PROVING CORRECTNESS OF REFINEMENT AND IMPLEMENTATION

by

Grant Malcolm and Joseph A. Goguen

Technical Monograph PRG-114

November 1994

Oxford University Computing Laboratory
Programming Research Group
Wolfson Building
Parks Road
Oxford OX1 3QD

Oxford University Computing Laboratory
Programming Research Group
Wolfson Building
Parks Road
Oxford OX1 3QD

Electronic mail:
`Grant.Malcolm@comlab.ox.ac.uk`
`Joseph.Goguen@comlab.ox.ac.uk`

Proving Correctness of Refinement and Implementation

Grant Malcolm
Joseph A. Goguen
Programming Research Group
University of Oxford
Oxford, U.K.

Abstract

The notions of state and observable behaviour are fundamental to many areas of computer science. Hidden sorted algebra, an extension of many sorted algebra, captures these notions through hidden sorts and the behavioural satisfaction of equations. This makes it a powerful formalisation of abstract machines, and many results suggest that it is also suitable for the semantics of the object paradigm. Another extension of many sorted algebra, namely order sorted algebra, has proved useful in system specification and prototyping because of the way it handles subtypes and errors. The combination of these two algebraic approaches, hidden order sorted algebra, has also been proposed as a foundation for object paradigm, and has much promise as a foundation for Software Engineering.

This paper extends recent work on hidden order sorted algebra by investigating the refinement and implementation of hidden order sorted specifications. We present definitions of refinement and implementation for such specifications, and techniques for proving that one specification refines or implements another. It is important that the notions of refinement and implementation be tractable, in the sense that there are efficient techniques for proving their correctness. The proof techniques given in this paper lead, we believe, to correctness proofs that are much simpler than others in the literature. We found that proving refinement is an effective way to prove implementation correctness. Some examples are given.

Any foundation for the semantics of programming should also support modular specifications. The 'institutions' developed by Goguen and Burstall are useful for this purpose. Institutions formalise the notion of logical system, and provide an encapsulation property for specifications: when one specification is imported into another, properties that hold of that specification in isolation remain true in its new context. An important technical result of this paper is that hidden order sorted algebra forms an institution, and therefore supports the modular specification of systems of objects. The paper also includes an exposition of hidden order sorted algebra, and brief introductions to many sorted algebra, order sorted algebra, and institutions.

Contents

1 Introduction

This paper presents an algebraic account of refinement and implementation that is applicable to a very general notion of abstract machine; in particular, it is applicable to the object paradigm. We define notions of refinement and implementation of specifications of observed hebaviours, and develop a proof technique that supports simple and direct proofs of correctness.

A key notion in the object paradigm is that 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 [17]. Accordingly, a correct refinement or implementation of an object specification need only have the required *visible* behaviour.

Our approach uses hidden order sorted algebra, which can be seen as a formalisation of the object paradigm [9, 10, 15]. The advantages of an algebraic approach include a high level of intellectual rigour, a large body of supporting mathematics, and simple, efficient proofs using equational logic. A wide variety of extensions to equational logic have been developed to treat various programming features while preserviug its essential simplicity. The particular extension considered in this paper combines order sorted and hidden sorted equational logics. Order sorted equational logic uses a notion of subsort to treat computations that may raise exceptions or fail to terminate, and hidden sorted logic extends standard equational logic to capture the important distinction between *immutable data types*, such as booleans and integers, and *mutable objects*, such as program variables and datahase entities. The terms *abstract data type* and *abstract object class* refer to these two kinds of entity. The former represeut 'visible' data values, while 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 refinements and implementations.

The simplicity of the underlying logic is important because we want a *tractable* approach in which refinements and implementations are as easily expressed and proved as possible. Both refinement and implementation involve moving from an abstract specification to a more concrete specification which displays the same behaviour. In our approach, a specification has syntactic and semantic components: the syntactic part declares a number of operations, which may manipulate either data or object states, while the semantic part consists of a set of equations which describe the behaviour of the declared operations. In other words, a specification is a *theory*. A model of a such a specification is something which implements the declared operations in such a way that the given equations are satisfied (up to observability). A refinement is expressed by a mapping from the syntax of the abstract specification to the syntax of the concrete specification; the refinemeut is correct if every model (i.e., implementation) of the concrete specification gives rise to a model of the abstract specification. We present a proof technique for showing correctness of refinement, which is based on proving that the equations of the abstract specification are satisfied up to observability in the concrete specification. Moreover, our definition of refinement is transitive in the sense that if specification S_1 is refined by S_2 and S_2 is refined by S_3, then S_1 is refined by S_3: this allows the familiar process of 'stepwise refinement', where a refinement is arrived at via a number of intermediate specifications, proceeding in small, manageable steps.

Our notion of implementation, on the other hand, is more concrete: we require the concrete specification to satisfy (again, up to observability) only the *ground equations* (i.e., equations with no variables) of the abstract specification. This makes sense if we consider terms built from the operations of the abstract specification to be programs, because we would not expect programs to contain variables. This definition of implementation would be a specialisation of the definition of refinement, in the sense that all *reachable* models of the concrete specification (i.e., models whose carriers are generated by the operations of the signature) give rise to models of the abstract specification. However, we illustrate the power of order sortedness by defining a notion of 'lazy' implementation, whereby we further require that the error-handling behaviour of the abstract specification is also captured by the concrete specification. This is treated in detail in Section 3.3. The proof technique that we give for showing correctness of refinement also applies to correctness of implementation.

In a model-based approach, like that of Hoare [25], where refinement is a relationship between particular models, it makes sense to map from the concrete variables to the abstract objects that they represent. However, such an approach has difficulties with the (often) complex representations of the concrete program, and the (usually) complex semantics of the programming language in which it is expressed. Our approach simplifies the first problem by considering *theories* for both the concrete and the abstract levels, while the complexity of the programming language semantics becomes a completely separate issue. In particular, our more abstract definition of refinement in terms of specifications and their models allows the process of stepwise refinement to begin before any concrete representation for variables has been chosen. In fact, choosing a concrete representation corresponds to choosing one particular model of a specification: it makes sense to delay such a commitment as long as possible. The correctness of a concrete representation now becomes the problem of showing it to be a model of the concrete theory, which should be much easier than showing that it satisfies the abstract specification, because of the closer match of representations. The perhaps initially mysterious fact that mappings go in opposite directions for specifications and models is explained at a higher level of abstraction by the theory of institutions [13], which is briefly discussed in Section 2.3. Hence this duality is very natural.

Finally, note that our use of hidden sorts allows some subtle changes of representation to be proved correct more easily. Indeed, the main motivation for our approach is to make proofs of correctness just as easy as possible.

This paper is organised as follows. The next section introduces notation for hidden order sorted specifications, and summarises the main algebraic notions and results used in this paper. Section 3 presents refinements and 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 4 gives examples of correctness proofs. In particular, Section 4.2 applies this technique to the refinement of collections of objects.

Acknowledgements

This paper is dedicated with warm affection to Tony Hoare, whose work on data representation and on concurrency has been an inspiration to us. We thank him for his encouragement and questions about the motivation of our work and its relationship to model-based approaches. We also thank Răzvan Diaconescu for his many helpful technical comments on this paper.

The research reported in this paper has been supported in part by the Science and Engineering Research Council, the CEC under ESPRIT-2 BRA Working Groups 6071, IS-CORE (Information Systems COrrectness and REusability) and 6112, COMPASS (COMPrehensive Algebraic Approach to System Specification and development), Fujitsu Laboratories Limited, and a contract under the management of the Information Technology Promotion Agency

(IPA), Japan. as part of the Industrial Science and Technology Frontier Program "New Models for Software Architectures," sponsored by NEDO (New Energy and Industrial Technology Development Organization).

2 Hidden order sorted algebra

Many sorted algebra (hereafter, 'MSA') was put into a form that is convenient for applications in Computing Science by Goguen [8] and was further refined by the ADJ group [22] and applied to abstract data types and other topics. The logic of MSA is first order equational logic, which provides a simple and familiar technical framework in which intuitions about data types can be realised. Several variations on the basic framework have been developed, including order sorted algebra [14] and hidden sorted algebra [10], in order to study such concepts as error handling and hidden local state. The following section summarises the main definitions and results of MSA, while Sections 2.2 and 2.4 describe order sorted specification and hidden order sorted specification, respectively. Section 2.5 gives further technical details necessary for the definition of refinement given in Section 3.1.

2.1 Many sorted algebra

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

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

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.

The notion of *sorted set* is used to specify the names of the carriers of algebras.

Definition 1 Given a set S, an S**-sorted set** is a collection of sets A_s indexed by elements $s \in S$. 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** $f : A \rightarrow B$ is a collection of functions indexed by S such that $f_s : A_s \rightarrow B_s$ 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 $\{\mathtt{Elt}, \mathtt{List}\}$-sorted set.

The notion of sorted set is also useful in specifying the names and types of the operations of algebras. The following definition introduces *signatures*, which specify the carriers and operations of algebras; equations are considered from Definition 8 onwards.

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 $w \in S^*$ and $s \in S$ then $\Sigma_{w,s}$ is a set of operation names. If Σ is clear from the context, we sometimes write $\sigma : w \to s$ instead of $\sigma \in \Sigma_{w,s}$ to emphasise that σ is intended to denote an operation mapping the sorts denoted by w to the sort denoted by s. Usually we abbreviate (S, Σ) to Σ. Elements of $\Sigma_{[],s}$ are referred to as **constants** of sort s.

An operation can be declared to have more than one type; for example, we might have $\sigma \in \Sigma_{w,s} \cap \Sigma_{w',s'}$ where w, s is different from w', s'. In this case, σ is said to be **overloaded.** □

Signatures provide a uniform notation for specifying the carriers and operations of many sorted algebras. Later sections consider refining 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 (f, g), where $f : S \to S'$ maps sorts in S to sorts in S', and g is a collection of functions indexed by $S^* \times S$ such that $g_{w,s} : \Sigma_{w,s} \to \Sigma'_{f^*(w), f(s)}$ for each $w, s \in S^* \times S$, where $f^*(w)$ denotes f applied componentwise to the list w; i.e., $f^*([]) = []$ and $f^*(s\,w) = (f(s))(f^*(w))$. We usually write ϕ instead of both f and $g_{w,s}$, so that if $\sigma \in \Sigma_{w,s}$, then $\phi(\sigma) \in \Sigma'_{\phi^*(w), \phi(s)}$. □

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 $A_\sigma \in A_s$ for each $s \in S$ and $\sigma \in \Sigma_{[],s}$; and for each non-empty list $w \in S^*$, and each $s \in S$ and $\sigma \in \Sigma_{w,s}$, an operation $A_\sigma : A_w \to A_s$, where if $w = s1 \ldots sn$ then $A_w = A_{s1} \times \cdots \times A_{sn}$.

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 $\sigma \in \Sigma_{[],s}$, then $h_s(A_\sigma) = B_\sigma$;
- given a non-empty list $w = s1 \ldots sn$ and $\sigma \in \Sigma_{w,s}$ and $ai \in A_{si}$ for $i = 1, \ldots, n$, then

$$h_s(A_\sigma(a1, \ldots, an)) \;=\; B_\sigma(h_{s1}(a1), \ldots, h_{sn}(an)).$$

□

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; then T_Σ is the least S-sorted set of strings over the alphabet $(\cup\Sigma) \cup \{\underline{(}, \underline{)}\}$ such that:

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

The special symbols '$\underline{(}$' and '$\underline{)}$' are used to emphasise that the carriers of T_Σ are sets of strings; from now on we usually write '$\sigma(t1, \ldots, tn)$' for '$\sigma\underline{(}t1 \ldots tn\underline{)}$'.

We give T_Σ the structure of a Σ-algebra by interpreting each operation name of Σ: for each $\sigma \in \Sigma_{[],s}$, the constant $(T_\Sigma)_f$ is the string $\sigma \in (T_\Sigma)_s$; for each non-empty list $w = s1 \ldots sn \in S^*$ and operation name $\sigma \in \Sigma_{w,s}$, the operation $(T_\Sigma)_\sigma : (T_\Sigma)_w \to (T_\Sigma)_s$ maps a tuple of strings $t1, \ldots, tn$ to the string $\sigma(t1, \ldots, tn)$. □

If Σ contains no overloaded symbols, then T_Σ 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 $\sigma \in \Sigma_{[],s}$, let $h_s(\sigma) = A_\sigma$;
- for each non-empty list $w = s1 \ldots sn$ and $\sigma \in \Sigma_{w,s}$ and $ti \in (T_\Sigma)_{si}$ for $i = 1, \ldots, n$, let

$$h_s(\sigma(t1, \ldots, tn)) \;=\; (A_\sigma)(h_{s1}(t1), \ldots, h_{sn}(tn)) \;.$$

□

The homomorphism h assigns values in A to Σ-terms by interpreting the operation names in Σ as the corresponding operations on A. If Σ contains overloaded operations, an initial algebra can still be constructed as a term algebra in which operation names are distinguished by 'tagging' them with their result sorts [14].

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 $C_{\mathtt{List}}$. 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 $w \in S^*$ and $s \in S$, if $w \neq []$ then $\Sigma_{w,s} = \emptyset$, and such that the sets $\Sigma_{w,s}$ are disjoint; that is, the operation names of ground signatures are distinct constants. □

We assume disjointness so that distinct variables cannot be identified by signature morphisms (cf. Proposition 45 below).

Ground signatures are essentially the same thing as disjoint S-sorted sets, because any disjoint S-sorted set X can be viewed as a ground signature according to the following equation:

$$X_{w,s} = \begin{cases} X_s & \text{if } w = [] \\ \emptyset & \text{otherwise.} \end{cases}$$

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 sometimes take advantage of this by treating ground signatures as S-sorted sets.

It is now a simple matter to introduce 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 $T_{\Sigma\cup X}$. The term algebra $T_{\Sigma\cup X}$ 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 $T_\Sigma(X)$. □

This prepares us for the following

Definition 10 A **Σ-equation** is a triple (X, l, r) where (S, X) is a ground signature and l and r are terms in $T_\Sigma(X)$ of the same sort; i.e., $l, r \in T_\Sigma(X)_s$ for some $s \in S$. If $X = \emptyset$, i.e., if l and r contain no variables, then we say that the equation is **ground**. We write equations 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 just (Σ, E). □

Our notation for equations makes explicit the intended universal quantification over variables. Wothout this, the usual rules of deduction from uusortes equational logic are unsound, as shown in [18].

The models of a specification are the Σ-algebras that satisfy the equations; we now consider 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. A Σ-algebra is not in general a $(\Sigma\cup X)$-algebra, because we do not know how to interpret the variables in X. However, if we can assign values to those variables, then we can assign values to terms containing those variables. That is the main idea of the following result:

Proposition 11 Given a Σ-algebra A and an S-sorted function $\theta : X \to A$ (called an **assignment** or 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 strng $x \in T_\Sigma(X)_s$. The homomorphism is defined as follows:

- for each $x \in X_s$, let $\bar\theta_s(x) = \theta_s(x)$;
- for each constant symbol $\sigma \in \Sigma_{[],s}$, let $\bar\theta_s(\sigma) = A_\sigma$;
- for each non-empty list $w = s1 \ldots sn$, $\sigma \in \Sigma_{w,s}$, and $ti \in T_\Sigma(X)_{si}$ for $i = 1, \ldots, n$, let

$$\bar\theta_s(\sigma(t1, \ldots, tn)) \;=\; A_\sigma(\bar\theta_{s1}(t1), \ldots, \bar\theta_{sn}(tn))\ .$$

□

The following little result is used several times in this paper; its proof is left as an exercise in initiality.

Lemma 12 Given Σ-algebras A and B, a Σ-homomorphism $h : A \to B$, and an assignment $\theta : X \to A$, then

$$\overline{(h\circ\theta)} \;=\; h \circ \bar\theta$$

□

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 13 A Σ-algebra A **satisfies** a Σ-equation $(\forall X)\ l = r$ iff $\bar{\theta}(l) = \bar{\theta}(r)$ 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 $e \in E$, and we write $E \models e$ iff $A \models E$ implies $A \models e$ for all Σ-algebras A.

Given a specification (Σ, E), a (Σ, E)**-model** is a Σ-algebra A such that $A \models E$. □

Just as each signature has an initial algebra, each specification has an initial model. The initial model is constructed from the term algebra by identifying exactly those terms that are 'equal' as a consequence of the given equations.

Each equation gives rise to a relation in the following way:

Definition 14 Given a Σ-algebra A and a Σ-equation e of the form $(\forall X)\ l = r$, define the relation $\mathsf{R}_A(e)$ on A by $a\ \mathsf{R}_A(e)\ b$ iff $a = \bar{\theta}(l)$ and $b = \bar{\theta}(r)$ for some $\theta : X \to A$. □

In other words, a is related to b by $\mathsf{R}_A(e)$ 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 will use this in defining an equivalence relation that contains all the relations derived from the equations of a specification, and that allows the substitution of equals for equals. This is formalised by the notion of *congruence* given below:

Definition 15 Given a Σ-algebra A, a Σ**-congruence** on A is an S-sorted equivalence relation R such that the following **substitutivity property** holds: for all $\sigma \in \Sigma_{w,s}$ and $x, y \in A_w$, if $x\ R_w\ y$ then $A_\sigma(x)\ R_s\ A_\sigma(y)$, where if $w = s1 \dots sn$, then $x \in A_w$ means $x = x1 \dots xn$ with $xi \in A_{si}$, and $x\ R_w\ y$ 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, such that $\mathsf{R}_A(e) \subseteq \equiv_{A,E}$ for each $e \in E$. We usually write $=_E$ instead of $\equiv_{T_\Sigma,E}$. □

Definition 16 Given a Σ-algebra A and a Σ-congruence R on A, we construct the Σ-algebra A/R, called the **quotient of A by R**, as follows:

- for $s \in S$, let $(A/R)_s = \{[a] | a \in A_s\}$, where $[a]$ is the equivalence class of a under R (i.e., the set of x such that $a\,R\,x$);
- for each constant symbol $\sigma \in \Sigma_{[],s}$, let $(A/R)_\sigma = [A_\sigma]$;
- for each non-empty list $w = s1 \dots sn$, $\sigma \in \Sigma_{w,s}$, and $[ai] \in (A/R)_{si}$ for $i = 1, \dots, n$, let

$$(A/R)_\sigma([a1], \dots, [an]) = [(A/R)_\sigma(a1, \dots, an)] .$$

The last equation is well-defined by the substitutivity property of the congruence R.

Given a set E of Σ-equations, we often write A/E instead of $A/\equiv_{A,E}$. □

By construction, $A/E \models E$; in fact, it is the 'least' (Σ, E)-model which can be constructed from A, in the sense of the following

Fact 17 Let A be a Σ-algebra, let E be a set of Σ-equations, and let $\eta : A \to A/E$ be the Σ-homomorphism which maps a to $[a]$. Then for any (Σ, E)-model B and any Σ-homomorphism $h : A \to B$, there is a unique Σ-homomorphism $h^\flat : A/E \to B$ such that $h^\flat \circ \eta = h$ (i.e., such that $h^\flat[a] = h(a)$). The situation is summarised in the following diagram.

$$\begin{tikzcd} A \arrow[r, "\eta"] \arrow[dr, "h"'] & A/E \arrow[d, "h^\flat"] \\ & B \end{tikzcd}$$

□

The Σ-congruence $=_E$ identifies those terms that are equal as a result of the equations in E, and allows the construction of an initial model for a given specification, as follows:

Proposition 18 Given a specification $(\Sigma. E)$ where Σ contains no overloaded operations, the initial (Σ, E)-model is the **quotient term algebra** $T_{\Sigma,E}$. which is the quotient $T_\Sigma/{=_E}$ of T_Σ by $=_E$. By construction, $T_{\Sigma,E}$ satisfies the equations E. □

The above proposition refers to 'the' initial (Σ, E)-model, although a specification may have more than one initial model. However, any two initial (Σ, E)-models are isomorphic, because the unique homomorphisms from each model to the other are inverses. Thus all initial models are 'abstractly the same'. ADJ [22] define an abstract data type to be the collection of initial models 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 explains what *is* the abstract data type of a specification.

There is a useful relationship between the congruences $\equiv_{A,E}$ and satisfaction of equations:

Proposition 19 Let E be a set of Σ-equations; then for all Σ-algebras A we have $A \models E$ iff $\equiv_{A,E} = id_A$.

Proof Note that $id_A \subseteq \equiv_{A,E}$ holds because $\equiv_{A,E}$ is a congruence. Because id_A is a congruence, it follows from Definition 15 that $\equiv_{A,E} \subseteq id_A$ iff $\mathsf{R}_A(e) \subseteq id_A$ for all $e \in E$. For any $e \in E$ of the form $(\forall X)\, l = r$, we have

$$\begin{array}{ll} & \mathsf{R}_A(e) \subseteq id_A \\ \Leftrightarrow & \\ & (\forall \theta : X \to A)\ \bar{\theta}(l) = \bar{\theta}(r) \\ \Leftrightarrow & \\ & A \models e \end{array}$$

Thus $\mathsf{R}_A(e) \subseteq id_A$ iff $A \models e$ for all $e \in E$, that is, iff $A \models E$. □

We also have the following relationship between congruences and entailment of equations.

Proposition 20 $E \models e$ iff $\equiv_{A,\{e\}} \subseteq \equiv_{A,E}$ for all Σ-algebras A.

Proof First note that by Proposition 19, $E \models e$ is equivalent to

(1) $\quad \equiv_{A,E} \subseteq id_A \quad$ implies $\quad \equiv_{A,\{e\}} \subseteq id_A \qquad$ for all Σ-algebras A.

This makes the 'if' direction immediate; we now show the 'only if' direction. Suppose $E \models e$; then because $A/E \models E$, we have $A/E \models e$. so by Fact 17 there is a homomorphism $\eta_E^\flat : A/\{e\} \to A/E$ such that $\eta_E^\flat \circ \eta_e = \eta_E$, where $\eta_e : A \to A/\{e\}$ takes $a \in A$ to its equivalence class modulo $\equiv_{A,\{e\}}$, and $\eta_E : A \to A/E$ takes $a \in A$ to its equivalence class modulo $\equiv_{A,E}$. This is summarised in the following diagram.

$$\begin{tikzcd}
A \arrow[r, "\eta_e"] \arrow[dr, "\eta_E"'] & A/\{e\} \arrow[d, "\eta_E^\flat"] \\
 & A/E
\end{tikzcd}$$

Now,

$$\begin{array}{ll}
 & a \equiv_{A,\{e\}} b \\
\Leftrightarrow & \\
 & \eta_e(a) = \eta_e(b) \\
\Rightarrow & \\
 & \eta_E^\flat(\eta_e(a)) = \eta_E^\flat(\eta_e(b)) \\
\Leftrightarrow & \\
 & \eta_E(a) = \eta_E(b) \\
\Leftrightarrow & \\
 & a \equiv_{A,E} b
\end{array}$$

which shows that $\equiv_{A,\{e\}} \subseteq \equiv_{A,E}$ as desired. □

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 valnes only on some subsort of its domain. For example, division of nnmbers 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 in which partial operations are treated as total operations on a subdomain, just as division is total on the subdomain of non-zero numbers. It also provides a model of inheritance that is useful in formalising the object paradigm. This subsectiou summarises definitions and results of OSA that are relevant to this paper. A comprehensive survey is given by Goguen and Diaconescu in [14].

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. Given 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 $s1 \ldots sn \leq s1' \ldots sn'$ iff $si \leq si'$ for $i = 1, \ldots, n$.

Definition 21 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 aud s' are connected, there is an s'' such that $s, s' \leq s''$. □

The notion of local filtering is surprisingly powerful, and allows many results of MSA to extend to OSA [14]. The main difference between MSA and OSA is captured in the following definition of sorted sets.

Definition 22 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*[1], to OSA; the main differences concern monotonicity.

Definition 23 An **order sorted signature** is a triple $(S, \leq, \Sigma)$ where $(S, \leq)$ is a locally filtered partial order and (S, Σ) is a many sorted signature which satisfies the **monotonicity requirement**: if $\sigma \in \Sigma_{w,s} \cap \Sigma_{w',s'}$ and $w \leq w'$ then $s \leq s'$. We usually abbreviate $(S, \leq, \Sigma)$ to just Σ.

An **order sorted signature morphism** $\phi : (S, \leq, \Sigma) \to (S', \leq', \Sigma')$ is a many sorted signature morphism such that $f : (S, \leq) \to (S', \leq')$ is monotonic. A signature morphism $\phi = (f, g)$ **preserves overloading** iff whenever $\sigma \in \Sigma_{w,s} \cap \Sigma_{w',s'}$ then $g_{w,s}$ applied to $\sigma \in \Sigma_{w,s}$ gives the same result as $g_{w',s'}$ applied to $\sigma \in \Sigma_{w',s'}$. □

A form of monotonicity is also needed for the algebras of an order sorted signature.

Definition 24 Given an order sorted signature $(S, \leq, \Sigma)$, an **order sorted Σ-algebra** is a many sorted Σ-algebra A such that A is an $(S, \leq)$-sorted set and A **is monotonic**, in the sense that for all $\sigma \in \Sigma_{w,s} \cap \Sigma_{w',s'}$ if $w \leq w'$ and $s \leq s'$ then $A_\sigma : A_w \to A_s$ is equal to $A_\sigma : A_{w'} \to A_{s'}$ on A_w.

For order sorted Σ-algebras A and B, an **order sorted Σ-homomorphism** $h : A \to B$ is a many sorted Σ-homomorphism (f, g) 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 [20]:

Definition 25 An order sorted signature Σ is **regular** iff for any $\sigma \in \Sigma_{w1,s1}$ and any $w0 \leq w1$ there is a least pair (w, s) such that $w0 \leq w$ and $\sigma \in \Sigma_{w,s}$. □

The importance of regularity is that terms can be parsed as having a least sort. Goguen and Diaconescu [14] 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 the 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.

[1]'S-sorted' should be changed to '$(S, \leq)$-sorted'.

Definition 26 Given an order sorted signature $(S, \leq, \Sigma)$, 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 specification** is an order sorted signature together with a set E of Σ-equations, and a **(Σ, E)-model** 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)-model [20].

We end our summary of OSA with 'retract specifications', which allow operations to be applied to argnments which may lie outside their domain of definitiou, 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 and error-handling (see [20, 14] for a full treatment).

Definition 27 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 $\Sigma^{\otimes}$ is Σ extended with a new operation $r_{s1,s2} : s1 {\to} s2$ for each $s1, s2 \in S$ such that $s2 \leq s1$, and $E^{\otimes}$ is E extended with an equation $(\forall S : s2)\ r_{s1,s2}(S) = S$ for each $s2 \leq s1$ as above. □

For example, consider a specification which declares a sort `Nat` of natural numbers to be a subsort of `Rat`, the rationals. This specification would be extented with a retract operation

$$r_{\mathtt{Rat},\mathtt{Nat}} : \mathtt{Rat} \to \mathtt{Nat}$$

and an equation

$$(\forall N : \mathtt{Nat})\ r_{\mathtt{Rat},\mathtt{Nat}}(N) \ = \ N\ .$$

If this specification also declares an operation `f` (think of factorial) which takes naturals as arguments, then $\mathtt{f}(6/3)$ is not a term of T_Σ; however, $\mathtt{f}(r_{\mathtt{Rat},\mathtt{Nat}}(6/3))$ is a term of $T_{\Sigma^{\otimes}}$; moreover, if the specification is such that $6/3 =_E 2$, then $\mathtt{f}(r_{\mathtt{Rat},\mathtt{Nat}}(6/3)) =_{E^{\otimes}} \mathtt{f}(2)$. Thus, if we consider the term $\mathtt{f}(r_{\mathtt{Rat},\mathtt{Nat}}(6/3))$ as a program, then we might say that it produces a 'well-defined' value, $\mathtt{f}(2)$, in the sense that this latter term contains no retracts. On the other hand, the term $\mathtt{f}(r_{\mathtt{Rat},\mathtt{Nat}}(5/3))$ does not produce a well-defined value, because it is not equal to any term of T_Σ; that is, every term equal to $\mathtt{f}(r_{\mathtt{Rat},\mathtt{Nat}}(5/3))$ contains the retract operation $r_{\mathtt{Rat},\mathtt{Nat}}$. We refer to such terms as **error terms**.

We wish the result of adding retracts to be a **conservative extension** of the given specification, in the sense that for all $t1, t2 \in T_\Sigma$, we have $t1 =_E t2$ iff $t1 =_{E^{\otimes}} t2$, i.e., the new equations added by introducing retracts do not cause distinct terms of T_Σ to become identified. Goguen and Meseguer [20] give sufficient conditions on specifications for adding retracts to be conservative. These conditions go beyond the scope of the present paper, hnt we note that all of our example specifications are such that their retract extensions are conservative.

Finally, we note that Definition 27 can be generalised in that every (Σ, E)-model can be freely extended to a $(\Sigma^{\otimes}, E^{\otimes})$-model. This extension will be used when we define satisfaction of eqnations in hidden order sorted algebra, and is defined as follows.

Definition 28 Let A be a (Σ, E)-model; the **free retract extension** of A, denoted $A^{\otimes}$, is defined to be the least S-sorted set such that:

(1) $A_s \subseteq A^{\otimes}_s$ for each $s \in S$;
(2) $\sigma(a) \in A^{\otimes}_s$ for each $\sigma \in \Sigma^{\otimes}_{w,s}$ and $a \in A^{\otimes}_w$;
(3) $\sigma(a) = A_\sigma(a)$ for each $\sigma \in \Sigma_{w,s}$ and $a \in A_w$;
(4) $r_{s1,s2}(a) = a$ for each $s1, s2$ such that $s2 \leq s1$ and $a \in A_{s2}$.

$A^{\otimes}$ is given the structure of a $\Sigma^{\otimes}$-algebra by defining $A^{\otimes}_\sigma(x) = \sigma(x)$ for each $\sigma \in \Sigma_{w,s}$ and $x \in A^{\otimes}_w$. □

For an arbitrary Σ-algebra A, $A^{\otimes}$ can be thought of as being built from $T_{\Sigma^{\otimes}}(A)$, i.e., terms with variables which are elements of A, and then quotiented by the equations given in (3) and (4) of the above definition. Moreover, $T^{\otimes}_{\Sigma} = T_{\Sigma^{\otimes}}$.

Because $\Sigma \subseteq \Sigma^{\otimes}$, we can form the reduct $A^{\otimes}|_{\Sigma}$ (cf. Section 2.3), and by the above definition we have an inclusion homomorphism $\imath_A : A \hookrightarrow A^{\otimes}|_{\Sigma}$. The freeness of $A^{\otimes}$ is expressed by the following.

Proposition 29 Let A be a Σ-algebra. Given a $\Sigma^{\otimes}$-algebra B and a Σ-homomorphism $h : A \to B|_{\Sigma}$, there is a unique $\Sigma^{\otimes}$-homomorphism $h^{\sharp} : A^{\otimes} \to B$ such that $i_A ; h^{\sharp}|_{\Sigma} = h$.

$$\begin{tikzcd}
A \arrow[r, "i_A"] \arrow[dr, "h"'] & A^{\otimes}|_{\Sigma} \arrow[d, "h^{\sharp}|_{\Sigma}"] & A^{\otimes} \arrow[d, "h^{\sharp}"] \\
 & B|_{\Sigma} & B
\end{tikzcd}$$

Proof The homomorphism $h^{\sharp}$ is defined inductively following Definition 28:

(1) $h^{\sharp}_s(a) = h_s(a)$ for $a \in A_s$;
(2) $h^{\sharp}_s(\sigma(a)) = B_\sigma(h^{\sharp}_w(a))$ for $\sigma \in \Sigma^{\otimes}_{w,s}$ and $a \in A^{\otimes}_w$.

Because B is a $\Sigma^{\otimes}$-algebra, this definition is well-defined with respect to the equations in (3) and (4) in Definition 28. The uniqueness of $h^{\sharp}$ can be proved by induction on $A^{\otimes}$. □

Finally, a property of retract extensions that we need later on is given in the following, which follows directly from Definition 28.

Fact 30 For all $\sigma \in \Sigma'^{\otimes}_{w,s}$ and all $x \in A^{\otimes}_w$, if $\sigma(x) \in A_s$ then $x \in A_w$. □

2.3 Institutions

Institutions were introduced by Goguen and Burstall [13] as an abstract model-theoretic formalisation of logical systems. An institution consists of notions of signatures, of models, of sentences (in the case of MSA, sentences are equations), and of satisfaction, with a technical requirement, called the 'Satisfaction Condition', which can be paraphrased as the statement that 'truth is invariant under change of notation'. The Satisfaction Condition is essential for reuse of specifications: it states that all properties that are true of a specification remain true in the context of another specification which imports that specification.

Definition 31 Given signatures $\Sigma \subseteq \Sigma'$ and a Σ'-algebra A', we write $A'|_\Sigma$ for A' viewed as a Σ-algebra by forgetting about all the sorts and operations in Σ' that are not in Σ. Moreover, given a Σ'-homomorphism $h : A'_1 \to A'_2$, we can form the Σ-homomorphism $h|_\Sigma : A'_1|_\Sigma \to A'_2|_\Sigma$ by defining $(h|_\Sigma)_s = h_s$ for $s \in S$. More generally, given a signature morphism $\phi : \Sigma \to \Sigma'$, we can define the Σ-algebra $\phi A'$, called the **reduct** of A' to Σ, by setting $\phi A'_s = A'_{\phi(s)}$ for $s \in S$ and $\phi A'_\sigma = A'_{\phi(\sigma)}$ for $\sigma \in \Sigma_{l,s}$. □

Similarly, given a Σ-equation e of the form $(\forall X)\, l = r$, we define $\phi(e)$ to be the Σ'-equation $(\forall X')\, \phi(l) = \phi(r)$, where X' is the S'-sorted set of variables with $X'_{s'} = \bigcup_{f(s)=s'} X_s$ for $s' \in S'$, and where $\phi(l)$ is the $\Sigma'(X')$-term obtained by replacing each operation name σ which occurs in l by $\phi(\sigma)$ (in this case, we view ϕ as the unique Σ-homomorphism from $T_\Sigma(X)$ to $\phi(T_{\Sigma'}(X'))$; see Proposition 45 in Subsection 2.5 below).

The **Satisfaction Condition** states that

$$\phi A' \models e \quad \text{iff} \quad A' \models \phi(e)$$

for all signature morphisms $\phi : \Sigma \to \Sigma'$, all Σ'-algebras A' and all Σ-equations e.

The Satisfaction Condition was shown to hold for MSA in [13], thereby showing that MSA constitutes an institution, and Goguen and Meseguer [20] show that OSA is an institution. Goguen and Diaconescu [15] show that hidden sorted algebra forms an institution, and discuss the significance of this for reusing object specifications and constructing systems of interacting objects. Section 2.4 shows that hidden order sorted algebra also forms an institution.

2.4 Hidden sorts

Hidden sorted algebra (hereafter 'HSA') was developed as a variation on MSA for objects with local states [10, 15]. 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. That is, visible sorts represent abstract data types, while hidden sorts represent abstract object classes. Subsection 2.4.1 summarises the basic definitions of HSA, and Subsection 2.4.2 combines HSA with OSA to give hidden order sorted algebra (hereafter, 'HOSA') and proves that HOSA is an institution.

2.4.1 Hidden sorted algebra

Signatures in HSA are defined with respect to a fixed **universe of data values**, which may be thought of as containing standard abstract data types such as the numbers, Booleans, lists, etc. This fixed universe is given by a triple (V, Ψ, D) where V is a set of visible sort names, (V, Ψ) is a many sorted signature, and D is a Ψ-algebra such that for each $d \in D_v$ with $v \in V$, there is a constant operation $\psi \in \Psi_{[],v}$ such that $D_\psi = d$.

Definition 32 A **hidden sorted signature** (over (V, Ψ, D)) is a pair (H, Σ) such that $(V \cup H, \Sigma)$ is a many sorted signature with $\Psi \subseteq \Sigma$, and such that the following two conditions hold:

(S1) if $w \in V^*$ and $v \in V$, then $\Sigma_{w,v} = \Psi_{w,v}$;
(S2) for each $\sigma \in \Sigma_{w,s}$, at most one element of w is in H.

The elements of V are referred to as **visible sorts**, and elements of H as **hidden sorts.**

A **hidden sorted signature morphism** $\phi : (H, \Sigma) \to (H', \Sigma')$ is a many sorted signature morphism $(V \cup H, \Sigma) \to (V \cup H', \Sigma')$ which is the identity on visible sorts (i.e., $\phi(v) = v$ for all $v \in V$) and which maps hidden sorts to hidden sorts (i.e., $\phi(h) \in H'$ for all $h \in H$). Moreover,

(M1) $\phi(\psi) = \psi$ for all $\psi \in \Psi_{w,v}$, and
(M2) for any $\sigma' \in \Sigma'_{w,s}$, if some sort in w is in $\phi(H)$ then $\sigma' = \phi(\sigma)$ for some σ in Σ.

We often abbreviate (H, Σ) to Σ.

A **hidden sorted** Σ**-algebra** is an $(H \cup V, \Sigma)$-algebra A such that $A|_\Psi = D$; that is, A interprets the visible sorts and operations in exactly the same way as D.

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

The HSA definition of satisfaction differs from that of 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 33 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 of the right sort. 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. □

A context of visible sort can be considered an experiment which, applied to an object's hidden state, gives a visible output. In HSA, 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 cannot be distinguished by any experiment. The definition of satisfaction uses the following

Notation 34 For a Σ-equation e of the form $(\forall X)\, l = r$ and a context $c \in T_\Sigma[z]_v$, we write $c[e]$ for the Σ-equation $(\forall X)\ c[l] = c[r]$. □

Definition 35 A hidden sorted Σ-algebra A **behaviourally satisfies** a Σ-equation e (indicated $A \models\!\!\!\equiv e$) iff $A \models c[e]$ 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 $A \models\!\!\!\equiv E$ iff $A \models\!\!\!\equiv e$ for all $e \in E$.

A **behavioural** (Σ, E)**-model** is a hidden sorted Σ-algebra A such that $A \models\!\!\!\equiv E$. □

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' $z \in T_\Sigma[z]_v$.

The notion of refinement that we use in the following sections is based on the idea that an object is refined by a behavioural model of its specification. Behavioural satisfaction of equations in HSA can also be expressed in terms of relations, as follows:

Proposition 36 $A \models\!\!\!\equiv E$ iff $(\equiv_{A,E})|_V = id_A|_V$ where $R|_V$ is the restriction of an S-sorted relation R to the sorts of V i.e., $R|_V$ is the V-sorted relation $(R_v)_{v \in V}$.

Proof: Note that $id_A|_V \subseteq (\equiv_{A,E})|_V$ because $\equiv_{A,E}$ is a congruence, so we need only show that $A \models\!\!\!\equiv E$ iff $(\equiv_{A,E})|_V \subseteq id_A|_V$. The 'if' implication is straightforward; we show the 'only if' implication. We construct a relation R on A such that $\equiv_{A,E} \subseteq R$ and $R|_V \subseteq id_A|_V$ so that $(\equiv_{A,E})|_V \subseteq id_A|_V$ as desired. Note that any context c for $t \in (T_\Sigma)_s$ of sort s' gives rise to an operation $A_s \to A_{s'}$ by composing the interpretations of the operation names in c; let us denote this operation by c^A. Now we define the relation R by $a\,R\,b$ iff $c^A(a) = c^A(b)$ for all $v \in V$ and $c \in T_\Sigma[z]_v$. If a and b are of visible sort, then we can take c to be the empty context (i.e., $c = z$), which shows that $R|_V \subseteq id_A|_V$. It only remains to show $\equiv_{A,E} \subseteq R$. It is straightforward to see that R is a Σ-congruence, so this inclusion follows by Definition 15 from $\mathsf{R}(e) \subseteq R$ for each $e \in E$. To show this inclusion, let $e \in E$ be an equation of the form $(\forall X)\, l = r$ and suppose that $a\,\mathsf{R}(e)\,b$ so that $a = \bar\theta(l)$ and $b = \bar\theta(r)$ for some $\theta : X \to A$. For any context c of visible sort, we have $c^A(a) = \bar\theta(c[l])$ and $c^A(b) = \bar\theta(c[r])$. If $A \models\!\!\!\equiv E$ then by Definition 35 we have $A \models (\forall X)\, c[l] = c[r]$, and therefore $c^A(a) = \bar\theta(c[l]) = \bar\theta(c[r]) = c^A(b)$ so that $a\,R\,b$. This shows that $\mathsf{R}(e) \subseteq R$ and concludes the proof. □

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

It is worth noting that with these definitions, HSA forms an *institution*, as shown in [15]; this also follows from the corresponding result for hidden order sorted algebra given below. The fact that HSA is an institution implies that inheritance and encapsulation of HSA modules behave in a coherent way: if a specification S is included in another, then things that are true of S in isolation remain true in the context of the signature in which S is included. In terms of objects, this means that the behaviour of an object is preserved if that object is included in a system of other objects. (See [15] for a more detailed discussion of these issues, with some examples.)

2.4.2 Hidden order sorted algebra

We now give the hidden sorted version of OSA, assuming a fixed universe $(V, \leq, \Psi, D)$ of data values; this differs from the HSA case in that $(V, \leq, \Psi)$ is an order sorted signature and D is an order sorted algebra.

Definition 37 An **HOSA signature** is a triple $(H, \leq, \Sigma)$ such that $(V \cup H, \leq, \Sigma)$ is an order sorted signature where the subsort ordering $\leq$ does not relate any visible sort to any hidden sort, and such that (H, Σ) is a hidden sorted signature.

An **HOSA signature morphism** $\phi : (H, \leq, \Sigma) \to (H', \leq', \Sigma')$ is both an order sorted signature morphism $(V \cup H, \leq, \Sigma) \to (V \cup H', \leq', \Sigma')$ and a hidden sorted signature morphism $(H, \Sigma) \to (H', \Sigma')$ which satisfies the following additional requirements:

(E1) for any $\sigma' \in \Sigma'_{w,s}$, if some sort in w is in $\phi(H)$ then there is a unique σ in Σ with $\sigma' = \phi(\sigma)$;

(E2) if $\phi(h) < h''$ then there is a unique $h' \in H$ such that $\phi(h') = h''$ and $h \leq h'$.

An **HOSA Σ-algebra** is an order sorted Σ-algebra which is also a hidden sorted Σ-algebra. An **HOSA specification** is a quadruple $(H, \leq, \Sigma, E)$ where $(H, \leq, \Sigma)$ is an HOSA signature and E is a set of Σ-equations (in the sense of OSA). □

As in HSA, satisfaction of equations in HOSA only requires that the visible consequences of an equation hold, and in this case the notion of visible consequence is defined in terms of contexts that contain retracts.

Definition 38 An HOSA Σ-algebra A **behaviourally satisfies** a Σ-equation e iff $A^{\otimes} \models c[e]$ for all $v \in V$ and $c \in T_{\Sigma^\otimes}[z]_v$. We write $A \models\!\!\!\equiv e$ to indicate that A behaviourally satisfies e, and $A \models\!\!\!\equiv E$ to indicate that A behaviourally satisfies each equation in E. A **behavioural** (Σ, E)**-model** is an HOSA Σ-algebra A such that A behaviourally satisfies E. □

Analogously to Proposition 36, we have the following reformulation of behavioural satisfaction in terms of Σ-congruences.

Proposition 39 An HOSA Σ-algebra A behaviourally satisfies E iff the free retract extension of A has no confusion in the sense of Proposition 36, i.e., $(\equiv_{A^{\otimes},E})|_V = id_{A^{\otimes}}|_V$. □

The proof is similar to that of Proposition 36. As stated in the following theorem, this definition of satisfaction makes HOSA into an institution. The institution of hidden order sorted algebra presented here is different from that giveu by Burstall and Diaconescu [3], because they define satisfaction using a retract extension that interprets operations strictly, i.e., if a retract is applied to something that does not belong to the carrier of the subsort, then the result is 'undefined' (a distinguished value denoted $\perp$), and all operations of the retract extension are strict in that they take $\perp$ to $\perp$.

Lemmas 42 and 43 below prove the following Satisfaction Condition for HOSA:

Theorem 40 The Satisfaction Condition holds for HOSA, i.e.,

$$\phi A \models\!\!\!\equiv_{\Sigma} e \quad \text{iff} \quad A \models\!\!\!\equiv_{\Sigma'} \phi(e)$$

for all HOSA signature morphisms $\phi : \Sigma \to \Sigma'$, hidden order sorted Σ'-algebras A, and Σ-equations e. □

The proof of this theorem is given by Lemmas 42 and 43 below. For the purposes of these two lemmas we assume given such a ϕ and A, and a Σ-equation e of the form $(\forall X)\, l = r$. However, first we need the following

Lemma 41 $(\phi A)^{\otimes} \simeq \phi^{\otimes} A^{\otimes}$.

Sketch of proof: The conditions (E1) and (E2) of Definition 37 allow the construction of a $\Sigma^{\otimes}$-homomorphism $g : \phi^{\otimes} A^{\otimes} \to (\phi A)^{\otimes}$ which is the inverse of $i^{\sharp} : (\phi A)^{\otimes} \to \phi^{\otimes} A^{\otimes}$, induced by $i : \phi A \hookrightarrow (\phi^{\otimes} A^{\otimes})|_{\Sigma}$. The morphism g is defined inductively on $(\phi^{\otimes} A^{\otimes})_h = A^{\otimes}_{\phi(h)}$:

- for $x \in A_{\phi(h)}$, let $g(x) = x$;
- for $\sigma' \in \Sigma'_{w,\phi(h)}$, by (E1) there is a unique σ in Σ with $\sigma' = \phi(\sigma)$; then for $y \in A^{\otimes}_w$ let $g(\sigma'(y)) = \sigma(g(y))$;
- for $\phi(h) \leq h''$, by (E2) there is a unique h' such that $\phi(h') = h''$, so for $y \in A^{\otimes}_{h''}$, let $g(r_{\phi(h),h''}(y)) = r_{h,h'}(g(y))$.

Conditions (E1) and (E2) of Definition 37 are necessary to make g a $\Sigma^{\otimes}$-homomorphism. For example, (E1) ensures that σ is uniquely chosen in the following calculation, which shows that g is a Σ-homomorphism.

$$\begin{array}{ll} & g((\phi^{\otimes} A^{\otimes})_{\sigma}(x)) \\ = & \\ & g(A^{\otimes}_{\phi\sigma}(x)) \\ = & \\ & g(\phi\sigma(x)) \\ = & \qquad \{\ (\text{E1})\ \} \\ & \sigma(g(x)) \\ = & \\ & (\phi A)^{\otimes}_{\sigma}(g(x)) \end{array}$$

Similarly, (E2) can be used to show that g distributes over retract operations, and is therefore a $\Sigma^{\otimes}$-homomorphism. □

Lemma 42 If $\phi A \models\!\!\!\equiv_{\Sigma} e$ then $A \models\!\!\!\equiv_{\Sigma'} \phi(e)$.

Proof: Assume $\phi A \models\!\!\!\equiv_{\Sigma} e$ so that $\phi A^{\otimes} \models_{\Sigma^{\otimes}} c[e]$ for all $\Sigma^{\otimes}$-contexts c; we have to show that $A^{\otimes} \models_{\Sigma'^{\otimes}} c[\phi(e)]$ for all $\Sigma'^{\otimes}$-contexts c. If c is a $\Sigma'^{\otimes}$-context, then by properties (S2) and (M2) of Definition 32, $c[z] = c_1[\phi^{\otimes}(c_2)[z]]$ for some $\Sigma'^{\otimes}$-context c_1 and $\Sigma^{\otimes}$-context c_2, so that $\phi A^{\otimes} \models_{\Sigma^{\otimes}} c_2[e]$. Now we can reason as follows:

$$\begin{array}{ll}
 & \phi A^{\otimes} \models_{\Sigma^{\otimes}} c_2[e] \\
\Leftrightarrow & \qquad \{ \text{ Lemma 41 } \} \\
 & \phi^{\otimes} A^{\otimes} \models_{\Sigma^{\otimes}} c_2[e] \\
\Leftrightarrow & \qquad \{ \text{ Satisfaction Condition for OSA } \} \\
 & A^{\otimes} \models_{\Sigma'^{\otimes}} \phi^{\otimes}(c_2[e]) \\
\Leftrightarrow & \\
 & A^{\otimes} \models_{\Sigma'^{\otimes}} \phi^{\otimes}(c_2)[\phi(e)] \\
\Rightarrow & \\
 & A^{\otimes} \models_{\Sigma'^{\otimes}} c[\phi(e)]
\end{array}$$

□

Lemma 43 If $A \models\!\!\!\equiv_{\Sigma'} \phi(e)$ then $\phi A \models\!\!\!\equiv_{\Sigma} e$.

Proof: Assume that $A \models\!\!\!\equiv_{\Sigma'} \phi(e)$ so that $A^{\otimes} \models_{\Sigma'^{\otimes}} c[\phi(e)]$ for all $\Sigma'^{\otimes}$-contexts c. If c is any $\Sigma^{\otimes}$-context then $\phi(c)^{\otimes}$ is a $\Sigma'^{\otimes}$-context, and so we have

$$A^{\otimes} \models_{\Sigma'^{\otimes}} \phi^{\otimes}(c)[\phi(e)] \ ,$$

and we reason further as follows:

$$\begin{array}{ll}
 & A^{\otimes} \models_{\Sigma'^{\otimes}} \phi^{\otimes}(c)[\phi(e)] \\
\Leftrightarrow & \\
 & A^{\otimes} \models_{\Sigma'^{\otimes}} \phi^{\otimes}(c[e]) \\
\Leftrightarrow & \qquad \{ \text{ Satisfaction Condition for OSA } \} \\
 & \phi^{\otimes} A^{\otimes} \models_{\Sigma^{\otimes}} c[e] \\
\Leftrightarrow & \qquad \{ \text{ Lemma 41 } \} \\
 & \phi A^{\otimes} \models_{\Sigma^{\otimes}} c[e]
\end{array}$$

□

These two lemmas establish the Satisfaction Condition for HOSA. In fact, with a little more work (showing functoriality of the constructions for equations and algebras with respect to signatures) we can show that HOSA forms an institution.

2.5 Horizontal and vertical signature morphisms

Signature morphisms perform two distinct rôles. One rôle is to express the importation of one specification into another or the passing of specifications as parameters; this is often referred to as *horizontal composition* [12, 28], and pertains to the modular structure of a system specification at a given level of abstraction. For instance, in the example of Subsection 4.1 below, a specification of the natural numbers is imported into a specification of stacks. It is desirable that such importation take place within an institution, for then the Satisfaction Condition guarantees that the inclusion morphism from the one signature to the other preserves properties of the imported module [13, 5]. The definitions of HOSA signature morphism in the previous subsection capture such *encapsulation* properties, so that when a specification of a class of objects is imported into a larger specification, the properties of the imported object classes are preserved [15].

The other rôle performed by signature morphisms is to compare two different specifications. This is referred to as *vertical composition*, and pertains to relationships between layers in a hierarchical system structure. In particular, a vertical signature morphism might express the fact that one specification in some system is refined by another specification, in the sense of Section 3.1 below. In such a case, we would not expect that signature morphisms encapsulate object class specifications, but rather expect that signature morphisms preserve the behaviour of object classes, in a sense that will be made precise in Definition 50 below. In this case, signature morphisms describe how the sorts and operations of the abstract specification are to be realised in the more concrete one. The following definition makes this precise:

Definition 44 Let $(H, \leq, \Sigma)$ and $(H', \leq', \Sigma')$ be two HOSA specifications over $(V, \leq, \Psi, D)$. A **vertical signature morphism** $\phi : (H, \leq, \Sigma) \to (H', \leq', \Sigma')$ is an OSA signature morphism $(V \cup H, \leq, \Sigma) \to (V \cup H', \leq', \Sigma')$ which maps hidden sorts to hidden sorts and is the identity on $(V, \leq, \Psi)$. □

That is, we no longer require conditions (E1) and (E2) of Definition 37.

A vertical signature morphism also provides a translation from terms to terms. In the case of a refinement, this states how programs of the abstract specification are to be 'compiled' into more concrete programs of the concrete specification. 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 which case all terms over the abstract signature are also terms over the concrete signature. However, non-inclusion translations are sometimes useful (see Subsection 4.1). The following states how an arbitrary (vertical) signature morphism extends to a translation of terms:

Proposition 45 Any OSA signature morphism $\phi = (f, g) : \Sigma \to \Sigma'$ that preserves overloading can be extended to a function ϕ with $\phi_s : (T_\Sigma)_s \to (T_{\Sigma'})_{f(s)}$ for all $s \in S$, defined as follows:

- for each constant symbol $\sigma \in \Sigma_{[],s}$, let $\phi_s(\sigma) = g(\sigma)$;
- for each non-empty sort list $w = s1 \ldots sn$, $\sigma \in \Sigma_{w,s}$, and $ti \in (T_\Sigma)_{si}$ for $i = 1, \ldots, n$, let

$$\phi_s(\sigma(t1, \ldots, tn)) = (g(\sigma))(\phi_{s1}(t1), \ldots, \phi_{sn}(tn)) .$$

In fact, ϕ is a Σ-homomorphism $T_\Sigma \to \phi(T_{\Sigma'})$. If f 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 all 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 $g^{\otimes}(r_{s1,s2}) = r_{f(s1),f(s2)}$. Finally, given any ground signature X of variable symbols, ϕ extends to $\phi_s : T_\Sigma(X)_s \to T_{\Sigma'}(X')_{f(s)}$ for each $s \in S$, where $X'_{s'} = \{x \in X_s \mid f(s) = s'\}$; 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. □

Our definition of refinement in the following section is phrased in terms of vertical signature morphisms. Because these need not satisfy conditions (E1) and (E2) of Definition 37, we cannot use any of the results of Section 2.4.2. However, given a vertical signature morphism $\phi : \Sigma \to \Sigma'$ and a Σ'-algebra A, we can establish a useful relation between $(\phi A)^{\otimes}$ and $\phi^{\otimes} A^{\otimes}$, using the homomorphism $i^{\sharp} : (\phi A)^{\otimes} \to \phi^{\otimes} A^{\otimes}$.

Lemma 46 For all $x \in (\phi A)^{\otimes}_s$, we have $i^{\sharp}(x) \in A_{\phi(s)}$ iff $x \in A_{\phi(s)}$.

Proof The 'if' direction is straightforward, because if $x \in A_{\phi(s)}$ then $i^{\sharp}(x) = x$. To see the 'only if' direction, an inductive argument can be sketched as follows. Let $x \in (\phi A)^{\otimes}_s$ be such that $i^{\sharp}(x) \in A_{\phi(s)}$. By Definition 27, either $x \in (\phi A)_s$, i.e., $x \in A_{\phi(s)}$ as desired, or x is of the form $\sigma(y)$ for some $\sigma \in \Sigma^{\otimes}_{w,s}$ and $y \in (\phi A)^{\otimes}_w$. In this case $i^{\sharp}(x) = \sigma(i^{\sharp}(y))$, and by Fact 30 we get $i^{\sharp}(y) \in A_{\phi(w)}$, so that by the induction hypothesis $y \in A_{\phi(w)}$, and therefore

$$i^{\sharp}(x) \;=\; \sigma(i^{\sharp}(y)) \;=\; \sigma(y) \;=\; x$$

and so $x \in A_{\phi(s)}$ as desired. □

As a consequence, we have

Lemma 47 $i^{\sharp}$ is injective.

Proof An inductive argument can be sketched as follows. Let $x, y \in (\phi A)^{\otimes}_s$ and suppose that $i^{\sharp}(x) = i^{\sharp}(y)$. If either one of x or y is in $A_{\phi(s)}$ then so is the other, and we have $x = i^{\sharp}(x) = i^{\sharp}(y) = y$. If neither x nor y are in $A_{\phi(s)}$, then both must have the same outermost symbol, e.g., $x = \sigma(x')$ and $y = \sigma(y')$, in which case $i^{\sharp}(x') = i^{\sharp}(y')$; by the induction hypothesis it follows that $x' = y'$, so that $x = y$. □

Now we can obtain our relationship between $(\phi A)^{\otimes}$ and $\phi^{\otimes} A^{\otimes}$:

Proposition 48 For all $\Sigma^{\otimes}$-equations e, if $\phi^{\otimes} A^{\otimes} \models e$ then $(\phi A)^{\otimes} \models e$.

Proof Suppose that $\phi^{\otimes} A^{\otimes} \models e$, where e is of the form $(\forall X)\, l = r$, and let $\theta : X \to (\phi A)^{\otimes}$. Then we have $i^{\sharp} \circ \theta : X \to \phi^{\otimes} A^{\otimes}$, and because $\phi^{\otimes} A^{\otimes}$ satisfies e we have

$$\overline{(i^{\sharp} \circ \theta)}(l) = \overline{(i^{\sharp} \circ \theta)}(r)$$

and therefore, by Lemma 12

$$i^{\sharp}(\bar{\theta}(l)) = i^{\sharp}(\bar{\theta}(r))$$

so, by Lemma 47, $\bar{\theta}(l) = \bar{\theta}(r)$, which shows that $(\phi A)^{\otimes} \models e$ as desired. □

The above proposition is used in Section 3.2 to give a sufficient condition for correctness of refinement. An interesting corollary is that one half of the Satisfaction Condition holds for vertical signature morphisms.

Corollary 49 For all vertical signature morphisms $\phi : \Sigma \to \Sigma'$, all HOSA Σ'-algebras A, and all Σ-equations e,

$$A \models_{\Sigma'} \phi(e) \quad \text{implies} \quad \phi A \models_{\Sigma} e\ .$$

Proof

$\phi A \models\!\!\equiv_\Sigma e$

$\Leftrightarrow$

$(\forall c \in T_{\Sigma^\otimes}[z]_v)\ (\phi A)^\otimes \models_\Sigma c[e]$

$\Leftarrow$ { Proposition 48 }

$(\forall c \in T_{\Sigma^\otimes}[z]_v)\ \phi^\otimes A^\otimes \models_\Sigma c[e]$

$\Leftrightarrow$ { Satisfaction Condition for OSA }

$(\forall c \in T_{\Sigma^\otimes}[z]_v)\ A^\otimes \models_{\Sigma'} (\phi^\otimes c)[\phi(e)]$

$\Leftarrow$

$(\forall c' \in T_{\Sigma'^\otimes}[z]_v)\ A^\otimes \models_{\Sigma'} c'[\phi(e)]$

$\Leftrightarrow$

$A \models\!\!\equiv_{\Sigma'} \phi(e)$

□

3 Refinement and Implementation

This section defines refinement and implementation for hidden order sorted specifications, and presents a proof technique for proving correctness of refinement which leads to simple correctness proofs. The proof technique is also applicable to proofs of correctness of implementation.

For the remainder of this section, we fix a universe of visible data values $(V, \leq, \Psi, D)$ and two HOSA specifications, $A = (HA, \leq_A, \Sigma A, EA)$ and $C = (HC, \leq_C, \Sigma C, EC)$, where A is for 'abstract' and C is for 'concrete', plus a vertical signature morphism $\phi : \Sigma A \to \Sigma C$ which preserves overloading. In many examples of refinement and implementation, the morphism ϕ is an inclusion of signatures, so that all of the sorts and operations in the abstract specification are also available in the concrete specification. However, there is no need for such a restriction on ϕ, and indeed it is sometimes useful to allow refinements that do not use signature inclusions; an example of this is given in Section 4.1.

3.1 Refinement

Refinement is the process of moving from one specification to another, more concrete, specification which displays the same behaviour. The phrase 'more concrete' is generally understood to refer to a specification which can, in some sense, be more efficiently or more directly implemented. The requirement that the concrete specification display the same behaviour means that all models of the concrete specification are implementations, i.e., models, of the original, 'abstract' specification.

The standard definition of refinement of algebraic specifications, whether in MSA, OSA or HSA [27, 28, 24], is this: for every model M of the concrete specification C, the reduct ϕM is a model of the abstract specification A. We define refinement of HOSA specifications in the same way.

Definition 50 A **is refined by** C iff for all HOSA C-models M, the reduct ϕM is a HOSA A-model. □

If we ignore, for the moment, the hidden sortedness of A and C, this means that for every ΣC-algebra M, if $M \models EC$ then $\phi M \models EA$; or equivalently, by the Satisfaction Condition for OSA, $M \models \phi(EA)$. If ϕ is an inclusion of signatures, the latter formula is the same as $M \models EA$, so that 'C refines A' means that all models of C are models of A. However, because ϕ is a vertical signature morphism and not an HOSA signature morphism, this line of reasoning, in particular the appeal to to the Satisfaction Condition, is not valid. Nevertheless, the main point is that 'A is refined by C' means that all models of C give rise to models of A by reduct along ϕ.

The following subsection investigates ways of proving refinement, and develops a proof technique which is exemplified in Subsections 4.1 and 4.2.

3.2 Proofs of refinement

Henniker [24] proposes a technique for proving correctness of refinement for HSA specifications which is based on showing that all visible consequences of the equations of the abstract specification are satisfied by the concrete specification. The notion of visible consequence is defined in terms of contexts, and Henniker's proof technique is based on induction over the size of contexts. Such induction proofs are often surprisingly complicated (cf. the statement in [7] that 'putting context induction into practise was less straightforward than expected'), so we seek both to extend Henniker's results to the order sorted case, and to simplify the inductive proofs. A useful hint as to how this can be achieved is obtained from the work of Schoett [29], which is concerned with data representation rather than refinement, and is set in the context of partial algebras rather than OSA. Schoett shows correctness of data representations by constructing congruence relations between the carriers of models, with the idea that these congruences relate behaviourally equivalent values.

In this section we present a variety of ways of proving correctness of refinement. First, we extend Henniker's technique to the order sorted case, to give a proof technique based on showing that visible consequences of the equations of the abstract specification are satisfied in the concrete specification. Then we generalise this by considering arbitrary congruence relations which relate the left- and right-hand sides of the equations of the abstract specification. Finally, we consider splitting the signature of the abstract specification into 'generators' and 'derived functions'; this gives rise to a proof technique which can greatly simplify proofs of refinement.

For the purposes of this subsection, we suppose a fixed pair of HOSA specifications A and C and a vertical signature morphism $\phi : \Sigma A \to \Sigma C$ as above. We also use the following abbreviations:

Notation 51 We write TA for the carrier of the term algebra $T_{\Sigma A}$; $TA^{\otimes}$ for that of $T_{\Sigma A^{\otimes}}$ (cf. Definition 27); $TA[z]$ for the contexts in $T_{\Sigma A}[z]$ and $TA^{\otimes}[z]$ for $T_{\Sigma A^{\otimes}}[z]$ (cf. Definition 33), and similarly for C; and we write ϕ for $\phi : TA \to TC$ as well as for $\phi^{\otimes} : TA^{\otimes} \to TC^{\otimes}$ (cf. Proposition 45). We also write SA for $V \cup HA$ and SC for $V \cup HC$. □

First of all, we want to reduce correctness of refinement to showing that the visible consequences of the equations in EA are consequences of EC. We require the following lemma.

Lemma 52 For all HOSA specifications $Sp = (H, \leq, \Sigma, E)$ and all HOSA Sp-algebras M and all visible $\Sigma^{\otimes}$-equations e, if $E^{\otimes} \models e$ then $M^{\otimes} \models e$.

Proof Suppose M is an HOSA Sp-algebra, so that by Proposition 39 we have

(2) $(\equiv_{M^{\otimes}, E^{\otimes}})|_V \subseteq id_{M^{\otimes}}$.

Suppose also that $E^{\otimes} \models e$ so that by Proposition 20 and the fact that $\equiv_{M^{\otimes}, E^{\otimes}}$ is a congruence, we have

(3) $\mathsf{R}_{M^{\otimes}}(e) \subseteq \equiv_{M^{\otimes},E^{\otimes}}$.

We need to show that $\equiv_{M^{\otimes},\{e\}} \subseteq id_{M^{\otimes}}$; because $id_{M^{\otimes}}$ is a congruence, it is sufficient to show that $\mathsf{R}_{M^{\otimes}}(e) \subseteq id_{M^{\otimes}}$. If $x \, \mathsf{R}_{M^{\otimes}}(e) \, y$ then by (3) we have $x \equiv_{M^{\otimes},E^{\otimes}} y$, and because e is of visible sort, so too are x and y, so (2) gives $x \, id_{M^{\otimes}} \, y$ as desired. □

This allows us to reduce proofs of refinement to showing that the concrete specification satisfies all visible consequences of the equations in the abstract specification.

Theorem 53 A is refined by C if $EC^{\otimes} \models \phi^{\otimes}(c[e])$ for all $e \in EA$ and visible contexts $c \in TA^{\otimes}[z]$.

Proof Suppose that $EC^{\otimes} \models \phi^{\otimes}(c[e])$ for all $e \in EA$ and visible contexts $c \in TA^{\otimes}[z]$, and let M be a HOSA C model. Then for any $e \in EA$,

$$\begin{array}{ll} & \phi M \models\!\equiv e \\ \Leftrightarrow & \\ & (\forall c \in TA^{\otimes}[z]_v) \ (\phi M)^{\otimes} \models c[e] \\ \Leftarrow & \qquad \{ \text{ Proposition 48 } \} \\ & (\forall c \in TA^{\otimes}[z]_v) \ \phi^{\otimes} M^{\otimes} \models c[e] \\ \Leftrightarrow & \\ & (\forall c \in TA^{\otimes}[z]_v) \ M^{\otimes} \models \phi^{\otimes}(c[e]) \\ \Leftarrow & \qquad \{ \text{ Lemma 52 } \} \\ & (\forall c \in TA^{\otimes}[z]_v) \ EC^{\otimes} \models \phi^{\otimes}(c[e]) \end{array}$$

which shows that ϕM is a HOSA A-model as desired. □

This theorem states that A is refined by C if the left- and right-hand sides of each equation in EA are related by $=_{EC^{\otimes}}$ in all visible $\Sigma A^{\otimes}$-contexts. We can generalise this to a requirement that the left- and right-hand sides are related by a certain kind of congruence relation.

Definition 54 Let $\phi : \Sigma \to \Sigma'$, and let R be an equivalence relation on some Σ'-algebra M'. We say that R is a **$\phi\Sigma$-congruence** iff for all $\sigma \in \Sigma_{w,s}$ and all $x, y \in M'_{\phi(w)}$, if $x \, R \, y$ then $M'_{\phi(\sigma)}(x) \, R \, M'_{\phi(\sigma)}(y)$. □

We also require the following notations.

Definition 55 If R is a relation on M' and $f : M \to M'$, then we write R^f for the relation on M such that $x \, R^f \, y$ iff $f(x) \, R \, f(y)$. □

Definition 56 If R is a relation on $TC^{\otimes}$ and X is a set of SC-sorted variables, then we write $R(X)$ for the relation on $TC^{\otimes}(X)$ such that

$$t \; R(X) \; t' \qquad \text{iff} \qquad (\forall \theta : X \to TC^{\otimes}) \; \bar{\theta}(t) \; R \; \bar{\theta}(t') \; .$$

□

Now we can generalise Theorem 53 by allowing the left- and right-hand sides of each of the abstract equations to be related by a $\phi\Sigma$-congruence whose restriction to visible sorts implies equality.

Theorem 57 A is refined by C if there exists a $\phi\Sigma A^{\otimes}$-congruence R on $TC^{\otimes}$ such that $l \; R(X)^{\phi} \; r$ for all equations $(\forall X) \, l = r$ in $EA^{\otimes}$ and such that $R|_V \subseteq =_{EC^{\otimes}}$. □

The proof of this theorem requires the following version of soundness for OSA [20], which says that all models of a specification satisfy an equation if the initial model satisfies it.

Lemma 58 For all $l, r \in T_\Sigma(X)$ and sets E of Σ-equations, if $l =_E(X)\ r$ then $E \models (\forall X)\, l = r$. $\Box$

Proof of Theorem 57 Let R be a $\phi\Sigma A^\otimes$-congruence satisfying the above conditions. By Theorem 53 it suffices to show that $EC^\otimes \models \phi^\otimes(c[e])$ for each $e \in EA$ and $c \in TC^\otimes[z]_v$. Let $e \in EA$ be of the form $(\forall X)\, l = r$. Then

$$
\begin{array}{ll}
 & l\ R(X)^\phi\ r \\
\Leftrightarrow & \qquad \{\text{ Definition 55 }\} \\
 & \phi(l)\ R(X)\ \phi(r) \\
\Rightarrow & \qquad \{\ R \text{ is a } \phi\Sigma A^\otimes\text{-congruence }\} \\
 & (\forall c \in TA^\otimes[z]_v)\ \phi^\otimes(c)[\phi(l)]\ R(X)\ \phi^\otimes(c)[\phi(r)] \\
\Rightarrow & \qquad \{\ R|_V \subseteq =_{EC^\otimes}\ \} \\
 & (\forall c \in TA^\otimes[z]_v)\ \phi^\otimes(c)[\bar\theta(\phi(l))]\ =_{EC^\otimes} (X)\ \phi^\otimes(c)[\bar\theta(\phi(r))] \\
\Rightarrow & \qquad \{\text{ Lemma 58 }\} \\
 & (\forall c \in TA^\otimes[z]_v)\ EC^\otimes \models \phi^\otimes(c[e])
\end{array}
$$

An obvious candidate for the relation R of this theorem is **behavioural equivalence**, which is defined as follows:

Definition 59 For $t, t' \in TC^\otimes$, let $t \sim t'$ iff $\phi^\otimes(c)[t] =_{EC^\otimes} \phi^\otimes(c)[t']$ for all $v \in V$ and $c \in TA^\otimes[z]_v$. $\Box$

This relation clearly satisfies the conditions of Theorem 57. However, using this 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, the only assumption we make about G and D is that their union is equal to $\Sigma A^\otimes$.) 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: two terms are behaviourally equivalent iff they give the same result in all visible contexts built from the operations of D.

Definition 60 For $t, t' \in TC^\otimes$, we define $t \smile t'$ iff $\phi^\otimes(c)[t] =_{EC^\otimes} \phi^\otimes(c)[t']$ for all $v \in V$ and $c \in T_D[z]_v$. $\Box$

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 4.1 and 4.2, in examples where all derived functions are unary, an assumption that allows us to state the property concisely:

Proposition 61 If all the operations of D have only one argument, then for $h \in HA$ and $t, t' \in TC^\otimes_h$, we have $t \smile t'$ iff $(\phi\sigma)(t) \smile (\phi\sigma)(t')$ for each $r \in SA$ and $\sigma \in D_{h,r}$. $\Box$

The relation $\smile$ is an equivalence relation, it contains $=_{EC^{\otimes}}$, and its restriction to visible sorts implies equality; moreover, $\smile$ is a ϕD-congruence, so to use Proposition 57, we need only show that it is also a ϕG-congruence. In fact, there is a nice relationship between our two notions of behavioural equivalence: from $D \subseteq \Sigma A^{\otimes}$, it follows that $\sim\ \subseteq\ \smile$; moreover, if $\smile$ is also a ϕG-congruence then the following proposition shows that $\smile\ \subseteq\ \sim$, and so $\smile\ =\ \sim$.

Proposition 62 If $\smile$ is a ϕG-congruence then $\smile\ =\ \sim$.

Proof: We have already noted that $\smile\ \supseteq\ \sim$, so it suffices to show that $\smile\ \subseteq\ \sim$. Now $\smile$ is a ϕD-congruence, so if it is also a ϕG-congruence, then because $\Sigma A^{\otimes} = G \cup D$, it is a $\phi\Sigma A^{\otimes}$-congruence. So:

$$\begin{array}{ll}
 & t \smile t' \\
\Rightarrow & \qquad \{ \smile \text{ is a } \phi\Sigma A^{\otimes}\text{-congruence} \} \\
 & (\forall v \in V)(\forall c \in TA^{\otimes}[z]_v)\ \phi(c)[t] \smile \phi(c)[t'] \\
\Rightarrow & \qquad \{ \smile\!|_V \subseteq =_{EC^{\otimes}} \} \\
 & (\forall v \in V)(\forall c \in TA^{\otimes}[z]_v)\ \phi(c)[t] =_{EC^{\otimes}} \phi(c)[t'] \\
\Leftrightarrow & \qquad \{ \text{Definition 59} \} \\
 & t \sim t' .
\end{array}$$

□

Theorem 57 and Proposition 62 together give the following sufficient condition for refinement:

Proposition 63 A is refined by C if $l \smile(X)^{\phi}\ r$ for each $(\forall X)\, l = r$ in EA and if $\smile$ is a ϕG-congruence. □

The importance of this result is that it greatly simplifies correctness proofs. With this Proposition, the correctness of a refinement is shown by proving that $\smile$ is a ϕG-congruence, and that the equations of the abstract specification are satisfied in the concrete specification in all D-contexts. This latter proof obligation can be shown by induction on the structure of D-contexts; because D is a subsignature of Σ, there will be fewer cases to consider in the induction steps. This is illustrated in the example proof in Section 4.1 below. In fact, sometimes it completely eliminates the need for an inductive proof altogether! Section 4.3 gives an example of a refinement where there is essentially only one D-context, and the proof can proceed by simple equational reasoning (the example is taken from Henniker [24], but there the correctness proof uses a rather complicated induction on Σ-contexts).

3.3 Implementation

Our notion of refinement is based on the idea that the concrete specification should display the same visible behaviour as the abstract one; that is, the reduct of any model of the concrete specification behaviourally satisfies all the equations of the abstract specification. If we regard terms over the abstract signature as programs that the concrete specification should implement, then we need consider only terms without variables: that is, our notion of behaviour is given by the ground equations of the abstract theory. If we ignore hidden and order sortedness, then we might say that A is implemented by C iff all ground equalities of A, when translated by ϕ, also hold in C; i.e.,

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

This is only a partial definition of correct implementation (see Definition 67 below); in fact, this definition is sometimes referred to as 'simulation' [27]. The intuitive meaning of (4) can be seen by considering t_1 to be a program that gives a result t_2 in the abstract specification; then the translation of t_1 should give the corresponding result in the implementation. More formally, (4) states that the ϕ-translations of the ground consequences of the equations in EA are entailed by EC. This is equivalent to requiring the ϕ-translations of ground instances of the equations in EA to be entailed by EC, i.e., that

(5) $\quad \phi(\bar{\theta}(l)) =_{EC} \phi(\bar{\theta}(r))$ for each $(\forall X)\, l = r \in EA$ and each $\theta : X \to TA$.

To see that (4) implies (5), note that for any equation $(\forall X)\, l = r$ in EA and any assignment $\theta : X \to TA$, we have $\bar{\theta}(l) =_{EA} \bar{\theta}(r)$, from which by (4) we conclude that $\phi(\bar{\theta}(l)) =_{EC} \phi(\bar{\theta}(r))$ as desired. To see the converse implication, note that (4) may be reformulated as $=_{EA} \subseteq (=_{EC})^{\phi}$, and this follows by definition of $=_{EA}$ (Definition 15) from

$$\mathsf{R}(e) \subseteq (=_{EC})^{\phi} \qquad \text{for each } e \in EA.$$

To show this, let $e \in EA$ be an equation of the form $(\forall X)\, l = r$, and suppose that $a\, \mathsf{R}(e)\, b$, so that $a = \bar{\theta}(l)$ and $b = \bar{\theta}(r)$ for some $\theta : X \to TA$. By (5) we have $\phi(a) =_{EC} \phi(b)$, i.e., $a\, (=_{EC})^{\phi}\, b$ as desired.

Although (4) and (5) are equivalent, the formulation in (5) is generally easier to prove, in that one need only show that ground instances of each of the equations in EA are satisfied in the concrete specification, rather than considering all consequences of these equations.

If we take hidden sortedness into account, we need only consider equalities of visible sort, so that the requirement (4) for implementation becomes

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

This is the definition given by Henniker [24], though only for the case that ϕ is a signature inclusion. Henniker also proposes an inductive method for proving implementation correctness, by restating this condition in terms of behavioural equivalence, which is defined as follows:

Definition 64 $t, t' \in TC$ are **A-behaviourally equivalent**, written $t \sim_A t'$, iff for all $v \in V$ and $c \in TA[z]_v$, we have $\phi(c)[t] =_{EC} \phi(c)[t']$. Implicitly, if the variable z has sort s, then t and t' have sort $\phi_1(s)$. We will generally omit the subscript A on $\sim_A$. □

We can use the notion of behavioural equivalence to reformulate (6) in terms of ground equalities of arbitrary sort:

Proposition 65 (6) is equivalent to

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

Proof: To show that (6) implies (7), let $t_1 =_{EA} t_2$ for some $s \in SA$ and $t_1, t_2 \in TA_s$, and let $v \in V$ and $c \in TA[z]_v$. We have $c[t_1] =_{EA} c[t_2]$, and because both terms are of visible sort, (6) gives $\phi(c[t_1]) =_{EC} \phi(c[t_2])$, i.e., $\phi(c)[\phi(t_1)] =_{EC} \phi(c)[\phi(t_2)]$, which shows that $\phi(t_1) \sim \phi(t_2)$ as desired. To show that (7) implies (6), let $t_1 =_{EA} t_2$ for some $v \in V$ and $t_1, t_2 \in TA_v$. By (7) we have $\phi(t_1) \sim \phi(t_2)$. Both terms are of visible sort, so taking c to be the empty context (i.e., $c = z$), Definition 64 gives $\phi(t_1) =_{EC} \phi(t_2)$ as desired. □

Once again, this condition for implementation can be reformulated in terms of the ground instances of each equation in EA. In particular, Henniker shows that (6), and therefore (7), is equivalent to:

(8) $\qquad \phi(\bar{\theta}(l)) \sim \phi(\bar{\theta}(r)) \qquad$ for each $(\forall X)\, l = r \in EA$ and $\theta : X \rightarrow TA$.

Proposition 66 (7) is equivalent to (8).

Proof: That (7) implies (8) is straightforward: for any equation $(\forall X)\, l = r$ in EA and any $\theta : X \rightarrow TA$ we have $\bar{\theta}(l) =_{EA} \bar{\theta}(r)$, whence by (7) we have $\phi(\bar{\theta}(l)) \sim \phi(\bar{\theta}(r))$ as desired. To show the converse implication, note that (7) can be reformulated as $=_{EA} \subseteq \sim^{\phi}$, which follows from $\mathsf{R}(e) \subseteq \sim^{\phi}$ for all $e \in EA$. To show this, let e be of the form $(\forall X)\, l = r$, and let $a\, \mathsf{R}(e)\, b$ so that $a = \bar{\theta}(l)$ and $b = \bar{\theta}(r)$ for some $\theta : X \rightarrow TA$. By (8) we get $\phi(a) = \phi(\bar{\theta}(l)) \sim \phi(\bar{\theta}(r)) = \phi(b)$ as desired. □

The equivalence of (4) and (5) is mirrored in that of (7) and (8). Both (5) and (8) have a form that simplifies the proof obligations. Henniker [24] investigates proofs of (8) by a form of induction on the size of contexts $c \in TA[z]_v$; however, such proofs can be very complicated.

The situation is more complex for HOSA because the definition of implementation has to consider the well-definedness of terms, which may amount to termination of programs. Schoett [29] 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 4.1 below), where `top` and `pop` both require a non-empty stack as argument, and suppose further that the specification contains the equation

$$(\forall X : \mathtt{Nat}, S : \mathtt{Stack})\ \mathtt{top}\ \mathtt{push}(X, S) \ = \ X \ .$$

The term `top push(0, pop empty)` can be viewed in two different ways: it either gives the value `0`, or else it is an error term. The first is a lazy evaluation view of error-handling, where terms with error subterms can still have well-defined values; the second view, which is 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, correct implementations 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. Moreover, we can handle the strict view by adding some 'error equations', which identify some error terms, as discussed by Goguen and Diaconescu in [14].

Our definition of implementation in HOSA is that C implements A iff whenever a visible sorted term t of $TA^{\otimes}$ is not an error term (i.e., is equal to a term t' of TA), then the ϕ-translation of t is equal to the ϕ-translation of t' in TC.

Definition 67 An HOSA specification C is a **partial behavioural implementation** of an HOSA specification A via the vertical signature morphism ϕ (we write $\phi : A \sqsubseteq C$) iff $t =_{EA^{\otimes}} t'$ implies $\phi(t) =_{EC^{\otimes}} \phi(t')$ for all $v \in V$, $t \in TA^{\otimes}_v$ 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 (6) to the hidden 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 by identifying some or even all data values, are allowed for partial behavioural implementations, but not for behavioural implementations proper. In the following, we concentrate on proofs of partial implementation, i.e., on showing that visible terms equal in the abstract specification are equal in the concrete. Additional techniques are required for showing correctness of 'total' behavioural implementation.

The notion of partial behavioural implementation is also significant at the level of algebras of a specification: if $\phi : A \sqsubseteq C$, then for any C-algebra M and any *ground* ΣA-equation e such that $EA \models\!\!\!\equiv e$, we have $\phi M \models\!\!\!\equiv e$.

3.4 Proofs of partial implementation

In this subsection we formnlate a sufficient condition for partial behavioural implementation which simplifies the proof obligations in the same way that (5) and (8) simplify (4) and (7). Much of the technical development in this section is analogous to that of Section 3.2 above. The main differences are that in this section we are concerned with ground equations rather than with arbitrary equations, and we treat error terms in a 'lazy' way. A sufficient condition for correctness of partial implementation is given in Corollary 73 below. Instances of this corollary may be proved by induction on contexts; the technique presented in Section 3.2, of splitting the abstract signature into generators and derived functions, can be adapted to proofs of partial implementation. This proof technique is presnted in Proposition 76, which corresponds to Proposition 63 in Section 3.2.

One way to show that C is a partial behavioural implementation of A is to construct an intermediate relation R on $\Sigma C^{\otimes}$-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 the definition of partial bebavioural implementation (Definition 67). If R^{ϕ} is also a $\Sigma A^{\otimes}$-congruence, then (a) holds iff R^{ϕ} extends the ground instances of the equations in $EA^{\otimes}$. This is the intuition behind Proposition 68 below, which is our main technical result. Its statement uses the following:

Proposition 68 If there exists an equivalence relation R on $TC^{\otimes}$ such that R^{ϕ} is a $\Sigma A^{\otimes}$-congruence and

(9) $\quad \bar{\theta}(l) \ R^{\phi} \ \bar{\theta}(r) \quad$ for each $(\forall X)\, l{=}r \in EA^{\otimes}$ and $\theta : X \to TA^{\otimes}$,

(10) $\quad$ if $t \, R^{\phi} \, t'$ then $\phi(t) =_{EC^{\otimes}} \phi(t')$ for all $v \in V$, $t \in TA^{\otimes}_{v}$ and $t' \in TA_{v}$,

then $\phi : A \sqsubseteq C$.

Proof: The relation $=_{EA^{\otimes}}$ is by definition the least $\Sigma A^{\otimes}$-congruence satisfying (9), so $=_{EA^{\otimes}} \subseteq R^{\phi}$. To show that $\phi : A \sqsubseteq C$, fix $v \in V$, $t \in TA^{\otimes}_{v}$, $t' \in TA_{v}$; 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, (10) gives $\phi(t) =_{EC^{\otimes}} \phi(t')$ as desired. □

A weaker, but very useful version of this result is obtained by strengthening (9):

Proposition 69 For any relation R on $TC^{\otimes}$, condition (9) of Proposition 68 follows from

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

(12) $\quad \bar{\theta}(l)\ R^{\phi}\ \bar{\theta}(r)$ for each $(\forall X)\, l{=}r \in EA$ and $\theta : X \rightarrow TA^{\otimes}$.

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 $(\forall S' : \phi(s2))\ r_{\phi(s1),\phi(s2)}(S') = S'$, so for any $\theta : \{S\} \rightarrow TA^{\otimes}$, we have $\phi(\bar{\theta}(r_{s1,s2}(S))) = r_{\phi(s1),\phi(s2)}(\phi(\bar{\theta}(S))) =_{EC^{\otimes}} \phi(\bar{\theta}(S))$. Therefore by (11), $\bar{\theta}(r_{s1,s2}(S))\ R^{\phi}\ \bar{\theta}(S)$, and combining this with (12) gives (9). □

The weakening of Proposition 68 by replacing (9) with (11) and (12) is useful because with (11), 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 (11), so that in proving partial implementation, we may concentrate on proving (12), 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 64; but the following relation is more general:

Definition 70 For $t, t' \in TC^{\otimes}$, **equivalence up to definition**, denoted $t \simeq t'$, means that $t =_{EC^{\otimes}} t''$ iff $t' =_{EC^{\otimes}} t''$ for all $t'' \in TC$. □

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

Definition 71 For any relation R on $TC^{\otimes}$, **behavioural R-equivalence**, denoted $\tilde{R}$, is defined for $t, t' \in TC^{\otimes}$ by $t\ \tilde{R}\ t'$ iff $\phi(c)[t]\ R\ \phi(c)[t']$ for all $v \in V$ and $c \in TA^{\otimes}[z]_v$. □

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 (11):

Proposition 72 When R is $=_{EC^{\otimes}}$ or $\simeq$, then $=_{EC^{\otimes}} \subseteq \tilde{R}$.

Proof: When R is $=_{EC^{\otimes}}$, the result is immediate. For the case that R is $\simeq$, let $t =_{EC^{\otimes}} t'$; for any context c we have $\phi(c)[t] =_{EC^{\otimes}} \phi(c)[t']$, so for any $t'' \in TC$ we have $\phi(c)[t] =_{EC^{\otimes}} t''$ iff $\phi(c)[t'] =_{EC^{\otimes}} t''$. That is, $\phi(c)[t] \simeq \phi(c)[t']$, and so $t\ \tilde{R}\ t'$. □

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.,

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

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

Because $\approx$ satisfies all requirements of Proposition 68 except (9), we obtain the following from Propositions 69 and 72:

Corollary 73 If all equations of EA are behaviourally satisfied by C, i.e., if $\bar{\theta}(l) \approx^{\phi} \bar{\theta}(r)$ for each $(\forall X)\, l{=}r$ in EA and $\theta : X \to TA^{\otimes}$, then $\phi : A \sqsubseteq C$.

Proof: We need to show that $\approx$ satisfies all requirements of Proposition 68 except (9). It is straightforward to show that $\approx$ is an equivalence relation and that $\approx^{\phi}$ is a ΣA-congruence; the only remaining requirement is (10), i.e.,

$$\text{if } t \approx^{\phi} t' \text{ then } \phi(t) =_{EC^{\otimes}} \phi(t') \quad \text{for all } v \in V,\, t \in TA^{\otimes}_{v} \text{ and } t' \in TA_{v}$$

Suppose $\phi(t) \approx \phi(t')$; since both terms are of visible sort, we may choose the empty context $c = z$ in (13) to obtain $\phi(t) \simeq \phi(t')$. Because $t' \in TA_v$ we have $\phi(t') \in TC_v$, and by Definition 70 we get $\phi(t) =_{EC^{\otimes}} \phi(t')$ iff $\phi(t') =_{EC^{\otimes}} \phi(t')$, so that $\phi(t) =_{EC^{\otimes}} \phi(t')$ as desired. □

This result can still lead to complicated proofs by context induction, but we can apply the proof technique of Section 3.2 of splitting the abstract signature into generators and derived functions: suppose that $\Sigma A^{\otimes} = G \cup D$ (again, there are no further assumptions about G and D). We define behavioural equivalence under D-contexts as follows.

Definition 74 For $t, t' \in TC^{\otimes}$, we define $t \smile t'$ iff $\phi(c)[t] \simeq \phi(c)[t']$ for all $v \in V$ and $c \in T_D[z]_v$. □

The relation $\smile$ is an equivalence relation, it contains $=_{EC^{\otimes}}$, and its restriction to visible sorts is the same as behavioural equivalence; moreover, corresponding to Proposition 62 we have the following

Proposition 75 $\smile\, = \,\approx$ if $t \smile t'$ implies $(\phi\,\sigma)(t) \smile (\phi\,\sigma)(t')$ for all $\sigma \in G_{w,s}$ and $t, t' \in TC^{\otimes}_{\phi^{*}(w)}$.

Proof: We have already noted that $\smile\, \supseteq\, \approx$, so it suffices to show that $\smile\, \subseteq\, \approx$. Now $\smile^{\phi}$ 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 $\Sigma A^{\otimes}$-congruence. So:

$$\begin{array}{ll}
 & t \smile t' \\
\Rightarrow & \qquad \{ \smile^{\phi} \text{ is a congruence } \} \\
 & (\forall v \in V)(\forall c \in TA^{\otimes}[z]_v)\ \phi(c)[t] \smile \phi(c)[t'] \\
\Rightarrow & \qquad \{ \smile\!|_V \subseteq \simeq \} \\
 & (\forall v \in V)(\forall c \in TA^{\otimes}[z]_v)\ \phi(c)[t] \simeq \phi(c)[t'] \\
\Leftrightarrow & \qquad \{ (13) \} \\
 & t \approx t' \,.
\end{array}$$

□

Corollary 73 and Proposition 75 together give the following sufficient condition for partial implementation:

Proposition 76 If $\bar{\theta}(l) \smile^{\phi} \bar{\theta}(r)$ for each $(\forall X)\, l{=}r$ in EA and $\theta : X \to TA^{\otimes}$, and if $t \smile t'$ implies $(\phi\,\sigma)(t) \smile (\phi\,\sigma)(t')$ for all $\sigma \in G_{w,s}$ and $t, t' \in TC^{\otimes}_{\phi^{*}(w)}$, then $\phi : A \sqsubseteq C$. □

The main differences with Proposition 63 for refinement are our use of behavioural equivalence up to termination, and the fact that this proposition considers only ground instances of equations.

4 Example Proofs

This section uses the results of Section 3.2 to prove the correctness of a number of examples of refinement.

4.1 Example: a stack object

In our first example, 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 refines stacks by arrays and pointers. This example, adapted from [9], is well-known, but we present it here to demonstrate that the proof we give is every bit as trivial as one could hope. The example also demonstrates refinement for the order sorted case, and a refinement that does not use a straightforward inclusion of signatures.

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

```
obj NAT is
  sort Nat .
  op  0 : -> Nat .
  op  s : Nat -> Nat .
  op  p : Nat -> Nat .
  var N : Nat .
  eq  p(0)  =  0 .
  eq  p(s(N))  =  N .
endo
```

```
obj STACK is pr NAT .
  classes  NeStack Stack .
  subclass  NeStack < Stack .
  op  empty : -> Stack .
  op  push : Nat Stack -> NeStack .
  op  top_ : NeStack -> Stack .
  op  pop_ : NeStack -> Stack .
  var S : Stack .    var I : Nat .
  eq  top push(I,S)  =  I .
  eq  pop push(I,S)  =  S .
endo
```

The OBJ keyword sort precedes the declaration of a (visible) sort name, while for the purposes of this paper, we adapt standard OBJ notation to let class(es) declare a hidden sort name or names. 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.

We let the fixed universe of data values be given by the module NAT, together with its standard interpretation as ω, the naturals. (Technically, we require that the signature of NAT be extended with a constant for each natural number in ω, cf. the comments preceding Definition 32. This means that NAT should be extended with an infinite number of constants 1, 2, etc.) Thus, in the above specification the visible sort is Nat and the hidden sorts are NeStack and Stack.

In order to demonstrate the use of signature morphisms in refinement, we give a concrete refinement 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 .
  class  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 .
    class Stack .
    op  <<_;_>> : Nat Arr -> Stack .
    op  1st_ : Stack -> Nat .
    op  2nd_ : Stack -> Arr .
    op  empty : -> Stack .
    op  push : Nat Stack -> Stack .
    op  top_ : Stack -> Nat .
    op  pop_ : Stack -> Stack .
    var I N : Nat .   var S : Stack .   var A : Arr .
    eq  1st << N ; A >>  =  N .
    eq  2nd << N ; A >>  =  A .
    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  =  << p(1st S) ; 2nd S >> .
  endo
```

The signature morphism ϕ from the abstract to the concrete specification maps both NeStack and Stack to the single sort Stack, aud 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.

```
                         Nat  |->  Nat
              Stack, NeStack  |->  Stack
            empty : -> Stack  |->  empty : -> Stack
push : Nat Stack -> NeStack   |->  push : Nat Stack -> Stack
       top : NeStack -> Nat   |->  top : Stack -> Nat
     pop : NeStack -> Stack   |->  pop : Stack -> Stack
```

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

$r_{\texttt{NeStack,Stack}}$: Stack -> NeStack .

Because ϕ identifies Stack and NeStack, this operation is mapped to (cf. Proposition 45) the operation

$r_{\texttt{Stack,Stack}}$: Stack -> Stack .

But by Definition 27, the retract extension of the concrete specification includes the equation

$(\forall \texttt{S : Stack})\ r_{\texttt{Stack,Stack}}(\texttt{S}) = \texttt{S}$,

which means that $r_{\texttt{Stack,Stack}}$ is the identity function in the concrete specification, so we may safely ignore retracts in what follows. Moreover, because the names of the operations are unchanged by this mapping, we can denote the ϕ-translation of a term hy the term itself.

We now prove the correctness of this refinement of STACK, 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 63, there are two proof obligations. The first is that the left- and right-hand sides of each equation are related by $\smile$.

(14) top push(I, S) $\smile$ I

(15) 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:

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

Requirement (14) is trivial, because the left-hand side is equal, in the concrete specification, to I. To show (15) and (16), we use the following lemma, which states that values on the 'wrong side' of the pointer can be ignored.

Lemma 77 `<< 1st s ; put(x,2nd s,n) >>` $\smile$ `s` if for all $i \geq 0$ it is not the case that p^i(`1st s`) $=_{EC^{\otimes}}$ n (i.e., if n > `1st s`).

Proof: To show that *lhs* $\smile$ *rhs*, it is sufficient to show that $c[lhs] =_{EC^{\otimes}} c[rhs]$ for all contexts c built from top and pop. Such contexts are necessarily of the form top pop^i z, where pop^i denotes i applications of pop. We proceed by induction on i. For the basis we have (writing '=' for '$=_{EC^{\otimes}}$'):

```
        top << 1st s ; put(x,2nd s,n) >>
  =
        put(x,2nd s,n)[1st s]
  =          { n > 1st s }
        (2nd s)[1st s]
  =
        top s
```

For the induction step,

```
        top pop^i pop << 1st s ; put(x,2nd s,n) >> = top pop^i pop s
  ⇔
        top pop^i << p(1st s) ; put(x,2nd s,n) >> = top pop^i pop s
  ⇔
        top pop^i << 1st pop s ; put(x,2nd pop s,n) >> = top pop^i pop s
  ⇐          { induction hypothesis }
        (∀j ≥ 0) ¬(p^j(1st pop s) = n)
  ⇐
        (∀j ≥ 0) ¬(p^j(1st s) = n)
```

□

This lemma is the heart of the correctness proof; the remaining proof obligations are straightforward. To show (15):

```
        pop push(I,S) ⌣ S
  ⇔          { reduce left-hand side }
        << 1st S ; put(I, 2nd S, s(1st S)) >> ⌣ S
  ⇔          { Lemma 77 }
        true
```

Similarly, (16) is demonstrated as follows:

```
      push(x1,s1)⌣push(x2,s2)
⇔           { Proposition 61 }
      top push(x1,s1)⌣top push(x2,s2) ∧
      pop push(x1,s1)⌣pop push(x2,s2)
⇔           { top push(X,S) reduces to X }
      x1⌣x2 ∧ pop push(x1,s1)⌣pop push(x2,s2)
⇐           { (15) }
      x1⌣x2 ∧ s1⌣s2
```

This concludes the proof of refinement. Lemma 77, 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 the equations of the concrete specification; this can easily be done using a system like OBJ3 [23].

4.2 Example: 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 [19] 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 [19]) 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

```
a(o !) = a(o) .
```

We show that this specification is refined 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 refinement of the methods push and pop, and attributes top and rest, is given 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 refinement need only consider the hidden equations:

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

We use Proposition 63, with $G = \{\texttt{push}, \texttt{pop}\}$ and $D = \{\texttt{top}, \texttt{rest}\}$. This division is natural, because G contains all the methods of the abstract specification, and D all the attributes. The proof obligations are:

(17) rest pop SV $\smile$ rest rest SV

(18) rest push(N,SV) $\smile$ SV !

(19) SV1 $\smile$ SV2 $\Rightarrow$ push(N,SV1) $\smile$ push(N,SV2)

(20) SV1 $\smile$ SV2 $\Rightarrow$ pop SV1 $\smile$ pop SV2

We use the following lemma:

Lemma 78 **If** $\texttt{val SV1} = \texttt{val SV2}$ then $\texttt{SV1} \smile \texttt{SV2}$.

Proof: To show that $\texttt{SV1} \smile \texttt{SV2}$, it suffices to show that $c[\texttt{SV1}] = c[\texttt{SV2}]$ for all visible contexts c built from **top** and **rest**. Such contexts are necessarily of the form $\texttt{top rest}^i\ z$ for some $i \in \omega$. We proceed by induction on i. For the basis, we have:

$$\begin{array}{ll} & \texttt{top SV1} = \texttt{top SV2} \\ \Leftrightarrow & \\ & \texttt{top val SV1} = \texttt{top val SV2} \\ \Leftarrow & \\ & \texttt{val SV1} = \texttt{val SV2} \end{array}$$

For the induction step,

$$\begin{array}{ll} & \texttt{top rest}^i\ \texttt{rest SV1} = \texttt{top rest}^i\ \texttt{rest SV2} \\ \Leftrightarrow & \\ & \texttt{top rest}^i\ \texttt{(SV1! := pop val SV1)} = \texttt{top rest}^i\ \texttt{(SV2! := pop val SV2)} \\ \Leftarrow & \quad \{ \text{ induction hypothesis } \} \\ & \texttt{val(SV1! := pop val SV1)} = \texttt{val(SV2! := pop val SV2)} \\ \Leftrightarrow & \\ & \texttt{pop val SV1} = \texttt{pop val SV2} \\ \Leftarrow & \\ & \texttt{val SV1} = \texttt{val SV2} \end{array}$$

□

Now (17) and (18) are easy consequences. To show (19) we calculate as follows:

$$\begin{array}{ll} & \texttt{push(N,SV1)} \smile \texttt{push(N,SV2)} \\ \Leftrightarrow & \quad \{ \text{ Proposition 61 } \} \\ & \texttt{top push(N,SV1)} \smile \texttt{top push(N,SV2)} \ \wedge \\ & \texttt{rest push(N,SV1)} \smile \texttt{rest push(N,SV2)} \\ \Leftrightarrow & \quad \{ \text{ first conjunct trivial, definition of } \texttt{push} \} \\ & \texttt{rest (SV1 := push(N, val SV1))} \smile \\ & \texttt{rest (SV2 := push(N, val SV2))} \\ \Leftrightarrow & \quad \{ \text{ definition of } \texttt{rest} \} \\ & \texttt{(SV1:= push(N,val SV1))! := val SV1} \smile \\ & \texttt{(SV2:= push(N,val SV2))! := val SV2} \\ \Leftarrow & \quad \{ \text{ see below } \} \\ & \texttt{SV1} \smile \texttt{SV2} \end{array}$$

The last step uses the fact that $\texttt{SV := val SV}' \smile \texttt{SV}'$, which is a consequence of Lemma 78 and the transitivity of $\smile$.

Finally, (20) is demonstrated by the following calculation.

$$\begin{array}{ll} & \texttt{pop SV1} \smile \texttt{pop SV2} \\ \Leftrightarrow & \quad \{ \text{ Proposition 61 } \} \\ & \texttt{top pop SV1} \smile \texttt{top pop SV2} \ \wedge \ \texttt{rest pop SV1} \smile \texttt{rest pop SV2} \\ \Leftarrow & \quad \{ \text{ visible equations hold; (17) } \} \\ & \texttt{top rest SV1} \smile \texttt{top rest SV2} \ \wedge \ \texttt{rest rest SV1} \smile \texttt{rest rest SV2} \\ \Leftarrow & \quad \{ \smile \text{ is a } D\text{-congruence } \} \\ & \texttt{SV1} \smile \texttt{SV2} \end{array}$$

We conclude that the refinement is correct.

4.3 Example: history lookup

This example is based on Henniker [24], where the correctness proof is performed by context induction. Our proof does not need induction at all.

The abstract specification defines a class of abstract machines:

```
obj STATE is pr NAT .
             pr QID .
  class State .
  op init : -> State .
  op update : Id Nat State -> State .
  op lookup : Id State -> Nat .
  vars X Y : Id .
  vars M N : Nat .
  vars S : State .
  eq lookup(X,init) = 0 .
  eq lookup(X, update(Y,N,S)) = if eq(X,Y) then N else lookup(X,S) fi .
  eq update(X, M, update(Y,N,S)) = if eq(X,Y) then update(X,M,S)
                                   else update(Y,N,update(X,M,S)) fi .
endo
```

where QID is a module which defines a sort Id of identifiers: we assume that this has an equality predicate eq. Henniker [24] proves that this specification is correctly refined by an abstract machine that keeps a history of all updates; this abstract machine therefore does not satisfy the third equation of the above specification, although it does behaviourally satisfy it.

However, the third equation of this specification is superfluous, because any algebra which satisfies the first two equations (which are of visible sort) will necessarily behaviourally satisfy the third equation. Formally, we have $E \models\!\!\!\equiv e$, where E is the set consisting of the first two equations, and e is the third equation; in other words, all behavioural E-models behaviourally satisfy e. We can prove that $E \models\!\!\!\equiv e$ using the proof technique of Section 3.2, because what we are proving is that the specification STATE is refined by STATE′, where STATE′ is STATE minus the third equation.

To show the correctness of this refinement, let $G = \{\texttt{init}, \texttt{update}\}$ and $D = \{\texttt{lookup}\}$. We must show that all equations of the abstract specification are related by $\smile$, and that $\smile$ is a G-congruence. Obviously, the first two equations of STATE hold in STATE′, and are therefore related by $\smile$; as for the third equation, note that the only D-contexts are of the form lookup(V,z) for some identifier V. Therefore we need only show that for all identifiers V, the following equation holds in STATE′:

```
(∀ X, Y, M, N, S)
 lookup(V, update(X, M, update(Y,N,S))) =
 lookup(V, if eq(X,Y) then  update(X,M,S)
           else update(Y,N,update(X,M,S)) fi).
```

This can be shown by case analysis on the equality of X, Y and V. For example, if V = X = Y then

```
    lookup(V, update(X, M, update(Y,N,S)))
  = M
  = lookup(V, update(X,M,S))
  = lookup(V, if eq(X,Y) then  update(X,M,S)
              else update(Y,N,update(X,M,S)) fi)
```

and if $\mathtt{V} = \mathtt{X} \neq \mathtt{Y}$ then

```
    lookup(V, update(X, M, update(Y,N,S)))
  = M
  = lookup(V, update(X,M,S))
  = lookup(V, update(Y,N,update(X,M,S)))
  = lookup(V, if eq(X,Y) then  update(X,M,S)
              else update(Y,N,update(X,M,S)) fi).
```

The case where $\mathtt{V} \neq \mathtt{X}$ is similar.

Finally, to show that $\smile$ is a G-congruence, we need only show the following implication for all states S1 and S2: if

```
  lookup(V, S1) = lookup(V, S2)
```

for all V, then

```
  lookup(V, update(X,N,S1)) = lookup(V, update(X,N,S2))
```

for all V,X and N. This is straightforward to show by case analysis on eq(X,V).

Note that in this example we do not need induction on contexts; because the set D of derived functions contains no 'recursive' operations (i.e., operations which take states to states), unlike the stack examples above, we need show satisfaction of the abstract equations in only a finite number of contexts built from D, which leads to very simple proofs.

5 Conclusion

We have given definitions of refinement and implementation for hidden order sorted specifications, and a technique for proving correctness of refinement which is based on splitting the abstract signature into generators and derived functions. This technique leads to proofs based on equational logic which seem much simpler than other correctness proofs in the literature. Moreover, we have shown that this technique also applies to proofs of correctness of partial implementations. Our approach applies directly to the object paradigm by associating visible sorts with data types, and hidden sorts with object classes. The proof technique is being implemented in the mechanised theorem prover 2OBJ [21, 30].

As noted in Section 3.1, our definition of refinement generalises that of Henniker [24] to the order sorted case. Henniker proposes a form of context induction as a technique for proving correctness of refinements; the proof technique we develop in Section 3.2, based on splitting a signature into generators and derived functions, seems to simplify such proofs by reducing the number of case analyses in induction steps. Moreover, as is clear from the example in Section 4.3, induction on contexts is only necessary when an object has attributes (i.e., derived functions) of hidden sort, such as pop in the stack example. If there are no attributes of hidden sort, then the number of contexts is essentially the same as the number of attributes, and behavioural satisfaction can be proved very simply.

The proof technique of Proposition 63 first appeared in [16]. A recent technical report by Bidoit and Henniker [1] uses a similar approach to provide a technique for proving behavioural satisfaction of equations in many sorted algebra. In particular, they are interested in identifying sets of contexts which are sufficient to establish behavioural equivalence, in the sense that if $c[l] = c[r]$ for all c in a given set of contexts then l and r are behaviourally equivalent. They use a notion of 'Observability Kernel', a finitary first order formula which, in the terminology of our Proposition 63, states the following: behavioural D-equivalence is equal to behavioural equivalence if behavioural D-equivalence is a ϕG-congruence, particular case where φ is the identity signature morphism, where D contains all operations in Σ which take a hidden sort to a visible sort, and G contains all operations taking hidden sorts to hidden sorts. For example, this very special case handles the division into generators and derived functions used in the example of Section 4.3, but is not appropriate for either of the other examples we give. Bidoit and Henniker develop some general sufficiency results for sets of contexts to prove behavioural equivalence, but we believe that in most applications, such sets of contexts will be those arising from subsignatures of generators and derived functions.

The main difference between our definitions of implementation and refinement is that refinement requires all models of the concrete specification to give rise to models of the abstract specification, while implementation simply requires all *ground equations* of the abstract theory to be satisfied, up to observability and termination, in the concrete theory. The notion of implementation, whose definition depends on the notion of term, and in particular on the notion of error term, is therefore less abstract and less easily understood than that of refinement. In general, it is more difficult to prove the correctness of an implementation than a refinement, since proving equivalence up to termination may be very complex. However, the notion of refinement seems sufficiently powerful for most examples that arise in computer science: we did not succeed in finding a convincing example of an implementation that was not actually a refinement.

Our use of hidden order sorted algebra leads to an abstract treatment of states of objects, and to a similarly abstract treatment of object refinement. Although our use of vertical signature morphisms means that refinements are not, in general, expressed by theory morphisms, our definition of refinement nevertheless exploits the duality between theories and models that is captured by the theory of institutions. In particular, because refinements in our approach are expressed by behaviour-preserving vertical signature morphisms, a refinement simply translates the syntax of the abstract theory into the syntax of the concrete theory, thus avoiding the possibly messy details of how states are represented. In particular, we do not rely upon a mapping from the concrete representation to the abstract representation, as do many other approaches, e.g., [25, 2, 6, 4]. This can be a significant simplification.

One issue not addressed in this paper is concurrency. Hidden sorted specifications can be thought of as specifying systems of concurrent, interacting objects. Our approach to refinement is obviously applicable to serial evaluation by term rewriting (as in OBJ), but less obviously to concurrent models of computation. Goguen and Diaconescu [15] give a construction for the concurrent interconnection of a collection 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 [11]) which addresses such issues and extends our notion of refinement to concurrent, interacting systems.

Another issue not addressed is that of 'bounded refinements' [26, 27], where some kind of size restriction is imposed on the concrete object. For example, stacks might be refined by stacks of a fixed maximum depth. In such cases, the concrete specification raises errors where the abstract does not, which is exactly the opposite of our definition of refinement, which allows the concrete specification to handle errors raised by the abstract one. It would be interesting to investigate whether our approach could be adapted to cover bounded refinements, for example by using sort constraints [31] to treat the case where au error is raised by a bound beiug exceeded, in much the same way that Kamin and Archer [26] use preconditions to specify when a hound will not be exceeded.

A final area worth further investigation is the relationship between vertical and horizontal structuring operations. This issue was raised in an abstract way by Goguen and Burstall in [12], who pointed out the desirability of a 2-dimensional category structure, and it has been further investigated by Sannella and Tarlecki [28], Ehrig [6] and others, for a variety of different notions of refinement. Our definition of refinement is transitive in the sense that if $\phi_1 : S_1 \to S_2$ is a refinement and $\phi_2 : S_2 \to S_3$ is another, then so is $\phi_1;\phi_2 : S_1 \to S_3$, i.e., refinement is compositional. We intend to explore our definition of refinement in relation to some of the other 'Laws of Software Engineering' mentioned in [12].

References

[1] Michel Bidoit and Rolf Henniker. Proving behavionral theorems with standard first-order logic. Technical Report LIENS-94-11, Laboratoire d'Informatique de l'Ecole Normale Supérieure, September 1994.

[2] D. Bjørner and Cliff Jones. *Formal Specification and Software Development.* Prentice-Hall International, 1982.

[3] Rod Burstall and Răzvan Diaconescu. Hiding and behaviour: an institutional approach. In A. W. Roscoe, editor, *A Classical Mind: essays dedicated to C.A.R. Hoare.* Prentice-Hall International, 1994.

[4] José Felix Costa, Amilcar Sernadas, and Cristina Sernadas. Inductive objects. INESC, Lisbon, 1992.

[5] Răzvan Diaconescu, Joseph Goguen, and Petros Stefaneas. Logical support for modularisation. In Gerard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 83–130. Cambridge, 1993. Proceedings of a Workshop held in Edinburgh, Scotland, May 1991.

[6] Hartmut Ehrig, Hans-Jörg Kreowski, Bernd Mahr, and Peter Padawitz. Algebraic implementation of abstract data types. *Theoretical Computer Science*, 20:209-263, 1983.

[7] Marie-Claude Gaudel and I. Privara. Context induction: an exercise. Technical Report 687, LRI, Univ. Paris Sud, 1991.

[8] Joseph Goguen. Semantics of computation. In Ernest G. Manes, editor, *Proceedings, First International Symposium on Category Theory Applied to Computation and Control*, pages 234–249. University of Massachusetts at Amherst, 1974. Also in Lecture Notes in Computer Science, Volume 25, Springer, 1975, pages 151–163.

[9] 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.

[10] 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 Confereuce held at Oxford, June 1989.

[11] 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.

[12] Joseph Goguen and Rod Burstall. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical Report Report CSL-118, SRI Computer Science Lab, October 1980.

[13] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992. Draft appears as Report ECS-LFCS-90-106, Computer Science Department, University of Edinburgh, January 1990; an early ancestor is "Introducing Institutions," in *Proceedings, Logics of Programming Workshop*, Edward Clarke and Dexter Kozen, Eds., Springer Lecture Notes in Computer Science, Volume 164, pages 221–256, 1984.

[14] Joseph Goguen and Răzvan Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4, 1994.

[15] Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. Iu Hartmut Ehrig and Fernando Orejas, editors, *Recent Trends in Data Type Specification*. Springer-Verlag Lecture Notes in Computer Science 785, 1994.

[16] Joseph Goguen and Grant Malcolm. Proof of correctness of object representations. In A. W. Roscoe, editor, *A Classical Mind: essays dedicated to C.A.R. Hoare*, chapter 8, pages 119–142. Prentice-Hall International, 1994.

[17] Joseph Goguen and José Meseguer. Universal realization, persistent interconnection and implementatiou 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.

[18] Joseph Goguen and José Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11(3):307–334, 1985. Preliminary versions have appeared in: *SIGPLAN Notices*, July 1981, Volume 16, Number 7, pages 24–37; SRI Computer Scieuce Lab, Report CSL-135, May 1982; aud Report CSLI-84-15, Center for the Study of Language and Information, Stanford University, September 1984.

[19] 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.

[20] 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.

[21] Joseph Goguen, Andrew Stevens, Keith Hobley, and Hendrik Hilberdink. 2OBJ, a metalogical framework based on equational logic. *Philosophical Transactions of the Royal Society, Series A*, 339:69–86, 1992. Also in *Mechanized Reasoning and Hardware Design*, edited by C.A.R. Hoare and M.J.C. Gordon, Prentice-Hall, 1992. pages 69–86.

[22] Joseph Goguen, James Thatcher, and Eric Waguer. 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.

[23] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Algebraic Specification with OBJ: An Introduction with Case Studies.* Cambridge, to appear 1994. Also to appear as Technical Report from SRI International.

[24] 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.

[25] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[26] Samuel Kamin and Myla Archer. Partial implementations of abstract data types: a dissenting view on errors. In Giles Kahn, David MacQueen, and Gordon Plotkin, editors, *Semantics of Data Types.* Springer-Verlag Lecture Notes in Computer Science 173, 1984.

[27] Fernando Orejas, Marisa Navarro, and Ana Sánchez. Implementation and behavioural equivalence: a survey. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification.* Springer-Verlag Lecture Notes in Computer Science 655, 1993.

[28] Donald Sannella and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications. *Acta Informatica*, 25:233–281, 1988.

[29] Oliver Schoett. Behavioural correctness of data representations. *Science of Computer Programming*, 14:43–57, 1990.

[30] Andrew Stevens and Joseph Goguen. Mechanised theorem proving with 2OBJ: A tutorial introduction. Technical report, Programming Research Group, University of Oxford, 1993.

[31] Han Yan. *Theory and Implementation of Sort Constraints for Order Sorted Algebra.* PhD thesis, Programming Research Group, Oxford University, to appear 1994.