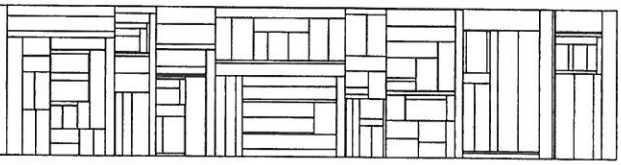
ISSN 0105-8517

Unified Algebras and Modules

Peter D. Mosses

DAIMI PB – 266
October 1988

AARHUS UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
Ny Munkegade 116 – DK 8000 Aarhus C – DENMARK
Telephone: + 45 86 12 71 88 *Telex: 64767 aausci dk*



Abstract

This paper concerns the algebraic specification of abstract data types. It introduces and motivates the recently-developed framework of unified algebras, and provides a practical notation for their modular specification. It also compares unified algebras with the well-known framework of order-sorted algebras, which underlies the OBJ specification language.

©ACM. A reformatted version of this report is to appear in POPL'89: *Proceedings of the Sixteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Austin, Texas, 11–13 January, 1989.*

1 Introduction

This paper proposes a radically new framework for the algebraic specification of abstract data types, called *unified algebras*, together with a simple yet convenient notation for modular specifications.

Unified algebras challenge a dogma that has been accepted since the earliest work on algebraic specifications: that the classification of elements into "sorts" and the elements themselves should be kept separate.

The main features of unified algebras are as follows:

- Classifications of elements into sorts are represented directly as values in the carriers of unified algebras. Operations for sort union and intersection are provided, as well as the empty sort and subsort inclusions.

- Operations on elements extend naturally to subsort-preserving operations on sorts. For example, the successor operation maps the sort of all natural numbers to the (sub)sort of positive natural numbers.

- There is no distinction between an element and the sort classifying only that element.

- Both partial and non-strict operations are allowed. In fact the undefined result is represented by the empty sort. (The conditional if-then-else operation is a good example of a useful non-strict operation.)

- Nondeterminism is allowed. A nondeterministic choice between some elements is not distinguished from the sort consisting of just those elements.

- Sort constructions are ordinary operations. For example, consider an operation that maps an integer i to the sort of all integers up to i; or an operation that maps a natural number n and a sort s to the sort of all lists of length n with components of sort s.

- Sorts may be subject to equations. Thus the sort of integers may be equated with the union of the natural numbers sort and the application of negation to that sort. (Union is idempotent, so zero does not get duplicated.)

- The signatures of unified algebras are very simple: they give just the number of arguments of each operation. They do *not* distinguish constants that denote sorts from those that denote elements; nor do they indicate how the sort of the result of an operation depends on the sorts of its arguments. (Such information may specified by axioms.)

- The axioms used to specify unified algebras are quite general: Horn clauses, involving equality, sort inclusion, and classification of elements into sorts.

- All operations are fully "polymorphic", and may be applied to arbitrary operands without prior "instantiation". However, operations may also be restricted so that they only give defined results on certain sorts of arguments. For example, the if-then-else operation may be restricted so that the result is only defined when the first argument is a truth-value, whereas the second and third arguments are left unrestricted.

- Constraints, analogous to so-called "data constraints", can be used to restrict parts of unified algebras to be freely-generated by other parts. For generic data types (such as lists) their parameters (such as the sort of components) are "loosely-specified" parts; instantiation is merely the specialization of such parts.

Section 2 explains the conceptual basis of unified algebras, and then sketches the foundations. (More details may be found in [21].) However, the emphasis of the present paper is on pragmatics, rather than foundations: the aim is to show that unified algebraic specifications can be just as concise and modular as those in other frameworks.

Section 3 introduces notation for basic specifications, and shows how order-sorted specifications (as in OBJ [6]) can be regarded as specifications of unified algebras.

Section 4 introduces a simple notation for modular specifications. Some pleasant pragmatic features of these modular specifications are:

- Modules may be declared in any order, and may be mutually recursive. Moreover, module declarations may be split up and interleaved, so that information essential to users (analogous to an "interface") may be specified separately from definitional details (which are analogous to an "implementation").

- Modules may be nested. A module may be split into sub-modules without affecting the use of the module. Also, a module may be "opened", so that the notation it specifies is tacitly made available to other modules.

- Basically, each operation symbol has a single interpretation throughout an entire specification: properties specified in separate modules are simply united. "Localization" of operation symbols may be achieved by renaming.

A direct semantics for "canonical" modular specification is given. It is shown how any modular specification can be reduced to canonical form, thus establishing an indirect "transformational semantics" [1] for arbitrary modular specifications.

Section 5 show how constraints are used to specify generic types, with instantiation being just specialization.

Section 6 compares unified algebras and modules to related frameworks —in particular, to order-sorted algebras and OBJ, which largely inspired the development of unified algebras.

Some concluding remarks report on the experience so far with using unified algebras and modules, and indicate where further development of the framework is needed.

The reader is assumed to be familiar with the initial algebra approach to the specification of abstract data types [12, 2, 15, 32, 5].

2 Unified Algebras

To start with, let us recall the basic concepts of abstract data types, and relate them to unified algebras.

2.1 Concepts

A *data type* consists of a set of *elements* (such as numbers or lists) together with a collection of named *operations* between elements—i.e., an algebra. An *abstract* data type is a *class* of algebras that share some properties.

In the so-called "algebraic" approach to specification of abstract data types, a basic specification consists of a *signature* and a set of logical *sentences*. The signature provides names for operations (constants are regarded as operations with no arguments). The satisfaction of the sentences provides properties of the operations. The specified class of algebras consists of all algebras that have (only) the named operations with the given properties. Note that the elements of these algebras may be any entities, abstract or concrete, provided that they are equipped with the proper operations.

When specifying an abstract data type algebraically, it is helpful to identify various *classifications* of elements into *sorts*, and to give for each operation, the relation between the sorts of its arguments and the sort of its result. If the arguments of an operation are restricted to subsorts of the specified sorts, the result may also be restricted to a subsort. In particular, when arguments are restricted to single element sorts, the result sort may be restricted to the result of applying the operation to these elements.

In general, however, few of the possible classifications are useful enough to merit the introduction of special names for them. For instance, consider the abstract data type of integers: apart from the sort of all integers, the sort of natural numbers is certainly useful enough, being closed under several integer operations; but how about the positive integers, the negative integers, the non-positive integers, the even integers, etc., etc.?

In unified algebras, sorts have the same status as elements—in particular, operations may be applied to sorts as well as to elements. It turns out that many classifications of secondary importance can be expressed by applying "elementary" operations to sort constants. For instance, the

sort of positive integers is expressed by the application of the successor operation to the sort of natural numbers; the sort of negative integers is given by applying negation to that sort; and so on. Thus it is not necessary to complicate signatures with constants that name such sorts.

Let us henceforth refer to sorts and elements together as *choices*. (In fact the development of unified algebras started from the observation that there is a close correspondence between sorts and non-deterministic choices. See [19] for more discussion of the treatment of nondeterminism in unified algebras.)

Unified algebras do not necessarily provide *all* possible choices between elements. However, the set of choices provided by a unified algebra always includes the vacuous choice, Hobson's choices[1] of single elements, and all finite choices. The set of choices is always closed under (finitary) union and intersection.

Choices are partially ordered by *inclusion*: if c_1 and c_2 are choices, then '$c_1 \leq c_2$' asserts that c_1 is included in c_2. An important special case of inclusion is *classification*: '$c_1 : c_2$' asserts that c_1 is the Hobson's choice of a single element, included in c_2. Different Hobson's choices are incomparable in the partial order. The vacuous choice, denoted 'nothing', is least in the partial order. The choice between two choices c_1, c_2, denoted '$c_1 \mid c_2$', is their least upper bound; their "agreement", denoted '$c_1 \& c_2$', is their greatest lower bound.

The set of choices between elements forms a distributive lattice with a bottom. Note that the Hobson's choices need not be the so-called "atoms" of the lattice (i.e., "just above" the bottom); but choices between them and the bottom are not much use, as they cannot include any elements. More generally, choices need not be "extensional": two distinct choices may classify the same set of elements.

NB! Choice inclusion must not be confused with computational approximation in Scott domains; in fact lattices here are *not* usually cpos.

As well as a set of choices, a unified algebra has constants that distinguish particular choices, and operations that map choices to choices—preserving choice inclusion. Thus operations are monotonic, but not necessarily continuous.

For example, consider the usual type of natural numbers, with ele-

[1]For the benefit of readers unfamiliar with this idiom: "Hobson's choice: option of taking the one offered or nothing [from T. Hobson, Cambridge carrier (d. 1631) who let out horses on this basis]." [4]

ments 0, 1, 2, ... This type can be represented by a unified algebra whose set of choices includes all finite *and cofinite* choices between these elements, together with the following constants and operations:

- '0', denoting the Hobson's choice of the single element 0;
- 'Natural', denoting the infinite choice between *all* the elements;
- 'successor_', denoting the operation that maps each element to its successor—and maps any choice between elements to the choice between their successors: 'Natural' is mapped to the choice between all non-zero natural numbers, 'nothing' is mapped to 'nothing';
- 'predecessor_', analogous to 'successor_', except that '0' is mapped to 'nothing', and 'Natural' is mapped to 'Natural'.

(Other operations would require further infinite choices, e.g., 'double_' would require choices between infinite sets of even numbers, etc.)

By the way, note that properties of operations do not always extend from elements to multiple choices—nor to the vacuous choice. For example, suppose that a unified algebra representing a data type of natural numbers has binary operations for addition and multiplication. The multiplication of a choice c by (the Hobson's choice of) 2 is not the same as the addition of c to c when c is a multiple choice; and the multiplication of 'nothing' by 0 is 'nothing', rather than 0.

So much for the concepts underlying unified algebras. Let us now consider their formalization.

2.2 Formalities

Before defining unified signatures and algebras, let us specialize the conventional notation for heterogeneous algebras to homogeneous algebras (eradicating sort-indexed sets).

First, let Symbol be the set of *symbols* used to name constants and operations, partitioned into disjoint subsets Symbol_n, $n \geq 0$. Let Variable be a set of *variables*, disjoint from Symbol.

A *homogeneous algebraic signature* is simply a subset Σ of Symbol. We write Σ_n for $\Sigma \cap \mathsf{Symbol}_n$, for $n \geq 0$. A *homogeneous algebraic signature morphism* $\sigma : \Sigma \to \Sigma'$ is a family of maps $\sigma_n : \Sigma_n \to \Sigma'_n$. We write $\sigma(f)$ for $\sigma_n(f)$, where $f \in \Sigma_n$.

A homogeneous Σ-*algebra* A consists of a set $|A|$ (of *choices*) and for each $f \in \Sigma_n$ a function $f_A : |A|^n \to |A|$ (called a *constant* when $n = 0$, otherwise an *operation*). A Σ-*homomorphism* $h : A \to B$ is a function from $|A|$ to $|B|$ such that for any $f \in \Sigma_n$ and $a_1, \cdots, a_n \in |A|$

$$h(f_A(a_1, \cdots, a_n)) = f_B(h(a_1), \cdots, h(a_n)).$$

So much for homogeneous algebras. Now for unified algebras.

A *unified signature* is a homogeneous algebraic signature that includes the constant symbol 'nothing' and the binary operation symbols '_ | _' and '_&_'. (Unified signature morphisms are homogeneous signature morphisms that preserve the given symbols.) We write **UniSign** for the set of unified signatures. Henceforth, let Σ always be a unified signature.

A Σ-*unified sentence* is a universal Horn clause with variables from Variable, operation symbols from Σ, and binary predicate symbols '$=$', '$\leq$', and ':'. We write **UniSen**(Σ) for the set of Σ-unified sentences.

A Σ-*unified algebra* A is a homogeneous Σ-algebra A such that:

- $|A|$ is a *distributive lattice* with $\_ \mid \__A$ as join, $\_\&\__A$ as meet, and nothing_A as bottom. Let $\leq_A$ denote the partial order of the lattice.
- There is a distinguished subset of incomparable values, $E_A \subseteq |A|$ (the *elements* of A). Note that E_A need not be the "atoms" of the lattice.
- For each $f \in \Sigma$, the function f_A is monotone with respect to $\leq_A$.

A Σ-*unified homomorphism* is a Σ-homomorphism that respects the partial order and maps elements to elements. We write **UniAlg**(Σ) for the class of Σ-unified algebras.

The binary predicate symbols '$=$', '$\leq$', and ':' are interpreted as follows in a unified algebra A:

- $x = y$ holds iff x is identical to y;
- $x \leq y$ holds iff $x \leq_A y$;
- $x : y$ holds iff $x \in E_A$ and $x \leq_A y$.

The *institution* UNI of unified algebras is defined in the usual way, in terms of the obvious categories of unified signatures, unified algebras, and the standard notion of satisfaction for universal Horn clauses.

By establishing the institution of unified algebras, not only do we identify all the relevant basic components of our framework, but also we make available the full power of Sannella and Tarlecki's institution-independent specification notation [26] (which doesn't seem to have a name—let's refer to it as 'S&T' here).

However, S&T is not intended as a practical specification language: it is a powerful kernel upon which practical specification languages may be built. For one thing, it does not provide notation for *naming* modules of specifications.

The following sections introduce a rather simple—yet quite practical—specification language. It would be possible to define the semantics of this language by reducing it to S&T; but that would be somewhat hard on readers who are not familiar with the details of S&T. Instead we give a direct definition of the semantics of "canonical" specifications, and show how arbitrary specifications can be reduced to canonical ones; this reduction provides a "transformational" semantics for our specifications. (In a more thorough treatment, a denotational semantics for the full specification language would be defined, and it would be proved that the given reduction preserves denotations.)

The various constructs of the specification language are introduced gradually, "bottom-up". First come *basic specifications*, which are essentially monolithic specifications of signatures and sentences. Then come *modular specifications*, where a specification is split into named parts, allowing the dependence of these parts upon each other to be made explicit. Finally come *constraints*, a special kind of sentences used to specify "standard" models and "generic" data types.

We don't bother to give an unambiguous concrete syntax for our specification language. Instead, we use ambiguous grammars to define its *abstract syntax*. The grammars are written in a minor variant of BNF: '$\geq$' stands for "produces", ' | ' stands for "alternatively", and terminal symbols are enclosed in quotation marks.

Each non-terminal of a grammar generates a set of strings (of terminal symbols); the derivation trees for these strings—equipped with the tree construction operations—are (essentially) the desired abstract syntactic entities. For writing examples of specifications, we use parentheses and indentation to indicate which abstract syntactic entities are intended, when this is not clear from the context.

3 Basic Specifications

In this section we first define the syntax and semantics of *canonical* basic specifications, which correspond directly to unified signatures and sentences. Such specifications are adequate in theory, but somewhat tedious to use in practice. Therefore we extend the syntax with some convenient abbreviations, which allow us to write specifications that resemble the order-sorted signatures and sentences used in OBJ. Finally, we show how the grammars that we use to define the syntax of basic specifications can themselves be regarded as basic specifications.

3.1 Canonical Basic Specifications

The abstract syntax of canonical basic specifications is defined by the grammar given below.

p : Positive

$\Longrightarrow$

Basic	$\geq$	"constant" Symbol_0 |
		"operation" Symbol_p |
		Clause | Basic Basic ;
Clause	$\geq$	Formula | Formula "$\Longrightarrow$" Clause ;
Formula	$\geq$	Term Relator Term ;
Relator	$\geq$	"$=$" | "$\leq$" | ":" ;
Term	$\geq$	Variable | Symbol_0 |
		Symbol_p Terms_p ;
Terms_1	$\geq$	Term ;
Terms_{p+1}	$\geq$	Term "," Terms_p

The grammar does not define the micro-syntax of symbols ('Symbol_n', $n \geq 0$) and variables ('Variable'). For symbols, let us use strings of characters in this sans serif font, with the number of occurrences of the place-holder character '_' determining the index (i.e., rank) of the symbol. For variables, let us use strings of letters in *this italic font*, optionally distinguished by numerical subscripts and/or primes.

Notice that the grammar is not quite context-free: the indices on the nonterminal symbols 'Symbol' and 'Terms' ensure that operation symbols are only applied to the number of arguments indicated by their indices. Each 'Symbol_n' (for $n \geq 0$) and 'Terms_p' (for $p \geq 1$) may be regarded as a distinct nonterminal symbol, if desired.

A simple example of a canonical basic specification is given below.

constant Truth-Value

constant true

constant false

true : Truth-Value

false : Truth-Value

Truth-Value = true | false

operation if_then_else_

$T \leq$ Truth-Value $\Longrightarrow$

(if T then X else Y) $\leq (X \mid Y)$

if true then X else $Y = X$

if false then X else $Y = Y$

if nothing then X else $Y =$ nothing

if $(T \mid U)$ then X else $Y =$

(if T then X else Y) | (if U then X else Y)

T & Truth-Value = nothing $\Longrightarrow$

(if T then X else Y) = nothing

There is no need to disambiguate the grouping of the basic specifications, as it is semantically irrelevant (in fact, so is the order). We exploit "mix-fix" notation (much as in OBJ) to write the application of an operation symbol '$S_0\_ \ldots \_S_n$' to terms '$T_1, \ldots, T_n$' as '$S_0 T_1 \ldots T_n S_n$'; e.g., we write 'if_then_else_(t,X,Y)' as 'if t then X else Y'.

The effect of specifying '$t : t$' is to insist that t is the Hobson's choice of a single element. More generally, a formula '$z : U$' insists that U includes some element, which (in a non-trivial specification) prevents U from being 'nothing'. (In general, let us follow the convention of writing constants and variables that necessarily stand for single elements in lower

case.)

Caveat: the examples given in this paper are intended mainly to illustrate the *form* of specifications; the choice of which operations and properties to specify is not always that which might be best in a practical specification. Moreover, unified algebraic specifications are no less prone to mistakes than many-sorted or order-sorted ones, and there has not been time to prove that the examples given here actually specify the intended classes of unified algebras.

Now let us define the semantics of canonical basic specifications. First of all, a basic specification is said to be *complete* when all the (constant and operation) symbols occurring in terms—except for the reserved symbols 'nothing', '_ | _', and '_&_'—are *declared* by 'constant S' or 'operation S'. We do not care to give a semantics for incomplete specifications (although it could be done).

The semantics of a complete basic specification B consists of two components: $\mathsf{Sig}[\![B]\!]$, the unified signature specified by B; and $\mathsf{Alg}[\![B]\!]$, the class of unified algebras specified by B. We define:

$$\begin{aligned}\mathsf{Sig}[\![B]\!] &= \{S \in \mathsf{Symbol} \mid S \text{ occurs in } B\} \cup \\ &\quad \{\text{'nothing'}, \text{'\_ | \_'}, \text{'\_\&\_'}\} \\ \mathsf{Alg}[\![B]\!] &= \{A \in \mathbf{UniAlg}(\mathsf{Sig}[\![B]\!]) \mid \\ &\quad A \text{ satisfies all the clauses in } B\}.\end{aligned}$$

3.2 Abbreviations

As may be seen from the specification of truth-values above, canonical basic specifications are a bit tedious. Let us introduce some abbreviations.

Actually, the first abbreviations we introduce would not shorten our specification of truth-values, but they are often convenient. The syntax is as follows (extending the previously given grammar):

Clause	$\geq$	Clause ";" Clause ;
Formula	$\geq$	Formula ";" Formula ;
Relator	$\geq$	"$\geq$" | ":-"

The symbol ";" stands for conjunction in clauses and formulae. The relators "$\geq$" and ":-" stand for the reversals of the relations '$\leq$' and ':', respectively. It is straightforward to reduce any clause using these

constructs to a combination of canonical (Horn) clauses; we omit the details.

Now consider the following extensions, which enable us to write basic specifications resembling those in OBJ—and more!

n : Natural ; p : Positive

$\Longrightarrow$

Basic	$\geq$	"constant" Symbol_0 Relator Term |
		"operation" Symbol_p ":" Functionality_p ;
Clause	$\geq$	Symbol_p ":" Functionality_p ;
Terms_2	$\geq$	Term "2" ;
Functionality_p	$\geq$	Terms_p "$\rightarrow$" Term | Terms_p "$\rightsquigarrow$" Term |
		Terms_p "$\Rightarrow$" Term | Attribute_p |
		Functionality_p Functionality_p ;
Attribute_2	$\geq$	"associative" | "commutative" |
		"idempotent" | "unit" Term ;
Attribute_p	$\geq$	"strict" | "defined" | "elementary"

Using the above constructs, we may abbreviate the specification of truth-values as follows:

constant Truth-Value = true | false

constant true : Truth-Value

constant false : Truth-Value

operation if_then_else_ : Truth-Value, X, $Y \Rightarrow (X \mid Y)$
nothing, X, $Y \rightsquigarrow$ nothing
defined elementary

if true then X else $Y = X$

if false then X else $Y = Y$

T & Truth-Value = nothing $\Longrightarrow$

(if T then X else Y) = nothing

Consider also the following abbreviated specification of natural numbers:

```
constant    Natural = 0 | successor Natural
constant    0 : Natural
operation   successor_ : Natural → Natural
constant    Positive = successor Natural
operation   natural predecessor_ : Natural ⇝ Natural
                                   Positive → Natural
                                   0 ⇝ nothing
N ≤ Natural ⟹
    natural predecessor(successor N) = N
operation   sum(_,_) : Natural² → Natural
                       Positive, Natural → Positive
                       associative commutative unit(0)
operation   product(_,_) : Natural² → Natural
                           Positive² → Positive
                           0, Natural → 0
                           associative commutative
                           unit(successor 0)
| m : Natural ; n : Natural
⟹
| sum(m, successor n) = successor sum(m,n) ;
| product(m, successor n) = sum(m, product(m,n))
```

Now, such specifications look quite nice—to the author, at least—but what is their semantics? Let us see how to reduce them to canonical basic specifications.

The construct 'constant S R T' merely abbreviates the combination of the declaration 'constant S' and the clause 'S R T'. Likewise, 'operation S : F' abbreviates the combination of 'operation S' and the clause abbreviation 'S : F', where F is a "functionality". Thus what appear to be order-sorted signature declarations are really abbreviations for combinations of (unsorted) unified signature declarations and clauses.

There are three main forms of functionality, concerned with so-called "total", "partial", and "general" operations. Total and partial functionalities may be explained in terms of general functionalities and "attributes", which we consider first.

The functionality '$S : T_1,\ldots,T_p \Rightarrow T$' abbreviates the clause (actually, formula) '$S(T_1,\ldots,T_p) \leq T$'. The monotonicity of all operations gives as a consequence that applying the operation S to any choices (or elements) included in the T_i always gives a result included in T.

Any attributes specified along with such a general functionality enhance it as follows (assuming all arguments are included in the T_i):

- 'strict' asserts that when any argument is 'nothing', the result is 'nothing';
- 'defined' asserts that when the result is 'nothing', at least one argument must be 'nothing';
- 'elementary' asserts that when all the arguments are elements, the result is either an element or 'nothing', and, moreover, that the operation is "linear", preserving '_ | _' and '_&_' in each argument separately;
- 'associative', 'commutative', 'idempotent', and 'unit T'' assert the obvious properties for binary operations. (By the way, 'T^2' abbreviates 'T,T'.)

Now it is easy to explain the "total" and "partial" functionalities:

- '$S : T_1,\ldots,T_p \rightarrow T$' abbreviates
  '$S : T_1,\ldots,T_p \Rightarrow T$ strict defined elementary' (the combination of 'defined' and 'elementary' implies that elements get mapped to elements, hence choices that include elements get mapped to choices that include elements);
- '$S : T_1,\ldots,T_p \rightsquigarrow T$' abbreviates
  '$S : T_1,\ldots,T_p \Rightarrow T$ strict elementary' (so elements may get mapped to 'nothing').

In practice, it is convenient to extend almost all operations from elements to choices by using "total" or "partial" functionalities. The "general" functionalities are needed only for non-strict operations (like 'if_then_else_') and for operations that are non-linear (like a sort constructor mapping sorts of components to sorts of lists).

As in order-sorted algebras, an operation may have more than one functionality: the clause '$S : F_1\ F_2$' abbreviates the conjunction '$S : F_1'$;

$S : F_2'$', where F_1' and F_2' each contain all the attributes of F_1 and F_2, and together contain all their total, partial, and general functionalities.

It is claimed that any clause of the form '$S : F$' can be reduced to a conjunction of clauses not involving functionalities, corresponding to the above informal descriptions. A formal specification of this reduction would define the semantics of all basic specifications; here the details are left to the reader's imagination.

3.3 Unified Abstract Syntax

The grammars used above to define the abstract syntax of basic specifications look a lot like basic specifications themselves. Let us see how such a grammar can be regarded as a formal abbreviation for a complete basic specification whose semantics (i.e., a class of unified algebras) corresponds to the intended abstract syntax.

First, consider the nonterminal symbols of the grammar. The unindexed nonterminal symbols, such as 'Basic', may be regarded as constants that stand for sorts of abstract syntactic entities. Indexed nonterminal symbols, such as 'Terms$_p$', may be regarded as operations from index *elements* to syntactic *sorts* (which would not be possible with conventional algebras). By the way, such operations extend naturally to index *sorts*, so we may express the union of all the 'Terms$_p$' by 'Terms $_{\mathsf{Positive}}$'.

Next, consider the alternatives on the right-hand-sides of the productions. We have agreed that nonterminal symbols stand for sorts; so each alternative must be the application of an operation to sorts (or just a constant, if there are only terminal symbols in the alternative). The *operation symbol* may be obtained by replacing all the sort arguments by place-holders. (Notice that the device of enclosing terminal symbols in quotation marks helps to prevent confusion between the implicit syntactic operation symbols and the operation symbols of other data types.) The sorts used in the alternative, together with the sort on the left-hand-side of the production, determine an appropriate (total) *functionality* for the operation.

However, "chain productions", such as 'Basic $\geq$ Clause', would involve an operation named by the invisible operation symbol '_'. It is preferable to avoid introducing this symbol, and to regard chain productions as specifying no more than the given sort inclusion.

The whole right-hand-side of a production is now a *choice* between

sorts corresponding to the alternatives. As choice is sort union, the production specifies that each of the alternatives is included in the sort corresponding to the nonterminal on the left-hand-side.

An example may help. Consider the following grammar:

Clause	$\geq$	Formula | Formula "$\Longrightarrow$" Clause ;
Formula	$\geq$	Term Relator Term ;
Relator	$\geq$	"$=$" | "$\leq$" | ":" ;
Term	$\geq$	"nothing"

The corresponding constants and operations are specified as follows:

constant	Clause $\geq$ Formula
constant	Formula
constant	Relator
constant	Term
operation	_ "$\Longrightarrow$" _ : Formula, Clause $\rightarrow$ Clause
operation	_ _ _ : Term, Relator, Term $\rightarrow$ Formula
constant	"$=$" : Relator
constant	"$\leq$" : Relator
constant	":" : Relator
constant	"nothing" : Term

Combining this basic specification with the original grammar, we get a complete specification whose semantics may be regarded as an abstract syntax. The constraints introduced in Section 5 can be used to restrict the class of unified algebras to those whose only elements are those implied by the above specification (even leaving some parts of the syntax unconstrained, for later specialization).

4 Modular Specifications

Here, a *module* comprises an *identification*, together with a *body*, which is a complete specification. A *modular specification* is a basic specification that is divided into such modules. If one forgets the module identifications, the semantics of a modular specification is just the same as that of the combination of its module bodies.

The modularization of specifications has several pragmatic benefits. First, it exhibits sub-specifications that have an independent meaning, which usually improves *comprehensibility*. Second, a sub-specification that is included in another may be replaced by a reference to the identification of the corresponding module; this allows *re-use* of sub-specifications within a specification, which usually facilitates *making changes*, and which also exhibits the *dependency* relation between modules. Last, it permits the *re-use* of parts of one specification in another specification, which would be essential for a specification "library" consisting of many independent parts specifying standard abstract data types.

We start by introducing *canonical* modular specifications, where modules do not refer to each other. Then we allow *recursive* specifications, with the possibility of mutual reference between modules. Next, we let modules be textually and logically *nested*. Finally, we introduce notation for *translating* and *localizing* modules. Note that we do *not* need to consider *parameterized* modules: generic data types are specified using constraints, as described in the next section.

4.1 Canonical Modular Specifications

The abstract syntax of canonical modular specifications is defined by the following grammar, which extends the grammar of basic specifications:

Modules $\geq$ Identification "." Basic | Modules Modules

The micro-syntax of identifications is not specified; in examples, we use words in **this bold font**.

The grouping and order of modules is irrelevant. For a modular specification to be called canonical, the identifications of all the modules must be distinct, and the bodies of all the modules must be complete.

For example, consider a specification with modules corresponding to truth-values and natural numbers:

Truth Values.

| ...

Numbers. Naturals.

| ...

(where the bodies of the modules have been elided).

The semantics of a canonical modular specification M is an *environment*, mapping module identifications to the semantics of the corresponding module bodies. We define

$$\mathsf{Env}[\![M]\!] = \{I \mapsto (\mathsf{Sig}[\![B]\!], \mathsf{Alg}[\![B]\!]) \mid \text{`}I.B\text{'} \text{ occurs in } M\}.$$

We may also extend $\mathsf{Sig}[\![\_]\!]$ and $\mathsf{Alg}[\![\_]\!]$ from basic specifications to modules:

$$\mathsf{Sig}[\![B]\!] = \cup\{\mathsf{Sig}[\![B]\!] \mid B \text{ occurs in } M\}$$

$$\begin{aligned}\mathsf{Alg}[\![B]\!] = \cup\{&A' \in \mathbf{UniAlg}(\mathsf{Sig}[\![M]\!]) \mid \\ &A' \dagger (\mathsf{Sig}[\![B]\!] \hookrightarrow \mathsf{Sig}[\![M]\!]) \in \mathsf{Alg}[\![B]\!] \\ &\text{for all } B \text{ that occur in } M\}.\end{aligned}$$

where for any $\Sigma \subseteq \Sigma'$ and $A' \in \mathbf{UniAlg}(\Sigma')$, the Σ-algebra $A' \dagger (\Sigma \hookrightarrow \Sigma')$ is obtained from A' by forgetting the operations of $\Sigma' \setminus \Sigma$, but keeping the same set of choices.

Notice that $\mathsf{Sig}[\![M]\!]$ and $\mathsf{Alg}[\![M]\!]$ could be defined in terms of $\mathsf{Env}[\![M]\!]$; but the given definitions make it obvious that they do not depend at all on the identifications of the modules, only on the bodies.

4.2 Recursive Modules

Obviously, canonical modular specifications would be tedious to use directly: notation that is used in several different modules has to be specified afresh in each of them. So let us allow module bodies to specify the inclusion of the bodies of other modules by referring to the corresponding identifications. The syntax for such references is:

Basic $\geq$ "use" Identification

It is not necessary to put any restriction on the usage of "use". In particular, mutual reference (i.e., recursion) is allowed. Duplicate references in a body may always be eliminated; likewise, any self-reference may be

removed: all the notation specified by such a reference is already available!

For a simple example, consider the following modular specification:

Numbers. Naturals.

| ...

Numbers. Integers.

| use **Numbers. Naturals**

| ...

The order of the modules is irrelevant, as with canonical modular specifications.

The semantics of specifications involving "use" is given by defining their reduction to canonical modular specifications. The following algorithm exploits the fact that basic specifications are essentially just sets of operation symbols and clauses; basic specifications with references to module identifications may therefore be regarded as monotonic functions from basic specifications to basic specifications.

Let M be a (recursive) modular specification. Let M_0 be obtained from M by replacing each module body by the vacuous trivial specification[2]. For $n \geq 0$, let M_{n+1} be obtained from M by replacing each 'use I' by whatever I identifies in M_n, then removing any duplicate parts of the resulting bodies. Clearly, the M_n form a non-decreasing chain. But the set of symbols and clauses in each module body is bounded by the set of all the symbols and clauses in M. Thus (as there are only a finite number of modules) the M_n must stabilize at some finite value of n; let M' denote the resulting modular specification.

We now regard M as *complete* if the module bodies in M' are complete (basic specifications). Thus for any complete recursive specification M, M' gives its reduction to a canonical modular specification.

A direct denotational semantics for recursive modular specifications would require making environments into a cpo and using least fixed points of continuous functions.

Before we add more syntax to our modular specification language, let us relax our requirements concerning the uniqueness of module identifications and the completeness of module bodies.

[2]which we may write as 'constant nothing'.

The idea is to allow the textual separation of "interfaces" from "details". Here, an interface does not "hide" anything; it merely draws attention to some particular operation symbols and (perhaps) properties. In practice, this rather trivial form of interface seems to be quite useful.

A specification with separated modules is reduced by combining the bodies of modules that have the same identification. This defines the semantics of those separated modular specifications that reduce to canonical modular specifications.

For an example consider

Numbers. Naturals.

| ... *the interface*

Numbers. Integers.

| use **Numbers. Naturals**

| ... *the interface*

... *some other modules*

Numbers. Naturals.

| ... *the details*

Numbers. Integers.

| 'use **Numbers. Naturals**' *need not be repeated*

| ... *the details*

4.3 Nested Modules

Let us next allow modules to be grouped together in *nests*, so that the identification of the nest may be used to refer to all the modules in the nest. To start with, we enhance the syntax of module *identifications*, to allow what we call "logical nesting"; afterwards, we permit module *bodies* to be bodies, to allow the "textual nesting" of modules.

The syntax for identifications reflects a convention that has been used in the examples above: identifications are essentially sequences of basic identifications.

Identification $\geq$ Identification "." Identification

Now we may regard 'use I' as an abbreviation for the combination of 'use I. I'' for every (relevant) identification I'.

For example, given the above examples, we may write just 'use **Numbers**' instead of

> use **Numbers. Naturals** ;
>
> use **Numbers. Integers**.

Moreover, if we forbid identifications where the same sub-identification occurs more than once, we may unambiguously abbreviate references by omitting a common prefix of the "source" and "target" identification of the reference. Thus 'use I' occurring in a module body with identification I' abbreviates 'use I''. I' for some unique prefix I'' of I'. (Such abbreviations are context-dependent, so they must be eliminated before module bodies are substituted for references.)

For example, the 'use **Numbers. Naturals**' in the module '**Numbers. Integers**' may be abbreviated to 'use **Naturals**'.

Now for textual nesting, which can be useful for emphasizing the "logical" nesting implied by the structure of identifications. The syntax is just

Modules $\geq$ Identification "." Modules

The semantics of 'I. M' is very simple: it is the same environment as that specified by M, except that all the identifications are prefixed by I. Obviously, such constructs can be eliminated syntactically, by distributing 'I.' in M.

We may now exhibit the nesting structure of '**Numbers**' by

> **Numbers.**
>
> > **Naturals.**
> >
> > > ...
> >
> > **Integers.**
> >
> > > use **Naturals**
> > >
> > > ...

The analogy between this notation for nested modular specifications and hierarchical file systems is rather obvious.

Actually, with the above syntax for modules, it is not always possible to convert canonical modular specifications into "fully-nested" modular

specifications with unique identifications at each nesting level. The problem arises when the identification of one basic specification is a proper prefix of that of another basic specification: the body identified by the shorter identification would be a mixture of a basic specification and an identified module, which is not allowed by our syntax so far.

It is a simple matter to extend the syntax to remove this problem:

Modules $\geq$ Basic

but the semantics requires careful consideration. The question is whether the notation declared in a basic specification at some level of nesting is made available to the identified sub-modules of that level, or not. By analogy with block structure in programming languages, we may expect that it should be. The semantics may then be defined by a reduction that distributes basic specifications at outer levels into all identified sub-modules.

But this weakens the modularization discipline somewhat: it is no longer the case that the notation used (but not declared) in a module comes entirely from explicitly-referenced modules: it may come from enclosing modules as well. In particular, notice that our modules may now consist of sequences of basic specifications and identified modules—and that 'use I' is a basic specification. So we may specify

Truth Values.

| ...

use **Truth Values**

As usual, 'use **Truth Values**' references the corresponding basic specification. Thus the notation for truth-values is made available to all the other modules in the specification—just as if it were "built in". The benefit of allowing this specification seems to outweigh the weakening of the modularization.

4.4 Translation and Localization

Sometimes it is convenient to specify several different abstract data types on the basis of a common part—for instance, flat lists and nested lists on the basis of general lists. But although such related data types may sensibly use the same symbols for "polymorphic" operations (such as

'cons(_,_)') and for constants that have the same interpretation (such as 'nil'), it would be inconsistent for them to use the same notation (such as 'List') for the classifications of their respective elements into sorts.

A *translation* allows us to make a copy of a specification with some changes to the symbols. The syntax is as follows:

n : Natural		
$\Longrightarrow$		
Basic	$\geq$	Basic Translation ;
Translation	$\geq$	Symbol_n ":=" Symbol_n |
		Translation ";" Translation

Let us restrict translations to '$S_1{:=}S_1'$; ...; $S_n{:=}S_n'$' where the S_i are all different—and do not include 'nothing', '_ | _', or '_&_'. The order and grouping in translations is irrelevant.

The semantics of 'B T', where B is a canonical basic specification and T is a translation '$S_1{:=}S_1'$; ...; $S_n{:=}S_n'$', is the same as that of the canonical basic specification obtained by replacing all occurrences of the S_i in B by the corresponding S_i' (leaving other symbols alone).

Notice that in practice, a translation is usually applied to a reference 'use I'. The semantics of such basic translations is determined by the given reduction of recursive modular specifications to canonical modular specifications.

The final syntactic construct for modules introduced here provides a simple form of "hiding", called *localization*:

Basic	$\geq$	"local" Basic ;
Symbol_n	$\geq$	Identification "." Symbol_n

The idea is to allow the introduction of auxiliary notation in a module, but without the danger that its specified properties might "conflict" with properties specified in other modules.

The semantics of 'local B' is given by reducing it to a translation 'B T'. The translation T translates every operation symbol in B to the same symbol prefixed by the *identification* of the module directly enclosing the localization. It is possible to specify properties of the translated operations in another module—but only by including an identification in the operation symbol. In practice, identifications would never be written

explicitly in operation symbols, so there is no conflict between the local operations of different modules.

Note that the reduction of localization to translation depends on (the identification of) the context, so it must be made before substituting the enclosing module for references to it—and before distributing basic specifications into identified modules.

5 Constraints

So far, the sentences allowed in basic specifications are (essentially) restricted to first-order universal Horn clauses involving predicates for equality, inclusion, and classification. It is well-known that Horn clauses are the most general sentences that can be used if one wants to exploit the so-called "initial algebra approach" to specification of abstract data types, where algebras with "junk" and "confusion" are eliminated by taking only the initial algebra of the specified class, as in [12] (see also [15]), or more generally by using "data constraints" [3, 9, 10] or "initial constraints" [24, 25]. (An alternative to the initial algebra approach is to allow first-order sentences that express inequality, to use them to specify away all possibility of "confusion", and then to impose "generating" or "reachability" constraints, see [26].)

The main idea of a data constraint on a specified class of algebras is that it restricts the class to those algebras where a certain "part" is "freely-generated" by another part. These parts may be identified by sub-specifications, where the specification of the generating part is a sub-specification of that of the generated part. (For full generality, a translation of the sub-specifications is allowed.) When freely-generated algebras determined by specification inclusions (technically, "theory morphisms") always exist, data constraints can be treated as sentences.

Usually, data constraints *cannot* be satisfied in homogeneous algebras: the class of algebras satisfying a homogeneous data constraint is empty. The problem is that the so-called "forgetful functor" $\_\dagger\sigma$ determined by a homogeneous signature morphism σ doesn't forget any *values* at all—only operations! (With heterogeneous algebras, forgetful functors may forget whole sorts of values.)

However, it turns out that the classification relation of unified algebras can be exploited to define a "more forgetful functor", which forgets values unless they are (or are generated by) *elements* included in a *denotable* value—which is quite analogous to the special way sorts are treated in heterogeneous algebras. This more forgetful functor can be used to define so-called "bounded data constraints" for unified algebras; the details are sketched in [21].

Our syntax for bounded data constraints assumes that the sub-specifications involved are always identified as modules:

Basic $\geq$ "constrain" Identification |

"constrain" Identification "over" Identification

Let us consider some examples. First,

Truth Values.

> **Base.**
>
> > constant Truth-Value = true | false
> >
> > constant true : Truth-Value
> >
> > constant false : Truth-Value
>
> constrain **Base**
>
> **Rest.**
>
> > use **Base**
> >
> > operation if_then_else_ : Truth-Value, X, $Y \Rightarrow (X \mid Y)$
> >
> > ...

The constraint restricts the specified class of algebras to those where the **Base**-part has no "junk" or "confusion". Here, the **Base**-part of an arbitrary Sig[[**Truth Values**]]-unified algebra A consists of those *elements* that are included in the choice Truth-Value$_A$, together with the choices denoted by terms built from the **Base** signature. Thus **Rest** must not contradict this constraint by adding further *elements* classified by 'Truth-Value' (although it may add new *choices* included in 'Truth-Value'), nor by causing **Base**-denotable values to be identified. By the way, the operation 'if_then_else_' does not generate new elements of any sort, so it may be specified in the **Rest** of **Truth Values**.

Finally, consider the specification of generic lists below. The constraint ensures that, whatever the elements of the sort 'Data' might be, the elements of sort 'List' are all finite lists of them. Without making the constraint relative to '**Data**', we would only get the empty list, since there are no elements specified to be included in 'Data'.

Lists.

| **Data.**

| | constant Data

| **Base.**

| | use **Data**

| | constant List = nil | cons(Data,List)

| | operation _(of _) : List, Data ⇒ List

| | constant nil : List(of nothing)

| | operation cons(_,_) : Data, List → List

| | $D \leq$ Data $\Longrightarrow$ nil(of D) = nil

| | | d : Data ; l : List ; $D \leq$ Data

| | $\Longrightarrow$

| | | cons(d,l)(of D) = cons(d&D, l(of D))

| constrain **Base** over **Data**

| **Rest.**

| | use **Base**

| | operation head_ : List ⇝ Data

| | operation tail_ : List ⇝ List

| | head nil = nothing

| | tail nil = nothing

| | | d : Data ; l : List

| | $\Longrightarrow$

| | | head cons(d,l) = d ;

| | | tail cons(d,l) = l

Note that we may (independently) specialize the module '**Data**' to include various elements, such as numbers; we may even constrain it to preclude further specialization. For instance:

Lists of Numbers.

> use **Lists**
>
> use **Numbers**
>
> Natural $\leq$ Data
>
> constrain **Lists of Numbers**

(It might be as well to translate the constants 'List' and 'Data', if other instantiations are contemplated.)

How about *nested* lists? Well, it is tempting just to add 'List $\leq$ Data' to the above instantiation. Unfortunately, this conflicts with the constraint in '**Lists**': the lists would no longer be *freely* generated by the elements of data. Instead, we should specify an unconstrained module, say '**General Lists**', much as '**Lists**', only we weaken the equation for 'List' to 'List $\geq$ nil | cons(Data,List)'; then both flat lists and nested lists can be obtained by specializing and constraining general lists. For example,

Nested Lists. Base.

> use **General Lists** (List := Nested-List)
>
> cons(_,_) : Nested-List, Nested-List $\rightarrow$ Nested-List
>
> constrain **Base** over **Lists. Data**

(Some minor extensions are needed to the '**Rest**' to define 'head_' and 'tail_' on nested lists.)

6 Related Work

Our notation for the modular specification of unified algebras may be compared with the OBJ specification language, which is based on the framework of order-sorted algebras. (The reader is now assumed to be familiar with order-sorted algebras [7, 11, 31] and OBJ [6, 8, 13].) There are substantial differences between the approaches, both at the technical and at the pragmatic level. First, some comments on order-sorted algebras:

- The signatures of order-sorted algebras are complex objects, giving the sorts of operation arguments and results separately for each "version" of polymorphic operations, subject to some constraints that guarantee that terms have least sorts.
- Order-sorted algebras do not allow sort constructors, nor sort union and intersection. Sort inclusions are allowed only in signatures, not in conditional axioms.
- Partial operations are represented in order-sorted algebras by the disciplined use of a constant that denotes a particular element. Notational conventions are required to ensure the proper treatment of this element (e.g., it is not allowed to test for equality with it).

One could in fact simulate unified algebras using order-sorted algebras: by introducing values that are *tokens* for sorts, and defining truth-valued operations on these values corresponding to inclusion and classification. But it is not clear that this simulation would be convenient enough for practical use.

Smolka [29] gave a reduction of order-sorted Horn logic to unsorted Horn logic using tokens for sorts, and treating inclusion and classification as predicates. Recently he has developed an unsorted Horn clause "type logic" [30] which is closely related to unified algebras. The main difference is that his framework is based on partial algebras, so only strict operations are considered; also, he leaves union and intersection of types to be specified by the user, rather than building them into the framework (one could do that with unified algebras too, but that would make unified specifiations more tedious). Scollo has reported [27] that Manca and Salibra [16] have recently proposed a framework somewhat similar

to that of Smolka. It will be interesting to see whether a useful notion of data constraint can be provided for these partial algebra frameworks.

Next, consider how generic types are specified in OBJ by parameterized modules:

- Explicit instantiation is needed, usually with translation of notation, which prevents instances from being regarded as subtypes (note also that OBJ's conditional operation needs to be a "built-in", otherwise every module using it would have to instantiate its parameterized specification).
- Generic types cannot be combined without specifying a new parameterized module.
- Elements can only be used as parameters of modules by introducing new modules just for them.

Finally, regarding module declarations in OBJ:

- They are sequential—mutual recursion is not possible;.
- They cannot be split into "interfaces" and "definitions".
- The default is for imported notation to be exported, and it is tedious to override this default.
- Artificial modules are needed to avoid unintentional duplication, when several modules are to share notation.

Against these rather negative comments should be set the fact that order-sorted algebras and OBJ are a great advance over many-sorted algebras and earlier specification languages; also that OBJ has been implemented and has been shown to be a useful tool. Indeed, this author previously adopted OBJ as the basis for specifying the action notation used in Action Semantics [20, 22], and used an early version of OBJ3 [13] to (partially) check an equational specification of functional actions. The development of unified algebras and modules took order-sorted algebras and OBJ as the starting point.

Finally, it should be noted that there are many other frameworks where "types" may be treated in the same way as elements, with operations on types; most of them originate from Scott's domain theory [28] or from Martin-Löf's type theory [17]. The foundations of these frameworks

seem to be essentially different from those of unified algebras. Moreover, they cater for higher-order functions, and it seems difficult to combine higher-order functions with type inclusion and monotonicity [18].

Conclusion

The framework of unified algebras and modules seem to have some attractive features, compared to alternative frameworks. However, it is too early to tell whether the unified framework will be useful enough in practice to justify its rather presumptuous name. So far, the only experience of using the framework is my own efforts to specify the abstract data and process types of "action notation", which is a (profoundly) polymorphic notation for use in Action Semantics [22].

One topic that needs to be investigated thoroughly is the right notion of "implementation" for unified algebras. As sorts correspond to nondeterministic choices, it seems natural to let implementations be more deterministic than specifications by contracting sorts.

Note that it is easy to extend unified algebras to allow the specification of predicates as well as operations; then so-called "Structural Operational Semantics" [23], also known as "Natural Semantics" [14], can be specified in the unified framework.

Acknowledgments. Inspiration for unified algebras came from the work of Goguen and Meseguer, Smolka, Aït-Kaci, and Wadge. The modules were inspired by the work of Bidoit, Burstall and Goguen, Clerici and Orejas, Reichel, Sannella, Tarlecki, and Wirsing. Thanks for correction and guidance during the development of unified algebras to Hartmut Ehrig, Joseph Goguen, Gert Smolka, and Andrzej Tarlecki, and for suggestions and encouragement to Brian Mayoh, Bernd Mahr, and David Watt.

References

[1] M. Broy, P. Pepper, and M. Wirsing. On the algebraic definition of programming languages. *ACM Trans. Prog. Lang. Syst.*, 9:54–99, 1987.

[2] R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In *Proc. 5th IJCAI*, 1977.

[3] R. M. Burstall and J. A. Goguen. The semantics of CLEAR, a specification language. In *Proc. Winter School on Abstract Software Specifications (Copenhagen, 1979)*. Springer-Verlag (*LNCS* 86), 1980.

[4] *The Concise Oxford dictionary of current English.* Oxford University Press, sixth edition, 1976.

[5] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1985.

[6] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proc. POPL'85.* ACM, 1985.

[7] J. A. Goguen. Order sorted algebra. Semantics and Theory of Computation Report 14, UCLA Computer Science Dept., 1978.

[8] J. A. Goguen. Higher order functions considered unnecessary for higher order programming. Technical Report SRI-CSL-88-1, Computer Science Lab., SRI International, Jan. 1988.

[9] J. A. Goguen and R. M. Burstall. Introducing institutions. In *Proc. Logics of Programming Workshop*, pages 221–256. Springer-Verlag (*LNCS* 164), 1984.

[10] J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for computer science. Report CSLI-85-30, CSLI, Stanford University, 1985.

[11] J. A. Goguen and J. Meseguer. Order-sorted algebra I: Partial and overloaded operators, errors and inheritance. Technical report, Computer Science Lab., SRI International, 1987.

[12] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. T. Yeh, editor, *Current Trends in Programming Methodology, Volume IV*. Prentice-Hall, 1978.

[13] J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, Computer Science Lab., SRI International, 1988.

[14] G. Kahn. Natural semantics. In *Proc. STACS'87*. Springer-Verlag (*LNCS* 247), 1987.

[15] B. Mahr and J. A. Makowsky. Characterizing specification languages which admit initial semantics. *Theoretical Comput. Sci.*, 31:49–59, 1984.

[16] V. Manca and A. Salibra. On the power of equational logic: Applications and extensions. In *Proc. 1st Int. Conf. on Algebraic Logic (Budapest, August 1988)*. To appear.

[17] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In *Proc. Logic Colloquium '73*. North-Holland, 1975.

[18] J. C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76:211–249, 1988.

[19] P. D. Mosses. Unified algebras and action semantics. In *Proc. STACS'89*. Springer-Verlag. To appear.

[20] P. D. Mosses. A basic abstract semantic algebra. In *Proc. Int. Symp. on Semantics of Data Types (Sophia-Antipolis)*. Springer-Verlag (*LNCS* 173), 1984.

[21] P. D. Mosses. Unified algebras and institutions (extended abstract). Internal Report DAIMI IR–83, Computer Science Dept., Aarhus University, 1988. Available from the author; full version in preparation.

[22] P. D. Mosses and D. A. Watt. The use of action semantics. In *Proc. IFIP TC2 Working Conference on Formal Description of Programming Concepts III (Gl. Avernæs, 1986)*. North-Holland, 1987.

[23] G. D. Plotkin. A structural approach to operational semantics. DAIMI FN–19, Computer Science Dept., Aarhus University, 1981. Available only from University of Edinburgh.

[24] H. Reichel. Initially-restricting algebraic theories. In *Proc. MFCS'80*, pages 504–514. Springer-Verlag (*LNCS* 88), 1980.

[25] H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras*, volume 2 of *The International Series of Monographs on Computer Science*. Oxford University Press, 1987.

[26] D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76:165–210, 1988.

[27] G. Scollo. Typed equational types: Pragmatics. Talk at 6th ADT Workshop, Berlin, Aug. 1988.

[28] D. S. Scott. Data types as lattices. *SIAM J. Comput.*, 5(3):522–587, 1976.

[29] G. Smolka. Order-sorted Horn logic: Semantics and deduction. SEKI Report SR–86–17, FB Informatik, Universität Kaiserslautern, 1986.

[30] G. Smolka. Type logic. Abstract for 6th ADT Workshop, Berlin, Aug. 1988.

[31] G. Smolka, W. Nutt, J. A. Goguen, and J. Meseguer. Order-sorted equational computation. SEKI Report SR–87–14, FB Informatik, Universität Kaiserslautern, 1987.

[32] A. Tarlecki. On the existence of free models in abstract algebraic institutions. *Theoretical Comput. Sci.*, 37:269–304, 1985.

Appendix: The Modular Specification Language

Abstract Syntax.

n : Natural ; p : Positive

$\Longrightarrow$

Modules	$\geq$	Identification "." Modules | Basic |
		Modules Modules ;
Basic	$\geq$	"use" Identification | Basic Translation |
		"local" Basic |
		"constant" Symbol_0 | "operation" Symbol_p |
		Clause | Basic Basic |
		"constant" Symbol_0 Relator Term |
		"operation" Symbol_p ":" Functionality_p |
		"constrain" Identification |
		"constrain" Identification "over" Identification ;
Identification	$\geq$	Identification "." Identification ;
Translation	$\geq$	Symbol_n ":=" Symbol_n |
		Translation ";" Translation ;
Symbol_n	$\geq$	Identification "." Symbol_n ;
Clause	$\geq$	Formula | Formula "$\Longrightarrow$" Clause |
		Clause ";" Clause | Symbol_p ":" Functionality_p ;
Formula	$\geq$	Term Relator Term | Formula ";" Formula ;
Relator	$\geq$	"=" | "$\leq$" | "$\geq$" | ":" | ":-"

p : Positive

$\Longrightarrow$

Term	$\geq$	Variable | Symbol_0 | Symbol_p Terms_p ;
Terms_1	$\geq$	Term ;
Terms_2	$\geq$	Term "2" ;
Terms_{p+1}	$\geq$	Term "," Terms_p ;
Functionality_p	$\geq$	Terms_p "$\rightarrow$" Term | Terms_p "$\rightsquigarrow$" Term |
		Terms_p "$\Rightarrow$" Term | Attribute_p |
		Functionality_p Functionality_p ;
Attribute_2	$\geq$	"associative" | "commutative" |
		"idempotent" | "unit" Term ;
Attribute_p	$\geq$	"strict" | "defined" | "elementary"