Proof of Correctness of Object Representations

Grant Malcolm and Joseph A. Goguen

1.1 Introduction

This paper presents an algebraic account of implementation that is applicable to the object paradigm. The key to its applicability is the notion of state: objects have local states that are observable only through their outputs. That is, objects may be viewed as abstract machines with hidden local state (as in [9]). Consequently, a correct implementation need only have the required visible behaviour.

We use hidden order sorted algebra to formalise the object paradigm [4, 5, 8]. Advantages of an algebraic approach include a high level of intellectual rigour, a large body of supporting mathematics, and simple, efficient proofs using only equational logic. A wide variety of extensions to equational logic have been developed to treat various programming features, while preserving its essential simplicity. For example, order sorted equational logic uses a notion of subsort to treat computations that may raise exceptions or fail to terminate.

Hidden sorted logic extends standard equational logic to capture an important distinction between *immutable data types*, such as booleans and integers, and *mutable objects*, such as program variables and database entities. The terms *abstract data types* and *abstract object classes* refer to these two kinds of entity. The former represent 'visible' data values; the latter represent data stored in a hidden state. In hidden sorted equational logic, an equation of hidden sort need not be satisfied in the usual sense, but only *up to observability*, in that only its visible consequences need hold. Thus, hidden sorted logic allows greater freedom in implementations.

The simplicity of the underlying logic is important, because we want a *tractable* approach in which implementations are as easily expressible and provable as possible. A specification is just a set of sentences in some logical system: that is, a *theory*. An algebraic specification is then a set of equations. An implementation is expressed by a *theory morphism*, which says how to interpret a theory into its implementation in such a way that each model of the concrete theory gives a model of the abstract theory. In this respect, our approach is similar to the seminal work of

Hoare on data refinement [15], in which correctness of implementation is expressed by a mapping from concrete variables to the abstract objects which they represent.

The following section introduces notation for hidden order sorted specifications, and summarises the main algebraic notions and results used in this paper. Order sorted algebra is the basis for the semantics of the executable specification language OBJ [13], and hidden order sorted algebra is the basis for an algebraic semantics of the object oriented language FOOPS [16]. Our examples of implementations use the notation of these languages. Section 1.3 presents implementations of hidden order sorted specifications, and a technique for proving correctness. We believe this technique leads to proofs that are simpler than those of other approaches. Section 1.3.3 applies this technique to the implementation of collections of objects.

This paper is dedicated with warm affection to Tony Hoare, whose work on data representation and on concurrency has been an inspiration to us.

1.2 Hidden order sorted algebra

Many sorted algebra (hereafter, 'MSA') was developed by the ADJ group [12] into a form suitable for abstract data types in computer science. The logic of MSA is first order equational logic, which is relatively simple. The following subsection summarises the main definitions and results of MSA, while Subsections 1.2.2 and 1.2.3 describe order sorted specification and hidden order sorted specification.

1.2.1 Many sorted algebra

An unsorted algebra is a set with 'structure' described by some operations and equations. The set is referred to as the carrier of the algebra. MSA extends this traditional view by defining an algebra to have any number of carriers. For example, what we might call a 'list algebra' is a quadruple (C, η, \oplus, e) , where the carriers are C_{Elt} and C_{List} , and $\eta : C_{\text{Elt}} \to C_{\text{List}}$ is a unary function, and $\oplus : C_{\text{List}} \times C_{\text{List}} \to C_{\text{List}}$ is an associative binary operation with neutral element $e \in C_{\text{List}}$; that is, the following equations are satisfied for all $x, y, z \in C_{\text{List}}$:

$$\begin{array}{rcl} x \oplus (y \oplus z) &=& (x \oplus y) \oplus z \\ e \oplus x &=& x \\ x \oplus e &=& x \end{array}$$

This specification of list algebras has three components: the carriers, named by the 'sorts' Elt and List; the operations η , \oplus and e; and the three equations above. We address each of these aspects in turn.

Definition 1 Given a set S, an S-sorted set is a collection $(A_s)_{s\in S}$ of sets indexed by elements of S. All set theoretic operations can be extended to operations on S-sorted sets; for example, if A and B are S-sorted sets, then $A \cup B$ is defined by $(A \cup B)_s = A_s \cup B_s$, and $A \subseteq B$ means that $A_s \subseteq B_s$ for each $s \in S$.

An S-sorted function $f : A \to B$ is a collection of functions indexed by S such that $f_s : A_s \to B_s$ for each $s \in S$. Similarly, an S-sorted relation R from A to B is a collection of relations indexed by S such that R_s is from A_s to B_s for each $s \in S$. We write the identity relation on an S-sorted set A as id_A .

For example, the carrier of a list algebra is an {Elt,List}-sorted set.

Definition 2 A many sorted signature is a pair (S, Σ) , where S is a set of sorts and Σ is an $(S^* \times S)$ -sorted set of operation names. Thus, if $l \in S^*$ and $s \in S$ then $\Sigma_{l,s}$ is a set of operation names. If Σ is clear from the context, we sometimes write $f: l \to s$ instead of $f \in \Sigma_{l,s}$ to emphasise that f is intended to denote an operation mapping the sorts denoted by l to the sort denoted by s. Usually we abbreviate (S, Σ) to Σ . Elements of $\Sigma_{\Pi,s}$ are referred to as **constants** of sort s.

An operation can be declared to have more than one type, e.g., we might have $f \in \Sigma_{l,s} \cap \Sigma_{l',s'}$ where l,s is different from l',s'. In this case, f is said to be **overloaded**.

Signatures provide a uniform notation for specifying the carriers and operations of many sorted algebras. Later sections consider implementing one specification by another; in order to compare two specifications, we use *signature morphisms*, which view one algebraic structure in terms of another.

Definition 3 A signature morphism $\phi : (S, \Sigma) \to (S', \Sigma')$ is a pair (ϕ_1, ϕ_2) , where $\phi_1 : S \to S'$ maps sorts in S to sorts in S', and ϕ_2 maps the operation names of Σ to operation names of Σ' in such a way that for each $f \in \Sigma_{l,s}$ we have $\phi_2(f) \in \Sigma'_{\phi_1^*(l),\phi_1(s)}$, where $\phi_1^*(l)$ denotes ϕ_1 applied componentwise to the list l; i.e., $\phi_1^*[] = []$ and $\phi_1^*(s l) = (\phi_1 s)(\phi_1^* l)$.

A useful example of a signature morphism is the inclusion of one signature in another: if $S \subseteq S'$ and $\Sigma \subseteq \Sigma'$, then there is an inclusion $\iota: (S, \Sigma) \to (S', \Sigma')$.

Signatures may be thought of as specifying algebras with no equations, and so we may speak of the algebras of a signature. An algebra for a signature Σ is an S-sorted set with the structure specified by the operation names of Σ .

Definition 4 For a many sorted signature Σ , a Σ -algebra A is given by the following data: an S-sorted set, usually denoted A, called the **carrier** of the algebra; an element $A_f \in A_s$ for each $s \in S$ and $f \in \Sigma_{[],s}$; and for each non-empty list $l \in S^*$, and each $s \in S$ and $f \in \Sigma_{l,s}$, an operation $A_f : A_l \to A_s$, where if $l = s1 \dots sn$ then $A_l = A_{s1} \times \cdots \times A_{sn}$.

Given Σ -algebras A and B, a Σ -homomorphism $h : A \to B$ is an S-sorted function $A \to B$ such that:

- given a constant $f \in \Sigma_{[],s}$, then $h_s(A_f) = B_f$;
- given a non-empty list $l = s1 \dots sn$ and $f \in \Sigma_{l,s}$ and $ai \in A_{si}$ for $i = 1, \dots, n$, then $h_s(A_f(a1, \dots, an)) = B_f(h_{s1}(a1), \dots, h_{sn}(an))$.

Thus, an algebra for a signature interprets the sort names as sets and the operation names as operations, while homomorphisms preserve the structure of the algebra in that they distribute over the operations of the algebra.

Given any signature, we can construct an algebra whose carriers are sets of terms built up from the given operation names viewed as symbols of an alphabet.

Definition 5 Given a many sorted signature Σ , the **term algebra** T_{Σ} is constructed as follows. Let $\cup \Sigma$ be the set of all operation names in Σ ; T_{Σ} is the least S-sorted set of strings over the alphabet $(\cup \Sigma) \cup \{(,)\}$ such that:

- for each constant symbol $f \in \Sigma_{[],s}$, the string $f \in (T_{\Sigma})_s$;
- for each non-empty list $l = s1...sn \in S^*$, and each $f \in \Sigma_{l,s}$, and all $ti \in (T_{\Sigma})_{si}$ for i = 1, ..., n, the string $f(t1...tn) \in (T_{\Sigma})_s$.

We show that T_{Σ} is a Σ -algebra by showing how the operation names of Σ are interpreted: for each constant $f \in \Sigma_{[],s}$, the constant $(T_{\Sigma})_f$ is the string $f \in (T_{\Sigma})_s$; for each non-empty list $l = s1 \dots sn \in S^*$ and operation name $f \in \Sigma_{l,s}$, the operation $(T_{\Sigma})_f : (T_{\Sigma})_l \to (T_{\Sigma})_s$ maps a tuple of strings $t1 \dots tn$ to the string $f(t1 \dots tn)$. The special symbols '(' and ')' are used to emphasise that the carriers of T_{Σ} are sets of strings; from now on we write ' $f(t1, \dots, tn)$ ' for ' $f(t1 \dots tn)$ '.

This shows that T_{Σ} is a Σ -algebra. In fact, if Σ contains no overloaded symbols, it has the special property of being an *initial* Σ -algebra.

Definition 6 An initial Σ -algebra is a Σ -algebra A such that for each Σ -algebra B there is exactly one Σ -homomorphism $A \to B$.

Proposition 7 If Σ contains no overloaded operation names, then T_{Σ} is an initial Σ -algebra. For any Σ -algebra A, the unique Σ -homomorphism $h: T_{\Sigma} \to A$ is defined recursively as follows:

- for each constant symbol $f \in \Sigma_{[],s}$, let $h_s(f) = A_f$;
- for each non-empty list l = s1...sn and $f \in \Sigma_{l,s}$ and $ti \in (T_{\Sigma})_{si}$ for i = 1, ..., n, let $h_s(f(t1, ..., tn)) = (A_f)(h_{s1}(t1), ..., h_{sn}(tn))$.

The homomorphism h assigns a value in A to terms by interpreting the operation names of Σ as the corresponding operations on A. If Σ contains overloaded operations, an initial algebra can still be constructed as a term algebra where the operation names are distinguished by 'tagging' them with their result sorts [7].

Let us now consider algebras with equations. An equation is usually presented as two terms (the left- and right-hand sides) which contain variables. For example, one of the equations for list algebras was $(x \oplus y) \oplus z = x \oplus (y \oplus z)$, where x, y and z are variables that range over C_{List} . Because variables only serve as placeholders for values of the sorts that they range over, any signature of constant symbols can be used to provide variables. Definition 8 A ground signature is a signature (S, Σ) such that for all $l \in S^*$ and $s \in S$, if $l \neq []$ then $\Sigma_{l,s} = \emptyset$, and such that the $\Sigma_{l,s}$ are disjoint; i.e., the operation names of ground signatures are distinct constants.

We assume disjointness so that distinct variables cannot be identified.

Ground signatures are essentially the same thing as S-sorted sets, because any S-sorted set X can be viewed as a ground signature by taking $X_{l,s}$ to be X_s if l = [] and the empty set otherwise. Moreover, a ground signature Σ can be viewed as the S-sorted set $(\Sigma_{[],s})_{s\in S}$. This determines a bijection between ground signatures and S-sorted sets; we take advantage of this by sometimes treating ground signatures as S-sorted sets. Now it is a simple matter to characterise terms containing variables:

Definition 9 Given a many sorted signature (S, Σ) and a ground signature (S, X)such that Σ and X are disjoint, **terms with variables from** X are elements of $T_{\Sigma \cup X}$. Now $T_{\Sigma \cup X}$ can be viewed as a Σ -algebra if we forget about the constants in X: when we view $T_{\Sigma \cup X}$ as a Σ -algebra, we write it as $T_{\Sigma}(X)$.

A Σ -equation is a triple (X, l, r), where (S, X) is a ground signature, and land r are terms in $T_{\Sigma}(X)$ of the same sort; i.e., $l, r \in T_{\Sigma}(X)_s$ for some $s \in S$. We write such an equation in the form $(\forall X) \ l = r$.

A specification is a triple (S, Σ, E) , where (S, Σ) is a signature and E is a set of Σ -equations. We usually abbreviate (S, Σ, E) to (Σ, E) .

Algebras of a specification are Σ -algebras that satisfy the equations; we turn now to what it means for an algebra to satisfy an equation. The first issue is how to interpret the left- and right-hand sides of an equation in an arbitrary Σ -algebra. Because $T_{\Sigma}(X)$ is a Σ -algebra, there is a homomorphism $T_{\Sigma} \to T_{\Sigma}(X)$, which is the inclusion of variable-free terms into terms with variables. However, T_{Σ} is not in general a $(\Sigma \cup X)$ -algebra, because we do not know how to interpret the variables in X. If we can assign values to those variables, then we can assign values to terms containing those variables.

Proposition 10 Given a Σ -algebra A and an S-sorted function $\theta: X \to A$ (often called an 'interpretation of variables'), there is a unique Σ -homomorphism $\bar{\theta}: T_{\Sigma}(X) \to A$ such that $\bar{\theta}(\iota(x)) = \theta(x)$ for all variables x, where $\iota: X \to T_{\Sigma}(X)$ maps $x \in X_s$ to the string $x \in T_{\Sigma}(X)_s$. The homomorphism is defined as follows:

- for each $x \in X_s$, let $\bar{\theta}_s(x) = \theta_s(x)$;
- for each constant symbol $f \in \Sigma_{[],s}$, let $\bar{\theta}_s(f) = A_f$;
- for each non-empty list l = s1...sn, $f \in \Sigma_{l,s}$, and all $ti \in T_{\Sigma}(X)_{si}$ for i = 1, ..., n, let $\bar{\theta}_s(f(t1, ..., tn)) = A_f(\bar{\theta}_{s1}(t1), ..., \bar{\theta}_{sn}(tn))$.

Equations have an implicit universal quantification over the variables. An algebra satisfies a given equation iff the left- and right-hand sides of the equation are equal under all interpretations of the variables:

Definition 11 A Σ -algebra A satisfies a Σ -equation $(\forall X) \ l = r$ iff $\overline{\theta}(l) = \overline{\theta}(r)$ for all $\theta: X \to A$. We write $A \models e$ to indicate that A satisfies the equation e. For a set E of equations, we write $A \models E$ iff $A \models e$ for each $e \in E$. Given a specification (Σ, E) , a (Σ, E) -algebra is a Σ -algebra A such that $A \models E$.

Just as each signature has an initial algebra, each specification has an initial algebra. The initial algebra is constructed from the term algebra by identifying terms that are 'equal' as a consequence of the given equations. This identification is achieved using the notion of *congruence*.

Each equation gives rise to a relation in the following way: given a Σ -algebra A let e be a Σ -equation $(\forall X) \ l = r$, and define the relation $\mathsf{R}(e) : A \sim A$ by $a \ \mathsf{R}(e) \ b$ iff $a = \overline{\theta}(l)$ and $b = \overline{\theta}(r)$ for some $\theta : X \to A$. In other words, a is related to b iff a is an instance of the left-hand side and b is an instance of the right-hand side, under some interpretation of the variables. We seek an equivalence relation that contains all the relations derived from the equations of a specification, and that allows the substitution of equals for equals.

Definition 12 Given a signature Σ and a Σ -algebra A, a Σ -congruence is an S-sorted equivalence relation R such that the following substitutivity property holds: for all $f \in \Sigma_{l,s}$ and $x, y \in A_l$, if $x \ R_l \ y$ then $A_f(x) \ R_s \ A_f(y)$, where if $l = s1 \dots sn$, then $x \in A_l$ means $x = x1 \dots xn$ with $xi \in A_{si}$, and $x \ R_l \ y$ means $xi \ R_{si} \ yi$ for $i = 1, \dots, n$.

If E is a set of Σ -equations and A is a Σ -algebra, then $\equiv_{A,E}$ denotes the least Σ -congruence on A which contains each equation in E; that is, for each $e \in E$, $\mathsf{R}(e) \subseteq \equiv_{A,E}$. We usually write $=_E$ instead of $\equiv_{T_{\Sigma},E}$.

The Σ -congruence $=_E$ allows the identification of terms which are equal as a result of the equations E.

Proposition 13 Given a specification (Σ, E) where Σ contains no overloaded operations, the initial (Σ, E) -algebra is the **quotient term algebra** $T_{\Sigma,E} = T_{\Sigma}/=_E$. That is, the carriers of $T_{\Sigma,E}$ are sets of equivalence classes under $=_E$; specifically, $(T_{\Sigma,E})_s = \{[t] \mid t \in (T_{\Sigma})_s\}$, where [t] denotes the equivalence class of t under $=_E$. The structure of $T_{\Sigma,E}$ as a Σ -algebra is given by:

- for each constant symbol $f \in \Sigma_{[],s}$, let $(T_{\Sigma,E})_f = [f];$
- for each non-empty list $l = s1 \dots sn$, $f \in \Sigma_{l,s}$, and $[ti] \in (T_{\Sigma,E})_{si}$ for $i = 1, \dots, n$, let $(T_{\Sigma,E})_f([t1], \dots, [tn]) = [(T_{\Sigma})_f(t1, \dots, tn)]$.

The last equation is well-defined by the substitutivity property of the congruence $=_E$. By construction, $T_{\Sigma,E}$ satisfies the equations E.

The above proposition refers to 'the' initial (Σ, E) -algebra, but a specification may have more than one initial algebra. However, any two initial (Σ, E) -algebras are isomorphic, because the unique homomorphisms from each algebra to the other are inverses. Thus all initial algebras are 'abstractly the same'. ADJ [12] define an abstract data type to be the collection of initial algebras of a specification. Such a collection is an equivalence class, since being isomorphic is an equivalence relation, and this equivalence class may be represented by $T_{\Sigma,E}$. The importance of initiality is that it gives a canonical interpretation of a specification as an abstract data type. Moreover, completeness results state that a Σ -equation is satisfied by all (Σ, E) -algebras iff it can be proved using the equations E and the standard properties of equality: reflexivity, symmetry, transitivity and substitutivity. This allows the use of equational deduction in prototyping and proving properties of specifications, for example, using OBJ [13].

1.2.2 Order sorted algebra

Partial operations and error handling play an important rôle in many computer science applications. A partial operation produces well-defined values only on some subsort of its domain. For example, division in a field produces a well-defined value only when the denominator is not zero. Order sorted algebra (hereafter, 'OSA') is a variation on MSA that allows algebras with partial operations. It also provides a model of inheritance that is useful in formalising the object paradigm. This subsection summarises definitions and results of OSA that are relevant to this paper. A comprehensive survey is given by Goguen and Diaconescu in [7].

Both OSA and MSA are based on the notion of S-sorted sets, but whereas in MSA S is a set, in OSA S is a partially ordered set. If S is a set of sort names, the partial order indicates the subsort relations between the carriers of algebras. For a partially ordered set (S, \leq) , we refer to \leq as **the subsort ordering**. We sometimes extend this ordering to lists over S of equal length by $s1...sn \leq s1'...sn'$ iff $si \leq si'$ for i = 0, ..., n.

Definition 14 Given a partial order (S, \leq) , an equivalence class of the transitive symmetric closure of \leq is called a **connected component**, and two elements of the same connected component are said to be **connected**. A partial order (S, \leq) is **locally filtered** iff any two connected sorts have a common supersort, that is, iff whenever s and s' are connected, there is an s'' such that $s, s' \leq s''$.

The notion of local filtering allows many results of MSA to extend to OSA [7].

Definition 15 An (S, \leq) -sorted set is an S-sorted set A such that whenever $s \leq s'$ then $A_s \subseteq A_{s'}$. An (S, \leq) -sorted function $f: A \to B$ is an S-sorted function such that whenever $s \leq s'$ then $f_s \subseteq f_{s'}$. An (S, \leq) -sorted relation R from A to B is an S-sorted relation such that if $s \leq s'$ and $x \in A_s$ and $y \in B_s$, then $x R_s y$ iff $x R_{s'} y$. We sometimes abbreviate (S, \leq) -sorted' to 'S-sorted'.

Most definitions of MSA apply, *mutatis mutandis*, to OSA; the main differences concern monotonicity.

Definition 16 An order sorted signature is a triple (S, \leq, Σ) where (S, \leq) is a locally filtered partial order and (S, Σ) is a many sorted signature which satisfies the monotonicity requirement: if $f \in \Sigma_{l,s} \cap \Sigma_{l',s'}$ and $l \leq l'$ then $s \leq s'$. We usually abbreviate (S, \leq, Σ) to just Σ .

An order sorted signature morphism $\phi : (S, \leq, \Sigma) \to (S', \leq', \Sigma')$ is a many sorted signature morphism such that $\phi_1 : (S, \leq) \to (S', \leq')$ is monotonic. A signature morphism ϕ preserves overloading iff whenever $f \in \Sigma_{l,s} \cap \Sigma_{l',s'}$ then ϕ_2 applied to $f \in \Sigma_{l,s}$ gives the same result as ϕ_2 applied to $f \in \Sigma_{l',s'}$.

Monotonicity is also needed for the algebras of an order sorted signature.

Definition 17 Given an order sorted signature (S, \leq, Σ) , an order sorted Σ algebra is a many sorted Σ -algebra A such that A is an (S, \leq) -sorted set and Ais monotonic, in the sense that for all $f \in \Sigma_{l,s} \cap \Sigma_{l',s'}$ if $l \leq l'$ and $s \leq s'$ then $A_f: A_l \to A_s$ is equal to $A_f: A_{l'} \to A_{s'}$ on A_l .

For order sorted Σ -algebras A and B, an order sorted Σ -homomorphism $h: A \to B$ is a many sorted Σ -homomorphism which satisfies the restriction condition: if $s \leq s'$ then $h_s = |h_{s'}|_{A_s}$ where $|h_{s'}|_{A_s}$ denotes the restriction of $h_{s'}: A_{s'} \to B_{s'}$ to A_s .

The construction of the term algebra is as in MSA, but requires the carrier of T_{Σ} to be (S, \leq) -sorted, so that $(T_{\Sigma})_s \subseteq (T_{\Sigma})_{s'}$ whenever $s \leq s'$. In general, T_{Σ} is not an initial Σ -algebra unless Σ satisfies a regularity condition [11]:

Definition 18 An order sorted signature Σ is **regular** iff for any $f \in \Sigma_{l_{1,s_1}}$ and $l_0 \leq l_1$ there is a least pair (l,s) such that $l_0 \leq l$ and $f \in \Sigma_{l,s}$.

The importance of regularity is that terms can be parsed as having a least sort. Goguen and Diaconescu [7] note that regularity is not essential, in that OSA can be developed in greater generality under the assumption only of local filtering. The construction of an initial algebra is then more complicated, and we do not give details here, as all specifications in this paper are regular.

Unlike in MSA, the left- and right-hand sides of an equation need not have the same sort; their sorts need only be connected.

Definition 19 Given an order sorted signature (S, \leq, Σ) , a Σ -equation is a triple (X, l, r), where X is a ground signature disjoint from Σ with $l \in T_{\Sigma}(X)_s$ and $r \in T_{\Sigma}(X)_{s'}$ for some connected $s, s' \in S$. We use the notation $(\forall X) \ l = r$.

The definitions of satisfaction of equations and congruence in OSA are as in MSA, but with 'S-sorted' everywhere changed to ' (S, \leq) -sorted'. An **order sorted spec-ification** is an order sorted signature together with a set E of Σ -equations, and a (Σ, E) -**algebra** is a Σ -algebra which satisfies all equations in E. The quotient term algebra $T_{\Sigma,E}$ is constructed as in MSA, dividing by the least (S, \leq) -sorted Σ -congruence which extends the equations of the specification. If the signature is regular, this gives an initial (Σ, E) -algebra. We end our summary of OSA with 'retract' specifications, which allow operations to be applied to arguments which may lie outside their domain of definition, possibly resulting in values that are 'ill-defined' in the sense that they involve the special 'retract' operations. This allows order sorted specifications to model partial operations (see [11, 7] for a full treatment).

Definition 20 Given an order sorted specification $P = (S, \leq, \Sigma, E)$, we write P^{\otimes} for its **retract extension** $(S, \leq, \Sigma^{\otimes}, E^{\otimes})$, where Σ^{\otimes} is Σ extended with an operation $r_{s_{1},s_{2}}: s_{1} \rightarrow s_{2}$ for each $s_{1}, s_{2} \in S$ such that $s_{2} \leq s_{1}$, and E^{\otimes} is E extended with an equation $(\forall S:s_{2}) r_{s_{1},s_{2}}(S) = S$ for each $s_{2} \leq s_{1}$ as above.

We wish the result of adding retracts to be a conservative extension of the given specification; that is, for all $t1, t2 \in T_{\Sigma}$, $t1 =_E t2$ iff $t1 =_{E^{\otimes}} t2$, i.e., the new equations added by the introduction of retracts do not cause distinct terms of T_{Σ} to become identified. Goguen and Meseguer [11] give sufficient conditions on specifications for adding retracts to be conservative. These conditions go beyond the scope of the present paper, but we note that all our example specifications are such that their retract extensions are conservative.

1.2.3 Hidden order sorted algebra

Hidden sorted algebra was developed as a variation on MSA for objects with local states [5, 8]. In a hidden sorted specification, the set of sort names is partitioned into 'visible' and 'hidden' sorts. Operations which return hidden sorted values correspond to the internal operations of an object, while visible sorted values correspond to an object's inputs and outputs. This subsection summarises the basic definitions of hidden sorted algebra, and then combines it with OSA to give hidden order sorted algebra (hereafter, 'HOSA').

Definition 21 A hidden sorted signature is a triple (S, V, Σ) , where (S, Σ) is a many sorted signature and $V \subseteq S$. The elements of V are referred to as visible sorts, and elements of S - V as hidden sorts.

A hidden sorted signature morphism $\phi : (S, V, \Sigma) \to (S', V', \Sigma')$ is a many sorted signature morphism which maps visible sorts to visible sorts and hidden sorts to hidden sorts, i.e., $s \in V$ iff $\phi_1(s) \in V'$.

A hidden sorted specification is a hidden sorted signature together with a set E of Σ -equations (in the sense of MSA).

We might refer to the above as 'pre-signature' and 'pre-signature morphism' (and similarly for the HOSA definitions below), since Goguen and Diaconescu [8] give extra restrictions on signatures which correspond to aspects of object oriented computation and make hidden sorted algebra an institution [1]. We do not make these extra restrictions here as they are not necessary for our results. However, we note that in [8] a hidden sorted specification includes a fixed 'data algebra' D for the visible sorts such that for any Σ -algebra A, $D_v = A_v$ for all visible sorts v.

The definition of satisfaction differs from MSA in that only the visible consequences of an equation need hold. The notion of 'visible consequence' is made precise by defining *contexts* for terms:

Definition 22 Given a term $t \in (T_{\Sigma})_s$, a context for t of sort s' is a term $c \in T_{\Sigma}(\{z\})_{s'}$ where z is a new variable of sort s, i.e., a context is just a term which contains a distinguished variable. We write $T_{\Sigma}[z]$ instead of $T_{\Sigma}(\{z\})$, and if c is a context for t, we write c[t] for the result of substituting t for z in c.

Contexts of visible sort can be considered experiments which, applied to an object's hidden state, give visible outputs. The definition of satisfaction for HSA says that two states are distinguished iff they give different results for some experiment, and an equation is behaviourally satisfied if its left- and right-hand sides always instantiate to states that cannot be distinguished by any experiment.

Definition 23 A Σ -algebra A behaviourally satisfies a Σ -equation e of the form $(\forall X) \ l = r$ (denoted $A \models_b e$) iff $A \models (\forall X) \ c[l] = c[r]$ for all $v \in V$ and $c \in T_{\Sigma}[z]_v$. Implicitly, the variable z has the same sort as l and r. For a set E of Σ -equations, we write $A \models_b E$ iff $A \models_b e$ for all $e \in E$.

If an equation has visible sort, then behavioural satisfaction is the same as satisfaction in MSA, because for c we can always choose the 'empty context' $z \in T_{\Sigma}[z]_{v}$. Behavioural satisfaction of equations can also be expressed more abstractly:

Proposition 24 $A \models_b E$ iff $(\equiv_{A,E})|_V \subseteq id_A|_V$ where $R|_V$ is the restriction of an S-sorted relation R to the visible sorts of V i.e., $R|_V$ is the V-sorted relation $(R_v)_{v \in V}$.

This can be read as saying that E does not identify distinct elements of A of visible sort, which we might summarise by saying there is 'no confusion'.

A **behavioural** (Σ, E) -algebra is a Σ -algebra A such that $A \models_b E$. The notion of implementation that we use in the following sections is based on the idea that an object is implemented by a behavioural algebra of its specification.

We now give the hidden sorted version of OSA.

Definition 25 An **HOSA signature** is a quadruple (S, V, \leq, Σ) where (S, \leq, Σ) is an order sorted signature, and the visible sorts $V \subseteq S$ partition the partially ordered set S in the sense that whenever $s \leq s'$ then $s \in V$ iff $s' \in V$.

An **HOS signature morphism** $\phi : (S, V, \leq, \Sigma) \to (S', V', \leq', \Sigma')$ is an order sorted signature morphism which maps visible sorts to visible sorts and hidden sorts to hidden sorts in the sense that $s \in V$ iff $\phi_1(s) \in V'$.

The definitions of algebra, equation, specification and retract are as in OSA, but satisfaction of equations in HOSA only requires that the visible consequences of an equation hold. Because of the order-sortedness of HOSA signatures, defining satisfaction in terms of contexts would require contexts that contain retracts. Burstall and Diaconescu [1] give an abstract categorical definition of behavioural satisfaction for OSA, which is beyond the scope of the present paper. Moreover, their definitions of HOSA signature, etc., are different from those given here; for example, they require that the visible sorts have a fixed interpretation (cf. the comments after Definition 21). Because of these differences, we say instead that a Σ -algebra A behaviourally satisfies E iff there is no confusion in the sense of Proposition 24 (i.e., $(\equiv_{A,E})|_V \subseteq id_A|_V$). A behavioural (Σ, E) -algebra is a Σ -algebra A such that A behaviourally satisfies E.

This definition of satisfaction generalises the definition for OSA, in that given an order sorted specification (S, \leq, Σ, E) , we can construct the HOSA specification (S, S, \leq, Σ, E) where all sorts are visible; then for any (S, \leq, Σ) -algebra A, we have $A \models E$ iff $A \models_b E$.

Section 1.3 considers implementations using a translation from terms of the abstract specification to terms of the concrete specification; this states how programs of the abstract specification are to be 'compiled'. The simplest way to achieve such a translation is by a signature morphism: if we know how to translate operations of the abstract specification, then we can translate terms built from those operations. Often the abstract signature is contained in the concrete; that is, all the sorts and operations of the abstract specification are available in the concrete one. In that case all terms over the abstract signature are also terms over the concrete signature. However, non-inclusion translations are sometimes useful (see Subection 1.3.2). The following states how an arbitrary signature morphism extends to a translation of terms.

Proposition 26 Every signature morphism $\phi: \Sigma \to \Sigma'$ which preserves overloading can be extended to a function ϕ such that $\phi: (T_{\Sigma})_s \to (T_{\Sigma'})_{\phi_1(s)}$ for all $s \in S$. This extension is defined as follows:

- for each constant symbol $f \in \Sigma_{[],s}$, let $\phi(f) = \phi_2(f)$;
- for each non-empty list l = s1...sn, $f \in \Sigma_{l,s}$, and all $ti \in (T_{\Sigma})_{si}$ for i = 1, ..., n, let $\phi(f(t1, ..., tn)) = (\phi_2(f))(\phi(t1), ..., \phi(tn))$.

If ϕ_1 is an inclusion of S into S' then ϕ is an S-sorted function $T_{\Sigma} \to T_{\Sigma'}$ and if $\Sigma \subseteq \Sigma'$ then ϕ is the unique inclusion homomorphism $T_{\Sigma} \to T_{\Sigma'}$, so that terms of T_{Σ} are also terms of $T_{\Sigma'}$.

Moreover, ϕ extends to $\phi^{\otimes}: T_{\Sigma^{\otimes}} \to T_{\Sigma'^{\otimes}}$ by setting $\phi_2^{\otimes}(r_{s1,s2}) = r_{\phi_1(s1),\phi_1(s2)}$. Finally, ϕ extends to $\phi_s: T_{\Sigma}(X)_s \to T_{\Sigma'}(X')_{\phi_1(s)}$ for each $s \in S$, where $X'_{s'} = \{x \in X_s \mid \phi_1(s) = s'\}$; thus, ϕ may change the sort but not the name of a variable. Note that because all the variables of X are distinct (cf. Definition 8), ϕ cannot identify distinct variables.

1.3 Implementation

This section defines implementation for HOSA specifications and presents a technique for proving correctness of implementation that is illustrated in two examples.

Let us now fix two HOSA specifications, $A = (SA, \leq_A, VA, \Sigma A, EA)$ and $C = (SC, \leq_C, VC, \Sigma C, EC)$, where A is for 'abstract' and C is for 'concrete', plus a signature morphism $\phi : \Sigma A \to \Sigma C$ which preserves overloading. We also use the following abbreviations:

Notation 27 Write TA for the carrier of the term algebra $T_{\Sigma A}$; TA^{\otimes} for that of $T_{\Sigma A^{\otimes}}$ (cf. Definition 20); TA[z] for the contexts in $T_{\Sigma A}[z]$ and $TA[z]^{\otimes}$ for $T_{\Sigma A^{\otimes}}[z]$ (cf. Definition 22); TA_s for the terms of sort s, etc., and similarly for C; and ϕ for $\phi: TA \to TC$ as well as for $\phi^{\otimes}: TA^{\otimes} \to TC^{\otimes}$ (cf. Proposition 26).

What do we mean by 'A is implemented by C'? If we ignore hidden and order sortedness, the answer is straightforward: all ground equalities in A, when translated by ϕ , should hold in C; i.e.,

(1.1) $t_1 =_{EA} t_2$ implies $\phi(t_1) =_{EC} \phi(t_2)$ for all $t_1, t_2 \in TA$.

The intuitive meaning is that if t_1 is some program that gives result t_2 in the abstract specification, then its translation should give the corresponding result in the implementation. More formally, (1.1) states that the ϕ -translations of the ground consequences of the equations EA are entailed by EC. It can be shown that this is equivalent to requiring the ϕ -translations of ground instances of the equations in EA to be entailed by EC, i.e., that

(1.2) $\phi(\bar{\theta}(lhs)) =_{EC} \phi(\bar{\theta}(rhs))$ for each $(\forall X) lhs = rhs \in EA$ and $\theta: X \to TA$.

This is equivalent to (1.1), but has a form that is generally easier to prove.

If we take hidden sortedness into account, we need only consider equalities of visible sort, so that the requirement (1.1) for implementation becomes

(1.3) $t_1 =_{EA} t_2$ implies $\phi(t_1) =_{EC} \phi(t_2)$ for all $v \in VA$ and $t_1, t_2 \in TA_v$.

This is the definition given by Henniker in [14], though only for signature inclusions. Henniker also proposes a method for proving implementation correctness based on induction over the structure of contexts, by restating this condition in terms of behavioural equivalence.

Definition 28 $t, t' \in TC$ are A-behaviourally equivalent, written $t \sim t'$, iff for all $v \in VA$ and $c \in TA[z]_v$, we have $\phi(c)[t] =_{EC} \phi(c)[t']$, where $\phi(c)[t]$ denotes the ϕ -translation of c with the term t substituted for z. Implicitly, if the variable z has sort s, then t and t' have sort $\phi_1(s)$.

It can be shown that (1.3) is equivalent to:

(1.4) $t_1 =_{EA} t_2$ implies $\phi(t_1) \sim \phi(t_2)$ for all $s \in SA$ and $t_1, t_2 \in TA_s$.

Henniker shows that (1.3), and therefore (1.4), is equivalent to:

(1.5) $\phi(\bar{\theta}(lhs)) \sim \phi(\bar{\theta}(rhs))$ for each $(\forall X) lhs = rhs \in EA$ and $\theta: X \to TA$.

The equivalence of (1.1) and (1.2) is mirrored in that of (1.4) and (1.5). Both (1.2) and (1.5) have a form that simplifies the proof obligations. However, proofs of (1.5) can still be surprisingly complicated (e.g., see [14]).

The situation is more complex for OSA because the definition of implementation has to consider well-definedness of terms, which may amount to termination of programs. Schoett [17] defines implementation for partial algebras, and gives a necessary and sufficient condition in terms of a congruence between models of the abstract and concrete specifications. Schoett's definition is stronger than that given below: he restricts attention to terms all of whose subterms are equal to a well-defined value (in our setting this means that they contain no retract operations). For example, consider an abstract specification of stacks with operations top, pop, empty and push (as in Section 1.3.2 below), where top requires a non-empty stack as argument, and should satisfy the equation $(\forall X : \text{Nat}, S : \text{Stack}) \text{ top push}(X, S) = X$. The term top push(0, pop empty) can be viewed in two ways: it either gives the value 0, or else is undefined. The first view corresponds to lazy evaluation, where terms with ill-defined subterms can still have well-defined values; the second view, implicit in Schoett's definition, corresponds to call-by-value, where any term with an undefined subterm is itself undefined. In Schoett's call-by-value approach, an implementation of stacks may allow top push(0, pop empty) to take any value at all. We consider 'lazy' implementation important because many programming languages either have lazy evaluation or else facilities for error handling.

Our definition of implementation in HOSA is that C implements A iff whenever a visible sorted term of TA^{\otimes} gives a well-defined value (i.e., a term of TA), then the ϕ -translation of that term gives the corresponding value in TC.

Definition 29 A specification C is a **partial behavioural implementation** of a specification A via the signature morphism ϕ (we write $\phi : A \sqsubseteq C$) iff $t =_{EA^{\otimes}} t'$ implies $\phi(t) =_{EC^{\otimes}} \phi(t')$ for all $v \in VA$, $t \in TA_v^{\otimes}$ and $t' \in TA_v$. We say that C behaviourally implements A iff the above implication is an equivalence.

This definition of partial implementation generalises (1.3) to the order sorted case. The difference between partial implementation and implementation is that in the latter the mapping ϕ from TA^{\otimes} to TC^{\otimes} , or more properly from $T_{\Sigma A^{\otimes}, EA^{\otimes}}$ to $T_{\Sigma C^{\otimes}, EC^{\otimes}}$, is injective on the visible sorts in the sense that it doesn't confuse distinct data values. Consequently, 'trivial' implementations, in which all equations are satisfied, are not allowed. If our definitions for HOSA had followed [8], in particular by requiring a fixed interpretation for visible data sorts, then $\phi : A \sqsubseteq C$ would imply that ϕ is injective on visible sorts because of their fixed interpretation. In the following, we concentrate on proofs of partial implementation, i.e., on showing that terms equal in the abstract specification are equal in the concrete.

We note that if ϕ is a signature inclusion and $\phi : A \sqsubseteq C$ then any behavioural algebra of C is also a behavioural algebra of A.

1.3.1 Proofs of partial implementation

One way to show that C implements A is to construct an intermediate relation R on C terms such that (a) if $t =_{EA^{\otimes}} t'$ then the ϕ -translations of t and t' are related by R, and (b) the restriction of R to visible sorted ϕ -translations is contained in $=_{EC^{\otimes}}$. Such a relation bridges the gap between the antecedent and consequent in Definition 29. If R is also a ΣA^{\otimes} -congruence, then (a) holds iff R extends the ground instances of the equations in EA^{\otimes} . This is the intuition behind Proposition 31 below, which is our main technical result. Its statement uses the following:

Notation 30 If R is a relation on TC^{\otimes} , then the relation R^{ϕ} on TA^{\otimes} is defined for $t, t' \in TA^{\otimes}$ by: $t R^{\phi} t'$ iff $\phi(t) R \phi(t')$.

Proposition 31 $\phi: A \sqsubseteq C$ if there exists an equivalence relation R on TC^{\otimes} such that R^{ϕ} is a ΣA^{\otimes} -congruence and

 $(1.6) \qquad \bar{\theta}(\mathit{lhs}) \quad R^{\phi} \quad \bar{\theta}(\mathit{rhs}) \quad \text{ for each } (\forall X) \mathit{lhs} \!=\! \mathit{rhs} \in EA^{\otimes} \ \text{ and } \ \theta: X \!\rightarrow\! TA^{\otimes} \ ,$

(1.7) if $t R^{\phi} t'$ then $\phi(t) =_{EC^{\otimes}} \phi(t')$ for all $v \in VA, t \in TA_v^{\otimes}$ and $t' \in TA_v$.

Proof: The relation $=_{EA^{\otimes}}$ is by definition the least ΣA^{\otimes} -congruence satisfying (1.6), so $=_{EA^{\otimes}} \subseteq R^{\phi}$. To show that $\phi : A \sqsubseteq C$, fix $v \in VA$, $t \in TA_v^{\otimes}$, $t' \in TA_v$; if $t =_{EA^{\otimes}} t'$ then because $=_{EA^{\otimes}} \subseteq R^{\phi}$, we have $t R^{\phi} t'$, and since t and t' are of visible sort, (1.7) gives $\phi(t) =_{EC^{\otimes}} \phi(t')$ as desired.

A weaker, but very useful version of this result is obtained by strengthening (1.6):

Proposition 32 For any relation R on TC^\otimes , condition (1.6) of Proposition 31 follows from

$$(1.8) =_{EC^{\otimes}} \subseteq R ,$$

(1.9)
$$\bar{\theta}(lhs) R^{\phi} \bar{\theta}(rhs)$$
 for each $(\forall X) lhs = rhs \in EA$ and $\theta: X \to TA^{\otimes}$.

Proof: EA^{\otimes} consists of EA plus equations of the form $(\forall S : s2) r_{s1,s2}(S) = S$. By construction, EC^{\otimes} contains the equation $(\forall S' : \phi_1(s2)) r_{\phi_1(s1),\phi_1(s2)}(S') = S'$, so for any $\theta : \{S\} \rightarrow TA^{\otimes}$, we have $\phi(\bar{\theta}(r_{s1,s2}(S))) = r_{\phi_1(s1),\phi_1(s2)}(\phi(\bar{\theta}(S))) =_{EC^{\otimes}} \phi(\bar{\theta}(S))$. Therefore by (1.8), $\bar{\theta}(r_{s1,s2}(S)) R^{\phi} \bar{\theta}(S)$, and combining this with (1.9) gives (1.6).

The weakening of Proposition 31 by replacing (1.6) with (1.8) and (1.9) is useful because with (1.8), in proving that two terms are related by R we may freely rewrite those terms using the equations of EC^{\otimes} ; moreover, the example relations R that we use below satisfy (1.8), so that in proving partial implementation, we may concentrate on proving (1.9), ignoring the retract equations.

To use these results, we need a suitable relation R. A likely candidate is behavioural equivalence, which we could define as in Definition 28; but the following relation is more general :

Definition 33 For $t, t' \in TC^{\otimes}$, equivalence up to definition, denoted $t \simeq t'$, is defined by: $t =_{EC^{\otimes}} t'' \Leftrightarrow t' =_{EC^{\otimes}} t''$ for all $t'' \in TC$.

Note that if $t, t' \in TC$, then $t \simeq t'$ iff $t =_{EC^{\otimes}} t'$.

Definition 34 For any relation R on TC^{\otimes} , behavioural R-equivalence is defined for $t, t' \in TC^{\otimes}$ by $t \tilde{R} t'$ iff $\phi(c)[t] R \phi(c)[t']$ for all $v \in VA$ and $c \in TA[z]_v^{\otimes}$.

Two natural choices for R in this definition are $=_{EC^{\otimes}}$ and \simeq . The first is sufficient for the examples given below, but the second is more general. Each choice satisfies condition (1.8):

Proposition 35 When R is $=_{EC^{\otimes}}$ or \simeq , then $=_{EC^{\otimes}} \subseteq \widetilde{R}$.

We note that behavioural $=_{EC^{\otimes}}$ -equivalence is the same as $=_{EC^{\otimes}}$ for visible sorts, because if t and t' are of visible sort, then we may take c to be the empty context, that is, c = z, so that $t =_{EC^{\otimes}} t'$.

In the sequel, we use only behavioural \simeq -equivalence, which we denote \approx , and refer to simply as **behavioural equivalence** i.e.,

(1.10) $t \approx t'$ iff $(\forall v \in V)(\forall c \in TA[z]_v^{\otimes}) \phi(c)[t] \simeq \phi(c)[t']$.

However, the results of this section can equally well be developed for behavioural $=_{EC\otimes}$ -equivalence.

The reader may check that \approx satisfies all requirements of Proposition 31 except (1.6). From Propositions 32 and 35, we obtain:

Corollary 36 $\phi: A \sqsubseteq C$ if all equations of EA are behaviourally satisfied by C, i.e., if $\bar{\theta}(lhs) \approx^{\phi} \bar{\theta}(rhs)$ for each $(\forall X) lhs = rhs$ in EA and $\theta: X \to TA^{\otimes}$.

This result can still lead to complicated proofs by context induction. A simpler proof method is obtained by splitting the signature of A^{\otimes} in two: suppose that $\Sigma A^{\otimes} = G \cup D$. (The letters stand for 'Generators' and 'Defined functions' to suggest the decomposition that we have in mind; however, we make no assumptions about G or D.) Typically, in proving that an equation is behaviourally satisfied, we wish to show that it holds in contexts made from defined functions only. This agrees with the intuition behind behavioural equivalence, that two terms are behaviourally equivalent if the same visible information can be extracted from each of them. Extracting information corresponds to applying a defined function, whereas constructors may be thought of as adding new information. This gives a notion of behavioural equivalence that is easier to check:

Definition 37 For $t, t' \in TC^{\otimes}$, we define $t \smile t'$ iff $\phi(c)[t] \simeq \phi(c)[t']$ for all $v \in VA$ and $c \in TD[z]_v$, where $TD[z]_v$ denotes the set $T_D[z]_v$ of contexts built from the operations of D.

A useful consequence of this definition is that terms of hidden sort are behaviourally equivalent iff their images under each operation of D are behaviourally equivalent. This is used in Subsections 1.3.2 and 1.3.3, in examples where all derived functions are unary, an assumption that allows us to state the property concisely:

Proposition 38 If all the operations of D have only one argument, then for $h \in SA - VA$ and $t, t' \in TC_h^{\otimes}$, we have $t \smile t'$ iff $(\phi_2 f)(t) \smile (\phi_2 f)(t')$ for each $r \in SA$ and $f \in D_{h,r}$.

The relation \smile is an equivalence relation, it contains $=_{EC^{\otimes}}$, and its restriction to visible sorts is the same as behavioural equivalence; moreover, \smile^{ϕ} is a *D*-congruence, so to use Proposition 32, we need only show that it is also a *G*-congruence. In fact, there is a nice relationship between our two notions of behavioural equivalence: from $D \subseteq \Sigma A^{\otimes}$, it follows that $\approx \subseteq \smile$; moreover, if \smile is also a *G*-congruence then the following proposition shows that $\smile \subseteq \approx$, and so $\smile = \approx$.

Proposition 39 $\smile = \approx$ if $t \smile t'$ implies $(\phi_2 f)(x) \smile (\phi_2 f)(y)$ for all $f \in G_{l,s}$ and $t, t' \in TC^{\otimes}_{\phi^*_1(l)}$.

Proof: We have already noted that $\smile \supseteq \approx$, so it suffices to show that $\smile \subseteq \approx$. Now \smile^{ϕ} is a *D*-congruence, so if it is also a *G*-congruence (as stated in the condition above), then because $\Sigma A^{\otimes} = G \cup D$, it is a ΣA^{\otimes} -congruence. So:

$$\begin{array}{l} t \smile t' \\ \Rightarrow \qquad \{ \smile^{\phi} \text{ is a congruence } \} \\ (\forall v \in VA)(\forall c \in TA[z]_v^{\otimes}) \ \phi(c)[t] \smile \phi(c)[t'] \\ \Rightarrow \qquad \{ \smile|_{VA} \subseteq \simeq \} \\ (\forall v \in VA)(\forall c \in TA[z]_v^{\otimes}) \ \phi(c)[t] \simeq \phi(c)[t'] \\ \Leftrightarrow \qquad \{ (1.10) \} \\ t \approx t' \end{array}$$

Corollary 36 and Proposition 39 together give the following sufficient condition for implementation:

Proposition 40 $\phi: A \sqsubseteq C$ if $\overline{\theta}(lhs) \smile^{\phi} \overline{\theta}(rhs)$ for each $(\forall X) lhs = rhs$ in EA and $\theta: X \rightarrow TA^{\otimes}$, and if $t \smile t'$ implies $(\phi_2 f)(t) \smile (\phi_2 f)(t')$ for all $f \in G_{l,s}$ and $t, t' \in TC^{\otimes}_{\phi^*_*(l)}$.

The following subsection shows that this proposition is especially useful when the operations of D are defined by structural induction over terms of G, because the second condition of the proposition then follows straightforwardly from the first.

1.3.2 A stack object

We now give an example proof of partial implementation using the technique developed in the previous subsection. The abstract specification defines a sort of stacks; a subsort relation makes operations top and pop defined only on non-empty stacks. The concrete specification implements stacks by means of arrays and pointers. This example, adapted from [4], is well-known, but we present it here to demonstrate that the proof we give is every bit as trivial as one could hope (cf. the statement in [3] that 'putting context induction into practise was less straightforward than expected').

The OBJ code which defines the abstract specification of stacks is given in the following two modules:

```
obj NAT is
                              obj STACK is pr NAT .
  sort Nat .
                                 sorts NeStack Stack .
  op 0 : -> Nat .
                                subsort NeStack < Stack .</pre>
  op
    s : Nat -> Nat .
                                op
                                     empty : -> Stack .
     p : Nat -> Nat .
                                op push : Nat Stack -> NeStack .
  ор
  var N : Nat .
                                    top_ : NeStack -> Stack .
                                 ор
  eq p(0) = 0.
                                 op pop_ : NeStack -> Stack .
  eq p(s(N)) = N.
                                                  var I : Nat .
                                 var S : Stack .
endo
                                 eq top push(I,S) = I.
                                 eq pop push(I,S) =
                                                      S .
                               endo
```

The OBJ keyword **sort** precedes the declaration of a sort name, and the keyword **op** precedes the declaration of an operation name; these declarations define the signature of the module. Equations are preceded by the keyword **eq**; these and the signature constitute the specification of the module. The keyword **pr** (for 'protecting') indicates that one module inherits the declarations of another; thus the module **STACK** contains all the declarations of the module **NAT**.

In order to demonstrate the use of signature morphisms in implementation, we give a concrete implementation of stacks using arrays and pointers that does not distinguish a subsort of non-empty stacks. The OBJ code for the concrete specification is given below:

```
obj ARR is pr NAT .
sort Arr .
op nil : -> Arr .
op put : Nat Arr Nat -> Arr .
op _[_] : Arr Nat -> Nat .
var I M N : Nat . var A : Arr .
eq nil[N] = 0 .
eq put(I,A,M)[N] = if M == N then I else A[N] fi .
endo
obj STACK is pr ARR .
```

```
sort Stack .
 op <<_;_>> : Nat Arr -> Stack .
 op 1st_ : Stack -> Nat .
 op 2nd_ : Stack -> Arr .
 op empty : -> Stack .
     push : Nat Stack -> Stack .
 ор
 op top_ : Stack -> Nat .
 op pop_ : Stack -> Stack .
 var I N : Nat . var S : Stack . var A : Arr .
 eq 1st \langle N ; A \rangle = N.
     2nd \ll N; A \gg = A.
 eq
 eq empty = << 0 ; nil >> .
 eq push(I,S) = << s(1st S) ; put(I, 2nd S, s(1st S)) >> .
 eq top S = (2nd S) [1st S].
 eq pop S = \langle p(1st S) ; 2nd S \rangle.
endo
```

The signature morphism ϕ from the abstract to the concrete specification maps both NeStack and Stack to the single sort Stack, and leaves the names of the operations unchanged. Note that the types of the operations *are* changed, because ϕ identifies NeStack and Stack. Specifically, ϕ is defined as follows.

If we let ΣA denote the signature of the abstract module, then ΣA^{\otimes} also contains the retract operation

```
r<sub>NeStack,Stack</sub> : Stack -> NeStack .
```

Because ϕ identifies Stack and NeStack, this operation is mapped to (cf. Proposition 26) the operation

```
rStack,Stack : Stack -> Stack .
```

But by Definition 20, the retract extension of the concrete specification includes the equation ($\forall S : Stack$) $r_{Stack,Stack}(S) = S$, which means that $r_{Stack,Stack}$ is the identity function in the concrete specification, so we may safely ignore retracts in what follows. Moreover, since the names of the operations are unchanged by this mapping, we can denote the ϕ -translation of a term by the term itself.

We now prove that the implementation of STACK is a partial behavioural implementation, where the set of visible sorts is $\{Nat\}$. For G, the set of generators, we take $\{empty, push\}$; for D, the set of defined functions, we take $\{top, pop\}$.

By Proposition 40, there are two proof obligations. The first is that the left- and right-hand sides of each equation are related by \smile :

(1.11) top push(I,S) \smile I

(1.12) pop push(I,S) \smile S

The second proof obligation is that \smile is preserved by the operations of G. Since empty is a constant and \smile is reflexive, we need only consider push:

(1.13) $x1 \smile x2$ and $s1 \smile s2$ imply $push(x1, s1) \smile push(x2, s2)$.

Requirement (1.11) is trivial, since the left-hand side is equal, in the concrete specification, to I. To show (1.12) and (1.13), we use the following:

Lemma 41 << 1st s; put(x,2nd s,n) >> \smile s if for all $i \ge 0$ it is not the case that $p^i(1st s) =_{EC^{\otimes}} n$ (i.e., if n > 1st s).

This can be proved by induction on the structure of contexts; only contexts built from top and pop need be considered.

Now, to show (1.12): note that pop push(I,S) is equal in the concrete specification to << 1st S ; put(I, 2nd S, s(1st S)) >> and by Lemma 41, this is related to S.

Similarly, (1.13) is demonstrated as follows:

This concludes the proof of partial implementation. Lemma 41, which relates **pop push(I,S)** to **S**, is the only part of the proof that is not extremely trivial: the remainder of the proof consists of rewriting terms by using equations.

1.3.3 Several stack objects

Hidden sorted specification is well suited to the object paradigm because objects may be thought of as automata with hidden local states, whose behaviour is observable only through their visible inputs and outputs. The object oriented language FOOPS [10] distinguishes between sorts and classes: the former refer to abstract data types; the latter to abstract object classes. Thus, a FOOPS specification distinguishes between hidden sorts for classes, and visible data sorts. A class of objects is specified by declaring some *methods*, operations that modify the state of an object, and some *attributes*, which give access to parts of an object's state. A method is typically defined by equations which state how that method modifies an object's attributes. Our proof technique is particularly useful in this context because the operations in a FOOPS specification are divided into methods and attributes, which correspond to generators and defined functions. In the following example, we do not give all formal details, but rather the broad outlines of the proof. In particular, we do not consider order sortedness.

The abstract specification (adapted from [10]) describes a class **Stackvar** of stack variables. The signature comprises that of **NAT**, as in the previous subsection, the class **Stackvar**, and the following operations:

```
me push : Nat Stackvar -> Stackvar .
me pop : Stackvar -> Stackvar .
at top : Stackvar -> Nat .
at rest : Stackvar -> Stackvar .
```

The FOOPS keyword 'me' declares a method; 'at' an attribute. The attribute rest is intended to represent the 'tail' of a stack variable. Note that this attribute has object values: one may think of stack variables as linked lists, whose state consists of a natural number (its top), and a pointer to another stack variable (its rest).

The methods push and pop are defined by the following equations, where N is a variable ranging over Nat, and SV is a variable ranging over Stackvar:

```
top pop SV = top rest SV .
rest pop SV = rest rest SV .
top push(N,SV) = N .
rest push(N,SV) = SV ! .
```

The postfix operation ! in the last equation is a polymorphic operation that exists for all FOOPS classes. Its operational semantics is that SV ! creates a copy of the object SV that has the same attributes. That is, for any attribute **a** and object **o**, we have $\mathbf{a}(\mathbf{o} \ !) = \mathbf{a}(\mathbf{o})$.

We show that this specification is implemented by a concrete specification which uses the abstract data type of stacks as defined in the previous subsection (though, for the sake of simplicity, we ignore its order sorted aspects). The concrete specification comprises the class name **Stackvar**, and two operations, one which assigns a value to a stack variable, and one which gives the value held by a stack variable:

me _:=_ : Stackvar Stack -> Stackvar .
at val_ : Stackvar -> Stack .

The assignment method (:=) is defined by the following equation, where SV is a variable ranging over Stackvar, and S is a variable ranging over the sort Stack:

val (SV := S) = S.

Thus stack variables in the concrete specification may be thought of as cells which hold values of sort Stack.

The concrete implementation of the methods push and pop, and attributes top and rest, is defined by the following equations.

```
push(N,SV) = SV := push(N, val SV) .
pop SV = SV := pop val SV .
top SV = top val SV .
rest SV = SV ! := pop val SV .
```

The operations push, etc., in the right-hand sides of these equations are the operations from STACK. The last equation perhaps requires some explanation. In the abstract specification, the attribute **rest** returns an object that is different from its argument (hence '!'), with value the 'tail' of its argument (hence 'pop').

The visible equations of the abstract specification hold in the concrete as a result of these equations, so a proof of partial implementation need only consider the hidden equations:

```
rest pop SV = rest rest SV .
rest push(N,SV) = SV ! .
```

We use Proposition 40, with generators $G = \{\texttt{push}, \texttt{pop}\}\)$ and defined functions $D = \{\texttt{top}, \texttt{rest}\}\)$. This division is natural, because G contains all the methods of the abstract specification, and D all the attributes. The proof obligations are:

We use the following lemma.

Lemma 42 If val SV1 = val SV2 then SV1 \smile SV2.

This lemma can be proved by induction on the structure of contexts built from D: since D contains only two operations, there are only two cases to consider.

Now (1.14) and (1.15) are easy consequences. To show (1.16):

The last step uses the fact that $SV := val SV' \cup SV'$, which is a consequence of Lemma 42. Finally, (1.17) follows straightforwardly from Proposition 38 and (1.14); we conclude that the partial implementation is correct.

1.4 Conclusion

We have given a definition of implementation for hidden order sorted specifications, and a technique for proving correctness of partial implementation by proving behavioural satisfaction of equations in the concrete specification. This technique leads to proofs based on term rewriting which seem much simpler than other proofs in the literature. Our approach is directly applicable to the object paradigm by associating visible sorts with data types, and hidden sorts with object classes.

Hidden sorted algebra leads to an abstract treatment of states of objects, and to a similarly abstract treatment of object implementation. The treatment of object implementation given by Costa *et al* [2] uses a concrete description of state in object specifications. Showing correctness of object implementation then requires a mapping from the states of the one object to the states of the other. In contrast, hidden sorted algebra provides a unified treatment of states, abstract data types and behaviour, abstracting away from details of how states are represented.

One question not addressed in this paper is concurrency. Hidden sorted specifications can be thought of as specifying networks of concurrent, interacting objects. Our approach to implementation is obviously applicable to serial evaluation by term rewriting (as in OBJ), but less obviously to concurrent models of computation. Goguen and Diaconescu [8] give a construction for the concurrent interconnection of collections of objects, and show how such interconnections can be enriched with interactions between component objects. We hope to develop a sheaf-theoretic semantics for FOOPS objects (as in [6]) which addresses such issues and extends our notion of implementation to concurrent, interacting systems.

A cknowledgements

The research reported in this paper has been supported in part by grants from the Science and Engineering Research Council, ESPRIT Working Group 6071, IS-CORE, and Fujitsu Laboratories Limited, and a contract with the Information Technology Promotion Agency, Japan, as part of the R & D of Basic Technology for Future Industries "New Models for Software Architecture" project sponsored by NEDO (New Energy and Industrial Technology Development Organization).

References

- Rod Burstall and Răzvan Diaconescu. Hiding and Behaviour: an Institutional Approach. This volume.
- [2] José Felix Costa, Amilcar Sernadas, and Cristina Sernadas. Inductive objects. INESC, Lisbon, 1992.
- [3] Marie Claude Gaudel and I. Privara. Context induction: an exercise. Technical Report 687, LRI, Univ. Paris Sud, 1991.
- [4] Joseph Goguen. An algebraic approach to refinement. In Dines Bjorner, C.A.R. Hoare, and Hans Langmaack, editors, Proceedings, VDM'90: VDM and Z - Formal Methods in Software Development, pages 12-28. Springer, 1990. Lecture Notes in Computer Science, Volume 428.
- [5] Joseph Goguen. Types as theories. In George Michael Reed, Andrew William Roscoe, and Ralph F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357-390. Oxford, 1991. Proceedings of a Conference held at Oxford, June 1989.
- [6] Joseph Goguen. Sheaf semantics for concurrent interacting objects. Mathematical Structures in Computer Science, 11:159-191, 1992. Given as lecture at Engeler Festschrift, Zürich, 7 March 1989, and at U.K.-Japan Symposium on Concurrency, Oxford, September 1989; draft as Report CSLI-91-155, Center for the Study of Language and Information, Stanford University, June 1991.
- [7] Joseph Goguen and Răzvan Diaconescu. A survey of order sorted algebra, 1992. Submitted to Mathematical Structures in Computer Science.
- [8] Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Proceedings, Tenth Workshop on Abstract Data Types. Springer, to appear 1993.
- [9] Joseph Goguen and José Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In M. Nielsen and E.M. Schmidt, editors, Proceedings, 9th International Conference on Automata, Languages and Programming, pages 265-281. Springer, 1982. Lecture Notes in Computer Science, Volume 140.
- [10] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, Research Directions in Object-Oriented Programming, pages 417-477. MIT, 1987.

- 24 Grant Malcolm and Joseph A. Goguen
- [11] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217-273, 1992.
- [12] Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. Technical Report RC 6487, IBM T.J. Watson Research Center, October 1976. In Current Trends in Programming Methodology, IV, Raymond Yeh, editor, Prentice-Hall, 1978, pages 80-149.
- [13] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, Applications of Algebraic Specification using OBJ. Cambridge, to appear 1993. Also to appear as Technical Report from SRI International.
- [14] Rolf Henniker. Context induction: a proof principle for behavioural abstractions. In A. Miola, editor, *Design and Implementation of Symbolic Computation Systems*. Springer-Verlag Lecture Notes in Computer Science 429, 1990.
- [15] C.A.R. Hoare. Proof of correctness of data representations. Acta Informatica, 1:271-281, 1972.
- [16] Lucia Rapanotti and Adolfo Socorro. Introducing FOOPS. Oxford University Computing Laboratory, 1992.
- [17] Oliver Schoett. Behavioural correctness of data representations. Science of Computer Programming, pages 43-57, 1990.