Accepted for publication in the *Procedings of the Eighth Workshop on Specification of Abstract Data Types, Dourdan, France, 26–30 August 1991,* Lecture Notes in Computer Science, Springer-Verlag, 1992.

Contents

The Use of Sorts in Algebraic Specifications

Peter D. Mosses*
Computer Science Department, Aarhus University,
Ny Munkegade, Bldg. 540, DK-8000 Aarhus C, Denmark

September 1992

Abstract

Algebraic specification frameworks exploit a variety of sort disciplines. The treatment of sorts has a considerable influence on the ease with which such features as partiality and polymorphism can be specified. This survey gives an accessible overview of various frameworks, focusing on their sort disciplines and assessing their strengths and weaknesses for practical applications. Familiarity with the basic notions of algebraic specification is assumed

1 Introduction

We are going to survey the variety of ways in which *sorts* are used in algebraic specifications. Let's agre first on some basic terminology. It is assumed that you are already familiar with the main concepts of algebraic specifications—otherwise see an expository text such as [12] or the Handbook chapter on algebraic specification [52]

An *algebra* consists essentially of a *universe*, or *carrier*, of *values*, together with some distinguished *operations* on values (operations of no arguments being called *constants*). The *signature* of an algebra provides *symbols* for distinguishing the operations. *Terms* are constructed from symbols and *variables*. In a particular algebra, an *assignment* of values to variables determines the values of terms.

An *algebraic specification* determines a signature, and restricts the class of all algebras with that signature using *axioms*: each algebra in the specified class has to *satisfy* all the axioms. An axiom is often just an equation between terms, universally quantified over all the variables that occur in it. A specification may also impose *constraints*, for instance to narrow the specified class to *initial* algebras, or to *reachable* ones. We call an algebra in the specified class a *model* for the specification. The class of specified algebras is generally called an *abstract*

*Internet: pdmosses@daimi.aau.dk

data type.[1]

The basic notions of signature, algebra, axiom, and satisfaction can be embellished in various ways, without departing from the fundamental idea of algebraic specification. This flexibility is captured formally by the notion of an *institution* [21, 22]. Roughly, an institution consists of particular kinds of signatures, structures, and axioms together with a satisfaction relation (between structures and axioms) that is invariant under signature translation. The structures may be algebra or they may be more general, e.g., first-order structures that have relations as well as operations; the definitions are formulated abstractly, using category theory. Some interesting results can be obtained independently of the details of particular institutions, for instance a general framework for modules has been provided [45]. Similarly for the notion of a *specification logic* [13], which is even more general than that of an institution.

The following variations on the theme of algebraic specfication—separately or together—correspond to particular institutions or specification logics.

- Operations may be *total* or *partial*.
- Values may be classified by *sorts*, and operations *restricted* to specified sorts of arguments.
- Carriers may be *structured*, e.g., as posets or lattices.
- *Relations* may be allowed, as well as operations.
- Operations may be *nondeterministic*.
- Operations may be *higher-order*, being considered as values themselves.
- Axioms may be restricted to special kinds of formulae, such as equations, conditional equations, Horn clauses, or first-order predicate sentences.
- Constraints may be allowed, e.g., to reachable, initial, or final algebras.
- Observationally or behaviourally equivalent algebras may be included in the class of specsed algebras even though they do not satisfy all the specified axioms.

Most of the above variations have been developed with the aim of improving the pragmatic aspects of algebraic specification of abstract data types, to make them better suited for *realistic* applications. Here, we are to focus on the use of *sorts* in algebraic specifications. This is quite a rich topic, particularly relevant to pragmatics. Subsidiary issues include the treatment of partiality and errors, subtypes, polymorphism, and type-checking. We restrict our attention to *first-order* frameworks, deferring a study of higher-order algebraic specifications to a future paper.

[1] Some authors prefer to reserve *abstract data type* for when the specified algebra form an isomorphism class.

2 Sorts

The essence of a *sort* is that it *classifies* a collection of *individual* values, according to some common properties. Thus a sort has an *extension*: the set of individuals that it classifies. But two sorts with the same extension may have different *intension*, and thus remain distinct. For instance, the sort of all integers greater than zero has the same extension as the sort of all natural numbers with a well-defined reciprocal, but these sorts may still be regarded as different; the difference is in their intension. When the *symbol* used to identify a sort is regarded as part of the sort's intension, different sort symbols always identify distinct sorts.

In everyday parlance, we tend to make little distinction between the nouns 'sort', 'type', and 'kind'. The conventional usage of these words in the computer science literature has, however, given them different connotations: *sorts* are rather mundane, subsidiary entities used for 'tidy housekeeping' in logic and algebraic specifications; *types* are generally much more exciting, as they have whole theories built around them, and they are related in interesting ways to logic; *kinds* are merely for classifying types. For simplicity, let's keep to the word 'sort' in this survey, even when we look at unorthodox algebraic specification frameworks where the usage of sorts is quite reminiscent of that of types and kinds. Sillily, we'll use 'individual' rather than 'object' or 'element'.

Now that we have some idea of *what* a sort is, let's consider *why* we should bother at all to specify sorts in algebraic specifications. Why is it off to classify a collection of individuals?

First of all, classification according to common properties is a fundamental *abstraction* principle, allowing us to perceive (or at least, to express our perception of) order amongst chaos. Simply by classifying a prices set of individuals together, we draw attention to the existence of *some* relationship between them.

Another important use of sorts is to allow us to specify the *functionalities* of operations, i.e., what sort of result each operation returns when applied to arguments of appropriate sorts. Some frameworks only allow one functionality to be specified for each operation, thus preventing so-called *overloading*, but this seems unfortunate: when exploited judiciously, overloading can be very useful. For instance, we might want to specify a print operation for all sorts of values. Or we might want an if-then-else operation, where the sort of the second and third arguments could be arbitrary. Functionality specifications can be regarded as particular kinds of axioma—although for technical reasons, they are more commonly treated as part of signatures, and usually at least one functionality for each operation must be specified.

When the extensions of sorts can overlap, we expect so-called *subsort polymorphism*. E.g., positive integers of sort pos may also be classified in the sort nat of all natural numbers. Then a product operation on natural numbers not only has functionality nat,nat→nat but also pos,pos→pos; similarly for sorts classifying just even (or just odd) numbers, and for the singleton sorts classifying just zero and one! However, many restrictions of the sorts of arguments lead to uninteresting sorts of returned values, and it is pointless to specify these

as functionalities.

In axioms, sorts are generally used for *restricting* assignments to *variables*. Axioms are often (implicitly or explicitly) universally quantified over all variables that occur in them; the use of sorts can restrict this quantification, and thereby the application of the axiom, to particular subsets of the universe. For instance, the product operation may be commutative on numbers but not on matrices; the commutativity axiom for product must then be restricted to numbers. In the absence of overlook operation, sort restrictions on variables are implicit, being determined by the functionalities of the operations applied to them.

Now let's consider what is perhaps the most common use of sorts: the attempt to avoid *partial* operations, by restricting each of them to a corresponding total operation on the domain of definition. This is desirable, because the logic and technical details of algebraic specifications are somewhat simpler when partial operations are avoided. On the other hand, we shall see that it isn't so easy to avoid partial operations completely.

When specifying mathematical structures such as groups and rings, the carrier of a specified algebra is *homogeneous*, and each operation can be applied to any individuals, usually returning a well-defined individual value. (Fields and categories involve partial operations, though.) But almost all interesting abstract data types for use in computer science—and a few in mathematics, such as vector spaces—involve *heterogeneous* carriers. In general, all operations on heterogeneous carriers are inherently partial, when considered as functions on the entire collection of individuals. For instance, an abstract data type of lists of numbers obviously involves numbers as well as lists; but there is no point in applying list operations (head, tail, etc.) to numbers, and although it can be useful to extend some numerical operations (product, for instance) to lists, this is by no means essential.

The use of sorts to classify the individuals of a heterogeneous collection, together with the specification of operation functionalities, allows us to forbid terms where operations are applied to unintended argument values. Thus we may regard operations as being restricted to values in the argument sorts specified in their functionalities. With values including both numbers and lists, for instance, product can be restricted to numbers, and head and tail to (nonempty) lists; the application of product to a list, or of head and tail to numbers, is simply forbidden, syntactically.

Often, it is easy to check for forbidden terms, using the functionalities of operations. This is important in connection with systems that implement algebraic specifications: the user can be warned about a simple mistake before a time-consuming and futile evaluation is started. But it isn't always so easy: for instance, it can be undecidable whether, in an application of the list operation head, the value of argument term is the empty list or not. If the application is forbidden, we get the anomalous situation where forbidden terms can be obtained from allowed ones, using the axioms as rewrite rules; if it is allowed, we are forced to take partial operations seriously.

An alternative technique for skirting around partial operations is to introduce

special *error* values, to represent the undefined results of operations when they are applied to unintended arguments. Then all operations are total, but of course they now have to be specified on the error values, as well as on the 'okay' values. This can be tedious, although certain assumptions, such as *strictness* on error values, allow most of the extra axioms to be left implicit.

Instead of trying to avoid partiality, one could simply accept it as a somewhat awkward fact of life, and consider *partial algebras* where operations are partial functions in the dual, mathematical sense. But one has to be careful about the precise interpretation of equality—strong or existential—and about the logic used for deduction. The notion of homomorphism between algebras is less obvious, too. In practice, because of such technical irritations, most popular approaches for algebraic specifications keep to total algebra, simulating partiality as best they can. In any case, there are other uses for sorts than trying to avoid partiality, and frameworks for partial algebra exploit sorts just as much as those for total algebras do. Hence we pay only scant attention to partial algebras in this paper.

To conclude this motivation for sorts, let us note a few uses of them that are perhaps somewhat less obvious than those explained above:

- *Term rewriting* is used to implement algebraic specifications. It generally involves large-scale searching of axioms (oriented as rewrite rules) to find a match with a part of a term to be evaluated. By keeping track of sorts (and subsorts), the search space can be dramatically reduced. Similarly for automatic theorem-proving.

- Sorts can be used to represent *nondeterministic* choices between individual. The individual classified by a sort are then regarded as alternative.

- Finally, sorts have major technical significance for the definition of so-called *initial* (or data) *constraints*. This is because the ordinary reduct functor forgets about operations, and about entire sorts of values; when there are no sorts, it doesn't forget any values at all!

So much for *what* sorts are, and *why* it is useful to specify them. But by focussing on sorts so much, we have been neglecting the individual values somewhat. It is important to bear in mind that we don't specify sorts for their own sake: it is the individual values, and the operations upon these, that are the aim of an algebraic specification, and the sorts are only there to *facilitate* the specification.

In the remaining sections we consider *how* sorts are specified in the major frameworks that have been developed over the past 15 years or more. The appendices illustrate the use of most of these frameworks to specify a simple abstract data type of lists.

3 Many-Sorted Algebras

Let us start with a brief look at the basic framework known as *many-sorted algebras* (MSA) in the formulation proposed by the ADJ group [24]. From a theoretical point of view, this framework is perhaps the most tractable of all those presented here; but it is also the one most beset by pragmatic deficiencies.

A signature of a many-sorted algebra consists of a set of sort symbols S together with a set of operation symbols and their functionalities. Overloading is (usually) allowed, so each operation symbol may have several functionalities. Axioms of MSA specifications are often restricted to sorted equations, or positive conditional equations; variables are restricted to specified sorts. Such specifications always have initial models.

The universe of a many-sorted algebra A consists of a family A_s of sets of values, one for each sort symbol $s \in S$. For each functionality $s_1, \ldots, s_n \to s$ of each operation symbol f in the signature, A provides a total function $f_A : A_{s_1} \times \cdots \times A_{s_n} \to A_s$. Note that the A_s may overlap, or even coincide.

The main pragmatic problem with ordinary MSA is how to accommodate *error* values. For instance, consider a specification of rational numbers: if there is only one sort of rational number, it ought to contain zero, but then division by zero cannot be forbidden by the functionality. Because all operations are required to be total, this results in a nonstandard rational value, i.e., an error value. (To specify the value of division by zero to be some particular standard rational value, e.g., zero, would amount to ignoring the existence of this error.) Similarly for a specification of possibly-empty lists: taking the head of the empty list gives an error value. Such error values are especially awkward in parameterized specifications, which are supposed to leave parameter sorts undisturbed.

The solution originally proposed [24] is quite costly. For each sort $s \in S$, one introduces an error value e_s, a truth-valued operation ok_s (specified to be true except on e_s), an if-then-else operation ife_s, and a plethora of derived operations. These operations can then be used to specify appropriate equations for error propagation. But 'the resulting total specification ... is unbelievably complicated' [24]. Se Sect. 9.1 for an example. The problems with this explicit treatment of errors led to the development of various frameworks where the treatment of errors can be specified more concisely; we consider the major ones in the next section.

Another pragmatic problem with MSA is the difficulty of specifying sort inclusions, i.e., *subsorts*. Suppose, for instance, that we want to have the sort nat for all natural numbers, and pos for positive ones. With MSA, we cannot specify that for each model A the carrier set A_{pos} *has* to be *included* in A_{nat} (or more precisely, that the difference between these carrier sets is just the value of the zero constant). Thus the relation of such $\{\mathsf{nat}, \mathsf{pas}\}$-sorted models to the standard mathematical algebra of natural numbers is obscure. Moreover, we have to specify each numerical operation twice: once on nat, and again on pos.

Order-sorted algebras address these problems directly; we shall consider them in Sect. 6. A less direct solution, available when conditional equations are

allowed, is to introduce an auxiliary operation $\mathsf{i} : \mathsf{pos} \to \mathsf{nat}$ and specify that it is injective, i.e., one-one. From a theoretical point of view, there is not much difference between an injective operation and a set inclusion. But in practice, specifications using such auxiliary operations explicitly would be rather tedious to write.

It is debatable whether *overloading* should be allows in MSA or not. Theoretically, overloading is dispensable: any class of algebras that can be specified with overloaded operation symbols can—up to a signature translation—also be specified without overloading. Pragmatically, however, overloading can be quite convenient, and it is often exploited in mathematical notation.

Some care is needed when overloading is allowed. For suppose that we have an overloaded constant c of two different sorts s_1, s_2, as well as an overloaded operation f with the functionalities $f : s_1 \to s$ and $f : s_2 \to s$. The term $f(c)$ is simply ambiguous! Its value could differ, according to whether the argument is regarded as being of sort s_1 or s_2. The sort of value required by the context should be the same in each case, namely s, so that doesn't help with disambiguation. Perhaps mathematicians don't worry too much about notational ambiguities arising from overloaded constants in their formulae, because they usually know which value is intended from a wider context. However, that hardly justifies allowing overloaded constants in algebraic specifications, where one generally aims for reusable modules that can be understood by themselves, independently of context.

Even when overloaded constants are forbidden in MSA, there is still a problem with ambiguity, although at a different level. Suppose two carrier sets A_{s_1} and A_{s_2} overlap, and let $x \in A_{s_1} \cap A_{s_2}$. When $f : s_1 \to s_1'$ and $f : s_2 \to s_2'$ is an overloaded operation, $f_A(x)$ is not, in general, well-defined: its value depends on whether x is regarded as an element of A_{s_I} or of A_{s_2}. Perhaps one should consider restricting modes of MSA specifications to those where each overloaded operation always gives the same result when applied to the same values, regardless of the sorts of the values? We shall return to this point when considering order-sorted algebras, where some attention has been paid to it.

Despite the problems concerning error values and overloading, much has been achieved within the MSA framework, and it provided the basis for the development and popularization of the entire topic of algebraic specification. Some users still prefer this straightforward framework to the more complicated ones considered in the rest of this survey.

4 Partiality

Having seen the complications that can arise in total algebraic specifications due to error values, we might be tempted to let operations be ordinary partial functions and represent errors by undefinedness, following Broy and Wirsing [8] and Reichel [44], see also [2, 9]. As ordinary partial functions are *strict* on undefinedness, error propagation is implicit; moreover, variables in axioms are only assigned defined values. Thus the auxiliary ok and ife operations introduced

in the preceding section are unnecessary here.

But there are some drawbacks. For instance, there is the dilemma of whether to interpret equations *existentially*, to hold only when the values of both the equated terms are defined, or *strongly*, to hold also when both the values are undefined. Similarly, homomorphisms could be total or partial functions, and each choice has certain merits.

Some care is needed to exclude models where operations are more partial than intended. As well as equations, one may specify definedness axioms $\mathsf{D}(t)$ to this end. See Sect. 9.2 for an example. In the framework of *hierarchical* specification [8], definedness is often implied by the presence of selector operations.

Kreowski [27] proposes that partial algebras can be simply obtains from *based* total algebras. The idea is to give first a base specification, whose initial algebra provides all the values of interest, for example, the usual natural numbers. Then one extends it to a specification with operations that may give errors, such as applying predecessor to zero or dividing by zero. The ordinary total initial algebra of the extended specification can then be made into a partial algebra by restricting operations to being defined when they return values of the base algebra, ignoring all the extra error values.

The basic framework of (first-order) partial algebras doesn't cater for non-strict operations, such as if-then-else. Astesiano and Cerioli [3] propose a framework of so-called *don't care algebras* that allows non-strict (monotonic) operations, and they make a revealing study of the relationship between total and partial algebras. However, most operations of ordinary abstract data types *are* strict, and when strictness is no longer implicit, it has to be specified explicitly by axioms, which might be tedious in practice. The relationship between total and partial algebras is further investigated in an abstract setting in [4].

Poigné [40] defines a framework that can be seen as a generalization of the standard partial algebra framework. He distinguishes between sorts and types: sorts are essentially syntactic, used in signatures for restricting the allows terms; types are semantic, with the classifiation of individual values into types being specified by axioms. The whole framework is based on Scott's logic of partiality. See also the description of Poigné's Typed Horn Logic in Sect. 7.5.

Now let us leave partial algebra, and look at some total algebra frameworks that deal with errors more efficiently than the original many-sorted algebra framework does.

5 Errors and Exceptions

Several proposals have been made for extending the basic framework of total many-sorted algebras to accommodate error values, aiming at better pragmatics. As well as the proposals considered in this section, also the order-sorted algebras considered in Sect. 6 cater for errors.

5.1 Error Algebras

The signatures of Goguen's *error algebras* [18] distinguish between *okay* and *error* operations. For each sort $s \in S$ the carrier set A_s is partitioned into okay values and error values. All operations are required to return an error value whenever any argument is an error value; error operations always return error values. Axioms are divided into okay equations and error equations. An okay equation $t_1 = t_2$ holds for all variable assignments such that both t_1 and t_2 have okay values; an error equation $t_1 = t_2$ holds whenever either t_1 or t_2 evaluates to an error.

However, this framework has the serious defect that when any operation in a specification has a *zero*, the strict treatment of errors conflicts with the zero axiom, causing all values (of the sort concerned, at least) to become error values! For instance, if err is an error value of sort nat, we have 0 = prod(0,err) = err ... . See Sect. 9.3 for an example of an error algebra specification where this problem doesn't arise.

5.2 Algebras with Okay Predicates

An alternative to dividing signatures into okay and error operations is to divide them into okay and *unsafe* operations as proposed by Gogolla, Drosten, Lipeck, and Ehrich [17], see also [15, 16]. Carriers are still divided into okay and error values, i.e., equipped with *okay predicates*. An okay operation always returns an okay value when all its arguments are okay values; an unsafe operation may or may not return an error value. Now all operations may return okay values, even when their arguments are error values, thus general error recovery and exception handling are possible—and inconsistency between zeros and error propagation can be avoided.

Variables for use in axioms are each declared to be okay or unsafe, and only okay values may be assigned to okay variables. As there is no implicit error propagation, one has to specify rather a lot of tedious axioms, unless one can accept the presence of a large number of distinct error values of each sort. See Sect. 9.4 for an example.

Another problem is that when the specified values are bounded, and exceeding the bounds is to give an error, the constructor operations are unsafe, and then the okay values of the initial model do not include the expected ones. (In fact one can use auxiliary sorts and operations to get around this problem, but the required specifications are too tedious for practical use.)

Essentially, it seems that this approach corresponds fairly closely to the way errors are treated using ordinal many-sorted algebras, but by keeping the okay predicates implicit and using two kinds of variables, the number of axioms required is kept down to an acceptable level. It can also be seen as a particular discipline in order-sorted algebraic specifications, called *stratification* [49].

5.3 Exception Algebras

The E, R *algebras* of Bidoit [7] provide not only implicit error propagation, but also specification of recovery cases, i.e., exception handling, which override propagation. But the interest of this framework is reduced by the fact that E,R algebraic specifications do not in general have initial models.

The *exception algebras* proposed by Bernet, Bidoit, and Choppy [5] develop the main ideas of E,R algebras, now providing a framework where specfications do have initial models. The aim is to cater not only for errors but also for general exception-handling.

An exception signature consists of an ordinary many-sorted signature, together with a set of exception *labels*, which correspond to a secondary classification of values, orthogonal to the sorts. An exception algebra has a family of sets of values, indexed by the exception labels together with a special label for okay values. Specification have to declare the so-called okay *forms*, which are used to determine the okay terms. This is more flexible than a mere division of operations into okay and error (or unsafe) operations; for instance, terms that evaluate to okay values of bounded structures can now be classified as okay.

The axioms are divided into okay axioms, labelling axioms, and generalized axioms. The okay axioms are positive conditional equations, to be satisfied only for assignments of okay *terms* to variables. An okay term may have the same value as a non-okay term, so in contrast to the exception labels, the okay label is attached to terms, rather than to values. For instance, when specifying bounded integers, max-int is an okay term but succ(max-int) is not, even though one specifies the equality of the values of these terms among the axioms.

The labelling axioms classify values using labels; a labelling may be conditional upon other labellings, and on equations. These labels are not automatically propagated by operations. The generalized axioms are equations (possibly conditional on labellings and equations) which specify exception-handling and error recovery.

The resulting framework seems extremely powerful, but specifications look somewhat intricate. See Sect. 9.5 for an example. Moreover, the axioms required to specify okay forms for bounded structures seem to require terms whose size is proportional to the bound!

5.4 Label Algebras

Bernot and Le Gall [6] further explore the idea of attaching labels to terms. They propose *label algebras*, where no distinction need be made between okay and exception labels: all labels are attached to terms rather than values, and used to restrict assignments to variables in axioms. They show examples where this extra generality is useful, in connection with exception-handling and with observability. Moreover, it is easy to specify that particular labels are determined by the values of the terms to which they are attached, whereupon they can be regarded as unary predicates on values, classifying them rather like conventional sorts do.

Label algebras are considered to be too low-level for direct use, in general, so we don't illustrate their specification here. Higher-level frameworks that correspond to particular disciplines of labelling can be defined by translation into label algebras. For instance, Bernot and Le Gall define a generalization (and simplification) of the exception algebras mentioned above in this way. However, the notion of satisfaction of axioms by label algebras involve consideration of term algebras freely generated by sets of values, which may seem a bit complicated. Label algebras do not provide an institution, at least not straightforwardly, it seems.

6 Sort Inclusions

Suppose that we have a collect ion of individual values, to be classified into sorts. There is no reason why each individual shoed necessarily be classified uniquely! For instance, the number one could be classified not only as a natural number, but also as a positive number, an integer, a rational number, etc. Moreover, some sorts are naturally sen as being *subsorts* of other sorts: we expect the natural numbers to be included in the integers, etc.

Technically, many-sorted algebras already allow the sort-indexed carrier sets to overlap, so that an individual value may in fact be classified as of more than one sort. But it is not possible to *specify* that such properties must hold, since an equation always relates individuals of the same sort (and no assumptions can be made about any relationship between the values returned by overloaded operation symbols). Any many-sorted algebraic specification may thus have models where carriers overlap, and models where they don't. Multiple classification of an individual here is merely accidental. However, there are technical and pragmatic reasons for allowing it: so that a single model value may *implement* several unrelated abstract data, for instance.

Now, the extensions of sorts are sets, and sets are partially-ordered by inclusion so it is natural to allow the specification of a partial order $\leq$ on the set of sorts S. (A pre-order might correspond better to the intensional nature of sorts. In practice, it doesn't usually matter whether mutually-included sorts are regarded as equal or not.) Signatures with partially-ordered sorts are called *order-sorted*, as are algebras with such signatures. Order-sorted algebras should not to be confused with *partially-ordered algebras* [33] where for each s in an ordinary *set* of sorts, the carrier set A_s is equipped with a partial order.

Order-sorted algebras support *subsort polymorphism* : when an operation is available both on some sort and on a subsort, both versions of it necessarily give the same result when applied to a value of the subsort. The partial order on sorts also supports an economical treatment of errors and exception-handling, allowing error supersorts as well as restrictions of partial operations to subsorts on which they are total.

For instance, with rational numbers of sort rat we may introduce extra values to represent errors such as trying to divide by zero, including them in a supersort, say errata, of rat. Then we may *extend* all the numerical operations from rat to

errata, specifying that they give error values when applied to error values. The simplest way to do this is perhaps to treat erroneous applications of operations themselves as error values, not specifying any equations for them at all.

Alternatively, we may *restrict* division to applications where the second argument is in the subsort of nonzero rationals, forbidding the user to write terms that involve division by zero when evaluated. Here, however, there is the problem that it is usually not *syntactically* apparent whether an argument term might evaluate to zero or not, just from the functionalities of the operations in it: addition maps a positive and a negative integer to an arbitrary integer, possibly zero, for example. We shall return to this problem shortly.

There are two main approaches to order-sorted algebras, one of them due to Goguen together with Meseguer, the other developed by Gogolla, Poigné, and Smolka. The two approaches differ in what at first might appear to be a trifling technical detail, but which on closer scrutiny turns out to be a major conceptual disagreement about the nature of subsorts, as explained below.

6.1 Overloaded Order-Sorted Algebras

Goguen [19] was the first to propose a framework for order-sorted algebras (OSA). Subsequently, together with Meseguer, he developed the framework in a series of papers, culminating so far in the first part of a definitive presentation [23]. This development was closely linked to that of the OBJ system [25], which implements OSA specifications.

The formal details of OSA signatures and algebras are a bit more burdensome than in the original many-sorted algebraic framework. The sorts of an order-sorted signature are equipped with a partial order $\leq$, and the functionalities of operations must be *monotonic*: when $f : s_1, \ldots, s_n \to s$, $f : s'_1, \ldots, s'_n \to s'$, and all $s_i \leq s'_i$ then $s \leq s'$. For simplicity, signatures are sometimes required to be *regular*, a condition which guarantees that every term has a least sort; a weaker condition that guarantees this is *preregularity* [23, Section 5.1].

An order-sorted algebra A is here a many-sorted algebra such that $s \leq s'$ implies $A_s \leq A_{s'}$ and moreover:

(*) when $f : s_1, \ldots, s_n \to s$, $f : s'_1, \ldots, s'_n \to s'$, and all $s_i \leq s'_i$ (hence also $s \leq s'$) then the restriction of $f_A : A_{s'_1} \times \cdots \times A_{s'_n} \to A_{s'}$ to $A_{s_1} \times \cdots \times A_{s_n}$ is the same function as $f_A : A_{s_1} \times \cdots \times A_{s_n} \to A_s$.

It is important to realize that when the sorts s_i and s'_i are *not* related by inclusion, the two functions denoted by f need not be related at all, and may return different results when applied to the same argument values! Goguen and Meseguer [23] argue for keeping this feature, which is known as *overloading*, or *ad hoc* polymorphism, for they want MSA to be a special case of OSA, obtainable merely by letting the inclusion order $\leq$ be the identity relation. But this seems to conflict with a basic intuition underlying the subsort relation: that sorts represent classification of a single collection of individuals. We shall return to this discussion in the next subsection, when we look at an alternative approach

to OSA.

Axioms of OSA specifications are similar to those of MSA specifications, except that the (least) sorts of terms in equations need not be the same—although they have to be *connected* by inclusions. *Sort constraints* are allowed, for use in specifying bonded structures; these correspond to conditional functionalities, but are axioms, rather than being part of signatures.

Not all terms that ought to denote values are allowed. For example, consider the standard specification of the sort nat of natural numbers with successor operation $\mathsf{succ} : \mathsf{nat} \to \mathsf{pos}$ and a partial predecessor operation $\mathsf{pred} : \mathsf{pos} \to \mathsf{nat}$, where $\mathsf{pos} \leq \mathsf{nat}$ is the subsort of positive integers: the term $\mathsf{pred}(\mathsf{pred}(\mathsf{succ}(\mathsf{succ}(0))))$ is not allowed! But with $\mathsf{pred}(\mathsf{succ}(n)) = n$ as an obvious axiom for $n : \mathsf{nat}$, we expect $\mathsf{succ}(0) = \mathsf{pred}(\mathsf{succ}(\mathsf{succ}(0)))$, hence $0 = \mathsf{pred}(\mathsf{succ}(0)) = \mathsf{pred}(\mathsf{pred}(\mathsf{succ}(\mathsf{succ}(0))))$. Thus allowed terms can be demonstrably equal to forbidden ones, which seems somewhat anomalous. This is essentially the same problem that arises with trying to forbid terms involving division by zero: the functionalities of operations are not sufficiently precise. Here we should have not only $\mathsf{succ} : \mathsf{nat} \to \mathsf{pos}$ but also $\mathsf{succ} : \mathsf{pos} \to \mathsf{pos2}$, where pos2 is the sort of all integers from 2 upwards, and so on *ad infinitum*.

Goguen and Meseguer [23, Section 3.3] propose inserting *retracts* in forbidden terms to give them 'the benefit of the doubt at parse time'. Here a retract maps a sort to a subsort, being identity on the values already in the subsort. For instance the insertion of the retract $\mathsf{r} : \mathsf{nat} \to \mathsf{pos}$ allows $\mathsf{pred}(\mathsf{r}(\mathsf{pred}(\mathsf{succ}(\mathsf{succ}(0)))))$ to be well-formed. The trouble is that although such retracts have a simple *operational* semantics in this framework, they are essentially *partial* operations. A term such as $\mathsf{r}(0)$ should surely not have a value of sort pos! Nevertheless, the insertion of retracts doesn't interfere with equality: it provides a so-called conservative extension—although only with respect to direct consequence, not the inductive consequences that hold in the initial model.

To give an *algebraic* interpretation of retracts, Goguen and Meseguer consider the specifications obtained by adding all possible retract operations, together with their defining equations, to order-sorted specifications. The initial algebras of the original specifications get homomorphically injected into those of the extended specifications. But now the *semantics* of each order-sorted specification has been changed, and the notion of such a specification being *correct* with respect to some intended model should presumably be redefined.

Despite the mentioned anomalies, the overloaded OSA framework succeeds in eliminating many of the pragmatic deficiencies of the MSA framework. See Sect. 9.6 for an example specification. Let us leave it at that, and turn to an alternative approach to order-sorted algebras.

6.2 Universal Order-Sorted Algebras

Rather than insisting that OSA be a true generalization of the traditional framework of MSA, one can take the view that the really essential notion is that of a *universe* of individuals, with each operation symbol identifying a *single* (par-

tial) operation on that universe, and with sorts corresponding to subsets of the universe.

This approach to OSA was developed by Gogolla [14], Poigné [39], and Smolka [47], see also [49]. Essentially, the difference from the overloaded OSA framework described in the previous section is that now the carrier sets are united into a single universe, and condition (*) is replaced by the stronger:

(**) when $f : s_1, \ldots, s_n \to s$ and $f : s'_1, \ldots, s'_n \to s'$, then the retrictions of the functions $f_A : A_{s_1} \times \cdots \times A_{s_n} \to A_s$ and $f_A : A_{s'_1} \times \cdots \times A_{s'_n}$ to the intersections $(A_{s_1} \cap A_{s'_1}) \times \cdots \times (A_{s_n} \cap A_{s'_n})$ are the same.

Notice that this condition is entriely independent of the sort inclusion relation. It can be understood as a condition for *sort-independent semantics*: a legal term should always have the same value, regardless of what sorts are ascribed to its subterms. Let us refer to this variant as *universal* order-sorted algebras. (Waldmann [51] calls them 'non-overloaded' algebras, since each operation symbol is interpreted as a single function; but that conflicts somewhat with the conventional understanding of what overloading is.)

Goguen and Meseguer [23, Section 5.2] mount an attack on universal OSA, claiming that it has 'serious drawbacks'; but their arguments do not seem terribly convincing. For instance, they want to allow an algebra in which 0 and 1 are both Booleans and naturals, and in which + is both addition and exclusive or of Booleans'. It seems that 0 and 1 here refer to the standard natural numbers, as the argument does not necessarily involve overloaded constants (which prevent regularity). Both the overloaded and universal OSA frameworks allow such values to occur in the carrier sets of *unrelated* sorts. The only disagreement here is about whether the symbol + can be overloaded so that $x + x$ returns 0 when x denotes 1 of sort Boolean, but 2 when x denotes 1 of sort natural; overloaded OSA allows this model, whereas universal OSA doesn't. The statement that order-sorted logic *must* be 'in principle a refinement of many-sorted logic' [23, Section 5.2] seems to be somewhat dogmatic.

Despite the conceptual and technical differences between the treatment of subsorts in overloaded OSA and universal OSA, for most purposes they can be used interchangeably. For instance, the example in Sect. 9.6 serves just as well for universa1 OSA as for overloaded OSA. However, universal and overloaded OSA do *not* provide the same notion of satisfaction. For example, suppose we have constants $\mathsf{a} : \mathsf{A}$, $\mathsf{b} : \mathsf{B}$ where A and B are subsorts of C, and let us specify $\mathsf{a} = \mathsf{b}$. If we have an overloaded operation f with $\mathsf{f} : \mathsf{A} \to \mathsf{D}$ and $\mathsf{f} : \mathsf{B} \to \mathsf{D}$, universal OSA modes always satisfy $\mathsf{f}(\mathsf{a}) = \mathsf{f}(\mathsf{b})$, but overloaded OSA ones don't!

For an analysis of numerous technical points concerning overloaded and universal OSA, see the forthcoming paper by Waldmann [51]. Goguen [20] proposes a *hidden-sorted* version of OSA where the values with hidden sorts represent internal states of *objects*. He also shows there how to define an institution for *behavioural* satisfaction.

6.3 Generalized Order-Sorted Algebras

Poigné [42] proposes a generalized framework for OSA, which includes universal OSA as a special case. The main idea is to use a *set* of partial orders on sorts, rather than just a single one. Sorts that occur in different partial orders are treated as in overloaded OSA, whereas those that occur in the same partial order are treated as in universal OSA.

The proposed framework does *not* fully include overloaded OSA as a special case: there are pathological examples of overloaded OSA models that are not generalized OSA models, for instance with a supersort for two otherwise unrelated sorts that have some nonexpressible value in common. However, this framework does successfully generalize both MSA and universal OSA.

6.4 Inclusions and Subtypes

Martí-Oliet and Meseguer [30] make a useful analysis of the difference between the notions of subtype as *inclusion* (as in OSA) and subtype as *coercion*. They conclude that as well as the inclusion partial order $\leq$ on sorts, one should as well consider a *generalized subtype* relation $\leq:$ containing $\leq$ as a subrelation, where $s \leq: s'$ holds when there is an implicit coercion from s to s'. They only require $\leq:$ to be a preorder. For instance, with Cartesian and polar coordinates, we might have coercions both ways.

Qian [43] has also developed an interesting framework catering for coercions, which are more general than subsort inclusions.

6.5 Generator Induction

Owe and Dahl [38] propose some restrictions on order-sorted algebras, so as to cater for generator inductive function definitions. They prohibit (incidental) overloading, and insist that minimal sorts denote disjoint sets of values; this allows Sinatra to be completed with all unions and intersections, and similarly for functionalities.

The emphasis of this work is on a particular style of functional programming, and the authors argue *against* the specification of general algebraic axioms.

7 Classified and Unified Algebras

The following approaches treat sorts *semantically*, using axioms to specify classification.

7.1 Classified Algebras

Perhaps the most *logical* way of classifying individuals into sorts is to let a sort be a unary predicate on the universe, i.e., a subset of it. This technique is well-known from first-order logic, under the name *relativization*. It was first proposed for use in algebraic specifications by Wadge [50] who defined a framework called

classified algebras—not to be confused with the framework of the same name later proposed by Poigné [40].

Essentially, a signature of a classified algebra is a pair (F, S) where F is a ranked[2] alphabet of operation symbols and S is an alphabet of sort symbols, including the distinguished symbol anything. Axioms are equations $t_1 = t_2$, and so-called declarations written $t \ : \ s$, where t is a term and $s \in S$. Variables in terms are written with sort symbols $s \in S$ as subscripts, and assignments to them are restricted to values that are classified as being of the specified sorts.

A classified algebra A has a nonempty universe, a *total* function on the universe for each $f \in F$, and a subset of the universe for each $s \in S$; the subset for anything is the entire universe. A declaration $t \ : \ s$ holds in a particular algebra if the value of the term t is an element of the set denoted by s, for all assignments to the variables in t of values from the sets denoted by their sorts. The satisfaction of equations by algebras is as usual, bearing in mind the same restrictions on assignments as for declarations.

It is easy to see that classified algebras can be regarded as a special case of unsorted Horn clause (with equality) specifications: sort symbols correspond to unary predicate symbols, declarations are just formulae that apply the predicates. Sort restrict ions on variables X_s correspond to hypotheses $X \ : \ s$ with unsorted variably. Thus we should expect classified algebra specifications to have initial models, as they do indeed.

Classified algebras combine technical simplicity with high expressiveness. For instance, partiality and errors can represented by operations returning values that are not classified by any sort—although one can also classify error values, if desired. There are usually lots of these error values, but they don't get in the way, because variables are automatically restricted to non-error values of particular sorts. Functionalities can be early concisely as declarations, i.e., axioms; so can sort inclusions. Overloading and subsort polymorphism are natural, since each operation symbol stands for a single operation on the entire classified universe. See Sect. 9.7 for an example.

Of course, there is a price to pay for such simplicity: sort checking is undecidable! This is because classification is essentially *semantic*, in contrast to the *syntactic* notion of sort in the frameworks we considered earlier. Whether $t \ : \ s$ holds (for a ground term t) depends in general not only on declarations concerning the symbol in t, but also on equations that relate t to other terms. Anyway, one could easily define restrictions on classified specifications to make sort checking decidable. Alternatively, it might be acceptable to use interactive theorem proving for sort checking in general classified specifications (by analogy with the Nuprl system [11]).

Wadge sketches some ideas for generalizing classified algebras, to allow user-specified operations on sorts, and classifications of sorts. But this seems to be getting into the realm of higher-order logic, and it is unclear whether initial models of such specifications word always exist. Wadge's paper makes refreshing reading, and it infuenced several other approaches.

[2]Wadge left alphabets unranked, for no apparent reason!

7.2 Galactic Algebras

The framework of *galactic algebras*, or *G-algebras*, proposed by Mégrelis [31, 32] is essentially a partial algebra variant of the classified algebras discussed in the preceding section. A signature consists of a set of sort symbols S and a ranked alphabet of operation symbols F. The special sort symbol '***' corresponds to 'anything' in classified algebras. Axioms are equations $t_1 = t_2$, membership formulae $t : s$ where $s \in S$, and domain (defining) formulae $\$t$. Moreover, each variable x is globally restricted to a prices sort by a confinement axiom $x :: s$.

A G-algebra A is like a classified algebra, having a nonempty universe and a subset of it for each sort symbol. Each operation symbol is here interpreted as a *partial* function on the universe of A, as are all terms (relative to a fixed ordering of the variables). Equations and membership formula are interpreted *existentially*, for example $t : s$ can only hold when t denotes a *total* function on the subsets of the universe of A indicated by the sorts of variables occurring in t.

Functionalities and subsort inclusions are provided as axioms, abbreviating particular sets of formulae. See Sect. 9.8 for an example of a G-algebra specification exploiting such axioms (written with a less Spartan notation than that used by Mégrelis) .

C. and H. Kirchner [26] find that the framework of G-algebra is particularly useful for order-sorted term rewriting. The basic idea is to compute sorts of terms and deduce equalities simultaneously, decorating terms with their currently-proved sorts. This allows the relaxation of some rather severe condition that have to be imposed in OSA frameworks to obtain completeness of rewriting with respect to deduction. See all [10, 51].

7.3 Polymorphically Order-Sorted Types

Smolka has developed a framework for logic programming over *polymorphically order-sorted types* [48]. It is not directly relevant to algebraic specifications, because value operations are always free constructors, and cannot be relate by equations. Thus natural numbers with zero and successor can be specified, but not together with the usual predecessor operation. Nevertheless, the treatment of sorts is particularly interesting, so let us look at it briefly.

The essential idea is to define *sort* constants and operations by equations. (Then relations on values are defined by declarations of the sorts of their arguments together with some definite clauses.) For example, the sort constant nat is equated to the union of the sorts zero and pos. The value constant 0 constructs the only value of sort zero. The value operation succ constructs values of sort pos from values of sort nat. Of course, when a sort is equated to a union of two other sorts, the latter are thereby subsorts of the former.

This framework provides support for parametric polymorphism in a way that is arguably superior to that of order-sorted algebra. The idea is to define sort operations in the same way as sort constants, using equations and sort union, with sort variables that range over all the specified sorts. For instance, one may

specify a sort operation list(T), thus providing list(pos) as a subsort of list(nat), as well as paticular sorts of nested lists such as list(list(nat)). Sort operations are always monotonic with respect to sort inclusion. See Sect. 9.9 for the full specification of lists (which is possible only because no value equations are required). Notice that the treatment of parametric polymorphism is achieved within the ordinary specification logic, rather than by using some meta-logic of module instantiation.

7.4 Equational Type Logic

Manca, Salibra, and Scollo [28, 29] propose *Equational Type Logic* (ETL), which is another framework where sorts are treated semantically. Like conventional frameworks, it caters for ordinary (conditional) equational algebraic specifications of individuals and operations upon them. It resembles classified algebra in the way that individuals can be classified into sorts, but now (as in Smolka's approach in the preceding section) sorts are values, not predicates, and operations on sorts can be specified. The result is a rather simple, elegant, and expressive framework, coping well with partiality and polymorphism. Let's look at the formal details.

Algebras in ETL are called *type algebras*. The signature of a type algebra is an unsorted, ranked alphabet of operation symbols. Thus there are no restrictions on term formation. A type algebra A is simply a conventional (total) unsorted algebra together with an arbitrary binary 'typing' relation $:_A$ on its carrier.

Axioms in specifications are simply Horn clauses involving equations $t_1 = t_2$ and/or sort assignments $t_1 \ : \ t_2$. Variables in axioms are unsorted, ranging over the entire universe of a type algebra. But the effect of sorted variables can easily be obtained, using conditions of the form $x \ : \ t$, which only hold when the value of x is an individual of sort t (an abitrary term, not merely a constant symbol). Specifications always have initial models.

Values that are not in the typing relation (on either side) are *underdefined*, and may be viewed as error values. Such an implicit specification of errors is usually very concise, but errors are not automatically propagated by operations, in general—except in initial models, under the assumption that all variables are restricted by sort assignments in conditions. Moreover, with implicit error specification, checking whether an axbitraxy (ground) term denotes an error or not is undecidable. Alternatively, one can explicitly classify error values, if desired, as in classified algebras. Automatic restriction of variables to non-error values is not available, however; such restrictions are to be specified explicitly by sort assignments in conditions.

Although sort inclusion is not provided as a relation, it can be represented straightforwardly by specifying a general sort union operation $\cup$ and using $x \cup y = y$ to express that x is a subsort of y. But sorts that happen to have the same extension are not necessarily equal, so some care is needed. (An alternative representation of sort inclusion directly as an operation is given in [46].) In contrast to Smolka's approach above, and to unified algebras below, sort operations are not necessarily monotonic with respect to sort inclusion, so if

one wants list(pos) to be a subsort of list(nat), this has to be specified explicitly. Thus subsort polymorphism is possible, but not guaranteed.

The functionality of an ordinary operation f that is total on individuals of sorts $s_1, \ldots, s_n$ can be expressed by a clause $x_1 : s_1, \ldots, x_n : s_n \Rightarrow f(x_1, \ldots, x_n) : s$, and overloading is obviously allowed. But note that such functionality clauses are not required at all! Parametric polymorphism can be specified, much as in Smolka's approach. See Sect. 9.10 for an example.

Finally, notice that the typing relation can be used not only to classify individual into sorts, but also sorts into meta-sorts, i.e., kinds. However, it doesn't seem that this feature is neded much in the practical examples of ETL specifications seen so far.

7.5 Typed Horn Logic

Poigné [41] proposes a further framework involving Horn clauses and typing relations. Here the primitive formulae are: $x : t$, asserting that x exists and is of type t; $t :: k$, asserting that t exists and is of 'order' (i.e., *kind*) k; and k ! , asserting merely that k in an existing order. No syntactic distinction is made between operations on individual, types, and orders. Equality is treated existentially, as a partial equivalence relation. Models are based on Scott's theory of partiality. Specifications resemble those in ETL above; see Sect. 9.11 for an example.

7.6 Term Declaration Logic

Aczel's *Term Declaration Logic* (TDL) [1] appears to be related to ETL and Typed Horn Logic. Here, a *pre-signature* consists of pairwise disjoints sets of sort symbols, operation symbols and variables. A signature consists of a presignature together with restrictions of its variables to particular sorts and *declarations*. A *formation* declaration is written $\tau \downarrow$, and a *sorting* declaration is written $\tau : s$, where τ is a *sort* term and s is a sort symbol. A subsort inclusion is declared using $s' : s$, and a functionality is written $f(s_1, \ldots, s_n) : s$. A pre-term t constructed from variables and operation symbols is deemed to be a *term* when $\sigma t \downarrow$ can be proved using some simple inference rules, where σt is the sort term obtained by replacing variables in t by the sorts to which they are restricted. Similarly, t is of sort s if $t : s$ can be proved.

A specification consists of a signature together with a set of equations between terms. A *pre-algebra* provides a universe of individuals, a subset of it for each sort symbol, and a partial function on it for each opration symbol. An *algebra* is (roughly) a pre-algebra where the domain of definition of each partial function corresponds to the argument sorts in some provable functionality for the function. See Sect. 9.12 for the usual example.

7.7 Unified Algebras

Let us conclude our survey of first-order frameworks for algebraic specification with so-called *unified algebras* [36, 34, 35]. I developed this framework for use in specifying data for action semantic descriptions of programming languages [37], but it might have more general applicability. The starting point was order-sorted algebras; the foregoing work of Wadge on classified algebras provided much inspiration, as did Smolka's work on the semantics of order-sorted Horn logic [47]. Unined algebras has much in common with ETL (above), although the initial developments of the two approaches were independent.

The signature of a unified algebra is just a ranked alphabet. The universe of a unified algebra A is a (distributive) lattice with a bottom value, together with a distinguished subset I_A of *individuals*. The operations of a unified algebra are required to be monotone (total) functions on the lattice; they are *not* required to be strict or additive, nor to preserve the property of individuality.

All the values of a unified algebra may be thought of as sorts, with the individual corresponding to singleton sorts. However, the individuals do *not* have to be the atoms of the lattice, just above the bottom: for instance, the meet of two individuals is below both of them, but need not be identified with the bottom value. The partial order of the lattice $\leq$ represents sort inclusion; join $x \mid y$ is sort union and meet $x \,\&\, y$ is sort intersection. Those values that do not include any individuals at all, such as the bottom value denote by the constant 'nothing', are vacuous sorts, representing the lack of a result from applying an operation to unintended arguments. A special case of a unified algebra is a *power algebra*, whose universe is a power set, with the singletons as individual [35].

The axioms of unified algebraic specifications are Horn clauses involving equations $t_1 = t_2$, inclusions $t_1 \leq t_2$, and *individual* inclusions $t_1 : t_2$. An equation holds in a unified algebra A when the terms have identical value, whether or not these values are individuals, proper sorts, or vacuous; an inclusion holds when the values of the terms are in the partial order of the sort lattice; and an individual inclusion $t_1 : t_2$ holds when the value of t_1 is not only included in that of t_2, but also in the distinguished subset of individuals I_A. See Sect. 9.13 for an example specification. Note that the example exploits some natural abbreviations for axioms that correspond to functionalities of (total, partial, or unrestricted) operations; the expansions of these formal abbreviation are defined in [37].

Unified algebraic specifications always have initial models, became they are essentially just unsorted Horn clause logic (with equality) specifications: the lattice structure and monotonicity of operations can all be captured by Horn claim. One reason for not restricting attention to the power algebras mentioned above is that even very simple specifications fail to have initial models. A similar point is made by Smolka [48].

Although it can be shown that unified algebras provide a *liberal* institution, with the dual notion of reduct functor, it is problematic to define useful constraints in such an unsorted framework, because the ordinary reduct functor

only forgets operations—never values. However, by using a *more forgetful* reduct functor (treating a ground terms as if they were sorts) one can simulate the way that many-sorted and order-sorted forgetful functors deal with values, thereby providing constraints that have the expected meaning.

The main virtues of unined algebras are, in my own view, as follows:

- It is easy to express conventional (universal) OSA specifications: functionalities and subsort inclusions in order-sorted signatures are simply expressed as axioms in unified algebraic specifications.
- Partial operations are represented semantically by total operations that may return vacuous sorts. The undefinedness of a value can be specified by equating it to the constant denoting the bottom value. The bottom value is included in every sort, and allows error *individuals* to be avoided.
- The monotonic extension of operations from individuals to proper sorts is useful for specifying sort equations, such as $\mathsf{nat} = \mathsf{0} \mid \mathsf{succ}(\mathsf{nat})$. There is no distinction between an individual such as $\mathsf{0}$ and the sort that includes just that individual (which is possible because sorts of sorts are not needed).
- Parametric polymorphism and generic data types can be easily specified, using sort restriction operations. For instance, consider a binary operation $L[\mathsf{of}\ D]$ which restricts the sort of lists L to those lists whose components are of data sort D. Notice that monotonicity gives $\mathsf{list}[\mathsf{of}\ \mathsf{0}] \leq \mathsf{list}[\mathsf{of}\ \mathsf{nat}] \leq \mathsf{list}[\mathsf{of}\ \mathsf{int}]$, assuming $\mathsf{nat} \leq \mathsf{int}$.
- Dependent sorts can be specified too, for instance using an operation that maps each individual natural number to the sort of lists with length at most that number. The fact that individuals are a special case of sorts avoids some pedantic details.
- Instantiation of generic specifications can be achieved by specifying sort equations as axioms, instead of having to use translation.

Perhaps there are some drawbacks too? Well, sorts are nonextensional, so unspecified values do not automatically get equated with the bottom value: a careless specification may give rise to a large number of distinct vacuous sorts, all representing errors. Although these don't get in the way at all, they are there. Sort checking is obviously undecidable in general—as for most systems that allow dependent sorts. Variables that range over all sorts, or over all subsorts of a specified sort, easily give rise to inconsistency between obvious-looking axioms: proper sorts correspond to nondeterministic choices between individuals, and it is well-known that extra care is needed with specifying nondeterministic operations. The assumption of monotonicity (useful for defining constraints) prevents a straightforward extension to higher-order unified algebras with function space construction as an ordinary operation. Finally, when a many-sorted

or order-sorted specification is translated straightforwardly into a unified algebraic specification, the loose semantics of the specifications are quite different, although their initial algebras are closely related.

8 Conclusion

A tentative conclusion might be that frameworks which cater for sort inclusions have superseded those that don't. In particular, it seems that error algebras and algebras with okay predicates can be regarded as particular disciplines of order-sorted algebras. Label algebras provide a rather different way of generalizing many-sorted algebras.

The tendency is perhaps also to move from a syntactic notion of sort to a semantic one, abandoning decidability of sort checking in favour of allowing an expressive algebra of sorts. Perhaps order-sorted algebras themselves are best regarded as particular, efficiently-implementable disciplines of the essentially unsorted frameworks of ETL [29] or unified algebra [35]? Further comparison of both the theoretical and pragmatic aspects of these frameworks is required before any definite conclusions can be drawn.

But however interesting sorts in algebraic specifications have become, remember: they are only there to help specifying individuals and their operations!

Acknowledgments: Many of the participants at the 8th WADT meeting in Dourdan contributed to this survey by pointing out inaccuracies in the preliminary version, and telling me about serious omissions that I had made. Răzvan Diaconescu kindly let me see a draft of his analysis of the relationship between various frameworks that support subsorts. Joseph Goguen patiently explained numerous technical and conceptual points that I had misunderstood concerning OSA. Axel Poigné gave me some useful comments, specially concerning the relationship between total and partial algebras. Thanks to Christine Choppy and Michel Bidoit for encouraging me to write this survey, and for inviting me to present it at the 8th WADT.

9 Examples

The examples given below illustrate most of the approaches that we have discussed above. For simplicity, the specified abstract data type in each case is merely *genetic lists of data*.[3] It is supposed to be erroneous to attempt to take the head or tail of an empty-list. We don't bother to specify module parameterization.

To ease the comparison of the examples, we use a uniform style of notation throughout. This often differs in appearance from that advocate and led by the authors of the illustrated approaches.

[3] To make at least a notational change from stacks!

Some of the examples assume that the Boolean truth-values and operations are already specified appropriately.

9.1 Many-Sorted Algebras

sorts: data, list.
ops: nil : $\rightarrow$ list,
cons : data, list $\rightarrow$ list,
head : list $\rightarrow$ data,
tail : list $\rightarrow$ list,
e_{data} : $\rightarrow$ data,
e_{list} : $\rightarrow$ list,
ifok : data, list, data $\rightarrow$ data,
ifok : data, list, list $\rightarrow$ list,
ok : data $\rightarrow$ boole,
ok : list $\rightarrow$ boole.
vars: d, d': data, l, l': list.

axioms:
(1) ifok($d,\ l$, head(cons(d, l))) $= d$.
(2) ifok($d,\ l$, tail(cons(d, l))) $= l$.
(3) head(nil) $= e_{data}$.
(4) tail(nil) $= e_{list}$.
(5) ok(e_{data}) $=$ false.
(6) ok(e_{list}) $=$ false.
(7) ok(nil) $=$ true.
(8) ok(cons(d, l)) $=$ ok(d) $\wedge$ ok(l).
(9) cons(e_{data}, l) $= e_{list}$.
(10) cons(d, e_{list}) $= e_{lidt}$.
(11) head(e_{list}) $= e_{data}$.
(12) tail(e_{list}) $= e_{list}$.
(13) ifok(d, l, d') $=$
ife(ok(d) $\wedge$ ok(l), d', e_{data}).
(14) ifok(d, l, l') $=$
ife(ok(d) $\wedge$ ok(l), l', e_{list}).

9.2 Partial Algebras

sorts: data, list.
ops: nil : $\rightarrow$ list,
cons : data, list $\rightarrow$ list,
head : list $\rightarrow$ data,
tail : list $\rightarrow$ list,
vars: d: data, l: list.

axioms:
(1) head(cons($d,\ l$)) $= d$.
(2) tail(cons($d,\ l$)) $= l$.
(3) D(nil).

9.3 Error Algebras

sorts: data, list.
ops: nil : $\rightarrow$ list,
cons : data, list $\rightarrow$ list,
head : list $\rightarrow$ data,
tail : list $\rightarrow$ list,
err-ops: e_{data} : $\rightarrow$ data,
e_{list} : $\rightarrow$ list.
vars: d : data, l : list.

ok-axioms:
(1) head(cons($d,\ l$)) $= d$.
(2) tail(cons($d,\ l$)) $= l$.

err-axioms:
(3) head((nil) $= e_{data}$.
(4) tail((nil) $= e_{list}$.

9.4 Algebras with Okay Predicates

sorts: data, list.
ops: nil : $\rightarrow$ list,
cons : data, list $\rightarrow$ list,
head : list $\rightarrow$ data($unsafe$),
tail : list $\rightarrow$ list($unsafe$),
e_{data} : $\rightarrow$ data($unsafe$),
e_{list} : $\rightarrow$ list($unsafe$),
vars: d : data, l : list.
d' : data, l' : list($unsafe$).

axioms:

(1) head(cons(d, l)) = d.
(2) tail(cons(d, l)) = l.
(3) head(nil) = e_{data}.
(4) tail(nil) = e_{list}.
(5) cons(e_{data}, l') = e_{list}.
(6) cons(d', e_{list}) = e_{list}.
(7) head(e_{list}) = e_{data}.
(8) tail(e_{list}) = e_{list}.

9.5 Exception Algebras

sorts: data, list.
ops: nil : $\rightarrow$ list,
cons : data, list $\rightarrow$ list,
head : list $\rightarrow$ data,
tail : list $\rightarrow$ list,

labels: e_{data}, e_{list}.

ok-forms:
(1) nil $\in$ Ok-Frm.
(2) $d \in$ Ok-Frm $\wedge$ $l \in$ Ok-Frm $\Rightarrow$
cons(d, l) $\in$ Ok-Frm.

ok-axioms:
(3) head((cons(d, l)) = d.
(4) tail(cons(d, l)) = l.

label-axioms:
(5) head(nil)$\in$ e_{data}.
(6) tail(nil)$\in$ e_{list}.
(7) $d \in e_{data} \Rightarrow$ cons(d, l) $\in e_{list}$.
(8) $l \in e_{list} \Rightarrow$ cons(d, l) $\in e_{list}$.
(9) $l \in e_{list} \Rightarrow$ head(l) $\in e_{data}$.
(10) $l \in e_{list} \Rightarrow$ tail(l) $\in e_{list}$.

general-axioms: *none*

9.6 Order-Sorted Algebras

sorts: data, list, nelist.

subsorts: nelist $\leq$ nelist.

ops: nil : $\rightarrow$ list,
cons : data, list $\rightarrow$ nelist,
head : nelist $\rightarrow$ data,
tail : nelist $\rightarrow$ list,

vars: d : data, l : list.

axioms:
(1) head((cons(d, l)) = d.
(2) tail(cons(d, l)) = l.

Using supersorts instead of subsorts:

sorts: data, list, err-data, err-list.

subsorts: data $\leq$ err-data.
list $\leq$ err-list.

ops: nil : $\rightarrow$ list,
cons : data, list $\rightarrow$ list,
head : list $\rightarrow$ err-data,
tail : list $\rightarrow$ err-list,

vars: d : data, l : list.

axioms:
(1) head((cons(d, l)) = d.
(2) tail(cons(d, l)) = l.

9.7 Classified Algebras

sorts: data, list.

ops: nil, cons, head, tail.

axioms:
(1) nil : list.
(2) cons(d_{data}, l_{list}) : list.
(3) head(cons(d_{data}, l_{list})) = d_{data}.
(4) tail(cons(d_{data}, l_{list})) = d_{list}.

Optionally, one may add:

sorts: nelist.

axioms:

(5) cons($d_{\text{data}}, l_{\text{list}}$) : nelist.

(6) head(l_{nelist}) : data.

(4) tail(l_{nelist}) : list.

9.8 G-Algebras

sorts: data, list.

ops: nil, cons(_,_), head(_), tail(_).

vars: d, l.

axioms:

(1) d :: data. l :: list.

(2) nil : list.

(3) cons(_,_) : data, list $\rightarrow$ nelist.

(4) head(_) : nelist $\rightarrow$ data.

(5) tail(_) : nelist $\rightarrow$ list.

(6) head(cons)$d, l)) = d$.

(7) tail(cons)$d, l)) = l$.

9.9 Polymorphically Order-Sorted Types

(1) list(T) := elist $\sqcup$ nelist(T).

(2) elist := elist $\sqcup$ nelist.

(3) elist := nil : [ ].

(4) nelist(T) := cons: $T\times$ list(T).

9.10 Equational Type Logic

(1) nil : list.

(2) d : data, l : list $\Rightarrow$ cons(d, l) : list.

(3) d : data, l : list $\Rightarrow$.
head(cons(d, l)) = d.

(4) d : data, l : list $\Rightarrow$.
tail(cons(d, l)) = l.

9.11 Typed Horn Logic

orders: type.

types: list(_), list+(_), type.

ops: nil, cons(_,_), head(_), tail(_).

decl:

(1) $\vdash$ type !.

(2) t :: type $\vdash$ list(t) : type.

(3) t :: type $\vdash$ list + (t) : type.

(4) t :: type $\vdash$ nil : list(t).

(5) t :: type, l : list(t), $d : t \vdash$
cons(d, l) : list + (t).

(6) t :: type, l : list + (t), $\vdash$ head(l) : list(t).

(7) t :: type, l : list + (t), $\vdash$ tail(l) : list(t).

axioms:

(8) t :: type l : list(t), d : $t \vdash$.
head(cons(d, l)) = d.

(9) t :: type l : list(t), d : $t \vdash$.
tail(cons(d, l)) = l.

9.12 Term Declaration Logic

sorts: data, list.

declarations:

(1) elist : elist.

(2) cons(data, list) : list.

(3) head(cons(data, list)) : data.

(4) tail(cons(data, list)) : list.

variables: d : data, l : list.

equations:

(5) head(cons list(d, l)) = d .

(6) tail(cons list(d, l)) = l .

In fact the last two declarations can be obtained as consepuences of the equations.

9.13 Unified Algebras

ops: data, list, nil, cons(_, _), head_ , tail_ .

axioms:

(1) nil : list.

(2) cons(_,_) :: data, list → list ($total$).

(3) head_ :: list → data ($partial$).

(4) tail_ :: list → list ($partial$).

(5) head cons(d : data, l : list, $= d$.

(6) tail cons(d : data, l : list, $= l$.

Optinal extras:

op: _[of _].

axioms:

(7) nil : list.

(8) t list = nil | cons(data, list).

(9) t list[of data] = list.

(10) t list[of nothing] = nil.

(11) $D \leq$ data, ⇒

(1) cons(_, _) ::.
D, list[of D] → list[of D];.

(2) head(_) :: list[of D] → D];.

(3) tail(_) :: list[of D] →list [of D];.

(4) nil[of D] = nil;

(5) cons(d:data, l:list)[of D[=.
cons(d & D, l[of D]);.

(6) nothing[of D] = nothing;

(7) (($L_1 \leq$ list) | ($L_2 \leq$ list)[of D] =.
L_1 [of D] | L_2[of D].

References

[1] P. Aczel. Term declaration logic and generalized composita. In *LICS'91, Proc. 6th Ann. Symp. on Logic in Computer Science,* pages 22–30. IEEE, 1991.

[2] H. Andréka, P. Burmeister, and I. Németi. Quasi-varieties of partial algebras – a unifying approach towards a two-valued model theory for partial algebras. Preprint 557, FB Mathematik und Informatik, TH Darmstadt, 1980.

[3] E. Astesiano and M. Cerioli. Non-strict don't care algebras and specifications. In *Proc. TAPSOFT'91, Brighton,* number 493 in Lecture Notes in Computer Science, pages 121–142. Springer-Verlag, 1991.

[4] E. Astesiano and M. Cerioli. Relationships between logical frameworks. This volume, 1992.

[5] G. Bernot, M. Bidoit, and C. Choppy. Abstract data types with exception handling: an initial approach based on a distinction between exceptions and errors. *Theoretical Comput. Sci.,* 46:13–46, 1986.

[6] G. Bernot and P. Le Gall. Label algebras. This volume, 1992.

[7] M. Bidoit. Algebraic specification of exception handling and error recovery by means of declarations and equations. In *ICALP'84, Proc. Int. Coll. on Automata, Languages, and Programming, Antwerp,* number 172 in Lecture Notes in Computer Science, pages 95–108. Springer-Verlag, 1984.

[8] M. Broy and M. Wirsing, Partial abstract types. *Acta Inf.,* 18:47–64, 1982.

[9] P. Burmeister. *A Model-Theoretic Oriented Approach to Partial Algebras.* Akademie-Verlag, Berlin, 1986.

[10] H. Comon. Completion of rewrite systems with membership constraints. Technical report, SUNY at Stony Brook, 1991.

[11] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

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

[13] H. Ehrig, P. Pepper, and F. Orejas. On recent trends in algebraic specification. In *ICALP'89, Proc. Int. Coll. on Automata, Languages, and Programming, Torino,* number 372 in Lecture Notes in Computer Science, pages 263–288. Springer-Verlag, 1989.

[14] M. Gogolla. Partially ordered sorts in algebraic specifications. In *Proc. 1984 Colloq. on Trees in Algebra and Programming, Bordeaux,* pages 139–153. Cambridge University Press, 1984.

[15] M. Gogolla. A final algebra semantics for errors and exceptions. In *Recent Trends in Data Type Specification,* number 116 in Series IFB, pages 89–103. Springer-Verlag, 1985.

[16] M. Gogolla. On parametric algebraic specifications with clean error handling. In *TAP-SOFT'87, Proc. Int. Joint Conf. on Theory and Practice of Software Development, Pisa, Volume 1,* number 249 in Lecture Notes in Computer Science, pages 81–95. Springer-Verlag, 1987.

[17] M. Gogolla, K. Drosten, U Lipeck, and H.-D. Ehrich. Algebraic and operational semantics of specifications allowing exceptions and errors. *Theoretical Comput. Sci,* 34:289–313, 1984.

[18] J. A. Goguen. Abstract errors for abstract data types. In *Proc. IFIP Working Conference on the Formal Description of Programming Concepts, St. Andrews, New Brunswick,* 1977. North-Holland, 1978.

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

[20] J. A. Goguen. Types as theories. In *Proc. Symposium on General Topology and Applicaions.* Oxford University Press, 1990.

[21] J. A. Goguen and R. M. Burstall. Introducing institutions. In *Proc. Logics of Programming Workshop,* number 164 in Lecture Notes in Computer Science, pages 221–256. Springer-Verlag, 1984.

[22] J. A. Goguen and R. M. Burstall. A study in the foundations of programming methodology: specifications, institutions, charters and parchments. In *Proc. Workshop on Category Theory and Computer Programming, Guildford,* number 240 in Lecture Notes in Computer Science, pages 313–333. Springer-Verlag, 1986.

[23] J. A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Technical Report SRI-CSL-89-10, Computer Science Lab., SRI International, July 1989.

[24] 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.

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

[26] C. Kirchner and H. Kirchner. Order-sorted computations in G-algebra. Draft, INRIA-Lorraine & CRIN, Nancy, December 1991.

[27] H.-J. Kreowski. Partial algebras flow from algebraic specifications, In *ICALP'87, Proc. Int. Coll. on Automata, Languages, and Programming, Karlsruhe,* number 267 in Lecture Notes in Computer Science, pages 521–530. Springer-Verlag, 1987.

[28] V. Manta, A. Salibra, and G. Scollo. Equational type logic. *Theoretical Comput. Sci,* 77:131–159, 1990.

[29] V. Manta, A. Salibra, and G. Scollo. On the expressiveness of equational type logic. In *The Unified Computation Laboratory.* Oxford University Press, 1992. To appear.

[30] N. Martí-Oliet and J. Mesguer. Inclusions and subtypes. Technical Report SRI-CSL-90-16, Computer Science Lab., SRI International, 1990.

[31] A. Mégrelis. *Algébre Galactique.* PhD thesis, Univ. de Nancy, 1990.

[32] A. Mégrelis. Partial algebra + order-sorted algebra = galactic algebra. Tech. Report CRIN 90-R-108, Centre de Recherche en Informatique de Nancy, 1990.

[33] B. Möller. On the algebraic spmification of infinite objects — ordered and continuous models of algebraic types. *Acta Inf.,* 22:537–578, 1985.

[34] P. D. Mosses. Unified algebras and action semantics. In *STACS'89, Proc. Symp. on Theoretical Aspects of Computer Science, Paderborn,* number 349 in Lecture Notes in Computer Science, pages 17–35. Springer-Verlag, 1989.

[35] P. D. Mosses. Unified algebras and institutions. In *LICS'89, Proc. 4th Ann. Symp. on Logic in Computer Science,* pages 304–312. IEEE, 1989.

[36] P. D. Mosses. Unified algebras and modules. In *POPL'89, Proc. 16th Ann. ACM Symp. on Principles of Programming Languages,* pages 329–343. ACM, 1989.

[37] P. D. Mosses. *Action Semantics.* Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[38] O. Owe and O.-J. Dahl. Generator induction in order-sorted algebra. *Formal Aspects of Computing,* 3:2–20, 1991.

[39] A. Poigné. Another look at parameterization using algebraic specifications with subsorts. In *MFCS'84, Proc. Symp. on Math. Foundations of Computes Science,* number 176 in Lecture Notes in Computer Science. Springer-Verlag, 1984.

[40] A. Poigné. Partial algebras, subsorting and dependent types. In D. Sannella and A. Tarlecki, editors, *Recent Trends in Data Type Specification,* number 332 in Lecture Notes in Computer Science, pages 208-234. Springer-Verlag, 1988.

[41] A. Poigné. Typed Horn logic, In *MFCS'90, Proc. Symp. on Math. Foundations of Computer Science,* number 452 in Lecture Notes in Computer Science, pages 470–477. Springer-Verlag, 1990.

[42] A. Poigné. Once more on order-sorted algebra. In *MFCS'91, Proc. Symp. on Math. Foundations of Computer Science,* number 520 in Lecture Notes in Computer Science, pages 397–405. Springer-Verlag, 1991.

[43] Z. Qian Relation-sorted algebraic specifications with built-in coercers: parameterization and parameter passing. In *Proc. Workshop on Categorical Methods in Computer Science with Aspects from Topology,* number 393 in Lecture Notes in Computer Science, page 244–260. Springer-Verlag, 1989.

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

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

[46] G. Scollo. On the use of equational type logic for software engineering and protocol design. In Numidio, editor, *Proc. 1st Maghrebin Conf. on Artificial Intelligece and Software Engineering, Constantine, Algeria, 1989,* pages 460–485, 1991. Also available as Memoranda Informatica 89-52, Univ. Twente.

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

[48] G, Smolka. *Logic Programming over Polymorphically Over-Sorted Types.* PhD thesis, Universität Kaiserslautern, 1989.

[49] G. Smolka, W. Nutt, J. A. Goguen, and J. Meseguer. Order-sorted equations computation. In *Proc. Colloq. on Resolution of Equations in Algebraic Structures, Anstin.* Academic Press, 1989.

[50] W. W. Wadge. Classified algebras. Theory of Computation Report 46, University of Warwick, 1982.

[51] U. Waldmann. Semantics of order-sorted specifications. To appear in *Theoretical Comput. sci.,* 1992.

[52] M. Wirsing. Algebraic specification. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science,* volume B, chapter 13. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.