denotes $ins(a, n, x)$.)

Remember that we also added a new element $\bullet$ to indicate the possible "undefined value" error. If we do not add a corresponding symbol to the signature, we will not, in general, be able to "talk about" such error situations. In a specific algebra, for instance, the one constructed as in example 1.10, there may be other terms denoting $\bullet$ – e.g. if $a = new[5]$, then $a(1)^{Z^{\lceil \mathsf{N} \rceil}} = ev(null(5), 1) = \bullet$. But this equality cannot be enforced by means of axioms if there is no constant denoting the error element – in other algebras $ev(null(x), y)$ may return defined values. The same applies to the "outside array bounds" error. E.g., for $a = ins(ins(null(2), 0, 0), 1, 1)$ the expected value of $ev(a, 5)$ will be

[2] The other aspect of this correspondance – "no confusion" – will be discussed later.

undefined, but one may easily imagine algebras where this application returns a defined value. In fact, the lack in many programming languages of a symbol (an operation) for such error situations implies that the respective errors are discovered at run-time and lead to abnormal termination. The programmer has to check explicitly whether indices he is using are within the array bounds. Other programming languages do provide the means for explicitly handling errors and exceptions.

Signatures define the syntax of legal expressions, that is, a language. We now give a simplified example of a more real application of the ideas introduced so far.

Example 1.30 [A Simple Programming Language]
Let us consider the following signature Π (for better readability, we split it into the subsignature Σ_{ET} and the rest):

$$\begin{array}{rlrr}
\Sigma_{ET} = \mathcal{S}: & E,T & & \\
\Omega: & & 0, x_1 \ldots x_n: & \to E \\
& & \mathsf{s},\mathsf{p}: & E \to E \\
& & +,-,*: & E \times E \to E \\
& & _ = _: & E \times E \to T \\
\Pi = \Sigma_{ET} \cup & & & \\
\mathcal{S}: & P & & \\
\Omega: & & _ := _: & E \times E \to P \\
& & _ ; _: & P \times P \to P \\
& & \mathtt{if}\ _\ \mathtt{then}\ _\ \mathtt{else}\ _\ \mathtt{fi}: & T \times P \times P \to P \\
& & \mathtt{repeat}\ _\ \mathtt{do}\ _\ \mathtt{od}: & E \times P \to P
\end{array}$$

The intention is that the terms of sort E are all legal Expressions (for simplicity, only integer expressions), T the (Boolean) Tests, and P the Programs which can be written in this language.

Let us use usual integer symbols instead of s and p terms, e.g., 2 for s(s(0)), -1 for p(0), etc. You should convince yourself that, for instance, '$0 + 5 * 8 = x_1 - 7$', '$x_2 + x_3 = 0$' are Test terms, while, for instance, '$x_1 := 0$', '`repeat` x_2 `do` $x_1 := x_1 + 1$ `od`' are Program terms.

Notice that $x_1 \ldots x_n$, which would correspond to program variables, are here treated as constants! If we did not include them in the signature, we would not be able to write programs with *program* variables. Consequently, all the above terms are ground! For y's being variables of appropriate sorts, we can also write non-ground (program) terms, e.g., '`if` y `then` $y_1 := 5$ `else` $y_2 := x_1$ `fi`'

The signature is not perfect – it allows us, for instance, to write the Program terms '$3 := 2$' or '$x_1 + 1 := 0$'. Designing a signature for a real programming language, one would certainly have to treat more detailed syntactic categories. (You are asked to fix this in exercise 1.3.) However, we can also write "meaningful" programs over Π, for instance:

$x_1 := 1$; $x_2 := 1$;
repeat 5 **do**
  $x_1 := x_1 * x_2$;
  $x_2 := x_2 + 1$
od

The only problem with this "meaningfulness" is that the above term, although most of us will recognize its intended meaning, does not have any meaning whatsoever! It is just a piece of text. What we need is to look at the possible semantics – algebras – for the signature Π. There will be a lot of them. The first may be the term algebra T_Π – it contains all the (ground) Π-terms, that is, all ground Expressions, Tests and, above all, all possible ground Programs. Many algebras will be useless in that they will interpret the Π-operations in a completely unintended way (for instance, the unit algebra). But $\mathsf{Alg}(\Pi)$ will also contain all possible implementations of the Π-language on actual machines! That is, all machines where these operations have been given the expected meaning.

Exercises (week 1)

EXERCISE 1.1 Formulate the signatures for the algebras from example 1.8 and 1.9.

EXERCISE 1.2 Let $\Sigma = \langle \mathsf{B}; \top : \to \mathsf{B}, \bot : \to \mathsf{B}, \vee : \mathsf{B} \times \mathsf{B} \to \mathsf{B} \rangle$. Which among the algebras named in examples 1.2, 1.3 and 1.4 can be considered as Σ-algebras? How do you interpret the Σ-operations in these algebras?

EXERCISE 1.3 Modify the signature Π from example 1.30 by extending it with a new sort V of variables and a new operation for **decl**aring variables: it should take a string (say, an element of a new sort S) and return a new variable. (You may let this operation be of sort P but also, if you like, you may introduce a new sort of Declarations.) You will then have to adapt some of the other operations: in Expression- and Test-building operations one should be able to use variables in the same way as Expressions (recall here remark 1.16). Modify also the $\_ := \_$ operation so that only variables can occur on its left hand side.

Try to sketch an idea of an appropriate algebra for interpreting your signature in the expected way. That is, what are the carriers of various sorts, and how are the operations from your signature evaluated in this algebra?

EXERCISE 1.4 Let S be any non-empty set (for the sake of example, it may have 3 elements), and $\wp(S)$ be its power set, i.e., $\wp(S) = \{X : X \subseteq S\}$. Consider the signature

$$\begin{array}{rll}
\Sigma = \mathcal{S}: & P & \\
\Omega: & 0: & \to P \\
& 1: & \to P \\
& \cap: & P \times P \to P \\
& \cup: & P \times P \to P
\end{array}$$

1. There is a very natural way of turning the set $\wp(S)$ into a Σ-algebra – i.e., of using it as a carrier on which one has defined all Σ-operations. How would you intepret the Σ-operations in order to endow $\wp(S)$ with the algebraic Σ-structure? Does this algebra contain any "junk"? What are the interpretations of terms $0 \cap 1$, $0 \cup 1$ and $0 \cup (1 \cap 0)$ in your algebra?
2. Check whether your interpretation makes the following equations hold in your algebra (X, Y, Z are variables referring to the elements of the carrier – the members of the power set $\wp(S)$, i.e., the subsets of S):
   1. $X \cup Y = Y \cup X$ 4. $X \cap Y = Y \cap X$
   2. $(X \cup Y) \cup Z = X \cup (Y \cup Z)$ 5. $(X \cap Y) \cap Z = X \cap (Y \cap Z)$
   3. $X \cup 0 = X$ 6. $X \cap 1 = X$

   If not, re-interpret Σ in $\wp(S)$ so that these equations hold.
3. Extend now Σ to Σ' by adding a new operation $\overline{} : P \to P$. Find an interpretation for this operation in the algebra you obtained in the previous point, so that the following equations hold:
   7. $X \cup \overline{X} = 1$ 8. $X \cap \overline{X} = 0$

EXERCISE 1.5 Let $\mathsf{B} = \{\mathbf{tt}, \mathbf{ff}\}$ be a two element set. Repeat the previous exercise with this set instead of $\wp(S)$, i.e., turn this set (not its power set!) into a Σ algebra so that it satisfies the axioms 1-6 (notice that here the variables will range over the only two possible elements of B), and then add an interpretation of the additional operation from Σ' so that axioms 7-8 are satisfied.

Week 2: Constructions with Algebras

- Products
- Subalgebras
- Congruences and Quotients

In this section we study various constructions on algebras. We fix a signature Σ and, unless stated otherwise, work within the class $\mathsf{Alg}(\Sigma)$.

2.1: Products

Given some algebras (over the same signature Σ) we may form a new algebra with the carrier being the cartesian product of the carriers of all the component algebras and with the Σ-operations interpreted "pointwise". First we construct a product of only two algebras.

Definition 2.1 Let $A, B \in \mathsf{Alg}(\Sigma)$. The *binary product algebra*, $A \times B \in \mathsf{Alg}(\Sigma)$ is defined as follows:

- the carrier $|A \times B|$ is the cartesian product $|A| \times |B|$ of the carriers (for each sort)
- for each $w = s_1 \ldots s_n$, $f \in \Sigma_{w,s}$ and $\langle a_i, b_i \rangle \in (A \times B)_{s_i}$ for $1 \leq i \leq n$, we let

$$f^{A \times B}(\langle a_1, b_1 \rangle \ldots \langle a_n, b_n \rangle) \stackrel{\text{def}}{=} \langle f^A(a_1 \ldots a_n), f^B(b_1 \ldots b_n) \rangle.$$

Example 2.2 [Product Algebras]
Consider the signature $\langle \underline{\mathsf{N}}, \underline{\mathsf{B}}; \top, \bot : \to \underline{\mathsf{B}},\ \underline{0} : \to \underline{\mathsf{N}},\ \underline{\mathsf{s}} : \underline{\mathsf{N}} \to \underline{\mathsf{N}},\ \underline{leq} : \underline{\mathsf{N}} \times \underline{\mathsf{N}} \to \underline{\mathsf{B}} \rangle$, and the algebra $N = \langle \mathsf{N}, \mathsf{B}; \mathsf{tt}, \mathsf{ff}, 0, \mathsf{s}, leq \rangle$ where $leq(x,y)^N = \mathsf{tt} \Leftrightarrow x \leq y$.

The product $N \times N$ will have the carrier containing all the pairs $\langle n, m \rangle$ for all $n, m \in \mathsf{N}$ and $\langle b, c \rangle$ for $b, c \in \mathsf{B}$. Using the definition of product, we will obtain, for instance, $\top^{N \times N} = \langle \mathsf{tt}, \mathsf{tt} \rangle$ and $\underline{\mathsf{s}}^{N \times N}(\langle n, m \rangle) = \langle \mathsf{s}(n), \mathsf{s}(m) \rangle$.

$\underline{leq}^{N \times N}(\langle n_1, m_1 \rangle, \langle n_2, m_2 \rangle) = \langle leq^N(n_1, n_2), leq^N(m_1, m_2) \rangle$, i.e., it will yield $\top^{N \times N} = \langle \mathsf{tt}, \mathsf{tt} \rangle$ iff $n_1 \leq m_1$ *and* $n_2 \leq m_2$. Thus, although leq is a total ordering on N, the induced interpretation in the product $N \times N$ won't give a total ordering since, for instance, for the pairs $\langle 2, 4 \rangle$ and $\langle 3, 1 \rangle$, $\underline{leq}^{N \times N}$ will give $\langle \mathsf{tt}, \mathsf{ff} \rangle$.

The binary product is actually a special case of the general construction of arbitrary products.

Definition 2.3 Let $\{A_i\}_{i \in I} \subseteq \mathsf{Alg}(\Sigma)$ be any family of Σ-algebras. We can define the *product algebra*, $\Pi A = \prod_{i \in I} A_i$ as follows:

- the carrier $|\Pi A|$ is the cartesian product $\prod_{i \in I} |A_i|$ of the carriers given by

$$(\Pi A)_s \stackrel{\text{def}}{=} \{\iota : I \to \bigcup_{i \in I} (A_i)_s : \forall i \in I\ \iota(i) \in (A_i)_s\}$$

- for each $w = s_1 \ldots s_n$, $f \in \Sigma_{w,s}$ and $(\iota_1 \ldots \iota_n) \in (\Pi A)^w$, we define $f^{\Pi A}(\iota_1 \ldots \iota_n)$ by:

$$f^{\Pi A}(\iota_1 \ldots \iota_n)(i) \stackrel{\text{def}}{=} f^{A_i}(\iota_1(i) \ldots \iota_n(i))$$

The algebra A_i is called the *i-th factor* of ΠA, and if for all $i \in I : A_i = B$ then ΠA is called the *I-th power* of B, which may be denoted B^I.

With each index $i \in I$, we can associate the (family of) *projection functions*, $\pi_i : \Pi A \to A_i$, defined by

$$\pi_i(\iota) \stackrel{\text{def}}{=} \iota(i) \tag{2.2}$$

We will later study some special properties of this construction. At the moment, we only register the obvious fact.

Fact 2.4 For any family $\{A_i\}_{i \in I} \subseteq \mathsf{Alg}(\Sigma)$ the product $\Pi A \in \mathsf{Alg}(\Sigma)$.

Observe the difference between product sorts (example 1.8) and product algebras. The former is a construction *within* a specific algebra. The latter is a construction *on* algebras which yields a new, in general, "more complicated" algebra (example 2.2). The two constructions are similar if we consider very simple algebras. Let Σ have only one sort symbol S and A, B be two Σ-algebras. Then their product algebra $A \times B$ will have the carrier which is a cartesian product $A_S \times B_S$, and we will have two projection functions, as in (2.2), $\pi_1(\langle a; b \rangle) = a$ and $\pi_2(\langle a; b \rangle) = b$. This algebra will be thus very similar, yet by no means the same, as the algebra with two sorts and their product sort.

2.2: Subalgebras

Definition 1.13 of a Σ-algebra requires merely that all the operation symbols be interpreted – in particular, that all constants and, consequently, ground terms have an interpretation. If the carrier of an algebra contains more elements than those interpreting the ground terms (i.e., some "junk"), they can be "removed" leading to a new algebra. More precisely:

Definition 2.5 Let A and B be two Σ-algebras. B is a *subalgebra* of A, $B \sqsubseteq A$, iff

- for each $s \in \mathcal{S} : B_s \subseteq A_s$, and
- for all $f \in \Sigma_{w,s}$ and $b^w \in B^w : f^B(b^w) = f^A(b^w)$.

A is *minimal* iff it has no *proper* subalgebra B, i.e., no $B \sqsubseteq A$ such that $B \neq A$.

Since all functions in an algebra are total, once an element is in the carrier all functions must have well-defined values when applied to this element. We say that an algebra must be *closed under all operations*: if $(a_1 \ldots a_n) \in |A|^w$ then $f^A(a_1 \ldots a_n) \in |A|$.

Example 2.6 [1.27 continued]
Both algebras $B3$ from example 1.4 and B_2 from example 1.2 are Σ_{B} algebras. In fact, since $B3$ merely extended the interpretation of operations to the new element $\bullet$, B_2 is a proper subalgebra of $B3$. I.e., $B3$ is not a minimal Σ_{B} algebra. B_2, on the other hand, is minimal: we cannot remove any elements from its carrier since each one is an interpretation of some constant from $\Sigma_{\mathsf{B}} : \mathsf{tt} = \top^{B_2}$ and $\mathsf{ff} = \bot^{B_2}$.

Lemma 2.7 Let A, B, $C \in \mathsf{Alg}(\Sigma)$. If $B \sqsubseteq A$, $C \sqsubseteq A$ and $|C| \subseteq |B|$ then $C \sqsubseteq B$.

Proof. For each constant $c \in \Sigma$ we have $c^C = c^A = c^B$, and for any $(c_1, \ldots, c_n) \in |C|^w$ and $f \in \Sigma_{w,s}$ we have $f^C(c_1, \ldots, c_n) = f^A(c_1, \ldots, c_n) = f^B(c_1, \ldots, c_n)$. (The first of the equalities holds because $C \sqsubseteq A$ and the second one because $B \sqsubseteq A$.)

A very important special case of subalgebras occurs when we consider only that part of the carrier which is "generated" by some subset of its elements.

Lemma 2.8 Let $A \in \mathsf{Alg}(\Sigma)$ and $X \subseteq |A|$ be any subset of the carrier. Define

$$A\lceil X \rceil = \bigcap \{B : X \subseteq |B| \ \wedge\ B \sqsubseteq A\}. \tag{2.3}$$

If $A\lceil X \rceil \neq \emptyset$ then it is the carrier of the smallest subalgebra $A\lceil X \rceil \sqsubseteq A$ containing X.

Proof. First, since $A\lceil X \rceil$ is an intersection of A's subalgebras, for any constant $c \in \Sigma$, we will have $c^A \in A\lceil X \rceil$. So consider any function symbol $f \in \Sigma_{w,s}$ and $(x_1, \ldots, x_n) \in |A\lceil X \rceil|^w$. By definition $(x_1, \ldots, x_n) \in |B|^w$ for all $B \sqsubseteq A$ with $X \subseteq |B|$ and therefore $f^A(x_1, \ldots, x_n) \in \bigcap \{B : X \subseteq |B| \ \wedge\ B \sqsubseteq A\}$. Hence (if $A\lceil X \rceil \neq \emptyset$) $A\lceil X \rceil \sqsubseteq A$.

Obviously $X \subseteq |A\lceil X \rceil|$. Suppose that there is a proper subalgebra $C \sqsubseteq A\lceil X \rceil$ with $X \subseteq |C|$ and $C \sqsubseteq A$. Then $C \in \{B : X \subseteq |B| \ \wedge\ B \sqsubseteq A\}$, and so $|A\lceil X \rceil| \subseteq |C|$ contradicting the assumption that C is a proper subalgebra of $A\lceil X \rceil$.

The subalgebra $A\lceil X\rceil$ is the least subalgebra of A containing X. It can be thought of as the closure of the set X under all the operations as they are defined in A.

Of particular importance are (sub)algebras which are generated by the (interpretation of) ground terms.

Definition 2.9 Let $A \in \mathsf{Alg}(\Sigma)$.

- A is *generated by* an $X \subseteq |A|$ iff $A = A\lceil X\rceil$.
- A is *generated* iff it is generated by $\emptyset$.
- B is a *generated subalgebra* of A iff $B \sqsubseteq A$ and B is generated.

Example 2.10 [2.6 continued]

Take the algebra $B3$ and let $X = \{\bullet\}$. Then $B3\lceil X\rceil = B3$, i.e., it is generated by this additional element. Since the elements $\mathtt{tt}, \mathtt{ff}$ interpret the constants from Σ_{B} as given in example 2.6, they will have to be present in all subalgebras of $B3$, i.e., for any X, the definition (2.3) from lemma 2.8 will yield a $B3\lceil X\rceil$ containing at least these two elements.

If we start with the empty subset of $B3$, we will get $B3\lceil\emptyset\rceil = B_2$. In fact, $B_2\lceil\emptyset\rceil = B_2$, i.e., B_2 is a generated subalgebra of $B3$.

Observe, that generatedness depends crucially on the signature. If, instead of Σ_{B} from example 2.6, we take $\Sigma_{\mathsf{B}}^{\bullet} = \Sigma_{\mathsf{B}} \cup \{\underline{\bullet} : \to \underline{\mathsf{B}}\}$ and consider $B3$ as a $\Sigma_{\mathsf{B}}^{\bullet}$-algebra it will be generated. Now $\bullet = \underline{\bullet}^{B3}$, i.e., it became an interpretation of the $\Sigma_{\mathsf{B}}^{\bullet}$-constant in the carrier of $B3$ and it cannot be removed from the carrier. In this setting, B_2 won't be a subalgebra of $B3$ since B_2 isn't a $\Sigma_{\mathsf{B}}^{\bullet}$-algebra.

A generated subalgebra of A can be thought of as that part of A which can be denoted by some ground term. In fact, an algebra A is generated iff for each $a \in |A|$ there is a $t \in \mathcal{T}(\Sigma)$ such that $a = t^A$. This fact is an immediate consequence of the following lemma.

Lemma 2.11 Let Σ be a non-void signature, Σ_ε the set of all constants in Σ, and $A \in \mathsf{Alg}(\Sigma)$. Then the following are equivalent:

1. A is minimal;
2. $A = A\lceil\emptyset\rceil$;
3. $A = A\lceil\Sigma_\varepsilon^A\rceil$.

Proof. $1 \Rightarrow 2$) Since Σ is non-void, $A\lceil\emptyset\rceil \neq \emptyset$ and so, by lemma 2.8, $A\lceil\emptyset\rceil \sqsubseteq A$. Since A is minimal, it has no proper subalgebra, and so $A = A\lceil\emptyset\rceil$.
$2 \Rightarrow 3$) $A\lceil\emptyset\rceil = \bigcap\{B : \emptyset \subset |B| \ \wedge\ B \sqsubseteq A\}$. In particular, since each B is a subalgebra of A, $\Sigma_\varepsilon^A = \Sigma_\varepsilon^B \subseteq |B|$, and so $A\lceil\emptyset\rceil = \{B : \Sigma_\varepsilon^A \subseteq |B| \ \wedge\ B \sqsubseteq A\} = A\lceil\Sigma_\varepsilon^A\rceil$.
$3 \Rightarrow 1$) Assume $A = A\lceil\Sigma_\varepsilon^A\rceil$. For any $B \sqsubseteq A$, $\Sigma_\varepsilon^A \subseteq |B|$, hence $B \sqsubseteq A \Rightarrow B = A$, i.e., A has no proper subalgebra.

Example 2.12 [2.10 continued]

The Σ_{B} algebra $B3$ from example 2.6 had a subalgebra B_2 and hence was not minimal. But B_2 was a minimal Σ_{B}-algebra. Notice, however, that it was *not* a term algebra: its carrier did not contain terms but truth values $\{\mathtt{tt}, \mathtt{ff}\}$. Also, the term algebra $T_{\Sigma_{\mathsf{B}}}$ from example 1.26 had infinitely many elements, while B_2 contains only two. However, as one can easily see (cf. lemma 2.14), this algebra is also minimal! Thus, the fact that an algebra is minimal does not mean that it has "smallest possible" number of elements – only that it does not contain any proper subalgebra. (In particular, the unit algebra, will also be a minimal Σ_{B}-algebra – which is not a subalgebra of either B_2 nor $T_{\Sigma_{\mathsf{B}}}$.)

Lemma 2.11 illustrates also the fact that any Σ-algebra must, at least, contain the interpretation of all the ground terms. If it does not contain anything else, it will be minimal. If it does, then these "additional" elements are a kind of "junk" – they cannot be reached, denoted by any ground term from the signature. Nevertheless, such an algebra will contain a minimal subalgebra.

Corollary 2.13 Let Σ be non-void. For any $A \in \mathsf{Alg}(\Sigma)$ there is a minimal subalgebra of A.

Proof. By lemma 2.8, $A\lceil\emptyset\rceil \sqsubseteq A$ which is minimal by lemma 2.11.

A very important minimal algebra is the term algebra.

Lemma 2.14 For a non-void Σ, T_Σ is minimal.

Proof. Obviously $T_\Sigma = T_\Sigma\lceil\Sigma_\varepsilon\rceil$, so by lemma 2.11, it is minimal.

Example 2.15 [Free Term Algebras]

More generally, let Σ be an arbitrary signature and X be an ($\mathcal{S}$-sorted) set of variables. Like in definition 1.25, we can form algebra $T_{\Sigma,X}$ of terms $\mathcal{T}(\Sigma, X)$ (definition 1.15). Thus, for instance, the algebra $T_{\Sigma_{\mathsf{B}},\{x\}}$ will, in addition to all the ground terms $\mathcal{T}(\Sigma)$, contain all Σ_{B}-terms containing x.

Notice that, for any set X, we will have $T_\Sigma \sqsubseteq T_{\Sigma,X}$. The algebra T_Σ is minimal (and generated) when Σ is non-void. The algebra $T_{\Sigma,X}$ will be generated if $\Sigma \cup X$, considered as a signature (with new constants X) is non-void, even if Σ is void.

Lemma 2.16 For any non-void $\Sigma \cup X$, the term algebra $T_{\Sigma,X}$ is generated by X.

Proof. We have to show that $T_{\Sigma,X} = T_{\Sigma,X}\lceil X\rceil$. Let $B \sqsubseteq T_{\Sigma,X}$ be any subalgebra of $T_{\Sigma,X}$ containing X. Then $X \subseteq |B|$, $\Sigma_\varepsilon \subseteq |B|$, and $f(t_1,\ldots,t_n) \in |B|$ whenever $t_1,\ldots,t_n \in |B|$ and $f \in \Sigma$. But this means that $\mathcal{T}(\Sigma, X) \subseteq |B|$, i.e., $T_{\Sigma,X} \sqsubseteq B$, and hence $B = T_{\Sigma,X}$.

The construction of the $T_{\Sigma,X}$ algebra in example 2.15 is actually a special case of a construction which can be done on any algebra. In example 1.4 we started with a Boolean algebra B_2 and extended it by adding one element to the carrier and defining the operations appropriately. The *free extension* – the construction from example 2.15 – is different in that we add new elements to the carrier whenever some operations must be defined anew.

Definition 2.17 Let A be any Σ-algebra and $X \cap A = \emptyset$ be a set. The *free extension of* A over X is the algebra $A\lfloor X\rfloor$ obtained as follows. The carrier is the carrier of A extended with the set X and, for all $f \in \Sigma_{w,s}$ and all $(a_1..x..a_n) \notin A^w$, the new element $\bullet_{f(a_1..x..a_n)}$. For all constants $c \in \Sigma$, we define $c^{A\lfloor X\rfloor} \stackrel{\text{def}}{=} c^A$ and all other $f \in \Sigma_{w,s}$:

$$f^{A\lfloor X\rfloor}(x_1,\ldots,x_n) \stackrel{\text{def}}{=} \begin{cases} f^A(x_1,\ldots,x_n) & \text{if } (x_1,\ldots,x_n) \in A^w \\ \text{the new element } \bullet_{f(x_1,\ldots,x_n)} & \text{if some } x_i \notin A \end{cases}$$

The carrier of $A\lfloor X\rfloor$ will be a strange mixture of the elements from A, X and the new elements, but it is a perfectly legal definition. It is easy to verify that, if X is a set of variables and we take the new elements to be the terms containing these variabels, then $T_{\Sigma,X} = T_\Sigma\lfloor X\rfloor$.

2.3: Congruences and Quotients

Given a set X an equivalence on X is a binary relation $\equiv\, \subseteq X \times X$ which is reflexive, symmetric and transitive. $\equiv$ partitions X into a set of disjoint *equivalence classes* $[x] = \{y \in X : y \equiv x\}$, and the quotient construction yields the set of such classes $X/_\equiv\ = \{[x] : x \in X\}$.

Remark 2.18

Note that the elements of $X/_\equiv$ are sets, namely the equivalence classes. Since these are new individuals, it is usually easier to work with only single representatives of such classes. The set of such representatives is, in certain contexts, called the set of *cannonical representatives* or *normal forms* for $X/_\equiv$.

When several equivalences on a given set may be involved, we will sometimes annotate the equivalence classes by the respective equivalence, e.g., $[x]_\equiv$.

Example 2.19

Consider the set of (almost) all pairs of integers $Q = \{\langle z; v\rangle : z, v \in \mathsf{Z} \wedge v \neq 0\}$. Each such pair $\langle z; v\rangle$ can be thought of as a representation of a rational number $\frac{z}{v}$. Obviously, for instance, the pairs $\langle 2; 4\rangle$,

$\langle 3;6\rangle$, $\langle 9;18\rangle$ all represent the rational number $\frac{1}{2}$, i.e., they are *equivalent* representations, or they represent the same number. This "represent the same rational number" is an equivalence relation on the set Q! More precisely, we can define this equivalence by (we write these pairs as fractions, i.e., $\frac{z}{v}$ instead of $\langle z;v\rangle$):

$$\frac{z_1}{v_1} \equiv \frac{z_2}{v_2} \Leftrightarrow z_1 * v_2 = v_2 * z_1. \tag{2.4}$$

First, we have to verify that it really is an equivalence.

1. $z * v = v * z \Rightarrow \frac{z}{v} \equiv \frac{z}{v}$, so the relation is reflexive;
2. $z_1 * v_2 = v_1 * z_2 \Rightarrow z_2 * v_1 = v_2 * z_1$, i.e., $\frac{z_1}{v_1} \equiv \frac{z_2}{v_2} \Rightarrow \frac{z_2}{v_2} \equiv \frac{z_1}{v_1}$, so the relation is symmetric;
3. assume that $\frac{z_1}{v_1} \equiv \frac{z_2}{v_2}$ and $\frac{z_2}{v_2} \equiv \frac{z_3}{v_3}$, i.e., $z_1 * v_2 = v_1 * z_2$ and $z_2 * v_3 = v_2 * z_3$. This implies that $z_1 * v_2 * v_3 = v_1 * z_2 * v_3$ which, by the second equation, gives $z_1 * v_2 * v_3 = v_1 * v_2 * z_3$. Since $v_2 \neq 0$, we can simplify obtaining $z_1 * v_3 = v_1 * z_3$, i.e., $\frac{z_1}{v_1} \equiv \frac{z_3}{v_3}$, and so the relation is transitive.

We can then form a quotient $\mathsf{Q} = Q/_{\equiv}$. Its members are equivalence classes – the sets of pairs which represent the same rational number. To compute with this set, for instance, to perform addition on rational numbers represented this way may seem a bit complicated, but we will see in a moment how this can be done.

Gathering elements into one equivalence class corresponds to making them "indisitnguishable" (equivalent) – they can be treated as one element. In the context of algebras, such indistinguishability would also require that we cannot see the difference when applying operations to such elements, i.e., if $a \equiv b$ then applying $f(a)$ should not yield a result distinguishable from applying $f(b)$. This requirement is formalized as *congruence* – an equivalence relation which is "compatible" with the operations in the algebra.

Definition 2.20 Let $A \in \mathsf{Alg}(\Sigma)$ and $\equiv$ be an equivalence on $|A|$ (i.e., an $\mathcal{S}$-indexed family of equivalences on the carrier of each sort). Then $\equiv$ is a Σ *congruence* on A iff for each $f \in \Sigma_{w,s}$ with $w = s_1 \ldots s_n$, and each $(a_1 \ldots a_n)$, $(b_1 \ldots b_n) \in A^w$

$$\text{if } a_i \equiv b_i \text{ for } 1 \leq i \leq n \text{ then } f^A(a_1 \ldots a_n) \equiv f^A(b_1 \ldots b_n)$$

Example 2.21 [2.19 continued]

In example 2.19 we defined an equivalence $\equiv$ on the set Q. There were no operations defined on this set, and so this equivalence will automatically be a congruence. Let us now define an operation of multiplication on Q by

$$\frac{z}{v} * \frac{a}{b} \stackrel{\text{def}}{=} \frac{z * a}{v * b}. \tag{2.5}$$

In other words, we are now considering the algebra $\mathsf{Q}_* = \langle Q, *\rangle$. We have seen that $\equiv$ is an equivalence and it only remains to verify that it is a congruence. So assume $\frac{z_1}{v_1} \equiv \frac{z_2}{v_2}$ and $\frac{x_1}{y_1} \equiv \frac{x_2}{y_2}$, i.e., $z_1 * v_2 = v_1 * z_2$ and $x_1 * y_2 = y_1 * x_2$. We have to show that this implies $\frac{z_1 * x_1}{v_1 * y_1} \equiv \frac{z_2 * x_2}{v_2 * y_2}$, i.e., $(*)$ $(z_1 * x_1) * (v_2 * y_2) = (v_1 * y_1) * (z_2 * x_2)$.

Multiplying both sides of the respective equations we obtain that $(z_1 * v_2) * (x_1 * y_2) = (v_1 * z_2) * (y_1 * x_2)$. But this means exactly that $(*)$ holds, and so implies that $\equiv$ is a congruence wrt. $*$.

Example 2.22 [Identity and Unit Congruences]

Given an $A \in \mathsf{Alg}(\Sigma)$, the identity $=_A$ on A, i.e., the relation $= \{(a,a) : a \in |A|\}$ is the *finest* congruence on A, with equivalence classes being singletons $[a]^{=_A} = \{a\}$. The unit congruence $\bullet_A = A \times A$ is the *coarest* congruence, which makes any two elements of A congruent.

We obviously have $=_A \subseteq \bullet_A$, and this is the general notion: $\equiv_1$ is *finer* then $\equiv_2$ iff $\equiv_1 \subset \equiv_2$. It is also easy to see that this relation holds iff for any $a \in |A| : [a]^{\equiv_1} \subset [a]^{\equiv_2}$. That $=_A$ is *the* finest congruence means in particular that for *any* other congruence $\equiv$ on A we will have: $a = b \Rightarrow a \equiv b$.

These two congruences exist for any $A \in \mathsf{Alg}(\Sigma)$. $=_A$ is obviously the least reflexive relation on A – it is easy to verify that it satisfies also the other congruence axioms. $\bullet_A$ "collapses" all elements to one and therefore will also satisfy all congruence axioms. If they are the only ones, we say that A is *simple*. Otherwise, there are other congruences which lie "between" $=_A$ and $\bullet_A$.

Example 2.23 [Integers `modulo n`]

Let Z be the algebra $\langle \mathsf{Z}; 0, +, -\rangle$ – the group of integers with addition $+$ and inverse $-$ over the signature $\Sigma_Z = \langle \underline{\mathsf{Z}}; \underline{0}, \underline{+}, \underline{-}\rangle$. For an $n \in \mathsf{N}$, define the following relation

$$z \equiv^n y \Leftrightarrow z \equiv y \ mod \ n, \tag{2.6}$$

i.e., iff there is a $k \in \mathsf{Z}$ such that $z - y = n * k$ (i.e., $z - y$ is divisible by n). For any n, it is a well known equivalence on Z:

1. obviously, $z - z = n * 0$, i.e., $z \equiv^n z$;
2. if $z \equiv^n y$ then $y \equiv^n z$; and
3. if $x \equiv^n y$ and $y \equiv^n z$, then $x - y = k_1 * n$ and $y - z = k_2 * n$, hence $x - z = (k_1 + k_2) * n$, i.e., $x \equiv^n z$.
4. It is also a Σ_Z-congruence: if $z \equiv^n y$ then obviously $-z \equiv^n -y$; and if $z_1 \equiv^n y_1$ and $z_2 \equiv^n y_2$, i.e., $z_1 - y_1 = k_1 * n$ and $z_2 - y_2 = k_2 * n$, then $(z_1 + z_2) - (y_1 + y_2) = (z_1 - y_1) + (z_2 - y_2) = (k_1 + k_2) * n$, i.e., $(z_1 + y_1) \equiv^n (z_2 + y_2)$.

For any n, the congruence yields n equivalence classes $[0], [1], \ldots, [n-1]$, with $[i] = \{i + k*n : k \in \mathsf{Z}\}$ for $0 \leq i < n$. The largest among such congruences will be $\equiv^1$ which would collapse all integers into one equivalence class $[0]$. The larger n, the finer will the congruence be, though for $\equiv^n \subset \equiv^m$ it obviously wouldn't suffice that $n > m$ (that, for some positive $k : n = k * m$, would). The finest among these congruences will again be the identity which, a bit oddly, could be here called "congruence modulo 0" and denoted $\equiv^0$.

As illustrated by the above examples, it may be possible to define several congruences on a given algebra. The following lemma states a general fact about the set of all congruences on any algebra.

Lemma 2.24 Let $\equiv(A)$ be the set of all Σ congruences on $A \in \mathsf{Alg}(\Sigma)$ ordered by $\subseteq$. $\equiv(A)$ is a complete lattice.

Proof. A complete lattice is a PO where each subset of elements has a lub and glb.[3] (In particular, there is the least and the greatest element.) Obviously, $=_A$ and $\bullet_A$ are, respectively, the least and the greatest element of $\equiv(A)$.

Let $\{\equiv_i\}_{i \in I} \subseteq \equiv(A)$. The glb is given by $\equiv \stackrel{\text{def}}{=} \bigcap_{i \in I} \equiv_i$. Since each $\equiv_i$ is reflexive, symmetric and transitive, so is $\equiv$. To show that it is also a congruence, consider any $f \in \Sigma$ and $a_j \equiv b_j$ for $1 \leq j \leq n$. Then $a_j \equiv_i b_j$ for all $i \in I$ and, since each $\equiv_i$ is a congruence, $f^A(a_1, \ldots, a_n) \equiv_i f^A(b_1, \ldots, b_n)$. But then also $f^A(a_1, \ldots, a_n) \equiv f^A(b_1, \ldots, b_n)$.
This is enough, for by a general fact, any PO P where each $Y \subseteq P$ has a glb is a complete lattice. (Let $X \subseteq P$, $Y = \{p \in P : p \geq X\}$ and $s = glb(Y)$. Then $s = lub(X)$, for $s \geq X$, and if $p \geq X$ then $p \in Y$, i.e., $s \leq p$.)

The congruence condition is exactly what is required to generalize the quotient construction on sets to algebras.

Definition 2.25 Let $\equiv$ be a Σ congruence on $A \in \mathsf{Alg}(\Sigma)$. The *quotient algebra* $A/_{\equiv}$ is a Σ-algebra given by:

- for each $s \in \mathcal{S}$, the carrier $(A/_{\equiv})_s \stackrel{\text{def}}{=} A_s/_{\equiv_s}$
- for each $f \in \Sigma_{w,s}$ and each $([a_1] \ldots [a_n]) \in |A/_{\equiv}|^w : f^{A/_{\equiv}}([a_1] \ldots [a_n]) \stackrel{\text{def}}{=} [f^A(a_1 \ldots a_n)]$.

Since on the right side of the last equation we might choose different representatives from each $[a_i]$ in order to determine the value of the function in $A/_{\equiv}$, we have to check that this value does not depend on the representatives chosen. This is exactly the place where we need the congruence condition.

[3] 'Least upper' and 'greatest lower' bounds, respectively. In a PO $\langle P, \leq\rangle$, a $lub(X)$ for an $X \subseteq P$ is defined as this element $p \in P$ for which $p \geq X$ (meaning $p \geq x$ for all $x \in X$) and, furthermore, whenever $q \geq X$ then $q \geq p$. The notion of glb is defined dually.

Lemma 2.26 $A/_{\equiv}$ is a Σ-algebra.

PROOF. For all Σ-constants we have $c^{A/\equiv} = [c^A]$. We only have to check that the second point yields well-defined functions. Assume $a_1 \equiv b_1, ..., a_n \equiv b_n$, for some $a_i, b_i \in |A|$. We must show that

$$f^{A/\equiv}([a_1]\ldots[a_n]) = f^{A/\equiv}([b_1]\ldots[b_n]) \tag{2.7}$$

i.e., that the values of function applications do not depend on the choice of the representatives. Since $\equiv$ is a congruence, we have that $f^A(a_1\ldots a_n) \equiv f^A(b_1\ldots b_n)$, i.e., $[f^A(a_1\ldots a_n)] = [f^A(b_1\ldots b_n)]$. But this implies exactly the required equation (2.7).

The lemma ensures that, if the equivalence is a congruence, then the results of applying operations in the quotient do not depend on the representatives chosen. That is, we may choose arbitrary representatives from each equivalence class and perform the required operation in the original algebra – the result in the quotient will be the equivalence class of the result thus obtained.

Example 2.27 [2.21 continued]

In example 2.21 we have shown that $\equiv$ was a congruence on the algebra $\mathsf{Q}_* = \langle Q, *\rangle$. Hence, multiplication in the quotient $\mathsf{Q}_*/_{\equiv}$ can be done by choosing arbitrary (most convenient) representatives from each class and multiplying them according to the definition (2.5) from example 2.21. For instance, $[\frac{3}{6}] * [\frac{16}{8}]$, in $\mathsf{Q}_*/_{\equiv}$ can be computed by choosing the representatives $\frac{1}{2}$ and $\frac{2}{1}$, multiplying them in Q_*, which yields $\frac{2}{2}$, and concluding that the result will be $[\frac{2}{2}] = [\frac{1}{1}]$.

Example 2.28 [2.23 continued]

We saw that $\equiv^n$ is a congruence on a Σ_Z-algebra Z. Thus, according to the lemma, $Z_n = Z/_{\equiv^n}$ is also a Σ_Z-algebra. Let us see what the operations $+$ and $-$ do in this quotient algebra.

Fix for example's sake $n = 5$ – then the carrier is the set of equivalence classes $\{[0], [1], [2], [3], [4]\}$. What is, for instance, $[3] + [4]$? From the definition 2.25, we have that $[3] + [4] = [3 + 4] = [7] = [2]$. In fact, addition in the quotient Z_n will be the expected addition modulo n!

What about the inverse? For instance, what is $-[3]$? From the congruence we have $-[3] = [-3]$, and we show that $[-3] = [2]$. Again from the congruence, the identity in Z_n is the equivalence class of the identity in Z, i.e., $[0]$. We must verify that $-[-3] + [2] = [3] + [2] = [0]$, which holds since $2 + 3 = 1 * 5 + 0$.

Thus, the quotient of the group Z by the congruence $\equiv^n$ gives us the group Z_n of integers modulo n.

2.3.1: Induced Congruences

If the carrier of an algebra is infinite, specifying the entire congruence relation might be rather time-consuming. Typically, one specifies congruence by a few equalities which *induce* a congruence. More generally, any binary relation can be closed to a minimal congruence according to the following definition.

Definition 2.29 Let A be a Σ-algebra and let $E \subseteq |A| \times |A|$ be a binary relation on $|A|$. The *congruence closure* of E is $\equiv_E \stackrel{\text{def}}{=} \bigcap\{\equiv \,:\, E \subseteq \equiv \;\wedge\; \equiv \in \equiv(A)\}$.

The set $\{\equiv \,:\, E \subseteq \equiv \;\wedge\; \equiv \in \equiv(A)\}$ is not empty since it contains, at least, the unit congruence. By (the proof of) lemma 2.24, $\equiv_E$ is a congruence and it is the least congruence containing E.

Example 2.30 [Congruence induced by equations]

Typically, a congruence is induced by a finite set of equations. For instance, in the example 2.23, the algebra $Z/_{\equiv_n}$ was obtained from Z by forcing the equations $z = z' \Leftrightarrow \exists k : z - z' = k * n$ to hold.

Let $A \in \mathsf{Alg}(\Sigma)$ and E' be a set of Σ-equations, i.e., pairs $s = t$ for some $s, t \in \mathcal{T}(\Sigma, X)$. The initial relation E on $|A|$ is obtained by interpreting all equations from E' in A for all assignments. Let X be all the variables in E'. If $\alpha : X \to |A|$ is an assignment, we let t^A_α be the element of the carrier denoted by the term t^A under this assignment (cf. definition 1.20). We get $E \stackrel{\text{def}}{=} \{\langle s^A_\alpha, t^A_\alpha\rangle : s = t \in E' \;\wedge\; \alpha : X \to |A|\}$ and carry out the construction from definition 2.29.

For instance, given $f(t, x) = g(s(x), y) \in E'$ and an α, $f(t, x)^A_\alpha$ is the element $f^A(t^A, \alpha(x))$. For such an equation we include, in E, the pairs $\langle f^A(t^A, \alpha(x)), g^A(s^A(\alpha(x)), \alpha(y))\rangle$, for all assignments $\alpha : \{x, y\} \to |A|$, and then carry out the construction.

A special, and very important case of the quotient construction, is taking a quotient of a term algebra induced by some equations.

Example 2.31 [2.15 continued]

Take the term algebra T_{Σ_1} from example 1.26 – the one with the carrier $\{0, \mathsf{s}0, \mathsf{ss}0, \mathsf{sss}0, ...\}$ and all combinations of +terms, like $+(0,0), +(\mathsf{s}0, 0), +(+(\mathsf{ss}0, \mathsf{s}0), \mathsf{sss}0)$... (We do not bother to underline the terms here.) Let $\equiv$ be the congruence induced on this algebra by the following equations:

$$
\begin{aligned}
+(x, 0) &= x && (2.8)\\
+(x, \mathsf{s}y) &= \mathsf{s}(+(x, y)) && (2.9)
\end{aligned}
$$

The equivalence class of 0 will then be $[0] = \{0, +(0,0), +(0, +(0,0)), +(+(0,0), 0), ...\}$, of $\mathsf{s}0$: $[s0] = \{\mathsf{s}0, \mathsf{s}(+(0,0)), \mathsf{s}(+(0, +(0,0))), ..., +(\mathsf{s}0, 0), +(\mathsf{s}0, +(0,0)), ..., +(0, \mathsf{s}0), +(+(0,0), \mathsf{s}0), ...\}$. The inclusion of the second and the following terms is forced by $0 \equiv +(0,0)$, congruence and because $\mathsf{s}0$ is included. The term $+(\mathsf{s}0, 0)$ is included by equality (2.8), the following ones by congruence, and $+(0, \mathsf{s}0)$ by equation (2.9), and the rest by congruence.

Example 2.32 [1.26 continued]

The signature $\Sigma_{\mathsf{B}} = \langle \underline{\mathsf{B}}; \top, \bot : \to \underline{\mathsf{B}},\ \neg : \underline{\mathsf{B}} \to \underline{\mathsf{B}},\ \wedge : \underline{\mathsf{B}} \times \underline{\mathsf{B}} \to \underline{\mathsf{B}}\rangle$ is a natural choice for a (minimal) signature for a Boolean algebra, for instance, algebra B_2 from example 1.2. However, we observed that the term algebra $T_{\Sigma_{\mathsf{B}}}$ will have the carrier with (inifinitely many) elements: $\top, \bot, \neg\bot, \neg\top, \neg\neg\bot, ..., \top \wedge \bot, \bot \wedge \top, (\bot \wedge \bot) \wedge \top, \top \wedge \neg\bot, ...$.

What is needed to make $T_{\Sigma_{\mathsf{B}}}$ into a "real" Boolean algebra B is to take an appropriate quotient, namely, such a one which would identify any $t \in \mathcal{T}(\Sigma_{\mathsf{B}})$ with either $\top$ or $\bot$. This can be done very simply, for instance, by the equation $x = \bot$. The intention is, however, that the operations from Σ_{B} be interpreted in the quotient B as usual Boolean operations. For instance, we would like to have $\neg(\bot \wedge \neg\top)$, when interpreted in B, yield the same as $\top$. This effect may be achieved by taking $B = T_{\Sigma_{\mathsf{B}}}/_{\equiv_E}$, where E is the set of Σ_{B}-equations corresponding to the truth table definitions of the operators $\neg$ and $\wedge$:

$$
E = \quad
\begin{array}{rr}
\top \wedge \top = \top & \bot \wedge \bot = \bot \\
\top \wedge \bot = \bot & \bot \wedge \top = \bot \\
\neg\top = \bot & \neg\bot = \top
\end{array}
$$

In exercise 2.8, you are asked to verify that the carrier of B will have only two elements (two equivalence classes) and that the interpretation of the operations will correspond to the usual Boolean operators.

Finally, we register a useful fact relating congruences and subalgebras.

Lemma 2.33 If A is generated by a set X, then $A/_{\equiv}$ is generated by $X/_{\equiv}$.

PROOF. Assume, contrapositively, that $A/_{\equiv}$ isn't generated by $X/_{\equiv}$, i.e., there is a proper subalgebra $B_1 \sqsubseteq A/_{\equiv}$ which contains $X/_{\equiv}$. We show that then $A \neq A\lceil X\rceil$ by showing that $B = \{b : [b] \in B_1\}$ is a proper subalgebra of A containing X.

Since B_1 is a proper subalgebra of A, there is at least one $[a] \in A/_{\equiv}$ such that $[a] \notin B_1$. So $B \subset A$. Since $X/_{\equiv} \subseteq B_1$, we have that $X \subseteq B$. We have to show that B is a Σ-algebra. Since $B_1 \sqsubseteq A/_{\equiv}$, the interpretation of all constants $c^{A/\equiv} = [c^A] \in B_1$, and so $c^A \in B$. Take any $b_1, \ldots, b_n \in B$ and a function symbol f. Since B_1 is a subalgebra, we have $[f^A(b_1, \ldots, b_n)] = f^{A/\equiv}([b_1], ..., [b_n]) \in B_1$. But then $[f^A(b_1, \ldots, b_n)] \subset B$ and, in particular, $f^A(b_1, \ldots, b_n) \in B$. So $X \subseteq B$, $B \sqsubseteq A$ and $B \neq A$, which implies $A \neq A\lceil X\rceil$.

Exercises (week 2)

EXERCISE 2.1 Take the signature with three constants $\Sigma_3 = \langle D; a, b, c : \to D\rangle$ and three sets $S_1 = \{0\}$, $S_2 = \{0, 1\}$ and $S_3 = \{0, 1, 2\}$ and define

1. a Σ_3-algebra G on the carrier S_1, i.e., an interpretation of Σ_3-symbols in the set S_1;
2. a generated Σ_3-algebra A on the carrier S_2;
   a Σ_3-algebra A_0 on the carrier S_2 such that $G \sqsubseteq A_0$;
   a Σ_3-algebra A_1 on the carrier S_2 such that $G \not\sqsubseteq A_1$;
3. a generated Σ_3-algebra B on the carrier S_3;
   a Σ_3-algebra B_0 on the carrier S_3 such that $A_0 \sqsubseteq B_0$;
   a Σ_3-algebra B_2 on the carrier S_3 such that $A \sqsubseteq B_2$;
   a Σ_3-algebra B_3 on the carrier S_3 such that $A \not\sqsubseteq B_3$ and $A_0 \not\sqsubseteq B_3$.

EXERCISE 2.2 Take the signature Σ_{N} from example 1.12 and the algebra $Z = \langle \mathsf{Z}; 0, +1\rangle$ of integers with $Z_{\underline{\mathsf{N}}} = \mathsf{Z}$, $\underline{0}^Z = 0$ and $\underline{\mathsf{s}}^Z = +1$. Is Z a generated Σ_{N}-algebra? What is the minimal subalgebra of Z?

EXERCISE 2.3 Suppose that A and B both are minimal Σ-algebras. Is also $A \times B$ a minimal Σ-algebra?

EXERCISE 2.4 Prove the statements made in example 2.22, i.e., that: identity $=_A$ and unit $\bullet_A$ relations on any A are congruences, and $=_A \subseteq \bullet_A$, i.e., $a =_A b \Rightarrow a \bullet_A b$.

EXERCISE 2.5 Proceeding as in example 2.27, define the addition on the set Q (i.e., an algebra $\mathsf{Q}_+ = \langle Q; +\rangle$ instead of Q_*) which would correspond to addition of rational numbers. Verify that the equivalence $\equiv$ from example 2.19 is a congruence wrt. to this addition. Use then this result and your definition of addition in Q_+ to evaluate $[\frac{3}{6}] + [\frac{14}{8}]$, $[\frac{7}{14}] + [\frac{19}{38}]$ in $\mathsf{Q}_+/_\equiv$.

EXERCISE 2.6 Let $N = \langle \mathsf{N}; 0, +1\rangle$ be the Σ_{N}-algebra of natural numbers, and $\equiv$ be a congruence on N induced by the equation $\underline{\mathsf{ss0}} = \underline{0}$. What will be the carrier of $N/_\equiv$? What will be the interpretation of term $\underline{\mathsf{sss0}}$ in this quotient algebra?

EXERCISE 2.7 Let Σ_{N} and N be as in exercise 2.6 and $x \notin \mathsf{N}$. What will be the carrier of the free extension $N\lfloor x\rfloor$? What will be the carrier of the quotient $N\lfloor x\rfloor$ by the congruence $\equiv$ from exercise 2.6?

EXERCISE 2.8 Refer to example 2.32

1. What will be the carrier of $B = T_{\Sigma_{\mathsf{B}}}/_{\equiv_E}$?
2. Evaluate the following terms in B (i.e., determine to which equivalence class they will belong – use the simplest possible representatives to represent each equivalence class):
   $\neg\neg\neg(\neg\bot \wedge \neg\top)$, and $\top \wedge \neg(\neg(\bot \wedge \top) \wedge (\top \wedge \top))$.
3. Which of the following statements are true (s, t are arbitrary Σ_{B}-terms)?
   (a) if $s^B = \top^B$ then $(\neg s)^B = \bot^B$;
   (b) if $s^B = \bot^B$ then $(\neg s)^B = \bot^B$;
   (c) if $s^B = t^B = \top^B$ then $(s \wedge t)^B = \top^B$;
   (d) if either $s^B = \bot^B$ or $t^B = \bot^B$ then $(s \wedge t)^B = \bot^B$;
   (e) $B \models \forall x(x = \top \vee x = \bot)$.

....
....

EXERCISE 2.9 Let Σ_1, Σ_2 be non-void signatures and $\Sigma_1 \subseteq \Sigma_2$. Is $\mathcal{T}(\Sigma_1) \subseteq \mathcal{T}(\Sigma_2)$? Is $T_{\Sigma_1} \sqsubseteq T_{\Sigma_2}$?

EXERCISE 2.10 Let Σ be non-void, $A \in \mathsf{Alg}(\Sigma)$ and $C \sqsubseteq A$ and $B \sqsubseteq A$. Show that

1. there is a subalgebra $M \sqsubseteq A$ such that $M \sqsubseteq B$ and $M \sqsubseteq C$;
2. there is an M like in 1 and such that, whenever for some other subalgebra $N \sqsubseteq B$ and $N \sqsubseteq C$, then $N \sqsubseteq M$.

(Lemma 2.7 is a special case of these facts.)

Week 3: Homomorphisms

- Definition
- Homomorphisms and Constructions on Algebras
- Homomorphism Theorems

Definition 3.1 Let $A, B \in \mathsf{Alg}(\Sigma)$. A Σ-*homomorphism* $\phi : A \to B$ is an $\mathcal{S}$-indexed family of mappings $\phi = \langle \phi_s : A_s \to B_s \rangle_{s \in \mathcal{S}}$, such that for each $f \in \Sigma_{w,s}$ with $w = s_1 \dots s_n$, and each $(a_1 \dots a_n) \in A^w$:

$$\phi_s(f^A(a_1 \dots a_n)) = f^B(\phi_{s_1}(a_1) \dots \phi_{s_n}(a_n)) \tag{3.10}$$

$$\begin{tikzcd} A_{s_1} \times \dots \times A_{s_n} \arrow[r, "f^A"] \arrow[d, "\phi_{S_1}"'] & A_s \arrow[d, "\phi_s"] \\ B_{s_1} \times \dots \times B_{s_n} \arrow[r, "f^B"] & B_s \end{tikzcd}$$

In particular, for any constant $c \in \Sigma$, the homomorphism condition (3.10) specializes to the requirement that $\phi(c^A) = c^B$.

Saying homomorphism we always mean Σ-homomorphism for a given Σ. Also, we will not be so carefull to emphasize that a ϕ is an $\mathcal{S}$-indexed family of mappings – saying that it is injective (or whatever) we will implicitly mean that each ϕ_s is so.

Example 3.2 [1.24 continued]

The homomorphism condition (3.10) amounts to the requirement that it preserves the algebraic Σ-structure. For instance, sets have no structure so mappings between them are usually just set-functions. Semigroups, on the other hand, have the structure: they are algebras over signature with one binary operation (which, in addition, must be associative), and this operation will have to be preserved by the semigroup morphisms. Thus, a semigroup morphism g between $\langle S_1; \circ_1 \rangle$ and $\langle S_2; \circ_2 \rangle$ will be just a set-function $S_1 \to S_2$ which, in addition, preserves the operations, i.e., $g(x \circ_1 y) = g(x) \circ_2 g(y)$. For the hierarchy of classes of algebras from the example 1.24, the homomorphism condition will translate into the following requirements:

a morphism of	between	must satisfy
groupoids	$g : \langle R_1; \circ_1 \rangle \to \langle R_2; \circ_2 \rangle$	$g(x \circ_1 y) = g(x) \circ_2 g(y)$
semigroups	$g : \langle S_1; \circ_1 \rangle \to \langle S_2; \circ_2 \rangle$	$g(x \circ_1 y) = g(x) \circ_2 g(y)$
monoids	$g : \langle M_1; \circ_1, 0_1 \rangle \to \langle M_2; \circ_2, 0_2 \rangle$	$g(x \circ_1 y) = g(x) \circ_2 g(y)$ $g(0_1) = 0_2$
groups	$g : \langle G_1; \circ_1, 0_1, {}^{-1} \rangle \to \langle G_2; \circ_2, 0_2, {}^{-2} \rangle$	$g(x \circ_1 y) = g(x) \circ_2 g(y)$ $g(0_1) = 0_2$ $g(x^{-1}) = g(x)^{-2}$

Notice that just like any group is a monoid is a semigroup and a groupoid, so any group morphism is a monoid morphism is a semigroup morphism and a groupoid morphism.

Example 3.3 [2.28 continued]

Take $\Sigma = \langle \mathsf{N}; 0, \mathsf{s}, + \rangle$ and consider the usual Σ-algebra N of natural numbers and the algebra N_n of natural numbers modulo n (which also is a Σ-algebra by the same argument as in example 2.23). Can we construct any homomorphisms between them for, say, $n = 4$?

For the first, there is no homomorphism $\psi : N_n \to N$. For if there were one, it would have to preserve the operations, in particular, $\psi(0^{N_n}) \stackrel{(3.10)}{=} 0^N$ and $\psi(\mathsf{s}^{N_n}(z)) \stackrel{(3.10)}{=} \mathsf{s}^N(\psi(z))$. But $\mathsf{s}^{N_4}(3) = 0^{N_n}$, so we would get $\mathsf{s}^N(\psi(3^{N_4})) = \psi(\mathsf{s}^{N_4}(3)) = \psi(0^{N_n}) = 0^N$. But there is no natural number in N whose successor is 0, i.e., there is no possible image of the element $\mathsf{s}^{N_4}(3)$ which would satisfy the homomorphism condition (3.10).

Let us see if the mapping $\nu : N \to N_4$, defined by $\nu(x) = x \ mod\ 4$ will work. We do have $\nu(0^N) = 0 \ mod\ 4 = 0^{N_4}$. We also have $\nu(\mathsf{s}^N(x)) = x + 1 \ mod\ 4 = (x \ mod\ 4) + (1 \ mod\ 4) = \mathsf{s}^{N_4}(\nu(x))$, as well as the required equality $\nu(x +^N y) = \nu(x) +^{N_4} \nu(y)$ – which follow by the same calculation as in example 2.23. Thus, we have verified that the mapping ν is a Σ-homomorphism from N into N_4.

Example 3.4 [3.3 continued]

Cnsider the algebra BN^2 of two-bits' words, i.e., with the carrier $\{00, 01, 10, 11\}$ over the signature Σ from example 3.3. We can define binary addition of such words (treating the rightmost bit as the least significant) which will give, for instance, $00 + 10 = 10$, $10 + 01 = 11$, $11 + 01 = 00$, $10 + 10 = 00$, etc.

As it is typically done in computers, we can consider these words to be binary representations of the natural numbers (modulo 4), i.e., word $b10b_0$ represents the number $b_1 * 2 + b_0$. Defining the s-operation on these words by $\mathsf{s}(b_1 b_0) = b_1 b_0 + 01$ we obtain a Σ-algebra. There are obvious homomorphisms $\phi : BN^2 \to N_4$ sending each word onto the number it represents and $\psi : N_4 \to BN^2$ sending each number (modulo 4) onto its binary representation.

More generally, for an arbitrary n, a binary word $b_{n-1} b_{n-2} \dots b_1 b_0$ of length n can represent the number $(b_{n-1} * 2^{n-1}) + (b_{n-2} * 2^{n-2}) + \dots (b_i * 2^i) + \dots (b_0 * 2^0)$. Generalizing the above mappings, we will obtain the homomorphisms $\phi : BN^n \to N_{2^n}$ from the binary representations of length n to the natural numbers modulo 2^n, and vice versa, $\psi : N_{2^n} \to BN^n$.

It is easy to verify that homomorphisms compose (exercise 3.4). Now, one of the central concepts in algebraic data types to be studied later is that of isomorphism.

Definition 3.5 For any algebra A there is a special *identity* homomorphism $id_A : A \to A$ defined by $id_A(a) = a$ for all $a \in |A|$.
An *isomorphism* is a homomorphism $\phi : A \to B$ such that there is a homomorphism $\phi^{-1} : B \to A$ with $\phi \circ \phi^{-1} = id_A$ and $\phi^{-1} \circ \phi = id_B$. If there is an isomorphism between A and B, we write $A \simeq B$.

Notice that for two algebras there may be several homomorphisms between them and not all of them need to be isomorphisms. But if only there exists at least one isomorphism, we call the two algebras isomorphic.

Example 3.6 [Isomorphism]

Consider two algebras over the signature Σ_{N} from example 1.12:

- the term algebra $T_{\Sigma_{\mathsf{N}}} = \langle \mathcal{T}(\Sigma); 0, \mathsf{s} \rangle$ with the carrier $\{0, \mathsf{s}0, \mathsf{ss}0, \mathsf{sss}0, \dots\}$ where the successor operation is simply $\mathsf{s}^{T_{\Sigma_{\mathsf{N}}}}(x) \stackrel{\text{def}}{=} \mathsf{s}x$; and
- the algebra $N = \langle \mathsf{N}; 0, +1 \rangle$ over natural numbers, where $0^N = 0$ and $\mathsf{s}^N(x) \stackrel{\text{def}}{=} x + 1$.

Define the homomorphism $\phi : N \to T_{\Sigma_{\mathsf{N}}}$ by $\phi(n) \stackrel{\text{def}}{=} \mathsf{s}^n(0)$, i.e., the image of a natural number n is the term with n-succesor applications to the term 0. It is easy to verify that ϕ is a homomorphism. For the constant 0, we have $\phi(0) = 0$, and for any n we have $\phi(\mathsf{s}^N(n)) = \phi(n+1) = \mathsf{s}^{n+1}0 = \mathsf{ss}^n0 = \mathsf{s}(\phi(n)) = \mathsf{s}^{T_\Sigma}(\phi(n))$.

We can define the converse mapping $\phi^- : T_{\Sigma_{\mathsf{N}}} \to N$ by $\phi^-(0) \stackrel{\text{def}}{=} 0$ and $\phi^-(\mathsf{s}x) \stackrel{\text{def}}{=} \phi^-(x) + 1$. It is equally trivial to verify that ϕ^- is a homomorphism.

We then have $\phi^-(\phi(n)) = \phi^-(\mathsf{s}^n(0)) = \underbrace{0 + 1 + \dots + 1}_{n} = n$, i.e., $\phi \circ \phi^- = id_N$. Similarly, for any $x = \mathsf{s}^n0 \in \mathcal{T}(\Sigma)$ we have $\phi(\phi^-(\mathsf{s}^n0)) = \phi(\underbrace{0 + 1 + \dots + 1}_{n}) = \phi(n) = \mathsf{s}^n0$, i.e., $\phi^- \circ \phi = id_{T_{\Sigma_{\mathsf{N}}}}$. Thus the term algebra $T_{\Sigma_{\mathsf{N}}}$ with successor and the algebra N over natural numbers with $+1$ are isomorphic.

Example 3.7 [3.4 continued]

The pairs ϕ, ψ of the homomorphisms from example 3.4 gave another example of isomorphism, for any n, between the algebra of binary representations BN^n and the algebra of naturals modulo 2^n. (These algebras were over signature with $0, \mathsf{s}$ and $+$, but one can easily verify that these mappings will remain isomorphisms if we extend the signature with, for instance, substraction and define this operation in both algebras in the natural way.)

In the example 3.3, on the other hand, we have shown that $N \not\simeq N_4$ – although there was a homomorphism $\nu : N \to N_4$, ther was no, in particular no inverse, homomorphism from N_4 to N.

Isomorphism means in the mathematical context "indistinguishable structure". Isomorphic algebras may have different carriers and operations may perform very different functions, but as far as the structures, or properties of these structures are concerned, they cannot be distinguished. In a sense, the only difference between two isomorphic algebras is the way in which they "represent" or "realize" the symbols from the signature. The s-operation in T_Σ amounts merely to adding one occurence of s to the argument term – it is a kind of unary representation of natural numbers. In N, this operation is realized by adding 1 to the argument. They work on different carriers – different representations – but these carriers are in a one-to-one correspondence not only as sets but as algebraic structures.

In exercise 3.5, you are asked to show that a homomorphism is an isomorphism iff it is both surjective and injective.

3.1: Homomorphisms and Subalgebras

The facts we emphasize here are that an image of a homomorphism is a subalgebra of the target, and that homomorphisms are uniquely determined by their image of the generating sets.

Lemma 3.8 Let $A, B \in \mathsf{Alg}(\Sigma)$ and $\phi : A \to B$ be a homomorphism. Define $\phi[A]$ as the subset of $|B|$ which is the image of A under ϕ, i.e.,

$$\phi[A] = \{b \in |B| : \exists s \in \mathcal{S},\ \exists a \in A_s : b = \phi_s(a)\}$$

Then $\phi[A] \sqsubseteq B$.

Proof. Obviously, for all constants $c \in \Sigma : c^B = \phi(c^A) \in \phi[A]$. We must show that $\phi[A]$ is closed under the operations in B, i.e., if $(b_1, \dots, b_n) \in (\phi[A])^w$ then $f^B(b_1, \dots, b_n) \in \phi[A]$. We have that for each $1 \leq i \leq n$ there is an $a_i \in A : b_i = \phi(a_i)$. Thus $f^B(b_1, \dots, b_n) = f^B(\phi(a_1), \dots, \phi(a_n)) = \phi(f^A(a_1, \dots, a_n))$ since ϕ is a homomorphism, and so $f^B(b_1, \dots, b_n) \in \phi[A]$ because $f^A(a_1, \dots, a_n) \in A$.

An important property of generating sets is that their image determines a unique homomorphism.

Lemma 3.9 Let $A, B \in \mathsf{Alg}(\Sigma)$, $X \subseteq |A|$, $A = A\lceil X\rceil$ and $\phi, \psi : A \to B$ be Σ-homomorphisms. Then $\phi = \psi \ \Leftrightarrow\ \forall x \in X : \phi(x) = \psi(x)$.

Proof. $\Rightarrow$) is obvious. So assume $\forall x \in X : \phi(x) = \psi(x)$. Let $Y \subseteq |A|$ be $\{a \in A : \phi(a) = \psi(a)\}$. We want to show that $Y = |A|$. Y is non-empty since $X \subseteq Y$ and $\Sigma_\varepsilon^A \subseteq Y$ (and $A = A\lceil X\rceil$, i.e., $X \cup \Sigma_\varepsilon^A \neq \emptyset$). Furthermore Y is closed under operations of A, since whenever $a_1, \dots, a_n \in Y$ we have

$$\phi(f^A(a_1, \dots, a_n)) = f^B(\phi(a_1), \dots, \phi(a_n)) = f^B(\psi(a_1), \dots, \psi(a_n)) = \psi(f^A(a_1, \dots, a_n))$$

Thus $Y \subseteq |A|$ – in fact, Y has the Σ-structure and $Y \sqsubseteq A$. But since $A = A\lceil X\rceil$ and $X \subseteq Y$, lemma 2.8 implies that $Y = A$.

Corollary 3.10 Let Σ be non-void, $A, B \in \mathsf{Alg}(\Sigma)$, $\phi : A \to B$, and A be minimal:

1. for any $\psi : A \to B : \phi = \psi$;
2. if also B is minimal and there is a $\psi : B \to A$ then $A \simeq B$.

Proof. The first point says that all homomorphisms from a minimal algebra are unique; the second that two minimal algebras with homomorphisms between them are isomorphic.

1. By lemma 2.11 $A = A\lceil\emptyset\rceil$, so it follows directly from lemma 3.9.

2. By point 1. both ϕ and ψ are unique. Now, $\phi \circ \psi$ is a homomorphism $A \to A$, and by lemma 3.9 it is unique. However, there is also the identity homomorphism $id_A : A \to A$, so we must have $\phi \circ \psi = id_A$. In the same way, $\psi \circ \phi = id_B$. Hence $A \simeq B$.

As we saw in example 2.12, the number of elements in the carrier of an algebra has nothing to do with minimality. This corollary provides a much better "characterization" of minimal algebras – there is at most one homomorphism from such an algebra to another one. Also, by lemma 3.8, if there is a homomorphism into a minimal algebra, it must be surjective – for otherwise, its image would be a proper subalgebra. Thus, if we want to show that an algebra is not minimal but have difficulties with showing a proper subalgebra, we may either try to construct a non-surjective homomorphism from another algebra into it or, alternatively, construct two distinct homomorphisms from it to some other algebra.

Example 3.11

Let Σ contain one sort S and two constants $a, b : \to S$. Consider the following four Σ-algebras A, B, C, D : $A_S = \{1, 2\}$ with $a^A = 1$, $b^A = 2$, $B_S = \{0\}$ with $a^B = b^B = 0$, $C_S = \{1, 2, 3, 4\}$ with $a^C = 1$, $b^C = 2$ and $D_S = \{4, 5, 6\}$ with $a^D = b^D = 5$.

Verify that both A and B are minimal while neither C nor D are. Thus, according to corollary 3.10, any homomorphism from A and B must be unique. In fact, there is a homomorphism from A to any other algebra determined by the interpretation of a and b. $\omega_B : A \to B$ given by $\omega_B(1) = \omega_B(2) = 0$, $\omega_C : A \to C$ given by $\omega_C(1) = 1$ and $\omega_C(2) = 2$, and $\omega_D : A \to D$ given by $\omega_D(1) = \omega_D(2) = 5$.

B is minimal too, but there is no Σ-homomorphism $B \to A$ or $B \to C$. For a and b are interpreted in A and in C as distinct elements of the carrier while in B they are both interpreted as the same element 0. The homomorphism condition (3.10) requires then for any Σ-algebra X and any homomorphism $\phi : B \to X$, that $a^X = \phi(a^B) = \phi(0) = \phi(b^B) = b^X$. I.e., if there is such a homomorphism ϕ then a and b *must be interpreted in* X as the same element of the carrier. Since this isn't the case in A and C, there can be no homomorphism from B to any of these algebras.

In D we have $a^D = b^D = 5$ and so there is a unique homomorphism $\phi : B \to D$ given by $\phi(0) = 5$.

3.2: The Universal Property of Products

The product of a family of algebras also has a relation to the homomorphisms. For the future use, we only register the following universal property of products.

Lemma 3.12 Let ΠA be the product algebra $\prod_{i \in I} A_i$, as given in definition 2.3 together with the projections π_i from (2.2).

1. The projections $\pi_i : \Pi A \to A_i$ are surjective Σ-homomorphisms.
2. Let C be any Σ-algebra with homomorphisms $\phi_i : C \to A_i$ for all i. There is a unique homomorphism $\omega_C : C \to \Pi A$ such that for all $i : \omega_A \circ \pi_i = \phi_i$. (The diagram illustrates it for a binary product.)

$$\begin{tikzcd}
& C \arrow[d, "\omega_C"] \arrow[ddl, "\phi_1"'] \arrow[ddr, "\phi_2"] & \\
& \Pi A \arrow[dl, "\pi_1"] \arrow[dr, "\pi_2"'] & \\
A_1 & & A_2
\end{tikzcd}$$

Proof.

1. First, we verify that the projections π_i from (2.2) are Σ-homomorphisms. For any constant, we have $c^{\Pi A}(i) = \iota \in |\Pi A| : \iota(i) = c^{A_i}$, and hence $\pi_i(c^{\Pi A}) = \iota(i) = c^{A_i}$. Let now $(\iota_1, \dots, \iota_n) \in |\Pi A|^w$. $\pi_i(f^{\Pi A}(\iota_1, \dots, \iota_n)) = \pi_i(\iota)$, where $\iota(i) = f^{A_i}(\iota_i(i), \dots, \iota_n(i))$, i.e., $\pi_i(f^{\Pi A}(\iota_1, \dots, \iota_n)) = f^{A_i}(\pi_i(\iota_1), \dots, \pi_i(\iota_n))$. Since for any $a \in A_i$, there is a $\iota \in \Pi A$ with $\iota(i) = a$, we will have that $\pi_i(\iota) = a$, i.e., the π_i are surjective.

2. Let now $C \in \mathsf{Alg}(\Sigma)$ be another algebra with homomorphisms $\phi_i : C \to A_i$. The unique $\omega_C : C \to \Pi A$ making $\omega_C \circ \pi_i = \phi_i$ for all i is given by $\omega_C(x) = \iota : \iota(i) = \phi_i(x)$. This mapping obviously satisfies the required equations. It is a Σ-homomorphism: for any constant $c : \phi_i(c^C) = c^{A_i}$ and so $\omega_X(c^X) = \iota : \iota(i) = c^{A_i}$, i.e., $\iota = c^{\Pi A}$. Similarly, for any $(x_1,\ldots,x_n) : \omega_C(f^C(x_1,\ldots,x_n)) = \iota$ such that $\iota(i) = \phi_i(f^C(x_1,\ldots,x_n)) = f^{A_i}(\phi_i(x_1),...,\phi_i(x_n)) = f^{A_i}(\iota_i(x_1),...,\iota_i(x_n))$. Since this holds for all $i \in I$ we get $\omega_C(f^C(x_1,\ldots,x_n)) = f^{\Pi A}(\omega_C(x_1),...,\omega_C(x_n))$, i.e., ω_C is a homomorphism.
   Uniqueness of ω_C follows easily from the requirement that for all $i : \omega_C \circ \pi_i = \phi_i$.

Point 2 expresses the so called *universal property* of products which, in a more general categorical setting, functions as *the* definition of a product. One of the main implications of the categorical perspective is that all concepts are defined only up to isomorphism. In fact, if there are two distinct objects P_1 and P_2 which both satisfy this property then they will be isomorphic. In our concrete language this means that, for instance, as a product of A_1 and A_2 we may take $A_1 \times A_2$ with the obvious projections or, equivalently, $A_2 \times A_1$ with the projections $\pi'_1 : A_2 \times A_1 \to A_1$ and $\pi'_2 : A_2 \times A_1 \to A_2$ where π'_1 returns the second, rather than the first, component.

One of the facts related to this universal property, to which we may refer later, is that it is meaningful to talk about empty products. It turns out that such products have a special property which we will later use to define the concept of terminal objects.

Remark 3.13 [Empty products]

Empty products are a special case of products, which is just a product of the empty family of objects. Recasting the universal property 2 from lemma 3.12, we obtain the following requirement. An *empty product* is:

- an object P (with no projections, since there are no objects to which projections would go)
- such that for any other object C (without any further ado, since there are no objects to which we could require homomorphisms from C)
- there is a unique homomorphism $\tau_C : C \to P$ (which does not have to do anything, since there is nothing to commute with).

In short, an empty product is an algebra to which there is a unique homomorphism from any other algebra (in the given class).

3.3: Homomorphisms and Congruences

In the rest of this section we will study a fundamental relation which exists between homomorphisms and congruences on an algebra.

Definition 3.14 *Kernel* of a homomorphism $\phi : A \to B$, is a relation $\equiv_\phi \subseteq |A| \times |A|$ given by $a \equiv_\phi b \Leftrightarrow \phi(a) = \phi(b)$.

The first crucial fact is:

Lemma 3.15 The kernel $\equiv_\phi$ of any Σ-homomorphism $\phi : A \to B$ is a congruence on A.

Proof. Since equality on B is an equivalence, $\equiv_\phi$ is an equivalence on A. To check that it is a congruence, let $a_1 \equiv_\phi b_1, ..., a_n \equiv_\phi b_n$, i.e., $\phi(a_1) = \phi(b_1), ..., \phi(a_n) = \phi(b_n)$ and $f \in \Sigma$. Since ϕ is a homomorphism, we have:

$$\phi(f^A(a_1,\ldots,a_n)) = f^B(\phi(a_1),...,\phi(a_n)) = f^B(\phi(b_1),...,\phi(b_n)) = \phi(f^A(b_1,\ldots,b_n))$$

i.e, $f^A(a_1,\ldots,a_n) \equiv_\phi f^A(b_1,\ldots,b_n)$.

Thus, any homomorphism from A induces a congruence on A. On the other hand, any congruence on A gives rise to a special, so called *natural*, homomorphism.

Lemma 3.16 For any congruence $\equiv$ on A the mapping $\nu : A \to A/_\equiv$ given by $\nu(a) = [a]$ is a surjective homomorphism.

Proof. We have seen in lemma 2.26 that $A/_\equiv$ is a Σ-algebra. For all constants, we have $c^{A/_\equiv} = [c^A] = \nu(c^A)$. For $a_1,\ldots,a_n \in A$ and $f \in \Sigma$ we have

$$\nu(f^A(a_1,\ldots,a_n)) = [f^A(a_1,\ldots,a_n)] \stackrel{2.25}{=} f^{A/_\equiv}([a_1],...,[a_n]) = f^{A/_\equiv}(\nu(a_1),...,\nu(a_n))$$

i.e., ν is a Σ-homomorphism and it is clearly surjective.

Theorem 3.17 [First Homomorphism Theorem] For any surjective $\phi : A \to B$ there is an isomorphism $\psi : A/_{\equiv_\phi} \to B$ such that $\phi = \nu \circ \psi$.

$$\begin{tikzcd} A \arrow[rr, "\phi"] \arrow[dr, "\nu"'] & & B \\ & A/_{\equiv_\phi} \arrow[ur, "\psi"'] & \end{tikzcd}$$

Proof. Define $\psi([a]) \stackrel{\text{def}}{=} \phi(a)$. If $[a] = [b]$ then $\phi(a) = \phi(b)$ and hence $\psi([a])$ is well (i.e., uniquely) defined. ψ is a homomorphism: for all constants we have $c \in \Sigma : \psi(c^{A/_{\equiv_\phi}}) = \psi([c^A]) = \phi(c^A) = c^B$, and for other $f \in \Sigma$:

$$\begin{array}{rcl} \psi(f^{A/_{\equiv_\phi}}([a_1],...,[a_n]) & = & \psi([f^A(a_1,\ldots,a_n)]) \\ & = & \phi(f^A(a_1,\ldots,a_n)) \\ & = & f^B(\phi(a_1),...,\phi(a_n)) \quad = \quad f^B(\psi([a_1]),...,\psi([a_n])) \end{array}$$

We obviously have that $\phi = \nu \circ \psi$. Also, since ϕ is surjective so is ψ. Finally, if $[a] \neq [b]$ then $\phi(a) \neq \phi(b)$, and so $\psi([a]) \neq \psi([b])$, i.e., ψ is injective, and hence an isomorphism.

An important consequence of this theorem is that each homomorphism ϕ has a kind of "normal form": it can be factorized into a surjective homomorphism into the quotient by $\equiv_\phi$ followed by an injection.

Corollary 3.18 For any homomorphism $\phi : A \to B$ there is a surjective $\nu : A \to A/_{\equiv_\phi}$ and an injective $\mu : A/_{\equiv_\phi} \to B$ such that $\phi = \nu \circ \mu$.

$$\begin{tikzcd} A \arrow[rr, "\phi"] \arrow[dr, "\nu"'] & & B \\ & A/_{\equiv_\phi} \arrow[ur, "\mu"'] & \end{tikzcd}$$

Proof. Follows from lemma 3.8 and theorem 3.17.

The following theorem is slightly more technical but almost equally important and useful.

Theorem 3.19 [Second Homomorphism Theorem] Let $\equiv_1 \subseteq \equiv_2$ be two congruences on A, and $\equiv_{1/2}$ be the congruence on $A/_{\equiv_1}$ given by

$$[a]_{\equiv_1} \equiv_{1/2} [b]_{\equiv_1} \Leftrightarrow a \equiv_2 b. \tag{3.11}$$

Then the mapping $\phi : ((A/_{\equiv_1})/_{\equiv_{1/2}}) \to A/_{\equiv_2}$ defined by $\phi([[a]_1]_{1/2}) \stackrel{\text{def}}{=} [a]_2$ is an isomorphism.

$$\begin{tikzcd} & A \arrow[r, "\nu_2"] \arrow[d, "\nu_1"'] & A_2 & = A/_{\equiv_2} \\ A/_{\equiv_1} = & A_1 \arrow[r, "\nu_{1/2}"'] & A_3 \arrow[u, "\phi"'] & = (A/_{\equiv_1})/_{\equiv_{1/2}} \end{tikzcd}$$

PROOF. To simplify the notation, let us use the abbreviations as in the diagram and write $\equiv_3$ for $\equiv_{1/2}$.

First we must show that $\equiv_3$ actually is a congruence on A_1. That it is an equivalence follows trivially from its definition (3.11). So, assume that $[a_i]_1 \equiv_3 [b_i]_1$. Then (3.11) implies that $a_i \equiv_2 b_i$ and, since $\equiv_2$ is a congruence on A, that $f^A(a_1,\ldots,a_n) \equiv_2 f^A(b_1,\ldots,b_n)$. So $[f^A(a_1,\ldots,a_n)]_1 \equiv_3 [f^A(b_1,\ldots,b_n)]_1$, and hence $f^{A_1}([a_1]_1,...,[a_n]_1) \equiv_3 f^{A_1}([b_1]_1,...,[b_n]_1)$, i.e., $\equiv_3$ is a congruence on A_1.

We show that ϕ is a homomorphism. For all constants $c \in \Sigma$, we have $\phi([[c]_1]_3) = [c]_2 = c^{A_2}$. For other symbols $f \in \Sigma$:

$$\begin{array}{rcll}\phi(f^{A_3}([[a_1]_1]_3,...,[[a_n]_1]_3)) & = & \phi([f^{A_1}([a_1]_1,...,[a_n]_1)]_3) & \\ & = & \phi([[f^A(a_1,...,a_n)]_1]_3) & \\ & = & [f^A(a_1,\ldots,a_n)]_2 & \\ & = & f^{A_2}([a_1]_2,...,[a_n]_2) & = \ f^{A_2}(\phi([[a_1]_1]_3),...,\phi([[a_n]_1]_3))\end{array}$$

The first equality follows from the fact that $A_3 = A_1/_{\equiv_3}$, the second one from $A_1 = A/_{\equiv_1}$, the third one from the definition of ϕ, the next one from $A_2 = A/_{\equiv_2}$, and the last one from the definition of ϕ.

Clearly ϕ is surjective. For any $[a]_1, [b]_1 \in A_1$, if $[[a]_1]_3 \neq [[b]_1]_3$, i.e., $[a]_1 \not\equiv_3 [b]_1$, then $a \not\equiv_2 b$ (3.11), and so $\phi([[a]_1]_3) = [a]_2 \neq [b]_2 = \phi([[b]_1]_3)$. So ϕ is injective and thus an isomorphism.

This theorem should be seen in the context of lemma 2.24 which stated that all the congruences on an algebra formed a complete lattice. According to the theorem, this lattice structure is carried over to the system of quotients: whenever $\equiv_1 \subseteq \equiv_2$, we will have a homomorphism $A/_{\equiv_1} \to A/_{\equiv_2}$. Even more interestingly, whenever two congruences $\equiv_1$ and $\equiv_2$ are not related by inclusion, they will have a $lub = \equiv_\cup$ and $glb = \equiv_\cap$. Thus there will be quotients $A/_{\equiv_\cup}$ and $A/_{\equiv_\cap}$ with the unique surjective homomorphisms $\phi_i : A/_{\equiv_\cap} \to A/_{\equiv_i}$ and $\psi_i : A/_{\equiv_i} \to A/_{\equiv_\cup}$.

Exercises (week 3)

EXERCISE 3.1 Define explicitly the mapping $\phi : BN^n \to N_{2^n}$ as described in example 3.4 and verify that it is a homomorphism wrt. to the signature Σ from example 3.3. Define then the operation of substraction modulo n in both algebras, and verify that ϕ is a homomorphism wrt. to this operation as well.

(Hint: Instead of defining substraction explicitly, it will be easier to define it indirectly using the unary minus (inverse) operation. I.e., define first the latter by the properties inverse must satisfy, verify that these properties determine a unique inverse for each element, and then check the homomorphism condition wrt. the substraction.)

EXERCISE 3.2 Consider the Σ_{N} algebras N and $N\lfloor x \rfloor$ from exercise 2.7.

1. Define a homomorphism $\omega : N \to N\lfloor x\rfloor$. Now, either define some other homomorphism $\phi : N \to N\lfloor x\rfloor$ or else give an argument why there is no other ϕ with $\phi \neq \omega$.
2. Define at least three different homomorphisms $N\lfloor x\rfloor \to N$. Verify that they are homomorphisms and indicate their kernels.

EXERCISE 3.3 Let $\Sigma = \langle \underline{\mathsf{B}}; \bot, \top : \to \underline{\mathsf{B}}, \neg : \underline{\mathsf{B}} \to \underline{\mathsf{B}}\rangle$, and consider the following two Σ-algebras:

$B = \langle \mathsf{B}; \mathbf{ff}, \mathbf{tt}, not\rangle$ – the algebra of Booleans with negation, i.e., $\underline{\mathsf{B}}^B \stackrel{\text{def}}{=} \mathsf{B}$, $\bot^B \stackrel{\text{def}}{=} \mathbf{ff}$, $\top^B \stackrel{\text{def}}{=} \mathbf{tt}$ and $\neg^B(x) \stackrel{\text{def}}{=} not(x)$ (in particular, $not(\mathbf{tt}) = \mathbf{ff}$, $not(\mathbf{ff}) = \mathbf{tt}$ and $\mathbf{tt} \neq \mathbf{ff}$), and

$N = \langle \mathsf{N}; 0, 1, +1\rangle$ – the algebra with natural numbers, $\underline{\mathsf{B}}^N \stackrel{\text{def}}{=} \mathsf{N}$, $\top^N \stackrel{\text{def}}{=} 1$, $\bot^N \stackrel{\text{def}}{=} 0$ and $\neg^N(x) \stackrel{\text{def}}{=} x + 1$.

Is there any Σ-homomorphism from B into N? There is exactly one homomorphism $\omega : N \to B$ – define it, verify that it is a homomorphism and determine its kernel $\equiv_\omega$. Show that B and $N/_{\equiv_\omega}$ are isomorphic by defining the isomorphism between them. (This follows immediately from theorem 3.17, but you are here asked to make an explicit construction of this isomorphism.)

EXERCISE 3.4 Show that if $\phi_1 : A \to B$ and $\phi_2 : B \to C$ are two homomorphisms, then so is their composition $\phi_1 \circ \phi_2 : A \to C$.

EXERCISE 3.5 Prove that

1. for any Σ-homomorphism $\phi : A \to B$, ϕ is an isomorphism iff it is surjective and injective homomorphism;
2. if $\phi : A \to B$ is an isomorphism with the inverse $\phi^{-1} : B \to A$, then also ϕ^{-1} is an isomorphism.

EXERCISE 3.6 Given a Σ-algebra A, a set of variables X and an assignment $\alpha : X \to |A|$, suppose that we have the Σ algebra $T_{\Sigma,X}$ (i.e., $\Sigma \cup X$ is non-void.) Define the mapping $\overline{\alpha} : T_{\Sigma,X} \to A$ by $\overline{\alpha}(t) \stackrel{\text{def}}{=} t^A_\alpha$, for all $t \in |T_{\Sigma,X}|$ (definition 1.20). Verify that this mapping is a Σ-homomorphism and that for all $x \in X : \overline{\alpha}(x) = \alpha(x)$.

Week 4: Some, all Σ-algebras, and all Algebras

- Closure Properties of $\mathsf{Alg}(\Sigma)$
- Free Algebras
- Subclasses of $\mathsf{Alg}(\Sigma)$
- The Class of All Algebras

4.1: Closure properties of $\mathsf{Alg}(\Sigma)$

We will now study subclasses of the class $\mathsf{Alg}(\Sigma)$ with respect to different closure properties. The definitions will be therefore given for an arbitrary class $\mathsf{K} \subseteq \mathsf{Alg}(\Sigma)$. We begin however by introducing the definitions of various properties and verifying whether the whole class $\mathsf{Alg}(\Sigma)$ satisfies them.

Definition 4.1 Let K be any class of Σ-algebras. K is *closed under* :

1. *isomorphisms* iff for any $A \in \mathsf{K}$, whenever $B \simeq A$ then $B \in \mathsf{K}$;
2. *quotients* iff for any $A \in \mathsf{K}$ and any $\equiv \in \equiv(A) : A/_{\equiv} \in \mathsf{K}$;
3. *homomorphic images* iff for any $A \in \mathsf{K}$, if $\phi : A \to B$ is a Σ-homomorphism then $\phi[A] \in \mathsf{K}$.
4. *subalgebras* iff whenever $A \in \mathsf{K}$ and $B \sqsubseteq A$ then $B \in \mathsf{K}$;
5. *(finite) products* iff for any (finite) collection $\{A_i\}_{i \in I} \subseteq \mathsf{K} : \prod_i A_i \in \mathsf{K}$;

Lemma 4.2 For any $\mathsf{K} \subseteq \mathsf{Alg}(\Sigma)$: K is closed under homomorphic images iff it is closed under isomorphisms and quotients.

Proof. Directly from the First Homomorphism theorem 3.17. $\Leftarrow$) Any homomorphic image can be obtained by the composition of a surjective mapping into the quotient and an isomorphism. $\Rightarrow$) Any isomorphisms is a homomorphism, and any image of a surjective homomorphism is isomorphic to a quotient.

Lemma 4.3 The class $\mathsf{Alg}(\Sigma)$ satisfies all the closure properties from definition 4.1.

Proof. Follows from earlier lemmata. Since $\mathsf{Alg}(\Sigma)$ contains *all* Σ-algebras, it is obviously closed under isomorphisms. 2 is lemma 2.26 and 3 follows then by lemma 4.2. 4 follows from the definition of subalgebra 2.5, and 5 is the fact 2.4.

4.1.1: Initial/Terminal Objects in $\mathsf{Alg}(\Sigma)$

The concept of initial algebras, although not the central one in the classical algebra, was the basis of the emergence of the field of algebraic specification. We will soon see its relevance – at the moment we only give the general definition and characterize the initial algebras in the whole class $\mathsf{Alg}(\Sigma)$. Terminal algebra is a dual concept of less importance.

Definition 4.4 An object $I \in \mathsf{K}$ is *initial* in K iff for any objetc $A \in \mathsf{K}$ there is a unique homomorphism $\omega_A : I \to A$.
An object $Z \in \mathsf{K}$ is *terminal* in K iff for any objetc $A \in \mathsf{K}$ there is a unique homomorphism $\tau_A : A \to Z$.

Theorem 4.5 For a non-void Σ, the term algebra T_Σ is initial in $\mathsf{Alg}(\Sigma)$.

Proof. It is enough to show that T_Σ is a minimal Σ-algebra and that there is a homomorphism $\omega_A : T_\Sigma \to A$ for any $A \in \mathsf{Alg}(\Sigma)$. Then, by corollary 3.10, each such ω_A will be unique.

That T_Σ is minimal was shown in lemma 2.14. So let A be an arbitrary Σ-algebra. Define $\omega_A : T_\Sigma \to A$ by $\omega_A(t) \stackrel{\text{def}}{=} t^A$ for each $t \in \mathcal{T}(\Sigma)$. Since distinct ground terms are distinct elements of the carrier of T_Σ this yields a well-defined mapping. Since for each $t \in \mathcal{T}(\Sigma)$: $t^{T_\Sigma} = t$, this is obviously a homomorphism.

Example 4.6 [Initial object]

Consider the signature Σ_{N} for natural numbers (example 1.12). There are a lot of different Σ_{N}-algebras, for instance, as observed in remark 1.14, the unit algebra $\{\bullet\}$. Such an algebra is hardly interesting. According to the theorem 4.5, the term algebra with the carrier $\{0, \mathsf{s}0, \mathsf{ss}0, ...\}$ will be initial in $\mathsf{Alg}(\Sigma)$. And, as a matter of fact, when writing the signature Σ_{N} one may expect that this is *the* (carrier of the) intended algebra.

The example illustrates the fact that initial objects often have exactly the properties which correspond to our intuitive understanding of the intended algebras. Other algebras in the model class, although satisfying the same axioms over the signature, may display additional features not intended by the specifier.

Example 4.7 [1.30 continued]

Recall the signature Π of a simple programming language. The carrier of the term algebra T_Π will have all Π-terms, i.e., expressions, tests and programs, as elements. As we observed, the class $\mathsf{Alg}(\Pi)$ will, among other things, contain all algebras which are intended implementations of this signature on concrete machines.

Since we know that T_Π is initial, there will be a unique homomorphism from it to any of these algebras. Such a homomorphism can be viewed as a *semantics* of the language. For instance, $\mathsf{s}(\mathsf{s}0+0)$ and $\mathsf{ss}0+0$ are two distinct expressions – they are different programs and, in T_Π will be different elements. However, one would certainly expect that evaluating these two program expressions should yield the same result. Similarly, we expect that the meaning of '`repeat` 1 `do` S `od`' is to execute the program S once, that is, these two programs should actually mean (do) the same thing. The first and crucial step in defining semantics of a language is to determine which expressions are equivalent – denote equal values. We want to enforce, for instance, the equalities $\mathsf{s}(\mathsf{s}0+0) = \mathsf{ss}0$ and `repeat` 1 `do` S `od` $= S$.

A set of such equalities induces a congruence on the set of terms, and a congruence determines a homomorphism. But also vice versa, any homomorphism will induce a congruence. And so the semantics of Π can be stated by associating T_Π with another algebra and an appropriate homomorphism. For instance, we can take an algebra A with the carrier $A_E = \mathsf{Z}$ and the homomorphism $\rho : T_\Pi \to A$ which will map, for instance, $\rho(\mathsf{s}(\mathsf{s}0+0)) = 2 = \rho(\mathsf{ss}0+0)$ onto the same element $2 \in A_E$.

The homomorphism condition, i.e., $\rho(P(p_1 \ldots p_n)^{T_\Pi}) = P^A(\rho(p_1)...\rho(p_n))$, expresses *compositionality* of the semantics – the meaning of the whole expression is determined by determining the meanings of all the subexpressions $\rho(p_i)$ and combining them by means of the meaning P^A. Thus, designing a programming language, one will define its signature with the associated initial term algebra, and a homomorphism to another algebra which defines the semantics of the language. (Example 4.10 and the subsequent remark 4.11 elaborate this idea and show how we can use homomorphism in compiler construction.)

The following lemma states a general fact about terminal algebras.

Lemma 4.8 For any signature Σ the unit algebra is terminal.

Example 4.9 [4.6 continued]

The terminal algebra for the signature from example 4.6 will collapse all the element to a single element. Terminal algebras play sometimes important role. However, as this and the previous examples illustrate, the initial algebras will often be of more interest.

Notice that lemma 4.8 states the existence of terminal algebras for any signature, wile theorem 4.5 guarantees the existence of initial algebras only for non-void signatures. Thus, for instance, in the class of groupoids (or semigroups, example 3.2) there will be no initial algebra since the signature is void. Unit algebra with one binary (associative) operation will be terminal in the class of groupoids (semigroups).

Later, studying various subclasses of $\mathsf{Alg}(\Sigma)$, we will discover more specific properties characterizing initial and terminal algebras. In such more restricted subclasses, terminal algebras may turn out to be more sophisticated and interesting than the unit algebras.

We now give a more elaborate example illustrating how one can use homomorphisms for defining semantics and compilers of a programming language.

Example 4.10 [Compilers as Homomorphisms]

This example will illustrate the idea of "compilers as homomorphisms". We have to simplify the example, so let us consider merely the language $\Sigma_{\mathsf{N}_+} = \langle \mathsf{N}; 0, \mathsf{s}, +, - \rangle$ of natural numbers with addition and substraction. (The latter is usually defined as *monus*, cf. example 1.3.) A compiler will translate expressions of this language to the expressions in a language L which, we assume, is a machine language for some concrete machine M.

1) Let us first consider a model of this machine. It has a number z of registers $R = \{R1, R2 ... Rz\}$, each of length n, i.e., each capable of storing n bits. In addition, there is a single bit cell, call it Curry – each operation on the registers can take into account the bit stored in C and, in addition to updating the registers, can update the contents of C.

The machine language L will have various operations for fetching and updating the contents of the registers, loading them to the "working" register, etc. We ignore these ones here. In addition, it will have operations for performing computations: given a content of some register(s), one can update various bits. In particular, one will have a set of basic operations for manipulating (pairs, or tuples of) single bits. Here are some examples of a few operations $B \times B \to B$:

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

AND	0	1
0	0	0
1	0	1

Treating 0 as 'false' and 1 as 'true', these are the well-known truth tables. Each such operation can also be applied to a pair of registers – then it will iterate through all the bits (usually, from right to left) applying the basic bit-operation "bitwise". For instance, applying $\mathtt{XOR}(R1, R2)$ to two registers with $R1 = 0010$ and $R2 = 0110$ will yield 0100. (This operation is called "exlusive OR".)

The above operations neither read nor write the curry cell C. Nevertheless, any operation has an implicit additional argument – and result – namely, the contents of C. Thus, defining a new operation, say $\mathtt{ADD} : \mathsf{N} \times \mathsf{N} \to \mathsf{N}$, we are really defining an operation $\mathtt{ADD} : \mathsf{N} \times \mathsf{N} \times B \to \mathsf{N} \times B$ (N is the sort of words, and B the sort of bits). This cell, additional bit B, is used as a flag indicating some special situations, like error, overflow, etc.

2) As the first thing, we have to define translation of the primitive operations of our programming language, i.e., of all the symbols from Π. Given the above set of L-primitives, let us see how we can define bitwise addition. We will use C to keep the curry. Let say that we are processing bits x and y (and C contains the curry resulting from processing the two previous bits). The result which should remain at the position we are processing, should correspond to the binary addition, i.e., it should be 1 if either exactly one or exactly three of the current bits are 1. This corresponds to $\mathtt{XOR}(\mathtt{XOR}(x, y), C)$.

Setting C to 1 should indicate that this last operation gave overflow, i.e., that both bits were 1. We must however remember that, when processing these last two bits, C contained the curry resulting from processing the previous pair of bits. Thus, we will make it 0 iff at most one of the three bits x, y or C is 1. We can write it in a more familiar way: C must become 'true' if at least two of the operands are initially 'true', i.e., $C := (x \wedge y) \vee (x \wedge C) \vee (y \wedge C)$. Distributing in order to minimize the number of operators, we get $C := (x \wedge y) \vee ((x \vee y) \wedge C)$, which in our L will look as $\mathtt{OR}(\ \mathtt{AND}(x, y),\ \mathtt{AND}(\mathtt{OR}(x, y), C)\ )$.

This leads to the following definition of the bitwise operation $\mathtt{ADD} : B \times B \times B \to B \times B$:

$$\mathtt{ADD}(x, y, C) \quad = \quad \langle\ \mathtt{XOR}(\ \mathtt{XOR}(x, y),\ C\ ) \quad ; \quad \mathtt{OR}(\ \mathtt{AND}(x, y),\ \mathtt{AND}(\mathtt{OR}(x, y), C)\ )\ \rangle \tag{4.12}$$

Addition of two numbers, i.e., words $n, m \in \mathsf{N}$ is then defined by $n + m \stackrel{\text{def}}{=} \mathtt{ADD}(n, m, 0)$, and the successor operation $\mathsf{s}(x) \stackrel{\text{def}}{=} \mathtt{ADD}(x, 0^{n-1}1, 0)$. You should verify that, applying this operation to the whole words (from right to left) yields expected results, for instance $\mathtt{ADD}(0110, 0011, 0) = \langle 1001; 0 \rangle$, $\mathtt{ADD}(0110, 1011, 0) = \langle 0001; 1 \rangle$.

3) Let T_L be the term algebra for this language. With the above definitions of $+$ and s, we may view this algebra as containing a subalgebra T_Π. A compiler will be a homomorphism $\gamma : T_\Pi \to T_L$! That is, given the above translations of the primitive Π-operations, any Π-program P will be compiled as a homomorphic image of P: from a program $P(p_1 \dots p_n)$ we will produce a term $\gamma(P(p_1 \dots p_n))$ by translating the innermost pieces $\gamma(p_1), ..., \gamma(p_n)$ and combining them with the translation of the outermost operator into $\gamma(P)(\gamma(p_1)...\gamma(p_n))$.

Of course, the compiler will, in addition, replace T_Π-terms by their T_L-definitions. For instance, given an input program, say $\mathsf{ss}0 + \mathsf{s}0$, it will first map the terms $\mathsf{ss}0$ and $\mathsf{s}0$ onto their binary representations (and, of course, add a lot of code, e.g., for storing them in some registers $R1$ and $R2$). The operation $+$ translates to $\mathtt{ADD}$ and the whole thing will result in an L-program term like $\gamma(\mathsf{ss}0 + \mathsf{s}0) = \mathtt{ADD}(\gamma(\mathsf{ss}0), \gamma(\mathsf{s}0), 0)$. Observe that although $\mathtt{ADD}$ is an operation with three arguments, $\mathtt{ADD}(\_, \_, 0)$ has only two arguments – the 0-bit appearing in the resulting program is an example of how a compiler adds some necessary details. (Of course, we have ignored a lot of similar other details.)

Remark 4.11

The example illustrated only the idea of translating expressions of a higher-level programming language Π into expressions of some machine language L. But, as indicated in example 4.7, any programming language is given with a semantics, i.e., some associated algebra. In this example, we certainly will expect that the legal algebras for the Π-programs should satisfy, for instance, $\mathsf{s}(\mathsf{s}0 + 0) = \mathsf{ss}0$.

1) That is, the starting point of the compiler construction process is the term algebra T_Π *together with* its semantics, e.g., a homomorphism to the algebra N of natural numbers with addition and substraction. (This algebra will ensure, for instance, that two *distinct* programs $\mathsf{s}(\mathsf{s}0 + 0)$ and $\mathsf{ss}0$ have *the same meaning*.) What we have actually implemented here is N_{2^n} – natural numbers modulo 2^n where n is the length of the words on our machine. More precisely, we have the machine-language L with the term algebra T_L which comes equipped with the semantics – the algebra BN^n. The compiler translates the expressions of the respective languages and this translation must be reflected at the semantic level. At the semantic level, we have actually several homomorphisms – in addition to the *sem*antic interpretation of the languages, we have also the homomorphisms $\nu : N \to N_{2^n}$ (example 3.3) and $\phi : BN^n \to N_{2^n}$ (example 3.4). The whole picture looks thus as follows:

$$\begin{tikzcd}
T_\Pi \arrow[rr, "\gamma"] \arrow[d, dotted, "sem"] \arrow[dr, "\rho"] & & T_L \arrow[d, "sem"] \\
N \arrow[r, dotted, "\nu"'] & N_{2^n} & BN^n \arrow[l, "\phi"]
\end{tikzcd}$$

Although we imagine Π to have the semantics N, in practice we have to take into account the fact that any real computer has a limit, *maxint*, which it can store and manipulate. The actual semantics of the language will then be something like $\rho : T_\Pi \to N_{2^n}$.[4] This algebra is represented on the actual machine – here, by BN^n. As we saw in example 3.4 these two are actually isomorphic but, in the general case, we will require the existence of a homomorphism (abstraction function) from the implementing to the implemented algebra which here is $\phi : BN^n \to N_{2^n}$. Then we require that going from our program in Π directly to its meaning along ρ should be the same as: 1) compiling the program along γ into machine code $\in T_L$, 2) evaluating its meaning on the machine M by means of *sem*, and finally 3) abstracting from the concrete representation in BN^n by means of ϕ. That is γ is a correct compiler if, $\rho = \gamma \circ sem \circ \phi$, i.e., if for any Π-program $p : \rho(p) = \phi(sem(\gamma(p)))$.

2) Observe that, although the Π-homomorphism from the initial T_Π algebra is unique, the compiler need not be so. That is, it will be unique when seen as such a homomorphism (e.g., it has to satisfy $\gamma(x + y) = \gamma(x) + \gamma(y)$) but, since the objective is to preserve the semantics, it may also modify the code as long as this semantics is not changed. In particular, the operations from Π may be defined in different ways in L. For instance, the resulting bit in (4.12) may be computed as $\mathtt{XOR}(x, \mathtt{XOR}(y, C))$. In more advanced examples, this kind of differences may lead to compilers of various quality. In this case what stands on the right hand side of the definition $x + y \stackrel{\text{def}}{=} ...$ in L will be different. Nevertheless, seen as a Π-homomorphism, γ still will be unique.

[4] This was actually the way early computers treated overflow. Modern machines will usually produce an overflow error, perhaps, abort the program when encountering such situations.

3) Also, the real compilers may perform some additional optimization. If we take $n = 4$, we obtain here an implementation of natural numbers modulo $maxint = 2^4 = 16$. Seen as an implementation of natural numbers, this will give correct results as long as we do not add numbers, e.g., $9 + 8$, which result in a number greater than $maxint$. Consider the expression $maxint + 2 - 5$. A simple compiler may parse it in the straightforward way producing a code $(\gamma(maxint) + \gamma(2)) - \gamma(5)$ which, almost for sure, gives wrong result. A "bit smarter" compiler may, instead, generate the code $\gamma(maxint) + (\gamma(2) - \gamma(5))$. Whether this is a homomorphic image of the original algebra (not T_Π but N_{2^n}) depends on how the semantics of Π was defined. As a matter of fact, we do not want the expected equality $(x + y) - z = x + (y - z)$ to hold (when $x = maxint$) and the semantics ρ should consider that. This semantics should take into account $maxint$ and provide the rules for how such situations should be treated. This means, the semantics of the programming language should be defined with enough degree of precision to make it possible to construct a compiler as a homomorphism.

4.1.2: Free Algebras

Initial objects are actually only a special case of a more general concept which has been more central in the traditional algebra.

Definition 4.12 Let $\mathsf{K} \subseteq \mathsf{Alg}(\Sigma)$ and X be any set. An algebra F (not necessarily in K) with $|F| \supseteq X$ is *free for* K *on* X iff for each $A \in \mathsf{K}$ and each assignment $\alpha : X \to |A|$ there exists a unique homomorphism $\overline{\alpha} : F \to A$ such that $\alpha = \iota \circ \overline{\alpha}$ where $\iota : X \hookrightarrow F$ is the inclusion mapping, i.e., such that the following diagram commutes:

$$\begin{tikzcd} X \arrow[r, hook, "\iota"] \arrow[dr, "\forall\ \alpha"'] & F \arrow[d, "\exists!\ \overline{\alpha}"] \\ & A \end{tikzcd}$$

If, in addition, $F \in \mathsf{K}$, we say that it is *free in* K on X.

Free algebras play very central role in the study of various subclasses of algebras because by means of the unique homomorphisms they, in a sense, "represent" the whole class. Now, if K has free algebras for each X then it has initial algebra! Simply, take $X = \emptyset$ and the definition 4.12 reduces to the definition of an initial object. Initial algebras "represent" the whole class in the similar manner as the free algebras do with one exception. There is a unique homomorphism from an initial algebra to any other algebra in the class but this homomorphism need not be surjective. If a class has all free algebras (for arbitrary X) then each algebra in the class will actually be (isomorphic to) the image of a free algebra.

Theorem 4.13 If F is free for K on X and, for an $A \in \mathsf{K}$ and each sort $s : |X_s| \geq |A_s|$ then A is isomorphic to a quotient of F.

Proof. Since $|X| \geq |A|$ we have a surjection $\alpha : X \to A$. By the freenes property, we then have a unique homomorphism $\overline{\alpha} : F \to A$ which, agreeing with α on X, will be surjective. So, by the First Homomorphism theorem 3.17 A is isomorphic to a quotient of F.

Another important property of free algebras (at least in the context we are studying them) is that they can be constructed from the given signature. First, we register this fact for the whole class $\mathsf{K} = \mathsf{Alg}(\Sigma)$.

Theorem 4.14 For any set of variables X, $T_{\Sigma,X}$ is free in $\mathsf{Alg}(\Sigma)$ on X.

Proof. Given an assignment $\alpha : X \to A$, it is easy to verify that the interpretation of terms $\mathcal{T}(\Sigma, X)$ in A under α gives a homomorphic extension $\overline{\alpha} : T_{\Sigma,X} \to A$, defined by $\overline{\alpha}(t) \stackrel{\text{def}}{=} t^A_\alpha$ (exercise 3.6). Since, by lemma 2.16, the algebra $T_{\Sigma,X}$ is generated by X, so by lemma 3.9, this extension is unique. Since $T_{\Sigma,X} \in \mathsf{Alg}(\Sigma)$ it is free *in* $\mathsf{Alg}(\Sigma)$.

Theorem 4.5 is now a simple corollary of this more general theorem. The fact that interpretation of terms t^A_α for $t \in \mathcal{T}(\Sigma, X)$ in an algebra A under an assignment α is actually a homomorphisms $\overline{\alpha} : T_{\Sigma,X} \to A$ is a very useful result which we will encounter several times.

Example 4.15 [4.7 continued]
Consider again the signature Π of a simple programming language. We saw that the unique homomorphism from T_Π to any given Π-algebra A might be considered as the semantics of the language. Now, let X be a set of variables. The carrier of the free algebra $T_{\Pi,X}$ will contain all the program expressions with the variables from X. We can think of these as "input variables" to the program. Freeness of $T_{\Pi,X}$ means that, given the semantics A and initial values for all the variables, i.e., an assignment $\alpha : X \to A$, there is a unique way of determining the result (under the semantics A) of any program expression containing these variables.

As a special case of theorem 4.14, we have that any substitution $\alpha : X \to \mathcal{T}(\Sigma, Y)$, which is actually an assignmnet to $T_{\Sigma,Y}$, induces a unique homomorphism between the respective term algebras $\overline{\alpha} : T_{\Sigma,X} \to T_{\Sigma,Y}$. This extension corresponds to the definition 1.17 of application of a substitution. That is, a substitution α determines the terms from $\mathcal{T}(\Sigma, Y)$ to be substituted for the variables X. The homomorphism $\overline{\alpha}$ extends this to the whole terms $\mathcal{T}(\Sigma, X)$ – for any $t \in \mathcal{T}(\Sigma, X)$ we will have $\overline{\alpha}(t) = t\alpha$.

Theorem 4.14 allows us to define a mapping $F : \mathsf{Set} \to \mathsf{Alg}(\Sigma)$ which sends each set X to the free algebra $T_{\Sigma,X}$. It also "transforms" set functions into Σ-homomorphisms between the respective algebras as follows. Let $f : X \to Y$ be a morphism in Set. It can be looked at as an assignment $\phi : X \to T_{\Sigma,Y}$, so by the theorem, it will induce a unique homomorphism $\overline{\phi} : T_{\Sigma,X} \to T_{\Sigma,Y}$.

Corollary 4.16 The mapping $F : \mathsf{Set} \to \mathsf{Alg}(\Sigma)$ defined by $F(X) \stackrel{\text{def}}{=} T_{\Sigma,X}$ and $F(\phi) \stackrel{\text{def}}{=} \overline{\phi}$ preserves compositions, i.e., if $X \xrightarrow{\phi} Y \xrightarrow{\psi} Z$ then $\overline{\phi} \circ \overline{\psi} = \overline{\phi \circ \psi} : T_{\Sigma,X} \to T_{\Sigma,Z}$.

Proof. If $\phi : X \to Y$ and $\psi : Y \to Z$, then for all $x \in X : \overline{\psi}(\overline{\phi}(x)) = \psi(\phi(x))$ and also $\overline{\phi \circ \psi}(x) = \psi(\phi(x))$. By lemma 3.9, we then get that $\overline{\phi} \circ \overline{\psi} = \overline{\phi \circ \psi}$.

$$\mathsf{Set} \xrightarrow{F} \mathsf{Alg}(\Sigma)$$

$$\begin{tikzcd} X \arrow[r, mapsto, "F"] \arrow[d, "\phi"] \arrow[dd, bend right=60, "\phi\circ\psi"'] & T_{\Sigma,X} \arrow[d, "\overline{\phi}"] \arrow[dd, bend left=60, "\overline{\phi\circ\psi}"] \\ Y \arrow[r, mapsto, "F"] \arrow[d, "\psi"] & T_{\Sigma,Y} \arrow[d, "\overline{\psi}"] \\ Z \arrow[r, mapsto, "F"] & T_{\Sigma,Z} \end{tikzcd}$$

This corollary adds to the theorem 4.14 the possibility of extending the functions between sets to the homomorphisms between the corresponding free algebras.

4.2: Subclasses of $\mathsf{Alg}(\Sigma)$

So far we have considered a very simple and general situation: the class of algebras was the whole class $\mathsf{Alg}(\Sigma)$. This class is in a sense fundamental because all other classes will be contained in it. What is of interest, however, and what we have seen several examples of, is the way of choosing a more specific class of algebras which have some desired properties. This is done by means of axiomatization expressed in some formal language. We will study this in the following sections. In this section, we no longer consider the whole class $\mathsf{Alg}(\Sigma)$ but study some semantic properties which various subclasses K of $\mathsf{Alg}(\Sigma)$ may or may not have.

The main message of this section will be that not all subclasses of $\mathsf{Alg}(\Sigma)$ have similarly strong properties, given in lemma 4.3, as the whole class itself. A simple example of the semantic properties we have in mind was given in lemma 4.2: any class is closed under homomorphic images iff it is closed under isomorphisms and quotients. However, not all classes K will have initial objects or free algebras, not all will be closed under products, etc. Here we will consider only characterization of initial/terminal algebras and the existence of free algebras in arbitrary subclasses of $\mathsf{Alg}(\Sigma)$. In later sections, we will study the closure properties in relation to the form of the axioms defining a subclass of $\mathsf{Alg}(\Sigma)$.

4.2.1: Initial/Terminal Objects in a Class of Algebras

The example 4.9 showed the importatant difference between initial and terminal objects which happens to apply to the general case of an arbitrary class of algebras.

Lemma 4.17 Let $\mathsf{K} \subseteq \mathsf{Alg}(\Sigma)$, and $I, Z \in \mathsf{K}$ be generated. Then

1. If I is initial in K then for all $s, t \in \mathcal{T}(\Sigma)$: $I \models s = t \;\Leftrightarrow\; \forall B \in \mathsf{K} : B \models s = t$
2. If Z is terminal in K then for all $s, t \in \mathcal{T}(\Sigma)$: $Z \models s = t \;\Leftrightarrow\; \exists B \in \mathsf{K} : B \models s = t$

Proof. 1. $\Leftarrow$) is obvious. For 1. $\Rightarrow$) let $\omega_B : I \to B$ be the unique homomorphism and assume $I \models s = t$ for some $s, t \in \mathcal{T}(\Sigma)$. Then $s^B = \omega_B(s^I) = \omega_B(t^I) = t^B$, i.e., $B \models s = t$.

2. $\Leftarrow$) Let $\tau_B : B \to Z$ be the unique homomorphism. By the same argument as above, we obtain that $s^B = t^B$ implies $s^Z = t^Z$. Since there is a unique τ_B into Z from each B, the claim follows. 2. $\Rightarrow$) is obvious, since $Z \in \mathsf{K}$. So if for each $B \in \mathsf{K} : B \not\models s = t$, then, in particular, $Z \not\models s = t$.

Remark 4.18

It may seem surprising that the notion of homomorphism which is the basis of the definition of intial/terminal algebras has such a tight relationship with the satisfaction of equations as stated in this lemma. The proof should clarify this relationship. As a rough informal intuition, one should bear in mind that homomorphisms induce and are induced by congruences on the algebras. Identity relation, which determines satisfaction of equations in an algebra, is the basic congruence – on this algebra. Since T_Σ is an initial algebra in $\mathsf{Alg}(\Sigma)$, there is a unique homomorphism ν_A from it to any other Σ-algebra A, in particular to any $A \in \mathsf{K}$ (even if $T_\Sigma \notin \mathsf{K}$). This homomorphism (which, by exercise 3.6, corresponds to the evaluation of ground terms in A) induces a congruence on the carrier of T_Σ, i.e., the set of ground terms $\mathcal{T}(\Sigma)$. The first point of the lemma says that such a congruence induced by $\nu_I : T_\Sigma \to I$ will be the least among all the congruences on $\mathcal{T}(\Sigma)$ induced by the unique homomorphisms $\nu_A : T_\Sigma \to A \in \mathsf{K}$

$$\begin{tikzcd} & T_\Sigma \arrow[dl, "\nu_I"'] \arrow[dr, "\nu_A"] & \\ I \arrow[rr, "\omega_A"] & & A \end{tikzcd}$$

The homomorphism ν_I will, typically, identify some elements and so will ω_A. Uniqueness of ν_A implies that $\nu_A = \nu_I \circ \omega_A$. Since there is a ω_A from I to any other $A \in \mathsf{K}$, the initial algebra I cannot identify interpretation of some ground terms which are not identified in some $A \in \mathsf{K}$. (The situation with the terminal algebra Z is analogous, but dual.)

Notice that the assumption that I, Z are generated is essential here. It means that ν_I (resp. ν_Z) is surjective. A non-generated algebra may satisfy the same ground equations as all (or some) other algebras from K without being initial (terminal).

Thus, initial algebra will distinguish as many elements of its generated carrier as possible, while a terminal one will identify them. This is the slogan that initial algebras contain "no confusion" – things are not identified unless this is explicitly required.

This was the case – in a rather simple setting – in example 4.9. The importance of lemma 4.17 consists in the fact that it characterizes the respective algebras in arbitrary class and not only in the whole $\mathsf{Alg}(\Sigma)$. Choosing such a class by means of appropriate axioms, we may obtain terminal algebras which are quite relevant.

Example 4.19 [Initial vs. Terminal Algebras]

Assume that we have given sorts N and B of usual natural numbers and Booleans. Consider the following axiomatization:

$$\begin{array}{rlrcl}
\text{SB} = \mathcal{S}: & S, \mathsf{N}, \mathsf{B} & & & \\
\Omega: & emp: & & \to S & \\
& ins: & \mathsf{N} \times S & \to S & \\
& isin: & \mathsf{N} \times S & \to \mathsf{B} & \\
\mathcal{A}: & ins(x, ins(y, Z)) & = & ins(y, ins(x, Z)) & \\
& isin(x, emp) & = & \bot & \\
& isin(x, ins(y, Z)) & = & or(eq(x, y), isin(x, Z)) &
\end{array}$$

Sort S represents collections of N. Operation ins inserts a natural number into a collection, and first axioms states that the order of such insertions is irrelevant. Operation $isin$ detemines whether its first argument occurs in the collection provided as the second argument.

In an inital model of SB, the sort S will contain *bags* of natural numbers, i.e., collections where the number of occurrences of an element matters. Two collections will be different iff at least one number has been inserted a different number of times (as many elements are distinct as possible). Thus, it will satisfy, for instance, the formula $ins(1, ins(1, emp)) \neq ins(1, emp)$.

In a terminal algebra two collections distinct only by the positive number of occurrences of various elements will be identified, since their difference does not follow from the axioms of SB. It will satisfy the formula $ins(x, ins(x, Z)) = ins(x, Z)$. Thus it will yield not bags but *sets* of natural numbers: two sets are different iff one contains an element not contained in the other. In fact, since the terminal algebra tends to identify everything possible, the first axiom of SB will be redundant – the terminal models will be isomorphic whether we include it or not.

As can be seen from the example and lemma 4.17, terminal semantics will collapse everything to a single point unless we have a means of preventing this. In the example we achieved this by requiring the sorts N and B to be included with their expected meaning. However, if the specification Nat from example 4.6 were given terminal semantics, we would obtain a trivial unit algebra, since there is nothing in the axioms forcing any two things to be different.

Since intial objects are special cases of the free algebras, we do not study the existence of the former but turn to the latter. Sufficient conditions for the existence of free algebras will also be sufficient for the existenco of the initial ones (but not vice versa!)

4.2.2: Existence of Free Algebras

Lemma 4.2 was an example of a semantic property of some class K of algebras. We will now prove another property of this kind (which is one of the central theorems from the classical algbera), namely, we will demonstrate the sufficient conditions (some closure properties) for an arbitrary non-empty class to have free algebras. In fact, we will show more – the proof will actually demonstrate how one can construct a free algebra for a class satisfying these closure properties.

Definition 4.20 Let K be a non-empty subclass of $\mathsf{Alg}(\Sigma)$ and X be a set of variables. By $\equiv_{\mathsf{K}}$ we denote the following relation on $\mathcal{T}(\Sigma, X)$

$$\equiv_{\mathsf{K}} \stackrel{\text{def}}{=} \bigcap_{A \in \mathsf{K}} \{\equiv_\phi\text{: where } \phi : T_{\Sigma,X} \to A \text{ is a homomorphism}\}$$

By $T_{\mathsf{K},X}$ we denote the quotient algebra $T_{\Sigma,X}/_{\equiv_{\mathsf{K}}}$.

Since each $\equiv_{\phi_i}$ is a congruence (lemma 3.15), so is $\equiv_{\mathsf{K}}$ (lemma 2.24) and thus $T_{\mathsf{K},X}$ is well defined.

Theorem 4.21 Assume that we have a Σ with term algebra $T_{\Sigma,X}$, and that $\mathsf{K} \subseteq \mathsf{Alg}(\Sigma)$ is non-empty. Then $T_{\mathsf{K},X}$ is free *for* K over $X/_{\equiv_{\mathsf{K}}}$.

Proof. Let $\alpha : X/_{\equiv_{\mathsf{K}}} \to A$ be an assignment to an arbitrary $A \in \mathsf{K}$. We have to show the existence of a unique homomorphic extension $\overline{\alpha} : T_{\mathsf{K},X} \to A$.

$$\begin{tikzcd} X \arrow[d, "k"'] \arrow[ddr, "\beta"] & & T_{\Sigma,X} \arrow[d, "\overline{k}"] \arrow[ddl, "\overline{\beta}"'] \\ X/_{\equiv_{\mathsf K}} \arrow[dr, "\alpha"'] & & T_{\mathsf{K},X} \arrow[dl, "\overline{\alpha}"] \\ & A & \end{tikzcd}$$

Let $k : X \to X/_{\equiv_{\mathsf K}}$ be the natural function mapping $x \mapsto [x]$, and $\overline{k} : T_{\Sigma,X} \to T_{\mathsf{K},X}$ its unique homomorphic extension (existing by theorem 4.14). Notice that neither $T_{\Sigma,X}$ nor $T_{\mathsf{K},X}$ need to belong to K. α induces then the assignment $\beta = k \circ \alpha : X \to A$ which, by theorem 4.14, gives a unique homomorphism $\overline{\beta} : T_{\Sigma,X} \to A$. Define $\overline{\alpha} : T_{\mathsf{K},X} \to A$ by $\overline{\alpha}([t]) \stackrel{\text{def}}{=} \overline{\beta}(t)$ for all $[t] \in T_{\mathsf{K},X}$. This is a well-defined mapping, for if $t_1, t_2 \in [t]$ then $t_1 \equiv_{\mathsf K} t_2$, i.e., for each $A \in \mathsf{K}$ and assignment $\beta : X \to A$ we have $(t_1)^A_\beta = (t_2)^A_\beta$, and so $\overline{\beta}(t_1) = \overline{\beta}(t_2)$.

$\overline{\alpha}$ is a Σ-homomorphism: for all constants c we have $\overline{\alpha}([c]) = \overline{\beta}(c) = c^A$, and for other function symbols f : $\overline{\alpha}(f^{T_{\mathsf{K},X}}([t_1], ..., [t_n])) = \overline{\alpha}([f^{T_{\Sigma,X}}(t_1, ..., t_n)]) = \overline{\beta}(f^{T_{\Sigma,X}}(t_1, ..., t_n)) = f^A(\overline{\beta}(t_1), ..., \overline{\beta}(t_n)) = f^A(\overline{\alpha}([t_1]), ..., \overline{\alpha}([t_n]))$.

Since $T_{\Sigma,X}$ exists, either Σ is non-void or X, and hence $X/_{\equiv_{\mathsf K}}$, is non-empty. $T_{\Sigma,X}$ is generated by X (lemma 2.16), so $T_{\mathsf{K},X} = T_{\Sigma,X}/_{\equiv_{\mathsf K}}$ is generated by $X/_{\equiv_{\mathsf K}}$ (lemma 2.33). So, by lemma 3.9, any homomorphism which agrees with $\overline{\alpha}$ on $X/_{\equiv_{\mathsf K}}$ is the same as $\overline{\alpha}$, i.e., $\overline{\alpha}$ is unique.

A and α were arbitrary so $T_{\mathsf{K},X}$ is free for K over $X/_{\equiv_{\mathsf K}}$.

We know that, for instance $T_{\Sigma,X}$, is free for any $\mathsf{K} \subseteq \mathsf{Alg}(\Sigma)$, since it is free for $\mathsf{Alg}(\Sigma)$. The importance of this theorem consists in constructing a more specific algebra which also is free for K. The central result, is the following theorem which gives sufficient conditions for the algebra $T_{\mathsf{K},X}$ to belong to K.

Theorem 4.22 [Birkhoff 1935] Assume that we have a Σ with term algebra $T_{\Sigma,X}$, and that a non-empty $\mathsf{K} \subseteq \mathsf{Alg}(\Sigma)$ is closed under isomorphisms, subalgebras and non-empty products. Then $T_{\mathsf{K},X}$ is free *in* K over $X/_{\equiv_{\mathsf K}}$.

Proof. To simplify the notation, let C be the set of all the congruences on $T_{\Sigma,X}$ from definition 4.20 and J an arbitrary indexing set for C, i.e., $C = \{\equiv_j\}_{j \in J} = \{\equiv_\phi\}$ where ϕ is some homomorphism from $T_{\Sigma,X}$ into some $A \in \mathsf{K}$, and $\equiv_{\mathsf K} = \bigcap_{j \in J} \equiv_j$.

1. By corollary 3.18, for each j, there is an $A \in \mathsf{K}$ and a subalgebra $B \sqsubseteq A$ such that $T_{\Sigma,X}/_{\equiv_j} \simeq B$. Since K is closed under subalgebras and isomorphisms, this means that $T_j = T_{\Sigma,X}/_{\equiv_j} \in \mathsf{K}$.

2. Since K is closed under non-empty products, $\Pi T = \prod_{j \in J} T_j \in \mathsf{K}$. $\equiv_{\mathsf K} \subseteq \equiv_j$ for each j, so by theorem 3.19 there is a homomorphism $\psi_j : T_{\mathsf{K},X} \to T_j$ for each j. By the categorical product property (lemma 3.12), there is a (unique) $\psi : T_{\mathsf{K},X} \to \Pi T$ making $\psi \circ \pi_j = \psi_j$ for each j.

3. In fact, ψ is injective because $\equiv_{\mathsf K} = \bigcap_{j \in J} \equiv_j$: if $[t]_{\equiv_{\mathsf K}} \neq [s]_{\equiv_{\mathsf K}}$ then there must be a j such that $\psi_j([t]_{\equiv_{\mathsf K}}) \neq \psi_j([s]_{\equiv_{\mathsf K}})$. But then $\pi_j(\psi([t]_{\equiv_{\mathsf K}})) = \psi_j([t]_{\equiv_{\mathsf K}}) \neq \psi_j([s]_{\equiv_{\mathsf K}}) = \pi_j(\psi([s]_{\equiv_{\mathsf K}}))$, so we must have that $\psi([t]_{\equiv_{\mathsf K}}) \neq \psi([s]_{\equiv_{\mathsf K}})$. This means that there is a subalgebra $\psi[T_{\mathsf{K},X}] \sqsubseteq \Pi T$ which is isomorphic to $T_{\mathsf{K},X}$. Since K is closed under subalgebras and isomorphisms, we obtain that $T_{\mathsf{K},X} \in \mathsf{K}$.

Corollary 4.23 Let Σ be non-void, and K be any subclass of $\mathsf{Alg}(\Sigma)$ closed under isomorphisms, subalgebras and non-empty products. Then $T_{\mathsf K}$ is initial in K.

We will later encounter important classes of algebras which have some stronger properties than those in the theorem 4.22. We therefor register the following consequence of this theorem and lemma 4.2.

Corollary 4.24 Any class $\mathsf{K} \subseteq \mathsf{Alg}(\Sigma)$ closed under subalgebras, all products and homomorphic images has free algebras. If Σ is non-void, then K has initial algebras.

4.3: The Class Alg

We have now seen the class $\mathsf{Alg}(\Sigma)$ and some of its subclasses characterized by some closure properties. In the study of abstract data types we will have to relate also algebras over different signatures. For this purpose, we will have to consider the class Alg which is a collection of all classes $\mathsf{Alg}(\Sigma)$ for all different signatures Σ. The crucial constructions in this large class are obtained by relating the different signatures.

Definition 4.25 Let Σ, Σ' be two signatures. A *signature morphism* $\sigma : \Sigma' \to \Sigma$ is a pair of mappings $\sigma_{\mathcal S} : \mathcal{S}' \to \mathcal{S}$ and $\sigma_\Omega : \Omega' \to \Omega$ such that if $f' \in \Sigma'_{w,s}$ then $\sigma_\Omega(f') \in \Sigma_{\sigma_{\mathcal S}(w),\sigma_{\mathcal S}(s)}$.

$$\begin{tikzcd} f' : \arrow[d, "\sigma_\Omega"'] & S'_1 \arrow[d, "\sigma_{\mathcal S}"'] \times \ldots \times S'_n \arrow[d, "\sigma_{\mathcal S}"'] \arrow[r] & S' \arrow[d, "\sigma_{\mathcal S}"] \\ f : & S_1 \times \ldots \times S_n \arrow[r] & S \end{tikzcd}$$

Example 4.26 [Signature morphism]
Take the following signatures:

$\Sigma_N =$	$\Sigma_{\mathsf N} =$	$\Sigma_+ =$	$\Sigma_{\mathsf Z} =$
$\mathcal{S}$: Nat	$\mathcal{S}$: N	$\mathcal{S}$: N	$\mathcal{S}$: Z
Ω: zer: $\to \mathsf{N}$	Ω: 0: $\to \mathsf{N}$	Ω: 0: $\to \mathsf{N}$	Ω: 0: $\to \mathsf{Z}$
suc: $\mathsf{N} \to \mathsf{N}$	s: $\mathsf{N} \to \mathsf{N}$	s: $\mathsf{N} \to \mathsf{N}$	s: $\mathsf{Z} \to \mathsf{Z}$
		$+$: $\mathsf{N} \times \mathsf{N} \to \mathsf{N}$	$+$: $\mathsf{Z} \times \mathsf{Z} \to \mathsf{Z}$
			$*$: $\mathsf{Z} \times \mathsf{Z} \to \mathsf{Z}$

The following are examples of possible signature morphisms:

$\sigma_1 : \Sigma_N \mapsto \Sigma_{\mathsf N}$	$\sigma_2 : \Sigma_N \mapsto \Sigma_+$	$\sigma_3 : \Sigma_+ \mapsto \Sigma_{\mathsf Z}$	$\sigma_4 : \Sigma_{\mathsf Z} \mapsto \Sigma_+$
$Nat \mapsto \mathsf{N}$	$Nat \mapsto \mathsf{N}$	$\mathsf{N} \mapsto \mathsf{Z}$	$\mathsf{Z} \mapsto \mathsf{N}$
$zer \mapsto 0$	$zer \mapsto 0$	$0 \mapsto 0$	$0 \mapsto 0$
$suc \mapsto \mathsf{s}$	$suc \mapsto \mathsf{s}$	$\mathsf{s} \mapsto \mathsf{s}$	$\mathsf{s} \mapsto \mathsf{s}$
		$+ \mapsto *$	$+ \mapsto +$
			$* \mapsto +$

σ_1 is a bijective morphism – it merely causes renaming which can be reversed. σ_2 is injective – it renames the symbols from Σ_N including them in the larger signature Σ_+. Notice that there can be no signature morphism from Σ_+ into Σ_N or $\Sigma_{\mathsf N}$ since the latter lack any operation which could possibly be the image of $+$.

σ_3 is similar to σ_2 in that it includes the subsignature Σ_+ into $\Sigma_{\mathsf Z}$. σ_4 is an opposite mapping which identifies the two operation symbols $+$ and $*$ – this is possible, since their profiles, after mapping $\mathsf{Z} \mapsto \mathsf{N}$ coincide.

Example 4.27
Recall example 1.8. We have a signature Σ_P with three sorts A, B and P, where P is ment to be a

product sort $[A \times B]$. We take also another signature Σ_C with sorts C and S where S is ment to be the binary product sort $[C \times C]$. We thus have the following signature:

$$\begin{array}{llll} \Sigma_P = & & & \\ \mathcal{S}: & A,\ B,\ P & & \\ \Omega: & \times: & A \times B \to P \\ & p_A: & P \to A \\ & p_B: & P \to B \end{array} \qquad \begin{array}{llll} \Sigma_C = & & & \\ \mathcal{S}: & C,\ S & & \\ \Omega: & \times: & C \times C \to S \\ & p_1: & S \to C \\ & p_2: & S \to C \end{array}$$

There is no signature morphism $\Sigma_C \to \Sigma_P$. But we may define a signature morphism $\sigma : \Sigma_P \to \Sigma_C$ sending $\sigma(A) = \sigma(B) = C$ and $\sigma(p_A) = p_1$ and $\sigma(p_B) = p_2$.

It may seem that signature morphisms allow one to define quite arbitrary mappings. However, such mappings are (or can be) reflected in the corresponding classes of algebras. In the example 4.27 we mapped a "more complicated" signature of arbitrary binary product sort to "simpler" signature of a 2nd-power sort. It shouldn't be surprising that any algebra over Σ_C can be seen as an algebra over Σ_P after we have renamed the operations appropriately. On the other hand, an arbitrary Σ_P-algebra may not correspond to any Σ_C algebra – the latter has only a sort which is a product of one single sort C, while the former may have a product sort of two different sorts A and B.

Let $\sigma : \Sigma' \to \Sigma$ be a signature morphism and $A \in \mathsf{Alg}(\Sigma)$. The important relation between the former and the latter is based on the fact that these two pieces of information allow us to "recover" a Σ'-algebra A' "burried within" A.

Definition 4.28 Let $\sigma : \Sigma' \to \Sigma$ be a signature morphism and $A \in \mathsf{Alg}(\Sigma)$. The *σ-reduct* of A, $A|_\sigma \in \mathsf{Alg}(\Sigma')$ is defined as follows:

- for each $s \in \mathcal{S}' : (A|_\sigma)_s \stackrel{\text{def}}{=} A_{\sigma(s)}$;
- for each $f \in \Sigma'_{w,s}$ define $f^{A|_\sigma} \stackrel{\text{def}}{=} \sigma(f)^A$.

Observe the important point that σ-reduct is contravariant wrt. σ, i.e., while σ is a morphism from Σ' into Σ, the σ-reduct goes in the opposite direction: it takes a Σ-algebra and delivers a Σ'-algebra.

Working within one class $\mathsf{Alg}(\Sigma)$ for some specific signature Σ, we related various Σ-algebras by means of Σ-homomorphisms. The reduct construction plays a corresponding role in relating the algebras of different signatures. It can be seen a bit more abstractly as relating *classes* of algebras. In a class $\mathsf{Alg}(\Sigma)$ the objects are Σ-algebras and a Σ-homomorphism brings us from one such algebra to another $\phi : A \to B$. In the class Alg the objects are *classes* of algebras (namely, classes $\mathsf{Alg}(\Sigma)$ for all possible Σ) and a σ-reduct (for some $\sigma : \Sigma' \to \Sigma$) brings us from one such class to another, $|_\sigma : \mathsf{Alg}(\Sigma) \to \mathsf{Alg}(\Sigma)'$, by taking each Σ-algebra and converting it into a Σ'-algebra.

The reduct construction achieves a range of different effects depending on the kind of signature morphism.

Example 4.29 [4.26 continued]

<u>σ is bijective – renaming:</u> σ_1 was a bijective morphism – thus for any Σ_{N}-algebra N, its reduct $N|_{\sigma_1}$ will be essentially the same algebra with renamed operations. The interpretation of the operation $suc^{N|_{\sigma_1}}$ will be s^N, etc.

<u>σ is injective/inclusion – forgetting:</u> σ_2 was injective (but not surjective). σ_2-reduct of any Σ_+-algebra N will be esentially the same algebra but without the $+$ operation which is "forgotten" – i.e., on the algebra $N|_{\sigma_2}$, we can perform operations zer and suc, but not $+$. Very similar effect of just forgetting some operations would be achieved if we tok a straight inclusion, for instance, of $\Sigma_{\mathsf{N}} \hookrightarrow \Sigma_+$. When the signature morphism is an inclusion $\sigma : \Sigma' \hookrightarrow \Sigma$, we also write $A|_{\Sigma'}$ instead of $A|_\sigma$.

Let Z be the usual Σ_{Z}-algebra of integers with addition and multiplication. $Z|_{\sigma_3}$ will be a Σ_+ algebra with the carrier Z the set of all integers, 0 and s operations interpreted in the same way as in Z and with $+$ operation interpreted as multiplication $(*)$ on integers. Here, in $Z|_{\sigma_3}$ we have forgotten the Σ_{Z} operation $*$ and (re)interpreted the Σ_+ operation $+$ as the operation $*$ on Z.

<u>σ is non-injective – separating:</u> Finally, let N be the Σ_+-algebra of natural numbers with addition. $N|_{\sigma_4}$ will be the Σ_{Z}-algebra with the Z-carrier the natural numbers, where both operations $+$ and $*$ will be the usual addition on natural numbers. That is, the algebra $N|_{\sigma_4}$ will have two Σ_{Z} operations $+$ and $*$, but they will be both interepreted as addition on natural numbers. Thus, the same semantic entity $+^N$ gets separated in $N|_{\sigma_4}$ as two operations $+^{N|_{\sigma_4}}$ and $*^{N|_{\sigma_4}}$.

Example 4.30 [4.27 continued]

Another example of "separation" follows from the morphism in example 4.27. Here we do not separate an operation but sorts. Let N be a Σ_C algebra where N_C is the set of natural numbers, N_S is the cartesian product $N_C \times N_C$, the operation $\times$ returns the pair and p_1, p_2 are the first and second projections. Taking the σ-reduct of N will amount to "separating" the sorts corresponding to the first and second component. $N|_\sigma$ will have three sorts: $(N|_\sigma)_A = N_C$, $(N|_\sigma)_B = N_C$, and $(N|_\sigma)_P = N_S$. The operations p_A and p_B will be just the first and second projections, i.e., $p_A^{N|_\sigma} = p_1^N$ and $p_B^{N|_\sigma} = p_2^N$.

Example 4.31 [4.10 continued]

Recall the construction of a compiler for the language Π to the machine-language L. In the example 4.10 we were actually cheating a bit claiming that the compiler was a Π-homomorphism. We compiled, for instance, the addition $x + y$ to the operation $\mathtt{ADD}(x, y, 0)$. What we really did was to design a signature morphism $\gamma : \Pi \to \Sigma_L$ which mapped, for instance, $+ \mapsto \mathtt{ADD}(\_,\_,0)$ and $\mathsf{s} \mapsto \mathtt{ADD}(\_, 0^{n-1}1, 0)$. These latter expressions can be seen as "derived operations" on the signature Σ_L, i.e., as a two-argument addition $Add(x,y) \stackrel{\text{def}}{=} \mathtt{ADD}(x,y,0)$, and one-argument successor $s(x) \stackrel{\text{def}}{=} \mathtt{ADD}(x, 0^{n-1}1, 0)$.

We also said that the compiler had to preserve the semantics of the language Π and expressed it, in remark 4.11, as the requirement that, for any Π-program expression $p : \rho(p)) = \phi(sem(\gamma(p)))$, where ρ was the semantics of Π and $\phi : BN^n \to N_{2^n}$ was the abstraction function mapping the binary representations back to the natural numbers (modulo n). In fact, this last homomorphism ϕ is the reduct corresponding to the signature morphism γ – it takes an L-algebra implementing the machine language L and recovers the respective Π-algebra by, among other things, mapping the binary words onto respective numbers and forgetting the low-level operations like $\mathtt{XOR}$ and $\mathtt{ADD}$.

<u>Exercises</u> (week 4)

<u>EXERCISE 4.1</u> Let $\Sigma = \langle D; \emptyset \rangle$ – have only one sort symbol D and no operation symbols. Is the class $\mathsf{Alg}(\Sigma)$ closed under products? What does the homomorphism condition (3.10) reduce to with this simple signature? Is $\mathsf{Alg}(\Sigma)$ closed under homomorphic images?

<u>EXERCISE 4.2</u> Let Set^4 be the collection of sets with at least 1 and at most 3 elements, considered as algebras over Σ from the previous exercise. Is this class closed under products? Under subalgebras? Under homomorphic images?

Let Set^3 be the class of sets with exactly three elements, considered as Σ-algebras. Is this class closed under products? Under subalgebras? Under homomorphic images?

(You may consult exercise 2.1 but be aware that the signature Σ_3 used there was different from Σ used here.)

<u>EXERCISE 4.3</u> In example 1.24 and 3.2 we saw the classes of monoids (over signature Σ_{M}) and groups (over signature Σ_{G}). Remember that both where not just the classes of Σ_{M}-, resp. Σ_{G}-algebras, but that they had to satisfy the respective axioms (example 1.24).

1. What will be an initial algebra in the class $\mathsf{Alg}(\Sigma_{\mathsf{M}})$? And in $\mathsf{Alg}(\Sigma_{\mathsf{G}})$? What will be terminal algebras in these two classes?

2. What will be an initial algebra in the class of monoids? And in the class of groups? What will be the respective terminal algebras?

3. Consider term algebras $T_{\Sigma_{\mathsf{M}},\{x\}}$, and $T_{\Sigma_{\mathsf{G}},\{x\}}$. Let $\equiv_{\mathsf{M}}$, resp. $\equiv_{\mathsf{G}}$ be the congruences induced on these term algebras (by the respective equations from example 1.24) according to the definition 2.29 (example 2.30). What will be the carrier of the quotient monoid $T_{\Sigma_{\mathsf{M}},\{x\}}/_{\equiv_{\mathsf{M}}}$? And of the quotient group $T_{\Sigma_{\mathsf{G}},\{x\}}/_{\equiv_{\mathsf{G}}}$?

(You may use here some well-known facts about monoids and groups: 1. if for all $x : x \circ 0 = 0 \circ x = x \circ a = a \circ x = x$, then $a = 0$, (i.e., the neutral element is unique), 2. if $a \circ x = x \circ a = 0$ and $b \circ x = x \circ b = 0$, then $a = b$ (i.e., the inverse is a function – for any x there is a unique inverse element x^-), 3. $(x^-)^- = x$ and 4. $(x \circ y)^- = y^- \circ x^-$.)

EXERCISE 4.4 Consider the class of algebras K determined by the following signature and satisfying the axioms E:

$$\begin{array}{rrrcl} \Sigma = & \mathcal{S}: & \mathsf{N} & & \\ & \Omega: & 0: & & \to \mathsf{N} \\ & & \mathsf{s}: & \mathsf{N} & \to \mathsf{N} \\ & & +: & \mathsf{N} \times \mathsf{N} & \to \mathsf{N} \\ & E: & x + 0 & = & x \\ & & x + \mathsf{s}y & = & \mathsf{s}(x + y) \end{array}$$

Show that

1. if $A \in \mathsf{K}$ and $B \sqsubseteq A$ then also $B \in \mathsf{K}$ (i.e., any subalgebra of an algebra satisfying E will satisfy E);
2. if $A, B \in \mathsf{K}$ then $A \times B \in \mathsf{K}$ (i.e., the product $A \times B$ of two algebras satisfying the axioms E will also satisfy E);
3. if $A \in \mathsf{K}$ and $B \simeq A$ then also $B \in \mathsf{K}$.

We do not prove it here but, in fact, the construction showing 2. can be generalized to show that K is closed under all non-empty products. Thus the conditions of theorem 4.22 are satisfied, and so K will have free algebras and, since Σ is non-void, an initial algebra $T_{\mathsf{K}} = T_\Sigma/_{\equiv_{\mathsf{K}}} \in \mathsf{K}$.

According to definition 4.20, $\equiv_{\mathsf{K}}$ is the least congruence on the ground terms $\mathcal{T}(\Sigma)$ which is a kernel of some homomorphism from T_Σ into some algebra $A \in \mathsf{K}$. Since all $A \in \mathsf{K}$ must satisfy the equations E, we will have that the respective terms will be congruent under $\equiv_{\mathsf{K}}$. (This is just another way of saying that $T_{\mathsf{K}} \in \mathsf{K}$.) For instance, since the equation $\mathsf{s}0 + 0 = \mathsf{s}0$ must be valid in any algebra $A \in \mathsf{K}$, i.e., $(\mathsf{s}0 + 0)^A = (\mathsf{s}0)^A$, we will have that for any homomorphism $\phi : T_\Sigma \to A$, $\phi(\mathsf{s}0 + 0) = \phi(\mathsf{s}0)$, i.e., $\mathsf{s}0 + 0 \equiv_\phi \mathsf{s}0$. Since this must hold for all $A \in \mathsf{K}$ and all ϕ, we obtain that $\mathsf{s}0 + 0 \equiv_{\mathsf{K}} \mathsf{s}0$. Write

4. at least three terms $t \in \mathcal{T}(\Sigma)$ such that $t \equiv_{\mathsf{K}} 0$;
5. at least three terms $t \in \mathcal{T}(\Sigma)$ such that $t \equiv_{\mathsf{K}} \mathsf{ss}0$;
6. "the simplest possible" term $t \in \mathcal{T}(\Sigma)$ such that $\mathsf{s}(\mathsf{s}0 + \mathsf{s}0) \equiv_{\mathsf{K}} t$.
7. Can you guess the structure of the carrier of T_{K}?

EXERCISE 4.5 Consider the following two signatures:

$$\begin{array}{rrrr} \Sigma_L = & \mathcal{S}: & D, L & \\ & \Omega: & \mathit{nil}: & \to L \\ & & \mathit{cons}: & D \times L \to L \\ & & \mathit{tail}: & L \to L \end{array} \qquad \begin{array}{rrrr} \Sigma_{\mathsf{Z}} = & \mathcal{S}: & \mathsf{Z} & \\ & \Omega: & 0: & \to \mathsf{Z} \\ & & \mathsf{s}: & \mathsf{Z} \to \mathsf{Z} \\ & & \mathsf{p}: & \mathsf{Z} \to \mathsf{Z} \\ & & +: & \mathsf{Z} \times \mathsf{Z} \to \mathsf{Z} \\ & & -: & \mathsf{Z} \times \mathsf{Z} \to \mathsf{Z} \end{array}$$

Let Z be a Σ_{Z}-algebra over integers with the natural interpretation of all the operations (i.e., zero, successor, predecessor, addition and multiplication). Define a signature morphism $\sigma : \Sigma_L \to \Sigma_{\mathsf{Z}}$ such that the reduct $Z|_\sigma$ is essentially the algebra of natural numbers with addition. This reduct algebra will not satisfy the equation $\mathit{cons}(x, y) = \mathit{cons}(y, x)$ – why?

EXERCISE 4.6 Let $\Sigma \subset \Sigma' \subset \Sigma''$ be three signatures and $A'' \in \mathsf{Alg}(\Sigma'')$, and $A' \in \mathsf{Alg}(\Sigma')$.

1. Give a counter-example (of Σ, Σ') showing that, in general, $T_\Sigma \neq T_{\Sigma'}|_\Sigma$. Then give an example where $T_\Sigma = T_{\Sigma'}|_\Sigma$.
2. Show that $(A''|_{\Sigma'})|_\Sigma = A''|_\Sigma$.
3. For a Σ-equation e, show that $A' \models e \Leftrightarrow A'|_\Sigma \models e$.

Week 5: Languages and Logics

- Various Equational Languages
- Reasoning Systems
- Inductive Calculi

5.1: Primitive vs. Defined Booleans

The language typically used, and the one we will be using to describe algebras is some subset of first-order language with equality. In fact, since algebras do not contain interpretation of predicates but merely functions, the only primitive operation of our language will be equality "$=$" interpreted as identity and, possibly, Boolean operators like "$\neg$", "$\vee$". Predicates appear in algebras only as defined functions returning Boolean values.

In several examples, we have introduced a sort B and then used the operators like $=$ and $\vee$ to specify some of its properties. But, apparently, we would like our Boolean sort to include such operators as well! Recall, for instance, example 4.19, where we used *defined* Boolean operations to say $isin(x, ins(y, Z)) = or(eq(x, y), isin(x, Z))$. Here *or* was a defined operator taking two arguments from the defined sort B and returning an element of this sort. Similarly *eq* was a defined equality operator returning an element of sort B. That is, in addition to "$\vee$" – which does *not* operate on the sort B – we have a disjunction operator $or : \mathsf{B} \times \mathsf{B} \to \mathsf{B}$ *within* our specification. This is a very subtle and important difference between the *defined Boolean sort and operations* and the used *Boolean meta-values and meta-operations* which correspond to the formulae of our specification language. The two must be clearly kept apart!

Example 5.1 [Primitive vs. defined Booleans]
Consider the following specification

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{BL} =} \\
\mathcal{S}: & \mathsf{B} & \\
\Omega: & \top: & \to \mathsf{B} \\
 & \bot: & \to \mathsf{B} \\
 & or: & \mathsf{B} \times \mathsf{B} \to \mathsf{B} \\
 & eqv: & \mathsf{B} \times \mathsf{B} \to \mathsf{B} \\
\Phi: & 1. & \top \neq \bot \\
 & 2. & eqv(\top, \bot) = \top \\
 & 3. & x = \top \;\vee\; x = \bot \\
 & 4. & or(eqv(x, \top), eqv(x, \bot)) = \top \\
 & 5. & or(x = \top, x = \bot) = \top
\end{array}
$$

The difference between $=$ and $\vee$ on the one hand, and *eqv* and *or* on the other is that the meaning of the former is built into the semantics of the specification language: $=$ *means* identity, and $\vee$ *means* disjunction. The semantics of the other two is determined from the axioms on the basis of the semantics of the specification language. The first axioms states that the interpretation of $\top$ and $\bot$ are distinct objects of the carrier of B, while the second says that the result of *operation eqv* applied to the arguments $\top$ and $\bot$ is identical to the iterpretation of $\top$. *eqv* is just an additional defined operation. Nowhere is it said that, for instance, $eqv(\bot, \top)$ is the same as $eqv(\top, \bot)$.

Similarly, axiom 3. says that the carrier of B has at most two elements interpreting $\top$ and $\bot$ (together with axiom 1. it amounts to B having exactly two elements). Axiom 4., on the other hand, specifies the defined operation *or* as returning $\top$ for the two arguments which are results returned by, respectively, $eqv(x, \top)$ and $eqv(x, \bot)$.

Axiom 5. is *syntactically incorrect*: the operation *or* requires two arguments (i.e., terms) of sort B, while $x = \top$ is not a term and does not return such an element – it has a Boolean *meta-value* which is used for interpreting the formulae of the specification language. We use the usual symbols $=, \vee$, etc. for the meta-operators building the formulae. Textual versions like eq, eqv are defined operators, i.e., terms over a given signature. $eqv(\top, x)$ is a term but not a formula; $\top = x$, on the other hand, is a formula but not a term.

A consequence of this distinction is that the homomorphism condition requires homomorphisms to preseve the operations like *eqv*, *or*, i.e., the defined operators. A homomorphism $\phi : A \to B$ between two algebras over the signature of BL, will have to satisfy $\phi(or(x, y)^A) = or^B(\phi(x), \phi(y))$. But there is no corresponding requirement that it also preserves the formulae.

5.2: The Hierarchy of Languages

From now on we will consider various subclasses $\mathsf{K} \subset \mathsf{Alg}(\Sigma)$, in particular, such subclasses which may be defined by means of some language and are ***model-classes***, i.e., such K that there is a logical language $\mathcal{L}$ and a set of $\mathcal{L}$-formulae Φ, such that K is the class of models of Φ, i.e., $\mathsf{K} = \mathsf{Alg}(\Sigma, \Phi) = \{A \in \mathsf{Alg}(\Sigma) : A \models \Phi\}$.

The axioms are given as a set $\Phi = \{\phi_1, \ldots, \phi_z\}$ where each ϕ_i is a formula of the language we are using at the moment. Given the primitive (meta) Booleans, such a set is understood to be the conjunction $(\forall \overline{x}_1 : \phi_1) \wedge \ldots \wedge (\forall \overline{x}_z : \phi_z)$, where $\overline{x}_i$ are all the variables occurring in ϕ_i. It is always implicitly assumed that variables occurring in our formulae are universally quantified. Since we know that any FOL formula has an equaivalent conjunctive normal form, assuming that each ϕ_i may be an arbitrary disjunction of equalities and negated equalities, i.e., an arbitrary *clause*

$$
\begin{array}{rl}
\phi: & s_1 \neq t_1 \vee \ldots \vee s_n \neq t_n \quad \vee \quad s'_1 = t'_1 \vee \ldots \vee s'_p = t'_p \\
\text{or equivalently}: & s_1 = t_1 \wedge \ldots \wedge s_n = t_n \quad \Rightarrow \quad s'_1 = t'_1 \vee \ldots \vee s'_p = t'_p
\end{array}
\tag{5.13}
$$

we would obtain the full power of universally quantified first order equational formulae. Such a language is quite expressive but, as we will see, expressiveness has its price which one is not always willing to pay. It turns out that the form of the formulae admitted among the axioms has a very close relationship with various properties of the class of models. This relationship will be now the topic of our study. For this purpose, we will arrange the languages in a hierarchy depending on how closely they appraoch the form (5.13). They are determined by the possible values of n and p, i.e., the admissible number of negative and positive equalities in one formula – see table 1.

	language of		n/p	formulae ϕ
1.	clauses	: $\mathcal{L}^{=}_{\vee}$	$n \geq 0,\ p \geq 0$	$s_1 \neq t_1 \vee \ldots \vee s_n \neq t_n \quad \vee \quad s'_1 = t'_1 \vee \ldots \vee s'_p = t'_p$
2.	equational Horn formulae	: $\mathcal{L}^{=}_{H}$	$n \geq 0,\ p \leq 1$	$s_1 \neq t_1 \vee \ldots \vee s_n \neq t_n \quad \vee \quad s = t$ $s_1 \neq t_1 \vee \ldots \vee s_n \neq t_n$
3.	conditional equations	: $\mathcal{L}^{=}_{\Rightarrow}$	$n \geq 0,\ p = 1$	$s_1 \neq t_1 \vee \ldots \vee s_n \neq t_n \quad \vee \quad s = t$
4.	equations	: $\mathcal{L}^{=}$	$n = 0,\ p = 1$	$s = t$
5.	specific equations		$n = 0,\ p = 1$	$s = t$ $s, t \in W \subset \mathcal{T}(\Sigma, X)$

Table 1: The hierarchy of languages

Formulae of $\mathcal{L}^{=}_{\Rightarrow}$ are conditional equations because of the equivalence mentioned in (5.13). As mentioned above, conjunction amounts to listing each conjunct as a separate axiom. In particular, $C \Rightarrow e_1 \wedge \ldots \wedge e_p$ will be expressed by p conditional formulae $C \Rightarrow e_1, \ldots, C \Rightarrow e_p$.

Each higher language contains the lower ones as special subcases. $\mathcal{L}^{=}_{H}$ contains also a special case which we did not mention explicitly of simple positive and negative equations (the third line). There is also a possible variant where all variables are quantified existentially rather than universally. Since existential quantification is much less used in algebra, we will not consider it in detail.

Example 5.2
Simple equations $s = t$ belong to all the languages 1.-4. A conditional equation, e.g., $p = r \Rightarrow s = t$ belongs to languages 1.-3. but not to 4. (nor 5.) Horn formula, unlike the conditional equations, do not require a positive equation, e.g., $s \neq t$ or $s \neq t \vee p \neq r$ will be a Horn formula (belongs to 1.-2.) but not a conditional equation (does not belong to 3.-4. or 5.) Finally, clauses 1. allow also disjunction of positive equations: $s = t \vee p = r$ belongs to 1. but not to any of the lower languages.

The specific equational languages 5. may vary depending on the kind of terms they admit to appear in the equations. In general, not all terms are allowed to occur there (but only some $W \subset \mathcal{T}(\Sigma, X)$). Often, one puts even more specific restrictions allowing only some terms in the LHSs and some (possibly others) in the RHSs of equations. These languages are often of capital importance in computing because they are *executable*. The theory of *term rewriting systems* studies such variants with the view to the properties of the computations which result from various choices of the combination of terms admissible in single equations.

As example 5.2 indicates, moving upwards in this hierarchy one does increase the expressive power of the language. Using merely equations one cannot require an equation to hold relative to other equations. Using only conditional equations one is still unable to require a negation of an equation to hold in all models. Finally, Horn formulae do not allow one to require a disjunction of two equations to hold without forcing one of them to hold in all models.

Notice that any set of axioms in the languages $\mathcal{L}^=$ and $\mathcal{L}^=_\Rightarrow$ will be consistent. These languages do not provide the possibility of expressing negation. First at the level of $\mathcal{L}^=_H$ we may run into a situation where both $s = t$ and $s \neq t$ are present among (or follow from) the axioms.

Given a set of axioms Φ in a higher langugage, it is not certain that one cannot find an equivalent set Φ' in one of the lower languages such that they determine the same model-class. In particular, since higher languages contain the lower ones, an axiomatization Φ' can always be considered as belonging to any higher language. Typically, when saying that Φ is a set of $\mathcal{L}$-formulae, we implicitly assume that it properly belongs to $\mathcal{L}$, i.e., there is no lower language $\mathcal{L}'$ and a set of $\mathcal{L}'$-formulae Φ' such that $\mathsf{Alg}(\Sigma, \Phi) = \mathsf{Alg}(\Sigma, \Phi')$. On the semantic side, saying that K is *equational* we mean that it is definable in $\mathcal{L}^=$, i.e., it is the model class of some set of equations. Saying that it is conditional we mean that it is definable in $\mathcal{L}^=_\Rightarrow$ which does not mean that it is not also equational.

Example 5.3 [Equational axiomatization of Lists]
We want to axiomatize some properties of lists (of integers). To construct lists, we need an operation generating a new (empty) list, and one adding an element (integer) to a list. We also want an operation telling us what is the head of a list, and one returning the tail of a list. Thus, we need the following signature:

$$\begin{array}{rl} \mathcal{S}: & \mathsf{Z}, List \\ nil: & \to List \\ cons: & \mathsf{Z} \times List \to List \\ head: & List \to \mathsf{Z} \\ tail: & List \to List \end{array}$$

and the axioms:

$$\begin{aligned} head(cons(x,l)) &= x \\ tail(cons(x,l)) &= l \end{aligned} \tag{5.14}$$

Notice that this is a very loose axiomatization. It puts only very general restrictions on the models. For instance, it does not say what happens when we try to find $head(nil)$ or $tail(nil)$. We may add the following axioms:

$$\begin{aligned} head(nil) &= 0 \\ tail(nil) &= nil \end{aligned}$$

But one should observe that these axioms are not necessary to express the intentions stated at the begining of the example. They represent a kind of "design decisions" restricting the class of models of (5.14) to the ones which treat these special situations in this particular way.

Let us now add the operation $app : List \times \mathsf{Z} \to List$ which appends an element at the end of the list. It will have to interact with the *cons* operation in the following way:

$$\begin{aligned} app(nil, x) &= cons(x, nil) \\ app(cons(y,l), x) &= cons(y, app(l, x)) \end{aligned} \tag{5.15}$$

The above example shows that some interesting classes of algebras can be axiomatized with simple equations. We will later see that this is not alwas possible – there are classes of algebras for which *no* equational axiomatization exists. Then one has to resort to a more general language.

Example 5.4
A *preorder* on a set S is a binary relation which is reflexive and transitive. Assuming an axiomatization of the Booleans, to axiomatize the algebras of preorders over signature $\langle \mathsf{B}, S; r : S \times S \to \mathsf{B} \rangle$ we need the following two axioms:

$$\begin{array}{rcll} r(x,x) & = & \top & \text{– reflexivity} \\ r(x,y) = \top \ \wedge \ r(y,z) = \top & \Rightarrow & r(x,z) = \top & \text{– transitivity} \end{array}$$

Transitivity requires the conditional equation – as we will see later, it is impossible to express this property by mere equations.

Finally, we give a more complicated axiomatization of finite arrays of integers.

Example 5.5 [1.29 continued]
We assume given an axiomatization of Booleans, integers and natural numbers – the latter with the comparison operations $<, \leq, >, \geq$. In addition, the signature will then contain:

$$\begin{array}{rrrl} \mathcal{S}: & \mathsf{Z}^{\lceil \mathsf{N} \rceil} & & \\ \Omega: & new: & \mathsf{N} \to \mathsf{Z}^{\lceil \mathsf{N} \rceil} & \text{– new array with upper bound} \\ & ins: & \mathsf{Z}^{\lceil \mathsf{N} \rceil} \times \mathsf{N} \times \mathsf{Z} \to \mathsf{Z}^{\lceil \mathsf{N} \rceil} & \text{– insert new value} \\ & ev: & \mathsf{Z}^{\lceil \mathsf{N} \rceil} \times \mathsf{N} \to \mathsf{Z} & \text{– evaluate} \\ & up: & \mathsf{Z}^{\lceil \mathsf{N} \rceil} \to \mathsf{N} & \text{– the upper bound of array} \end{array}$$

Notice that we do not introduce the error constant for the possible error element $\bullet$. Consequently, some of our algebras will and some will not have such an element. The axiomatization requires merely that the results of *ev*aluating arrays outside their bounds are consistently the same.

$$\begin{array}{lrcl} \mathcal{A}: & up(new(n)) & = & n \\ & up(ins(a,n,x)) & = & up(a) \\ & ev(new(m),n) & = & ev(new(p),r) \\ & n \geq up(a) & \Rightarrow & ev(a,n) = ev(new(0),0) \\ & n < up(a) = \top \wedge n = m & \Rightarrow & ev(ins(a,m,x),n) = x \\ & n < up(a) = \top \wedge n \neq m & \Rightarrow & ev(ins(a,m,x),n) = ev(a,n) \end{array}$$

The first two axioms ensure that the *up*per bound of an array is determined at the moment it is created. The third axiom then says that evaluation of a *new* array at any point returns the same value – what this value is, will depend on the actual algebra. The fourth axiom then ensures that evaluating any array at a point outside its *up*per bound will return the same value as *ev*aluating uninitialized array. The last two axioms give the expected relation between *ins*ert and *ev*aluate operations withing the array bounds.

In order to obtain the algebra from example 1.10 among the models of this axiomatization, we have to extend it with the operation *up*. Verify that it will then satisfy all the above axioms.

Verify then that an array algebra constructed as in example 1.10 but *without* any additional elements of sort Z, in which $ev(new(m), n) = 0$, also satisfies the above axioms.

5.3: Reasoning Systems

An axiomatization in some language itself, as described in the previous section, determines a class of its models and provides a syntactic means for studying the properties of these models. However, a model of a given set of axioms Φ will, typically, satisfy many more formulae than those included in Φ. In order to see what consequences our axioms have, the language must be equipped with a reasoning system enabling us to deduce the implications of the axioms we have written.

I. Equational Logic :
The language is $\mathcal{L}^{=}$, that is the formulae are simple equations $s = t$ where $s, t \in \mathcal{T}(\Sigma, X)$. E's denote sets of equations, x is a variable, and the rules of EQ are:

(I.0) $\quad \dfrac{}{E \vdash s = t}$ for any $s = t \in E$

(I.1) $\quad \dfrac{}{E \vdash t = t}$ (reflexivity)

(I.2) $\quad \dfrac{E \vdash t_0 = t_1}{E \vdash t_1 = t_0}$ (symmetry)

(I.3) $\quad \dfrac{E \vdash t_0 = t_1 \ ; \ E_1 \vdash t_1 = t_2}{E \cup E_1 \vdash t_0 = t_2}$ (transitivity)

(I.4) $\quad \dfrac{E \vdash t_0 = t_1}{E \vdash t[x/t_0] = t[x/t_1]}$ (substitutivity-of)

(I.5) $\quad \dfrac{E \vdash t_0 = t_1}{E \vdash t_0[x/t] = t_1[x/t]}$ (substitutivity-in)

Of course, since any equational theory is also a FOL theory, one might use any reasoning system for FOL to prove consequences of a set of equations. The importance of this calclulus (and the following theorem) consists in the fact that it is much simpler than any general system for FOL. Notice that all the rules, except for (I.3), have only single equation in the premise. Thus, a derivation will often consist of just a series of substitutions which can be "glued together" (chained) using the transitivity rule. In many (though not all) cases the reasoning with EQ can be fully automated as it is studied in the theory of term rewriting systems.

The rule (I.0) allows us simply to deduce any axiom from a given set of axioms E. The following three rules (I.1) through (I.3) ensure that the syntactic equality $=$ is an equivalence relation on terms. The last one combines two derivations: if $t_0 = t_1$ can be deduced from the set of equations E, and $t_1 = t_2$ can be deduced from E_1, then the union of the equations in E and E_1 entail the equality $t_0 = t_2$.

The substitutivity rules correspond to the congruence claim. (I.4) allows us to substitute two equal terms into another term: if $t_0 = t_1$ then we can take any term $t(..x..x..)$, possibly containing a variable x, and substitute for *all occurrences* of x, i.e., deduce $t(..t_0..t_0..) = t(..t_1..t_1..)$. The rule (I.5) is a kind of dual allowing us to substitute a term into equal terms: if $t_0(...x...) = t_1(...x...)$ then using this rule we can conclude that $t_0(...t...) = t_1(...t...)$.

Example 5.6
Let $\Sigma = \langle\{S, S_1, S_2\} : a, b : \to S_1, c, d : \to S_2, f : S_1 \times S_2 \to S\rangle$, and $E = \{a = b, c = d\}$. From these two axioms, we prove $f(a, c) = f(b, d)$ as follows:

$$\dfrac{\dfrac{\dfrac{}{E \vdash a = b}\,(\mathrm{I.0})}{E \vdash f(a,c) = f(b,c)}\,(\mathrm{I.4}) \qquad \dfrac{\dfrac{}{E \vdash c = d}\,(\mathrm{I.0})}{E \vdash f(b,c) = f(b,d)}\,(\mathrm{I.4})}{E \vdash f(a,c) = f(b,d)}\,(\mathrm{I.3})$$

In the application of rule (I.4) on the left side, we used the term $t = f(x, c)$, where $f(a, c) = f(x, c)[x/a]$ and $f(b, c) = f(x, c)[x/b]$. In the application on the right side, we substituted the equals $c = d$ into the term $f(b, y)$.

The example illustrates the fact that the relation of EQ-provability, i.e., defined by

$$t \equiv_E s \Leftrightarrow E \vdash_{EQ} s = t, \tag{5.16}$$

is a congruence on the set of terms.

Lemma 5.7 Given a set of Σ-equations E over the set of variables X, the relation $\equiv_E$ is a congruence on the set of terms $\mathcal{T}(\Sigma, X)$.

Proof. That it is an equivalence follows from the rules of reflexivity, symmetry and transitivity. Congruence follows from the substitutivity rules: if $t_0 \equiv_E t_1$, i.e., $E \vdash_{EQ} t_0 = t_1$, then for any term t, in particular for any function symbol $f \in \Sigma$ of arity 1, the rule (I.4) yields $E \vdash_{EQ} f(t_0) = f(t_1)$, i.e., $f(t_0) \equiv_E f(t_1)$. For function symbols of arity 2, this derivation involves rule (I.3) (example 5.6), and for arities greater than 2, also rule (I.5). Details are left as exercise 5.7.

The crucial result concerning this calculus – its soundness and completeness – was proved by Birkhoff in 1935.

Theorem 5.8 Let E be an arbitrary set of Σ-equations and $\mathsf{Alg}(\Sigma, E)$ the set of all Σ-algebras with non-empty carriers satisfying all E. Then, for any Σ-equation $e : \mathsf{Alg}(\Sigma, E) \models e \quad \Leftrightarrow \quad E \vdash_{EQ} e$.

Proof. (Soundness) We only show soundness of (I.1) and (I.4) – the rest is left as exercise 5.8. Let A be an arbitrary algebra in $\mathsf{Alg}(\Sigma, E)$, and $t, t_0, t_1 \in \mathcal{T}(\Sigma, X)$.

To show soundness of (I.1), we have to verify that $A \models t = t$, i.e., that for an arbitrary assignment $\alpha : X \to A$, $t^A_\alpha = t^A_\alpha$. By theorem 4.14 α induces a unique homomorphism $\overline{\alpha} : T_{\Sigma,X} \to A$ which, by exercise 3.6, coincides with the interpretation of t in A under α. Thus we have $t^A_\alpha = \overline{\alpha}(t) = t^A_\alpha$. Since A and α were arbitrary, the claim follows.

For (I.4), let X be the variables occurring in t_0, t_1; Y the variables occuring in $t[x/t_0]$ and $t[x/t_1]$; and Z the set of variables in $t[x]$. (We write here explicitly the possible occurrence(s) of x in t.) Notice that $X \subseteq Y$ and $Y = X \cup Z \setminus \{x\}$ (without loss of generality we may assume that $x \notin X$).

Assume that $A \models t_0 = t_1$, i.e., for all $\alpha : X \to A$, $(t_0)^A_\alpha = (t_1)^A_\alpha$. We have to show that then $A \models t[x/t_0] = t[x/t_1]$. So let $\alpha : Y \to A$ be an arbitrary assignment. Since $X \subseteq Y$, the assumption $A \models t_0 = t_1$ implies that $(t_0)^A_\alpha = (t_1)^A_\alpha = a$ for some $a \in A$. Let $\gamma : Z \to A$ be the same as α with $\gamma(x) \stackrel{\text{def}}{=} a$, which makes $(t_0)^A_\alpha = \overline{\alpha}(t_0) = \gamma(x) = \overline{\alpha}(t_1) = (t_1)^A_\alpha$. From soundness of (I.1) we have that $t[x]^A_\gamma = t[x]^A_\gamma$. But then we also have that $(t[x/t_1])^A_\alpha = t^A_\alpha[\overline{\alpha}(t_1)] = t^A_\alpha[\gamma(x)] = t^A_\gamma[\gamma(x)] = t[x]^A_\gamma = t^A_\gamma[\gamma(x)] = t^A_\alpha[\gamma(x)] = t^A_\alpha[\overline{\alpha}(t_1)] = (t[x/t_1])^A_\alpha$. Since A and α were arbitrary, we obtain that the rule (I.4) is sound.

(Completeness) Write $T_{E,X}$ for $T_{\Sigma,X}/_{\equiv_E}$ - cf. lemma 5.7. We first show

$$T_{E,X} \models s = t \;\Leftrightarrow\; E \vdash_{EQ} s = t \tag{5.17}$$

which, in particular, means that the quotient algebra $T_{E,X} \in \mathsf{Alg}(\Sigma, E)$. We have to show that for any assignment $\alpha : X \to T_{E,X}$, $T_{E,X}$ satisfies all and only such $s = t$ that $E \vdash_{EQ} s = t$, i.e., $s^{T_{E,X}}_\alpha = t^{T_{E,X}}_\alpha \Leftrightarrow E \vdash_{EQ} s = t$.

$\Leftarrow$) Assume $E \vdash_{EQ} s = t$. Any α can be factorized (not necessarily uniquely) as $\alpha' \circ \nu_E$, where $X \xrightarrow{\alpha'} T_{\Sigma,X} \xrightarrow{\nu_E} T_{E,X}$ and $\alpha'(x) \in \alpha(x)$ while $\nu_E(t) = [t]$. Since $T_{\Sigma,X}$ is free in $\mathsf{Alg}(\Sigma)$ and hence, at least, for $\mathsf{Alg}(\Sigma, E)$ over X, we have unique homomorphic extensions $\overline{\alpha'} : T_{\Sigma,X} \to T_{\Sigma,X}$ and $\overline{\alpha' \circ \nu_E} : T_{\Sigma,X} \to T_{E,X}$ which agree on X with, respectively α' and $\alpha' \circ \nu_E$

$$\begin{tikzcd} X \arrow[r, hook, "\iota"] \arrow[dr, "\alpha'"] \arrow[ddr, "\alpha"'] & T_{\Sigma,X} \arrow[d, "\overline{\alpha'}"] \\ & T_{\Sigma,X} \arrow[d, "\nu_E"] \\ 1. & T_{E,X} \end{tikzcd} \qquad \begin{tikzcd} X \arrow[r, hook, "\iota"] \arrow[dr, "\alpha' \circ \nu_E = \alpha"'] & T_{\Sigma,X} \arrow[d, "\overline{\alpha' \circ \nu_E} = \overline{\alpha}"] \\ 2. & T_{E,X} \end{tikzcd}$$

The mappings $\overline{\alpha'} \circ \nu_E$ and $\overline{\alpha' \circ \nu_E}$ will agree on X and so, since $T_{\Sigma,X}$ is generated by X (lemma 2.16), these two mappings will be identical (lemma 3.9) and both equal to $\overline{\alpha}$. Hence, for any $t \in T_{\Sigma,X}$, we will have that $\overline{\alpha}(t) = \nu_E(\overline{\alpha'}(t))$.

Now, by the assumption $E \vdash_{EQ} s = t$ and hence, by (possibly repeated applications of) the rule (I.5), $E \vdash_{EQ} \overline{\alpha'}(s) = \overline{\alpha'}(t)$.[5] That is $\overline{\alpha'}(s) \equiv_E \overline{\alpha'}(t)$, and so $\overline{\alpha}(s) = \nu_E(\overline{\alpha'}(s)) = \nu_E(\overline{\alpha'}(t)) = \overline{\alpha}(t)$. Since α was arbitrary, we obtain that $T_{E,X} \models s = t \Leftarrow E \vdash_{EQ} s = t$, in particular $T_{E,X} \in \mathsf{Alg}(\Sigma, E)$.

$\Rightarrow$) For the opposite, suppose $E \not\vdash_{EQ} s = t$. Consider now the assignment α as in the diagram 2. above with $\alpha' = id$, i.e., $id(x) = x$ for all $x \in X$. As above, we obtain the first of the following equalities $\overline{id \circ \nu_E} = \overline{id} \circ \nu_E = \nu_E$. Since $E \not\vdash_{EQ} s = t$, $s \not\equiv_E t$ and so $\nu_E(s) \neq \nu_E(t)$, i.e., $T_{E,X} \not\models s = t$. Thus (5.17) holds.

So, since $E \vdash_{EQ} e$ for all $e \in E$, we have that $T_{E,X} \in \mathsf{Alg}(\Sigma, E)$ (by $\Leftarrow$). If for some $e : E \not\vdash_{EQ} e$, then $T_{E,X} \not\models e$ (by $\Rightarrow$) and hence $\mathsf{Alg}(\Sigma, E) \not\models e$. This proves the required implication $\mathsf{Alg}(\Sigma, E) \models e \Rightarrow E \vdash_{EQ} e$.

II. Clausal Equational Logic :
Equational calculus is one of the central tools in algebra since many imoprtant classes of algebras are defined by means of simple equations. We will not study in details the calculi for more general languages but merely give the rules for the calculus CEQ of Clausal Euqational Logic. The calculus for Conditional Equations can be obtained from CEQ by restricting all the formulae occurring in the rules to conditional equations. The following formulation is not necessarily the simplest one, but we choose it because its rules correspond very closely to the rules of EQ.

The langugage is $\mathcal{L}^{=}_{\vee}$, i.e., all formulae C are clauses considered as sets of equations and negated equations (i.e., multiplicity and order of their occurrence do not matter): we separate them with comma rather than $\vee$. Φ is a set of clauses, and the rules of CEQ are as follows:

(II.0) $\dfrac{}{\Phi \vdash C, D}$ for any $C \in \Phi$ and any D

(II.1) $\dfrac{}{\Phi \vdash C}$ where $t = t \in C$ or $\{t_0 = t_1, t_0 \neq t_1\} \subseteq C$

(II.2) $\dfrac{\Phi \vdash C, t_0 \sim t_1}{\Phi \vdash C, t_1 \sim t_0}$ $\sim$ is $=$ or $\neq$

(II.3) $\dfrac{\Phi \vdash C, t_0 \sim t_1 \;\; ; \;\; \Phi_1 \vdash C, t_1 = t_2}{\Phi \cup \Phi_1 \vdash C, t_0 \sim t_2}$ $\sim$ is $=$ or $\neq$

[5] It is helpfull (if not necessary) to remember that $\alpha' : X \to T_{\Sigma,X}$ is just a substitution, and its homomorphic extension $\overline{\alpha'} : T_{\Sigma,X} \to T_{\Sigma,X}$ application of this substitution to terms (cf. theorem 4.14 and comment after example 4.15). In particular, the effect of $\overline{\alpha'}(s)$ will be to substitute $\alpha'(x)$ for each variable x occuring in s.

(II.4) $\dfrac{\Phi \vdash C, t_0 = t_1}{\Phi \vdash C, t[x/t_0] = t[x/t_1]}$ (substitutivity-of)

(II.5) $\dfrac{\Phi \vdash C}{\Phi \vdash C[x/t]}$ (substitutivity-in)

(II.6) $\dfrac{\Phi \vdash C, t_0 = t_1 \;\; ; \;\; \Phi_1 \vdash C, t_0 \neq t_1}{\Phi \cup \Phi_1 \vdash C}$ (resolution)

Notice that the rule (II.4) substitutes only into one equation, while the rule (II.5) replaces *all* occurrences of x in the whole clause by a new term t. If the clause C happens to be a single equation we will obtain the corresponding rule (I.5) from EQ. In fact, all the rules of EQ are special cases of the rules given here. The resolution rule does not apply in the simple equational case, since there are no inequations there.

Example 5.9 [5.6 continued]
Consider the same signature as in example 5.6 but without any axioms. Using $\mathcal{L}^{=}$, we could not write (an instance of) the congruence axiom $a = b \wedge c = d \Rightarrow f(a,c) = f(b,d)$ explicitly – instead, we showed that under the assumption that the antecedent of this implication holds (axioms E), then so does the conclusion. In the language $\mathcal{L}^{=}_{\Rightarrow}$, this formula should be provable without any assumptions. In fact, we have the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash a = b, c = d \Rightarrow a = b}\,(\mathrm{II.1})}{\vdash a = b, c = d \Rightarrow f(a,y) = f(b,y)}\,(\mathrm{II.4})}{\vdash a = b, c = d \Rightarrow f(a,c) = f(b,c)}\,(\mathrm{II.5}) \qquad \dfrac{\dfrac{}{\vdash a = b, c = d \Rightarrow c = d}\,(\mathrm{II.1})}{\vdash a = b, c = d \Rightarrow f(b,c) = f(b,d)}\,(\mathrm{II.4})}{\vdash a = b, c = d \Rightarrow f(a,c) = f(b,d)}\,(\mathrm{II.3})$$

Notice that the left derivation might have been shortened by using the same schema as in example 5.6, i.e., applying the rule (II.4) directly to term $f(x,c)$ instead of $f(x,y)$.

5.4: Inductive Calculi

The calculus EQ is sound and complete with respect to the class of *all* models $\mathsf{Alg}(\Sigma, E)$ (of a given set E of equations). However, there are situations (which we will encounter when studying abstract data types) when one is interested only in special subclasses of all models, in particular, only in generated or else only the initial ones. (We will later see that, for a non-void Σ, any class $\mathsf{Alg}(\Sigma, E)$ has an initial model.) Soundness of EQ implies that whatever it proves will hold also in these subclasses. But completeness does not carry over – a proper subclass will have fewer models then the whole class $\mathsf{Alg}(\Sigma, E)$, so it may very well happen that these fewer models satisfy more equations than the whole class.

Example 5.10 [Incompleteness of EQ wrt. initial models]
Consider the following axiomatization (exercise 4.4)

$$\begin{array}{rlrcl} \mathrm{N{\scriptstyle AT}}_+ = \mathcal{S}: & \mathsf{N} & & & \\ \Omega: & 0: & & \to & \mathsf{N} \\ & \mathsf{s}: & \mathsf{N} & \to & \mathsf{N} \\ & +: & \mathsf{N} \times \mathsf{N} & \to & \mathsf{N} \\ E_+: & x + 0 & = & x & \\ & x + \mathsf{s}y & = & \mathsf{s}(x + y) & \end{array}$$

We do not verify it here (we will in example 6.9), but for instance the natural numbers with addition N_+ may serve as an initial model $I(\Sigma, E_+) \in \mathsf{Alg}(\Sigma, E_+)$. So, in particular, $I(\Sigma, E_+) \models x + y = y + x$ since addition in N_+ is commutative. However, this equation is *not* provable from E_+ with EQ. The simplest way to convince oneself about it is to find a model M of NAT_+ which does not satisfy this equation. (Then, since EQ is sound, we will know that it cannot prove it.)

Such a model M may be the following: the carrier is the set of all 2×2-matrices of natural numbers, where 0^M is the matrix $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, the successor s^M is the scalar multiplication by an arbitrary number $n > 0$, and $+^M$ is the matrix multiplication. Verify that it does yield a model of NAT_+. But it is well known (and easy to check for those who do not know it) that the matrix multiplication isn't commutative. So $M \not\models x + y = y + x$.

The example shows that initial models of some E may satisfy more equations than all the models of E, and hence some equations which are not provable by EQ. The following result is weaker than 5.8 in that it addresses only ground equations.

Theorem 5.11 Let Σ be non-void, E be a set of Σ-equations and e be an arbitrary ground Σ-equation. Then $I(\Sigma, E) \models e \Leftrightarrow E \vdash_{EQ} e$.

PROOF. $\Leftarrow$) Soundness is a direct consequence of soundness of EQ. Since, for any equation $e : E \vdash_{EQ} e \Rightarrow \mathsf{Alg}(\Sigma, E) \models e \Rightarrow I(\Sigma, E) \models e$, this holds in particular for ground e.

$\Rightarrow$) Follows from completeness of EQ and lemma 4.17. Σ is non-void and the initial model $I(\Sigma, E)$ is generated. By lemma 4.17, $I(\Sigma, E)$ satisfies exactly the same ground equations as the whole class $\mathsf{Alg}(\Sigma, E)$.

Example 5.12 [5.10 continued]

For any ground terms s, t we will be able to prove $E_+ \vdash_{EQ} s + t = t + s$. For instance,

$$E_+ \vdash_{EQ} \quad \mathsf{ss0} + \mathsf{s0} = \mathsf{s}(\mathsf{ss0} + 0) = \mathsf{sss0} = \mathsf{s}(\mathsf{s}(\mathsf{s0} + 0)) = \mathsf{s}(\mathsf{s0} + \mathsf{s0}) = \mathsf{s0} + \mathsf{ss0}$$

And so, each such equation will hold in all the models of E_+. In the model M (for instance, with $n = 2$), this equation will look as follows:

$$(2 * 2 * \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}) * (2 * \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}) = \begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix} * \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 8 & 0 \\ 0 & 8 \end{bmatrix} =$$

$$\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} * \begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix} = (2 * \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}) * (2 * 2 * \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix})$$

One of the specific features of initial models is that they are generated. This means that even if we write a non-ground equation, its variables may refer to elements of the carrier which, in the initial models, will also be denotable by some ground terms. In other words, in a generated (and hence also initial model), if all ground instances of an equation hold, then the equation itself holds. For instance, if $x + y = y + x$ holds for all ground substitutions for x and y, then in a generated model the equation itself will hold. In a general model, however, which may contain undenotable "junk" elements, the fact that all ground instances of an equation hold does not necessarily mean that there are no other (undenotable) elements in the carrier for which the equation does not hold. This observation is expressed in the following lemma.

Lemma 5.13 Let Σ be non-void, $I(\Sigma, E)$ be an initial model of a set of equations E and $Gen(\Sigma, E)$ be the class of all generated models of E. Let e be an arbitrary (not necessarily ground) Σ-equation. Then $I(\Sigma, E) \models e \Leftrightarrow Gen(\Sigma, E) \models e$.

PROOF. $\Leftarrow$) is trivial since $I(\Sigma, E) \in Gen(\Sigma, E)$.

$\Rightarrow$) If $I(\Sigma, E) \models e$ then it models all ground instances $e\sigma$ of e (where σ ranges over all ground substitutions to the variables in e). Hence $\mathsf{Alg}(\Sigma, E) \models e\sigma$ and, in particular, $Gen(\Sigma, E) \models e\sigma$ for all ground σ. But since for each $G \in Gen(\Sigma, E)$, for any element $x \in |G|$ there is a ground term t with $t^G = x$, this means that $G \models e$.

The observation that satisfaction of all ground instances of an equation by a generated model G implies that $G \models e$, gives rise to the extension of the calculus EQ with the so called ω-rule:

III. Inductive Calculus :
EQI is obtained by augmenting the rules of EQ with the ω-rule. Let $s, t \in \mathcal{T}(\Sigma, X)$, and $\sigma : X \to \mathcal{T}(\Sigma)$ be ground substitution:

$$\text{(III.1)} \qquad \frac{E \vdash_{EQI} s\sigma = t\sigma \quad \text{for all ground } \sigma}{E \vdash_{EQI} s = t} \qquad (\omega\text{-rule})$$

With this extension, we obtain the result for the initial (and hence generated) models which coresponds to theorem 5.8.

Theorem 5.14 Let Σ be non-void, E a set of Σ-equations and e an arbitrary Σ-equation. Then $I(\Sigma, E) \models e \Leftrightarrow Gen(\Sigma, E) \models e \Leftrightarrow E \vdash_{EQI} e$.

As a matter of fact, the ω-rule can be added to other calculi to obtain the same effect. Adding the corresponding rule (where instead of a simple equation $s = t$ we have a clause C) to CEQ will give us a sound and complete calculus for clauses.

5.4.1: Structural Induction

Notice that, unlike the other rules we have seen so far, this one is (in general) *infinitary*, i.e., it has infinitely many premisses, since the set of ground terms is typically (though not always) infinite. This makes it very hard to use in practice, although there are techniques of inductive proofs which in some cases allow one to automate the use of this rule.

Among various strategies for using the ω-rule, one of the central ones is *structural induction* on ground terms. Assuming that Σ is non-void, we first prove the premise of the ω-rule for all constants Σ_ε. Then, for any function symbol (of appropriate sort) $f : s_1 \times ... \times s_n \to s$, we show the premise for $f(t_1...t_n)$ *under the assumption* that it holds for all $t_1...t_n$ (of appropriate sorts).

Example 5.15

In a somehow informal way, exercise 4.4 attempted to show that each equivalence class in the algebra T_K contained a *cannonical representative*, namely a term of the form s^n0. We now show that, in fact, each congruence class determined by the signature and axioms as in this exercise (and example 5.10) contains a term of this form, i.e., $\forall [t] \in |T_E|\ \exists n \geq 0 : \mathsf{s}^n0 \in [t]$ or, equivalently:

$$\forall t \in \mathcal{T}(\Sigma)\ \exists n \geq 0 : E \vdash_{EQ} t = \mathsf{s}^n0 \tag{5.18}$$

and we prove it by structural induction on Σ-terms:

1. For $t = 0$, we have trivially $0 = 0 = \mathsf{s}^00$.
2. For $t = \mathsf{s}x$, by IH we may assume that $x = \mathsf{s}^n0$ and so $t = \mathsf{s}^{n+1}0$.
3. For $+$ we have to show that the assumption of IH, i.e., $x = \mathsf{s}^n0$ and $y = \mathsf{s}^m0$, implies that also $x + y = \mathsf{s}^k0$ for some $k \geq 0$. We prove by (sub)induction on m that $k = n + m$:
   (a) $m = 0$ and first axiom yield $x + 0 = \mathsf{s}^n0 + 0 \stackrel{1}{=} \mathsf{s}^{n+0}0$.
   (b) As the induction hypothesis IH' we have now that, for all $x = \mathsf{s}^n0$ and for an $y = \mathsf{s}^m0$, $x + y = \mathsf{s}^{n+m}0$. Then $x + \mathsf{s}y \stackrel{2}{=} \mathsf{s}(x + y) = \mathsf{s}(\mathsf{s}^n0 + \mathsf{s}^m0) \stackrel{\text{IH'}}{=} \mathsf{s}(\mathsf{s}^{n+m}0) = \mathsf{s}^{n+m+1}0$.

Thus we have shown (5.18). Notice that we did *not* prove that such an n is *unique* for each t!

Remark 5.16

What we actually showed in example 5.15 is that congruence classes wrt. the congruence $\equiv_E$ as defined in (5.16) satisfy the claim. In exercise 4.4 we indicated that this congruence is related to (finer than) the congruence $\equiv_\mathsf{K}$ from definition 4.20. That these two actually coincide will be shown later in lemma 6.6.

Example 5.17 [5.10 continued]
Take the signature and the axioms from the example 5.10 and consider the formula $x+y=y+x$. Its inductive proof by structural induction will go as follows. Since there are two variables in the formula, we have to perform induction on both. We use nested induction – first on y with subinduction on x. In the proof, we use symbols m for x and n for y to emphasize the fact that they are not usual variables but range over "simpler" terms.

1. $n=0$: We must show $m+0=0+m$ - since 0 is the only constant. We show it by (sub)induction on m.
   (a) $m=0$: We have to show $0+0=0+0$ which follows immediately by rule (I.1).
   (b) $\mathsf{s}m$: So, assuming the (sub)induction hypothesis $IH_m : m+0=0+m$ we have to show that it implies $\mathsf{s}m+0=0+\mathsf{s}m$. We have $\mathsf{s}m+0 \stackrel{1}{=} \mathsf{s}m \stackrel{1}{=} \mathsf{s}(m+0) \stackrel{IH_m}{=} \mathsf{s}(0+m) \stackrel{2}{=} 0+\mathsf{s}m$.
2. $\mathsf{s}n$: Now, assuming the induction hypothesis IH_n, namely that $m+n=n+m$ holds for all m, we have to show $m+\mathsf{s}n=\mathsf{s}n+m$. Again, we use subinduction on m:
   (a) $m=0$: We have to show $0+\mathsf{s}n=\mathsf{s}n+0$. We have $0+\mathsf{s}n \stackrel{1}{=} \mathsf{s}(0+n) \stackrel{IH_n}{=} \mathsf{s}(n+0) \stackrel{1}{=} \mathsf{s}n \stackrel{1}{=} \mathsf{s}n+0$.
   (b) $\mathsf{s}m$: Here, in addition to the main induction hypothesis IH_n (which applies to all m), we assume the subinduction hypothesis $IH_m : m+\mathsf{s}n=\mathsf{s}n+m$ and show $\mathsf{s}m+\mathsf{s}n=\mathsf{s}n+\mathsf{s}m$ as follows: $\mathsf{s}n+\mathsf{s}m \stackrel{2}{=} \mathsf{s}(\mathsf{s}n+m) \stackrel{IH_m}{=} \mathsf{s}(m+\mathsf{s}n) \stackrel{2}{=} \mathsf{ss}(m+n) \stackrel{IH_n}{=} \mathsf{ss}(n+m) \stackrel{2}{=} \mathsf{s}(n+\mathsf{s}m) \stackrel{IH_n}{=} \mathsf{s}(\mathsf{s}m+n) \stackrel{2}{=} \mathsf{s}m+\mathsf{s}n$.

Notice that in the last point it was necessary to use both – the main and the subinduction hypothesis.

Although the structural induction is a powerful mechanism it is not, unlike the ω-rule, complete. I.e., there are cases when some equations valid in the initial model won't be provable by the structural induction.

Exercises (week 5)

EXERCISE 5.1 Recall the algebra of infinite arrays from example 1.9. Assuming some given sort A and natural numbers N, write the signature for these algebras with the operations $null$, evaluate and insert. We want to put the following requirements on these algebras:

- Two arrays $g, h \in A^{\mathsf{N}}$ should be considered equal if they have exactly the same values stored at the same indices, i.e., if for all $i \in \mathsf{N} : ev(g,i) = ev(h,i)$. (In particular, the order in which elements are inserted at different indices should not matter.)
- Evaluating the value stored at an index i returns the element which was inserted there most recently.

Express these claims as formulae over your signature. To which of the languages from table 1 does your axiomatization belong?

EXERCISE 5.2 Use the axioms E from exercise 4.4 and the equational calculus to prove that $E \vdash_{EQ} \mathsf{ss}0 + \mathsf{s}0 = \mathsf{sss}0$. Specify explicitly the substitutions whenever you are using the rules (I.4) or (I.5).

EXERCISE 5.3 Recall the axiomatization of groups from example 1.24. Use EQ to show that for any group G and any $x, y \in G$ the following equations hold

1. $(x^-)^- = x$
2. $(x \circ y)^- = y^- \circ x^-$

EXERCISE 5.4 Using the axioms from example 5.3 prove the following two equations:

1. $head(app(cons(x,l),y)) = x$
2. $tail(app(cons(x,l),y)) = app(l,y)$.

Now, extend the signature with a new operation $rev : List \to List$ which reverses a list (e.g., $rev([1,3,5,2]) = [2,5,3,1]$). Write equational axioms for this operation, and then prove that $rev(cons(1,cons(2,nil))) = cons(2,cons(1,nil))$. Can you prove from your axioms that $rev(rev(l)) = l$?

EXERCISE 5.5 Axiomatize the algebras of stacks of integers over the following signature:

$$\Sigma = \langle \mathsf{Z}, St \ ; \ e : \to St; \ push : \mathsf{Z} \times St \to St, \ pop : St \to St \rangle$$

The constant e returns an empty stack, $push(x,S)$ pushes element x on the top of the stack S, and $pop(S)$ returns the stack S with the topmost element (the one $push$ed most recently) removed. pop of the empty stack may return the empty stack (since there is nothing to remove from its top).

1. Write equational axioms (in $\mathcal{L}^=$) ensuring these properties.
2. Extend Σ with a new operation $top : St \to \mathsf{Z}$. We want it to return the topmost element of the stack (the one which is removed by pop). Write necessary axioms in appropriate language (not necessarily in $\mathcal{L}^=$) for this operation. Notice that $top(e)$ is a special case.
3. Can you prove from your axioms the equation $push(top(S), pop(S)) = S$? If not, give an algebra satisfying all your axioms but where this equation does not hold.

EXERCISE 5.6 Refer to example 2.32 and exercise 2.8. Prove by induction on the structure of Σ_{B}-terms that given the equations E from example 2.32, any ground Σ_{B}-term is identified with either $\top$ or $\bot$.

EXERCISE 5.7 Complete the proof of lemma 5.7. For any function symbol $f \in \Sigma$ with a given number n of arguments, you have to show that from $\vdash_{EQ} s_1 = t_1, \ldots, \vdash_{EQ} s_n = t_n$, one can obtain a proof $\vdash_{EQ} f(s_1,\ldots,s_n) = f(t_1,\ldots,t_n)$. Fix an n and show this by induction going from 1 to n. I.e., show first that from $\vdash_{EQ} s_1 = t_1$ you can get a proof of $\vdash_{EQ} f(s_1,x_2,\ldots,x_n) = f(t_1,x_2,\ldots,x_n)$, and then that induction hypothesis for i enables you to perform one more substitution of s_{i+1}, resp. t_{i+1}, for x_{i+1}. (Example 5.6 shows a special case – you will need all the rules (I.4), (I.5) and (I.3).)

EXERCISE 5.8 Prove soundness part of theorem 5.8.

Week 6: Equational Classes and Variety Theorem

- Classes of Algebras axiomatized by Equations
- Initial Algebras in Equational Classes
- Variety Theorem

We say that a *language $\mathcal{L}$ is closed under* some semantic construction (isomorphisms, quotients, homomorphic images etc.) iff for an arbitrary signature Σ and set of axioms $\Phi \subseteq \mathcal{L}(\Sigma)$, the model class $\mathsf{Alg}(\Sigma, \Phi)$ is closed under this construction. Notice that this statement does not mean that there is no Φ for which the model class is closed under the construction – only that using this language we are *not guaranteed* that it is.

As a terminological variation, one speaks about respective classes: an *equational class* is a class of models of some set of axioms in $\mathcal{L}^{=}$, a *conditional equational class* a class of models of a set of axioms in $\mathcal{L}^{=}_{\Rightarrow}$. Thus saying that "every equational class is closed under products" means the same as saying that "$\mathcal{L}^{=}$ is closed under products". For simplicity, we will drop Σ from the notation – it is always implicitly assumed given.

In this section we study model-classes of axioms which are simple equations and prove the classical theorem that there is a one-to-one correspondence between such classes and the so called *varieties*. The corresponding properties of more expressive languages will be the object of our study in next week.

Definition 6.1 A class $\mathsf{K} \subseteq \mathsf{Alg}(\Sigma)$ is a *variety* iff it is closed under subalgebras, homomorphic images and products (also empty).

Notice that since closure under homomorphic images implies closure under quotients, a variety K will always have the unit algebra obtained by taking the quotient of an arbitrary $A \in \mathsf{K}$ by the maximal congruence $A \times A$.

Before looking closer at varieties, we prove a lemma which will be useful in several contexts.

Lemma 6.2 Let $\{A_i\}_{i\in I} \subseteq \mathsf{Alg}(\Sigma)$ be non-empty, $\Pi A = \prod_{i\in I} A_i$ be its product, $s = t$ be a Σ-equation with variables X, and $\alpha : X \to \Pi A$ be any assignment. Then $\overline{\alpha}(t) = \overline{\alpha}(s) \Leftrightarrow \forall i : \overline{\alpha \circ \pi_i}(t) = \overline{\alpha \circ \pi_i}(s)$. (In particular, $\Pi A \models s = t \Leftrightarrow A_i \models s = t$ for all $i \in I$.)

Proof. The main claim is that ΠA satisfies an equation under an assignment α iff all components satisfy the equation under the respective assignments $\alpha \circ \pi_i$. Informally, elements of the product are tuples $a = \langle a_1, a_2 ... a_n\rangle$ and two such tuples are equal $a = b$ if they are equal componentwise, i.e., for all $i : a_i = b_i$. This gives the intuitive meaning to the claim.

More formally, let X be all the variables in an equation $s = t$, and $\alpha : X \to \Pi A$ be any assignment. We have the following diagram for each i

$$\begin{tikzcd}
X \arrow[d, "\alpha\circ\pi_i"'] \arrow[rd, "\alpha"' near start] & T_{\Sigma,X} \arrow[ld, "\overline{\alpha\circ\pi_i}"' near start] \arrow[d, "\overline{\alpha}"] \\
A_i & \Pi A \arrow[l, "\pi_i"]
\end{tikzcd}$$

Since $T_{\Sigma,X}$ is free for $\mathsf{Alg}(\Sigma)$, the assignments α and $\alpha \circ \pi_i$ induce the unique homomorphic extensions $\overline{\alpha}$ and $\overline{\alpha \circ \pi_i}$, and by the universal product property (lemma 3.12) $\overline{\alpha \circ \pi_i} = \overline{\alpha} \circ \pi_i$, and π_i are surjective.

$\Rightarrow$) If $\overline{\alpha}(t) = \overline{\alpha}(s)$, then $\overline{\alpha \circ \pi_i}(t) = \pi_i(\overline{\alpha}(t)) = \pi_i(\overline{\alpha}(s)) = \overline{\alpha \circ \pi_i}(s)$. (In particular, if $\overline{\alpha}(t) = \overline{\alpha}(s)$ holds for all α, i.e., $\Pi A \models t = s$, then, since π_i are surjective, we have that $A_i \models t = s$.)

$\Leftarrow$) Conversly, assume that for all $i : \overline{\alpha \circ \pi_i}(t) = \overline{\alpha \circ \pi_i}(s)$, and let $\iota_t = \overline{\alpha}(t)$, $\iota_s = \overline{\alpha}(s)$. Then $\iota_t(i) = \pi_i(\overline{\alpha}(t)) = \overline{\alpha \circ \pi_i}(t) = \overline{\alpha \circ \pi_i}(s) = \pi_i(\overline{\alpha}(s)) = \iota_s(i)$. Since we thus have that for all $i : \iota_t(i) = \iota_s(i)$, it follows that $\iota_t = \iota_s$, i.e., $\overline{\alpha}(t) = \overline{\alpha}(s)$. (In particular, if for all $i : A_i \models t = s$, then for all i and $\alpha_i : X \to A_i$ we have $\overline{\alpha_i}(t) = \overline{\alpha_i}(s)$. By the universal property of products we then get a unique $\overline{\alpha} : T_{\Sigma,X} \to \Pi A$ such that $\overline{\alpha_i} = \overline{\alpha} \circ \pi_i$ and so, by the above argument, $\overline{\alpha}(t) = \overline{\alpha}(s)$. Actually, $\overline{\alpha}$ will be induced by an $\alpha : X \to \Pi A$ with $\alpha_i = \alpha \circ \pi_i$, and any assignment $X \to \Pi A$ can be obtained in this way. Hence we get $\Pi A \models t = s$.)

This lemma states the important property relating the semantic construction (of non-empty products) with the fact of satifaction of equations. Its concise formulation would be: equations are preserved and reflected under non-empty products. That they are *preserved* under non-empty products means that if all the algebras $\{A_i\}_{i\in I}$ satisfy an equation then so does their product. That equations are *reflected* by non-empty products means that if a product of a non-empty family $\{A_i\}_{i\in I}$ satisfies an equation then so do all the component algebras A_i (or dually, if some A_i does *not* satisfy an equation then neither does non-empty product involving A_i.)

As a consequence, we obtain that if E is a set of simple Σ-equations and $\{A_i\}_{i\in I} \subseteq \mathsf{Alg}(\Sigma, E)$ is any family of models of E, then also their product will satisfy E, i.e., it will belong to $\mathsf{Alg}(\Sigma, E)$. In short, $\mathsf{Alg}(\Sigma, E)$ is closed under products and, since E was arbitrary, we can say that $\mathcal{L}^{=}$ is closed under products.

However, equational classes are guaranteed to preserve more properties – in fact, each equational class is a variety.

Theorem 6.3 Every equational class $\mathsf{K} \subseteq \mathsf{Alg}(\Sigma)$ is a variety.

Proof. By assumption, there is a set of Σ-equations E such that $\mathsf{K} = \mathsf{Alg}(\Sigma, E)$. Let X be the set of all variables occurring in E.

1. K is closed under subalgebras.
   Let $A \in \mathsf{K}$ and $B \sqsubseteq A$. Any assignment $\alpha : X \to B$ is also an assignment $\alpha : X \to A$. So, for any equation $s = t \in E$, we will have $s^B_\alpha = s^A_\alpha = t^A_\alpha = t^B_\alpha$, where the middle equality holds since $A \models E$, and the others since $B \sqsubseteq A$.
2. K is closed under homomorphic images.
   Let $A \in \mathsf{K}$ and $\phi : A \to B$ be a surjective homomorphism. Then, any assignment $\beta : X \to B$ can be factorized (not necessarily uniquely) as $\beta = \alpha \circ \phi$ where $\alpha : X \to A$. Let $s = t$ be an arbitrasry equation in E. Then, since $A \models E$ so $s^A_\alpha = t^A_\alpha$, i.e., $\phi(s^A_\alpha) = \phi(t^A_\alpha)$, which implies the middle equality in $s^B_\beta = s^B_{\alpha\circ\phi} = t^B_{\alpha\circ\phi} = t^B_\beta$. Since β was arbitrary, this means that $B \models s = t$.
3. K is closed under products.
   Closure under non-empty products follows directly from lemma 6.2. Unit algebra is terminal in $\mathsf{Alg}(\Sigma)$ and it satisfies all equations, hence it belongs to any equational class.

6.1: Initial algebras in equational classes

As an immediate consequence of theorem 6.3, we conclude that

Corollary 6.4 For any set of Σ-equations E, $\mathsf{K} = \mathsf{Alg}(\Sigma, E)$ contains free algebras. If Σ is non-void, then K has initial objects.

Proof. Since K is a variety, it satisfies the conditions of theorem 4.22.

In fact, the congruence $\equiv_{\mathsf{K}}$ used in the proof of theorem 4.22 (definition 4.20) will be the same as the congruence $\equiv_E$ induced by the equations E. Hence a free algebra in K can be obtained as the quotient $T_{\Sigma,X}/_{\equiv_E}$. We show this fact using the following two lemmata:

Lemma 6.5 Let $\mathsf{K} \subseteq \mathsf{Alg}(\Sigma)$ and X be a set of variables. Then $\equiv_{\mathsf{K}} = \sim_{\mathsf{K}}$ where both relations are the following congruences on $\mathcal{T}(\Sigma, X)$:

$$\equiv_{\mathsf{K}} \stackrel{\text{def}}{=} \bigcap_{A\in\mathsf{K}} \{\equiv_\phi : \text{ where } \phi : T_{\Sigma,X} \to A \text{ is a homomorphism}\} \quad \text{(definition 4.20)}$$
$$\sim_{\mathsf{K}} \stackrel{\text{def}}{=} \{\langle s; t\rangle : A \models s = t \text{ for all } A \in \mathsf{K}\}$$

Proof. We have the following chain of equalities:

$$\begin{array}{rcl}
\sim_{\mathsf{K}} & \stackrel{\text{def}}{=} & \{\langle s;t\rangle : A \models s = t \text{ for all } A \in \mathsf{K}\} \\
& = & \bigcap_{A\in\mathsf{K}}\{\langle s;t\rangle : A \models s = t\} \\
& = & \bigcap_{A\in\mathsf{K}}\{\langle s;t\rangle : s^A_\alpha = t^A_\alpha \text{ where } \alpha : X \to A\} \\
& = & \bigcap_{A\in\mathsf{K}}\{\langle s;t\rangle : s \equiv_{\overline{\alpha}} t \text{ where } \alpha : X \to A\} \\
& = & \bigcap_{A\in\mathsf{K}}\{\equiv_{\overline{\alpha}} : \text{where } \alpha : X \to A\} \\
& = & \bigcap_{A\in\mathsf{K}}\{\equiv_{\phi} : \text{where } \phi : T_{\Sigma,X} \to A \text{ is a homomorphism}\} \stackrel{\text{def}}{=} \equiv_{\mathsf{K}}
\end{array}$$

The last but one equation follows since any assignment α induces a unique homomorphism $\overline{\alpha} : T_{\Sigma,X} \to A$ while any homomorphism $\phi : T_{\Sigma,X} \to A$ maps all the variables X to some elements in A, i.e., is also induced by such an assignment.

Lemma 6.6 Let E be some set of equations with variables X, and $\mathsf{K} = \mathsf{Alg}(\Sigma, E)$. Then $\equiv_{\mathsf{K}} = \equiv_E$, where for $s, t \in \mathcal{T}(\Sigma, X) : s \equiv_E t \stackrel{(5.16)}{\Leftrightarrow} E \vdash_{EQ} s = t$.

Proof. Direct consequence of lemma 6.5 and theorem 5.8. We have

$$s \equiv_E t \stackrel{\text{def}}{\Leftrightarrow} E \vdash_{EQ} s = t \stackrel{5.8}{\Leftrightarrow} \mathsf{K} \models s = t \stackrel{\text{def}}{\Leftrightarrow} s \sim_{\mathsf{K}} t \stackrel{6.5}{\Leftrightarrow} s \equiv_{\mathsf{K}} t.$$

We thus obtain a simple corollary which, actually, is a very significant result in the theory of abstract data types.

Theorem 6.7 For any set of equations E and variables X, $T_{\Sigma,X/\equiv_E}$ is free in $\mathsf{Alg}(\Sigma, E)$. (If Σ is non-void, then $T_{\Sigma}/_{\equiv_E}$ is initial in $\mathsf{Alg}(\Sigma, E)$.)

Proof. By theorem 6.3, $\mathsf{K} = \mathsf{Alg}(\Sigma, E)$ is a variety, so (by theorem 4.22) $T_{\Sigma,X/\equiv_{\mathsf{K}}}$ is free in K. But by lemma 6.6, $\equiv_{\mathsf{K}} = \equiv_E$, and so $T_{\Sigma,X/\equiv_{\mathsf{K}}} = T_{\Sigma,X/\equiv_E}$.

Remark 6.8 [Construction of initial algebras]

The significance of theorem 6.7 consists in that it gives us the means of actually *constructing* initial (free) algebras for equational axioms. Congruence $\equiv_E$ is determined "syntactically" and thus much more tractable than $\equiv_{\mathsf{K}}$. Typically, writing such axioms we have a specific algebra A in mind and we are trying to axiomatize it – that is, we want to obtain a model class where A is initial. To verify that we have succeded, we have then to check that the quotient algebra $T_{\Sigma}/_{\equiv_E}$ is isomorphic to our intended A. This is done as follows:

1. Verify that $A \models E$, i.e., $A \in \mathsf{Alg}(\Sigma, E)$.
2. Determine the carrier of $T_E = T_{\Sigma}/_{\equiv_E}$. Typically, try to find the *cannonical representatives* for each equivalence class $[t]$, i.e., a unique term $C[t] \in \mathcal{T}(\Sigma)$ such that $C[t] \in [t]$, and $[s] \neq [t]$ iff $C[s] \neq C[t]$. (This is typically an elaborate inductive argument.)
3. Initiality of T_Σ in $\mathsf{Alg}(\Sigma)$ and of T_E in $\mathsf{Alg}(\Sigma, E)$, gives the following homomorphisms:

$$\begin{tikzcd}
T_\Sigma \arrow[rr, "\omega"] \arrow[dr, "\nu_E"'] & & A \\
& T_E \arrow[ur, "\phi"'] &
\end{tikzcd}$$

   For any $t \in \mathcal{T}(\Sigma) : \nu_E(t) = [t]$ and, since both ω and ϕ are unique, $\omega = \nu_E \circ \phi$. This means that the image of $[t]$ under ϕ can be determined as the value of its $C[t]$ in A, i.e., $\phi([t]) = \phi([C[t]]) = \phi(\nu_E(C[t])) = \omega(C[t]) = C[t]^A$. This forces ϕ to be a homomorphism and there is no need to verify it each time. (The burden of this verification was moved to finding the cannonical representatives.)
4. This fact is used to show that ϕ is an isomorphism. For $C[t] \neq C[s]$ we have to show that $C[t]^A = \omega(C[t]) \neq \omega(C[s]) = C[s]^A$ (injectivity), and that for any $a \in |A|$ there is a $C[t] \in \mathcal{T}(\Sigma) : a = C[t]^A$ (surjectivity).

Example 6.9 [5.15 continued]

Recall the signature and axiomatization from exercise 4.4 and example 5.15. Its intention was to specify the usual natural numbers with addition, i.e., the algebra $N_+ = \langle \mathsf{N}; 0, \_+1, \_+\_\rangle$. The exercise attempted to indicate (if not verify) that the carrier of the quotient algebra will contain exactly the congruence classes $[0], [\mathsf{s}0], [\mathsf{ss}0]$, etc. In particular, each such a congruence class contains the cannonical representative $\mathsf{s}^n 0$ for $n \geq 0$. This, i.e., the claim (5.18), was proved in example 5.15.

What we did not proved there, was the *uniqueness* of such representatives. We have to show not only that for each term t there is an n and a proof $E \vdash_{EQ} t = \mathsf{s}^n 0$, but also that for no t there exist $n \neq m$ with $E \vdash_{EQ} \mathsf{s}^n 0 = t$ and $E \vdash_{EQ} t = \mathsf{s}^m 0$. Since these would give a proof of $\mathsf{s}^n 0 = \mathsf{s}^m 0$, it suffices to show that no such equality is provable, i.e., for any $n \neq m : E \not\vdash_{EQ} \mathsf{s}^n 0 = \mathsf{s}^m 0$. This is a hard part, which does not follow by a mere syntactic proof but requires a semantic argument. It can be verified by, for instance, checking that the intended natural numbers algebra N_+ satisfies the axioms E and hence belongs to the model class (this does not, as yet, show that it is initial!). N_+ does not satisfy any such equality, and so, since EQ is sound, no such proof can exist.

Having thus determined the cannonical representatives for each class, we obtain the homomorphism into N_+ by $\phi([\mathsf{s}^n 0]) = (\mathsf{s}^n 0)^{N_+} = n$. The general facts (from remark 6.8) guarantee that it is a homomorphism. Checking that it is injective and surjective is trivial.

6.2: Variety Theorem

The real significance of varieties consists in the fact that they not only are among the equational classes but are exactly the equational classes.

Theorem 6.10 [Variety Theorem] For any Σ and $\mathsf{K} \subseteq \mathsf{Alg}(\Sigma)$: K is a variety iff it is equational.

Proof. $\Leftarrow$) is theorem 6.3 so, for proving $\Rightarrow$), assume that K is a variety. We have to find a set of equations E such that $\mathsf{K} = \mathsf{Alg}(\Sigma, E)$. K is non-empty, since closure under products (also empty products!) means that the unit algebra – the empty product $\Pi\emptyset \in \mathsf{K}$.

(1) Let X be an infinite set of variables and let

$$Eq_{\mathsf{K}}(\Sigma, X) \stackrel{\text{def}}{=} \{s = t : s, t \in \mathcal{T}(\Sigma, X) \wedge \mathsf{K} \models s = t\}$$

be the set of all Σ, X-equations which hold in each $A \in \mathsf{K}$. Consider so the class of algebras

$$\mathsf{K}^* \stackrel{\text{def}}{=} \mathsf{Alg}(\Sigma, Eq_{\mathsf{K}}(\Sigma, X))$$

Obviously $\mathsf{K} \subseteq \mathsf{K}^*$. Showing the opposite inclusion, we will prove the theorem.

(2) We first show that for any set of variables $Y : Eq_{\mathsf{K}}(\Sigma, Y) = Eq_{\mathsf{K}^*}(\Sigma, Y)$. For any $e \in Eq_{\mathsf{K}^*}(\Sigma, Y) : \mathsf{K}^* \models e$, so since $\mathsf{K} \subseteq \mathsf{K}^*$, $\mathsf{K} \models e$ and hence $e \in Eq_{\mathsf{K}}(\Sigma, Y)$. Conversely, let $e \in Eq_{\mathsf{K}}(\Sigma, Y)$, V be all the variables occurring in e, and $\sigma : V \to X$ be an injection (renaming the V-variables to X – such a σ exists since V is finite while X is infinite). Then $e\sigma \in Eq_{\mathsf{K}}(\Sigma, X)$ and by definition of K^* we have that $\mathsf{K}^* \models e\sigma$. But this means that $\mathsf{K}^* \models e$, i.e., $e \in Eq_{\mathsf{K}^*}(\Sigma, Y)$.

(3) Let $A \in \mathsf{K}^*$, and Y be a set of variables such that $|Y| \geq |A|$, and $\alpha : Y \to A$ a surjective assignment. $T_{\Sigma,Y}$ is free for $\mathsf{K}^* \subseteq \mathsf{Alg}(\Sigma)$ (theorem 4.14), and so by theorem 4.13, A is isomorphic to a quotient of $T_{\Sigma,Y}$, i.e., $A \simeq T_{\Sigma,Y/\equiv_{\overline{\alpha}}}$, where $\overline{\alpha} : T_{\Sigma,Y} \to A$ is the unique surjective homomorphic extension of α. Thus, by the first homomorphism theorem, $\overline{\alpha}$ can be factorized through $T_{\Sigma,Y/\equiv_{\overline{\alpha}}}$ as $\overline{\alpha} = \overline{\alpha'} \circ \iota$ where ι is isomorphism (cf. the diagram below).

The kernel of $\overline{\alpha}$ satisfies $\equiv_{\mathsf{K}^*} \subseteq \equiv_{\overline{\alpha}}$: for $A \in \mathsf{K}^*$ and so if $t \equiv_{\mathsf{K}^*} s$ then $t^A_\alpha = s^A_\alpha$ for all $A \in \mathsf{K}^*$ and all α, i.e., $s \equiv_{\overline{\alpha}} t$. So, by the second homomorphism theorem 3.19, $\overline{\alpha'}$ can be factorized through $T_{\Sigma,Y/\equiv_{\mathsf{K}^*}}$ as $\overline{\alpha'} = \omega \circ \epsilon$ where ϵ is surjective.

$$\begin{tikzcd}
T_{\Sigma,Y} \arrow[d, "\omega"'] \arrow[dr, "\overline{\alpha'}"] \arrow[drr, "\overline{\alpha}"] & & \\
T_{\Sigma,Y/\equiv_{\mathsf{K}}} \arrow[r, "\epsilon"'] & T_{\Sigma,Y/\equiv_{\overline{\alpha}}} \arrow[r, leftrightarrow, "\iota"'] & A
\end{tikzcd}$$

Target of ω is $T_{\Sigma,Y}/_{\equiv_{\mathsf{K}}}$ because the identity $T_{\Sigma,Y}/_{\equiv_{\mathsf{K}}} = T_{\Sigma,Y}/_{\equiv_{\mathsf{K}^*}}$ follows from (2) and lemma 6.5: $Eq_{\mathsf{K}}(\Sigma, Y) = Eq_{\mathsf{K}^*}(\Sigma, Y)$ implies $\sim_{\mathsf{K}} = \sim_{\mathsf{K}^*}$, and so $\equiv_{\mathsf{K}} \stackrel{6.5}{=} \sim_{\mathsf{K}} \stackrel{(2)}{=} \sim_{\mathsf{K}^*} \stackrel{6.5}{=} \equiv_{\mathsf{K}^*}$.

K is a variety so corollary 4.24 implies that it has free algebras, in particular, $T_{\Sigma,Y}/_{\equiv_{\mathsf{K}}} \in \mathsf{K}$ (which is free in K by theorem 4.21). Since ϵ, and so $\epsilon \circ \iota$, is surjective, A is a homomorphic image of $T_{\Sigma,Y}/_{\equiv_{\mathsf{K}}} \in \mathsf{K}$. Since K is closed under homomorphic images we obtain that $A \in \mathsf{K}$.

Since A was arbitrary, we have shown $\mathsf{K}^* \subseteq \mathsf{K}$, and so $\mathsf{K} = \mathsf{Alg}(\Sigma, Eq_{\mathsf{K}}(\Sigma, X))$.

Thus we have shown that any variety K can be axiomatized by a set of equations $Eq_{\mathsf{K}}(\Sigma, X)$. Obviously, this set will be infinite: first because X was infinite but, secondly, because typically there will be an infinite number of equations (up to renaming of variables) satisfied by K. Whether, and when, a variety can be axiomatized by a finite number of equations is another, difficult problem which we do not address here.

The strength of this theorem, except that it is a very elegant characterization of equational classes, lies in the implication that if we generalize the language $\mathcal{L}^=$ (for instance to conditional equations, or allowing negation, etc.), then at least one of the varieties' closure properties (under subalgebras, products or homomorphic images) gets lost.

6.3: Applications of Variety Theorem

Variety theorem 4.22 gives us a very powerful tool in determining whether some specific class of algebras can or cannot be axiomatized by means of simple equations. We may try to look for an appropriate set of equational axioms which would determine the required properties. However, if no such axiomatization exists, we might spend quite a long time in this process. Using instead the Variety theorem, it may be very easy to see that the answer is no.

Example 6.11

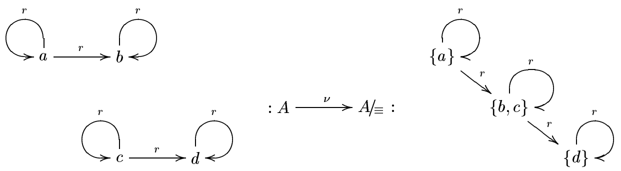
Suppose we want to specify a class of algebras which have exactly two elements in their carriers. Let us consider the signature with one sort and two constants $\Sigma_2 = \langle S; a, b : \to S \rangle$. Can we axiomatize this class using only equations?

Let A be an algebra with two elements a and b in the carrier, and $\equiv$ be the congruence on A induced by $a \equiv b$. Then the quotient $A/_{\equiv}$ will have one elemnt, i.e., such a class would not be closed under homomorphic images. Hence it is not a variety and so is not axiomatizable by mere equations.

Example 6.12 [5.4 continued]

Consider the algebras of preorders over signature from example 5.4. We claimed that they are not axiomatizable by mere equations. We show that the class of preorders is not closed under homomorphic images – Variety theorem implies then that no equational axiomatization exists for this class of algebras.

Let A be a preorder algebra with $A_S = \{a, b, c, d\}$ where $r(a, b)$ and $r(c, d)$ hold. It is obviously a preorder (we have also $r(x, x)$ for all $x \in A_S$). Consider the congruence $b \equiv c$ – the homomorphism $\nu : A \to A/_{\equiv}$ will result in the following



The resulting algebra $A/_{\equiv}$ is not a preorder because the relation r is not transitive. The homomorphism condition implies only the relations indicated in the diagram – but $A/_{\equiv} \not\models r(a, d) = \top$.

Example 6.13

Let Σ_i be the signature with two sorts *Data* and *Labels* and an operation $mark : Data \to Labels$. The intention is to specify a class K of algebras where the *mark*ing function is injective, i.e., where different data elements have different marks.

K can not be specified equationally – it is not a variety, because it is not closed under homomorphic images. Take an algebra A with $A_{Data} = \{0, 1\}$ and $A_{Labels} = \{a, b\}$, and where $mark^A$ can be given by $mark(0) = a$ and $mark^A(1) = b$. $mark^A$ is injective so A belongs to K. The equation $a = b$ gives immediately a congruence on A, so we have a homomorphism $A \to B = A/_{a=b}$. But $mark^B$ is not injective, so $B \notin \mathsf{K}$.

Example 6.14

Let us consider a signature with one sort D(ata) and an if_then_else operation $if : D \times D \times D \times D \to D$ which we want to work according to the definition

$$if(x_1, x_2, x_3, x_4) = \begin{cases} x_3 \text{ if } x_1 = x_2 \\ x_4 \text{ if } x_1 \neq x_2 \end{cases}$$

Such an alegbra clearly exists, for instance, A with $A_{Data} = \mathsf{Z}$, and if defined by the above equation. However, the class of such algebras cannot be defined by simple equations since it is not closed under, for instance, products. In the product $A \times A$, we will for instance have

$$if^{A \times A}(\langle 1,2 \rangle, \langle 1,3 \rangle, \langle 5,6 \rangle, \langle 7,8 \rangle) \stackrel{\text{def}}{=} \langle if^A(1,1,5,7), if^A(2,3,6,8) \rangle = \langle 5,8 \rangle$$

which is not the result $\langle 7,8 \rangle$ prescribed by the definition of if. Since the class isn't closed under products, it isn't a variety and hence no equational axiomatization for this class exists.

Notice the negative character of all these examples: Variety theorem is used in all of them to show that an equational axiomatization does *not* exist. This isn't a coincidence and it is the usual (though not the only) way of using such characterization results. If no equational axiomatization exists, it would be very difficult, if at all possible, to prove it without such a characterization. If, on the other hand, such an axiomatization exists, it will often be easy to find it directly instead of checking all the required closure properties.

Exercises (week 6)

EXERCISE 6.1 Verify that the class D of Σ_i algebras with injective *mark*ing function from example 6.13 is closed under subalgebras and products.

EXERCISE 6.2 Axiomatize finite sets of natural numbers over the following signature Σ_{Set} :

$\mathcal{S}$: N, *Set*
Ω : $0 : \;\to \mathsf{N}$
$\mathsf{s} : \mathsf{N} \to \mathsf{N}$
$\emptyset : \;\to Set$ – empty set
$add : \mathsf{N} \times Set \to Set$ – inserts a new N into set

1. Give an equational axiomatization E_{Set} of the *add* operation (you'll need two axioms expressing, respectively, that a set is unordered and that each element can occur at most once in a set). What will be the initial term algebra T of your specification? What will be a terminal algebra?

2. Add to the signature Σ_{Set} the operation $\cup : Set \times Set \to Set$ (for union of two sets) and axiomatize it in $\mathcal{L}^=$. (Try to write these axioms so that each congruence class in the term algebra will have a representative built exclusively from $\emptyset$ and *add* (and N-terms).)

3. Since $\mathsf{Alg}(\Sigma_{Set}, E_{Set})$ is a variety it is closed under homomorphic images. Let $\approx$ and $\equiv$ be two congruences induced on the initial algebra T from 1. by the respective equations:

$$\mathsf{ss0} \approx \mathsf{sss0} \qquad\qquad add(0, add(1, \emptyset)) \equiv add(2, add(3, \emptyset))$$

What will be the carriers of the quotients $T/_{\approx}$ and of $T/_{\equiv}$?

EXERCISE 6.3 Let Σ_{Str} be the following signature for *Str*ings of symbols from some alphabet S :

$$\begin{array}{lrrl}
\mathcal{S}: & S,\ Str & & \\
\Omega: & \varepsilon: & \to Str & \text{– empty string} \\
 & conc: & Str \times Str \to Str & \text{– concatenates two strings} \\
 & en: & S \to Str & \text{– a string with one element } S \\
 & La: & S \times Str \to Str & \text{– prepends an element on the left of string} \\
 & Ra: & Str \times S \to Str & \text{– appends an element on the right of string}
\end{array}$$

Write first axioms for the *conc*atenation and then define the La and Ra operations using en – all the axioms should be in $\mathcal{L}^{=}$. Does the model class of your specification have initial algebras?

EXERCISE 6.4 Let us consider the following Σ_{Str}-algebra $A : A_S = \{0,1\}$, $A_{Str} = (A_S)^*$ (i.e., all the strings (also the empty one) of symbols 0 and 1), and operations given by: ε^A gives the empty string, $en^A(x) = x$ (i.e., a one-element string 'x'), $conc^A(s,t) \stackrel{\text{def}}{=} st$, $La^A(x,s) \stackrel{\text{def}}{=} xs$ and $Ra^A(s,x) \stackrel{\text{def}}{=} sx$ (where st is the string obtained by concatenating s and t, while sx (resp. xs) is obtained by appending (resp. prepending) the symbol x to the string s.) Following the remark 6.8 and example 6.9, we will verify that A is a free algebra over set $\{0,1\}$ of the specification from exercise 6.3.

1. Verify that A satisfies the axioms E you have written in exercise 6.3, i.e., that $A \in \mathsf{Alg}(\Sigma, E)$.

2. Now, let $X = \{x, y\}$ be the set of two elements (variables) of sort S, and consider the free Σ_{Str}-algebra $T_{E,X}$ (i.e., the quotient $T_{\Sigma_{Str},X}/_{\equiv_E}$ by the congruence $\equiv_E$ induced by your equational axioms E). We assume that in each congruence class $[t] \in (T_{E,X})_{Str}$ (except for $[\varepsilon]$) there is a unique canonical representative $C[t]$ of the form $La(z_1, La(z_2, ...La(z_n, \varepsilon)...))$, where each z_i is either x or y. That is, $[s] \neq [t]$ iff either $C[s]$ and $C[t]$ have different lengths, or else for some i, z_i in $C[s]$ is different from the respective z_i in $C[t]$. (You may try to prove this by structural induction but the proof will be rather elaborate.)

3. Using this assumption, show that $A \simeq T_{E,X}$. Consider the assignment $\alpha : X \to A_S$ given by $\alpha(x) = 0$ and $\alpha(y) = 1$. By theorem 6.7, the algebra $T_{E,X}$ is free in $\mathsf{Alg}(\Sigma, E)$, so, since $A \in \mathsf{Alg}(\Sigma, E)$, the assignment α induces a unique homomorphism $\overline{\alpha} : T_{E,X} \to A$. By the proof of theorem 4.21, this homomorphism will coincide with the evaluation of terms in A under α, i.e., for each $[t] : \overline{\alpha}([t]) = t^A_\alpha$. Showing that this homomorphism is injective and surjective will demonstrate the required isomorphism.

EXERCISE 6.5 In lemma 6.6 and (the proof of) theorem 5.8 we used the relation $\equiv_E$ on $\mathcal{T}(\Sigma, X)$ defined in (5.16) by

$$s \equiv_E t \ \Leftrightarrow \ E \vdash_{EQ} s = t \tag{6.19}$$

where E is a set of equations with variables X and $s, t \in \mathcal{T}(\Sigma, X)$. (In lemma 5.7 we showed that it is a congruence on the algebra $T_{\Sigma,X}$.) Given such a set of equations, we can, instead, define the congruence $\equiv_{\overline{E}}$ on $T_{\Sigma,X}$ using the definition 2.29 (cf. example 2.30). I.e., as the basis relation we take the set $\overline{E}$ of all pairs (of terms) obtained by performing all substitutions to the equations from E :

$$\overline{E} \stackrel{\text{def}}{=} \bigcup_{\sigma : X \to \mathcal{T}(\Sigma,X)} \{\langle s\sigma, t\sigma\rangle : s = t \in E\}$$

and then taking the minimal congruence on $T_{\Sigma,X}$ containing this relation, i.e.:

$$\equiv_{\overline{E}} \stackrel{\text{def}}{=} \bigcap \{\equiv \, : \ \overline{E} \subseteq \, \equiv \ \wedge \ \equiv \ \text{ is a congruence on } T_{\Sigma,X}\} \tag{6.20}$$

Show that (6.19) and (6.20) actually define the same relation, i.e., that $\equiv_E \, = \, \equiv_{\overline{E}}$.

1. First show that $\overline{E} \subseteq \, \equiv_E$ and use this fact to conclude that $\equiv_{\overline{E}} \ \subseteq \, \equiv_E$.

2. For the opposite inclusion, use the fact that $\overline{E} \subseteq \, \equiv_{\overline{E}}$ to show first that $T_{\Sigma,X}/_{\equiv_{\overline{E}}} \in \mathsf{Alg}(\Sigma, E)$. Then use theorem 5.8 (and the fact (5.17)) to conclude that $\equiv_E \, \subseteq \, \equiv_{\overline{E}}$. (Here you will use the fact that, for any congruence $\equiv$ on $T_{\Sigma,X}$ we have $T_{\Sigma,X}/_{\equiv} \models s = t \ \Leftrightarrow s \equiv t$.)

Week 7: Closure Properties of Other Languages

- Conditional Equations
- Horn Equational Formulae
- Clauses

We will now study the closure properties of the more general equational language. We will move along the hierarchy from table 1 on page 44 and see, how moving to a more expressive language, single closure properties get lost.

7.1: Conditional Equations : $\mathcal{L}^{=}_{\Rightarrow}$

Recall that formulae of $\mathcal{L}^{=}_{\Rightarrow}$ are of the form $s_1 = t_1 \wedge ... \wedge s_n = t_n \Rightarrow s = t$, for $n \geq 0$. Generalizing the language $\mathcal{L}^{=}$ to conditional equations we lose closure under homomorphic images.

Lemma 7.1 $\mathcal{L}^{=}_{\Rightarrow}$ is not closed under homomorphic images.

Proof. We show it by a simple counterexample. Let Φ contain one conditional equation $s = t \Rightarrow p = r$, and A be such that $A \models s \neq t$, and $A \models p \neq r$. Obviously $A \in \mathsf{Alg}(\Phi)$. Let $\equiv$ be the congruence on A given by $s \equiv t$. Then $A/_{\equiv} \models s = t$ but $A/_{\equiv} \not\models p = r$. Thus, although $A/_{\equiv}$ is a homomorphic image of A it is not a model of Φ.

Example 7.2 [2.2 continued]

Recall example 2.2. We used there the signature Σ :

$$\begin{array}{rrcl}
\mathcal{S}: & \underline{\mathsf{N}}, \underline{\mathsf{B}}; & & \\
\Omega: \top, \bot: & & \rightarrow & \underline{\mathsf{B}} \\
\underline{0}: & & \rightarrow & \underline{\mathsf{N}} \\
\underline{\mathsf{s}}: & \underline{\mathsf{N}} & \rightarrow & \underline{\mathsf{N}} \\
\underline{leq}: & \underline{\mathsf{N}} \times \underline{\mathsf{N}} & \rightarrow & \underline{\mathsf{B}} \\
\underline{eq}: & \underline{\mathsf{N}} \times \underline{\mathsf{N}} & \rightarrow & \underline{\mathsf{B}}
\end{array}$$

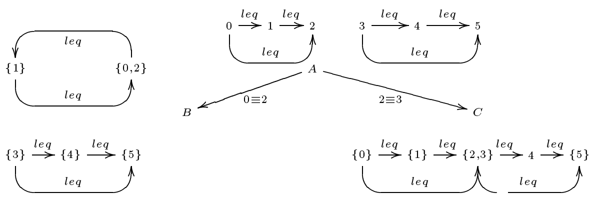
and the algebra $N = \langle \mathsf{N}, \mathsf{B}; \mathtt{tt}, \mathtt{ff}, 0, \mathsf{s}, leq, eq \rangle$ where $leq^N(x, y) = \mathtt{tt} \Leftrightarrow x \leq y$.

We saw that the total order leq ceased to be total in the product algebra $N \times N$ and became a partial order (this fact is commented in example 7.14). The product algebra $N \times N$ can be seen as a Σ-algebra satisfying the partial order axioms:

$$\begin{array}{lrcl}
1. & leq(x,x) & = & \top \\
2. & leq(x,y) = \top \wedge leq(y,z) = \top & \Rightarrow & leq(x,z) = \top \\
3. & leq(x,y) = \top \wedge leq(y,x) = \top & \Rightarrow & eq(x,y) = \top
\end{array} \tag{7.21}$$

The last two axioms are conditional equations and, as stated in lemma 7.1, need not be preserved under homomorphis images.

To simplify the example, let's take a smaller PO algebra A (satisfying (7.21)) with the carrier $\{0, 1, 2, 3, 4, 5\}$ instead of N and with with $\underline{eq}^A$ being the identity on A_{N} and $\underline{leq}^A(0, 1) = \underline{leq}^A(1, 2) = \mathtt{tt}$ and $\underline{leq}^A(3, 4) = \underline{leq}^A(4, 5) = \mathtt{tt}$. (Axiom 2 forces then $\underline{leq}^A(0, 2)$ and $\underline{leq}^A(3, 5)$. We do not have any axioms for $\underline{\mathsf{s}}$, so we may take it to be identity on A_{N}.)



Let B be the quotient of A by the congruence $0 \equiv 2$. We then get the equivalence class $[0] = [2] = \{0, 2\}$ as an element in the carrier of B, with $\underline{leq}^B(\{0, 2\}, 1) = \mathtt{tt}$ and $\underline{leq}^B(1, \{0, 2\}) = \mathtt{tt}$ but with $\{0, 2\} \neq 1$, i.e., $\underline{eq}^B(\{0, 2\}, 1) \neq \mathtt{tt}$ and so B does not satisfy the third axiom.

Similarly, if we let C be the quotient of A by the congruence $2 \equiv 3$, the resulting algebra will not satisfy the second axiom. We will have $\underline{leq}^C(1, \{2, 3\}) = \mathtt{tt}$ and $\underline{leq}^C(\{2, 3\}, 4) = \mathtt{tt}$ but not $\underline{leq}^C(1, 4) = \mathtt{tt}$.

In fact, there is a general theorem (by Lyndon) that only positive formulae are preserved by homomorphisms, i.e., if e is a positive formula, $A \models e$ and B is a homomorphic image of A, then also $B \models e$. (Positive formulae are those built without use of negation, in our case, only simple equations and their conjunctions or disjunctions.) Since a conditional equation $e_1 \wedge ... \wedge e_n \Rightarrow e$ is equivalent to $\neg e_1 \vee ... \vee \neg e_n \vee e$, it follows that it is not, in general, preserved under homomorphic images.

All the other properties of varieties remain valid for conditional equational classes.

Theorem 7.3 $\mathcal{L}^{=}_{\Rightarrow}$ is closed under isomorphisms, subalgebras and products.

Proof. Isomorphisms and subalgebras are easy and left as exercise. Closure under non-empty products follows from lemma 6.2. Let $\{A_i\}_{i \in I} \subseteq \mathsf{K}$ where $\mathsf{K} = \mathsf{Alg}(\Sigma, C)$ with C a set of conditional equations. Let $c : t_1 = s_1 \wedge ... \wedge t_n = s_n \Rightarrow t = s$ be any of the formulae in C. Let $\alpha : X \to \Pi A$ be any assignemnt and assume that for $1 \leq j \leq n : \overline{\alpha}(t_j) = \overline{\alpha}(s_j)$. (Otherwise, we get immediately $\Pi A \models c$ under α.) By lemma 6.2 in all A_i we will have $\overline{\alpha \circ \pi_i}(t_j) = \overline{\alpha \circ \pi_i}(s_j)$ and so all $\overline{\alpha \circ \pi_i}(t) = \overline{\alpha \circ \pi_i}(s)$ since $A_i \in \mathsf{K}$. So by lemma 6.2 $\overline{\alpha}(t) = \overline{\alpha}(s)$. Since α was arbitrary, it follows that $\Pi A \models c$.

Empty product – a terminal unit algebra Z – satisfies all equations, in particular, all the equations occurring in the right hand side of $\Rightarrow$ in all $c \in C$. Thus $Z \models C$ and so $Z \in \mathsf{K}$.

Using theorem 4.22 the lema yields.

Corollary 7.4 Let C be a set of conditional Σ-equations and $\mathsf{K} = \mathsf{Alg}(\Sigma, C)$. Then K contains free algebras and, if Σ is non-void, initial algebras.

There exists an analogous characterization of conditional classes as the Variety theorem 4.22 for the equational classes. It requires however the construction of so called *ultraproducts* which we do not introduce here.

Example 7.5 [6.11 continued]

Example 6.11 showed that the class of algebras with exactly two element carriers cannot be axiomatized by mere equations. Will conditional equations do? Let A be an algebra with two elements in the carrier. Then $A \times A$ will have four elements, i.e., such a class would not be closed under products. Since, by theorem 7.3 any conditional equational class must be closed under products, we see that $\mathcal{L}^{=}_{\Rightarrow}$ is not sufficient to axiomatize this class.

Example 7.6 [6.13 continued]

The class of Σ_i-algebras with injective marking function did not have an equational axiomatization – it wasn't closed under homomorphic images.

In order to see whether D can be axiomatized in $\mathcal{L}^{=}_{\Rightarrow}$ we should check if it is closed under products and subalgebras. (In fact, it is, as we saw in exercise 6.1). But here it may be easier to just observe that the required property of injectivity is expressed by the standard conditional equation: $mark(x) = mark(y) \Rightarrow x = y$.

Example 7.7 [6.14 continued]

Algebras with one sort and if_then_else were not closed under (non-empty) products, hence they cannot be axiomatized by conditional equations.

7.2: Equational Horn Formulae : $\mathcal{L}^=_H$

Formulae of $\mathcal{L}^=_H$ are either conditional equations $s_1 \neq t_1 \vee ... \vee s_n \neq t_n \vee s = t$, for $n \geq 0$, or disjunctions of inequations $s_1 \neq t_1 \vee ... \vee s_n \neq t_n$, for $n \geq 1$. Since $\mathcal{L}^=_\Rightarrow \subset \mathcal{L}^=_H$, lemma 7.1 implies that $\mathcal{L}^=_H$ is not closed under homomorphic images. But there is also another price we have to pay when moving from $\mathcal{L}^=_\Rightarrow$ to $\mathcal{L}^=_H$.

The classes we have considered so far were guaranteed to be non-empty. Any equational or conditional equational class was closed under empty products, i.e., it contained at least the unit algebra. This reflects the fact that the languages $\mathcal{L}^=$ and $\mathcal{L}^=_\Rightarrow$ are not capable of expressing negation, and hence, of introducing a possibly inconsistent set of axioms. An obvious example would be the set $\{s = t, s \neq t\}$ which certainly does not have any model. But $s \neq t$ is not expressible in $\mathcal{L}^=_\Rightarrow$. Horn formulae are the first on our way where such inconsistencies may occur. $s \neq t$ is a Horn equation(al formula) which is neither conditional nor simply equational. This possibility of writing inconsistent axioms corresponds to the possibility of having empty model class. Thus, since existence of empty products guarantees non-emptiness of the model class, the following lemma should not be surprising.

Lemma 7.8 $\mathcal{L}^=_H$ is not closed under empty products.

Proof. Since empty product is the same as terminal object, we show that we may have a situation where no terminal object exists in the model class.

Let Φ contain one disjunction $a_1 \neq a_2 \vee b_1 \neq b_2$, for $a_1, a_2, b_1, b_2 \in \mathcal{T}(\Sigma)$. Let A, B be such that $A \models a_1 = a_2$, $A \models b_1 \neq b_2$, and $B \models a_1 \neq a_2$, $B \models b_1 = b_2$. Obviously $A, B \in \mathsf{Alg}(\Phi)$. If Z is terminal in $\mathsf{Alg}(\Phi)$ then there exist unique homomrphisms $\omega_A : A \to Z$ and $\omega_B : B \to Z$. We then must have that $a_1^Z = \omega_A(a_1^A) = \omega_A(a_2^A) = a_2^Z$, and similarly, $b_1^Z = \omega_B(b_1^B) = \omega_B(b_2^B) = b_2^Z$. But this means that $Z \models a_1 = a_2$ and $Z \models b_1 = b_2$, so $Z \notin \mathsf{Alg}(\Phi)$.

Another simple counterexample would be the set of axioms $\{s = t, s \neq t\}$. Its model class is empty and hence it does not contain empty products. In fact, the existence of unit algebras is what separates Horn classes from conditional equational classes.

Lemma 7.9 Assume that $\mathsf{K} \subseteq \mathsf{Alg}(\Sigma)$ is an equational Horn class. Then K is a conditional equational class iff it contains a unit algebra.

Proof. $\Rightarrow$) follows from theorem 7.3.

$\Leftarrow$) Since K is Horn, it is axiomatized by a set H of Horn formulae. The only formulae which are Horn but not conditional are of the form $h : \neg(t_1 = s_1) \vee ... \vee \neg(t_n = s_n)$. So suppose that such a formula belongs to H. But a unit algebra Z will satisfy all the equations $Z \models t_i = s_i$, i.e., $Z \not\models h$, so if $Z \in \mathsf{K}$, no such h may be among the axioms H.

This lemma is *not* a general characterization of the conditional equational classes, i.e., it does not say that a class is conditional equational iff it contains unit algebras (and, perhaps, satisfies the conditions of theorem 7.3). It says when a *Horn equational class* is also a conditional class. If we wanted to achieve a full characterization of conditional classes, we could now look for such a characterization of Horn classes and add to its properties the existence of unit algebras. However such a characterization of Horn classes requires some other semantic constructions which we have not introduced (*reduced products*) and therefore we won't attempt to do it here. The model classes of equational Horn formulae are called *quasivarieties*. We only show that except for empty products, quasivarieties are closed under the same constructions as conditional classes.

Theorem 7.10 $\mathcal{L}^=_H$ is closed under subalgebras, isomorphisms and non-empty products.

Proof. Again, subalgebras and isomorphisms are left as exercise. Let $\mathsf{K} = \mathsf{Alg}(\Sigma, H)$ where H is a set of Horn equational formulae, and let $\{A_i\}_{i \in I} \subseteq K$ be non-empty subclass of

K. We have to show that $\Pi A \models H$. From theorem 7.3 we know that ΠA will model all the conditional equations from H, so we have to show it only for those $h \in H$ which are disjunctions of negated equations, i.e., of the form $h : \neg(t_1 = s_1) \vee ... \vee \neg(t_n = s_n)$.

This follows again from lemma 6.2. For any $\alpha : X \to \Pi A$, we have that $\forall i \in I\ \exists 1 \leq j \leq n$: $\overline{\alpha \circ \pi_i}(t_j) \neq \overline{\alpha \circ \pi_i}(s_j)$. Then lemma 6.2 implies that $\overline{\alpha}(t_j) \neq \overline{\alpha}(s_j)$ i.e., ΠA satisfies h under assignment α. Since α was arbitrary, this means that $\Pi A \models h$.

Example 7.11 [7.5 continued]

Example 7.5 showed that the class of Σ_2-algebras with exactly two elements in the carrier wasn't closed under non-empty products. Hence even Horn formulae will not be sufficient to specify this class.

Similarly, the class of one sorted algebras with if_then_else from example 6.14 wasn't closed under non-empty products, and hence cannot be axiomatized by Horn formuale.

Corollary 7.12 Let $\mathsf{K} \subseteq \mathsf{Alg}(\Sigma)$ be a non-empty Horn class. It contains free algebras and, if Σ is non-void, initial objects.

Notice the qualification "be a non-empty" which did not occur in earlier versions of this statement (for $\mathcal{L}^=$ or $\mathcal{L}^=_\Rightarrow$, e.g., in corollary 7.4). This qualification is necessary here because a set of Horn formulae can be inconsistent. Then, obviously, the model class won't possess free or initial objects. We say that $\mathcal{L}^=_H$ *admits* free (initial) objects meaning exactly this: if the axiom set is consistent then the model class has free algebras.

7.3: Equational Clauses : $\mathcal{L}^=_\forall$

Clauses of $\mathcal{L}^=_\forall$ are arbitrary disjunctions $s_1 \neq t_1 \vee ... \vee s_n \neq t_n \vee s'_1 = t'_1 \vee ... \vee s'_p = t'_p$, for $n \geq 0$ and $p \geq 0$. Having extended the language to $\mathcal{L}^=_H$ we are left with model classes which are only guaranteed to be closed under isomorphisms, subalgebras and non-empty products. Moving now to the language $\mathcal{L}^=_\forall$ of clauses, we lose this last property.

Lemma 7.13 $\mathcal{L}^=_\forall$ is not closed under non-empty products and does not admit initial objects.

Proof. Let Φ contain one disjunctive equation $a_1 = a_2 \vee b_1 = b_2$, and A, B be such that $A \models a_1 = a_2$, $A \models b_1 \neq b_2$, and $B \models a_1 \neq a_2$, $B \models b_1 = b_2$. Obviously $A, B \in \mathsf{Alg}(\Phi)$. However, satisfaction of the disjunction in their product $A \times B$ would require that $\langle a_1^A; a_1^B \rangle = \langle a_2^A; a_2^B \rangle \vee \langle b_1^A; b_1^B \rangle = \langle b_2^A; b_2^B \rangle$. The first disjunct does not hold since $a_1^B \neq a_2^B$ and the second one does not hold because $b_1^A \neq b_2^A$. Thus $A \times B \notin \mathsf{Alg}(\Phi)$.

The same counter-example shows the possibility of non-existence of inital objects. If all $a_1, a_2, b_1, b_2 \in \mathcal{T}(\Sigma)$ and I is an initial object, it must satisfy at least one of the equations. If it satisfies $a_1 = a_2$ then there is no homomorphism $\omega_B : I \to B$ and if it satisfies $b_1 = b_2$ then there is no homomorphism $\omega_A : I \to A$.

Since existence of free objects implies existence of initial objects, this leamma implies that $\mathcal{L}^=_\forall$ does not admit free objects either. By lemmata 7.1 and 7.8, $\mathcal{L}^=_\forall$ isn't closed under homomorphic images or empty products.

Example 7.14 [7.2 continued]

The order leq on N (i.e., $\leq$ on N) from example 2.2 is total, i.e., in addition to the PO axioms (7.21) from example 7.2, it satisfies the axiom

$$4.\ leq(x, y) = \top \ \vee\ leq(y, x) = \top \tag{7.22}$$

This disjunctive formula is not equivalent to any simpler (conditional or single) equational formula and so, by lemma 7.13, need not be preserved under (non-empty) products. We saw this in example 2.2 where the algebra N satisfied this axiom, while the product $N \times N$ did not.

Theorem 7.15 $\mathcal{L}^{=}_{\forall}$ is closed under subalgebras and isomorphism.

PROOF. In spite of its simplicity, this is a classical theorem (of Łoś and Tarski). Its original formulation says that the model class of any *FOL* theory which uses only universally quantified formulae is closed under subalgebras and isomorphisms. Isomorphisms are obvious, so let $h(x_1,\ldots,x_n)$ be any opened formula with only the free variables $X = \{x_1,\ldots,x_n\}$, and suppose $A \models \forall x_1,\ldots,x_n : h(x_1,\ldots,x_n)$. This means that for all assignments $\alpha : X \to |A|$, i.e., for all elements $a_1,\ldots,a_n \in |A|$, $h^A(a_1,\ldots,a_n)$ is satisfied. If now $B \sqsubseteq A$, then $|B| \subseteq |A|$ and interpretation of all symbols in B agrees with the interpretation in A. Thus for any $b_1,\ldots,b_n \in |B| \subseteq |A|$ we will have $h^B(b_1,\ldots,b_n) = h^A(b_1,\ldots,b_n)$, which will be satisfied by assumption.

Example 7.16 [7.11 continued]
The class of two-element Σ_2-algebras was not axiomatizable in any of the languages weaker than $\mathcal{L}^{=}_{\forall}$. The first language (in our hierarchy) which does not imply closure under non-empty products was $\mathcal{L}^{=}_{\forall}$. As a matter of fact, the following axiomatization will yield the required class of models:

$$\begin{array}{rcl} a & \neq & b \\ x = a & \vee & x = b \end{array}$$

Example 7.17 [7.7 continued]
Similar trick will solve the problem with axiomatizing the *if_then_else* function. With Horn formulae (and even with the conditional equations) we could express only the first, but not the second, of the following axioms:

$$\begin{array}{rcl} x_1 = x_2 & \Rightarrow & if(x_1,x_2,x_3,x_4) = x_3 \\ x_1 = x_2 & \vee & if(x_1,x_2,x_3,x_4) = x_4 \\ (\text{i.e., } x_1 \neq x_2 & \Rightarrow & if(x_1,x_2,x_3,x_4) = x_4) \end{array}$$

Notice a subtle point concerning the axiomatization of such an *if_then_else*. If the first argument is of a Boolean sort, for which we have two constants $\top$ and $\bot$, we can axiomatize it using simple equations:

$$\begin{array}{rcl} if(\top,x_3,x_4) & = & x_3 \\ if(\bot,x_3,x_4) & = & x_4 \end{array}$$

This, however, presupposes that our Boolean sort has *exactly two elements* interpreting these two constants. As example 7.16 showed, axiomatization of such a sort requires $\mathcal{L}^{=}_{\forall}$. (Unless we can use initial semantics, in which case, the constants will be the only two distinct elements, provided that the axioms ensure equality of any other Boolean term with exactly one of them.)

7.4: Summary

The table 2 contains the summary of the model theoretic results from this section. The places marked with double symbols indicate the actual lemmata we have proved – other cases follow from these results (negative downwards and positive upwards). The intuitive content of these relations can be summarized thus. The more specific properties we want a class of algebras to have, the more expressive language we need to axiomatize them. By "the more specific properties" one should not understand "the more closure properties". Rather, we mean that greater expressiveness of a language allows us to exclude or include models according to more specific criteria. In order to define the class of all Σ-algebras we do not actually need any language (except the signature) – and the poverty of this language is reflected in the richness of the closure properties posessed by $\mathsf{Alg}(\Sigma)$. If we use merely simple equations, the corresponding model-classes will have all the closure proerties we have defined. These restricted languages do not provide the means for, for instance, including only some models and their products but excluding homomorphic images. But

	closure of $\mathsf{Alg}(\Phi)$ under						
			products		hom.	admits objects	
Φ	iso	subalg	empty	non-empty	images	free	initial
$\mathcal{L}^{=}$	+	++	++	++	++	+	+
$\mathcal{L}^{=}_{\Rightarrow}$	+	+	+	+	=	+	+
$\mathcal{L}^{=}_{H}$	+	+	=	+	–	+	+
$\mathcal{L}^{=}_{\forall}$	+	+	–	=	–	–	=

Table 2: Languages and Closure Properties

a more expressive language, for instance, one allowing us to express implication, will enable us to include products but exclude homomorphic images. A more expresive language provides more sophisticated means for "distinguishing" structures which "cannot be distinguished", and hence excluded, by a less expressive language. In short – the more expressive language, the more freedom in manipulating the model classes and, hence, the fewer guaranteed closure properties.

Exercises (week 7)

EXERCISE 7.1 We will specify a variant of infinite arrays with *co*unters where the *ins*ert operation places the new element at the end of the array increasing the value of the counter by 1. We assume given an $\mathcal{L}^{=}$-specification of natural numbers; the signature will be:

$$\begin{array}{rrll} \mathcal{S}: & \mathsf{N}, A, A^{\mathsf{N}} & & \\ null: & \to A^{\mathsf{N}} & & \text{– an empty array with 0 counter} \\ co: & A^{\mathsf{N}} \to \mathsf{N} & & \text{– counter is the last occupied index} \\ ins: & A \times A^{\mathsf{N}} \to A^{\mathsf{N}} & & \text{– increases the counter and places the new element there} \\ ev: & A^{\mathsf{N}} \times \mathsf{N} \to A & & \text{– returns the element stored at the index} \end{array}$$

(We ignore the question of what should be the value of $ev(L,i)$ when no element has been stored in L at the index i.) Write equational axioms (in $\mathcal{L}^{=}$) for the operation *co*, and then axiomatize the other operations.

In the axiomatization of *ev* you should have needed a more general language than $\mathcal{L}^{=}$ – which one? Is it closed under empty products? Does the unit algebra belong to the model class of your axioms? If the first answer is no and the second yes – how do you explain this?

EXERCISE 7.2 Recall exercises 2.1, 4.1 and 4.2. Let Σ_3 be the signature with one sort symbol and three constant symbols, i.e., $\Sigma_3 = \langle D; a, b, c : \to D\rangle$.

1. Write the (set of) axioms E_4 (in the appropriate language) such that the class of algebras $\mathsf{K}_4 = \mathsf{Alg}(\Sigma_3, E_4)$ contains all and only Σ_3-algebras with at most three elements in the carrier.
2. Verify that the class K_4 is closed under homomorphic images. Now, E_4 certainly isn't purely equational, $E_4 \not\subseteq \mathcal{L}^{=}$ – if it is you did something wrong! To which among the languages from table 2 does E_4 belong? How would you explain the apparent contradictiction that this language is not closed under homomorphic images while K_4 is?
3. Write the (set of) axioms E_3 (possibly, in a more general language) such that the class of algebras $\mathsf{K}_3 = \mathsf{Alg}(\Sigma_3, E_3)$ contains all and only Σ_3-algebras with exactly three elements in the carrier. (Obviously, we have that $\mathsf{K}_3 \subset \mathsf{K}_4$, so you should also have $E_3 \supset E_4$, i.e., axiom(s) E_4 are among the axioms E_3.)
4. Is K_4 closed under homomorphic images? Either prove that it is or provide a counter-example.

<u>EXERCISE 7.3</u> A directed multigraph is a directed graph where there may be more than one edge between any pair of nodes. Such a multigraph can be described as an algebra with two sorts: N(odes) and E(dges), and two operations $s, t : E \to N$ which for each edge return, respectively, its s(ource) and t(arget) node.

1. Write the signature Σ_G and, if necessary axioms, which would define the class of multigraph algebras. Which closure properties does the model class of your axiomatization possess? Write the homomorphism condition (3.10) for your signature Σ_G, and give an example of two multigraphs with a Σ_G-homomorphism between them. Does the class have initial models?
2. A directed graph is a special case of a multigraph where there is at most one edge between any two nodes. Add the necessary axioms to your definition of multigraphs in order to obtain the class of graph algebras.
   Most probably you have lost some of the closure properties – say which, and give example(s) illustrating that the class of graphs does not posses such property(ies).
3. The graphs you have defined so far should allow self-loops, i.e., edges with source and target being the same node. Extend your graph definition with the necessary axioms excluding such edges from the graphs. What closure properties will this class have?

Week 8: Programs and Abstract Data Types

- PROGRAMS ARE ALGEBRAS
- SPECIFICATION DETERMINES A CLASS OF ALGEBRAS
- CONSTRUCTOR-SPECIFICATIONS

8.1: PROGRAMS ARE ALGEBRAS

We have seen in examples 4.7, 4.15 that semantics of a programming language determined by a signature Π, can be given by means of a homomorphism from the term algebra T_Π to the appropriate Π-algebra representing the actual semantics of the language. The following example gives a more detailed idea of what kind of an algebra might be appropriate as the semantics of a simple imperative language.

Example 8.1 [Algebraic Semantics of a Programming Language]
Let us start with the assumption that we have a set $Atoms = \{a_1, \ldots, a_n\}$ of atomic programming constructs (like assignment `x:=5`, `skip`, etc.) and a set $Test$ of tests (like `y=5`, `z<100`, etc.). We define the syntax of a simple programming language by the following signature:

$$\begin{array}{lrrl} \Sigma_P = & \mathcal{S}: & \multicolumn{2}{l}{Test, Prog} \\ & \Omega: & a_1, \ldots, a_n: & \to Prog \\ & & \_;\_: & Prog \times Prog \to Prog \\ & & \texttt{if}\ \_\ \texttt{then}\ \_\ \texttt{else}\ \_\ \texttt{fi}: & Test \times Prog \times Prog \to Prog \\ & & \texttt{while}\ \_\ \texttt{do}\ \_\ \texttt{od}: & Test \times Prog \to Prog \end{array}$$

Any program in this language, will be a Σ_P ground term. Semantics of this language is defined as a particular Σ_P-algebra A, and semantics of all programs will be the meaning of ground terms in A.

For Σ_P-algebras we have various choices but we will try to make the example as simple as possible. The intended meaning of a program is a state transformation, where a state is just a state of the store. Think of the store as an array indexed by the program variables with some actual values, e.g., as $S : X \to \mathsf{Z}$. A program simply changes the values stored at various locations. For instance, the assignment `x:=5` amounts to changing the state so that evaluation of `x` returns `5`. Thus the meaning of a program will be a function $f : S \to S$, where S is the set of states. In each state we can evaluate the variables and, in particular, we can test some conditions by evaluating the state to a Boolean value. I.e., the set T of tests is the set of functions $b : S \to \{\mathtt{tt}, \mathtt{ff}\}$.

So, let A be some Σ_P algebra with fixed sets S, $P : S \to S$ and $T : S \to \{\mathtt{tt}, \mathtt{ff}\}$, and with some fixed interpretation of the atomic constants $Atoms$. The meaning of the three operations from the signature will be given with the help of the three functions in A, namely, $sq : P \times P \to P$ (sequencing), $cd : T \times P \times P \to P$ (conditional) and $it : T \times P \to P$ (iterator). All pe and be are program- and test-expressions over Σ_P, $p \in P$ are programs and $s \in S$ is an arbitrary state:

Σ_P ground term		meaning in A	with semantic definition
$pe_1; pe_2$	$\mapsto$	$sq(pe_1^A, pe_2^A)$	$sq(p_1, p_2)(s) \stackrel{\text{def}}{=} p_2(p_1(s))$
`if` be `then` pe_1 `else` pe_2 `fi`	$\mapsto$	$cd(be^A, pe_1^A, pe_2^A)$	$cd(b, p_1, p_2)(s) \stackrel{\text{def}}{=} \begin{cases} p_1(s) & \text{if } b(s) = \mathtt{tt} \\ p_2(s) & \text{if } b(s) = \mathtt{ff} \end{cases}$
`while` be `do` pe `od`	$\mapsto$	$it(be^A, pe^A)$	$it(b, p)(s) \stackrel{\text{def}}{=} cd(b, sq(p; it(b, p)), id_S)(s)$

Notice that in the definition of the semantics of `while` we have used id_S – the identity transformation on the set of states. It is not certain whether there is a corresponding atomic operation in the actual langugae (`skip` or `nop`). This definition says the following: the meaning of `while` b `do` p `od` is a state transformer of the form $cd(b, R, id_S)$. When applied to any state s, it first evaluates $b(s)$ – if the result is $\mathtt{ff}$ the whole evaluation returns $id_S(s)$, that is s. If $b(s) = \mathtt{tt}$, then one evaluates $R = sq(p, it(b, p))$, i.e., evaluates $p(s) = s'$ and continues with $it(b, p)(s')$.

As any programmer knows, a `while`-loop may not terminate – if our algebra A is to be total, we have to equip it with some element of the sort S which would represent the "state" resulting from such a non-terminating computation.

We will not spend time on more detailed examples of how one can define actual atomic programs, test expressions, semantics of declarations, etc. (cf. examples 1.30, 4.7, exercise 1.3). The example illustrates the main intuition of the slogan "programs as algebras". An algebra for a programming language will typically be generated. Nevertheless, this is not a strict requirement – one may introduce operations in the algebra, like the id_S transformation, for which there are no corresponding constructs in the syntax of the language. Also, the language itself will often correspond only to some high-level operations, thus representing an abstraction from many low-level operations which have to be implemented in the actual algebra (as we saw in example 4.10).

Furthermore, while not all algebras are *executable*, the ones for programming languages certainly have to be. But even then the general algebraic perspective offers invaluable abstraction mechanisms. What we did not consider before was the possibility of looking at such an algebraic semantics from, so to speak, different levels of abstraction.

Example 8.2 [Algebra for a program]
For instance, assume that the following piece of code is written in some programming language Π:

```
integer array tab(1:20);
procedure min(t:integer array, f,l:integer):integer;
begin
  integer i, m;
   m:=f;
  for i:= f step 1 until l do
    if t(i)<t(m) then m:= i;
  return m;
end;
```

It can be looked at as defining a very concrete, executable algebra given by the detailed semantics of the programming language. But we may also view it as an algebra $[\![P]\!]$ over the following signature ($\mathsf{Z}^{\lceil \mathsf{N} \rceil}$ is the sort of bounded arrays of integers – cf. example 1.29)

$$\begin{array}{lrrl} \Sigma = & \mathcal{S}: & \multicolumn{2}{l}{\mathsf{Z}^\bullet, \mathsf{Z}^{\lceil \mathsf{N} \rceil}, \mathsf{B}} \\ & \Omega: & tab: & \to \mathsf{Z}^{\lceil \mathsf{N} \rceil} \\ & & min: \ \mathsf{Z}^{\lceil \mathsf{N} \rceil} \times \mathsf{Z}^\bullet \times \mathsf{Z}^\bullet & \to \mathsf{Z}^\bullet \\ & & \top, \bot: & \to \mathsf{B} \\ & & \leq: \ \mathsf{Z}^\bullet \times \mathsf{Z}^\bullet & \to \mathsf{B} \end{array}$$

Observe that min or tab need not be the operations in Π – they are only defined using Π. Thus, we do not have $\Sigma \subseteq \Pi$. Nevertheless, the semantics of $[\![P]\!]$ is obviously determined by the semantics of the language Π. But the meaning of P can (and should) be expressed in terms of the formulae over its own signature Σ. For instance, we would expect that $[\![P]\!]$ satisfies the following formula:

$$f \geq 1 = \top \wedge l \leq 20 = \top \wedge f \leq i = \top \wedge i \leq l = \top \ \Rightarrow\ min(tab, f, l) \leq tab(i) = \top.$$

The signature Σ does not correspond to all the details occurring within the declaration of the **procedure** `min`. (In this narrow context, tab is a constant – the signature $\mathsf{Z} \times \mathsf{Z} \to \mathsf{Z}^{\lceil \mathsf{N} \rceil}$ would be the signature for a function modelling array declarations.) Nevertheless, Σ is – or may be seen as – the signature of this program's algebra, since it is the interface of the program for its user. To cover all the details, we would also need, for instance, a signature corresponding to this from example 1.29 with the operations for creating, reading and updating arrays.

(Notice again that approaching the actual program code we almost inevitably have to deal with partiality of operations: we have finite arrays (example 1.10), and `min`, when given `f, l` which are outside the array bounds will have to react appropriately (or inappropriately).)

Thus we do not have to consider all the details of the actual programming language algebra – we may view it at a more convenient, more abstract level. The signature which we gave for $[\![P]\!]$ was only a small part of all the detailed operations which had to be involved in the implementation of the program P. What we did, was to take the reduct of the actual algebra with respect to the operations which interested us. (The same was done in example 4.10 – the algebra of binary words implemented on the machine M had a lot of operations which were not present in the algebra of the programming language Π.) This abstraction from the low-level details is one of the main strength of the algebraic approach to program development.

8.2: Algebraic Program Development

Since programs are algebras we can use various means of describing and manipulating algebras for the purpose of program specification and development. A set of axioms determines its model-class which, roughly speaking, may be viewed as the class of possible, also programmable, realizations of these axioms. Thus a pair $\langle \Sigma; \Phi \rangle$, where Φ is a set of axioms in an appropriate language is the basic form of a *specification*. Initially, these axioms will state only the most general and abstract requirements identifying the desired properties of the intended system rather than its behaviour and operational description ("what" rather than "how"). *Program development* amounts then to gradual refinement of these initial requirements towards a more and more concrete and, eventually, executable description. The main idea of program development from algebraic specifications is to include, at each level of such a refinement process, only the necessary information defining the properties of interest. Transition from one level to the next amounts to *refining* the specification by adding new axioms or augmenting the signature with additional operations. That is, starting with a very large, loosely described class of algebras – a *requirement specification* – one makes more and more specific design decisions arriving eventually at a concrete program which, from the very begining was one of the algebras in the model class of the initial specification:

$$\begin{array}{ccccccccc} SP_0 & \longrightarrow & SP_1 & \longrightarrow & \ldots & \longrightarrow & SP_n & \longrightarrow & Prog \\ Alg(SP_0) & \supseteq & Alg(SP_1) & \supseteq & \ldots & \supseteq & Alg(SP_n) & \ni & [\![Prog]\!] \end{array} \tag{8.23}$$

The model class inclusion is the basic algebraic notion of *implementation*: a specification SP_1 implements SP_0, $SP_0 \longrightarrow SP_1$, iff $Alg(SP_0) \supseteq Alg(SP_1)$.

Example 8.3 [Implementation by model-class inclusion]

Assuming given a specification Int, we specify stacks of integers:

$$\begin{array}{rl} \text{Stack0} = & \text{Int} \oplus \\ \mathcal{S}: & ST \\ \Omega: & empty: \quad \to ST \\ & push: \quad \mathsf{Z} \times ST \to ST \\ & pop: \quad ST \to ST \\ & top: \quad ST \to \mathsf{Z} \\ \mathcal{A}: & pop(push(x,S)) = S \\ & top(push(x,S)) = x \end{array}$$

This puts only very general restrictions on the models. In particular, it does not address the question as to what the results of *pop* or *top* of en *empty* stack might be. So, at the next step we could decide that *pop*(*empty*) should give us an *empty* stack, i.e., we add one axiom:

$$\begin{array}{rl} \text{Stack1} = & \text{Stack0} \oplus \\ \mathcal{A}: & pop(empty) = empty \end{array}$$

The question of *top*(*empty*) is a bit more involved since it obviously presents us with the problems of partiality. Its solution here amounts to a new design decision. We may decide that its result is some chosen Z element, say 0. Alternatively, we may extend the signature with an additional element $\bullet :\to \mathsf{Z}$ which we will use to mark the results of such applications.

$$\begin{array}{rl} \text{StackE} = & \text{Stack1} \oplus \\ \Omega: & \bullet: \quad \to \mathsf{Z} \\ \mathcal{A}: & top(empty) = \bullet \end{array}$$

This is far from the most satisfying solution but let us stop here for now. We now have that Stack0 ⊃ Stack1 ⊃ StackE, and we can see how this chain of model-class inclusions reflects the more detailed design decisions. The last inclusion involves actually a more elaborate construction since the models of StackE have different signature than those of Stack0. But it is easy to see that each model of the former can be converted (by reduct) to a model of the latter by forgetting the symbol $\bullet$.

This view of development expresses so called *loose* semantics – the whole model class is taken as the meaning of a specification. The *abstractness* of the approach is expressed at two levels.

- Axiomatic descriptions are used to determine the desired properties of the system. On the semantic side, one has a class of models rather than one unique model.
- One always requires such classes to be closed under isomorphisms.

According to the second point, even when only one isomorphism class is determined or chosen, it still abstracts away from the possible, concrete representations of various elements of the carrier. (This does not, of course, apply at the last step where only one program is taken as the implementation.) This leads to the following definition

Definition 8.4 An *abstract data type* is a class of algebras closed under isomorphisms.

Often, it is necessary to introduce additional restrictions on the model class in order to obtain intended algebras. For instance, the closer we are approaching the design stadium where specification becomes almost programmable, we may want to say that our signature contains now all the necessary operations and that we want to consider only algebras generated from this signature. We may want to be even more specific: as we saw in example 4.6, one may be interested in choosing only initial model since this is the model one "really has in mind".

Such considerations lead to a variety of algebraic semantics. The most important variants are given in the following definition.

Definition 8.5 A specification $SP = \langle \Sigma, \Phi \rangle$ can be assigned semantics by the following functions:

$$\begin{array}{llrcl} 1. & loose & & = & Alg(SP) \\ 2. & reachable & Gen(SP) & = & \{G \in Alg(SP) : G \text{ is generated from } \Sigma\} \\ & & Gen_C(SP) & = & \{G \in Alg(SP) : G \text{ is generated from } C \subset \Sigma\} \\ 3. & initial & I(SP) & = & \{I \in Alg(SP) : I \text{ is initial in } Alg(SP)\} \\ 4. & terminal & Z(SP) & = & \{Z \in Alg(SP) : Z \text{ is terminal in } Alg(SP)\} \end{array}$$

The development process from (8.23) can be then refined by taking into account the possibility of determining the semantics of a specification (at least at the later stages of development) by means of these more specific semantic functions:

$$\begin{array}{ccccccccccc} SP_0 & \longrightarrow & SP_n & & & \longrightarrow & SP_m & \longrightarrow & SP_z & \longrightarrow & Prog \\ Alg(SP_0) & \supseteq & Alg(SP_n) & \supseteq & Gen(SP_n) & \supseteq & Gen(SP_m) & \supseteq & I(SP_z) & \ni & [\![Prog]\!] \end{array} \tag{8.24}$$

We have to explain the new concept of $Gen_C(SP)$ appearing in the definition 8.5.

8.3: Reachability and Constructor-Specifications

The concept of *constructor-specification* hidden behind the notation $Gen_C(SP)$ plays a very important role in developing executable specifications. We have seen the concept of generated algebras as well as the associated reasoning by means of induction principle. These concepts were "total" in the sense that they required that an algebra be generated from arbitrary ground terms in the signature. Often, however, this is all too weak a restriction.

Example 8.6 [8.3 continued]

Consider the specification Stack0. One would expect the stacks to satisfy the following property:

$$S \neq empty \Rightarrow S = push(top(S), pop(S)), \tag{8.25}$$

namely, that each non-empty stack S can be obtained by *push*ing its *top* onto the stack S with its *top* element removed. However, this equation does not follow from the axioms of Stack0! Simply, because the loose semantics admits models where some stacks are not constructed (reachable) by applications of $push$. Take, for instance, an initial model of STACK0 and extend it freely with a new ST-element X. This new model will have to satisfy the axioms, i.e., applying pop (resp. top) to any $push(x, X)$ will have to return X (resp. x). Nothing, however, is said what $pop(X)$ (resp. $top(X)$) must be. Furthermore, since X is a new – unreachable – element of sort ST, it isn't really a result of applying $push$ to any other stack and number.

So let us take only the generated models Gen(STACK0). Can we prove the formula (8.25) now? No – because, although all ST-elements are now reachable from Σ, they may be reachable from "wrong" terms. In the initial model we will have elements denoted by terms of the form $pop^n(empty)$ for all $n > 0$, for which we again cannot deduce the required equality. (This can be remedied by adding the axiom as in STACK1, but the present case illustrates the general situation.)

Intuitively, we would like to have all the elements of sort ST generated exclusively by means of the $empty$ and $push$ operations, even though the signature contains other operations returning elements of the sort ST. In other words, we want to say that $empty$ and $push$ are the *constructors* which generate (possibly) new ST-elements, while all the other operations of sort ST are *non-constructors*, i.e., return elements which are generated by constructors.

This is the meaning of $Gen_C(SP)$ where C is a subset of (operations from) the signature of SP. It requires that sorts which are in the range of the C-operations be generated by these operations. Saying that our semantics is $Gen_{\{empty,push\}}$(STACK0) amounts to taking only these models of STACK0 where the sort ST is generated by the applications of $empty$ and $push$. Notice that $push$ has also an argument of sort Z – this sort itself need not be generated, but for any non-$empty$ ST-element S there must be a Z-element z and another generated ST-element S' such that $S = push(z, S')$. (For an SP over signature $\Sigma = \langle \mathcal{S}, \Omega\rangle$, $Gen(SP)$ is the same as $Gen_\Omega(SP)$.)

Definition 8.7 Let $\Sigma = \langle \mathcal{S}, \Omega\rangle$, $C \subseteq \Omega$ and $\mathcal{S}_C \subseteq \mathcal{S}$ be the set of all sorts of operations in C. A Σ-algebra A is *generated w.r.t. the constructors* C, $A \in Gen_C(\Sigma)$, iff for every $S \in \mathcal{S}_C$ and every $a \in A_S$, there is

1. a constructor term $t \in \mathcal{T}(\langle \mathcal{S}_C; C\rangle, X)$ with variables X ranging only over sorts in $\mathcal{S} \setminus \mathcal{S}_C$, and
2. an assignment $\alpha : X \to |A|$

such that $t^A_\alpha = a$.

Remark 8.8

The qualification that $t \in \mathcal{T}(\langle \mathcal{S}_C; C\rangle, X)$ implies that the "constructor part" of t (and all its subterms) is "constructor-ground". For the signature of stacks from example 8.3 and $C = \{empty, push\}$, the term $push(z_1, push(z_2, empty))$ will be a constructor term (z_1, z_2 do not range over sort ST), while the term $push(z_1, push(z_2, S))$ is not a constructor term since it contains the variable S of sort ST.

The condition implies also that the outermost operation in t as well as all its subterms of sorts from $\mathcal{S}_C$ is from C. Let, for instance, $\Sigma = \langle T; a, b : \to T, f : T \to T\rangle$ and $C = \{b, f\}$. The term $f(a)$ is not a constructor term w.r.t. C because $a \notin C$. An algebra $A = \langle \mathsf{N}; 0, 2, \mathsf{s}\rangle$ with $a^A = 0$, $b^A = 2$, $f^A = \mathsf{s}$ is not in $Gen_C(\Sigma)$ because the element 0 is not reachable by constructors. Similarly, $1 = f(a)^A$ is not reachable by constructors, even though f is a constructor operation. (This is rather unusal example. Typically, one does not exclude constants from the set of constructors, because this requires that the interpretation of the excluded constants (in the C-generated models) coincides with the interpretation of some other constructor terms – cf. example 8.11.)

8.3.1: STRUCTURAL INDUCTION ON CONSTRUCTOR-SPECIFICATIONS

Just like with the semantics $Gen(SP)$ we associate the general ω-rule (III.1), so with the $Gen_C(SP)$ we associate a version of this rule ω_C which restricts the substitutions σ not merely to the ground substitutions but to all and only *"constructor-ground"* substitutions.

Example 8.9 [8.6 continued]

For the semantics $Gen_{\{empty,push\}}$(STACK0) and our formula (8.25), we will have to prove that any S built from $empty$ and $push$ is either $empty$ or it satisfies the equation in the consequent. Of course, if $S = empty$, then (8.25) is trivially true, so let us verify that it holds for $S \neq empty$. The structural induction will then continue as follows:

1. $S = push(x, nil)$: here we have $pop(S) = nil$ and $top(S) = x$, so the conclusion follows trivially: $S = push(x, nil) = push(top(S), pop(S))$.
2. $S = push(x, T)$ where T satisfies the induction hypothesis $T = push(top(T), pop(T))$. We then get $push(top(S), pop(S)) = push(top(push(x, T)), pop(push(x, T))) = push(x, T) = S$.

Notice that x remains an arbitrary variable here. In the second point, the induction hypothesis wasn't actually used at all.

Example 8.10

We can choose different signatures or, more precisely, different sets of constructors, for building and axiomatizing finite sets:

$$
\begin{array}{rrrcl}
\Sigma = \mathcal{S}: & E, Set & & & \\
\mathcal{F}_1: & \emptyset: & & \to & Set \\
 & add: & E \times Set & \to & Set \\
\mathcal{F}'_1: & \{\_\}: & E & \to & Set \\
 & \cup: & Set \times Set & \to & Set \\
\mathcal{A}_1: & & add(x, add(x,S)) & = & add(x,S) \\
 & & add(x, add(y,S)) & = & add(y, add(x,S)) \\
\mathcal{A}'_1: & & \{x\} & = & add(x,\emptyset) \\
 & & S \cup \emptyset & = & S \\
 & & S \cup add(x,T) & = & add(x, S \cup T)
\end{array}
\qquad
\begin{array}{rrrcl}
\Sigma = \mathcal{S}: & E, Set & & & \\
\mathcal{F}_2: & \emptyset: & & \to & Set \\
 & \{\_\}: & E & \to & Set \\
 & \cup: & Set \times Set & \to & Set \\
\mathcal{F}'_2: & add: & E \times Set & \to & Set \\
\mathcal{A}_2: & & S \cup \emptyset & = & S \\
 & & S \cup S & = & S \\
 & & S \cup T & = & T \cup S \\
 & & (S \cup T) \cup P & = & S \cup (T \cup P) \\
\mathcal{A}'_2: & & add(x,S) & = & \{x\} \cup S
\end{array}
$$

We thus have two Σ-specifications $SP_1 = \langle \Sigma, \mathcal{C}_1\rangle$ and $SP_2 = \langle \Sigma, \mathcal{C}_2\rangle$, where $\Sigma = \langle \mathcal{S}, \Omega_1 \cup \Omega'_1\rangle = \langle \mathcal{S}, \Omega_2 \cup \Omega'_2\rangle$ and $\mathcal{C}_1 = \mathcal{A}_1 \cup \mathcal{A}'_1$ and $\mathcal{C}_2 = \mathcal{A}_2 \cup \mathcal{A}'_2$. The (implicit) intention here would be that the unprimed parts (at least Ω_1 in SP_1) are the constructors (with the respective axioms – $\mathcal{A}_1$ in SP_1). Without this assumption, it is easy to verify that

$$SP_2 \models \mathcal{C}_1 \quad \text{while} \quad SP_1 \not\models \mathcal{C}_2 \tag{8.26}$$

namely, $SP_1 \not\models S \cup T = T \cup S$. However,

$$\mathbf{reach}_{\Omega_1}(SP_1) \models \mathcal{C}_2 \tag{8.27}$$

We show $\mathbf{reach}_{\Omega_1}(SP_1) \models S \cup T = T \cup S$, by structural induction on T and subinduction on S:

1. $T = \emptyset$, subinduction on S:
   (a) for $S = \emptyset : \emptyset \cup \emptyset = \emptyset = \emptyset \cup \emptyset$
   (b) for $add(x, S) : IH = S \cup \emptyset = \emptyset \cup S \;\Rightarrow$
   $\emptyset \cup add(x,S) \stackrel{2}{=} add(x, \emptyset \cup S) \stackrel{\mathrm{IH}}{=} add(x, S \cup \emptyset) = add(x,S) = add(x,S) \cup \emptyset$
2. $IH_T = \forall S : S \cup T = T \cup S$ should imply $\quad \Rightarrow \; S \cup add(x,T) = add(x,T) \cup S$
   (a) for $S = \emptyset : \emptyset \cup add(x,T) = add(x, \emptyset \cup T) \stackrel{IH_T}{=} add(x, T \cup \emptyset) = add(x,T) = add(x,T) \cup \emptyset$
   (b) for $add(y,S) : IH_S = S \cup add(x,T) = add(x,T) \cup S$
   . $\quad \Rightarrow \; add(y,S) \cup add(x,T) = add(x,T) \cup add(y,S)$
   $add(y,S) \cup add(x,T) = add(x, add(y,S) \cup T) \stackrel{IH_T}{=} add(x, T \cup add(y,S)) =$
   $add(x, add(y, T \cup S)) \stackrel{IH_T}{=} add(x, add(y, S \cup T)) = add(y, add(x, S \cup T)) =$
   $add(y, S \cup add(x,T)) \stackrel{IH_S}{=} add(y, add(x,T) \cup S) = add(x,T) \cup add(y,S)$

What does it mean? (8.26) means that $SP_1 \longrightarrow SP_2$ but not vice versa. Also, that SP_1 is "reasonable" when we take the reachable part. In fact, SP_2 looks more "natural" as a possible initial specification of abstract requirements on sets. (8.27) does mean that $SP_2 \longrightarrow \mathbf{reach}_{\Omega_1}(SP_1)$ and this will most probably be the development direction.

The constructive character of such a decision has many facets. One is that the desirable properties, like the commutativity of $\cup$, follow in $\mathbf{reach}_{\Omega_1}(SP_1)$ from the axioms defining $\cup$ in terms of constructors. In fact, any Set-term in SP_1 involving $\cup$ will be provably equal to a one containing only Ω_1. Typically, one will define possible new operations in such a way, rather than impose independent axioms on such a new operation (as it would be if we just added axioms $\mathcal{A}'_2$ to the specification SP_1).

Thus, any new operation will require more axioms in SP_2 than in SP_1. E.g. adding the operation $\in : E \times Set \to \mathsf{B}$, will require axioms:

$$\mathcal{A}_1'' : \quad \begin{array}{rcl} x \in \emptyset & = & \bot \\ x \in add(y,S) = \top & \Leftrightarrow & x = y \vee \\ & & x \in S = \top \end{array} \qquad \mathcal{A}_2'' : \quad \begin{array}{rcl} x \in \emptyset & = & \bot \\ x \in \{y\} = \top & \Leftrightarrow & x = y \\ x \in S \cup T = \top & \Leftrightarrow & x \in S = \top \vee \\ & & x \in T = \top \end{array}$$

Constructor-specifications are very handy but they have to be used with a great care. Careless application of this method may lead to various problems of which we give only a simple example.

Example 8.11

Let $\Sigma = \langle A; a, b, c : \to A \rangle$, and consider $Gen_{\{a,b\}}(\Sigma)$, i.e., the Σ-algebras generated by a and b. Obviously, in this class c must be interpreted as the same element as either a or b but we do not know which one. The induced induction principle enables us to prove $c = a \vee c = b$. If the specification is more complex and a and b may have some different properties, this will lead to uncertainity as to which of these properties pertain to c.

As a more grave problem, take the specification SP over the same Σ and with axioms $a \neq c$ and $b \neq c$. Then $Gen_{\{a,b\}}(SP)$ is inconsistent! The axioms state exactly the negation of $c = a \vee c = b$ which can be proved by the induction principle induced by the generating restriction.

The theory of constructor-specifications is quite an elaborate research area but we won't study it in more detail. In what follows we will at most use the possibility of restricting the semantics of the specifications by means of Gen_C rather than only Gen. Then, in the proofs, we will of course take recourse to the associated induction principle. It can, however, be used as a general guideline for writing specifications (both in determining the signature and axiomatiziation) – even if we do not, at the moment, make an explicit decision what the constructors are, we can still *think* as of some of the operations as constructors and of others as definable in terms of these "not yet constructors".

8.4: Languages for Basic Specifications

Recall that, for instance, the existence of generated models depends on the signature (which must be non-void), while the existence of initial/terminal objects on the form of the axioms (the language for writing the axioms). This means that intending some particular semantics for our specifications, we have to consider what language ensures that this semantics will exist.

Since the final specification, i.e., the program, is an executable piece of code, the language in which it is written is much more restricted than, for instance, FOL, or even the merely equational part of FOL. Also, as we know from week 6, if we are interested in initial semantics, we cannot, in general, use more expressive language than $\mathcal{L}^{=}_{\Rightarrow}$. In fact, the development process as illustrated in (8.24) reflects not only the gradual restriction of the model class. Typically, one will start with a more powerful language for writing the general requirements and, moving along the chain of gradual design decisions, restrict the language coming closer to a possible programming language.

There are some types which it is common to have defined with a fixed, often initial, semantics. If such a semantics is not the one we are using, one has to resort to a more expressive language in order to achieve analogous (or even the same) result.

Example 8.12 [Initial vs. non-initial specification of Booleans]

Consider the following specification of Booleans:

$$\begin{array}{lll} \text{BOOL} = & & \\ \mathcal{S}: & \mathsf{B} & \\ \Omega: & \top : & \to \mathsf{B} \\ & \bot : & \to \mathsf{B} \\ \Phi: & & \top \neq \bot \\ & & x = \top \ \vee \ x = \bot \end{array}$$

It tells what we want to tell about Boolean sort: it has exactly two distinct elements interpreting, respectively, $\top$ and $\bot$.

This particular specification does have the initial semantics. However, it is only a coincidence – the language $\mathcal{L}^{=}_{\vee}$ used here does not admit initial semantics in general. Thus, BOOL may be some early specification of the intended Booleans made at the stage where we do not focus on the existence of initial models.

After having performed some refinement steps on our specification (BOOL is supposedly only a piece of a bigger whole), we may have arrived at the stage where we want to consider initial semantics. At this point, we may merely write:

$$\begin{array}{lll} \text{IBOOL} = & & \\ \mathcal{S}: & \mathsf{B} & \\ \Omega: & \top : & \to \mathsf{B} \\ & \bot : & \to \mathsf{B} \end{array}$$

The intention was from the very begining to have just two Boolean elements **tt** and **ff**, corresponding to $\top$ and $\bot$. In BOOL this was achieved by means of the axioms in a more expressive language. Since IBOOL is equational (no axioms are, in particular, equational axioms), we can exclude all its "junky"models by saying $I(\text{IBOOL})$. (Due to the simplicity of this examples, this model will be the same as an initial model of BOOL.) Notice (and verify, exercise 8.1) that the axioms of BOOL will be satisfied by the initial model of IBOOL.

This simple example illustrates that sometimes we may have a choice of the language for specifying our algebras. It does not, however, convey the generality of the problem which is this: there are some data types (algebras) which cannot be axiomatized in all languages. I.e., axiomatization of some data types may require that we use a language as expressive as, for instance, $\mathcal{L}^{=}_{\Rightarrow}$ or $\mathcal{L}^{=}_{\vee}$.

Example 8.13 [No equational specification]

Let Σ_* be the signature Σ_{N} (example 4.26) with one new operation symbol $** : \mathsf{N} \to \mathsf{N}$. A known theorem says that the Σ_*-algebra N with the carrier (isomorphic to) the natural numbers and $**^N$ interpreted as the square function (i.e., $**(n)^N \stackrel{\text{def}}{=} n^2$) has no (finite) equational specification. Even worse: no matter what equations we choose, and in addition require initial semantics, there is still no way to obtain N as the model.

We do not prove this fact here but only notice that if we had an operation of multiplication $*$ we could define $**(n) = n * n$. However, addition of $*$ means extension of the signature as well as the algebra. We would obtain an algebra which, perhaps, is our intended N but which in addition contains the multiplication.

The closure properties studied in the previous week provide the tools for determining what kind of language one may need in order to specify some intended data type.

Example 8.14 [7.16 continued]

Intuitively, the data type Bool should have exactly two elements in the carrier. The example 6.11 demonstrated that this cannot be achieved by mere equations, and example 7.16 that we do need $\mathcal{L}^{=}_{\vee}$, as it was done in example 8.12, for this purpose.

In the sequel, we will always allow ourselves to choose appropriate language. This is motivated mainly by the fact that we do not focus on any specific methodology or formalism but consider the general loose semantics and model-class inclusion as the notion of implementation. One should however always attempt to give specifications in simplest possible form, i.e., in the simplest possible language, and resort to a more expressive language only when it is necessary.

Exercises (week 8)

EXERCISE 8.1 Show the statement from example 8.12.

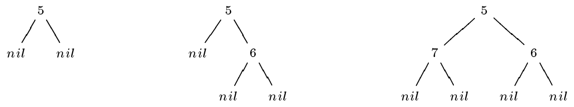
1. Show that $\text{IBOOL} \vdash_{CEQI} x = \top \vee x = \bot$. ($CEQI$ is like EQI – CEQ extended with ω-rule.) Assuming an analogue of theorem 5.14 for $CEQI$ wrt. generated models (i.e., $\Phi \vdash_{CEQI} C \Leftrightarrow Gen(\Phi) \models C$) conclude that $I(\text{IBOOL}) \models x = \top \vee x = \bot$.

2. Show then that IBOOL $\not\vdash_{EQI}$ $\top = \bot$ using theorem 5.14 (i.e., by finding a generated model which does not satisfy this equation). Use this and lemma 4.17 to conclude that $I(\text{IBOOL}) \models \top \neq \bot$.

<u>EXERCISE 8.2</u> We will specify BINT – a data type of binary trees of integers over the following signature Σ_{BINT}

$$
\begin{array}{rlll}
\mathcal{S}: & \mathsf{Z}, BT & & \\
nil: & & \to BT & \text{– empty tree} \\
tr: & BT \times \mathsf{Z} \times BT & \to BT & \text{– a new tree with root } \mathsf{Z}\text{, left and right subtrees} \\
L, R: & BT & \to BT & \text{– return left, resp. right subtree – } nil's \text{ for } nil \\
val: & BT & \to \mathsf{Z} & \text{– returns the value at the root of the tree – 0 for } nil \\
sum: & BT & \to \mathsf{Z} & \text{– returns the sum of all nodes} \\
hgh: & BT & \to \mathsf{Z} & \text{– returns the height of the tree – 0 for } nil
\end{array}
$$

1. Write the expressions which would naturally denote the following trees:



2. Assuming given a specification of natural numbers with all the operations you might find necessary, write equational axioms in $\mathcal{L}^=$ for the operations from this signature.

3. Which among the semantics from definition 8.5 will correspond best to the intended interpretation as binary trees? If it is of the kind Gen_C, which constructors will you choose?

4. Assuming that $T \neq nil$, can you prove from your axioms that $T = tr(L(T), val(T), R(T))$? If not, choose appropriate constructors and then try to prove this fact by induction on T.

5. A node in a (binary) tree is a leaf iff both its subtrees are nil. Write the axioms (not necessarily in $\mathcal{L}^=$) for the operation $leaf : BT \to \mathsf{B}$ which returns $\top$ iff its argument is a leaf.

<u>EXERCISE 8.3</u> We will axiomatize a very simple line-editor. A line is thought of as consisting of two strings represententing the contents of the line, respectively, to the left and to the right of the cursor. (I.e., cursor is always positioned *between* symbols.) We will have three sorts: S and Str as in exercise 6.3, and $Line$, and an operation $ln : Str \times Str \to Line$. For instance $ln(ab, cde)$ represents the line with the text $abcde$ and the cursor placed between b and c. Moving cursor one position to the left would then result in the line $ln(a, bcde)$, and moving it to the right end in $ln(abcde, \varepsilon)$. We want the line-editor to perform the following operations:

1. in addition to the operation ln generating a new line,
2. nl – generating a new, empty line;
3. ins – inserting a symbol from S at the right end of the string to the left from the cursor (e.g., $ins(x, ln(ab, de))$ should give the line $ln(abx, de)$);
4. del – deleting the symbol (if any) to the left of the cursor;
5. mL (mR) – moving cursor one position to the left (right);
6. beg (end) – moving cursor to the begining (end) of the line.

1. Assuming given the signature and axioms for Strings from exercise 6.3, write the signature and then axiomatize these operations of the line-editor. (Notice that the operations del, mL and beg (resp. mR, end) should do nothing if the string to the left (resp. right) of the cursor is empty.)

2. Extend now the signature with new operations: cl – clearing the whole line, and $delR$ – for deleting the symbol (if any) to the right of the cursor. Give the axioms defining these operations in terms of (some of) the operations 1-6.

Week 9: Structured Specifications

- The Need for Structuring Mechanisms
- Basic Specification-Building Operations
- Structured Algebraic Specifications

9.1: The Need for Structured Specifications

Development process, as illustrated in (8.23), isn't very realistic. The reason is that during development, we will certainly introduce new features into our models of which we were not aware at the begining. It is doubtful whether one would start a specification of, for instance, a sorting program with the full signature – including all the auxiliary operations – which, eventually, will have to be implemented in a sorting program.

Example 9.1

Suppose that we want to specify sorting the lists of integers of arbitrary length and that we have specifications Int and Bool of integers and Booleans. We might start with the following specification of lists of integers:

$$
\begin{array}{lrrcl}
\multicolumn{5}{l}{\textsc{List} = \textsc{Int} \oplus \textsc{Bool} \oplus} \\
\mathcal{S}: & \multicolumn{4}{l}{List} \\
\Omega: & nil: & & \to & List \\
 & cons: & \mathsf{Z} \times List & \to & List \\
 & head: & List & \to & \mathsf{Z} \\
 & tail: & List & \to & List \\
 & \in: & \mathsf{Z} \times List & \to & \mathsf{B} \\
\Phi: & & head(cons(x,l)) & = & x \\
 & & tail(cons(x,l)) & = & l \\
 & & cons(x,l) & \neq & nil \\
 & & cons(x,l) = cons(x',l') & \Rightarrow & x = x' \wedge l = l' \\
 & & x \in nil & = & \bot \\
 & & x \neq y & \Rightarrow & x \in cons(y,l) = x \in l \\
 & & x \in cons(x,l) & = & \top
\end{array}
$$

Realizing the need for a sorting operation on such lists, we would then extend this specification:

$$
\begin{array}{lrrcl}
\multicolumn{5}{l}{\textsc{SList1} = \textsc{List} \oplus} \\
\Omega: & sorted: & List \to \mathsf{B} & & \\
 & sort: & List \to List & & \\
\Phi: & & sorted(nil) & = & \top \\
 & & sorted(cons(x,l)) = \top & \Leftrightarrow & \forall y(y \in l = \top \Rightarrow x \le y = \top) \wedge sorted(l) = \top \\
 & & sorted(sort(l)) & = & \top \\
 & & x \in l & = & x \in sort(l)
\end{array}
$$

Remark 9.2

Notice that the specification of *sort* in exampel 9.1 uses an auxiliary predicate *sorted* which merely states the property of a list being sorted. The operation *sort* which is actually supposed to perform sorting is specified merely by claiming that its result is *sorted*.

This property is specified *loosely* by the mere requirements that the sorted list contains exactly the same values as the original one and that they now appear in a sorted order. However, no operational definition of sorting is implied and even the result of sorting is not uniquely determined! For instance $sort(\langle 2,1,1,nil\rangle)$ may in some models result in the list $\langle 1,1,2,nil\rangle$ (writing $\langle x,y...z,L\rangle$ instead of $cons(x,cons(y,..,cons(z,L)..)))$ – check that this list satisfies the *sorted* predicate. In other models, however, the *sort* operation may, as a side-effect, remove the duplicates from the list – it may result in the list $\langle 1,2,nil\rangle$, which also satisfies this predicate. As a matter of fact, we will also have models in which application $sort(\langle 2,1,1\rangle)$ results, for instance, in $\langle 1,1,1,2,2,2\rangle$! The above specification simply does not address the question of duplicates at all.

Such a stepwise and piecewise development of specifications is a very natural process and this raises a very important question of relating various pieces of specification to each other. The implementation relation is the central issue but there is also another aspect. Software systems tend to be very large and complicated, and so do specifications. Ideally, one would like to split the task of specifying into smaller subtasks and, after having completed each subtask, being able to compose the whole system from smaller – and hence more manegable – pieces. In the above examples, for instance, we did not bother to write the specifications of Booleans and integers in full. Instead we used the symbol $\oplus$ and names of existing specifications. This is a very common situation where one wants to *reuse* an existing specification. We will now address this problem of composing and relating specifications in a general setting. Later, we will return to the implementation issue.

9.2: Basic Specification-Building Operations

Since we want to design some operations manipulating specifications, we need a precise statement as to what kind of objects specifications are. There are at least three basic views on what may be taken as the semantics of a specification:

1. Specification can be seen at the *presentation level*, i.e., as a syntactic object consisting of a signature and a set of formulae: $SP = \langle \Sigma, \Phi \rangle$. Although this is how specifications look in practice, it presents some problems since one can easily imagine several equivalent presentations. This notion of "equivalence" is incorporated into the two following views.
2. Seen at the *theory level*, a specification is a signature together with the class of formulae closed under some logical consequence relation: $SP = \langle \Sigma, Th(\Phi) \rangle$. Here, two "equivalent" specifications will be actually the same (modulo renaming of the symbols in the signature). This is still a syntactic notion, although the class $Th(\Phi)$ is typically infinite and not recursive.
3. No matter which of the above two notions we choose, it is always the case that a specification determines a class of models. At the *model-class* level, one sees a specification as consisting of a signature and a class of models $\mathsf{K} \subseteq \mathsf{Alg}(\Sigma)$, i.e., $SP = \langle \Sigma, \mathsf{K} \rangle$. This meaning abstracts away from the syntactic form and axioms, taking into account only their possible realizations.

Since specifications are ment to describe classes of programs which we want to view as their correct realizations, i.e., *specifications ultimately determine classes of algebras*, we will follow this last view. (In any case, the natural mappings from presentations to theories and from theories to model-classes, mean that this underlies the other two views as well.)

For an $SP = \langle \Sigma, \mathsf{K} \rangle$ we let $[\![SP]\!] = \mathsf{K}$ denote the class of models of SP, and $Sig(SP) = \Sigma$ the signature. Thus $[\![SP]\!] \subseteq \mathsf{Alg}(Sig(SP))$. Of course, as the minimal requirement, we demand that $[\![SP]\!]$ be closed under isomorphisms.

A *specification language* will provide the syntactic means for 1) writing basic specifications (signatures and axioms) and 2) expressing various relationships between them. Its semantics will be given in form of the *specification building operations* which are the mappings $\alpha : Spec \to Spec$ between specifications, i.e., for a $\langle \Sigma'; \mathsf{K}' \rangle \in Spec : \alpha(\langle \Sigma'; \mathsf{K}' \rangle) = \langle \Sigma; \mathsf{K} \rangle \in Spec$. We introduce four basic specification constructs for writing the *flat* specifications, *extending* specifications with additional sort or function symbols and axioms, for *renaming* and *hiding* parts of a specification, and for picking only generated models from the model class.

Basic (flat) specification (9.3) :

$$
\begin{array}{rcl}
\multicolumn{3}{l}{SP = \langle \Sigma; \Phi \rangle} \\
Sig(SP) & = & \Sigma \\
[\![SP]\!] & = & \{A \in \mathsf{Alg}(\Sigma) : A \models \Phi\}
\end{array}
$$

Here Φ is a set of Σ-formulae in the chosen logical language. We are using some subset of first order logic but, in general, the language may be much more powerful. Also, we have written SP as a pair $\langle\Sigma;\Phi\rangle$ rather than as $\langle\Sigma;\mathsf{K}\rangle$ since for flat specifications the class $[\![SP]\!]$ is uniquely determined by the axioms as their model-class.

Enrichment (9.4) :

$$\begin{array}{rcl} SP' = SP \;\oplus\; [\mathcal{S}:S;\Omega:F;\mathcal{A}:\Psi] & & \\ Sig(SP') & = & Sig(SP)\cup S\cup F \\ [\![SP']\!] & = & \{A\in \mathsf{Alg}(Sig(SP')) : A|_{Sig(SP)}\in [\![SP]\!] \wedge\; A\models\Psi\} \end{array}$$

This is typically written as "**enrich** SP **by** **sorts** : S, **ops** : F, **axioms** : Ψ". Notice that while the first operand of $\oplus$ is a specification, the other one is an arbitrary list of sort and function symbols and axioms. It is, of course, understood that the new operations have profiles over sorts from the new signature $Sig(SP')$.

Derive (9.5) : For $Sig(SP)=\Sigma$ and a signature morphism $\sigma:\Sigma'\to\Sigma$

$$\begin{array}{rcl} SP' = SP|_\sigma & & \\ Sig(SP') & = & \Sigma' \\ [\![SP']\!] & = & \{A|_\sigma : A\in[\![SP]\!]\} \end{array}$$

This is typically written as "**derive from** SP **by** σ".

Finally, we mention also an operator for restricting the model-class to the models generated from some ground terms. The semantic construction $Gen_C(SP)$ was discussed in subsection 8.3.

Reach (9.6) : for SP over a non-void signature $\langle\mathcal{S};\Omega\rangle$ and $C\subseteq\Omega$:

$$\begin{array}{rcl} SP' = \mathbf{reach}_C(SP) & & \\ Sig(SP') & = & Sig(SP) \\ [\![SP']\!] & = & Gen_C(SP) \end{array}$$

9.3: Variants and Examples

9.3.1: Derive : Hiding and Renaming

The semantics of the **derive** operation is defined as the reduct. Consequently, as the effect of reduct varies depending on the kind of signature morphism (example 4.29), this operation can be used to perform different functions depending on the actual signature morphism. When it is a simple inclusion, i.e., $\Sigma=\Sigma'\cup S\cup F$, we obtain a variant called

Hiding (9.7 from 9.5) : for inclusion $\sigma:\Sigma'\hookrightarrow\Sigma$, instead of $SP'=SP|_{\Sigma'}$ we write

$$\begin{array}{rcl} SP' = SP\ominus[\mathcal{S}:S,\Omega:F] & & \\ Sig(SP') & = & Sig(SP)\setminus S\setminus F \\ [\![SP']\!] & = & \{A|_{\Sigma'} : A\in[\![SP]\!]\} \end{array}$$

This is typically written as "**hide sorts** : S, **ops** : F **in** SP". It should be considered as a mere abbreviation for the more elaborate explicit specification of the actual signature morphism.

Hiding removes the unnecessary parts of the signature and makes them "invisible" to the outside world. It does not, however, remove the properties of the models which are implied by the properties of these hidden parts.

Example 9.8 [9.1 continued]

Recall the specification of sorting lists obtained by enriching the specification List as follows:

$$\begin{array}{lrcl} \textsc{SList1} = \textsc{List}\oplus & & & \\ \Omega: & sorted: & List\to\mathsf{B} & \\ & sort: & List\to List & \\ \Phi: & sorted(nil) & = & \top \\ & sorted(cons(x,l))=\top & \Leftrightarrow & \forall y(y\in l=\top\Rightarrow x\le y=\top)\wedge sorted(l)=\top \\ & sorted(sort(l)) & = & \top \\ & x\in l & = & x\in sort(l) \end{array}$$

Writing this specification we are not really interested in the auxiliary predicate *sorted*. It is doubtful that the eventual program sorting lists will have to contain the implementation of this predicate. Thus, having written this specification, we should say that what we want to implement is:

$$\textsc{SList} = \textsc{SList1}\ominus[\Omega:sorted]$$

If we did not hide *sorted*, we would be forced in the further development process to implement this operation – hidding releases us from doing that. Nevertheless, the models of SList are not just the models of the reduced signature – they are *reducts* of the models of the *full* signature. Thus they will have to satisfy *all* the axioms – also these involving the hidden operations like *sorted*.

Example 9.9 [8.3 continued]

The chain of model-class inclusions in the implementation process from example 8.3 was not strictly correct because the signature of StackE, including the additional symbol $\bullet:\to\mathsf{Z}$, was different from the signature of Stack1. Using the hiding operation we can now achieve the required effect as follows

$$\textsc{Stack0}\longrightarrow\textsc{Stack1}\longrightarrow\lceil\textsc{StackE}\ominus[\Omega:\bullet]\rfloor \tag{9.28}$$

Another very common operation which can be obtained from **derive** amounts to simultaneously *hiding and renaming* whatever is not hidden. Here we use not a mere inclusion but an injective signature morphism

Hide&Rename (9.10 from 9.5) : for injective $\sigma:\Sigma'\to\Sigma$, and $Sig(SP)=\Sigma$

$$\begin{array}{rcl} SP' = SP|_\sigma & & \\ Sig(SP') & = & \Sigma' \\ [\![SP']\!] & = & \{A|_\sigma : A\in[\![SP]\!]\} \end{array}$$

When σ is only injective (but not surjective) this construction hides some operations and renames the rest. The effect of full renaming is achieved when σ is bijective.

Example 9.11 [Hiding-and-Renaming as **derive**]

When we have collected several pieces of specifications in a library, their reuse will often require renaming the components of one in order to be usable by another. Assume that we have the List specification from example 9.1, and now want to produce a specification for dealing with trains and railroad cars.

$$\begin{array}{lrcl} \textsc{Trains} = & & & \\ \mathcal{S}: & Train, rCar & & \\ \Omega: & one: & & rCar\to Train \\ & addC: & & rCar\times Train\to Train \\ & first: & & Train\to rCar \\ & rest: & & Train\to Train \\ \Phi: & first(one(x)) & = & x \\ & rest(one(x)) & = & one(x) \\ & first(addC(x,l)) & = & x \\ & rest(addC(x,l)) & = & l \\ & addC(x,l) & \ne & one(y) \\ & addC(x,l)=addC(x',l') & \Rightarrow & x=x'\wedge l=l' \end{array}$$

The operation *one* produces a one-car train and, unlike in LIST, we do not have an empty train. There is, however, a close similarity between the two specifications: *addC* corresponds to *cons*, *rest* to *tail* and *first* to *head*. In particular, if we have an implementation of LIST, it might be tempting to use it to implement TRAIN. We would then have to hide the unnecessary bits of (the implementation of) LIST, point out the above correspondance and adjust LIST to include the operation *one* and modify *tail*. So let us define

$$\begin{array}{lrcl} \text{LIST2} = & \text{LIST} \oplus & & \\ \Omega: & one: & & \mathsf{Z} \to List \\ & rest: & & List \to List \\ \mathcal{A}: & one(x) & = & cons(x, nil) \\ & rest(one(x)) & = & one(x) \\ & l \neq nil & \Rightarrow & rest(cons(x,l)) = tail(l) \end{array}$$

and $\sigma : \Sigma_{\text{TRAIN}} \to \Sigma_{\text{LIST2}}$ be given by

$$\sigma \;=\; \{rCar \mapsto \mathsf{Z},\ Train \mapsto List,\ addC \mapsto cons,\ first \mapsto head,\ rest \mapsto rest,\ one \mapsto one\}. \quad (9.29)$$

Then, instead of the above specification, we may write

$$\text{TRAIN} = \textbf{derive from} \ \lceil \ \textbf{hide} \ \Omega : nil, \in, tail \ \textbf{in} \ \text{LIST2} \ \rfloor \ \textbf{by} \ \sigma \quad (9.30)$$

Notice that since **hide** is a variant of **derive**, and what is hidden is not in the image of σ, this may be written equivalently as

$$\text{TRAIN} = \textbf{derive from} \ \text{LIST2} \ \textbf{by} \ \sigma.$$

Another relevant observation one may make here is that the specification $\text{LISTNE} = \lceil \ \textbf{hide} \ \Omega : nil, tail, \in \ \textbf{in} \ \text{LIST2} \ \rfloor$ used as the intermediary argument in the specification (9.30) of TRAIN describes essentially non-empty lists. It is to be expected that we will have such a specification around in our library, in which case trains would require merely that we rename – but not hide anything from – the specification LISTNE. This would look the same, i.e., TRAIN = **derive from** LISTNE **by** σ, only that now σ as in (9.29) is bijective.

9.3.2: TRANSLATE: RENAME

The situation like above with a mere *renaming*, can be covered by **derive** with a bijective signature morphism. But the construction may look a bit strange: a signature morphism $\sigma : \Sigma' \to \Sigma$ effects a renaming in the *opposite* direction, i.e., the semantics of the resulting specification is obtained by taking each model of the argument specification over Σ and constructing its Σ'-reduct. Therefore, this construction is typically used in conjunction with hiding as it was done in example 9.11. Intuitively, however, one might expect that such a σ allows us to **translate** any Σ'-specification into a Σ-specification. Such a translation is a different operation defined as follows:

Translate (9.12) : for any signature morphism $\sigma : \Sigma' \to \Sigma$

$$SP = \textbf{translate} \ SP' \ \textbf{by} \ \sigma$$
$$\begin{array}{rcl} Sig(SP) & = & \Sigma \\ [\![SP]\!] & = & \{A \in \mathsf{Alg}(\Sigma) : A|_\sigma \in [\![SP']\!]\} \end{array}$$

Notice that this is a *new* operation which is not a special case of any one previously defined. Here we inclde in the result all the Σ-algebras such that their σ-reduct is a model of the argument specification. The semantics of **derive** was defined, so to speak, in the opposite way: we took all the models of the argument specification and converted them by the σ-reduct into models of the result. It is easy to verify that, in the special case (and only) when σ is a bijective signature morphisms, with the inverse morphism σ^-, the two yield the same result, i.e.:

$$\text{for bijective } \sigma: \ \textbf{derive from} \ SP' \ \textbf{by} \ \sigma^- = \textbf{translate} \ SP' \ \textbf{by} \ \sigma \quad (9.31)$$

In such cases, we often write simply **rename** SP' **by** σ. Thus, instead of the last equation for TRAIN from example 9.11, we would write **rename** LISTNE **by** σ^-, where σ is as in (9.29).

9.3.3: ENRICH : QUOTIENT, UNION AND SUM

The **enrich** construction is a very powerful operation since it essentially allows one to write an entirely new specification. It, too, has some more specific variants depending on the form of the second argument. The first one is obtained when we enrich only with some new *equational* axioms. Adding equations to a specification amounts to forcing several identities to hold in the models. Identities, being congruences, yield quotients, and so we obtain:

Quotient (9.13 from 9.4) : for E a set of Σ-equations, instead of $SP \oplus \mathcal{A} : E$, we write

$$SP' = SP/_{\equiv_E}$$
$$\begin{array}{rcl} Sig(SP') & = & Sig(SP) \\ [\![SP']\!] & = & \{A/_{\equiv_E} : A \in [\![SP]\!]\} \end{array}$$

This is also written as "**quotient** SP **by** E". The congruence $\equiv_E$ is induced by E as in example 2.30, exercise 6.5.

Example 9.14

Let us consider the following specification (which is just a part of the specification LIST from example 9.1):

$$\begin{array}{rlrcl} \text{LIST0} = & \text{INT} \oplus & & & \\ \mathcal{S}: & List & & & \\ \Omega: & nil: & & & \to List \\ & cons: & \mathsf{Z} \times List & \to & List \\ & head: & List & \to & \mathsf{Z} \\ F: & head(cons(x,L)) & = & x & \end{array}$$

nil generates an empty list, *cons* prepends a new element to the list and *head* returns the head element of the list (the one which has been inserted most recently).

Consider now the congruence induced by the equation

$$E : cons(x_1, cons(x_2, cons(x_3, L))) = cons(x_1, cons(x_2, L'))$$

It will identify all lists which have the same first two elements. There isn't much to check here since this axiom implies immediately congruence wrt. *cons*, i.e., if $L \equiv_E L'$ then L and L' have the same two first elements, and so $cons(x, L) \equiv_E cons(x, L')$. The same holds for *head*, since if L and L' have the same two frontmost elements then certainly $head(L) = head(L')$. Thus taking the quotient $\text{LIST0}/_{\equiv_E}$ yields a specification of lists with at most two elements. Any model of $\text{LIST0}/_{\equiv_E}$ is obtained by taking a quotient of a model of LIST0.

Let us now suppose that we have an extended LIST0 specification with the operation returning also the last element of a list:

$$\begin{array}{rlrcl} \text{LIST1} = & \text{LIST0} \oplus & & & \\ \Omega: & last: & List \to \mathsf{Z} & & \\ F1: & & last(cons(x,nil)) & = & x \\ & & last(cons(x,cons(y,L))) & = & last(cons(y,L)) \end{array}$$

What happens if we attempt to take the quotient of LIST1 with the congruence induced by the axiom E? A bad and, most probably, unintended thing! The axioms $F1$ say, for instance, that for the two lists $L = cons(1, cons(2, cons(3, nil)))$ and $L' = cons(1, cons(2, nil))$, $last(L) = 3$, while $last(L') = 2$. But according to E, L and L', having the same two frontmost elements, are to be identified. The congruence requirement wrt. the *last* operation will then imply that also $last(L) = 3$ and $last(L') = 2$ should be identified. In fact, all the Z-elements will be identified in this way, and we will be left with only three equivalence classes in the resulting quotient construction, namely $[nil]$, $[cons(\bullet, nil)]$ and $[cons(\bullet, cons(\bullet, nil))]$.

Although almost the same, **quotient** is not exactly the same as **enrich** by equations. Just observe that the semantics of the two is defined in a slightly different ways. Enrichment by new equations requires the (new) models to satisfy both the old and the new axioms. Quotient merely performs a construction on the existing models.

Example 9.15 [**quotient** vs **enrich**]

Let SP have one sort, two constants s and t, and one axiom $s \neq t$. The semantics of $SP_e = \lceil$**enrich** SP **by** $s = t\rfloor$ given in (9.4) will here reduce to $[\![SP_e]\!] = \{A : A \models s = t \wedge A \models s \neq t\} = \emptyset$ – the new axioms are inconsistent with the old ones. The semantics of $SP_q = \lceil$**quotient** SP **by** $s = t\rfloor$ on the other hand, is defined as $[\![SP_q]\!] = \{A/_{s=t} : A \in [\![SP]\!]\}$, that is, we take each model of SP and perform the quotient construction by the congruence induced by $s = t$ on it. This class obviously will not be empty. Each model in $[\![SP_q]\!]$ will satisfy the new equation $s = t$ but not the old axiom $s \neq t$.

This is a subtle difference and in most practical cases we may consider **quotient** as a special case of **enrichment**. In particular, if all our axioms are simple or conditional equations, such inconsistencies will never arise. Care is needed, however, when the axioms are written in a language allowing us to express negation and hence when there is a possibility of inconsistency.

Another specific variant of **enrich** is when the second argument is an entire specification. Even more special case is obtained when both specifications have the same signature. Then combining the two amoutns to their union and so is this variant called

Union (9.16 from 9.4) : for $Sig(SP_1) = Sig(SP_2) = \Sigma$, instead of $SP_1 \oplus SP_2$ we write

$$SP' = SP_1 \cup SP_2$$
$$\begin{array}{rcl} Sig(SP') & = & \Sigma \\ [\![SP']\!] & = & [\![SP_1]\!] \cap [\![SP_2]\!] \end{array}$$

Example 9.17 [Union as **enrich**]

Let SP_1 be a specification of Booleans and integers with sorts B and Z and some operations. Let SP_2 be the specification LIST from example 9.1. Although it is perfectly fine to obtain SP_2 by enriching SP_1 as it was done in example 9.1, we cannot write $SP_1 \cup SP_2$ since the two specifications have different signatures.

Thus, applicability of $\cup$ is restricted by the requirement that both specifications must have the same signature. It will often be the case that two specifications share only some parts of the signatures (e.g., Booleans, integers) which we therefore would like to identify while combining the remaining parts. We may then **rename** (appropriate parts of) the argument specifications before constructing their union. As an alternative, there is a more general form of union, namely sum:

Sum (9.18) :

$$SP' = SP_1 + SP_2$$
$$\begin{array}{rcl} Sig(SP') & = & \Sigma' = \Sigma_1 \cup \Sigma_2 \\ [\![SP']\!] & = & \{A \in \mathsf{Alg}(\Sigma') : A|_{\Sigma_1} \in [\![SP_1]\!] \wedge A|_{\Sigma_2} \in [\![SP_2]\!]\} \end{array}$$

Notice (and verify, exercise 9.8) that since we have two inclusions, i.e., injective signature morphisms $\iota_i : \Sigma_i \to \Sigma' = \Sigma_1 \cup \Sigma_2$, the following equality holds

$$SP_1 + SP_2 = \lceil\textbf{translate } SP_1 \textbf{ by } \iota_1\rfloor \;\cup\; \lceil\textbf{translate } SP_2 \textbf{ by } \iota_2\rfloor \qquad (9.32)$$

Example 9.19 [9.17 continued]

Let Σ' be the union of the signatures of SP_1 and SP_2 (i.e., it is essentially the signature of SP_2 which includes Booleans and integers. The specification $SP_1 + SP_2$ will result in the identification of these common sorts and operations. Notice that this requires the common operations in $[\![SP_1]\!]$ and in $[\![SP_2]\!]$ to be compatible, in the sense that, their axiomatizations in both specifications are not mutually contradictory. If, for instance, SP_1 contains a unary Boolean operation $e : \mathsf{B} \to \mathsf{B}$ with the axiom $e(x) = \top$, while SP_2 contains the same operation with the axiom $e(x) = \bot$, then taking the sum of both specifications will yield the empty class of models (unless there are models satisfying $\top = \bot$).

If such conflicts may occur or, more generally, if we do *not* want to identify some operations with the same names, we have to design a new signature with two copies, e.g., $e_1, e_2 : \mathsf{B} \to \mathsf{B}$ and specify in the signature morphisms that, for instance, $\iota_1(e) = e_1$ and $\iota_2(e) = e_2$.

Exercises (week 9)

EXERCISE 9.1 Write the full specification-expression for the specification SLIST from example 9.8. (I.e., make explicit the nesting of various specification building expressions.)

EXERCISE 9.2 Let SP be a specification with the signature $\Sigma = \langle A, B; a : \to A, b : B \to B\rangle$. Use this specification and appropriate specification building operations (but not **enrich**!) to obtain a structured specification with the signature $\Omega = \langle A, A_1, B; a : \to A, a_1 : \to A_1, b : B \to B\rangle$. (State the possible signature morphisms explicitly.)

EXERCISE 9.3 Let SP have signature $\Sigma_N = \langle \mathsf{N}; 0, \mathsf{s}, +, *\rangle$ and appropriate axiomatization of these operations. Build from SP a structured specification over the signature $\Sigma = \langle \mathsf{N}; 0, \mathsf{s}, ** : \mathsf{N} \to \mathsf{N}\rangle$, where $**$ is the square function, i.e., write the required specification expression adding and axiomatizing the new operation and hiding the undesired operations. Indicate explicitly the signature morphisms.

EXERCISE 9.4 Combine the specifications BINT from exercise 8.2 and LIST from example 9.1 in order to obtain a specification with the operation $in : BT \to List$ which traverses a binary tree and constructs the inorder list of its nodes. (For instance, the list returned by in applied to the last tree in exercise 8.2, should be $\langle 5, 7, 6, nil\rangle$.)

Write an appropriate structured specification with the signature which is the union of the two signatures extended with the in operation. (To axiomatize in you will probably need an additional operation – but this auxiliary operation shall not be available in the final specification. You may also recall remark 1.16.)

EXERCISE 9.5 Let SFA stand for a possible enrichment of a specification, i.e., $\mathcal{S} : S; \Omega : F; \mathcal{A} : A$. Show that the following specifications are equal (i.e., have the same signature and model classes – you may consult exercise 4.6):

$$\textbf{enrich } \lceil\textbf{enrich } SP \textbf{ by } SFA'\rfloor \textbf{ by } SFA'' \;=\; \textbf{enrich } SP \textbf{ by } SFA' \cup SFA''$$

Does this imply that also the following equality holds

$$\textbf{enrich } \lceil\textbf{enrich } SP \textbf{ by } SFA'\rfloor \textbf{ by } SFA'' \;=\; \textbf{enrich } \lceil\textbf{enrich } SP \textbf{ by } SFA''\rfloor \textbf{ by } SFA' \;\;?$$

If not, then what conditions must be satisfied for this equality to hold?

EXERCISE 9.6 Refer to example 9.9. What is the difference between **reach**(STACKE) $\ominus [\Omega : \bullet]$ and **reach**(STACKE $\ominus [\Omega : \bullet]$)? Characterize a model of the former specification which is *not* a model of the latter.

Assuming that in the specification INT we have the operations $0, \mathsf{s}, \mathsf{p}$ returning integers Z, will there be any difference between the two specifications $\textbf{reach}_{\{0,\mathsf{s},\mathsf{p}\}}$(STACKE) $\ominus [\Omega : \bullet]$ and $\textbf{reach}_{\{0,\mathsf{s},\mathsf{p}\}}$(STACKE $\ominus [\Omega : \bullet]$)?

EXERCISE 9.7 Verify that the union operator (9.16) is idempotent, commutative and associative.

EXERCISE 9.8 Verify the equality (9.32) claimed in the definition (9.18) of the **sum**-operator.

Week 10: Implementation

- DATA REFINEMENT
- CONSTRUCTOR IMPLEMENTATIONS
- VERIFICATION

The specification building operations studied in the last section serve the purpose of structuring the *text* of specification. They allow one to split a specification into smaller, and hence more manegable pieces, and then to combine such pieces into a whole. This whole denotes then a class of algebras which is determined by the semantics of the parts as well as the operations used to combine them.

This structuring may be called the "horizontal" aspect of specification – it arranges various pieces into one whole. By and for itself it does not have any direct bearing on the notion of implementation, i.e., the "vertical" aspect, as we defined it in (8.23) and (8.24). Nevertheless, as illustrated for instance in example 8.3, these structuring mechanisms will be typically used in order to achieve a smooth chain of model-class inclusions. In this section we will study some more specific – and useful – aspects of the generic notion of implementation as described in (8.23) and their interaction with the specification building operations. In subsection 10.3 we will also consider formal verification of correctness of implementation.

10.1: DATA REFINEMENT

One of the first algebraic notions of implementation concerned the relation between algebras and was formulated as the requirement of the existence of a (surjective) homomorphism from the implementing to the implemented algebra. Such a requirement is the same as requiring the existence of a congruence on the implementing algebra which corresponds to the (implementation of the) equality in the implemented algebra. That is, the concrete (implementing) data type may have several elements which are equivalent representatives – constitute one equivalence class – of one abstract (implemented) value.

In terms of specifications – that is, *classes* of algebras – this notion corresponds to the statement that we can design a homomorphism from each model of the implementing specification to some model of the implemented one. This is typically achieved by augmenting the implementing specification with some equalities which induce a congruence on each model and such that the quotient by this congruence yields a model of the implemented specifcation.

Example 10.1 [Sets by Lists]

Consider the following specification of sets (of integers).

$$\begin{array}{rl}
\textsc{Set} = & \textsc{Int} \oplus \textsc{Bool} \oplus \\
\mathcal{S}: & Set \\
\Omega: & \begin{array}{rl} \emptyset: & \to Set \\ add: & \mathsf{Z} \times Set \to Set \\ eq: & Set \times Set \to \mathsf{B} \end{array} \\
\Phi_{Set}: & \begin{array}{rcl} eq(add(x, add(y, S)), add(y, add(x, S))) & = & \top \\ eq(add(x, add(x, S)), add(x, S)) & = & \top \end{array}
\end{array}$$

The *eq* axioms in SET make the order and number of occurrences of an element in a set irrelevant. Now, recall the specification LIST from example 9.1, in particular, the operations *nil* generating an empty list and *cons* prepending an element to a list. These two operations may be used to implement the above SET. The only difficulty is that the axioms 4. and 5. specified equality on *List* which made, for instance, $cons(x, cons(x, L)) \neq cons(x, L)$ and $cons(x, cons(y, L)) \neq cons(y, cons(x, L))$. To actually implement SET by LIST we have to impose additional equalities corresponding to the *eq* axioms. So let $\equiv$ be the congruence induced on *List* by the axioms

$$\begin{array}{rcl} cons(x, cons(y, L)) & \equiv & cons(y, cons(x, L)) \\ cons(x, cons(x, L)) & \equiv & cons(x, L) \end{array} \qquad (10.33)$$

Then the quotient of any model of LIST will satisfy the axioms of SET. Notice that when an $A \in [\![\textsc{List}]\!]$ then in $A/_{\equiv}$ we will have, for instance, $cons(x, cons(x, nil)) \neq cons(x, nil)$ but $cons(x, cons(x, nil)) \equiv cons(x, nil)$, i.e., these two lists, although different, will belong to the same equaivalence class – they will represent the same abstract set value $\{x\}$.

In order to obtain a proper implementation of SET we have to hide the redundant parts of LIST (as well as rename the remaining ones). We have

$$\textsc{Set} \longrightarrow \lceil \textbf{quotient } \lceil \textbf{derive from } \textsc{List} \textbf{ by } \sigma \rfloor \textbf{ by } \equiv' \rfloor \qquad (10.34)$$

where $\sigma = \{Set \mapsto List, \emptyset \mapsto nil, add \mapsto cons\}$, and $\equiv'$ is obtained from $\equiv$ (10.33) by obvious renaming.

Remark 10.2 [Forget-Identify vs. Identify-Forget]

Notice that we have to perform **derive** *before* **quotient**, i.e., we have to first forget (redundant operations) before we identify some elements under quotient. This gave the rise to the name for such an implementation construction "Forget-Identify", or just "FI". The point is that the identification step takes a quotient under a congruence. If we did not hide other operations before inducing the congruence from the axioms (10.33), we would obtain some weird results.

For instance, $\equiv$ would have to be a congruence also with respect to the *head* operation. This was defined (example 9.1) by $head(cons(x, L)) = x$. Now the axioms (10.33) state that $cons(x, cons(y, L)) \equiv cons(y, cons(x, L))$. So if $\equiv$ is to be a congruence, this would require $x = head(cons(x, cons(y, L))) \equiv head(cons(y, cons(x, L))) = y$, i.e., some further identifications like here $x \equiv y$. This certainly is not our intention.

Thus, the example illustrates the important fact that, in general,

$$\lceil \textbf{quotient } \lceil \textbf{derive from } SP \textbf{ by } \sigma \rfloor \textbf{ by } \equiv' \rfloor \quad \neq \quad \lceil \textbf{derive from } \lceil \textbf{quotient } SP \textbf{ by } \equiv \rfloor \textbf{ by } \sigma \rfloor$$

This form of implementation (using quotient construction) corresponds to *data refinement* in the sense that abstract data elements are refined in the implementation to several, more concrete data elements (each set is represented by various lists). Since quotient construction requires a congruence relation, we have the following well-know diagram expressing this relation:

$$\begin{array}{ccc} \text{implemented} & \qquad & \begin{array}{ccc} X & \xrightarrow{F} & Y \\ \uparrow & & \uparrow \\ {[x]} & \xrightarrow{f} & {[y]} \end{array} \\ \nu \uparrow & & \\ \text{implementing} & & \end{array}$$

ν is the *abstraction function* which is just the natural homomorphism from a model of the implementing specification to its quotient which is a model of the implemented specification. The congruence claim amounts here to saying that: whenever an abstract value X is represented by a set of equivalent concrete values x, then function f implementing F when applied to any concrete representative x must return a concrete y representing the abstract result Y of applying F to X. This was the point violated when we used wrong order IF instead of FI in remark 10.2: the operation *head* applied to two $\equiv$-equivalent lists returned non-equivalent values.

10.2: CONSTRUCTOR IMPLEMENTATION

We will now study a special kind of the general notion of implementation – the so called *constructor implementation* of which we have already seen some examples.[6] For instance, the implementation of sets by lists from example 10.1 was a constructor implementation. To begin with let's look at the following example.

[6]Constructor *implementation* has nothing to do with the constructor *specifications* which were discussed in subsection 8.3.

Example 10.3 [10.1 continued]

The quotient by the congruence $\equiv$ induced on *List* by the equations (10.33) was, in a sense, an "external" construction performed on each model of List. In particular, it did not belong to the specification List. But one may have a justified feeling that, given an implementation of List we could construct an implementation of Set if only we implemented this congruence. In other words, whenever we wanted to check whether two sets are equal under *eq*, we would call the operation $\equiv$ which implements *eq* when *Set* is represented (implemented) by *List*.

This can be done very simply. Just add a new Boolean operation to the specification List, and impose the axioms ensuring that it will be a congruence wrt. to the operations left after hidding *head*, *tail* and $\in$:

$$\begin{array}{rl}
\text{ListS} = & \text{List} \oplus \\
\Omega: & \equiv :\ List \times List \to \mathsf{B} \\
Fs: & \begin{array}{rcl}
cons(x, cons(y, L)) \equiv cons(y, cons(x, L)) & = & \top \\
cons(x, cons(x, L)) \equiv cons(x, L) & = & \top \\
L \equiv L & = & \top \\
(L \equiv L') = \top & \Rightarrow & (L' \equiv L) = \top \\
(L \equiv L') = \top \wedge (L' \equiv L'') = \top & \Rightarrow & (L \equiv L'') = \top \\
(L \equiv L') = \top & \Rightarrow & (cons(x, L) \equiv cons(x, L')) = \top
\end{array}
\end{array}$$

Notice that an implementation of ListS will require implementation of the three operations from its signature, in particular, of $\equiv$. Notice also that since we are here adding explicit congruence axioms for $\equiv$, we do not run into the problem discussed in remark 10.2 – we just do not require $\equiv$ to be a congruence wrt. *head* or *tail*.

We now have the obvious implementation

$$\text{Set} \longrightarrow \lceil \textbf{derive from } \text{ListS } \textbf{by } \sigma \rfloor \tag{10.35}$$

where σ is the obvious renaming (and forgetting) of ListS-operations to Set-operations (i.e., $\sigma = \{Set \mapsto List, \emptyset \mapsto nil, add \mapsto cons, eq \mapsto \equiv\}$).

One difference between the implementation from example 10.1 and this example 10.3 is of the same subtle kind as the difference between **quotient** and **enrich** (example 9.15). In 10.1 we implement the *eq* operation on sets by *performing a construction* on the models of List; in 10.3 we "internalize" this construction including it into the implementing specification.

Thus the other difference between the two implementations is that the implementing specification ListS from (10.35) which is renamed, is "more complex" than the implementing specification from (10.34) on which we take quotient ("implemeting" refers here to the argument of the outermost specification building operator on the right-hand-side of $\longrightarrow$). To make it entirely clear, the following two specifications correspond to the implementing specifications from, respectively, (10.34) and (10.35).

$$\begin{array}{lll}
\text{Set} & \longrightarrow & \textbf{quotient } \lceil \textbf{derive from } \text{List } \textbf{by } \sigma' \rfloor \textbf{ by } \equiv' \qquad (10.34) \\
\text{Set} & \longrightarrow & \textbf{derive from } \lceil \textbf{enrich } \text{List } \textbf{by } \Omega := \equiv, \mathcal{A} : Fs \rfloor \textbf{ by } \sigma \qquad (10.35)
\end{array}$$

Here $\equiv'$ is $\equiv$ renamed appropriately and σ' is σ without the component mapping $eq \mapsto \equiv$. Obviously, the specification List'$=\lceil$**derive from** List **by** $\sigma'\rfloor$ has fewer operations and axioms (none of Fs) than ListS. Instead, these additional information has been burried in the **quotient** construction (with the understanding that the induced congruence $\equiv'$ corresponds to the operation *eq*).

The question may now arise: Is it a general goal of implementation process to reduce the complexity of implementing specifications in this way? If the answer is yes, then we may always reduce the whole process to a single step! In our example:

$$\text{Set} \longrightarrow \textbf{enrich } \text{Empty } \textbf{by } \text{Set} \tag{10.36}$$

Empty is the empty (and hence the simplest possible) specification. This shows that it is not entirely correct to call the argument to the outermost specification building operator on the right-hand-side of $\longrightarrow$ an "implementing specification". Applying the schema (10.36) the Empty specification would be implementing any other specification.

The point is that **quotient** is a construction on the models – it does something concrete to each model of its argument specification producing out of it a new structure which is the model of the resulting specification. This construction actually does a concrete job which, in this case, corresponds to refining the specification by additional operations ($\equiv$) and axioms (Fs). The **enrich** does not do anything of the sort – its semantics does not utilize the *structure* of the models of the argument specification but introduces a new class of models. The definitions of (the semantics of) various specification building operations κ from 9.2 can be divided into two kinds: for $\kappa : Spec(\Sigma) \to Spec(\Sigma')$

$$\llbracket \kappa(SP) \rrbracket = \{A \in \mathsf{Alg}(\Sigma') : A \text{ satisfies some conditions}\} \tag{10.37}$$
$$\llbracket \kappa(SP) \rrbracket = \{k(A) : A \in \llbracket SP \rrbracket\} \text{ for some } k : \mathsf{Alg}(\Sigma) \to \mathsf{Alg}(\Sigma') \tag{10.38}$$

The distinction between these two kinds of specification building operators is expressed in the following definition.

Definition 10.4 A specification building operation $\kappa : Spec(\Sigma) \to Spec(\Sigma')$ is a *constructor* iff its semantics is defined as a function on algebras, i.e., $\llbracket \kappa(SP) \rrbracket = \{k_{SP}(A) : A \in \llbracket SP \rrbracket\}$, for some $k_{SP} : \mathsf{Alg}(\Sigma) \to \mathsf{Alg}(\Sigma')$.

Typically, we use the same symbol κ for both the specification building operator and the corresponding functions on algebras k_{SP}. Notice that we indexed (somehow imprecisely) k's by SP. This is so because, for instance, **derive** in itself is not a uniquely defined operation – it depends on the argument specification and the actual signature morphism. That is, for each signature Σ and each possible morphism $\sigma : \Sigma' \to \Sigma$, we will have another variant of the reduct construction $\_|_\sigma : \mathsf{Alg}(\Sigma') \to \mathsf{Alg}(\Sigma)$. If a specification building operation κ is a constructor, one usually writes $SP \xrightarrow{\kappa} SP'$ instead of $SP \longrightarrow \kappa(SP')$ to indicate that SP is implemented by applying the construction κ to specification SP.

For instance, the semantics of **quotient** is defined as such a construction $\llbracket \textbf{quotient } SP \textbf{ by } \equiv \rrbracket = \{A/_{\equiv} : A \in \llbracket SP \rrbracket\}$, while the semantics of **enrich** is not. Similarly, the semantics of **derive** is defined by the reduct construction, $\llbracket \textbf{derive from } SP \textbf{ by } \sigma \rrbracket = \{A|_\sigma : A \in \llbracket SP \rrbracket\}$, so **derive** is a constructor. Since **rename** and **hide** were special cases of **derive** they are constructors too. Finally, also the **reach** operation is a constructor. The rest of the operations we have introduced are not constructors.

Remark 10.5 [**reach** is a constructor!]

In the definition of the semantics of **reach** (9.6 on p. 83) we refered to the definition 8.5, according to which we have that

$$\llbracket \textbf{reach}_C(SP) \rrbracket \overset{(9.6)}{=} Gen_C(SP) \overset{8.5}{=} \{G \in Alg(SP) : G \text{ is generated from } C \subseteq \Sigma\} \tag{10.39}$$

This does not at all look like a constructor definition – in fact, it has the form of (10.37). Nevertheless, it can be expressed also in the form of (10.38). For the sake of simplicity, let us consider the case of $Gen(SP)$ – the general case for $Gen_C(SP)$ is analogous. To get a constructor definition for **reach**, we have to make some additional assumptions.

Firstly, even if $\mathsf{Alg}(SP) \neq \emptyset$, (10.39) may yield an empty class – namely, if Σ is void. Constructor operators, on the other hand, never yield an empty class from a non-empty one. But there seems to be no practical reason to admit this kind of generality. Restricting the model class to generated models, we should be able to assume this class to be non-empty. Then for non-void Σ we can define an operation $gen : \mathsf{Alg}(\Sigma) \to \mathsf{Alg}(\Sigma)$ on Σ-algebras which, for any A yields its minmal subalgebra

$$\llbracket \textbf{reach}(SP) \rrbracket \overset{\text{def}}{=} \{gen(A) : A \in \llbracket SP \rrbracket\} \tag{10.40}$$

By corollary 2.13, if Σ is non-void such a minimal subalgebra (is generated and) exists for any $A \in \mathsf{Alg}(\Sigma)$.

Now, the class defined in (10.39) was obviously a subclass of $[\![SP]\!]$. This need not be case for (10.40) – it is possible that $[\![SP]\!]$ is *not* closed under subalgebras and that some of such generated subalgebras are not members of $[\![SP]\!]$. Then they will not be in the class defined as in (10.39) but will be in the one defined as in (10.40). If this is (or may be) the case these two definitons will simply define different specification building operations. However, as we have seen in theorem 7.15, even the most general of the languages (of clauses) we have considered guarantees closure under subalgebras. Thus, as long as we remain within this language, the definition (2.3) will yield a subclass of generated algebras from the argument class $[\![SP]\!]$. This is the usual understanding of the **reach** operator.

The definition 10.4 is very liberal because it allows one to define new kinds of specification building operations by using any kind of constructions on algebras. For instance, we could define a specification building operator π by $[\![\pi(SP)]\!] \stackrel{\text{def}}{=} \{A \times A : A \in [\![SP]\!]\}$ – the construction here is just taking the binary product of each model of the argument specification. Another variant might be a more general constructor taking binary product of two specifications over the same signature, e.g., $[\![\pi(SP_1, SP_2)]\!] \stackrel{\text{def}}{=} \{A_1 \times A_2 : A_1 \in [\![SP_1]\!] \wedge A_2 \in [\![SP_2]\!]\}$.

The intuition behind the concept of a constructor is that since algebras are programs, constructors are functions which given a program (an algebra) as an argument return a new program. There are programming languages which allow one to actually write such program, i.e., to define operations which take one data type as an argument and return a new data type. In the languages where one cannot do that, one should still be able to program a particular application of a constructor.

Example 10.6 [10.1 continued]

Consider the implementation from (10.34)

$$\text{SET} \longrightarrow \lceil \textbf{quotient}\ \lceil \textbf{derive from}\ \text{LIST}\ \textbf{by}\ \sigma \rfloor\ \textbf{by}\ \equiv' \rfloor$$

It is a constructor implementation of SET by LIST since both **quotient** and **derive** are constructors. (It is easy to verify that composition of constructors yields a constructor.) Suppose now that we have obtained an actual program P implementing LIST. You should be able to imagine how to augment P with additional pieces of code which, firstly hide and rename operations in P, and then implement the congruence $\equiv'$.

The first step might amount to just introducing new procedures with the required names. In P we will have, for instance, a '**procedure** `cons(x:`**integer**`, L:List):List`'. In the first step we might add a '**procedure** `add(x:`**integer**`, L:Set):Set = cons(x,L)`', which simply calls the old procedure disguising it under the new name. (Here we have to somehow ensure that we can use the name `Set` instead of the name `List`.)

Having thus obtained a program P' which implements $\lceil$**derive from** LIST **by** $\sigma\rfloor$, we would implement the **quotient** construction by adding a new Boolean '**procedure** `eq(x:Set,y:Set):`**boolean**' which would return `true` iff the sets (i.e., lists) `x` and `y` were equal up to the order and number of occurrences of the same elements.

10.3: Verification

One of the central tenets of formal program development is the attempt to ensure correctness of the resulting software. Methodologically, one postulates *modular* development, i.e., splitting the – typically large and complicated – task into more managable pieces which can be constructed independently and then combined into a desired system. The use of specification building operations and, in particular, of constructor implementations, faciliates such a divide-and-conquer develompment strategy. In the next section we will see an even more powerful mechanism of parameterization designed for this purpose.

The stepwise refinement process is postulated with the same goal in mind – a high-level, abstract desrciption is typically more intuitive and less error-prone than a detailed code. It is easier to verify correctness of a more concise, shorter description of the desired properties than of a large, detailed program. Proceeding carefully from such a description towards a concrete code, although does not necessarily eliminate all possible errors, significantly increases the chances of obaining a correct final product. This, however, presupposes that each refinement step *preserves correctness* – we want to be sure that having obtained a satisfactory abstract specification which captures intended features in a correct way, the development process, i.e., each refinement step, will yield a correct implementation of the specification obtained at the previous stage. In some cases this can be ensured automatically by applying various forms of program- or specification-transformations. In general, however, one has to verify that the refinement step preserves the properties of the refined specification according to (8.23), i.e., to verify the semantic relation

$$[\![SP_2]\!] \;\subseteq\; [\![SP_1]\!]. \tag{10.41}$$

Verification means that this relation has to be somehow checked at the syntactic level. Formal verification of correctness of a development (step) as in (10.41) amounts to *proving* that the resulting specification entails the required properties, i.e., the properties which ensure that its models are among the models of the refined specification. Ideally, verifying that for all formulae

$$\forall \phi \;:\; [\![SP_1]\!] \models \phi \;\Rightarrow\; [\![SP_2]\!] \models \phi \tag{10.42}$$

would entail the conclusion (10.41). This, obviously, is impossible in practice since there will be typically infinitely many ϕ's of this kind. For the simple (flat) specifications, i.e., of the form $\langle \Sigma, \Phi\rangle$, the relation (10.42) can be verified by checking that

$$\forall \phi \in \Phi_1 \;:\; \Phi_2 \vdash_C \phi. \tag{10.43}$$

But this is necessary only if $[\![SP_2]\!]$ is the class of *all* models of Φ_2: it will ensure that this class – $[\![SP_2]\!] = \mathsf{Alg}(\Phi_2)$ – is included in $\mathsf{Alg}(\Phi_1) = [\![SP_1]\!]$. However, as we have seen, e.g., in example 8.10, it may happen that we cannot verify (10.43) simply because (10.41) does not hold (cf. (8.26)). In the example we had to take only generated models – we had to use induction principle to verify (8.27), i.e., we used not merely calculus C but IC (actually, it was IEQ) to prove (10.43). That is, if $[\![SP_2]\!]$ is only the class of generated, or initial models, verification will often require use of some form of induction.

A real problem arises when also $[\![SP_1]\!]$ comprises only generated models. Then verification of (10.43) certainly does not suffice, because the generatedness constraint on SP_1 implies validity in the class $[\![SP_1]\!]$ of many additional formulae which are not among the axioms Φ_1 (but follow from them only by induction). These observations indicate the rule of thumb that generatedness/initiality constraints should be used, preferably, only at the very last stage(s) of development. We will focus only on the simplified situation where the implemented specification SP_1 is a full model class and does not contain any generatedness/initiality constraints, i.e., when it suffices to show (10.43) – possibly using induction – in order to verify correctness of the implementation.

10.3.1: Calculus of Structured Specifications

Formal proofs of this kind have two natural aspects. The one is concerned with deriving the consequences of a given set of axioms, i.e., with reasoning within the language in which the axioms are written. The axioms of a *flat* specification $SP = \langle \Sigma; \Phi\rangle$ may belong to various languages (some of which were discussed in section 5) and in order to see whether $SP \vdash \phi$ we have to verify whether $\Phi \vdash_C \phi$, where C is the calculus associated with the language of Φ.

The other aspect is concerned with deriving the consequences of specifications *built in a structured way*. For instance, given a specification SP and its consequences, one will be interested in deriving the consequences of the specification $SP' = \lceil \textbf{enrich}\ SP\ \textbf{by}\ \mathcal{S} : S; \Omega : F; \mathcal{A} : \Psi \rfloor$, if such an SP' appears somewhere during the development process. Ideally, a proof system for structured specifications should be ***compositional***, i.e., it should be possible to obtain the consequences of a structured specification from the consequences of its components. Here, knowing the consequences of SP and of Ψ, one would like to be able to find *all* the consequences of SP'. In general, this is not possible, and except for some special situations, one does not have a complete compositional proof system for structured specifications. The least necessary requirement is thus that such a system must be sound.

The following is a sound system $S[C]$ for the structuring operations **enrich** and **derive** relative to a sound system C for the underlying language for flat specifications

IV. Calculus of Structured Specifications :

$$\text{(IV.1)}\qquad \frac{\phi \in \Phi}{\langle \Sigma; \Phi \rangle \vdash \phi}$$

$$\text{(IV.2)}\qquad \frac{\phi \in \Phi}{\lceil \textbf{enrich } SP \textbf{ by } \mathcal{S} : S; \Omega : F; \mathcal{A} : \Phi \rfloor \vdash \phi}$$

$$\text{(IV.3)}\qquad \frac{SP \vdash \phi}{\lceil \textbf{enrich } SP \textbf{ by } \mathcal{S} : S; \Omega : F; \mathcal{A} : \Phi \rfloor \vdash \phi}$$

$$\text{(IV.4)}\qquad \frac{SP \vdash \sigma(\phi')}{\lceil \textbf{derive from } SP \textbf{ by } \sigma \rfloor \vdash \phi'} \qquad \text{for } \sigma : \Sigma' \to \Sigma \text{ and } \phi' \text{ a } \Sigma'\text{-formula.}$$

$$\text{(IV.5)}\qquad \frac{SP \vdash \phi_1 \ \ldots\ SP \vdash \phi_n \quad \{\phi_1, \ldots, \phi_n\} \vdash_C \phi}{SP \vdash \phi}$$

This calculus is "compositional" in the sense that all the rules are formulated without "looking inside" the structure of the argument specification. E.g., the rule (IV.4), can be applied to any specification with **derive** as the outermost operator, no matter what the structure of the argument SP is. Another way of expressing this "compositionality" is to say that the proof does not require to change the structure of specification – a non-compositional calculus would be, for instance, a one which assumed that each specification has some special, "normal form", and provided only rules for such forms. Transformation of a specification into such a "normal form" would be another task.

The first rule (IV.1) is the obvious basis for proofs of properties of flat specifications. The rule (IV.2) is analogous for enriched specifications.

The following two rules derive consequences of the structured specifications by deriving some consequences of the simpler argument specifications. Rule (IV.3) includes all the consequences of the argument specification into the set of the consequences of the enriched specification. According to the rule (IV.4), a Σ'-formula ϕ' is a consequence of the specification $\lceil$**derive from** SP **by** $\sigma\rfloor$ (which is over Σ') if the Σ-formula $\sigma(\phi')$, obtained by translating ϕ' with the obvious extension of the signature morphism σ, is a consequence of the original specification SP. In the case when **derive** is actually **hide**, i.e., σ is an inclusion, this rule will not involve any translation and its premise will be simply $SP \vdash \phi'$ with the side condition that ϕ' is over the smaller signature Σ'.

Finally, the rule (IV.5) combines the two levels of proof. The first n premises are obtained from the rules (IV.1) through (IV.4). The last premise is obtained using the appropriate calculus C for the language in which axioms are written. Thus the rules (IV.1) and this rule (IV.5) together imply that all the C-provable consequences of the axioms of a flat specification are actually provable in this extended system, i.e.,

$$\langle \Sigma; \Phi \rangle \vdash_{S[C]} \phi \quad \Leftrightarrow \quad \Phi \vdash_C \phi$$

Rule (IV.5) combined with the enrich-rules amounts to a kind of "flattening" of the set of all axioms. We have that

$$\lceil \ldots \lceil \langle \Sigma; \Phi \rangle \ \oplus\ SF_1; \mathcal{A} : \Phi_1 \rfloor \ldots \ \oplus\ SF_n; \mathcal{A} : \Phi_n \rfloor \vdash_{S[C]} \phi \quad \Leftrightarrow \quad \Phi \cup \Phi_1 \cup \ldots \cup \Phi_n \vdash_C \phi \qquad (10.44)$$

(Recall the corresponding semantic equivalence from exercise 9.5.) Although this means that, in general, such a flattening may be necessary, one may still use some heuristics to improve the proof search. For instance, if the formula ϕ contains only the symbols from Σ (but none from the added SF's) it will be natural to attempt proving $\Phi \vdash_C \phi$.

Example 10.7 [9.8 continued]

We check that the specification of sorted lists $\textsc{SList} = \textsc{SList1} \ominus [\Omega : \mathit{sorted}]$ satisfies some expected properties. Recall from example 9.1 that $\textsc{SList1}$ had the form $\lceil \textsc{List} \ \oplus\ S\rfloor$, where S contained the profiles and axioms of the *sort*-operation and *sorted*-predicate, and $\textsc{List} = \lceil \textsc{Int} \oplus \textsc{Bool} \ \oplus\ L\rfloor$ where L contained all the list operations and axioms.

For simplicity, let us write $\langle a_1, a_2, \ldots a_n, L\rangle$ for $cons(a_1, cons(a_2, cons(\ldots cons(a_n, L)\ldots)))$. As a trivial example, we show that $\textsc{SList} \vdash 2 \in \langle 0, 2, L\rangle$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{axiom\ of\ \textsc{List}}{\textsc{List} \vdash x \in \langle x, L\rangle = \top}\text{(IV.1)} \qquad \{x \in \langle x, L\rangle = \top\} \vdash 2 \in \langle 2, L\rangle = \top}{\textsc{List} \vdash 2 \in \langle 2, L\rangle = \top}\text{(IV.5)} \qquad (2) \qquad (3)}{\textsc{List} \vdash 2 \in \langle 0, 2, L\rangle = \top}\text{(IV.5)}}{\textsc{SList1} \vdash 2 \in \langle 0, 2, L\rangle = \top}\text{(IV.3)}}{\textsc{SList} \vdash 2 \in \langle 0, 2, L\rangle = \top}\text{(IV.4)}$$

where (2) is the following simple derivation:

$$\dfrac{axiom\ of\ \textsc{List}}{\textsc{List} \vdash x \neq y \Rightarrow x \in \langle y, L\rangle = x \in L}\text{(IV.1)}$$

and (3) the following entailment:

$$\{2 \in \langle 2, L\rangle = \top,\ 0 \neq 2,\ x \neq y \Rightarrow x \in \langle y, L\rangle = x \in L\} \vdash 2 \in \langle 0, 2, L\rangle = \top$$

Notice that the property we are proving here is essentially a property of $\textsc{List}$. This is reflected in the proof which uses the axioms of $\textsc{List}$ to establish the required conclusion which is then carried over by the two final applications of (IV.3) and (IV.4).

As we have stated before, the minimum requirement on the calculus of structured specifications is its soundness.

Theorem 10.8 Let C be a sound calculus for some fixed language $\mathcal{L}$ such that all the axioms of the specifications are $\mathcal{L}$-formulae. Then for any $\mathcal{L}$-formula ϕ and any specification SP, if $SP \vdash_{S[C]} \phi$ then $[\![SP]\!] \models \phi$.

PROOF [IDEA]. Soundness of rules (IV.1) and (IV.2) follows directly from the definition of the semantics of flat specifications (9.3), p. 82, and of enrichment (9.4).

For rule (IV.3), assume (by induction) that $SP \vdash \phi \ \Rightarrow\ [\![SP]\!] \models \phi$, in particular, ϕ is a formula over $Sig(SP)$. For any $A \in [\![SP']\!] = [\![SP \ \oplus\ S', F', \Psi]\!]$, definition (9.4) implies that $A|_{Sig(SP)} \in [\![SP]\!]$, in particular, $A|_{Sig(SP)} \models \phi$. Since the signature of SP is a subsignature of SP', this reduct is simply the part of A without interpretation of all the symbols from S', F'. But since ϕ is a $Sig(SP)$-formula, interpretation of these new symbols does not affect its validity, i.e., $A \models \phi \Leftrightarrow A|_{Sig(SP)} \models \phi$.

For rule (IV.4), let $\phi = \sigma(\phi')$, $SP' = \lceil$**derive from** SP **by** $\sigma\rfloor$, and $\Sigma' = Sig(SP')$. Assuming (by induction) that $SP \vdash_{S[C]} \phi \ \Rightarrow\ [\![SP]\!] \models \phi$, we have to show $[\![SP']\!] \models \phi'$. So let $A' \in [\![SP']\!]$, i.e., by definition (9.5) of **derive**, there is an $A \in [\![SP]\!]$ such that $A' = A|_\sigma$ (and $A \models \phi$). Let $\alpha : X \to A'$ be an arbitrary assignment of elements from A' to the free variables in ϕ'. By definition 4.28 of reduct, this can be seem as an assignment $\alpha : X \to A$ (since each carrier of A' is actually a carrier of A). We then have that $A \models_\alpha \sigma(\phi)$ and, since for each Σ'-symbol $f : f^{A|_\sigma} = \sigma(f)^A$, that $A|_\sigma \models_\alpha \phi$. α was arbitrary and so the claim follows.

(As an example, let ϕ' be $f(x) = g(x)$ with f, g being of sort S and $\sigma = \{S \mapsto T, f \mapsto m, g \mapsto n\}$. Then $\sigma(\phi')$ is $m(x) = n(x)$ and, by assumption, for any $\alpha : X \to A$ we have $m^A(\alpha(x)) = n^A(\alpha(x))$. Now, $A|_\sigma \models f(x) = g(x)$ iff, for any $\alpha : X \to A|_\sigma$, $f^{A|_\sigma}(\alpha(x)) = g^{A|_\sigma}(\alpha(x))$. By definition of reduct $(A|_\sigma)_S = A_T$, and so $f^{A|_\sigma}(\alpha(x)) \stackrel{\text{def}}{=} m^A(\alpha(x)) \stackrel{\text{ass.}}{=} n^A(\alpha(x)) \stackrel{\text{def}}{=} g^{A|_\sigma}(\alpha(x))$.)

Soundness of the last rule follows from soundness of the underlying calculus C. For if, by assumption $[\![SP]\!] \models \phi_i$ for $1 \le i \le n$, and C is sound, then $\{\phi_1, \ldots, \phi_n\} \vdash_C \phi$ implies that $[\![SP]\!] \models \phi$.

Example 10.9 [9.1 continued]

In remark 9.2, we observed that SList1 $\not\models sort(\langle 2,1,1\rangle) = \langle 1,1,2\rangle$ and SList1 $\not\models sort(\langle 2,1,1\rangle) = \langle 1,2\rangle$. Soundness of our calculus implies then that neither of these formulae is provable, i.e., SList1 $\not\vdash sort(\langle 2,1,1\rangle) = \langle 1,1,2\rangle$ and SList1 $\not\vdash sort(\langle 2,1,1\rangle) = \langle 1,2\rangle$.

Now, the only way we can obtain the consequences of the specification SList = SList1 $\ominus[\Omega : sorted]$, is by means of the rule (IV.4), i.e., by deriving consequences of SList1. Thus, we may conclude that these formulae are not derivable for this specification either. In fact, the "strange" kinds of models indicated in remark 9.2 will be among the models of SList.

Finally, we should mention the possible rule for the $\mathbf{reach}_C$ construct. As discussed in section 8, the logical counterpart of the $Gen_C(SP)$ semantics is the ω_C-rule. Thus, we may use the following rule:

$$\text{(IV.6)} \quad \frac{SP \vdash_{\omega_C} \phi}{\mathbf{reach}_C(SP) \vdash \phi}$$

This rule does not compose with the others in a sound way and great care must be taken when using it in a general way. The problem is that its conclusions do not necessarily carry over to the higher (outer) levels of a specification which contains a reachable subcomponent

Example 10.10

Let $\Sigma = \langle T; a, b : \to T\rangle$ and $SP = \mathbf{reach}_{\{a,b\}}(\Sigma)$. Then the corresponding $\omega_{\{a,b\}}$-rule allows us to prove $SP \vdash x = a \vee x = b$.

Take now $SP' = \mathbf{enrich}\ SP\ \mathbf{by}\ \Omega : c : \to T; \mathcal{A} : \{c \neq a, c \neq b\}$. Applying the rule (IV.3) we would conclude $SP' \vdash x = a \vee x = b$ which, obviously, does not hold since SP' extends SP exactly in the way violating this conclusion.

Thus, as a general rule of thumb, one may postulate that this rule can be used *only at the outermost level* of specification. This postulate corresponds exactly to the earlier one (discussed under equation (10.43)), namely, that generatedness constraints – being much less tractable than the usual semantic consequence – should be introduced only at the very last stage(s) of development.

Example 10.11 [10.9 continued]

Specification SList does not entail, for instance, the formula $head(sort(\langle 2,1,1,nil\rangle)) = 1$. The reason is analogous to this from example 8.6 – the axioms (in List) tell something about *head* of a *cons* list, but do not force the result of *sort* to be of this form. Similarly, the *sorted* predicate in SList1 is defined only for the lists of form *cons*.

Since this may seem a bit unfortunate, one might try to fix the problem by writing the axioms for *sorted* in a different way, for instance, replacing the two axioms by the following one:

$$sorted(l) = \top \;\Leftrightarrow\; [l = nil \;\vee\; tail(l) = nil \;\vee\; (head(l) \le head(tail(l)) = \top \wedge sorted(tail(l)) = \top)]$$

However, even this does not entail the desired conclusion, since *head* is only defined for *cons*-lists and we have no means to conclude that $sort(l)$ yields such a list. As before, at some point in the development we may have arrived at the specification of all the operations of interest and decide which among them are to be the constructors – here *nil* and *cons*. We take this step by saying

$$\text{Sort} = \mathbf{reach}_{\{nil,cons\}}(\text{SList}) \tag{10.45}$$

The corresponding induction principle will now make it possible to prove the desired formula. The full formal proof would be rather complicated but informally, the induction principle allows us to assume that $sort(\langle 2,1,1,nil\rangle)$ has the form $cons(x, L)$ and so, by axiom 1. from List, $head(sort(\langle 2,1,1,nil\rangle)) = x$. By the last axiom from SList1, this x must be either 1 or 2, and so by the second axiom it must be 1.

10.3.2: Summary

The objective of verification technique is to provide a syntactic means for proving correctness of an implementation step $SP_1 \longrightarrow SP_2$, i.e., showing that the relation (10.41) holds. The general proof obligation, as expressed in (10.42), is impractical. We therefore try to indicate more specific proof obligations, by specifying such a (finite) set of formulae Φ, that verifying

$$[\![SP_2]\!] \models \Phi \tag{10.46}$$

will suffice for concluding (10.41). We also indicate syntactic means, i.e., formulae and calculus, which can be used in verifying the relation (10.46). We have the following special cases:

1. $SP_1 = \langle\Sigma_1, \Phi_1\rangle$ and $SP_2 = \langle\Sigma_2, \Phi_2\rangle$, i.e., both are flat.
   Then, showing $\Phi_2 \vdash_C \Phi_1$ is sufficient to conclude (10.46).
2. $SP_1 = \langle\Sigma_1, \Phi_1\rangle$ is flat and SP_2 is structured.
   In general, the proof obligation is then to show $SP_2 \vdash_{S[C]} \Phi_1$, i.e., we use the calculus of structured specifications to derive the consequences of SP_2 and to verify that they entail the axioms of SP_1.

   A special case occurs when SP_2 has the form $\mathbf{reach}_F(SP)$. The proof will then, typically, have to apply appropriate induction on F, i.e., we have to show $SP_2 \vdash_{S[IC]} \Phi_1$.
3. SP_1 is structured. We can consider the subcases depending on the outermost specification building operator in SP_2.

3.1 $SP_1 = \mathbf{enrich}\ SP\ \mathbf{by}\ [\mathcal{S} : S, \Omega : F, \mathcal{A} : \Phi]$.
The models of SP_2 have then to satisfy both Φ and (their reducts) to belong to $[\![SP]\!]$. We have to verify that $SP_2 \vdash_{S[C]} \Phi$ and $SP_2 \vdash_{S[C]} \Phi_{SP}$, where Φ_{SP} are the formuale needed to verify the implementation of SP.

3.2 $SP_1 = \mathbf{derive\ from}\ SP\ \mathbf{by}\ \sigma$.
Consider the basic case when **derive** is used merely for hidding (i.e., σ is injective but not surjective). The important point is that the models of SP_1 will, still, satisfy the axioms of SP which involved the hidden symbols. What one has to prove here is that, given any model of SP_2, we can add the operations from SP hidden in SP_1 and then, assuming that they satisfy the axioms from SP, all the other axioms follow. This is trivial if SP_1 hides some operations which are not used in the specification of the non-hidden operations. However, as hidden operations are typically used for defining the non-hidden ones, the proof obligation is non-trivial (e.g., implementation of SList from example 9.8).

So let SF_{hide} be the symbols from SP hidden in SP_1 and Φ_{hide} be the axioms defining only these operations in SP. We have to show that $SP_2 \cup \Phi_{hide} \vdash_{S[C]} \Phi_{SP} \setminus \Phi_{hide}$.

In general, **derive** may also involve renaming: $\sigma : \Sigma_1 \to \Sigma$ is a signature morphism from the signature of SP_1 into SP (which even may identify some symbols from Σ_1). Thus Σ_1 is also the signature of the implementing specification SP_2. The general form of the proof obligation is thus: $\sigma(SP_2) \cup \Phi_{hide} \vdash_{S[C]} \Phi_{SP} \setminus \Phi_{hide}$, where $\sigma(SP_2)$ indicates translation of SP_2's axioms (and possibly all derived consequences) into formulae over Σ.

3.2 $SP_1 = \mathbf{reach}_C(SP)$. This case is, as we have observed, intractable and difficult, since the generatedness constraint implies validity in $[\![SP_2]\!]$ of many formulae which are not directly transparent from the axioms.

(In all these cases, if SP_2 is flat, we will prove $\Phi_2 \vdash_C$ .. instead of $SP_2 \vdash_{S[C]}$ ...)

Exercises (week 10)

EXERCISE 10.1 Design an implementation of STACK1 from example 8.3 by the specification LIST from example 9.1. Find an appropriate specification expression sp which will make STACK1 $\longrightarrow$ sp(LIST). Verify correctness of your implementation by showing that sp(LIST) satisfies all the axioms of STACK1. Is it a constructor implementation?

EXERCISE 10.2 The following specification of arrays with counters corresponds to exercise 7.1.

$$
\begin{array}{lll}
\text{ArC} = & \text{Nat} \oplus \text{Bool} \oplus & \\
\mathcal{S}: & A, A^{\mathsf{N}} & \\
\Omega: & null: & \to A^{\mathsf{N}} \\
& co: & A^{\mathsf{N}} \to \mathsf{N} \\
& ins: & A \times A^{\mathsf{N}} \to A^{\mathsf{N}} \\
& ev: & A^{\mathsf{N}} \times \mathsf{N} \to A \\
\mathcal{A}: & & \\
\end{array}
$$

$$
\begin{array}{rcl}
co(null) & = & 0 \\
co(ins(x,A)) & = & co(A)+1 \\
ev(ins(x,A), co(ins(x,A))) & = & x \\
n < co(ins(x,A)) = \top & \Rightarrow & ev(ins(x,A),n) = ev(A,n)
\end{array}
$$

Recall the specification LIST from example 9.1. Design an appropriate specification expression sp which will yield the implementation LIST $\longrightarrow$ sp(ARC). (You may assume that LIST specifies lists of A-elements, i.e., it does not import the INT component as it did in the example.) Is it a constructor implementation? Show correctness of this implementation by verifying that sp(ARC) satisfies all the axioms of LIST.

(You will need to extend ARC with new operations and axioms implementing *tail*, *head* and $\in$ and determine the appropriate signature morphism renaming and hiding parts of ARC.)

EXERCISE 10.3 Give a proof or a counter-example for each of the following equalities:

1. $\mathbf{reach}_C(SP_1) \cup SP_2 = \mathbf{reach}_C(SP_1 \cup SP_2)$
2. if $C \subseteq C'$ then $\mathbf{reach}_C(\mathbf{reach}_{C'}(SP)) = \mathbf{reach}_C(SP)$.

EXERCISE 10.4 Assume that in example 10.11, instead of constructing SORT from equation (10.45), we apply $\mathbf{reach}_{\{nil,cons\}}$(LIST) and then proceeded as before towards SLIST, i.e., construct the specification

$$\text{SORT'} = \mathbf{hide}\ [\Omega : sorted]\ \mathbf{in}\ \lceil \mathbf{enrich}\ \lceil \mathbf{reach}_{\{nil,cons\}}(\text{LIST}) \rfloor\ \mathbf{by}\ \text{...stuff from SLIST1...} \rfloor$$

Will SORT' $\vdash head(sort(\langle 2,1,1,nil \rangle)) = 1$? And more generally – will the equality SORT = SORT' hold?

Week 11: Parameterization

- Parameterized Specifications
- Implementation of Parameterized Specifications
- Specification of Parameterized Programs

11.1: Parameterized Specifications

Many data types have some "generic" part which would look the same even if the rest of the specification were changed. For instance, the specification List from example 9.1 extends the specification of integers and Booleans with the generic list-operations. These list-operations could be exactly the same if we wanted to specify the lists of natural numbers or any other elements.

Parameterization is the technique of encapsulating a piece of specification (or software) and abstracting from some names or subexpressions in order to replace them in various contexts by different actual parameters. One does not want to repeat the specification of lists each time one needs lists of new elements. Ideally, one should be able to specify lists of arbitrary elements – when lists of some specific elements, e.g., of integers, are needed, one would instantiate this generic specification with the actual specification of integers.

There are various approaches to parameterization of algebraic specifications. In this section we will study only some pragmatic aspects of one simple variant allowing us to modularize the *text* of specifications and enabling its reuse..

Example 11.1 [Parameterized Specification]

Instead of the specification List from example 9.1, we would write a parameterized specification:

$$\begin{array}{rl}
\textsc{List}[X:\textsc{El}] = & \lceil\ \textsc{El} = \mathcal{S} : El\ \rfloor \oplus \\
\mathcal{S}: & List[El] \\
\Omega: & \begin{array}{rrl} nil: & & \to List[El] \\ cons: & El \times List[El] & \to List[El] \\ head: & List[El] & \to El \\ tail: & List[El] & \to List[El] \end{array} \\
\mathcal{A}: & \begin{array}{rcl} head(cons(x,l)) & = & x \\ tail(cons(x,l)) & = & l \\ cons(x,l) & \neq & nil \\ cons(x,l) = cons(x',l') & \Rightarrow & x = x' \wedge l = l' \end{array}
\end{array}$$

The name of the parameterized specification contains the list of formal parameters. Here X : El means that the specification takes one parameter X which is "of type" El. This type is again a specification – it identifies the necessary requirement which any actual parameter must meet. The formal parameter specification follows – here we only require that any actual parameter must have (at least) one sort which will correspond to the sort El. Then we have a usual specification of the body of the parameterized specification. Together, the specification of the formal parameter(s) and of the body constitute a complete specification of a data type.

Notice that we are also using a structured name for the sort of lists $List[El]$. This convention is not necessary but it will be useful when instantiating parameterized specifications.

A bit formally we may say that a parameterized specification expression is a list of specifications $\langle X_1 : SP_1, ..., X_n : SP_n, SP\rangle$ where each formal parameter specification $SP_i \subseteq SP$ is a subspecification of (the whole) SP. Of course, for convenience we adopt a more user-friendly notation as in the above example.

The semantics of a parameterized specification expression like List$[X :$ El$]$ can be defined as a specification building operation $Spec \to Spec$ which, for any actual parameter specification SP : El, returns a new specification List$[SP]$. We will not consider the, rather complicated, details of such a semantics. We choose to look at parameterized specifications as the means of structuring merely the *text* of specifications. We have to consider the mechanism of instantiation – parameter passing. The specification resulting from instantiating all the formal parameters by the actual ones will have the semantics defined in the same way as the specifications we have seen so far.

11.1.1: Actual Parameter Passing

First, any actual parameter must satisfy the formal parameter specification. This is what is ment by the notation X : El and, for an actual parameter, SP : El. In the above example, this boils down to the requirement that the actual parameter SP must have a sort corresponding to El. However, it is dubious whether an actual parameter will have a sort with the name El. Instantiating X to Int, we want to obtain $List[\mathsf{Z}]$ and not $List[El]$. The inteded correspondance is obtained by determining a so called *fitting morphism* – a signature morphism from the signature of the formal to the signature of the actual parameter. Instantiation List[Int] requires that we specify a morphism $\sigma : \langle El; \emptyset\rangle \to \Sigma_{\mathsf{Z}}$ – here it will simply rename the sort $\sigma : El \mapsto \mathsf{Z}$.

Furthermore, a parameterized specification has a signature Σ_P which contains, as a subsignature, the signature Σ_{FP} of the formal parameter. In the example it is

$$\langle El, \emptyset\rangle \ = \ \Sigma_{El} \ \subseteq \ \Sigma_{List} \ = \ \langle El, List[El]; nil, cons, head, tail\rangle.$$

Thus, we have two signature morphisms: the inclusion $\iota : \Sigma_{El} \hookrightarrow \Sigma_{List}$ and the fitting morphism $\sigma : \Sigma_{El} \to \Sigma_{\mathsf{Z}}$. From these two morphisms, one obtains the resulting signature Σ of the instantiated specification List[Int] by taking the union of the signature of the actual parameter Σ_{Z} and the signature Σ_{List} renamed by σ (or more precisely, by the extension $\overline{\sigma}$ of σ which renames appropriately the symbols from Σ_{El} and leaves the other symbols from Σ unchanged). A bit informally, we can say that

$$\Sigma \ = \ \Sigma_{AP} \cup \lceil \textbf{translate } \Sigma_P \textbf{ by } \overline{\sigma} \rfloor$$

and that the resulting specification is obtained by the corresponding equation

$$SP \ = \ SP_{AP} \cup \lceil \textbf{translate } SP_P \textbf{ by } \overline{\sigma} \rfloor \qquad (11.47)$$

Remark 11.2

This is an example of the so called push-out construction illustrated in the diagram below. It identifies the elements in Σ_P and Σ_{AP} originating from Σ_{FP} (i.e., $f \in \Sigma_P$ and $g \in \Sigma_{AP}$ will be identified resulting in g iff there is a $p \in \Sigma_{FP}$ with $f = \iota(p)$ and $g = \sigma(p)$).

$$\begin{tikzcd}
\Sigma_{FP} \arrow[r, hook, "\iota"] \arrow[d, "\sigma"] & \Sigma_P \arrow[d, "\overline{\sigma}"] \\
\Sigma_{AP} \arrow[r, hook, "\overline{\iota}"] & \Sigma
\end{tikzcd}$$

Thus, instantiation of a parameterized specification $\langle X_1 : FP_1, ..., X_n : FP_n, SP\rangle$ has the form $SP[AP_1 \textbf{ by } \sigma_1, ..., AP_n \textbf{ by } \sigma_n]$ where each $\sigma_i : Sig(FP_i) \to Sig(AP_i)$ is a respective fitting morphism. For instance, List[Int **by** σ] will yield the specification of lists of integers which is essentially the same as the specification List from example 9.1 (here without the operation $\in$ and with the sort $List[\mathsf{Z}]$ instead of just $List$).

11.1.2: Formal Parameter with Axioms

If the formal parameter specification FP contains any axioms Φ, when passing an actual parameter, one has to specify the fitting morphism and then verify that these axioms will be satisfied by the actual parameter AP, i.e., for each $\phi \in \Phi$, one has to check that $AP \models \overline{\sigma}(\phi)$.

Example 11.3

Let us specify binary search trees.The parameter must be a total ordering, i.e., it must contain a specification of Booleans and an ordering relation $\prec$. Unlike binary trees (exercise 8.2), BST are constructed using nil and ins, the latter inserting a new element into a tree. The search invariant is specified using the auxiliary predicates $lt(T,x)$ (resp. $gt(T,x)$) stating that all the values in the tree T are smaller (resp. greater) than the value x.

$$
\begin{array}{rl}
\mathrm{BST{\small REE}}[X:\mathrm{T{\small O}}] = & \lceil\ \mathrm{T{\small O}} = \ \mathrm{B{\small OOL}}\ \oplus \\
\mathcal{S}: & El \\
\Omega: & \prec:\ \ El \times El \to \mathsf{B} \\
\mathcal{A}_{To}: & \begin{array}{rcl} & & x \prec x \ = \ \bot \\ x \prec y = \top \wedge y \prec z = \top & \Rightarrow & x \prec z = \top \\ x \prec y = \top & \Rightarrow & y \prec x = \bot \\ x \neq y & \Rightarrow & x \prec y = \top \vee y \prec x = \top\ \rfloor\ \oplus \end{array}
\end{array}
$$

$$
\begin{array}{rrl}
\textbf{hide}\ [lt,\ gt]\ \textbf{in}\ \lceil\ \mathcal{S}: & BST[El] & \\
\Omega: & nil: & \to BST[El] \\
& ins: & El \times BST[El] \to BST[El] \\
& val: & BST[El] \to El \\
& L,R: & BST[El] \to BST[El] \\
& lt,gt: & BST[El] \times El \to \mathsf{B} \\
\mathcal{A}: & 1. & L(nil) = nil \\
& 2. & R(nil) = nil \\
& 3. & val(ins(x,nil)) = x \\
& 4. & lt(nil,x) = \top \\
& 5. & gt(nil,x) = \top \\
& 6. & lt(ins(y,t),x) = \top \Leftrightarrow y \prec x = \top\ \wedge\ lt(t,x) = \top \\
& 7. & gt(ins(y,t),x) = \top \Leftrightarrow y \prec x = \bot\ \wedge\ gt(t,x) = \top \\
& 8. & lt(L(ins(x,t)), val(ins(x,t))) = \top \\
& 9. & gt(R(ins(x,t)), val(ins(x,t))) = \top\ \rfloor
\end{array}
$$

Let now NATL be the following specification of natural numbers with the "less than" operation:

$$
\begin{array}{rlll}
\mathrm{N{\small AT}L} = & \mathrm{B{\small OOL}}\ \oplus & & \\
\mathcal{S}: & \mathsf{N} & \mathcal{A}: & x < x = \bot \\
\Omega: & 0: \ \to \mathsf{N} & & x < 0 = \bot \\
& \mathsf{s}: \ \mathsf{N} \to \mathsf{N} & & 0 < \mathsf{s}x = \top \\
& <: \ \mathsf{N} \times \mathsf{N} \to \mathsf{B} & & \mathsf{s}x < \mathsf{s}y = x < y
\end{array}
$$

Can we instantiate BSTREE$[X:\mathrm{To}]$ with NATL? The obvious fitting morphism $\sigma : \Sigma_{\mathrm{TO}} \to \Sigma_{\mathrm{NATL}}$ would map $El \mapsto \mathsf{N}$ and $\prec\ \mapsto <$. However, the axioms of TO, except for the first one, do not follow from those of NATL! You should convince yourself that they will become valid if we, instead of NATL, take $\mathbf{reach}_{\{0,\mathsf{s}\}}(\mathrm{NATL})$. Passing the parameter as BSTREE[$\mathbf{reach}_{\{0,\mathsf{s}\}}(\mathrm{NATL})$ **by** σ] will be legal and yield the desired specification of binary search trees with natural numbers.

If we considered only fitting morphism, nothing would be wrong with passing NATL as the actual parameter. However, since the specification BSTREE presupposes that its argument satisfies TO-specification, we could not be sure that what we obtain in this way will actually be the intended specification. Therefore, the requirement for the actual parameter is always, in addition to the definition of the fitting morphism σ, satisfaction of the axioms (their translation under σ) of the formal by the actual parameter.

11.1.3: ACTUAL PARAMETER PROTECTION

Parameterization, as we are treating it here, allows one to arbitrarily extend the formal parameter specification. The intention, however, is that it structures the text in an adequate and purposeful manner – the body of a parameterized specification should be a generic specification which does not modify the parameter specification but merely adds the new structure on the top of it. Sometimes, this may be more tricky than expected.

Example 11.4

Following example 5.3, we define a parameterized specification of lists. Specification will extend the parameter sort with a new error element $\bullet_E$ and impose some natural restrictions on this element.

$$
\begin{array}{rrl}
\mathrm{L{\small IST}}[X:\mathrm{E{\small L}}] = & \lceil\ \mathrm{E{\small L}} = \mathcal{S}: El\ \rfloor\ \oplus & \\
\mathcal{S}: & LS[El] & \\
\Omega: & nil: & \to LS[El] \\
& cons: & El \times LS[El] \to LS[El] \\
& head: & LS[El] \to El \\
& tail: & LS[El] \to LS[El] \\
& \bullet: & \to El \\
\mathcal{A}: & 1. & head(cons(x,l)) = x \\
& 2. & head(nil) = \bullet \\
& 3. & tail(cons(x,l)) = l \\
& 4. & cons(\bullet,l) = l
\end{array}
$$

This looks rather innocent. Axioms 1. and 3. are as before, 2. says what $\bullet$ is for and 4. makes it behave "neutrally" wrt. *cons*. As a matter of fact, this is disastrous! What we can now prove is the following:

$$\bullet \stackrel{1}{=} head(cons(\bullet, cons(x,nil))) \stackrel{4}{=} head(cons(x,nil)) \stackrel{1}{=} x$$

That is, for any x of sort El we have the equation $\bullet = x$. Passing as an actual parameter a decent specification of natural numbers, or whatever, will result in a specification where all these numbers collapse to the one element $\bullet$ and the only one list *nil*.

Example 11.5

Introducing an error element, one will often treat it as a "genuine" error, in the sense that, any function applied to it should itself return an error (of appropriate sort). That is, we introduce the *strictness assumption* – all functions are strict in all arguments – and axiomatize it. The following specification of *ST*acks is essentially a specification of lists (with the "stack-like" operation names). We extend the parameter sort with a new error element $\bullet_E$. All operations, when applied to $\bullet_E$ should return error, in particular, we also need a new stack error $\bullet_S$.

$$
\begin{array}{rrl}
\mathrm{S{\small TACK}}[X:\mathrm{E{\small L}}] = & \lceil\ \mathrm{E{\small L}} = \mathcal{S}: El\ \rfloor\ \oplus & \\
\mathcal{S}: & ST[El] & \\
\Omega: & \bullet_E: & \to El \\
& \bullet_S: & \to ST[El] \\
& empty: & \to ST[El] \\
& push: & El \times ST \to ST \\
& pop: & ST[El] \to ST[El] \\
& top: & ST[El] \to El \\
\mathcal{A}: & 1. & top(push(x,S)) = x \\
& 2. & top(empty) = \bullet_E \\
& 3. & pop(push(x,S)) = S \\
& 4. & pop(empty) = empty \\
& 5. & top(\bullet_S) = \bullet_E \\
& 6. & push(\bullet_E, S) = \bullet_S \\
& 7. & push(x, \bullet_S) = \bullet_S \\
& 8. & pop(\bullet_S) = \bullet_S
\end{array}
$$

The intention here is that we construct stacks of the parameter sort El by adding the error element $\bullet_E$ which is the result of $top(empty)$. We also want all the operations to be strict – whenever an argument of some operation is the error-element ($\bullet_E$ for El or $\bullet_S$ for ST) the result of the whole operation is error. This claim is expressed by the axioms 5.-8. The strictness axioms may look plausible but, as in example 11.4, they collapse all Elements. We can namely derive the following:

$$x \stackrel{1}{=} top(push(x, push(\bullet_E, S))) \stackrel{6}{=} top(push(x, \bullet_S)) \stackrel{7}{=} top(\bullet_S) \stackrel{5}{=} \bullet_E$$

In oder words, we have that for any $x : x = \bullet_E$, i.e., this specification collapses the whole sort of El to the single error-element $\bullet_E$! No matter what actual parameter specification SP we will pass, the resulting specification STACK[SP] will have the El sort with only this one element.

The moral is that writing parameterized specifications one has to display particular care to *protect the actual parameter*. There are settings in which this can be enforced semi-automatically by verifying some restrictions on the form of the axioms, or else imposing some restrictions on the actual semantics of parameter passing. However, in the general setting in which we are working, the responsibility for such a protection is delegated to the person writing the specification.

11.2: Implementation of Parameterized Specifications

We have not defined any formal semantics of parameterized specifications but merely said that the semantics is obtained in the usual way whenever we pass an actual parameter.[7] As a consequence, parameterization acts as a merely textual structuring. In particular, it does not imply that the implementation will reflect the structure of a specification. Thus, we do not implement the specification $\mathrm{BSTree}[X : \mathrm{To}]$ – but given an actual parameter, e.g., $\mathrm{BSTree}[\mathbf{reach}_{\{0,\mathsf{S}\}}(\mathrm{NatL})\ \mathbf{by}\ \sigma]$, we obtain a specification which we can implement according to our earlier definition (8.23).

Sometimes this is acceptable but very often splitting specification in the parameter and body part indicates that we are interested in modularization of our software development – we would like to develop the piece of software corresponding to the body independently from the development of the parameter part. That is, we would like to give the parameterized specification (with the formal parameter) to a developer and obtain a program (or a more refined specification) which could be combined with any actual parameter specification to yield the overall result. This leads to the following generalization of the implementation concept.

Definition 11.6 Let $P = B[X : FP]$, $P1 = B1[X : FP]$ be two parameterized specifications with the same formal parameter. We say that $P1$ is an implementation of P, $P \longrightarrow P1$, iff for any legal actual parameter $AP : B[AP] \longrightarrow B1[AP]$.

According to this definition, implementation of a parameterized specification will affect its body but not the formal parameter. In fact, if $P1$ is a program implementing the original parameterized specification P, the definition requires that this program can be "instantiated" with a possible actual parameter yielding an implementation of the corresponding instantiation of P, i.e., that it is a program parameterized by a program. This certainly is not possible in all programming languages, although many modern programming languages do provide sufficiently powerful structuring mechanisms to faciliate, if not exactly this, so at least similar modularization. Methodological implications of this definition boil down to the requirement that structure of software should reflect, as far as it is possible, the structure of the specification from which it was developed. This intention can be illustrated by the following picture

$$\begin{tikzcd}
& B[X] \arrow[r] & B_1[X] \arrow[r, dotted, no head] & B_m[X] \arrow[dr] & \\
B[X{:}FP] \arrow[ur] \arrow[dr] & & & & B_m[SP_n] \\
& SP \arrow[r] & SP_1 \arrow[r, dotted, no head] & SP_n \arrow[ur] &
\end{tikzcd} \tag{11.48}$$

The horizontal transitions are usual implementation steps. Thus $B[X] \longrightarrow B_m[X]$ and $SP \longrightarrow SP_n$. The claim is that then the vertical composition of $B_m[SP_n]$ at the last step yields an implemetation of the specification $B[SP]$ – one says that *"parameterization commutes with implementation"*.

In order to verify implementation of parameterized specification according to definition 11.6, we will do the same as for the non-parameterized specifications – verify that the axioms of (the implementing specification) $P1$ imply the (appropriate translations of the) axioms of the (implemented specification) P.

Example 11.7 [11.3 continued]

We will implement the (parameterized) binary serach trees (example 11.3) by the binary trees from

[7] Actually, the equation (11.47) could be used to define the semantics of a parameterized specification as the model class of the structured specification from this equation. Then, instantiation of the formal parameter would *almost* correspond to refining this parameter, and the resulting specification would be *almost* an implementation of the parameterized one in the usual sense. These "alomost" refer to the fact that the signature of the instantiated specification would not be the same as of the original parameterized specification.

exercise 8.2, i.e., $\mathrm{BSTree}[X : \mathrm{To}] \longrightarrow \mathrm{BST}[X : \mathrm{To}]$. Notice that the signature of the former contained only the operation *ins*ert which was not present in the signature of the latter. The auxiliary operations lt and gt were hidden, i.e., they need not be implemented. We must however ensure that the search tree axioms involving these operations are satisfied by our implementation.

We can easily generalize the specification of binary trees from exercise 8.2 to the parameterized specification extended with the ins operation as follows:

$$\begin{array}{lllrcl}
\mathrm{BTree}[X : \mathrm{To}] = & \mathrm{To}\ \oplus & & & & \\
\mathcal{S}: & BT[El] & & & & \\
\Omega: & nil & & & \to & BT[El] \\
& tr & & BT[El] \times El \times BT[El] & \to & BT[El] \\
& L, R & & BT[El] & \to & BT[El] \\
& val & & BT[El] & \to & El \\
& ins & & El \times BT[El] & \to & BT[El]
\end{array}$$

$$\begin{array}{lllcl}
\mathcal{A}': & 1'. & val(tr(S,x,R)) & = & x \\
& 2'. & L(tr(S,x,T)) & = & S \\
& 3'. & R(tr(S,x,T)) & = & T \\
& 4'. & L(nil) & = & nil \\
& 5'. & R(nil) & = & nil \\
& 6'. & ins(x,nil) & = & tr(nil,x,nil) \\
& 7'. & T \neq nil \wedge x \prec val(T) = \top & \Rightarrow & ins(x,T) = tr(ins(x,L(T)),val(T),R(T)) \\
& 8'. & T \neq nil \wedge x \prec val(T) = \bot & \Rightarrow & ins(x,T) = tr(L(T),val(T),ins(x,R(T)))
\end{array}$$

The first 5 axioms are just binary tree axioms. The last three ones define the ins-operation in terms of the binary tree constructors nil and tr. In fact, we do not want to consider all possible binary trees – as the search trees we will only use those which can be constructed using the nil and this new ins operation. We claim that

$$\mathrm{BSTree}[X : \mathrm{To}] \longrightarrow \mathrm{BST}[X : \mathrm{To}] = \mathbf{hide}\ [\Omega : tr]\ \mathbf{in}\ \lceil \mathbf{reach}_{\{nil,ins\}} \mathrm{BTree}[X : \mathrm{To}] \rfloor$$

To verify this, we have to check that the models of BST are among the models of the BSTree – by showing $[\![\mathrm{BST}]\!] \models \mathcal{A}$(xioms) of BSTree. (Actually, definition 11.6 requires that this holds for any actual parameter specification. But if we show (using appropriate induction on $C = \{nil, ins\}$) that $\mathcal{A}' \vdash_{\omega_C} \mathcal{A}$, then the corresponding result will follow for arbitrary axioms of any actual parameter, i.e., $\mathcal{A}' \cup \mathcal{B} \vdash_{\omega_C} \mathcal{A} \cup \mathcal{B}$.)

The axioms 1.-3. are trivially implied by our current specification. However, the models of BSTree were reducts of the models which satisfied all the axioms involving also the operations lt and gt, so we have to check that having these operations defined in BTree as before in BSTree, the corresponding axioms will hold. So, let lt be defined on our $BT[El]$ by axioms 4. and 6. from example 11.3. We show that then axiom 8. will hold, i.e., $lt(L(ins(x,T)), val(ins(x,T))) = \top$.

We show it by induction on T. For $T = nil$ we have $ins(x,T) = tr(nil,x,nil)$. Then $L(ins(x,T)) = nil$ and $val(ins(x,T)) = x$ so, by axiom 4. we do have $lt(nil,x) = \top$.

Assuming that the axiom holds for $T \neq nil$, we have to show 8. for $ins(x,T)$. Since $\prec$ is a total order we have that if $x \neq val(T)$ then either $x \prec val(T) = \top$ or $x \prec val(T) = \bot$. In the first case, we obtain (axiom 7'. above) $ins(x,T) = tr(ins(x,L(T)), val(T), R(T))$. Then we have $x \prec val(T) = \top$ and, by induction hypothesis, $lt(L(T), val(T)) = \top$. But, by the axiom 7., this is exactly the required conclusion. In the second case, 8' gives $ins(x,T) = tr(L(T), val(T), ins(x,R(T)))$, and so $L(ins(x,T)) = L(T)$ and $val(ins(x,T)) = val(T)$. The desired equation holds then by the induction hypothesis. (The argument for gt will be exactly analogous.)

Notice the natural character of this implementation. BSTree puts only the search invariant requirement by using the auxiliary predicates lt and gt. BST refines this general requirement by actually specifying the construction of binary search trees using the tree building operation tr. At the same time, we have to hide tr since it could be used to construct binary trees which are not search trees. This was the intention of putting the reachability constraint – as binary search trees we will consider only those which can be constructed by the special kinds of applications of tr, namely the ones which occur in the definition of ins.

11.2.1: Specification of Parameterized Programs

As we said, in general, the parameterized specifications provide useful way of structuring the *text* of the specification. It is possible (and has been done by several people) to endow this mechanism with a precise semantics. Typically, it will be something like this: a specificaation identifies a class of algebras (its models); in particular, parameter of a parameterized specification identifies such a class; a parameterized specification takes then such a class as an argument and returns (specifies) a new class. If we let FP denote the specification of the formal parameter and $B[FP]$ the resulting parameterized specification, the meaning of a parameterized specification is that taking a subclass of $\mathsf{A} \subseteq \mathsf{Alg}(FP)$ which corrresponds to the actual parameter, it produces a subclass $\mathsf{R} \subseteq \mathsf{Alg}(B[FP])$ of respective resulting algebras,[8] that is

$$[\![B[FP]]\!]^{ps} \in \{b : \wp(\mathsf{Alg}(FP)) \to \wp(\mathsf{Alg}(B[FP]))\} \tag{11.49}$$

This, however, is not sufficient if we want our specifications to imply some structuring of the resulting programs. Any such a function b need not preserve the *structure* of the algebras. Similarly, if we treat the semantics of a parameterized specification as we did in (11.47), i.e., as a usual class of models, its implementation amounts to picking arbitrary subclass which does not have to reflect the structure of the original parameterized specification.

The intention of such a reflecting or preserving the structure of the specification underlied the requirement of actual parameter protection and the definition 11.6 of implementation of a parameterized specification. Its meaning, as illustrated in figure (11.48), was that one could implement the body of the parameterized specification and its parameter independently from each other. Thus, instead of (11.49), a parameterized specification would denote a (set of) function(s) taking individual algebras, i.e., programs, from $\mathsf{Alg}(FP)$, and returning a corresponding program:

$$[\![B[FP]]\!]^{pp} \in \{b : \mathsf{Alg}(FP) \to \mathsf{Alg}(B[FP])\} \tag{11.50}$$

In this view, the semantics of a parameterized specification is actually a *parameterized program* – a function which given an FP-algebra (program) produces a $B[FP]$-algebra (program). Then one may say that given signatures Σ_{FP} and $\Sigma_{B[FP]}$ of parameter and parameterized specifications, respectively, we can consider a new kind of signature $\Sigma_{FP} \to \Sigma_{B[FP]}$ – the algebras over such a signature will be exactly the algebras parameterized by algebras as in (11.50).

Recall the definition of a constructor implementation 10.4 – it was exactly this concept! A constructor was (a specification building) operation with the semantics defined as a function mapping algebras to algebras, i.e., as in (11.50). The intention was just the same – one wanted to split the implementation task into smaller subtasks, each resulting in a program P_i, and then to combine these by means of a constructor – a program κ – into an implementation of the original specification. In fact, any constructor implementation $SP \longrightarrow \kappa(SP_i)$ can be viewed as an implementation of SP by a parameterized specification with the body κ and the parameter SP_i.

There was however the obvious restriction – not all useful specification building operations are constructors. Thus the parameterization of specifications as we have considered it (corresponding to (11.49)) is a more general tool. Also, it is not always the case (especially at the early stage of specifying the general requirements) that we really want the resulting program to reflect the structure of the specification. At this stage, parameterization will often function as a means of efficient structuring and reuse of the texts of specifications.

11.3: Specification of Backtracking Strategy

We give a more elaborate example of parameterization by specifying a backtracking strategy which, parameterized by a problem description, can be used to find the solution of the problem.

We assume given a problem description as follows: a sort of STates which can be transformed by an operation nxt, with a predicate ok determining the legal ST, i.e., defining possible restrictions on the state-space. We also assume given an $init$ial STate as well as the $goal$ to be reached.

[8] Modulo the possible extensions and renaming of the formal parameter by the actual one.

Transistion from one state to another is a function of the current STate and an additional parameter P which is provided as the second argument to nxt. Thus, for each STate we also want to be able to determine, from the problem description, which transitions are possible. We therefore require a function $items$ returning, for each STate, the list of P-items which can be used in this STate to determine the next one. This list is specified by means of a parameterized specification SQ of sequences. Thus we have the following specification of formal parameter Prob:

$$\begin{array}{rl}
\mathrm{SQ}[X:\mathrm{El}] = & \lceil\ \mathcal{S} : El\ \rfloor \ \oplus\ \mathrm{Bool}\ \oplus \\
\mathcal{S}: & SQ[El] \\
\Omega: & \begin{array}{rrl} \varepsilon : & & \to SQ[El] \\ \vdash : & SQ[El] \times El & \to SQ[El] \\ \dashv : & El \times SQ[El] & \to SQ[El] \\ hd : & SQ[El] & \to El \\ \in : & El \times SQ[El] & \to \mathsf{B} \end{array} \\
\mathcal{A}: & \begin{array}{rcl} \varepsilon \vdash x & = & x \dashv \varepsilon \\ x \dashv (S \vdash y) & = & (x \dashv S) \vdash y \\ hd(s \vdash x) & = & x \\ x \in \varepsilon & = & \bot \\ x \in S \vdash y = \top & \Leftrightarrow & x = y \vee x \in S = \top \end{array}
\end{array}
\qquad
\begin{array}{rl}
\mathrm{Pro} = & \mathrm{Bool}\ \oplus \\
\mathcal{S}: & ST, P \\
\Omega: & \begin{array}{rrl} init : & & \to ST \\ goal : & & \to ST \\ nxt : & ST \times P & \to ST \\ ok : & ST & \to \mathsf{B} \end{array}
\end{array}$$

$$\begin{array}{rl}
\mathrm{Prob}[X:\mathrm{Pro}] = & \mathrm{SQ}[X\ \mathbf{by}\ El \mapsto P]\ \oplus \\
\Omega: & items :\ \ ST \to SQ[P]
\end{array}$$

Notice how in Prob we used an instantiation of SQ in order to obtain lists of items P.

A solution to a problem Pr : Prob amounts to constructing a sequence of transitions from the $init$ial to the $goal$ STate. We want to ensure that such a solution will be found – provided that it exists. That is, whether it will be found or not depends on the actual problem description, in particular, the specific definitions of $items$ and nxt.

Thus we want to ensure that *if* the actual problem description admits a solution *then* it will be found, i.e.: given some initial state s_1, *if* there is a sequence of transitions $s_1 s_2 ... s_g$ such that 1) $s_g = goal$, and for all $1 \leq i < g$ 2) $ok(s_i)$, and 3) $s_{i+1} = nxt(s_i, k)$ for some k among the $items(s_i)$, *then* a solution will be found, i.e., $hd(slv(\varepsilon \vdash s_1)) = goal$, where slv is the function we want to specify. In addition, under the above conditions, slv should return a legal sequence, i.e., where each state is ok and obtained by a transition from the previous state. So let the predicate SOL say exactly this – we define the following specification Slv with the function of interest slv :

$$\begin{array}{rl}
\mathrm{Slv} = [Pr : \mathrm{Prob}[X:\mathrm{Pro}]] & \mathrm{SQ}[Pr\ \mathbf{by}\ El \mapsto ST]\ \oplus \\
\mathbf{hide}\ [OK, SOL]\ \mathbf{in}\ \lceil\ \Omega: & \begin{array}{rl} slv : & SQ[ST] \to SQ[ST] \\ OK : & SQ[ST] \to \mathsf{B} \\ SOL : & SQ[ST] \to \mathsf{B} \end{array} \\
\mathcal{A}: & \begin{array}{rcl} OK(\varepsilon \vdash s) & = & ok(s) \\ OK(sq \vdash s \vdash t) = \top & \Leftrightarrow & \exists k : [k \in items(s) = \top \wedge t = nxt(s,k)] \\ & & \wedge\ ok(t) = \top\ \wedge\ OK(sq \vdash s) = \top \\ SOL(sq) = \top & \Leftrightarrow & hd(sq) = goal\ \wedge\ OK(sq) = \top \\ \exists sq : SOL(s \dashv sq) = \top & \Rightarrow & SOL(slv(\varepsilon \vdash s)) = \top\ \rfloor \end{array}
\end{array}$$

Again, this specification uses instantiation of SQ – this time by Pr – in order to obtain the sort of lists of STates. Notice that a SOL sequence exists iff there exists such a sequence which is non-circular, i.e., where for $i \neq j : s_i \neq s_j$. (Implication from left to right is obtained by removing, for each $i < j$ with $s_i = s_j$ the segments $s_i s_{i+1} ... s_{j-1}$ – the resulting sequence satisfies SOL but is non-circular.)

11.3.1: Backtracking

We want to define the function slv which should look for a solution checking, in each STate, that

1. the state is not the $goal$ STate and, furthermore,
2. satisfies restrictions determined by the function ok from the actual problem, and

3. the seqeuence of states so far does not contain duplicates (if it does, we should backtrack).

If the first condition is false, the current sequence is returned. Otherwise, one will examine (recursively) the possible transitions with the consequtive items from the list *items* of P. For each of these items, one will perform the corresponding *nxt* transition, appending the new *ST*ate to the sequence, and calling recursively the *slv* operation. If it fails, the next item is probed. If none of the possibilities leads to a solution, the constant *bck* is returned from the call indicating the need for backtracking. The third condition ensures termination (at least for a finite state-space).

$$
\begin{array}{lrl}
\textsc{BckT}[Pr:\textsc{Prob}[X:\textsc{Pro}]] = & \mathrm{SQ}[Pr\ \mathbf{by}\ El \mapsto ST] \ \oplus & \\
\Omega: & slv: & SQ[ST] \to SQ[ST] \\
 & try: & SQ[ST]\times SQ[P] \to SQ[ST] \\
 & bck: & \to SQ[ST] \\
 & circ: & SQ[ST] \to \mathsf{B} \\
 & occ: & ST \times SQ[ST] \to \mathsf{B}
\end{array}
$$

$$
\begin{array}{llrcl}
\mathcal{A}: & & & & \\
 & 0. & hd(bck) & \neq & goal \\
(slv) & 1. & hd(sq) = goal & \Rightarrow & slv(sq) = sq \\
 & 2. & hd(sq) \neq goal \ \wedge \ ok(hd(sq)) = \top & & \\
 & & \wedge \ circ(sq) = \bot & \Rightarrow & slv(sq) = try(sq, items(hd(sq))) \\
 & 3. & hd(sq) \neq goal \ \wedge \ ok(hd(sq)) = \bot & \Rightarrow & slv(sq) = bck \\
 & 4. & hd(sq) \neq goal \ \wedge \ circ(sq) = \top & \Rightarrow & slv(sq) = bck \\
(try) & 5. & try(sq,\varepsilon) & = & bck \\
 & 6. & hd(slv(sq \vdash nxt(hd(sq),i))) \neq goal & \Rightarrow & try(sq, si \vdash i) = try(sq, si) \\
 & 7. & hd(slv(sq \vdash nxt(hd(sq),i))) = goal & \Rightarrow & try(sq, si \vdash i) = slv(sq \vdash nxt(hd(sq),i)) \\
(circ) & 8. & circ(\varepsilon) & = & \bot \\
 & 9. & circ(sq \vdash s) & = & occ(s,sq) \\
(occ) & 10. & occ(s,\varepsilon) & = & \bot \\
 & 11. & occ(s, sq \vdash t) = \top & \Leftrightarrow & s = t \ \vee \ occ(s,sq) = \top
\end{array}
$$

Axiom 9. does not check fully for non-circularity but merely that the state s does not occur in sq. Non-circularity of the whole sequence $sq \vdash s$ follows then only if also sq itself is not circular.

We would like to know that BckT does define a correct strategy, that is, that it implements the specification $\textsc{Slv}[Pr:\textsc{Prob}[X:\textsc{Pro}]]$. The OK part can be verified by induction on the length of the resulting sequence of states. A bit informally, assume that $slv(\varepsilon \vdash s_1)$ did find a solution sequence. $OK(\varepsilon \vdash s_1)$ holds because it is trivially true for one-element lists. Given an $OK(s_1...s_{i-1})$, the next state s_i will be appended as follows: the conditions of axioms 1., 3., 4. are false and only the condition of axiom 2. holds. Thus *slv* will evaluate *try* for consequtive $items(s_{i-1})$ until a k is found which makes the condition of axiom 7. true (and which exists by the assumption of the last axioms of Slv). The result will then be $s_i = nxt(s_{i-1},k)$, with $k \in items(s_{i-1})$. The condition of axiom 2. was true for it (if it wasn't, the recursive call would return *bck*). But this condition states exactly that also $ok(s_i)$.

As to the last axioms of Slv, assume that we have constructed the sequence $s_1 s_2 ... s_{i-1}$ (initially just s_1). If $s_{i-1} = goal$, axiom 1. terminates the search. Otherwise, assuming that the sequence so far is OK (and is not circular), the condition of axiom 2. is true, so we continue by *try*ing all the *items* available in the state s_{i-1}. If *try*ing an item leads to *bck*, we try the next one (since $hd(bck) \neq goal$ by axiom 0.). We are guaranteed that some k among these *items* does lead to a correct next state, i.e., eventually we will find such a k for which $hd(slv(s_1 s_2 ... s_{i-1} \vdash nxt(s_{i-1},k))) = goal$. This will result in using the axiom 7., i.e., in appending this new state $nxt(s_{i-1},k)$ to the sequence and continuing, recursively, *try*ing to find the next state.

An important observation concerns the conditions in axioms 6. and 7. They check whether the head of the sequence is *goal* or not. It looks like cheating, since this means that we append the state to the solution sequence only after we have found out that this sequence will lead to the *goal* by constructing it. This kind of "cheating" is very common – simply because we merely state

static properties, axioms, which have to be satisfied. We do not *construct* anything, and merely *think of* evaluation of various functions defined by the axioms in operational terms. Furthermore, these conditions implicitly hide the decision as to what to do if the search space is infinite. If in a state s_i we have a list of items $a \vdash b \vdash c$, and it is the item a which leads to the solution, it may happen that item b (which will be *try*ied before a) leads to an infinite and non-circular sequence of states. The result of $slv(s_1...s_i)$ will then be different from *bck* but won't give a solution (unless, rather counter-intuitively, we define the *hd* of an infinite list to be *goal*). In fact, the specification SQ defines only *hd* of finite lists. In other words, our specification does not provide a constructive way of discovering that one enters on an infinite non-circular path – it merely assumes that for such a path its *hd* is different from *goal* and so one will, somehow, discover this fact. (An alternative, which would make this possibility more transparent, would be to use the conditions $slv(sq \vdash nxt(hd(sq))) = bck$, resp. $\neq bck$ in axioms 6., resp. 7.)

11.3.2: The Farmer, the Wolf, the Goat and the Cabbage

We now give a specification of a well-known problem which we will then use as an actual parameter to the BckT in order to find its solution. Notice that it is very natural to think of BckT as a module which should be implemented independently from any actual problem description. In practice, we would like to have such an implementation and to use it for any particular program which defines an actual problem.

A farmer is sitting on the left bank of a river with a goat, cabbage and a wolf, which he wants to transport on the right bank. He has a ferry but can load only one of these "items" at a time. On the other hand, he cannot leave both the goat and cabbage, or else both the wolf and the goat on one bank (since one of them would eat the other). The question is, of course, how to manage it, but here we will merely specify the problem and its restrictions.

We have three sorts of Items, Banks (with the obvious constants) and States which, for each item, assign the current Bank. Application of a State to an Item is performed by the *pos*ition function. The *mv* operation changes the States by moving the Farmer and at most one more item from the current Bank to the other one. In order to specify the restrictions, we use the Boolean function *ok* returning $\top$ iff, in the argument State, the Farmer is on the same bank as either Goat or Cabbage and, at the same time, on the same bank as either Wolf or Goat.

$$
\begin{array}{lrl}
\textsc{Farmer} = & \textsc{Bool} \ \oplus & \\
\mathcal{S}: & I, B, S = [I \to B] & \\
\Omega: & F,W,G,C: & \to I \\
 & L,R: & \to B \\
 & init, goal: & \to S \\
 & pos: & S \times I \to B \\
 & mv: & S \times I \to S \\
 & ok: & S \to \mathsf{B} \\
 & op: & B \to B
\end{array}
$$

$$
\begin{array}{llrcl}
\mathcal{A}: \ (aux) & 1. & op(L) & = & R \\
 & 2. & op(R) & = & L \\
 & 3. & pos(init,x) & = & L \\
 & 4. & pos(goal,x) & = & R \\
(mv) & 5. & pos(mv(s,x),F) & = & op(pos(s,F)) \\
 & 6. & pos(mv(s,x),x) & = & op(pos(s,x)) \\
 & 7. & y \neq x \wedge y \neq F & \Rightarrow & pos(mv(s,x),y) = pos(s,y) \\
(ok) & 8. & ok(s) = \top & \Leftrightarrow & (pos(s,F) = pos(s,G) \vee pos(s,F) = pos(s,C)) \ \wedge \\
 & & & & (pos(s,F) = pos(s,G) \vee pos(s,F) = pos(s,W))
\end{array}
$$

Since it is not obvious (from the form of the axioms) that we can use initial semantics, we should ensure the desirable inequalities:[9]

$$L \neq R \qquad F \neq G \qquad F \neq W \quad \dots \quad G \neq C$$

as well as "no junk" requirement, either by the generatedeness constraints, or by the axioms:[10]

$$x = L \vee x = R \qquad\qquad y = F \vee y = G \vee y = C \vee y = W$$

Notice that the axioms cannot here ensure generatedeness of the sort S. In general, we would rather use the constrained specification $=$ $\mathbf{reach}_{\{L,R,F,G,W,C,init,mv\}}$(FARMER) implying the above axioms as well as the fact that each State is either $init$ or is reachable by mv from $init$.

Finally, we should say which states are considered equal, namely:[11]

$$s = t \quad \Leftrightarrow \quad \forall x : pos(s,x) = pos(t,x)$$

11.3.3: INSTANTIATION

Inspecting the specification FARMER, we see that it possesses the required operations needed for using it as the actual parameter to the specification PROB. We thus instantiate the formal parameter PRO by FARMER using the obvious fitting morphism. We follow this instantiation immediately by the axioms defining the required operation $items$. For this purpose, we introduce three auxiliary operations: all returning the list of all items involved, Bk which tells the bank of the river at which the ferry (and the farmer) are in the current state, and its which recursively goes through the list of all items checking which are at the same bank as the ferry. The actual parameter specification of the farmer, wolf, etc. problem, extended with these definitions will then look as follows.

$$\begin{array}{rll}
\text{PROBF} = & \text{PROB}[\text{FARMER } \mathbf{by}\ P, ST, nxt \mapsto I, S, mv] \ \oplus & \\
\Omega: & all : \quad \to SEQ[I] & \\
& Bk : \quad S \to B & \\
& its : \quad S \times B \times SEQ[I] \to SEQ[I] & \\
\end{array}$$

$$\begin{array}{rrrcl}
\mathcal{A}: & 1. & all & = & \varepsilon \vdash F \vdash C \vdash W \vdash G \\
(Bk) & 2. & Bk(init) & = & L \\
& 3. & Bk(mv(s,x)) & = & op(Bk(s)) \\
(items) & 4. & items(s) & = & its(s, Bk(s), all, \varepsilon) \\
& 5. & its(s,x,\varepsilon,si) & = & si \\
& 6. & pos(s,i) = x & \Rightarrow & its(s,x,sq \vdash i, si) = its(s,x,sq,si \vdash i) \\
& 7. & pos(s,i) \neq x & \Rightarrow & its(s,x,sq \vdash i, si) = its(s,x,sq,si)
\end{array}$$

Now, we can instantiate the solution strategy BCKT by the above problem specification

$$\text{BCKF} = \quad \text{BCKT}[\text{PROBF } \mathbf{by}\ P, ST, nxt \mapsto I, S, mv]$$

The call $slv(\varepsilon \vdash init)$ will produce a sequence of transitions leading to the solution. It may be rather annoying (though not impossible) to trace the backtracking path of the resulting strategy. Nevertheless let us look at it. We write a state as a pair of list of items at, resp., L and R bank. The list containing F indicates the current bank and its contents are sorted in the order in which farmer

[9] Notice that 7. can be seen as an axiom schema abreviating all possible combinations of x and $y \neq F$. But we are then still left with axiom 8.

[10] The first of these axioms is actually necessary to ensure that for any State s and Item y : $pos(s,y) = L \vee pos(s,y) = R$. From this, there also follows the equivalence of the ok-axiom 8. with the alternative of replacing the two conjuncts by, respectively, $pos(s,G) = pos(s,C) \Rightarrow pos(s,F) = pos(s,C)$ and $pos(s,G) = pos(s,W) \Rightarrow pos(s,F) = pos(s,W)$.

[11] Also this axiom can be seen as an abbreviation for five condiational axioms $pos(s,W) = pos(t,W) \wedge \ldots \wedge pos(s,F) = pos(t,F) \ \Rightarrow \ s = t$, and $s = t \Rightarrow pos(s,x) = pos(t,x)$, for $x = F, W, G, C$.

will attempt to transport the $items$. The first call $slv(\varepsilon \vdash init)$ evaluates $try(\varepsilon \vdash init, items(init))$ where $items(init) = its(init, L, all, \varepsilon) = all$.

$$\overset{(1)}{\begin{array}{c|c} F & \\ C & \\ W & \\ G & \end{array}} \longrightarrow \overset{(2)}{\begin{array}{c|c} C & F \\ W & G \\ & \\ & \end{array}} \ \dots \ \overset{(3)=\,1}{\begin{array}{c|c} F & \\ C & \\ W & \\ G & \end{array}} \longrightarrow \overset{(4)}{\begin{array}{c|c} F & G \\ C & \\ W & \\ & \end{array}} \longrightarrow \overset{(5)}{\begin{array}{c|c} C & F \\ & W \\ & G \\ & \end{array}} \longrightarrow \overset{(6)}{\begin{array}{c|c} F & W \\ C & \\ G & \\ & \end{array}} \ \dots \ \overset{(7)=\,5}{\begin{array}{c|c} C & F \\ & W \\ & G \end{array}}$$

The attempt after state (2) to take G back to L would give a sequence $sq = \varepsilon \vdash 1 \vdash 2 \vdash 3$ for which $circ(sq) = \top$ since $3 = 1$. Similarly, the attempt to take G to R after (6). Thus, one will get to the following state (8), after which there follows a series of failed attempts:

$$\overset{(8)}{\begin{array}{c|c} G & F \\ & C \\ & W \\ & \end{array}} \longrightarrow \overset{(9)}{\begin{array}{c|c} F & C \\ W & \\ G & \end{array}} \longrightarrow \overset{(10)}{\begin{array}{c|c} W & F \\ & C \\ & G \end{array}} \ \dots \ \overset{(11)=\,9}{\begin{array}{c|c} F & C \\ W & \\ G & \end{array}} \ \dots \ \overset{(12)=\,4}{\begin{array}{c|c} F & G \\ C & \\ W & \end{array}} \ \dots \ \overset{(13):\ \neg ok}{\begin{array}{c|c} F & C \\ W & G \\ & \end{array}} \ \dashv$$

Consequently, we have to backtrack from the state (10) and try other items from state (9)

$$(9) \ \dots \ \overset{(14)=\,8}{\begin{array}{c|c} G & F \\ & C \\ & W \end{array}} \ \dots \ \overset{(15):\ \neg ok}{\begin{array}{c|c} W & F \\ G & C \\ & \end{array}} \ \dashv \ (8) \ \dots \ \overset{(16)=\,5}{\begin{array}{c|c} F & W \\ C & \\ G & \end{array}} \longrightarrow \overset{(17)}{\begin{array}{c|c} F & C \\ G & W \\ & \\ & \end{array}} \longrightarrow \overset{(18)}{\begin{array}{c|c} & F \\ & C \\ & W \\ & G \end{array}}$$

The first two attempts end the possibilities of moves from state (9), so we have to backtrack to (8) and try the next item from its list. This brings us to the state (16) wich is = (5). Thus we are left with the last item, F, which results in state (17). A single transition yields the state (18)= $goal$.

Notice that the definition of the $items$ function has crucial bearing on the efficiency of the resulting backtracking. We chose it so that this result is pretty satisfactory. With other choices, the search for the solution would terminate successfully but would require more steps.

Exercises (week 11)

EXERCISE 11.1 Recall exercise 10.2. Modify the specification ARC which you used there, so that it is parameterized by the specification (sort) of the elements.

Verify then that this specification implements the parameterized specification LIST[X : EL] from example 11.1.

EXERCISE 11.2 Assume that we have a specification GROUP of groups (as in example 1.24). Design a specification MATRIX[X : GROUP] of 2×2 matrices ($M[S]$) parameterized by the groups. We want to be able to

a) construct a matrix, i.e., given four elements from the group, an operation mtr should give us a matrix,
b) identify the 0-matrix (i.e., to have a constant returning the matrix of 0's),
c) perform the group operation $\circ$ on the matrices (i.e., thinking of $\circ$ as addition, we want to be able to perform the addition on matrices by adding the respective entries),
d) find the inverse of a matrix wrt. $\circ$.

1. Give the signature for the parameterized specification of matrices.
2. Give (equational) axioms defining the matrix operations b)-d).

3. Verify that the resulting specification itself satisfies the specifcation GROUP, i.e., that taking the sort $M[S]$ generated by the mtr-operation, as the carrier of the sort S of GROUP, the operations you have defined in 2. satisfy the axioms of GROUP. (In other words, show that $\mathbf{reach}_{\{mtr\}}(\text{MATRIX}[X : \text{GROUP}]) \models \mathcal{A}$ where $\mathcal{A}$ are the axioms from GROUP.)

EXERCISE 11.3 You will here specify another problem so that it can be passed as an actual parameter to the BCKT specification from 11.3.

For clarity, we simplify this old toy-problem. Monkey can move along in a corridor either to the left or to the right (provided it is not at the respective end of the corridor). Somewhere on the floor there is a box and somewhere under the ceiling hangs a banana. The only way to get the banana – which is the monkey's goal – is to move the box under it and climb the box. Thus:

- given the (horizontal coordinate) of the *ban*ana, and some initial state: the (horizontal) coordinates of the box and the monkey,
- the monkey can move *L*eft or *R*ight or, if it is in the same position as the box, can either climb *U*p the box, climb *D*own from the box, or else take the box with it to the left or right.

1. Determine a signature for describing this problem, including the function mv corresponding to a move of the monkey. (A mv is a transformation of the total state, so decide first how you'll represent the states.) You may assume that monkey, whenever at the same position as the box, will always drag the box with it when moving left or right.

2. Then give the axioms for the operations you have introduced.

3. Adjust your specification so that it can be given as an actual parameter to the specification PROB from 11.3. Determine the adequate signature morphism.

4. Instantiate PROB with the result of point 2. and provide a definition of the *items* function.

5. Indicate how the backtrcking will work after this instantiation,

It may be useful to think of the corridor as a sequence of positions $0, 1, 2, ..., n$, where n is the length of the corridor, with two operations r, l for, respectively, a move to the right and to the left. You may use for this purpose the following specification of the corridor – an interval $[0...n]$

CORRIDOR =

$$\begin{array}{lllll}
\mathcal{S}: & H(orizontal) & & & \\
\Omega: & 0, n: \quad \to H & & & \\
 & \mathsf{r}, \mathsf{l}: \ H \to H & & & \\
\mathcal{A}: & n = \underbrace{\mathsf{rr}\ldots\mathsf{r}}_{n}0 & \mathsf{r}n = n & x \neq n \Rightarrow \mathsf{lr}x = x \\
 & & \mathsf{l}0 = 0 & x \neq 0 \Rightarrow \mathsf{rl}x = x
\end{array}$$