A Modular Framework for Specification and Implementation[*]

Martin Wirsing Manfred Broy

Fakultät für Mathematik und Informatik
Universität Passau
Postfach 2540
D-8390 Passau

Abstract

A modular framework for the formal specification and implementation of large families of sorts and functions is introduced. It is intended to express generation principles, to rename, combine and construct implementations of specifications in flexible styles. Parameterization is also included.
The main characteristics of this approach are the inclusion of predicates in the signature of specifications and the use of an ultra-loose semantics. Signatures are triples of sets of sorts, sets of function symbols and sets of predicate symbols; the latter contain among others also standard predicate symbols, in particular the equality symbols as well as predicate symbols expressing generation principles which hold for an object if and only if it can be denoted by a term of a specific signature. These standard predicate symbols lead to an ultra-loose semantics for specifications: models are not required to be term-generated; instead, the term-generated subalgebra of a model is required to satisfy the axioms. Main advantages of this approach are the simplicity of the notion of implementation and the simplicity of the corresponding language for writing structured specifications.

1. Introduction

Algebraic specifications provide a flexible and abstract way for the description of data structures. The basic idea of the algebraic approach consists in introducing names for the different sets of data and for the basic functions operating on the data and by giving logical requirements for the data that may be associated with those names. The requirements are described by equations or by first-order formulas using equations.
Writing large specifications in this basic way can be very time consuming. For their use in requirements engineering and software design applications it seems decisive that specifications can be modularized and manipulated by appropriate language constructs and be written in a parameterized way. In order to support formal software development, specifications have to be equipped with a notion of implementation which allows to verify the transitions from abstract descriptions to more concrete ones. Consequently, any formal framework for specifications should comprise language constructs for building structured specifications and a notion of implementation. The first of such frameworks was given by the ADJ-group [Goguen et al. 78] who introduced the

[*]This paper has been partially sponsored by the ESPRIT project METEOR.

initial algebra approach together with the notion "Forget-Restrict-Identify" (FRI) for implementation. This suggestion has been completed on the one hand by Goguen and Burstall with the languages CLEAR and OBJ ([Burstall, Goguen 80], [Futatsugi et al. 85]) and on the other hand by Ehrig and his group who added parameterization and structuring operators with the language ACT ONE [Ehrig, Mahr 85] and studied thoroughly the mathematical properties of the FRI notion [Ehrig, Kreowski 82], [Ehrig et al. 82].

The hierarchical approach to specifications with loose semantics [Wirsing et al. 83] has been incorporated in the language CIP-L [Bauer, Wössner 82]. There also the "FRI"-relation is the main notion of implementation. The language ASL ([Sannella, Wirsing 83], [Wirsing 86]) is also based on loose semantics but a much simpler notion of implementation is chosen, namely the model class inclusion. This is possible due to the power of the specification operators of ASL which comprise in particular an operator for behavioural abstraction. The latter notion represents a third actual research stream on the semantics of specifications showing a number of encouraging results [Hennicker 89]. Actual language developments such as ACT TWO [Ehrig, Weber 86], combine loose and initial semantics or elaborate better the notion of module (see ACT TWO, Larch [Guttag et al. 85], Extended ML [Sannella, Tarlecki 85]. However, all these powerful languages (including ASL) have the drawback that they do not well support the FRI-notion of implementation: the horizontal composition theorem for FRI-implementation does only hold under severe restrictions. A comparison of all these different approaches seems to indicate a principal difficulty: either a conceptually simple language (such as ACT ONE) is combined with a complex notion of implementation (such as FRI) or a simple notion of implementation (such as model class inclusion) is combined with powerful, but mathematically difficult operators (such as behavioural abstraction).

In the following we introduce a formal framework for writing, manipulating and combining specifications in which we try to avoid the above difficulty: the notion of implementation is model class inclusion, the language (similar to ACT ONE) comprises only three basic and mathematically simple operators which support syntactic manipulations of specifications such as export and renaming; horizontal and vertical composition of implementations holds without any restrictions. Technically, the novelty of the approach consists in the inclusion of standard predicate symbols in the signature of specifications and in the use of an ultra-loose semantics: the models of an algebraic specification with signature Σ and axioms E are all first-order Σ-structures which satisfy the axioms E. Therefore, the models are not restricted to term-generated structures - junk is allowed. However, properties of the term-generated elements are easily expressible due to the use of standard generating predicate symbols. Moreover, the interpretation of the equality symbol in a model A of a specification SP does not necessarily coincide with the equality between the elements of A; it has only to be an appropriate congruence relation. Hence, the models of SP coincide exactly with the "implementations" of SP.

The paper is organized as follows: In section 2, the notions of signatures and structures with predicates are introduced. In section 2.1 the basic definitions are given; in section 2.2 these notions

are extended to comprise the standard predicate symbols and the generating predicate symbols.
In section 3 many-sorted first-order formulas with standard predicates are studied. In particular, proof rules are given and it is shown how reachability is expressible by such formulas.
In section 4 flat specifications with constructors are introduced and the class of models of well-known examples such as total orderings, truth values and natural numbers is studied. It is shown that natural numbers are admitted as model (i.e. implementation) of truth values and similarly that integers are a model of natural numbers.
In section 5 the three basic structuring operators for specifications are introduced. The operators "translate" and "derive" are deduced from both directions of a signature morphism, the operator "+" combines two specifications without taking care of name clashes. Using "+" the operator "enrich" can be defined which is used for the structured specification of two examples, sets of natural numbers and sequences of integers.
In section 6 properties of the specification operators are studied. It is shown that exporting, hiding and copying can be defined using the basic operators; a number of algebraic identities are given which will be useful for proving properties of implementations such as horizontal and vertical composition of ERE-implementations (cf. section 8).
In section 7 model class inclusion is introduced as notion of implementation. It is shown that after extensions and renamings integers implement natural numbers, ordered sequences of integers implement sets of natural numbers. The most difficult part of these extensions is the axiomatization of the congruence to be implemented. On the other hand, for the verification of the implementation only the validity of the axioms has to be checked, semantic manipulations such as restrictions and identification as for FRI-implementations are not necessary.
In section 8 parameterization of specifications is defined by lambda-abstraction; partial and total implementations of parameterized specifications are introduced. It is shown that all specification operators are monotonic w.r.t the refinement and implementation relation which implies the horizontal and vertical composition theorem. Finally, the implementation relation is generalized to include export, rename and extension (as in the examples of section 6). For this so-called ERE-implementations vertical and horizontal composition theorems are proven as well.
In the concluding remarks some additional possible standard predicate symbols are shortly discussed.

2. Signatures and structures

The semantics of algebraic specifications is determined by the notions of signature and structure or algebra. In contrast to the classical approaches (cf. e.g. [Goguen et al. 82], [Guttag 75]), signatures do not consist only of sorts and function symbols but contain predicate symbols as well. Moreover, all signatures are supposed to contain a number of standard predicate symbols. For algebraic specifications the most important standard symbols are the equality symbols $. =_s .$ (indexed by sorts s) and the generating predicate symbols $. \in \Sigma$ (for expressing that an element is denotable by a ground Σ-term).

2.1 Signatures and structures with predicates

The syntactic structure of a data type D is determined by its signature. Usually, a signature Σ is given as a set S of names of different kinds of data (the sorts) and a family F of notations for distinguished data and operations (the function symbols). In our approach, also predicate symbols are included in the signature.

Def. 2.1.1 A **signature** Σ consists of a triple $< S,F,P >$ where S is a set (of **sorts**), F is a set (of **function symbols**), and P is a set (of **predicate symbols**) such that F is equipped with a mapping type: $F \rightarrow S^* \times S$ and P is equipped with a mapping type: $P \rightarrow S^+$. For any $f \in F$ (or $p \in P$, resp.), the value type(f) (or type(p), resp.) is the **type** of f (or p, resp.). We write sorts(Σ) to denote S, opns(Σ) to denote F, preds(Σ) to denote P, f: $w \rightarrow s$ to denote $f \in F$ with type(f)=w,s and p:w to denote $p \in P$ with type(p)=w.

Let X be an S-sorted set. For every sort $s \in S$ the set, $T(<S,F>, X)_s$, of **terms of sort** s (containing elements in X) is defined as usual (as the least set built using X and the function symbols $f \in F$). Terms without elements of X are called **ground terms** and the set $T(<S,F>, \emptyset)$ of **all ground terms** is denoted by $T(<S,F>)$.We also write $T(\Sigma,X)$ for $T(<S,F>,X)$ and similarly, $T(\Sigma)$ for $T(<S,F>)$.

A signature morphism is defined as usual as a mapping from one signature into another such that the types of function and predicate symbols are compatible with the mapping of the sorts, i.e. for signatures $\Sigma=< S,F,P >$ and $\Sigma'=< S',F',P' >$ a **signature morphism** $\sigma:\Sigma \rightarrow \Sigma$ is a triple $<\sigma_{sorts}, \sigma_{opns}, \sigma_{preds}>$ where $\sigma_{sorts}:S \rightarrow S'$. $\sigma_{opns}: F \rightarrow F'$ and $\sigma_{preds}: P \rightarrow P'$ are mappings such that for any f: $w \rightarrow s \in F$, type(σ_{opns}(f))=$\sigma^*(w),\sigma(s)$ and

for any p: $w \in P$, type(σ_{preds}(p))=$\sigma^*(w)$

where denotes $\sigma^*(s_1...s_n)$ the extension of σ to words, i.e. $\sigma^*(s_1...s_n) =_{def} \sigma_{sorts}(s_1)...\sigma_{sorts}(s_n)$ for $s_1, ..., s_n \in S$.

Any mapping between sorts, function and predicate symbols induces a number of signature morphisms: let $\varphi=<\varphi_{srt}, \varphi_{opn}, \varphi_{prd}>$ be a triple of mappings φ_{srt}: $S1 \rightarrow S2$, φ_{opn}: $F1 \rightarrow F2$, φ_{prd}: $P1 \rightarrow P2$ where S1, S2 are sets of sorts, F1, F2 are sets of function symbols and P1, P2 sets of predicate symbols; moreover let $\Sigma=<S,F,P>$ be a signature. Then $_\Sigma\varphi$: $\Sigma \rightarrow \Sigma'$ denotes the signature morphism defined by

$$_\Sigma\varphi(x) =_{def} \begin{cases} \varphi(x) & \text{if } x \in S1 \cup F1 \cup P1, \\ x & \text{otherwise} \end{cases}$$

where $\Sigma'=<S',F',P'>$ is a signature defined by

$S' =_{def} {}_{\Sigma}\varphi(S)$,
$F' =_{def} \{{}_{\Sigma}\varphi(f): ({}_{\Sigma}\varphi)^*(w) \to {}_{\Sigma}\varphi(s) \mid f: w \to s \in F\}$,
$P' =_{def} \{{}_{\Sigma}\varphi(p): ({}_{\Sigma}\varphi)^*(w) \mid p: w \in P\}$.

If the signature Σ'' contains Σ' (i.e. $\Sigma' \subseteq \Sigma''$) then ${}_{\Sigma}\varphi_{\Sigma''}: \Sigma \to \Sigma''$ is the signature morphism ${}_{\Sigma}\varphi$ with range Σ''. If instead of Σ the signature Σ'' is given, we define the signature morphism $\varphi_{\Sigma''}: \Sigma \to \Sigma''$ analogously. We often omit indices from ${}_{\Sigma}\varphi$, $\varphi_{\Sigma''}$ or ${}_{\Sigma}\varphi_{\Sigma''}$ if the indexed signatures are obvious from the context. Moreover, if $x_1, \ldots, x_n$ are pairwise different sorts, function and predicate symbols and φ maps x_i to y_i for $i=1,\ldots,n$, we write $[x_1/y_1, \ldots, x_n/y_n]$ for ${}_{\Sigma}\varphi$ (and thus for all induced signature morphisms). If φ is injective, then we call it a **renaming**.

A general Σ-structure has a carrier set (the elements of the data type) for each sort, a function on these sets for each function symbol and a relation for each predicate symbol.

Def. 2.1.2 Let $\Sigma = <S, F, P>$ be a signature. A **general Σ-structure** A consists of an S-sorted family of non-empty carrier sets $\{A_s\}_{s \in S^*}$ and, for each f: s1, ..., sn $\to$ s $\in$ F, of a total function $f^A: A_{s1} \times \ldots \times A_{sn} \to A_s$, and for each p: s1, ..., sn $\in$ P, of a relation $p^A \subseteq A_{s1} \times \ldots \times A_{sn}$.

The relationship between terms and structures is as usual given by the notion of interpretation.
For any valuation v: X $\to$ A of an S-sorted set X into A we denote the **interpretation** (w.r.t. v) of a term t $\in$ T(Σ, X) by $v^*(t)$; if t is a ground term, then its interpretation does not depend on v and we write t^A instead of $v^*(t)$.
To any reachable Σ-structure A one may associate a term structure T(A) with carrier sets $T(\Sigma)_s$ as follows.

Def. 2.1.3 Let A be a reachable Σ-structure. The **Σ-term structure** T(A) associated with A is defined as follows:

(1) for each s $\in$ sorts(Σ), $T(A)_s =_{def} T(\Sigma)_s$;
(2) for each f: s1, ..., sn $\to$ s $\in$ F and each $t_1 \in T(A)_{s1}, \ldots, t_n \in T(A)_{sn}$,
$f^{T(A)}(t_1, \ldots, t_n) =_{def} f(t_1, \ldots, t_n)$
(3) for each p: s1, ..., sn $\in$ P and each $t_1 \in T(A)_{s1}, \ldots, t_n \in T(A)_{sn}$,
$<t_1,\ldots,t_n> \in p^{T(A)}$ iff $<t_1^A, \ldots, t_n^A> \in p^A$.

2.2 Standard predicates

In the remainder of this paper it is assumed that any signature contains the following set St of standard predicate symbols which depends on the available sorts and function symbols.

Def 2.2.1 Let S be a set of sorts and F be an S-sorted set of function symbols. The set $\mathbf{St}_{S,F}$ of **standard predicate symbols** for S, F consists of the following elements:

$. =_s . : s,s$ the **equality symbols** for all $s \in S$,
$. \in \langle S',F' \rangle_s : s$ the **generating predicate symbols** for any $S' \subseteq S$,
any subsignature $\langle S',F' \rangle$ of $\langle S,F \rangle$ and any $s \in S'$ such that $T(\langle S',F' \rangle)_s \neq \emptyset$.

In order to distinguish the equality symbols of different specifications, these symbols may get a name as additional index. Hence the set $St_{S,F}$ of standard predicate symbols contains an equality symbol for any sort of S and a family of generating predicate symbols for any subsignature of <S,F>. Moreover, if the sort s can be derived from the context then we write often = instead of $=_s$.

A simple example for a signature with standard predicate symbols is the signature $\Sigma B0$ of the data structure of truth values, which consists of one sort, two constants, one unary function symbol and the set $St_{\langle SB0,FB0 \rangle}$ of standard predicate symbols.

$\Sigma B0 = \langle SB0, FB0, St_{\langle SB0,FB0 \rangle} \rangle$ is defined by
$SB0 =_{def} \{Bool\}$,
$FB0 =_{def} \{true, false: \to Bool, not: Bool \to Bool\}$,
$St_{\langle SB0,FB0 \rangle} =_{def} \{ =_{Bool} : Bool, Bool;$
$. \in \langle \{Bool\}, \{true, false, not\} \rangle_{Bool}$,
$. \in \langle \{Bool\}, \{true, not\} \rangle_{Bool}$,
$. \in \langle \{Bool\}, \{false, not\} \rangle_{Bool}$,
$. \in \langle \{Bool\}, \{true, false\} \rangle_{Bool}$,
$. \in \langle \{Bool\}, \{true\} \rangle_{Bool}$,
$. \in \langle \{Bool\}, \{false\} \rangle_{Bool} : Bool \}$

Thus the generating predicate symbols contain all those subsignatures of < SB0, FB0 > which admit at least one ground term; for < {Bool}, {not} > no generating predicate symbol is defined. Notice the choice of "true, false, not" as boolean function symbols; in logic programming one would have considered them as predicate symbols.

The standard predicate symbols do not have arbitrary relations as interpretations: equality symbols must always be associated with congruence relations and the generating predicate symbols are associated with the corresponding sets of ground terms.

Def 2.2.2 Let $\Sigma = \langle S, F, P \cup St_{S,F} \rangle$ be a signature with standard symbols. A **Σ-structure** A is a general Σ-structure such that

(1) for any generating predicate symbol $\in \Sigma'_s$, for all $a \in A_s$:
$a(\in \Sigma'_s)^A$ iff there exists $t \in T(\Sigma')_s$ with $a = t^A$;

(2) $=^A$ is a Σ-congruence relation,
i.e. the following holds for all $a \in A_s$, $a_1,b_1 \in A_{s1}, \ldots, a_n,b_n \in A_{sn}$, where $s,s1,\ldots,sn \in S$,

(refl) $a =_s^A a$

(subst$_f$) $a_1 =_{s1}^A b_1 \wedge \ldots \wedge a_n =_{s1}^A b_n \Rightarrow f^A(a_1, \ldots, a_n) =_s^A f^A(b_1, \ldots, b_n)$
for each $f: s1, \ldots, sn \to s \in F$;

(subst$_p$) $a_1 =_s^A b_1 \wedge \ldots \wedge a_n =_{sn}^A b_n \wedge <a_1, \ldots, a_n> \in p^A \Rightarrow <b_1, \ldots, b_n> \in p^A$ for each $p:s1, \ldots, sn \to s \in P$.

Notice that symmetry and transitivity of $=_s^A$ follow from reflexivity (refl) and the substitution property (subst$_=$) for equality symbols: let p: s, s be $=_s$ and choose for proving symmetry $a_1 =_{def} a_2 =_{def} b_2 =_{def} a$, $b_1 =_{def} b$ and for proving transitivity $a_1 =_{def} b_1 =_{def} a$, $a_2 =_{def} b$, $b_2 =_{def} c$.

Given interpretations for sorts and function symbols the interpretation of the generating predicate symbols is uniquely determined whereas there may exist several congruence relations. Therefore, in the examples for structures usually the interpretation of the equality symbols is omitted if it coincides with the equality in the structure.

Examples for ΣB0-structures are the standard structure B of truth-values and the following two structures N1 and N2 with the natural numbers as carrier sets; we write "$\in \Sigma_{\{true,false\}}$" instead of "$\in < \{Bool\}, \{true,false\} >_{Bool}$" and analogously for the other generating predicate symbols.

$B_{Bool} =_{def} \{O,L\}$,
$true^B =_{def} L$, $false^B =_{def} O$, $not^B(L) =_{def} O$, $not^B(O) =_{def} L$
$x(\in \Sigma_{\{true,false\}})^B \Leftrightarrow x(\in \Sigma_{\{true,not\}})^B \Leftrightarrow x(\in \Sigma_{\{false,not\}})^B \Leftrightarrow x(\in \Sigma_{\{true,false,not\}})^B \Leftrightarrow x \in \{O,L\}$,
$x =^B y \Leftrightarrow x = y$;

$N1_{Bool} =_{def} \mathbb{N}$,
$true^{N1} =_{def} 1$, $false^{N1} =_{def} 0$, $not^{N1}(x) =_{def} x+1$,
$x(\in \Sigma_{\{true,false\}})^{N1} \Leftrightarrow x \in \{0,1\}$,
$x(\in \Sigma_{\{true,not\}})^{N1} \Leftrightarrow x \geq 1$, $x(\in \Sigma_{\{false,not\}})^{N1} \Leftrightarrow x(\in \Sigma_{\{true,false,not\}})^{N1} \Leftrightarrow x \in \mathbb{N}$,
$n =^{N1} m \Leftrightarrow n \bmod 2 = m \bmod 2$;

$N2_{Bool} =_{def} \mathbb{N}$,
$true^{N2} =_{def} 1$, $false^{N2} =_{def} 0$, $not^{N2}(0) =_{def} 1$, $not^{N2}(1) =_{def} 0$, $not^{N2}(n+2) =_{def} n+3$
$x(\in \Sigma_{\{true,false\}})^{N2} \Leftrightarrow x(\in \Sigma_{\{true,not\}})^{N2} \Leftrightarrow x(\in \Sigma_{\{false,not\}})^{N2} \Leftrightarrow x(\in \Sigma_{\{true,false,not\}})^{N2} \Leftrightarrow x \in \{0,1\}$,
$x =^{N2} y \Leftrightarrow x = y$.

For B and N2 the interpretation of the equality symbol coincides with the equality between the elements of the structure.

Def. 2.2.3 For any signature Σ with standard symbols a Σ-structure A is called

(1) **Σ-algebra** if for each $s \in S$ the equality symbol $=_s$ has the standard equational interpretation, i.e. for all $a, b \in A_s$, $a = b \Leftrightarrow a =_s^A b$;

(2) **Σ-reachable** if for each $s \in S$ and $a \in A_s$ there exists a ground Σ-term $t \in T(\Sigma)_s$ such that $a =_s^A t^A$

(3) **Σ-standard**, if A is a Σ-reachable Σ-algebra.

We will denote the class of all Σ-structures by **Struct(Σ)**, the class of all Σ-algebras by **Alg(Σ)** and the class of all Σ-standard structures by **Gen(Σ)**.

For example, the structure B is ΣB0-standard, the structure N1 is ΣB0-reachable but not a ΣB0-algebra, and the structure N2 is a ΣB0-algebra but it is not ΣB0-reachable.

A Σ-homomorphism is a mapping from a Σ-structure into another which is compatible with all function and predicate symbols.

Def. 2.2.4 Let $\Sigma = \langle S,F,P\rangle$ be a signature with standard symbols and let A, B be Σ-structures. A **Σ-homomorphism** $h: A \to B$ is a family of maps $\{h_s: A_s \to B_s\}_{s\in S}$ which is compatible with function and predicate symbols, i.e. for each f: s1, ..., sn -> s $\in$ F, each p: s1, ..., sn $\in$ P and each $a_1 \in A_{s1}, \dots, a_n \in A_{sn}$,

$$h_s(f^A(a_1, \dots, a_n)) =_s^B f^B(h_{s1}(a_1), \dots, h_{sn}(a_n)) \text{ and}$$
$$\langle a_1, \dots, a_n\rangle \in p^A \Rightarrow \langle h_{s1}(a_1), \dots, h_{sn}(a_n)\rangle \in p^B .$$

Notice that for checking the homomorphism property it is sufficient to consider the non-standard predicate symbols and the equality symbols; then the conditions for the generating predicate symbols are automatically satisfied.

Fact 2.2.5

(1) Every Σ-structure A has an **associated Σ-algebra A/=** which is defined as the quotient of A by the Σ-congruence $=^A$. Moreover, it has an associated Σ-reachable substructure, the **least Σ-substructure** $\langle A\rangle_\Sigma$ of A.

(2) There exists a unique (surjective) Σ–homomorphism from A onto A/= and there exist unique Σ–homomorphisms from $\langle A\rangle_\Sigma$ into A and into A/=.

(3) For any reachable Σ-structure A there exists a unique Σ-homomorphism from the Σ-structure T(A) onto A.

Every signature morphism $\sigma: \Sigma \to \Sigma'$ (where $\Sigma=\langle S,F,P\rangle$ and $\Sigma'=\langle S',F',P'\rangle$) induces an adjoint morphism on structures. Let A be a Σ'-structure; then $A|\sigma$, the **σ-reduct** of A, denotes the following Σ-structure:

$$(A|\sigma)_s =_{def} A_{\sigma(s)} \text{ for } s\in S, \quad f^{A|\sigma} =_{def} \sigma(f)^A \text{ for } f\in F, \quad p^{A|\sigma} =_{def} \sigma(p)^A \text{ for } p\in P.$$

If $\Sigma \subseteq \Sigma'$ and σ denotes the embedding from Σ into Σ' (i.e. $\sigma(x) = x$ for all $x\in\Sigma$), then we write $A|\Sigma$ for $A|\sigma$. For a class C of Σ'- structures we denote $\{A|\sigma \mid A\in C\}$ by $C|\sigma$.

3. Formulas

Properties of many-sorted structures are expressed by many-sorted first-order formulas which are defined as usual.

Def. 3.1 The set **WFF(Σ) of Σ-formulas** is the least set satisfying the following properties:

(i) if $p \in P_{s1,\dots,sn}$ and t_i, $i=1,\dots,n$, are Σ-terms of sort si, then $p(t_1,\dots,t_n) \in WFF(\Sigma)$;

(ii) if $G, H \in WFF(\Sigma)$ then $(\neg G)$, $(G \wedge H) \in WFF(\Sigma)$;

(iii) if $x \in X_s$ and $G \in WFF(\Sigma)$ then $(\forall x{:}s.G) \in WFF(\Sigma)$.

Formulas of the form $p(t_1,\dots,t_n)$ are called **atomic Σ-formulas**; in particular, an atomic formula $=_s(t_1,t_2)$ (often written $t_1=t_2$) is called **Σ-equation.** The generating predicate symbols are often used as restrictions for quantifications. We write

$\forall$ x: Σ_s.A for $\forall$x: s. $x \in \Sigma_s \Rightarrow A$ and

$\exists$ x: Σ_s.A for $\exists$x: s. $x \in \Sigma_s \wedge A$.

Further logical operators such as $\vee$, $\Rightarrow$ and $\exists$ are defined as usual abbreviations. Superfluous brackets will be omitted. By V(G) we denote the set of all variables occurring in the Σ-formula G. A Σ-formula without free variables is called **Σ-sentence.** It is called **ground** if it is without variables at all.

Def. 3.2 For any Σ-structure A, valuation $v{:}X \to A$ and Σ-formula G, the relation **A satisfies G w.r.t. v**, written $A,v \models G$, is inductively defined as usual.

(i)	$A,v \models p(t_1,\dots,t_n)$	iff $<v^*(t_1),\dots,v^*(t_n)> \in p^A$ (for $p \in P$),
(ii)	$A,v \models (\neg G)$	iff $(A,v \models G)$ does not hold,
(iii)	$A,v \models (G \wedge H)$	iff $(A,v \models G)$ and $(A,v \models H)$,
(iv)	$A,v \models (\forall x{:}s.G)$	iff $(A,v_x \models G)$ for all valuations v_x: $X \to A$ with $v_x(z)=v(z)$ for $z \neq x$.

The interpretation of the relativized quantification is a consequence of this definition:

Fact 3.3 Let $\Sigma = < S, F, P >$ be a signature with standard symbols, $\Sigma' = < S', F' >$ with $S' \subseteq S$ and $F' \subseteq F$, $s \in S'$. Then for any Σ-structure A, valuation v: $X \to A$ and Σ-formula G

$A,v \models \forall x{:}\Sigma'_s.G$ iff $(A,v_t \models G)$ for all $t \in T(\Sigma')_s$ and all valuations v_t: $X \to A$ with

$$v_t(z) =_{def} \begin{cases} t^A & \text{for } z=x, \\ v(z) & \text{otherwise.} \end{cases}$$

The structure **A satisfies G,** written $A \models G$, if $A,v \models G$ holds for all valuations v.

A Σ-formula G is **valid** in a class K of Σ-structures if each $A \in K$ satisfies G.
The satisfaction relation is closed under isomorphism and w.r.t. the associated Σ-algebras.

Fact 3.4 Let A,B be two Σ- structures such that their associated Σ-algebras A /= and B /=

are isomorphic. Then the following four propositions are equivalent:

$A \models G$, $A/{=} \models G$, $B/{=} \models G$, $B \models G$.

Reachability can be expressed using "$\forall\exists$-formulas".

Fact 3.5 A Σ-structure A is Σ-reachable iff for each $s \in sorts(\Sigma)$, $A \models \forall x{:}s\ \exists y{:}\Sigma_s.x=y$.

With every class K of Σ-structures one may associate the **theory of** K, Th(K), i.e. the set of all Σ-formulas which are valid in K. For example as a consequence of fact 3.4., theories are invariant under isomorphism and transition of K to the class of Σ-algebras associated with K.
On the other hand, with every signature Σ and every set E of Σ-formulas one may associate the class Mod (Σ,E) of all **models** of E, i.e. of all Σ-structures which satisfy all formulas from E.

According to fact 3.5. for $E_0 =_{def} \{\forall x{:}s\ \exists y{:}\Sigma_s.x=y \mid s \in S\}$, $Mod(\Sigma,E_0)$ is the class of all Σ-reachable Σ-structures. Similarly, standard arithmetic can be axiomatized by Σ-formulas. Hence, Σ-formulas are able to express reachability and therefore there exists Σ_1, E_1 such that the theory of $Mod(\Sigma_1, E_1)$ is not recursively enumerable. Thus there cannot exist any formal system computing this theory, but there exists a **semi-formal** system for which a completeness result can be proven.

Let *F* be any formal system given by a set of **logical and nonlogical axioms** and a set of **inference rules** which is sound and complete for many-sorted first-order predicate logic with equality (cf. e.g. [Barwise 77]). Let " $\vdash$" denote the binary relation $E \vdash G$ which holds, if and only, if the Σ-formula G is **derivable** from E using *F*. Define a derivation relation $\vdash_I$ by adding for every Σ-formula G and every subsignature $\Sigma' = < S',F' >$ (with $S' \subseteq S$ and $F' \subseteq F$) and every sort $s \in S'$ the following induction rule II_s to the rules of *F*:

(II_s) Infinite Induction

If G[t/x] is derivable for all $t \in T(\Sigma')_s$, then so is $\forall x{:}\Sigma_s'.G$, i.e. for all sets E of Σ-formulas, $E \vdash_I G[t/x]$ for all $t \in T(\Sigma')_s$ implies $E \vdash \forall x{:}\Sigma'.G$.

The logical and non-logical axioms remain the same.
A Σ-formula is called **ω-derivable** from E if $E \vdash_I G$.

Theorem 3.6 (Completeness Theorem) Let E be a set of Σ-sentences. A Σ-formula G is ω-derivable from E (i.e. $E \vdash_I G$), if and only if, $Mod(\Sigma,E) \models G$.

4. Algebraic specifications

Following Hoare [Hoare 69], a specification is a formal documentation for a data type D which

guarantees properties of implementations of D for use in proving correctness of programs.
The syntax of a data type can be expressed by a signature Σ and the properties by Σ-formulas. This motivates the first part of the following definition.

Def. 4.1 Let $\Sigma = < S, F, P >$ be a signature with standard symbols.

(1) A **(flat) specification** SP consists of a pair $<\Sigma, E>$ where E is a set of Σ-sentences.

(2) A specification $SP = <\Sigma, E>$ is called **term-oriented** if all quantifiers occurring in E range over sets of ground terms; i.e. each quantified subformula in E has the form $\forall x{:}\Sigma_s{'}.G$ or $\exists x{:}\Sigma_s{'}.G$ with $\Sigma' = < S', F' >$ with $S' \subseteq S$ and $F' \subseteq F$, $s \in S'$.

(3) A specification $SP = <\Sigma, E>$ is called **Horn-specification**, if SP is term-oriented and all axioms of E are Horn-sentences (of the form $\forall x1{:}\ W1 \ldots \forall xn{:}\ Wn.\ G_1 \wedge \ldots \wedge G_m \Rightarrow G$ where $G_1, \ldots, G_m$ are atomic Σ-formulas).

The notation "term-oriented" is motivated by the "term generation principle" (cf. e.g. [Bauer, Wössner 82]) which requires that every element of a data type can be represented by a term. Thus properties of data types can always be reduced to properties of sets of ground terms.
Our notion of specification differs in two respects from the usual one. First syntactically, also predicates are allowed so that the notion of "logic program" would be appropriate as well. We will call a specification **equational Horn-specification** if all predicates in its axioms are equality symbols. Second semantically, the models of specifications may contain arbitrary "junk" (i.e. non-reachable elements) for which nothing is required by the axioms, and, moreover, the (non-standard) interpretation of the equality symbols allows for multiple representants of congruence classes; in particular, the set $T(\Sigma)$ of all terms is always an appropriate carrier set.
The signature of a specification $SP = <\Sigma,E>$ will be denoted by **sig(SP)** and the class of all models of SP will be denoted by **Mod(SP)**, i.e. $sig(SP) =_{def} \Sigma$ and $Mod(SP) =_{def} Mod(\Sigma,E)$.
The operators "sig" and "Mod" induce an equivalence relation on specifications.

Def. 4.2 Two specifications SP1 and SP2 are called **equivalent**, written SP1 = SP2, if sig(SP1) = sig(SP2) and Mod(SP1) = Mod(SP2).

For example, a flat specification $<\Sigma,E>$ is equivalent to the specification derived from the theory of their models.

Fact 4.3 Let $<\Sigma,E>$ be a flat specification. Then $<\Sigma,E> = <\Sigma,Th(Mod(<\Sigma,E>))>$.

In the following we use the notation

spec SP $\equiv$ **sorts** S **functions** F **predicates** P **axioms** E **endspec**

to denote a specification $<\Sigma,E>$, where $\Sigma =_{def} <S,F,P \cup St_{S,F}>$. Hence, the standard symbols are always implicitly contained in any signature Σ. We often omit the brackets "{" and "}" around sets of sorts, function symbols, predicate symbols and axioms.

Example 4.4

(1) **Total ordering**

The following specification consists of one sort s for which a total ordering relation $\leq$ is defined.

```
spec T0 ≡
        sort s
        predicates ≤: s,s
        axioms     ∀ x,y,z : s.
                   x≤x  ∧
                   (x≤y ∧ y≤z ⇒ x≤z) ∧
                   (x≤y ∧ y≤x ⇒ x=y) ∧
                   (x≤y ∨ y≤x)
endspec
```

All models of T0 are partial orderings with respect to the equality relation = which is implicitly declared. But note that in some of these models the interpretation of = does not coincide with the equality within the structure. For instance, the structure N3 defined by

$N3_s =_{def} \mathbb{N}$,

$n =^{N3} m \Leftrightarrow_{def} n \bmod 3 = m \bmod 3,\ n \leq^{N3} m \Leftrightarrow_{def} n \bmod 3 \leq m \bmod 3,$

is a model of T0 where the equality relation $=^{N3}$ does not coincide with the equality in $N3_s$.

(2) **Truth values**

The following specification BOOL0 consists of two (different) constants "true", "false" and the unary boolean function symbol "not".

```
spec BOOL0 ≡
        sort Bool
        functions  true, false :→Bool,
                   not: Bool → Bool
        axioms     true ≠ false,
                   not(true) = false,
                   ∀x: ΣB0_Bool. not(not(x)) = x
endspec
```

All models M of BOOL0 have at least two elements "$true^M$" and "$false^M$" representing "true" and "false". Moreover, the interpretation of "not" has the usual meaning on these two elements, but otherwise, it may have completely arbitrary values. E.g. the structures B, N1 and N2 (see section 2.2) are models of BOOL0. For reachable models of BOOL0, the two constants true and false are **constructors**: all ground terms $t \in T(\Sigma B0)$ are equivalent to one of these constants; i.e. the formula

$\forall x : \Sigma B0_{Bool} \exists y: < \{Bool\}, \{true,false\} >_{Bool}.\ x=y$

holds in BOOL0.

The third axiom is equivalent to the following infinite number of ground equations.

$$\text{not}(\text{not}(t)) = t \quad \text{for } t \in \{\text{not}^k(\text{true}) \mid k \geq 0\} \cup \{\text{not}^k(\text{false}) \mid k \geq 0\}\ .$$

All these equations can be deduced from the second axiom together with the equation not(false)=true. Hence BOOL0 is equivalent to the following specification BOOL1.

```
spec BOOL1 ≡
        sort  Bool
        functions true, false:→Bool,
                  not: Bool→Bool
        axioms    true ≠ false,
                  not(true) = false,
                  not(false) = true
endspec
```

The following specification BOOL2 differs from BOOL0 just by quantifying over all elements, not only over terms.

```
spec BOOL2 ≡
        sort  Bool
        functions true, false:→Bool,
                  not: Bool→Bool
        axioms    true ≠ false,
                  not(true) = false,
                  ∀x: Bool. not(not(x)) = x
endspec
```

All models of BOOL2 are also models of BOOL1, but not vice versa: N2 is not a model of BOOL2. □

Quantification ranging over the set of all (interpretations of) constructor terms as in BOOL0 is common for algebraic specifications and deserves a special notation: the set C of constructor function symbols of a specification SP will be indicated by the keyword "**constructors**" and axioms of the form Q x: $<S,C>_s$.G with Q∈ {∀,∃} will be written as Q **cons** x:s.G. We write "**cons**$_{SP1}$" if **cons** refers to the set of constructors of another specification SP1.

Example 4.5

(1) **Truth values (continued)**

The following is an equivalent notation for BOOL0:

```
spec BOOL3 ≡
        sort Bool
        constructors true, false :→ Bool
        functions    not: Bool → Bool
        axioms       true≠false,
                     not(true) = false,
                     ∀ cons x: Bool. not(not(x)) = x
    endspec
```

(2) Natural numbers with ordering

The following specification describes natural numbers together with the usual "less-or-equal"-relation.

```
    spec NAT0 ≡
        sort Nat
        constructors    0 :→ Nat,
                        succ: Nat → Nat
        functions       .+. : Nat,Nat → Nat
        predicate       ≤ : Nat, Nat
        axioms          ∀ cons m,n:Nat.
                        0≠succ(n) ∧ (succ(n) = succ(m) ⇒ n=m) ∧
                        n + 0 = n  ∧ n + succ(m) = succ(n+m) ∧
                        (n≤m ⇔ ∃ cons b: Nat . n+b=m)
    endspec
```

The function symbols 0 and succ are constructors, i.e. the quantification

"$\forall$ **cons** m,n:Nat" is equivalent to "$\forall$ m,n: $<\{Nat\},\{0,succ\}>_{Nat}$".

The (up to isomorphism) only standard model N0 of NAT0 consists of the standard natural numbers together with the usual less-or-equal-relation:

$N0_{Nat} =_{def} \mathbb{N}$, $0^{N0} =_{def} 0$, $succ^{N0}(n) =_{def} n+1$, $n+^{N0}m =_{def} n+m$, $n\leq^{N0}m$ iff $n\leq m$.

Another (non-standard) model is the algebra Z0 of negative integers, i.e.

$Z0_{Nat} =_{def} \mathbb{Z}$, $0^{Z0} =_{def} 0$, $succ^{Z0}(n) =_{def} n-1$, $n+^{Z0}m =_{def} n+m$, $n\leq^{Z0}m$ iff $|n|\leq|m|$.

Notice that for Z0 the last axiom holds (for constructor terms) but it does not hold for all elements: $0 \leq^{Z0} 1$ but there does not exist a term t of the form $succ^k(0)$, $k\geq0$, such that $0+t^{Z0} = 1$ since each t^{Z0} is a non-positive integer.

As a third example for a model of NAT0 consider natural numbers with a bottom element. The algebra $N\perp$ is defined as follows:

$N\perp_{Nat} =_{def} \mathbb{N} \cup \{\perp\}$, $0^{N\perp} =_{def} 0$, $n\leq^{N\perp} m$ iff $n\neq\perp \wedge m\neq\perp \wedge n\leq m$.

$succ^{N\perp}(n)$	$=_{def} n+1$,	if $n\neq\perp$;	and $\perp$, otherwise,
$n+^{N\perp} m$	$=_{def} n+m$,	if $n\neq\perp \wedge m\neq\perp$;	and $\perp$, otherwise.

(3) Integers

Integers can be specified similar to natural numbers. In two cases, quantification ranges over constructors for natural numbers, not for integers.

```
    spec INT ≡
        sort Int
        constructors 0 :→ Int
                     pred, succ: Int→ Int
        functions    .+. : Int, Int→ Int
```

predicate	$\leq$: Int, Int
axioms	$\forall n: <\{Int\}, \{0,succ\}>_{Int} .\ 0 \neq succ(n)$
	$\forall$ **cons** y,z: Int.
	$pred(succ(z)) = z \wedge succ(pred(z)) = z \wedge$
	$z+0 = z \wedge$
	$z + succ(y) = succ(z+y) \wedge$
	$z+pred(y) = pred(z+y) \wedge$
	$(y \leq z \Leftrightarrow \exists n : T(<\{Int\}, \{0,succ\}>) .\ y+n = z)$

endspec

As for natural numbers, the standard model Z1 of integers is up to isomorphism the only such model of INT.

$Z1_{Int} =_{def} \mathbb{Z}$,

$0^{Z1} =_{def} 0$, $succ^{Z1}(z) =_{def} z+1$, $pred^{Z1}(z) =_{def} z-1$, $y+^{Z1}z =_{def} y+z$,

$y \leq^{Z1} z$ iff $y \leq z$.

The algebra Z0 (which is a model of NAT0, cf. 4.5(2)) can be extended to a sig(INT)-algebra by choosing the obvious definition for $pred^{Z0}$. However, it is not a model of INT since it does not satisfy the last axiom.

Similar to $\mathbb{N}\perp$, one can define the algebra $\mathbb{Z}\perp$ of integers with bottom element:

$\mathbb{Z}\perp_{Int} =_{def} \mathbb{Z} \cup \{\perp\}$, $0^{\mathbb{Z}\perp} =_{def} 0$, $y \leq^{\mathbb{Z}\perp} z$ iff $y \neq \perp \wedge z \neq \perp \wedge y \leq z$,

$succ^{\mathbb{Z}\perp}(z) =_{def} z+1$,	if $z \neq \perp$;	and $\perp$, otherwise,
$pred^{\mathbb{Z}\perp}(z) =_{def} z-1$,	if $z \neq \perp$;	and $\perp$, otherwise,
$y +^{\mathbb{Z}\perp} z =_{def} y+z$,	if $y \neq \perp \wedge z \neq \perp$;	and $\perp$, otherwise

5. Structured specifications

For writing large specifications it is convenient to design specifications in a structured fashion by combining and modifying smaller specifications. This supports a modular decomposition into specifications of manageable size and helps to master the complexity originating from a large number of (function and predicate) symbols and axioms.

In the following we introduce three specification building operators which are derived from the operators of the specification language ASL [Sannella, Wirsing 83]. For this language an institution-independent semantics has been given by [Sannella, Tarlecki 85a] which makes it easy to derive (more exactly to instantiate) a semantics appropriate for our approach.

The first operator "derive from • by •" will be used for renaming, hiding, exporting and copying.

Def. 5.1 Let SP1 be a specification with signature $\Sigma1$ and let $\sigma:\Sigma \rightarrow \Sigma1$ be a signature morphism, then

derive from SP1 **by** σ

denotes the specification SP defined by

$sig(SP) =_{def} \Sigma$,

$Mod(SP) =_{def} \{A|\sigma \in Struct(\Sigma) \mid A \in Mod(SP1)\}$.

Using "derive", renaming of specifications can be defined as follows: if SP1 is a specification and $[x_1/y_1, \ldots, x_n/y_n] : \Sigma \rightarrow sig(SP1)$ denotes a renaming (cf. section 2.1), then

rename SP1 **by** $[x_1/y_1, \ldots, x_n/y_n] =_{def}$ **derive from** SP1 **by** $[x_1/y_1, \ldots, x_n/y_n]$

denotes a specification SP which is the same as SP1 but with $y_1, \ldots, y_n$ renamed into $x_1, \ldots, x_n$. The second operator "translate . with ." is the converse of "derive" [Sannella, Tarlecki 87].

Def. 5.2. Let SP1 be a specification with signature $\Sigma 1$ and $\sigma: \Sigma 1 \rightarrow \Sigma$ be a signature morphism, then

translate SP1 **with** σ

denotes the specification SP defined by

$sig(SP) =_{def} \Sigma$,

$Mod(SP) =_{def} \{A \in Struct(\Sigma) \mid A|\sigma \in Mod(SP1)\}$.

The third operator, called "+", combines two specifications SP1 and SP2 (without taking care of name clashes).

Def. 5.3 Let SP1 and SP2 be two specifications with signatures $\Sigma 1$ and $\Sigma 2$ resp..
Then SP1 + SP2 denotes a specification SP defined by

$sig(SP) =_{def} \Sigma 1 \cup \Sigma 2$,

$Mod(SP) =_{def} \{A \in Struct(sig(SP)) \mid A|\Sigma 1 \in Mod(SP1) \text{ and } A|\Sigma 2 \in Mod(SP2)\}$.

The models of SP1+SP2 can be defined in terms of "translate":

$Mod(SP1+SP2) = Mod($**translate** SP1 **with** ${}_{\Sigma 1}in) \cap Mod($**translate** SP2 **with** ${}_{\Sigma 2}in)$

where ${}_{\Sigma i}in: \Sigma i \rightarrow sig(SP1 + SP2)$, i=1,2, is the canonical embedding, defined by ${}_{\Sigma i}in(x)=x$ for $x \in \Sigma i$.

As an example for an operator which can be explicitly defined using "+" we introduce simple enrichments:
Let SP1 be a specification with signature $\Sigma 1 = < S1, F1, P1 >$ and let S,F,P,W be sets of sorts, function symbols, predicate symbols and formulas respectively.
Then

enrich SP1 **by sorts** S **functions** F **predicates** P **axioms** E

denotes the specification $SP =_{def} SP1 + <\Sigma,E>$ where Σ is the union of the signature $\Sigma 1$ with the new symbols and with new implicitly defined equality symbols and generating predicate symbols for the new sorts, i.e. $\Sigma =_{def} < S1 \cup S,\ F1 \cup F,\ P1 \cup P \cup St_{S1 \cup S, F1 \cup F} >$.
Notice that the equality symbols of Σ are interpreted as Σ-congruence relations, whereas in SP1 the same equality symbols (on the sorts of S1) are interpreted as $\Sigma 1$-congruence relations.
Thus, for $=_s$, $s \in S1$, implicitly new substitution axioms have been added.
If, as before, the constructor notation is used, then in E "**cons**" denotes the set

$T(<S1 \cup S, C1 \cup C>)$ of ground terms where C1 is the set of constructor functions of SP1 and C is the set of constructor functions introduced by the enrichment. In the set E1 of axioms of SP1, **cons** denotes the set $T(< S1, C1 >)$ of ground SP1-constructor terms.

Example 5.4

(1) **Finite sets of natural numbers**

The following specification describes finite sets of natural numbers.

```
spec SETNAT ≡
        enrich NAT0 by
                sort Set
                constructors empty:→ Set,
                             add: Nat, Set → Set
                predicates   .ε. : Nat, Set
                axioms       ∀ cons m,n: Nat,  s : Set .
                             ¬(n ε empty) ∧
                             (nεadd(m,s) ⇔ (n=m ∨ nεs)) ∧
                             add(n,add(n,s)) = add(n,s) ∧
                             add(n,add(m,s)) = add(m,add(n,s))
endspec
```

The (up to isomorphism) only standard model S0 of SETNAT consists of finite sets of natural numbers; i.e.

$S0 \mid NAT0 =_{def} N0$, $S0_{Set} =_{def} \{s \subseteq \mathbb{N} \mid s \text{ is finite}\}$

$empty^{S0} =_{def} \emptyset$, $add^{S0}(m,s) =_{def} \{m\} \cup s$, $n\, \varepsilon^{S0}\, s$ iff $n \in s$.

The power set of integers and ordered sequences of natural numbers form two other models S1 and S2:

$S1 \mid NAT0 =_{def} Z0$, $S1_{Set} =_{def} \{s \mid s \subseteq \mathbb{Z}\}$

$empty^{S1} =_{def} \emptyset$, $add^{S1}(m,s) =_{def} \{m\} \cup s$, $n\, \varepsilon^{S1}\, s$ iff $n \in s$;

$S2 \mid NAT0 =_{def} N0$, $S2_{Set} =_{def} \{<n_1,\ldots,n_k> \mid k \geq 0 \wedge n_1 \leq \ldots \leq n_k\}$,

$empty^{S2} =_{def} <>$ "the empty sequence",

$add^{S2}(n,<n_1,\ldots,n_k>) =_{def} <n_1,\ldots,n_{i-1},n,n_i,\ldots,n_k>$ if $n_{i-1} \leq n \leq n_i$ for some $i \in \{1,\ldots,k\}$,

$n\varepsilon^{S2}<n_1,\ldots,n_k>$ iff there exists $i \in \{1,\ldots,k\} : n=n_i$.

For S2, the interpretation of the equality symbol $=_{Set}$ does not coincide with the equality of sequences:

for all $s,s' \in S2_{Set}$, $s(=_{Set})^{S2}s'$ iff for all $n \in \mathbb{N}$: $n\varepsilon^{S2}s \Leftrightarrow n\varepsilon^{S2}s'$.

(2) **Finite sequences of integers**

Finite sequences of integers can be specified as follows

```
spec SEQINT ≡
        enrich INT by
                sort Seq
                constructors emptyseq:→ Seq,
                         <.>: Int→ Seq,
                         .°.: Seq, Seq → Seq
          functions      first: Seq→Int,
                         rest: Seq→Seq
          axioms         ∀ cons x: Int,  s,s1,s2 : Seq .
                         emptyseq ° s = s ∧
                         s ° emptyseq = s ∧
                         s ° (s1 ° s2) = (s ° s1) ° s2 ∧
                         first(<x> ° s) = x ∧
                         rest (<x> ° s) = s
endspec
```

SEQINT has infinitely many standard models since the values of "first(emptyseq)" and "rest(emptyset)" are not fixed by the axioms. The algebra S⊥ of sequences with bottom element is a standard model, whereas the structure PA of arrays with pointers is a non-standard model of SEQINT.

$S\bot \mid \mathrm{INT} =_{def} \mathbb{Z}\bot$, $S\bot_{Seq} =_{def} \mathbb{Z}^* \cup \{\bot_{Seq}\}$,

$\mathrm{emptyseq}^{S\bot} =_{def} <>$,

$<z>^{S\bot}$	$=_{def} <z>$,	if $z \neq \bot$;	and $\bot_{Seq}$, otherwise;
$s \circ^{S\bot} s'$	$=_{def} <y_1,\ldots,y_k,z_1,\ldots,z_m>$,	if $s=<y_1,\ldots,y_k> \in \mathbb{Z}^* \wedge s'=<z_1,\ldots,z_m> \in \mathbb{Z}^*$;	and $\bot_{Seq}$, otherwise;
$\mathrm{first}^{S\bot}(s)$	$=_{def} x$,	if $s=<x,x_1,\ldots,x_k> \in \mathbb{Z}^*$;	and $\bot$, otherwise;
$\mathrm{rest}^{S\bot}(s)$	$=_{def} <x_1,\ldots,x_k>$,	if $s=<x,x_1,\ldots,x_k> \in \mathbb{Z}^*$;	and $\bot_{Seq}$, otherwise.

Each non-bottom element of the carrier set PA_{Seq} of arrays with pointers is represented by a pair, consisting of a natural number (called pointer) and a mapping $\alpha \in [\mathbb{N}_+ \to \mathbb{Z}]$ from the positive natural numbers $\mathbb{N}_+ =_{def} \mathbb{N}\setminus\{0\}$ into the integers.

$PA \mid \mathrm{INT} =_{def} \mathbb{Z}\bot$, $PA_{Seq} =_{def} (\mathbb{N} \times [\mathbb{N}_+ \to \mathbb{Z}]) \cup \{\bot_{Seq}\}$

$\mathrm{emptyseq}^{PA} =_{def} <0,\lambda x.0>$

$<z>^{PA} =_{def} <1,\alpha_2>$, if $z \neq \bot$; and $\bot_{Seq}$, otherwise;

where $\alpha_2(x) =_{def} z$, if $x=1$; and 0, otherwise;

$s \circ^{PA} s' =_{def} <n+m,\gamma>$, if $s=<n,\alpha>$ and $s'=<m,\beta>$; and $\bot_{Seq}$, otherwise;

where $\gamma(x) =_{def} \alpha(x)$, if $1 \leq x \leq n$; and $\beta(x-n)$, if $n < x \leq n+m$; and 0, otherwise;

$\mathrm{first}^{PA}(s) =_{def} z$,	if $s=<n,\alpha>$ and $\alpha(n)=z$;	and $\bot$,	otherwise;
$\mathrm{rest}^{PA}(s) =_{def} <n-1,\alpha>$,	if $s=<n,\alpha>$;	and $\bot_{Seq}$,	otherwise.

The equality symbol is interpreted in PA in a non-standard way:

$s\ (=_{Seq})^{PA}\ s'$ iff $s=s'=\bot_{Seq}$ or there exist n, α, β such that
$s=<n,\alpha>$, $s'=<n,\beta>$ and $\forall x \in \{1,\ldots,n\}.\ \alpha(x) = \beta(x)$.

6. Properties of specification operators

A number of other specification building operators can be defined using <.,.>, + and derive.

Example 6.1

(1) Exporting and hiding can be explicitly defined as follows. Let SP1 be a specification with

signature $\Sigma 1 = <S1,F1,P1>$ and let $\Sigma = <S,F,P> \subseteq sig(SP1)$ be a signature. Then

export Σ **from** SP1 $=_{def}$ **derive from** SP1 **by** $_{\Sigma}in$

where $_{\Sigma}in: \Sigma \rightarrow sig(SP1)$ is the canonical embedding from Σ into sig(SP1), i.e. $_{\Sigma}in(x)=x$ for all $x \in \Sigma$. Hiding can be explicitly defined in terms of "export".

Let $s_0 \in S1$ be a sort of SP1 and $f_0 \in F1 \cup P1$ be a function or predicate symbol of SP1. Then hiding s_0 means to forget s_0 and all function and predicate symbols with s_0 in their domain or range, whereas hiding f_0 means just to forget f_0.

hide s_0 **from** SP1 $=_{def}$ **export** $\Sigma 1\text{-}s_0$ **from** SP1,

hide f_0 **from** SP1 $=_{def}$ **export** $\Sigma 1 \setminus \{f_0\}$ **from** SP1,

where $\Sigma 1\text{-}s_0 =_{def} < S1^-, \{F1_{w,s}\}_{w \in (S1^-)^*, s \in S1^-}, \{P1_w\}_{w \in (S1^-)^*} >$, and $S1^- =_{def} S1 \setminus \{s_0\}$.

(2) Exporting and renaming can be combined by the following specification operator which exports Σ after renaming $y_1,\dots,y_n$ into $x_1,\dots,x_n$.

export Σ **with** $[x_1/y_1,\dots,x_n/y_n]$ **from** SP $=_{def}$ **derive from** SP **by** $_{\Sigma}[x_1/y_1,\dots,x_n/y_n]_{sig(SP)}$

where $_{\Sigma}[x_1/y_1,\dots,x_n/y_n]_{sig(SP)}: \Sigma \rightarrow sig(SP)$ denotes the signature morphism induced by $[x_1/y_1,\dots,x_n/y_n]$ with domain Σ and range sig(SP) (cf. section 2.1).

(3) Non-injective signature morphisms allow for the copying of structures.

As example, we consider the specification INT of 4.4(3) with signature sig(INT).

We extend sig(INT) by a new sort Nat, function symbols 0: $\rightarrow$Nat, succ: Nat$\rightarrow$Nat and +: Nat,Nat$\rightarrow$Nat, predicate symbols $\leq$: Nat,Nat and standard symbols. Call this signature ΣINT-NAT and consider the signature morphism

copy: ΣINT-NAT$\rightarrow$sig(INT) defined by

copy(Nat) $=_{def}$ Int, copy(0) $=_{def}$ 0, copy(succ) $=_{def}$ succ, copy(+) $=_{def}$ +,

copy($\leq$) $=_{def}$ $\leq$, copy($=_{Nat}$) $=_{def}$ $=_{Int}$, copy(x) $=_{def}$ x for $x \in sig(INT)$.

Then

derive from INT **by** copy

denotes a specification with signature ΣINT-NAT. Each model M of this specification consis of a model of INT together with a copy of itself the carrier set of which is named "Nat" but where the operation pred is missing. Hence, the carrier set M_{Nat} is always isomorphic to a superset of the natural numbers, it is never the set $\mathbb{N}$ of all standard natural numbers. A predecessor function for natural numbers can be introduced as follows.

```
spec INTNAT ≡
    enrich
          derive from INT by copy
    by    functions pred: Nat→Nat
          axioms pred(0)=0, ∀cons_NAT0 x: Nat. pred(succ(x))=x
endspec
```

Here "$\forall$**cons**$_{NAT0}$ x" is an abbreviation for quantification over the constructors of Nat0, i.e. for $\forall x$: $<\{Nat\},\{0,succ\}>_{Nat}$.

The specification operators satisfy a number of algebraic identities which are useful for transforming specification expressions. Such identities are given e.g. in [Sannella, Wirsing 83]. A comprehensive study of an algebra of specification expressions can be found in [Bergstra et al. 86].

For example, let the signature morphism $\sigma:\Sigma\rightarrow\Sigma'$ be given as well as the specifications SP and SP′ with signatures Σ and Σ' resp. Then we have

Mod(**translate (derive from** SP′ **by** σ) **with** σ) =

$\{A' \in \mathrm{Struct}(\Sigma') \mid A'|\sigma \in$ Mod(**derive from** SP′ **by** σ)$\}$ =

$\{A' \in \mathrm{Struct}(\Sigma') \mid A'|\sigma \in \{A|\sigma \in \mathrm{Struct}(\Sigma) \mid A \in \mathrm{Mod}(SP')\}\}$

This shows that Mod(**translate (derive from** SP′ **by** σ) **with** σ) $\supseteq$ Mod(SP′). We even have

translate (derive from SP′ **by** σ) **with** σ = SP′

provided σ is surjective. If σ is not surjective, then the carrier sets for sorts and the interpretations for function symbols that are not in the range of σ can be choosen arbitrarily.

Vice versa we have:

Mod(**derive from (translate** SP **with** σ) **by** σ) =

$\{A'|\sigma \in \mathrm{Struct}(\Sigma) \mid A' \in$ Mod(**translate** SP **with** σ)$\}$ =

$\{A'|\sigma \in \mathrm{Struct}(\Sigma) \mid A' \in \{A' \in \mathrm{Struct}(\Sigma) \mid A'|\sigma \in \mathrm{Mod}(SP)\}\}$ =

$\{A'|\sigma \in \mathrm{Struct}(\Sigma) \mid A'|\sigma \in \mathrm{Mod}(SP)\}$

This shows that Mod(**derive from (translate** SP **with** σ) **by** σ)$\subseteq$Mod(SP). We even have

(**derive from (translate** SP **with** σ) **by** σ) = SP

provided σ is injective. If σ is not injective then those models of SP are excluded in (**derive from (translate** SP **with** σ) **by** σ) in which sorts (or function symbols or predicates) that occur twice in the range are associated with different carrier sets (functions or predicates resp.).

Fact 6.2 Let SP, SP1 and SP2 be specifications.

(1) SP+SP = SP,
SP1+SP2 = SP2+SP1,
SP+(SP1+SP2) = (SP+SP1)+SP2,

(2) if σ_1: $\Sigma_1\rightarrow$sig(SP), σ_2: $\Sigma_2\rightarrow\Sigma_1$ are signature morphisms, then
derive from (derive from SP **by** σ_1) **by** σ_2 = **derive from** SP **by** $\sigma_1 \circ \sigma_2$

(3) if σ is an injective mapping with range(σ)$\subseteq$sig(SP2) inducing the signature morphisms
${}_{\Sigma}\sigma$: $\Sigma\rightarrow$sig(SP2), ${}_{\mathrm{sig(SP1)}}\sigma$: sig(SP1)$\rightarrow\Sigma'$ (where $\Sigma' = {}_{\mathrm{sig(SP1)}}\sigma(\mathrm{sig(SP1)})$),
${}_{\mathrm{sig(SP1)}\cup\Sigma}\sigma$: sig(SP1)$\cup\Sigma\rightarrow\Sigma'\cup$sig(SP2) (for the notions see section 2.1), then
SP1+**derive from** SP2 **by** ${}_{\Sigma}\sigma$ =
derive from (translate SP1 **with** ${}_{\mathrm{sig(SP1)}}\sigma$)+SP2 **by** ${}_{\mathrm{sig(SP1)}\cup\Sigma}\sigma$

provided that the hidden symbols (sig(SP2)$\setminus{}_{\Sigma}\sigma(\Sigma)$) do not interfere with SP1 (i.e.

$(sig(SP1)\backslash\Sigma) \cap (sig(SP2)\backslash_{\Sigma}\sigma(\Sigma)) = \emptyset)$.

Corollary 6.3 For all specifications SP, enrichments Δ, $\Delta1$, signatures $\Sigma, \Sigma1$, substitution lists $\alpha, \alpha1$ such that the following expressions are well-defined, the following holds.

(1) **enrich (enrich SP by Δ) by $\Delta1$ = enrich SP by $\Delta \cup \Delta1$,**

(2) if $_{\Sigma}\alpha$: $\Sigma \rightarrow sig(SP)$ and $_{\Sigma1}\alpha1$: $\Sigma1 \rightarrow \Sigma$ are signature morphisms induced by α and $\alpha1$ (cf. section 2.1), then
export $\Sigma1$ with $\alpha1$ from (export Σ with α from SP) =
export $\Sigma1$ with $\alpha \circ \alpha1$ from SP,

(3) if $sig(SP) \cap sig(\alpha(\Delta)) = \emptyset$, then
enrich (export Σ with α from SP) by Δ =
export $\Sigma \cup sig(\Delta)$ with α from (enrich SP by $\alpha(\Delta)$)
where $sig(\Delta)$ denotes the sorts, function and predicate symbols from Δ and where $\alpha(\Delta)$ denotes the substitution of $x_1, \ldots, x_n$ by $y_1, \ldots, y_n$ in Δ, i.e. $\alpha = [x_1/y_1, \ldots, x_n/y_n]$.

Flat specifications are equivalent to their theory (cf. fact 4.3). The result of the combination of two flat specifications by the operator "+" is also a flat specification.

Fact 6.4 Let SP1 = < S1, F1, E1 > and SP2 = < S2, F2, E2 > be two flat specifications. Then
SP1+SP2 = < Σ, E1$\cup$E2 >
where $\Sigma =_{def} < S1 \cup S2, F1 \cup F2, P1 \cup P2 \cup St_{S1 \cup S2, F1 \cup F2} >$.

On the other hand as a consequence of the "hiding facility" of the operator "derive", not all specifications (using derive) are equivalent to a flat specification. Examples for data structures that can be specified using "hidden functions" (i.e. with "derive") but not as flat specifications (with finitely many axioms) are given e.g. in [Majster 77], [Bergstra, Tucker 86] and [Wirsing 86]. In this case, a specification may not be equivalent to the theory of their models.

Fact 6.5 For all specifications SP, $Mod(SP) \subseteq Mod(Th(Mod(SP)))$.

There exist examples for which this inclusion is strict. Therefore, the theory operator is a closure operator which abstracts from the particular models of a specification and considers only the properties of the class of all models.

7. Implementation

The programming discipline of stepwise refinement suggests that a program be evolved from a high-level specification by working gradually via a series of successively more detailed lower-level intermediate specifications. A formalization of this approach requires a precise definition of the concept of refinement, i.e. of the implementation of a specification by another. A specification SP′ implements another specification SP if it has the same signature and all its models are models of SP.

Def 7.1 Let SP and SP′ be two specifications. SP′ is an **implementation** of SP, written SP ~~~> SP′, if sig(SP)=sig(SP′) and Mod(SP′)$\subseteq$Mod(SP).

This notion of implementation (used e.g. by [Sannella, Wirsing 83], [Sannella, Tarlecki 87]) is conceptually simpler than other notions known from the literature such as "forget-restrict-identify"-implementations (cf. e.g. [Ehrig et al. 82]). Our notion represents the idea that a specification describes those properties which have to be respected by the implementations. For flat specifications, the implementation relation can be characterized as follows.

Fact 7.2 Let SP=<Σ,E> and SP′=<Σ,E′> be two flat specifications with the same signature. Then the following properties are equivalent:

(i) SP ~~~> SP′,

(ii) $E' \vdash_I e$ for all $e \in E$

(iii) Mod(SP′)$\subseteq$Mod(SP).

In the following we give a number of simple examples for implementations. Here it is often necessary to define congruence relations w.r.t the signature of the specification to implement. In order to avoid writing standard axioms we introduce the following notation. Let SP be a specification with signature Σ. We write

equalities $=_{SP,s}$: s,s for $s \in$ sorts(Σ)

as abbreviation for the following predicate symbols and axioms:

predicates $=_{SP,s}$: s,s for $s \in$ sorts(Σ)
together with the additional axioms for a Σ-congruence:
$\forall$**cons**$_{SP}$ x:s. $x =_{SP,s} x$, for all $s \in S$,
$\forall$**cons**$_{SP}$ x_1,y_1:s1, ... ,x_n,y_n:sn. $x_1 =_{SP,s1} y_1 \wedge \ldots \wedge x_n =_{SP,sn} y_n \Rightarrow f(x_1,\ldots,x_n) =_{SP,s} f(y_1,\ldots,y_n)$
for all f: s1,...,sn$\to$s $\in$ opns(Σ),
$\forall$**cons**$_{SP}$ x_1,y_1:s1, ... ,x_n,y_n:sn. $x_1 =_{SP,s1} y_1 \wedge \ldots \wedge x_n =_{SP,sn} y_n \wedge p(x_1,\ldots,x_n) \Rightarrow p(y_1,\ldots,y_n)$
for all p: s1,...,sn $\in$ preds(Σ).

Example 7.3 Implementations

(1) Truth-values by natural numbers

The specification BOOL3 of truth-values can be implemented by extending the specification NAT0, renaming Nat into Bool and exporting the signature of BOOL3.

```
spec BOOL_by_NAT ≡
      export ΣB0 with [Bool/Nat, true/0, not/succ, $=_{Bool}/=_{BOOL3,Nat}$] from
            enrich NAT0
            by  functions   false: →Nat
                equalities  $=_{BOOL3,Nat}$: Nat, Nat
```

```
                axioms     false=succ(0),
                           ∀cons x: Nat. succ(succ((x)) $=_{BOOL3,Nat}$ x
    endspec
```

The equality symbol for truth values has to be added to NAT0. It is <u>not</u> a congruence w.r.t. the signature of NAT0; in particular, the predicate "≤" does not satisfy the substitution property w.r.t. $=_{BOOL3,Nat}$. Nevertheless, it is easy to see that all axioms of BOOL3 hold in BOOL_by_NAT. Hence, BOOL3 ~~~> BOOL_by_NAT holds.

(2) <u>Natural numbers by integers</u>

Natural numbers are implemented by integers in the obvious way.

```
    spec NAT_by_INT ≡
        export sig(NAT0) with [Nat/Int]
            from INT
    endspec
```

Using the "export-after-rename"-operator the sort "Int" is renamed into "Nat" and the operation "pred" is forgotten. The equality relation for integers is the same as for natural numbers. The axioms of NAT0 hold in NAT_by_INT: NAT0 ~~~> NAT_by_INT.

(3) <u>Finite sets by ordered sequences</u>

Finite sets of natural numbers can be implemented by ordered sequences of integers as follows.

```
    spec SET_by_ORDSEQINT ≡
        export sig(SETNAT) with [Set/Seq, Nat/Int, $=_{SET}/=_{SETNAT}$] from
        enrich SEQINT
        by  functions   empty: →Seq,
                        add: Int, Seq→Seq
            predicates  .ε.: Int, Seq
            equalities  $=_{SETNAT,Seq}$: Seq, Seq,
                        $=_{SETNAT,Int}$: Int, Int
            axioms      empty=emptyseq,
                        ∀cons x,y: Nat, s: Set.
                            add(x,emptyseq)=<x> ∧
                            (x≤y ⇒ add(x,<y>°s)=<x>°<y>°s) ∧
                            (y≤x ∧ y≠x ⇒ add(x,<y>°s)=<y>°add(x,s)) ∧
                            ¬(x ε emptyseq) ∧
                            [x ε (<y>°s) ⇔ (x=y ∨ (y≤x ∧ x ε s))],
                        ∀x,y:<{Int},{0,succ}>$_{Int}$. x $=_{SETNAT,Int}$ y ⇔ x=y,
                        ∀x:<{Int},{0,succ}>$_{Int}$, s:<{Seq,Int},{0,succ,empty,add}>$_{Seq}$.
                            add(x,add(x,s)) $=_{SETNAT,Seq}$ add(x,s)
    endspec
```

Natural numbers are represented by positive integers, sets are represented by ordered sequences with repetition. As a consequence, the specification SET_by_ORDSEQINT satisfies the axiom

(*) add(x,add(y,s)) = add(y,add(x,s))

for all ordered sequences of integers representing the left-hand side or (equivalently) the right-hand side of this equation. The other characteristic equation for sets

add(x,add(x,s)) = add(x,s)

does not hold in the enrichment part of SET_by_ORDSEQINT for the equality between sequences; but according to the last axiom of SET_by_ORDSEQINT the equation

$\mathrm{add}(x,\mathrm{add}(x,s)) =_{\mathrm{SETNAT,Seq}} \mathrm{add}(x,s)$

holds for all those sequences which represent sets of natural numbers, i.e. for all (interpretations of) ordered sequences of positive integers. Therefore, the equation (*) holds in SET_by_ORDSEQINT (after the renaming for all constructor terms of SETNAT). All other axioms of SETNAT can easily be derived from the other axioms of SET_by_ORDSEQINT. Hence SET_by_ORDSEQINT is an implementation of SETNAT, i.e.

SETNAT ~~~> SET_by_ORDSEQINT.

The implementation relation is a partial ordering on the class of all specifications and the specification operators are monotonic.

Theorem 7.4

(1) The implementation relation is a partial ordering w.r.t. the equivalence = of specifications: for all specifications SP, SP1 and SP2,

(i) SP ~~~> SP

(ii) SP ~~~> SP1 and SP1 ~~~> SP implies SP = SP1

(iii) SP ~~~> SP1 and SP1 ~~~> SP2 implies SP ~~~> SP2

(2) The specification operators are monotonic; i.e. for all signatures Σ, sets of Σ-formulas E1, E2, specifications SP1, SP2, SP1´, SP2´ and signature morphisms $\sigma:\Sigma \rightarrow \mathrm{sig(SP1)}$, $\sigma':\mathrm{sig(SP1)} \rightarrow \Sigma'$ the following holds:

(i) $E1 \subseteq E2$ implies $<\Sigma,E1>$ ~~~> $<\Sigma,E2>$,

(ii) SP1 ~~~> SP1´ implies (**derive from** SP1 **by** σ) ~~~> (**derive from** SP1´ **by** σ),

(iii) SP1 ~~~> SP1´ implies (**translate** SP1 **with** σ') ~~~> (**translate** SP1´ **with** σ'),

(iv) SP1 ~~~> SP1´ and SP2 ~~~> SP2´ implies SP1+SP2 ~~~> SP1´+SP2´.

Hence in particular, the implementation relation is "horizontally" and "vertically" composable [Burstall, Goguen 80], since vertical composition corresponds to the transitivity and horizontal composition corresponds to the monotonicity of the implementation relation.

Common techniques for defining the notion of implementation of algebraic specifications can be seen as particular case of our notion. We call a specification SP' an <u>FRI-implementation</u> of SP (FRI stands for "<u>f</u>orget-<u>r</u>estrict-<u>i</u>dentify") if sig(SP)$\subseteq$sig(SP') and if there exists a sig(SP)-congruence relation $\sim_i$ on T(sig(SP)) such that for all standard models $A' \in \mathrm{Gen(SP')}$, the Σ-standard structure

$<A'>_\Sigma/\sim_i$ is a model of SP; SP′ is called an FIR-implementation of SP, if in the second condition $\sim_i$ is a sig(SP′)-congruence on T(sig(SP′)) and $<A'/\sim_i>_\Sigma$ is a model of SP. Obviously, every FIR-implementation is also an FRI-implementation.

Fact 7.5 Let SP = <Σ,E> and SP′=<Σ′,E′> be two flat algebraic specifications (cf. def. 4.1). If SP′ is a FRI-implementation of SP, then the specification SP″ is an implementation of SP where

```
spec SP″ ≡
        export Σ from
                enrich SP′ by
                        equalities $=_{SP,s}$: s,s for  s∈sorts(Σ)}
                        axioms { $t =_{SP,s} t'$ | t,t′∈$T(\Sigma)_s$, s∈sorts(Σ), $t \sim_i t'$}
```

and $\sim_i$ denotes the Σ-congruence associated with FRI.

8. Parameterization

Parameterization is the process of encapsulating a piece of software and abstracting from some names (or more generally, from some subexpressions) occurring in it in order to replace them in other contexts by different actual parameters.

Parameterization of structured specifications is done by means of λ-abstraction: a parameterized specification is considered as a mapping taking specifications as arguments and giving a specification as a result. This is formalized by a version of typed λ-calculus, the so-called λπ-calculus which has been introduced by [Feijs 89].

Def. 8.1 A (structured) **parameterized specification** PSP is a λ-expression of the form $\lambda X{:}SP_{par}.SP_{body}[X]$ where SP_{par}, $SP_{body}[X]$ are specification expressions built using the specification operators <.,.>, **derive from . by .**, **translate . with .**, .+., and where X is an identifier not occurring in SP_{par}. Moreover, it is required that $SP_{body}[SP_{par}/X]$ forms a well-defined specification. SP_{par} is called **formal requirement specification** and $SP_{body}[X]$ is called **target specification.**

For the semantics, the "refinement" relation for specifications is considered.

Def. 8.2 For any two specifications SP1 and SP2, SP2 is a **refinement** of SP1, written $SP1 \sqsubseteq SP2$, if $sig(SP1) \subseteq sig(SP2)$ and $Mod(SP2)|sig(SP1) \subseteq Mod(SP1)$.

Fact 8.3 The refinement relation is a partial ordering on the set of specification expressions (w.r.t "=") which is compatible with the specification operators, i.e. for all specifications SP, SP1, SP2,

(1) $SP \sqsubseteq SP$ (reflexivity),

(2) $SP \sqsubseteq SP1$ and $SP1 \sqsubseteq SP2$ implies $SP \sqsubseteq SP2$ (transitivity),
(3) $SP \sqsubseteq SP1$ and $SP1 \sqsubseteq SP$ implies $SP = SP1$ (antisymmetry),
(4) for all signature morphisms $\sigma: \Sigma \rightarrow sig(SP1)$,
$SP \sqsubseteq SP1$ implies **derive from** SP **by** $\sigma|(\sigma^{-1}(sig(SP)) \sqsubseteq$ **derive from** SP1 **by** σ,
(5) for all signature morphisms $\sigma: sig(SP1) \rightarrow \Sigma$,
$SP \sqsubseteq SP1$ implies **translate** SP **with** $\sigma|sig(SP) \sqsubseteq$ **translate** SP1 **with** σ,
(6) for all specifications SP1′, SP2′,
$SP1 \sqsubseteq SP1'$ and $SP2 \sqsubseteq SP2'$ implies $SP1+SP2 \sqsubseteq SP1'+SP2'$.

The semantics of any λ-expression is defined by function application.

Def. 8.3 For any specification SP and parameterized specification $\lambda X{:}SP_{par}.SP_{body}[X]$, the expression

$(\lambda X{:}SP_{par}.SP_{body}[X])(SP)$

is well-defined, if $SP_{par} \sqsubseteq SP$. In this case, it denotes the specification $SP_{body}[SP/X]$.

Example 8.4

(1) The specification SETNAT of 5.4.(1) can be parameterized by replacing the natural numbers by arbitrary data.

pspec SET ≡ $\lambda X{:}ELEM.SET_{ELEM}[X]$,

where

spec ELEM ≡ **sort** Elem **endspec**

and where $SET_{ELEM}[X] =_{def}$ SETNAT[X/NAT0, Elem/Nat].

(2) Similarly, the specification SEQINT can be parameterized by replacing INT by arbitrary data.

pspec SEQ ≡ $\lambda X{:}ELEM.SEQ_{ELEM}[X]$,

where $SEQ_{ELEM}[X] =_{def}$ SEQINT[X/INT, Elem/Int].

(3) Derived operators can be considered as parameterized specifications. E.g. for all extensions Δ, substitution lists α, and signatures Σ, let

pspec $enr_{\Delta} \equiv \lambda X{:}R_1$.**enrich** X **by** Δ,

pspec $expren_{\Sigma,\alpha} \equiv \lambda X{:}R_2$.**export** Σ **with** α **from** X

where $R_1 =_{def} <free(\Delta),\emptyset>$ denotes the specification formed by the symbols occurring in Δ but <u>not</u> declared in Δ and where $R_2 =_{def}$ **translate** $<\Sigma,\emptyset>$ **with** $_{\Sigma}\alpha: \Sigma \rightarrow sig(X)$. Note that Δ, α, Σ are expressions which may depend on X.

The notion of implementation generalizes straightforwardly to parameterized specifications (cf. [Sannella, Wirsing 82], [Wirsing 86]).

Def 8.5 Let $SP1 = \lambda X{:}SP1_{par}.SP1_{body}$ and $SP2 = \lambda X{:}SP2_{par}.SP2_{body}$ be two parameterized specifications.

(1) SP2 is called **(partial parameterized) implementation** of SP1, written SP1~~~> SP2, if $SP1_{par} \sqsubseteq SP2_{par}$ and for all actual parameters SP of SP2 (with $SP2_{par} \sqsubseteq SP$) the application SP2(SP) of SP2 to SP is an implementation of the application SP1(SP) of SP1 to SP, i.e. SP1(SP) ~~~> SP2(SP).

(2) A partial parameterized implementation SP2 of SP1 is called **total**, if every actual parameter of SP1 is also an actual parameter of SP2, i.e. if $SP2_{par} \sqsubseteq SP1_{par}$.

Example 8.6 The parameterized specification

```
pspec SET_by_ORDSEQ ≡
      λX: rename T0 by [Elem/s].
      export sig(SET(X)) with [Set/Seq, =/=_SET(X)] from
      enrich X
      by Δ[Elem/Int, SET(X)/SETNAT, cons_X/{0, succ}, cons_SET(X)/{0, succ, empty, add}]
endspec
```

(where T0 is the specification of total orderings of example 4.4(1) and where Δ denotes the enrichment of SEQINT in example 7.3(3)) is a partial parameterized implementation of SET. It is not total because the formal parameter of SET_by_ORDSEQ requires a total ordering relation whereas the formal parameter of SET is trivial.

In order to prove a horizontal composition theorem for implementations we generalize the notion of refinement to parameterized specifications.

Def 8.7 Let $SP1 = \lambda X{:}SP1_{par}.SP1_{body}$ and $SP2 = \lambda X{:}SP2_{par}.SP2_{body}$. Then SP2 is called **refinement** of SP1, written $SP1 \sqsubseteq SP2$, if

(1) $SP2_{par}$ is a refinement of $SP1_{par}$, i.e. $SP1_{par} \sqsubseteq SP2_{par}$, and

(2) for all actual parameters SP of SP2 (i.e. $SP2_{par} \sqsubseteq SP$) SP2(SP) is a refinement of SP1(SP), i.e. $SP1(SP) \sqsubseteq SP2(SP)$.

For the horizontal composition theorem for refinements we need the following lemma.

Lemma 8.8 Let $SP1 = \lambda X{:}SP1_{par}.SP1_{body}$ be a parameterized specification and let SPA and SPB be actual parameters of SP1 (i.e. $SP1_{par} \sqsubseteq SPA$ and $SP1_{par} \sqsubseteq SPB$). If SPB is a refinement of SPA, then SP1(SPB) is a refinement of SP1(SPA).

Proof by structural induction on the form of $SP1_{body}$ by using the monotonicity of the specification operators w.r.t. $\sqsubseteq$ (see 8.3). □

Theorem 8.9 (Horizontal composition of refinements)
Let $SP1 = \lambda X{:}SP1_{par}.SP1_{body}$ and $SP2 = \lambda X{:}SP2_{par}.SP2_{body}$ be parameterized specifications and

let SPA be an actual parameter of SP1 and SPB be an actual parameter of SP2 (i.e. $SP1_{par} \sqsubseteq SPA$ and $SP2_{par} \sqsubseteq SPB$).
If SP2 is a refinement of SP1 and SPB is a refinement of SPA, then SP2(SPB) is a refinement of SP1(SPA).

Proof Since SPB is a refinement of SPA it is also an actual parameter of SP1. By lemma 8.8 we have $SP1(SPA) \sqsubseteq SP1(SPB)$. Since SP2 is a refinement of SP1, $SP1(SPB) \sqsubseteq SP2(SPB)$ holds. Thus by transitivity of $\sqsubseteq$ (see 8.3(2)) we have $SP1(SPA) \sqsubseteq SP2(SPB)$. □

As a corollary we get the horizontal composition theorem for implementations.

Corollary 8.10 Let SP1, SP2 and SPA, SPB be as in theorem 8.9. If SP2 is an implementation of SP1 and SPB is an implementation of SPA, then SP2(SPB) is an implementation of SP1(SPA).

Application of a parameterized specification $\lambda X{:}SP_{par}.SP_{body}[X]$ to an actual parameter SP as defined above requires in particular that the signature of SP contains the signature of SP_{par}. But this is almost <u>never</u> satisfied as e.g. the examples 8.4, (1) and (2), show: nor $ELEM \sqsubseteq NAT$ neither $ELEM \sqsubseteq INT$ do hold. In order to be able to perform parameter passing for arbitrary actual parameters, it is necessary to rename the requirement specification appropriately.

Def 8.11 Let SP be a specification and $\lambda X{:}SP_{par}.SP_{body}[X]$ be a parameterized specification.

(1) A signature morphism $\rho{:}sig(SP_{par}) \rightarrow sig(SP)$ is called **parameter passing morphism**, if (**translate** SP_{par} **with** $\rho) \sqsubseteq SP$.

(2) Function application takes two arguments, the actual parameter SP and parameter passing morphism ρ, and is defined as follows.

$$(\lambda X{:}SP_{par}.SP_{body}[X])(SP,\rho) =_{def} (\lambda X{:}\rho \bullet SP_{par}.\rho \bullet SP_{body}[X])(SP)$$

where $\rho \bullet SP_{par} =_{def}$ **translate** SP_{par} **with** ρ_X: $sig(SP_{par}) \rightarrow sig(X)$,
$\rho \bullet SP_{body}[X] =_{def}$ **translate** $SP_{body}[X]$ **with** $_{\Sigma[X]}\rho$,
$\Sigma[X] =_{def} sig(SP_{body}[X])$.

Notice that it may be necessary to rename also parts of $SP_{body}[X]$ in order to avoid name clashes with SP. The examples for implementations (in section 7) suggest the following definition [Sannella, Tarlecki 87].

Def. 8.12 For any two specifications SP and SP1 and parameterized specification γ, SP1 is called an **implementation of SP via** γ, written SP $\leadsto_{\gamma}$ SP1, if SP $\leadsto$ $\gamma(SP1,\rho)$ for some appropriate parameter passing morphism ρ.

In many examples of implementations, γ has the form "$expren_{\Sigma,a}$ o enr_{Δ}", e.g. for

BOOL_by_NAT (cf. 7.3.(1)), $\gamma = \text{expren}_{\Sigma B0,\alpha 0} \circ \text{enr}_{\Delta 0}$ where $\alpha 0 =_{def}$ [Bool/Nat, true/0, not/succ, $=_{Bool}/=_{BOOL3,Nat}$] and $\Delta 0 =_{def}$ <false: $\rightarrow$Nat, $=_{BOOL3,Nat}$: Nat,Nat, {false=succ(0), $\forall$**cons** x: Nat. succ(succ(x)) $=_{BOOL3,Nat}$ x}>. All examples of implementations in this paper are of this form. In the following we will call such implementations ERE-implementations (where ERE stands for enrich-rename-export).

Def. 8.13

(1) A specification SPB is called **ERE-implementation** of the specification SPA, written SPA $\sim\sim\sim>_{ERE}$ SPB, if there exists an enrichment Δ, a substitution list α and signature Σ such that SPB is an implementation of SPA via $\text{expren}_{\Sigma,\alpha} \circ \text{enr}_{\Delta}$.

(2) A parameterized specification $SP2 = \lambda X{:}SP2_{par}.SP2_{body}$ is called **(partial parameterized) ERE-implementation** of the parameterized specification $SP1 = \lambda X{:}SP1_{par}.SP1_{body}$, written SP1 $\sim\sim\sim>_{ERE}$ SP2, if $SP1_{par} \subseteq SP2_{par}$ and there exists an enrichment Δ, a substitution list α and a signature Σ such that for each actual parameter SP of SP2, SP2(SP) is an implementation of SP1(SP) via $\text{expren}_{\Sigma[SP/X],\alpha[SP/X]} \circ \text{enr}_{\Delta[SP/X]}$.

Theorem 8.14 (Vertical composition of ERE-implementations)
For any specifications SP1, SP2 and SP3, if SP1 $\sim\sim\sim>_{ERE}$ SP2 and SP2 $\sim\sim\sim>_{ERE}$ SP3, then SP1 $\sim\sim\sim>_{ERE}$ SP3.

Proof Let SP2 be an implementation of SP1 via $\text{expren}_{\Sigma 2,\alpha 2} \circ \text{enr}_{\Delta 2}$ and let SP3 be an implementation of SP2 via $\text{expren}_{\Sigma 3,\alpha 3} \circ \text{enr}_{\Delta 3}$ such that $\text{sig}(\text{enr}_{\Delta 3}(SP3)) \cap \text{sig}(\alpha 3(\Delta 2)) = \emptyset$. Notice, that $\alpha 3$ can always be chosen in such a way that this syntactic condition is satisfied. Then by transitivity of the implementation relation, SP1 $\sim\sim\sim>$ γ (SP3) holds where $\gamma = \text{expren}_{\Sigma 2,\alpha 2} \circ \text{enr}_{\Delta 2} \circ \text{expren}_{\Sigma 3,\alpha 3} \circ \text{enr}_{\Delta 3}$. According to corollary 6.3 we have

$\gamma =$ [by 6.3(3)]

$\text{expren}_{\Sigma 2,\alpha 2} \circ \text{expren}_{\Sigma 3 \cup \text{sig}(\Delta 2),\alpha 3} \circ \text{enr}_{\alpha 3(\Delta 2)} \circ \text{enr}_{\Delta 3} =$ [by 6.3(1)-(2)]

$\text{expren}_{\Sigma 2,\alpha 3 \circ \alpha 2} \circ \text{enr}_{\alpha 3(\Delta 2) \cup \Delta 3}$.

Hence SP3 is a ERE implementation of SP1. □

The full specification language supports the following version of a horizontal composition theorem.

Theorem 8.15 (Horizontal composition for ERE-implementations, I)
Let $SP1 = \lambda X{:}SP1_{par}.SP1_{body}$ and $SP2 = \lambda X{:}SP2_{par}.SP2_{body}$ be parameterized specifications, let SPA be an actual parameter of SP1 and SPB be an actual parameter of SP2.
If SP2 is an ERE-implementation of SP1 and SPB is an ERE-implementation of SPA using the substitution list α and the enrichment Δ, then SP2(**rename** enr_{Δ}(SPB) **by** α) is an ERE-implementation of SP1(SPA).

Proof Let SP1 $\leadsto$ $\text{expren}_{\Sigma2,\alpha2}$ o $\text{enr}_{\Delta2}$(SP2) and SPA $\leadsto$ $\text{expren}_{\Sigma,\alpha}$ o enr_{Δ}(SPB) where α and Δ are given above. Then by monotonicity of SP1, SP1(SPA) $\leadsto$ SP1(SP′) where SP′ $=_{\text{def}}$ $\text{expren}_{\Sigma,\alpha}$ o enr_{Δ}(SPB). Because of SP′$\sqsubseteq$SP″ for SP″ $=_{\text{def}}$ **rename** (enr_{Δ}(SPB)) **by** α, we get by monotonity w.r.t.$\sqsubseteq$

SP1(SP′)$\sqsubseteq$SP1(SP″).

Then

SP1(SP″) $\leadsto$ $\text{expren}_{\Sigma2[SP''],\alpha2[SP'']}$ o $\text{enr}_{\Delta2[SP'']}$ o SP2(SP″)

where Γ[SP″] $=_{\text{def}}$ Γ[SP″/X, sig(SP″)/sig(SP2_{par})] for $\Gamma \in \{\Sigma2,\alpha2,\Delta2\}$. Hence by transitivity of $\sqsubseteq$, SP1(SPA) $\sqsubseteq$ $\text{expren}_{\Sigma2[SP''],\alpha2[SP'']}$ o $\text{enr}_{\Delta2[SP'']}$ o SP2(SP″). Cutting down the export signature of the right-hand side to Σ' $=_{\text{def}}$ sig(SP1(SPA)) we get the expected result:

SP1(SPA) $\leadsto$ $\text{expren}_{\Sigma',\alpha2[SP'']}$ o $\text{enr}_{\Delta2[SP'']}$ o SP2(SP″).

Therefore SP2(SP″) is an ERE-implementation of SP1(SPA). □

It is possible to strengthen this result by merging the renaming α of SPB with the renaming of SP2, if SP2 commutes with α: the parameterized specification SP2 is said to **commute with the substitution list** α if for each actual parameter SP of SP2,

SP2(**rename** SP **by** α) = **rename** (SP2(SP)) **by** α.

For example, if SP2 is built only by using the operators <.,.>, enrich, +, then SP2 commutes with any α. Renamings and exports may also occur in SP2 if they are invariant w.r.t. the signature of the formal parameter and if there are no name clashes of α with symbols declared in the body of SP2, i.e. in sig(SP2)\sig(SP2_{par}).

Corollary 8.16 (Horizontal composition for ERE-implementations, II)
Let SP1, SP2, SPB and SPA as in theorem 8.14.
If SP2 is an ERE-implementation of SP1 (via $\text{expren}_{\Sigma2,\alpha2} \circ \text{enr}_{\Delta2}$), SPB is an ERE-implementation of SPA (via $\text{expren}_{\Sigma,\alpha} \circ \text{enr}_{\Delta}$) and α commutes with SP2, then SP2(enr_{Δ}(SPB)) is an ERE-implementation of SP1(SPA)
(via $\text{expren}_{\Sigma',\alpha2[SP']\text{o}\alpha} \circ \text{enr}_{\Delta2[SP']}$ where Σ' $=_{\text{def}}$ sig(SP1(SPA)), SP′ $=_{\text{def}}$ enr_{Δ}(SPB) and Γ[SP′] $=_{\text{def}}$ Γ[SP′/X, sig(SP′)/sig(SP2_{par})] for $\Gamma \in \{\alpha2,\Delta2\}$).

9. Concluding remarks

In the previous sections a general framework for algebraic specification has been developed including a semantic basis, language constructs for structuring specifications and an appropriate notion of implementation. The key idea was the use of standard predicate symbols and of an ultra-loose semantics based on total algebras.

For other semantic approaches different standard symbols are considered: e.g. for the partial algebra approach (cf. e.g. [Wirsing et al. 83]) a so-called "definedness" predicate symbol D is used which holds in a structure A for a term t, if and only if the interpretation of t in A is defined. For monotonic and continuous specifications partial ordering predicate symbols are used instead of equality symbols (cf. e.g. [Möller 87]). For observation-oriented specifications Hennicker uses a so-called observability predicate "Obs" [Hennicker 89]. The generating predicate symbols of section 2.2 have a simple form. It may also be interesting to consider predicate symbols "$\in$ <S,F,X>" where $x \in$ <S,F,X> holds in a structure A if x is the interpretation of a term $t \in T(\Sigma,X)$ with variables in X. More generally, it may be interesting to consider a binary predicate symbol ":". For a structure A the expression x:W holds, if and only if (the interpretation of) x is an element of (the interpretation of) W where W is an arbitrary set of terms; this leads to an approach to algebraic specifications where sets are "first-class citizens": there exist not only function and predicate symbols for elements but also for sets of elements. For example, there may exist polomorphic sorts (as e.g. in ML), sort-building function symbols (as e.g. in ASL [Wirsing 86]), predicate symbols on sorts (such as subsorting in OBJ [Futatsugi et al. 85]), or predicate symbols relating elements with sets (such as ":" above). Due to the modularity of our approach which allows to choose (relatively) freely the standard symbols there is some hope that it will not be difficult to integrate many of these additional features.

Acknowledgement

Thanks go to Rolf Hennicker for reading carefully drafts of this paper and pointing out many inconsistencies and also to Stefan Gastinger for carefully typing this paper.

References

[Barwise 77] J.K. Barwise (ed.): Handbook of Mathematical Logic. Amsterdam: North Holland, 1977, 1091-1132.

[Bauer, Wössner 82] F.L. Bauer, H. Wössner: Algorithmic Language and Program Development. Berlin: Springer, 1982.

[Bergstra, Tucker 86] J.A. Bergstra, J.V.Tucker: Algebraic specificatios of computable and semicomputable data types. Mathematical Centre, Amsterdam, Dept. of Computer Science Research Report CS-R8619, 1986.

[Bergstra et al. 86] J.A. Bergstra, J. Heering, P. Klint: Module algebra. Math. Centrum Amsterdam, Report CS-R8615.

[Broy et al. 86] M. Broy, B. Möller, P. Pepper, M. Wirsing: Algebraic implementations preserve program correctness. Sci. Comput. Programming 7 ,1986, 35-53

[Burstall, Goguen 80] R.M. Burstall, J.A. Goguen: The semantics of CLEAR, a specification language. Proc. Advanced Course on Abstract Software Specifications, Copenhagen, Lecture Notes in Computer Science 86. Berlin: Springer, 1980, 292-232.

[Ehrig, Kreowski 82] H. Ehrig, H.-J.Kreowski: Parameter passing commutes with implementation of parameterized data types. Proc. 9th Int. Colloquium on Automata, Languages and Programming, Aarhus, Lecture Notes in Computer Science 140. Berlin: Springer, 1982, 197-211.

[Ehrig, Mahr 85] H. Ehrig, B. Mahr: Foundations of Algebraic Specifications I: Equations and Initial Semantics. Berlin: Springer, 1985.

[Ehrig, Weber 86] H. Ehrig, H. Weber: Programming in the large with algebraic module specifications. In: H. Kugler (ed.): Proc. 10th IFIP World Congress, Dublin, 1986.

[Ehrig et al. 82] H. Ehrig, H.-J. Kreowski, B. Mahr and P. Padawitz: Algebraic implementation of abstract data types. Theoret. Comput. Sci. 20 ,1982, 209-263.

[Feijs et al. 89] L.M.G. Feijs: The calculus λπ. To appear in: Algebraic methods: Theory, Tools and Applications. Lecture Notes in Computer Science. Berlin: Springer 1989.

[Futatsugi et al. 85] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, J. Meseguer: Principles of OBJ2. Proc 12th ACM Symp. on Principles of Programming Languages, New Orleans, 1985, 52-66.

[Goguen et al. 78] J.A. Goguen, J.W. Thatcher and E.G. Wagner: An initial algebra approach to the specification correctness, and implemantation of abstract data types, IBM Research Rept. RC-6487, also in: R.T. Yeh, ed., Current Trends in Programming Methodology, Vol. 4: Data Structuring. Englewood Cliffs: Prentice Hall, 1978, 80-149.

[Guttag 75] J.V. Guttag: The specification and application to programming of abstract data types. Ph.D. Thesis, Univ. of Toronto, 1975.

[Guttag et al. 85] J.V. Guttag, J.J. Horning, J.M. Wing: Larch in fife easy pieces. Digital Systems Research Center, Palo Alto, Technical Report 5, 1985.

[Hennicker 89] R. Hennicker: Observational implementation. To appear in Proc. STACS 89, Paderborn, Lecture Notes in Computer Science. Berlin: Springer, 1989.

[Hoare 69] C.A.R. Hoare: An axiomatic basis for computer programming. Comm. ACM 12, 1969, 576-583.

[Majster 77] M.E.Majster: Limits of the "algebraic" specification of abstract data types, ACM-Sigplan Notices 12, October 1977, 37-42.

[Möller 87] B. Möller: Higher-order algebraic specifications. Fakultät für Mathematik und Informatik der TU München, Habilitationsschrift, 1987.

[Mosses 89] P. Mosses: Unified algebras and modules. Computer Science Dept. Aarhus University, Technical Report DAIMIPB-266, also in: Proc. Symp. on Principles of Programming Languages 89. To appear.

[Sannella, Tarlecki 85a] D.T. Sannella and A. Tarlecki: Building specifications in an arbitrary institution, Internat. Symp. on Semantics of Data Types, Sophia-Antipolis, Lecture Notes in Computer Science 173. Berlin: Springer 1985, 337-356

[Sannella, Tarlecki 85b] D.T. Sannella, A. Tarlecki: Program specification and development in Standard ML. Proc. 12th ACM Symposium on Principles of Programming Languages, New Orleans, 1985, 67-77.

[Sannella, Tarlecki 87] D.T. Sannella, A. Tarlecki: Toward formal development of programs from algebraic specifications: implementations revisited. In: H. Ehrig et al. (eds.): TAPSOFT '87, Lecture Notes in Computer Science 249. Berlin: Springer, 1987, 96-100.

[Sannella, Wirsing 82] D.T. Sannella, M.Wirsing: Implementation of parameterised specifications, Rept. CSR-103-82, Dept. of Computer Science, Univ. of Edinburgh; extended abstract in: Proc. 9th Internat. Coll. on Automata, Languages and Programming, Aarhus, Denmark, Lecture Notes in Computer Science 140. Berlin: Springer, 1982, 473-488.

[Sannella, Wirsing 83] D.T. Sannella, M.Wirsing: A kernel language for algebraic specification and implementation, Coll. on Foundations of Computation Theory, Linköping, Sweden; 1983, Lecture Notes in Computer Science 158. Berlin: Springer, 1983, 413-427.

[Wirsing 86] M. Wirsing: Structured algebraic specifications: a kernel language, Theoretical Computer Science 42, 1986, 123-249.

[Wirsing et al. 83] M. Wirsing, P. Pepper, H. Partsch, W. Dosch, M. Broy: On hierarchies of abstract data types. Acta Inform. 20, 1983, 1-33.