Specifying Type Systems with Multi-Level Order-Sorted Algebra[†]

Martin Erwig
FernUniversität Hagen, Praktische Informatik IV
58084 Hagen, Germany
erwig@fernuni-hagen.de

Abstract

We propose to use order-sorted algebras (OSA) on multiple levels to describe languages together with their type systems. It is demonstrated that even advanced aspects can be modeled, including, parametric polymorphism, complex relationships between different sorts of an operation's rank, the specification of a variable number of parameters for operations, and type constructors using values (and not only types) as arguments.

The basic idea is to use a signature to describe a type system where sorts denote sets of type names and operations denote type constructors. The values of an algebra for such a signature are then used as sorts of another signature now describing a language having the previously defined type system. This way of modeling is not restricted to two levels, and we will show useful applications of three-level algebras.

1 Introduction

The concept of multi-level algebra (MLA) was initiated from our work on extending data models by new data types [2]. Although many-sorted algebra can be conveniently used to describe non-standard data models many important aspects remain unformalized. Even the generalization to OSA, though nicely expressing subtypes and the notions of inheritance and overloading (Section 2), is not able to model the powerful concept of parametric polymorphism. Parametric order-sorted algebra [4] offers a partial solution, but there are still dependencies that cannot be expressed. For example, it is not clear, in general, how to define a parametric module that is not allowed to accept an instance of itself as a parameter. This is needed, for instance, to define a sequence constructor that is not allowed to be nested. In contrast, this is possible with *two levels* of OSA which is demonstrated in Section 3. After introducing the notion of *lifting* in Section 4 we will consider three-level algebras in Section 5. In Section 6 we use the formalism developed thus far to specify type systems of data models. Finally, concluding remarks and a comparison with other approaches is given in Section 7.

[†]In: *3rd Int. Conf. on Algebraic Methodology and Software Technology*, Springer, 1993, pp. 177–184.

2 Order-Sorted Algebra: Modeling Subtype Polymorphism and Overloading

An operation symbol that is used to denote different functions is said to be *overloaded*. When these functions are only loosely related, this is also called *ad hoc polymorphism*. On the other hand, the types on which the different functions operate may be related by subtyping, i.e., one argument type is a subset of another argument type for the same operation symbol. Then, no matter which function is taken, an application (whenever this makes sense) for one argument always yields the same result.[1] Such a situation is called *subtype polymorphism*. If a function is applicable to a class of types without needing to know the exact extent or structure of any type, this is an example of *parametric polymorphism*. This includes the identity function (applicable to all types) and the length function (applicable to all sequences). The code for such functions is independent of the type parameter since it does not inspect objects of that type.

For standard definitions of OSA refer to [3]. Now, consider the following signature (on the left) for the polymorphic operation $+$.

types nat, int
order nat $\leq$ int
funs 0: $\rightarrow$ int
0: $\rightarrow$ nat
$+$: int $\times$ int $\rightarrow$ int
$+$: int $\times$ nat $\rightarrow$ int
$+$: nat $\times$ int $\rightarrow$ int
$+$: nat $\times$ nat $\rightarrow$ nat

types nat, int
order nat $\leq$ int
funs 0: $\rightarrow$ nat
$+$: int $\times$ int $\rightarrow$ int
$+$: nat $\times$ nat $\rightarrow$ nat

Even this small example indicates that repeating an operation with many different ranks may become very cumbersome. Instead, it would be nice to define an operation with a "high" rank (with respect to the subtype order $\leq$) once and let the lower ranks be inferred automatically. This can be achieved by using a *signature specification* (shown on the right).

Definition 1 Any order-sorted signature $(S, \leq, \Sigma)$ is at the same time a *signature specification*. The *induced signature* $(S, \leq, IND(\Sigma))$ is defined by $IND(\Sigma) = \Sigma \cup \{\sigma_{w'',s} \mid \exists\, \sigma \in \Sigma_{w,s} : (w'' \leq w \wedge \forall\, \sigma \in \Sigma_{w',s'} : w' \leq w \Rightarrow w' \leq w'')\}$ □

Using signature specifications means to factorize operations' types along a $\leq$-chain of their arguments. The induced signature amounts to a feature which is termed *inheritance* in object-oriented languages.

Now suppose we have to define an operation "$<$" on numbers and strings (which are assumed to be not related by $\leq$). One approach is to give each signature entry separately. This becomes tedious as the number of data types for which "$<$" is defined grows. So it is much more convenient to group all the

[1] For example, the $+$ operation is defined on $\mathbb{N}$ and $\mathbb{R}$. Formally, we have two different functions $+_{\mathbb{N}}$ and $+_{\mathbb{R}}$, but their behavior is identical on $\mathbb{N} \cap \mathbb{R}$.

sorts in a *kind* [1], e.g., $\mathsf{ORD} = \{\mathsf{nat}, \mathsf{int}, \mathsf{str}\}$, and then to define all signature entries by a type scheme:

$\forall\ ord \in \mathsf{ORD}.\ <\colon\ ord \times ord \rightarrow \mathsf{bool}$

Apart from saving space, this notation is more descriptive w.r.t. the language being defined since the overloading of "<" is not "scattered" over different places in the signature. To use such type schemes in signatures we need the notion of an *extended signature specification*. Therefore, let K be the set of kinds where a kind is a set of sorts. The idea is to allow kind variables to be used like sorts in signatures.

Definition 2 Given a set of sorts S and a kind-sorted family $Y = \{Y_k \mid k \in K\}$ of disjoint sets of variables, an *Y-extended signature specification* is an order-sorted signature $(S', \leq, \Sigma)$ with $S' = S \cup Y$ and $\leq\ \subseteq S' \times S'$. The *induced signature* $(S', IND(\leq), IND(\Sigma))$ is defined by[2]

$IND(\Sigma) = \{\sigma_{w',s'} \mid \exists\ \sigma \in \Sigma_{w,s} : (w', s') = [s''/y](w, s) \wedge y \in Y_k \wedge s'' \in k\}$

$IND(\leq) = \{(s', t') \mid \exists\ (s, t) \in \leq\colon\ (s', t') = [s''/y](s, t) \wedge y \in Y_k \wedge s'' \in k\}$ □

Note that we now have two levels of signature specification available, i.e., the order-sorted signature induced by an extended signature specification can again be regarded as a signature specification (in the sense of Definition 1) which itself induces an order-sorted signature.

We observe that the $\leq$ order relates types in a way that induces kinds similarly to the above example: Each type s defines a kind SUB_s which contains s and all subtypes (with respect to $\leq$) of s. Subtype polymorphism can then be expressed by using kind variables ranging over such SUB kinds, or, to put it in other words, we could, in principle, define OSA in terms of many-sorted algebra plus an appropriate kind structure.

3 Two-Level Algebra: Describing Parametric Polymorphism and Type Constructors

A type constructor takes one or more types as arguments and produces a new type as result. The sequence constructor (seq), e.g., takes a type, say, int, and produces the type containing all sequences of integers. Of course, seq may be applied to other types as well, but in some languages where nested sequences are not allowed (for instance, database languages) it must not be applied to sequence types. In that case, the argument types for seq are a proper subset of all types and can be grouped into an appropriate kind. Similarly, the result types form a kind, too.

We can regard kinds and type constructors as sorts and operations, respectively, of an order-sorted signature. The example of unnested sequences can then be expressed as shown below (nesting of sequences could be allowed by simply defining $\mathsf{SEQ} \leq \mathsf{ARG}$).

[2] The notation $[s/y]t$ denotes the substitution of all occurrences of the variable y in t by s.

typesystem UNNESTED
kinds ARG, SEQ
tcons int, str, bool: $\rightarrow$ ARG
seq: ARG $\rightarrow$ SEQ

language LISTS
types from UNNESTED
funs nil: $\rightarrow$ *seq*
cons: *arg* $\times$ seq(*arg*) $\rightarrow$ seq(*arg*)
hd: seq(*arg*) $\rightarrow$ *arg*
tl: seq(*arg*) $\rightarrow$ seq(*arg*)
length: *seq* $\rightarrow$ int

Note that "$\forall$ *seq* $\in$ SEQ. *seq*" denotes the same types as "$\forall$ *arg* $\in$ ARG. seq(*arg*)". Thus, we can use *seq* in the type specifications for nil and length since we do not need to refer to the argument type of the respective sequences.

Also note carefully that type expressions containing variables like *seq* are not types themselves, but simply specifications of sets of types. One impact is that expressions like nil or cons(nil, nil) are not type-correct since the type of the overloading of nil cannot be resolved. This situation is called *predicative polymorphism* [10] and is certainly a somewhat restricted form of parametric polymorphism. The good thing about it is that we can have set-theoretical models, which is not true for more general forms of polymorphism.

The signature UNNESTED defines merely the typing of type constructors. The semantics usually consists of two parts: On the one hand, algebraic properties of type constructors can be specified by equations (for instance, associativity of a product operator). The set of sorts is then taken modulo such a specification (in our example this was not necessary). On the other hand, the effects of type constructors on the carrier sets need to be given by additional functions.

Definition 3 An order-sorted signature is a 1^{st}-*level signature*, and an order-sorted algebra is a 1^{st}-*level algebra*. Given an n^{th}-level signature $(S', \leq', \Sigma')$ and an n^{th}-level Σ'-algebra $\mathcal{B}$, an order sorted signature $(S, \leq, \Sigma)$ is an $n+1^{\text{st}}$-*level signature* depending on Σ' and $\mathcal{B}$ if $S = \bigcup_{s \in S'} s^{\mathcal{B}}$. A Σ-algebra $\mathcal{A}$ is an $n+1^{\text{st}}$-*level algebra* if for each $\sigma_{w,s} \in \Sigma'$ there is a functor $\overline{\sigma}_{w,s}$ (called *type constructor*) and if for each $s \in S$ such that $s = \sigma^{\mathcal{B}}_{w,s}(t_1, \ldots, t_n)$ (with $w = s_1 \ldots s_n$ and $t_i \in s_i^{\mathcal{B}}$ for $1 \leq i \leq n$) we have $s^{\mathcal{A}} = \overline{\sigma}_{w,s}(t_1^{\mathcal{A}}, \ldots, t_n^{\mathcal{A}})$. The functions $\overline{\sigma}_{w,s}$ define the *constructor semantics* for Σ', and $\mathcal{A}$ *depends* on (the higher level) $\mathcal{B}$ and the constructor semantics for Σ'. □

Note that the individual algebra levels are denoted by counting backwards (with regard to the construction history), i.e., an $n+1^{\text{st}}$-level algebra $\mathcal{A}$ depending on the n^{th}-level algebra $\mathcal{B}$ is said to be on the first level whereas $\mathcal{B}$ is said to be on second level, etc.. In particular, when Σ' is used to describe types, we also say that Σ' is on *type level* and Σ is on *language level*. In the following we will always work with term algebras, i.e., we have, e.g., $\mathsf{ORD}^{\mathcal{B}} = \{\mathsf{nat}, \ldots\}$.

The constructor semantics for the seq constructor is defined by:

$$\mathsf{seq}(s)^{\mathcal{A}} = \overline{\mathsf{seq}}(s^{\mathcal{A}}) = (s^{\mathcal{A}})^*$$

4 Algebra Lifting

All type constructors presented so far have built new types from other types. But there are some type constructors that are also based on *values*. The array constructor, e.g., takes in addition to the component type two values of a scalar type. Other examples are constructors for fixed length strings or subranges.

In order to retain the clear separation of the kind/type/value levels Cardelli [1] proposes to "lift" values onto the type level. For instance, introduce for each value $n \in \mathsf{nat}$ a new type $\underline{n}$ with the carrier being $\underline{n}^{\mathcal{A}} = \{n\}$. Moreover, create a new kind, $\underline{\mathsf{nat}}$, with $\underline{\mathsf{nat}}^{\mathcal{B}} = \{\underline{n} \mid n \in \mathsf{nat}^{\mathcal{A}}\}$. Then array can be used exclusively on the type level, as in $\mathsf{array}(\underline{1}, \underline{9}, \mathsf{bool})$.

Let Σ'_L denote the set of type constructors that need lifted types. In order to specify a type system and a language using Σ'_L the following steps have to be performed (for a two-level algebra):

(i) Define the type system without Σ'_L. Call the signature Σ'_0.
(ii) Define Σ_0, the part of the language not needing types constructed by Σ'_L.
(iii) Perform lifting of Σ'_0 and Σ_0, and add Σ'_L to $\underline{\Sigma'_0}$, i.e., let $\Sigma' = \underline{\Sigma'_0} \cup \Sigma'_L$.
(iv) Finally, define Σ with regard to Σ'.

We can specify Σ_L together with Σ_0 in one step. Thus, array is defined by:[3]

typesystem ARRAYS
kinds ARR, ANY
order ARR $\leq$ ANY
tcons nat, int, str, bool: $\rightarrow$ ANY
array: $\underline{\mathsf{nat}} \times \underline{\mathsf{nat}} \times \mathsf{ANY} \rightarrow \mathsf{ARR}$

The constructor semantics is given by:

$$\mathsf{array}(\underline{n}, \underline{m}, t)^{\mathcal{A}} = \overline{\mathsf{array}}(\underline{n}^{\mathcal{A}}, \underline{m}^{\mathcal{A}}, t^{\mathcal{A}}) = \overline{\mathsf{array}}(\{n\}, \{m\}, t^{\mathcal{A}}) = \{n, \ldots, m\} \rightarrow t^{\mathcal{A}}$$

Array operations can be defined by (assume quantifications "$\forall\ \underline{n}, \underline{m} \in \underline{\mathsf{nat}}$"):

types from ARRAYS
order nat $\leq$ int
funs newarray: nat $\times$ nat $\times$ *any* $\rightarrow$ array$(\underline{n}, \underline{m}, any)$
select: array$(\underline{n}, \underline{m}, any) \times$ nat $\rightarrow$ *any*
update: array$(\underline{n}, \underline{m}, any) \times$ nat $\times$ *any* $\rightarrow$ array$(\underline{n}, \underline{m}, any)$

Note that with the above definition range checking (for select/update) is not expressible on the type level since an expression select(newarray(1,9,true),15) is type correct w.r.t. to the above signature. This can be remedied by using three levels of algebras. Another application of lifting arises in the modeling of operations with a variable numbers of parameters. Note that lifting of operations is also conceivable. Then type constructors taking predicates and denoting subtypes w.r.t. theses predicates can be defined.

[3] We do not list lifted kinds explicitly.

5 Three-Level Algebras

Consider the function [ ] for constructing sequences, which is defined for an arbitrary number of arguments. The signature entries are:

$[\,]\colon \rightarrow \mathit{seq}$
$[\,]\colon \mathit{arg} \rightarrow \mathsf{seq}(\mathit{arg})$
$[\,]\colon \mathit{arg} \times \mathit{arg} \rightarrow \mathsf{seq}(\mathit{arg})$
$\ldots$

To denote these signature entries we need for each argument type t a kind containing all product types over t. This can be achieved as follows: We define a *kind constructor* list (this is an operation on level three with the same semantics as seq). Now, $\mathsf{list}(K)$ denotes for a kind K all sequences of sorts from K. If, e.g., $K^{\mathcal{B}}_{\mathsf{nat}} = \{\mathsf{nat}\}^4$, the quantification "$\forall\ \mathit{natlist} \in \mathsf{list}(K_{\mathsf{nat}})$" binds the sequences $\langle\rangle$, $\langle\mathsf{nat}\rangle$, $\langle\mathsf{nat}, \mathsf{nat}\rangle$, ... to *natlist*. The desired product types can be obtained by "inserting" a "$\times$" type constructor between each two adjacent types in a sort sequence. This is done by the higher order function fold:

$$\begin{aligned}
\overline{\mathsf{fold}}(\sigma, \langle\rangle) &= \epsilon \\
\overline{\mathsf{fold}}(\sigma, \langle t_1\rangle) &= t_1 \\
\overline{\mathsf{fold}}(\sigma, \langle t_1, t_2, \ldots, t_n\rangle) &= \sigma(t_1, \overline{\mathsf{fold}}(\sigma, \langle t_2, \ldots, t_n\rangle))
\end{aligned}$$

Now the type of [ ] (for nat-sequences only) can be specified by:

$[\,]\colon \mathsf{fold}(\times, \mathit{natlist}) \rightarrow \mathsf{seq}(\mathsf{nat})$

A more precise account of this kind of specification requires higher order algebras [8] and lifting on higher levels. Finally, for the convenient specification of multi-level algebras we need a language that allows for the use of terms of all levels in the definition of operations' ranks. This is covered in the long version of this paper.

6 Specifying Data Models

A relation is a set of tuples, and the components of the tuples each have a fixed type and are named. In the "flat" relational model, relations are built only over atomic types, whereas in the NF^2 model tuples may contain whole relations. The relational model demonstrates the use of two languages on different levels:

(i) The first is the language of schemas, or, tuple types. A schema is a type, and we have operations on schemas (i.e., type constructors), such as adding an identifier/type pair or merging two schemas. Another type constructor builds relation types from schemas.

(ii) The second language is that of relational algebra working on relations that are instances of relation types. Operations on relations, such as natjoin or select, belong to the language level.

The following type system defines the operations on schemas and, of course, the rel constructor. The fun constructor will be needed for selection expressions.

[4] K_{nat} can be obtained by lifting.

typesystem RELMODEL
kinds TUP, REL, ATOM, FUN
tcons null: $\rightarrow$ TUP
int, str, bool: $\rightarrow$ ATOM
add: $\underline{\text{str}} \times \text{ATOM} \times \text{TUP} \rightarrow \text{TUP}$
merge: $\text{TUP} \times \text{TUP} \rightarrow \text{TUP}$
mix: $\text{TUP} \times \text{TUP} \rightarrow \text{TUP}$
names: $\text{TUP} \rightarrow \text{list}(\underline{\text{str}})$
rel: $\text{TUP} \rightarrow \text{REL}$
fun: $\text{TUP} \times \underline{\text{bool}} \rightarrow \text{FUN}$

Note that we use lifted strings as identifiers in tuples. add, merge, and mix are type constructors which will be used on the language level to perform pattern matching on relational schemas, and names yields the sequence of identifiers of a tuple.

As it stands, the structure of schemas is specified only very loosely. This would be sufficient as long as we considered only schemas (in fact, the constructor semantics for schema type constructors is not really needed since schemas are used on the language level only to build relations over them). However, being faced with the need to describe the constructor semantics of rel it is very helpful to fix a concrete representation for schemas, in particular, it is convenient to think of a schema as a sequence of (identifier, type name) pairs. This can be done by introducing on the third level a kind constructor schema which is defined to provide just this representation (see full paper). For now we just presume that representation of schemas. We have:

$$\langle (i_1, t_1), \ldots, (i_n, t_n) \rangle^{\mathcal{A}} = i_1^{\mathcal{A}} \times t_1^{\mathcal{A}} \times \ldots \times i_n^{\mathcal{A}} \times t_n^{\mathcal{A}}$$

Now we can assign a constructor semantics to rel.

$$\text{rel}(\langle (i_1, t_1), \ldots, (i_n, t_n) \rangle)^{\mathcal{A}} = \overline{\text{rel}}(i_1^{\mathcal{A}} \times t_1^{\mathcal{A}} \times \ldots \times i_n^{\mathcal{A}} \times t_n^{\mathcal{A}}) = 2^{i_1^{\mathcal{A}} \times t_1^{\mathcal{A}} \times \ldots \times i_n^{\mathcal{A}} \times t_n^{\mathcal{A}}}$$

With the auxiliary function *isect* we can give the following definitions for the remaining type constructors.

$$\begin{aligned}
isect(s, s') &= \langle (i, a') \in s' \mid \exists a : (i, a) \in s \rangle \\
\text{add}(i, t, s) &= \text{merge}(\langle (i, t) \rangle, s) \\
\text{merge}(s, s') &= s \cdot (s' - isect(s, s')) \\
\text{mix}(s, s') &= \text{merge}(s, s') \cdot \langle (i', a) \mid \exists (i, a) \in isect(s, s') \rangle \\
\text{names}(s) &= \langle i \mid \exists (i, a) \in s \rangle
\end{aligned}$$

(For simplicity we assume in the definition of mix that i' does not occur as an identifier in s or s'. Since i is a lifted string, say, $\underline{s}$, we can think of i' as a suitable renaming of s, say, s', resulting in $\underline{s'}$.)

Finally, we can define the language of relational algebra (we only give operations on relations and omit, e.g., comparison operators). The function tuple is needed to construct (one-tuple) relations from lifted strings and atomic values.

language RELALG
types from RELMODEL
funs tuple: $tup \rightarrow \mathsf{rel}(tup)$
union: $rel \times rel \rightarrow rel$
project: $\mathsf{rel}(\mathsf{merge}(tup_1, tup_2)) \times \mathsf{names}(tup_1) \rightarrow \mathsf{rel}(tup_1)$
product: $\mathsf{rel}(tup_1) \times \mathsf{rel}(tup_2) \rightarrow \mathsf{rel}(\mathsf{mix}(tup_1, tup_2))$
natjoin: $\mathsf{rel}(tup_1) \times \mathsf{rel}(tup_2) \rightarrow \mathsf{rel}(\mathsf{merge}(tup_1, tup_2))$
select: $\mathsf{rel}(tup) \times \mathsf{fun}(tup, \mathsf{bool}) \rightarrow \mathsf{rel}(tup)$
attrib: $\mathsf{add}(\underline{str}, atom, tup) \times \underline{str} \rightarrow atom$

The definition of an NF^2 model [14] can be obtained by simply adding an order specification $\mathsf{REL} \leq \mathsf{ATOM}$ to the type system. This means that attributes need not be atomic values any more. Otherwise, type system and language require no changes.

The two operations nest and unnest are defined by:

funs nest: $\mathsf{names}(tup_1) \times \underline{str} \times \mathsf{rel}(\mathsf{merge}(tup_1, tup_2)) \rightarrow \mathsf{rel}(\mathsf{add}(\underline{str}, \mathsf{rel}(tup_1), tup_2))$
unnest: $\mathsf{rel}(\mathsf{add}(\underline{str}, \mathsf{rel}(tup_1), tup_2)) \times \underline{str} \rightarrow \mathsf{rel}(\mathsf{mix}(tup_1, tup_2))$

Note how nicely these typings reflect the fact that unnest is the inverse operation of nest.

7 Conclusions and Related Work

Two-level algebras were already used in [13] to specify categories with certain properties for theoretical investigation and in [7] for the formalization of the composition of specifications. Meinke [9] gives a categorical semantics for two-level specifications. In contrast, our concern is the specification of type systems, more specifically, the formal description of data models and query languages.

Unlike [13, 7, 9] MLA is not limited to two levels, and we have indicated that especially a third level can be extremely helpful: One usage is to describe overloading of operations with functions on different numbers of parameters. Another application is the generalization of certain type constructors (e.g., the definition of an array type constructor not only for a fixed index type but for a class of scalar index types). In some cases this also results in sharper type descriptions (the generalized array definition prevents out-of-range errors).

Another difference between [13, 7] and our work is that we employ more than only one sort on level two. This is necessary, for instance, to facilitate the description of a type constructor for unnested sequences.

Focusing on two levels and ignoring extensions, such as lifting, the work in [9] is in a sense more general than MLA since the first level (called "combinator signature") is fully parameterized by a second level whereas in MLA, a second level is fixed and a first level is constructed w.r.t. such a fixed second level; the dependency is "encapsulated" by construction. The advantage of our approach is that theorems which have to be proved in [9] are trivially true in MLA.

Finally, let us summarize some points counting in favor of using MLA and exhibiting its scope.

- All kinds of polymorphism (subtype, ad hoc, parametric) are describable within one formalism.
- Type systems can be easily extended by new structures. This is important to meet changing requirements of new applications.
- The definition of properties of type constructors (e.g., associativity) is separated from the constructor semantics.
- Recently, general approaches to the type checking of languages that use overloading in a systematic way have become available [11, 6, 12]. In many cases these methods are directly applicable to languages defined by MLA.

References

[1] L. Cardelli. Types for Data Oriented Languages. In *Conf. on Extending Database Technology*, LNCS 303, pp. 1–15, 1988.

[2] M. Erwig and R. H. Güting. Explicit Graphs in a Functional Model for Spatial Databases. Report 110, FernUniversität Hagen, 1991.

[3] M. Gogolla. Partially Ordered Sorts in Algebraic Specifications. In *9th Coll. on Trees in Algebra and Programming*, pp. 139–153, 1984.

[4] J. A. Goguen. Higher-Order Functions Considered Unnecessary for Higher-Order Programming. In D. A. Turner, ed., *Research Topics in Functional Programming*, pp. 309–352. Addison-Wesley, 1990.

[5] J. A. Goguen and J. Meseguer. Order-Sorted Algebra I. Report, SRI Int., 1989.

[6] S. Kaes. Type Inference in the Presence of Overloading, Subtyping and Recursive Types. In *ACM Conf. on Lisp and Functional Programming*, pp. 193–204, 1992.

[7] J. Leszczylowski and M. Wirsing. Polymorphism, Parameterization and Typing: An Algebraic Specification Perspective. In *STACS 91*, LNCS 480, pp. 1–15, 1991.

[8] K. Meinke. Universal Algebra in Higher Types. In *Workshop on Specification of Abstract Data Types*, LNCS 534, pp. 185–203, 1990.

[9] K. Meinke. Equational Specification of Abstract Types and Combinators. In *5th Workshop on Computer Science Logic*, LNCS 626, pp. 257–271, 1991.

[10] J. C. Mitchell. Type Systems for Programming Languages. In J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. B*, pp. 367–458. Elsevier, 1990.

[11] T. Nipkow and C. Prehofer. Type Checking Type Classes. In *ACM Symp. on Principles of Programming Languages*, pp. 409–418, 1993.

[12] T. Nipkow and G. Snelting. Type Classes and Overloading Resolution via Order-Sorted Unification. In *Conf. on Func. Progr. and Comp. Arch.*, LNCS 523, pp. 1–14, 1991.

[13] A. Poigné. On Specifications, Theories, and Models with Higher Types. *Information and Control*, 68:1–46, 1986.

[14] H.-J. Schek and M. H. Scholl. The Relational Model with Relation-Valued Attributes. *Information Systems*, 11:137–147, 1986.