Towards an Algebraic Semantics for the Object Paradigm*

Joseph A. Goguen and Răzvan Diaconescu
Programming Research Group, Oxford University

Abstract: This paper surveys our current state of knowledge (and ignorance) on the use of hidden sorted algebra as a foundation for the object paradigm. Our main goal is to support equational reasoning about properties of concurrent systems of objects, because of its simple and efficient mechanisation. We show how equational specifications can describe objects, inheritance and modules; our treatment of the latter topic emphasises the importance of reuse, and the rôle of the so-called Satisfaction Condition. We then consider how to prove things about objects, how to unify the object and logic paradigms by using logical variables that range over objects, and how to connect objects into concurrent systems. We provide a universal characterisation of parallel connection, and more generally, of parallel connection with synchronisation, and show how the former construction gives a class manager that provides unique identifiers for its objects. The paper concludes with some topics for further research.

1 Introduction

This paper presents some initial steps towards an algebraic foundation for what we will call the *object paradigm*. This paradigm consists of specification, programming and refinement, in the context of objects, classes, inheritance and concurrency. These issues are increasingly important for applications to information systems, operating systems, protocols, programming languages, and more generally, to distributed systems of any kind, as well as to aspects of areas like conceptual modelling, natural language processing, knowledge representation and software engineering.

There are several reasons for preferring an algebraic approach. Perhaps the most compelling is the extensive algorithmic and theoretical support available for equational reasoning. This includes algorithms for term rewriting, unification, narrowing and completion, and theorems for the completeness of equational reasoning with respect to algebras, and the soundness of constructor induction. Many important problems are decidable or semi-decidable for equational logic that are not for more complex logics. Also, efficient implementations of many useful algorithms are available in a variety of tools. It follows that we should use equational logic wherever possible. Of course, it is *not* always possible; but this paper will show that many aspects of the object paradigm can be treated with equational reasoning.

This paper continues the programme initially sketched in [11], formally begun in [15], and further elaborated in [19]. This programme aims to generalise and integrate the classical initial algebraic semantics for Abstract Data Types (ADTs) with relevant aspects of process algebra, such as concurrency, hiding, non-determinism, complex data structures, complex objects, and sharing for objects and methods (i.e., synchronisation), while avoiding the *ad hoc* combination of process algebra and ADTs that is found for example in LOTOS.

*The research reported in this paper has been partially supported by the Science and Engineering Research Council, Fujitsu Laboratories Limited, the Information Technology Promotion Agency, Japan, as part of the R & D of Basic Technology for Future Industries "New Models for Software Architecture" project sponsored by NEDO (New Energy and Industrial Technology Development Organization), and the CEC under ESPRIT-2 BRA Working Groups 6071, ISCORE (Information Systems Correctness and REusability) and 6112, COMPASS (COMPrehensive Algebraic Approach to System Specification and development).

It seems important to consider complex systems that are built from several objects. For this purpose, we should have *modules* for the object paradigm, including *parameterised* modules, and not just inheritance; and for verification, we will of course need to have *semantics* for objects, modules, and their interconnections. An important property of modules in this context is that whatever we prove about a module remains true of that module wherever it is used; we will call this property *encapsulation*; it is vital for the reuse of modules, and differs from the weaker sense of encapsulation that refers to merely hiding the representation of data items. We will see that the so-called Satisfaction Condition plays an important rôle here. Our approach is "constraint based" rather than oriented towards input-output, in that we use equations to describe relationships that necessarily hold among attributes, methods, and processes.

Section 5 discusses how to prove things about objects, while Section 6 unifies the object and logic paradigms with logical variables that range over objects, and Section 7 considers the parallel connection of objects to form systems. The main results here are universal characterisations for parallel connection, and more generally, for parallel connection with synchronisation. We use the former construction to define a class manager for objects, each having a unique identifier. Because the result of a parallel connection (with or without synchronisation) is given by an equational specification, we can again apply equational reasoning to systems.

It is worth noting that the goal of our general programme for semantics, of which this paper is a part, is not to model the warts and tumours of existing languages (such as C++ or Smalltalk), but rather to construct simple but powerful new theories that will guide the design of new languages, and support efficient calculational reasoning about properties of their programs. Thus, our notions of inheritance and module are not merely based upon those of existing languages. Nevertheless, our notions can be used to give semantics for existing languages, although the details will necessarily involve a degree of awkwardness matching that of the language involved. We feel that the cleanliness of semantic-based design is particularly appropriate for languages (like FOOPS [23, 35] and OOZE [2, 1]) that are intended to be used for specification and prototyping[1].

2 Prerequisites and Preliminaries

We assume familiarity with basics of the object paradigm (e.g., the notions of object, class, attribute, method and inheritance), with overloaded many sorted algebra, and with the notation of OBJ (for which see [25]). In order to fix notation, and also to set the stage for later developments, we briefly review some basics of overloaded many sorted algebra; further details may be found in [16] and [30]. We emphasise overloading because of its importance for the object paradigm. Many readers may wish to skip directly to Section 3.

2.1 Overloaded Many Sorted Algebra

Given a **sort set** S, an S-**indexed** (or **sorted**) **set** A is a family $\{A_s \mid s \in S\}$ of sets indexed by the elements of S. In this context, $a \in A$ means that $a \in A_s$ for some $s \in S$. Similarly, $A \subseteq B$ means that $A_s \subseteq B_s$ for each $s \in S$, and an S-**indexed** (or **sorted**) **function** $f : A \to B$ is a family $\{f_s : A_s \to B_s \mid s \in S\}$. Also, we let S^* denote the set of all finite sequences of elements from S, with $[]$ the empty sequence. Given an S-indexed set A and $w = s_1 ... s_n \in S^*$, we let $A^w = A_{s_1} \times \cdots \times A_{s_n}$; in particular, we let $A^{[]} = \{\star\}$, some one point set.

A (n S-**sorted**) **signature** (S, Σ) is an $S^* \times S$-indexed set $\Sigma = \{\Sigma_{w,s} \mid w \in S^*,\ s \in S\}$; we often write just Σ instead of (S, Σ). Note that this definition permits *overloading*, in that the sets $\Sigma_{w,s}$ need *not* be disjoint. Call $\sigma \in \Sigma_{[],s}$ a **constant symbol** of sort s. A **signature morphism** Φ from a

[1] These remarks are in response to a criticism of the paper raised by one of its referees.

signature (S, Σ) to a signature (S', Σ') is a pair (f, g) consisting of a map $f\colon S \to S'$ of sorts and an $S^* \times S$-indexed family of maps $g_{w,s}\colon \Sigma_{w,s} \to \Sigma'_{f^*(w),f(s)}$ on operation symbols, where $f^*\colon S^* \to S'^*$ is the extension of f to strings[2]. We may write $\Phi(s)$ for $f(s)$, $\Phi(w)$ for $f^*(w)$, and $\Phi(\sigma)$ for $g_{w,s}(\sigma)$ when $\sigma \in \Sigma_{w,s}$.

A Σ-**algebra** A consists of an S-indexed set A and a function $\sigma_A\colon A^w \to A_s$ for each $\sigma \in \Sigma_{w,s}$; the set A_s is called the **carrier** of A of sort s. If $\sigma \in \Sigma_{[],s}$ then σ_A determines a point in A_s which may also be denoted σ_A. A Σ-**homomorphism** from one Σ-algebra A to another B is an S-indexed function $h\colon A \to B$ such that

$$h_s(\sigma_A(a_1, ..., a_n)) = \sigma_B(h_{s_1}(a_1), ..., h_{s_n}(a_n))$$

for each $\sigma \in \Sigma_{w,s}$ and $(a_1, ..., a_n) \in A^w$. (When $n = 0$, this condition just says that $f(\sigma_A) = \sigma_B$.) A Σ-homomorphism $h\colon A \to B$ is a Σ-**isomorphism** iff each function $h_s\colon A_s \to B_s$ is bijective (i.e., one-to-one and onto, in an older terminology).

Given a many sorted signature Σ, an S-indexed set X will be called a set of **variable symbols** if the sets X_s are disjoint from each other and from all of the sets $\Sigma_{w,s}$. Given a set X of variable symbols, we let $\Sigma(X)$ denote the signature formed by adding the elements of X to Σ as new constants, and we let $T_\Sigma(X)$ denote $T_{\Sigma(X)}$ viewed as a Σ-algebra. It is called the Σ-**term algebra** or **free** Σ-**algebra** generated by X, and has the property that if $\theta\colon X \to A$ is an **assignment**, i.e., a (S-sorted) function to a Σ-algebra A, then there is a unique extension of θ to a Σ-homomorphism $\theta^*\colon T_\Sigma(X) \to A$. (Strictly speaking, the usual term algebra is not free unless the constant symbols in Σ are mutually disjoint; however, even if they are not disjoint, a closely related term algebra, with each constant annotated by its sort, is free.) Also, we let T_Σ denote the **initial** term Σ-algebra $T_\Sigma(\emptyset)$, noting that this means there is a unique Σ-homomorphism $T_\Sigma \to A$ for any Σ-algebra A. Call $t \in T_\Sigma$ a **ground** Σ-**term**.

A special case of free extension is **substitution**, where the target algebra A is a term algebra, often $T_\Sigma(X)$ itself; in this case, we may use the notation $t(x_1 \leftarrow t_1, ..., x_n \leftarrow t_n)$ for $\theta^*(t)$ when $\theta(x_i) = t_i$ and the elements of X are $x_1, ..., x_n$. The following result is often useful:

Lemma 1: Given an assignment $\theta\colon X \to A$ and a Σ-homomorphism $f\colon A \to B$, then

$$(\theta; f)^* = \theta^*; f\colon T_\Sigma(X) \to B.$$

Proof: By freeness, there is exactly one Σ-homomorphism from $T_\Sigma(X)$ to B extending $\theta; f$. $\square$

A **conditional** Σ-**equation** consists of a variable set X, terms $t, t' \in T_\Sigma(X)_s$ for some sort s, and terms $t_j, t'_j \in T_\Sigma(X)_{s_j}$ for $j = 1, ..., m$. Such an equation is generally written in the form

$$(\forall X)\, t = t' \textbf{ if } t_1 = t'_1, ..., t_m = t'_m\ .$$

The special case where $m = 0$ is called an (**unconditional**) **equation**, written $(\forall X)\, t = t'$. A **ground equation** has $X = \emptyset$. A Σ-algebra A **satisfies** a conditional equation, written

$$A \models_\Sigma (\forall X)\, t = t' \textbf{ if } t_1 = t'_1, ..., t_m = t'_m\ ,$$

iff for all assignments $\theta\colon X \to A$, we have $\theta^*(t) = \theta^*(t')$ whenever $\theta^*(t_j) = \theta^*(t'_j)$ for $j = 1, ..., m$. Given a set E of (possibly conditional) Σ-equations, we call any Σ-algebra that satisfies E a (Σ, E)-algebra.

[2] This extension is defined by $f^*([]) = []$ and $f^*(ws) = f^*(w)f(s)$, for w in S^* and s in S.

2.2 Congruence and Quotient

A Σ-**congruence** on a Σ-algebra A is an S-sorted family of relations, $\equiv_s$ on A_s, each of which is an equivalence relation, and which also satisfy the **congruence property**, that given any $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ and any $a_i, a'_i \in A_{s_i}$ for $i = 1, ..., n$, then $\sigma_A(a_1, ..., a_n) \equiv_s \sigma_A(a'_1, ..., a'_n)$ whenever $a_i \equiv_{s_i} a'_i$ for $i = 1, ..., n$. The **quotient** of A by $\equiv$, denoted $A/\equiv$, has carriers $(A/\equiv)_s = A_s/\equiv_s$, which inherit a Σ-algebra structure by defining $\sigma_{A/\equiv}([a_1], \ldots, [a_n]) = [\sigma_A(a_1, \ldots, a_n)]$, where $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ and $a_i \in A_{s_i}$, where $[a]$ denotes the $\equiv$-equivalence class of a.

2.3 Equational Deduction

We now consider the *logic* of many sorted algebra, that is, rules for deducing new equations from old ones.

Definition 2: Given a set E of (possibly conditional) Σ-equations, we define the (unconditional) Σ-equations that are **derivable** from E recursively, by the following **rules of deduction**:

(0) Base: Each unconditional equation in E is derivable.

(1) Reflexivity: Each equation $(\forall X)\, t = t$ is derivable.

(2) Symmetry: If $(\forall X)\, t = t'$ is derivable, then so is $(\forall X)\, t' = t$.

(3) Transitivity: If $(\forall X)\, t = t'$ and $(\forall X)\, t' = t''$ are derivable, then so is $(\forall X)\, t = t''$.

(4) Congruence: If $(\forall X)\, t_i = t'_i$ is derivable, where $t_i, t'_i \in T_\Sigma(X)_{s_i}$ for $i = 1, ..., n$, then for any $\sigma \in \Sigma_{s_1 \ldots s_n, s}$, the equation $(\forall X)\, \sigma(t_1, \ldots, t_n) = \sigma(t'_1, \ldots, t'_n)$ is also derivable.

(5) Modus Ponens: Given $(\forall X)\, t = t'$ **if** $t_1 = t'_1, \ldots, t_m = t'_m$ in E and given a substitution $\theta\colon X \to T_\Sigma(Y)$ such that $(\forall Y)\, \theta^*(t_j) = \theta^*(t'_j)$ is derivable for $j = 1, ..., m$, then $(\forall Y)\, \theta^*(t) = \theta^*(t')$ is also derivable.

Given a set E of Σ-equations, let $E^{\bigcirc}$ denote the S-sorted set of pairs (t, t') of *ground* Σ-terms such that $(\forall \emptyset)\, t = t'$ is derivable from E. Then $E^{\bigcirc}$ is a Σ-congruence by rules (1)–(4). $\square$

The following completeness result was first proved by Goguen and Meseguer [20], although the unconditional one sorted form is very well known, going back to Birkhoff [4] in 1935:

Theorem 3: Given a set E of (possibly conditional) Σ-equations, an unconditional Σ-equation is satisfied by every (Σ, E)-algebra iff it is derivable from E using the rules (0)–(5). $\square$

Goguen and Meseguer [20] use the above to prove the following basic result:

Theorem 4: The Σ-algebra $T_{\Sigma,E} = T_\Sigma/E^{\bigcirc}$ is an initial (Σ, E)-algebra, in the sense that for any (Σ, E)-algebra A there is a unique Σ-homomorphism $h\colon T_{\Sigma,E} \to A$. $\square$

Of course, there are many other initial Σ, E-algebras, but they are all Σ-isomorphic to this one. Also, there are many other complete sets of rules of deduction, some of which are more convenient in practice; but the above rules are especially convenient for proving Theorem 4.

2.4 The Satisfaction Condition

Given signatures $\Sigma \subseteq \Sigma'$ and a Σ'-algebra M', let us write $M'\restriction_\Sigma$ for M' viewed as a Σ-algebra by dropping any operations in Σ' that are not in Σ, and also dropping any carriers of sorts in Σ' that are not in Σ; call this the **reduct** of M' to Σ. Moreover, given a Σ'-homomorphism $h\colon M'_1 \to M'_2$, we can form $h\restriction_\Sigma : M'_1\restriction_\Sigma \to M'_2\restriction_\Sigma$ by defining $(h\restriction_\Sigma)_s = h_s$ for $s \in S$ (the sort set of Σ); it is not difficult to show that this is a Σ-homomphism.

More generally, given a signature morphism $\Phi\colon \Sigma \to \Sigma'$ and a Σ'-algebra M', we can define the **reduct** of M' to Σ, denoted $\Phi(M')$ or $M'\restriction_\Phi$, to have carriers $M'_{\Phi(s)}$ for $s \in S$, and to have operations $\sigma_{\Phi(M')}$ for $\sigma \in \Sigma_{w,s}$ defined by $\sigma_{\Phi(M')}(m) = \Phi(\sigma)_{M'}(m)$ for $m \in M^w$. Also, given a Σ'-homomorphism $h\colon M'_1 \to M'_2$, we can define $h\restriction_\Phi : M'_1\restriction_\Phi \to M'_2\restriction_\Phi$ by $(h\restriction_\Phi)_s = h_{\Phi(s)}$ for $s \in S$.

Similarly, given a Σ-equation e of the form $(\forall X)\ t = t'$, we define $\Phi(e)$ to be the Σ'-equation $(\forall X')\ \overline{\Phi}(t) = \overline{\Phi}(t')$, where X' is the S'-indexed set, also denoted $\Phi(X)$, with $X'_{s'} = \bigcup_{\Phi(s)=s'} X_s$ for $s' \in S'$, and where $\overline{\Phi}\colon T_{\Sigma(X)} \to T_{\Sigma'(X')}$ is the S-indexed function defined by viewing $T_{\Sigma'(X')}$ as a $\Sigma(X)$-algebra using the reduct construction given above, and then the initiality of $T_{\Sigma(X)}$.

An important property of these translations on algebras and equations under signature morphisms is called the **Satisfaction Condition**, which expresses the invariance of satisfaction under change of notation:

Theorem 5: Given an overloaded many sorted signature morphism $\Phi\colon \Sigma \to \Sigma'$, a Σ'-algebra M', and a Σ-equation e, then

$$\Phi(M') \models_\Sigma e \text{ iff } M' \models_{\Sigma'} \Phi(e).$$

□

This theorem was first proved in the original version of [18]. We will later relate the Satisfaction Condition to the encapsulation of modules, and in Section 4.1, we will see that it is an important part of the formalisation of logical system called an *institution* by Burstall and Goguen [6, 17, 18].

2.5 Other Topics

Section 3.1 assumes familiarity with basic order sorted algebra, as given for example in [22]. For simplicity of exposition, this paper will first treat the many sorted case, and then treat the order sorted case more briefly afterwards. Some examples assume familiarity with some basics of term rewriting, including confluence, termination and narrowing; these are explained, for example, in [7] and [16].

We sometimes use basic notions from category theory, including category, functor, and initial object. For an introduction to category theory, see [3] or [27]. We use the "bbold" font to denote categories, e.g., $\mathbb{C}$. Given morphisms $f\colon A \to B$ and $g\colon B \to C$, we let $f;g$ denote their composition, a morphism $A \to C$; also, we let 1_A denote the identity morphism at an object A. Sections 4.1 and 7 use colimits, and Section 7 also uses universal constructions. A general discussion of category theory for Computing Science is given in [14].

3 Hidden Sorted Algebra

This section gives a brief summary of basic concepts from (overloaded many sorted) hidden sorted algebra, with some motivation, following the lines of [15]. The basic intuition is that an object has a *state*, which is *hidden*, i.e., only *observed* through the effect of methods on attributes. This approach is a variant of algebra that:

0. fixes an algebra D of data values; in typical applications, D might include natural numbers, booleans, character strings, etc.;
1. models states with *hidden sorts*;
2. models data with *visible sorts*;
3. models attributes with visible valued functions that are *monadic* in states, in the sense of having exactly one hidden sorted argument; and
4. models methods with hidden valued functions that are monadic in states.

The following is our first step in making this precise:

Definition 6: A **hidden sorted signature** consists of

1. a set S of **sorts**,
2. a subset $V \subseteq S$ of **visible** sorts, where $H = S - V$ is called the set of **hidden** sorts,
3. an S-sorted signature Σ,
4. a V-sorted subsignature $\Psi \subseteq \Sigma$ called the **data signature**, and
5. a Ψ-algebra D, called the **data algebra**,

such that

(S1) for each $d \in D_v$ with $v \in V$ there is some $\psi \in \Psi_{[],v}$ such that ψ is interpreted as d in D; for simplicity, we can assume that $D_v \subseteq \Psi_{[],v}$ for each $v \in V$,

(S2) each $\sigma \in \Sigma_{w,s}$ with $w \in V^*$ and $s \in V$ lies in $\Psi_{w,s}$, and

(S3) each $\sigma \in \Sigma_{w,s}$ has at most one element of w in H.

To indicate this entire situation, we may write (S, Σ, V, Ψ, D) or (Σ, D), or even just Σ if context permits. If $w \in S^*$ contains a hidden sort, then $\sigma \in \Sigma_{w,s}$ is called a **method** if $s \in H$ and an **attribute** if $s \in V$.

A **hidden sorted specification** is a tuple $(S, \Sigma, V, \Psi, D, E)$, where (S, Σ, V, Ψ, D) is a hidden sorted signature and E is a set of Σ-equations; we may abbreviate this (Σ, D, E) or even just (Σ, E) if the context permits. We will also call it a **module**. □

Condition (S1) says that all data items are named by constants in Ψ, while (S2) is a data encapsulation condition, that Σ cannot add any new operations on data items. Condition (S3) says that methods and attributes act on (the states of) single objects.

The following example may help to clarify this definition. The code in this and subsequent examples uses the syntax of OBJ3 [25], but the intended semantics is not initial algebra semantics, but rather a loose hidden sorted semantics that we will soon explain.

Example 7: We specify a flag object, where the intuitive idea is that a flag is either up or down, and there are methods to put it into either state, and also to reverse its state; when initially created, the flag is down.

```
obj FLAG is
   pr DATA .
   sort Flag .
   op newf : -> Flag .
   ops (up_) (dn_) (rev_) : Flag -> Flag .
   op up?_ : Flag -> Bool .
   var F : Flag .
   eq up? up F = true .
   eq up? dn F = false .
   eq up? rev F = not up? F .
   eq up? newf = false .
endo
```

Here `FLAG` is the name of the module and `Flag` is the name of the class of flag objects, representing the states of such objects. Then `newf` is a constant which represents the initial state of a flag object, while `up`, `dn` and `rev` are methods which change the state of flag objects, and `up?` is an attribute that tells whether or not the flag is up. The underbar characters indicate where an argument goes, so that all three methods and the attribute have prefix syntax.

The imported module `DATA` defines the data types that will be used, which should include at least the Booleans. Then Ψ is the signature of `DATA`, and D is some standard model for the given data types. Later, we will see that a model of this `FLAG` specification is an algebra such that its restriction to Ψ is D, providing functions for the given methods and attributes, and behaving as if it satisfied the given equations; i.e., it is an implementation that provides the given data, the one attribute, the three methods, and the desired behaviour.

In ordinary programming notation, one would write things like

```
up.F; dn.F; rev.F; up?.F
```

where `F` is some particular flag object, but in the more algebraic notation that we are using here, the above instead takes the form

```
up? rev dn F
```

□

To make our notion of model precise, we must first define behavioural satisfaction. Intuitively, this means that the two terms of an equation "look the same" under every "experiment," consisting of some methods, followed by an observation of some attribute. More formally, such an "experiment" is given by a *context*, which is a term of visible sort having one free variable of hidden sort. This motivates the following:

Definition 8: Given a hidden sorted signature (S, Σ, V, Ψ, D) and an S-sorted set X of variable symbols, then a Σ-**context** is a visible sorted Σ-term having a single occurrence of a new variable symbol z. Call such a context **appropriate** for a term t iff the sort of t matches the sort of z.

A Σ-algebra A **behaviourally satisfies** a Σ-equation $(\forall X)\ t = t'$ iff A satisfies each equation $(\forall X)\ c(t) = c(t')$ where c is an appropriate Σ-context; in this case, we may write $A \models_{\Sigma} (\forall X)\ t = t'$, and we may drop the subscript Σ if it is not needed.

Similarly, A **behaviourally satisfies** a conditional equation e of the form

$$(\forall X)\, t = t' \textbf{ if } t_1 = t'_1, ..., t_m = t'_m$$

iff for every interpretation $\theta\colon X \to A$, we have

$$\theta^*(c(t)) = \theta^*(c(t'))$$

for all appropriate contexts c whenever, for $j = 1, ..., m$,

$$\theta^*(c_j(t_j)) = \theta^*(c_j(t'_j))$$

for all appropriate contexts c_j. As with unconditional equations, we write $A \models_\Sigma e$. □

The algebraic approach to the object paradigm is prefigured in work of Goguen and Meseguer on (what they then called) abstract machines [20, 30]; the hidden sorted algebra approach differs from this mainly in its use of behavioural satisfaction for equations, an idea introduced by Reichel [32] in the context of partial algebras. Reichel [33] later introduced the related idea of behavioural equivalence for states, which we also use here later on.

Example 9: The following are some contexts for `FLAG`:

```
c(z) = up? z
c(z) = up? up rev z
c(z) = up? dn dn z
```

□

We can now make the idea of model precise:

Definition 10: Given a hidden sorted specification $(S, \Sigma, V, \Psi, D, E)$, then a **model** of that specification is a Σ-algebra A such that

1. $A\restriction_\Psi = D$, and
2. each equation in E is behaviourally satisfied.

A model of $(S, \Sigma, V, \Psi, D, E)$ is also called a $(S, \Sigma, V, \Psi, D, E)$-**algebra**, or a (Σ, D, E)-algebra for short. A Σ-algebra that satisfies just the first condition is called a (S, Σ, V, Ψ, D)-algebra, or for short, a (Σ, D)-algebra, or a **hidden sorted algebra**. □

The following gives some (abstract) implementations of flag objects to illustrate the above:

Example 11: Let's first define a simple Boolean cell C as a hidden sorted algebra. Here, $C_{\texttt{Flag}} = C_{\texttt{Bool}} = \{true, false\}$, `up` $F = true$, `dn` $F = false$, `up?` $F = F$, `rev` $F = not\ F$, and `newf` $= false$.

A more complex implementation H keeps complete histories of interactions, so that the action of a method is merely to concatenate its name to the front of a list of method names. Then $H_{\texttt{Flag}} = \{up, dn, rev\}^*$, the lists from $\{up, dn, rev\}$, while $H_{\texttt{Bool}} = \{true, false\}$, `up` $F = up^\frown F$, `dn` $F = dn^\frown F$, `rev` $F = rev^\frown F$, while `up?` $up^\frown F = true$, `up?` $dn^\frown F = false$, and `up?` $rev^\frown F = not$ `up?` F, where $^\frown$ is the concatenation operation.

Note that C and H are not isomorphic. □

For visible sorted equations, there is no difference between ordinary satisfaction and behavioural satisfaction. But these concepts can be different for hidden sorted equations. For example, the equation

```
eq rev rev F = F .
```

is strictly satisfied by the Boolean cell model C, but it is *not* satisfied by the history model H (however, it is *behaviourally* satisfied by both models).

3.1 The Order Sorted Case

We first consider a simple example, and then give the general definitions.

Example 12: It is very traditional to consider the stack object, which is interesting because one of its equations, namely

```
eq pop push(N,S) = S .
```

is not actually satisfied by some important models, including the traditional pointer-array pair implementation. Here is the code:

```
obj STACK is
  pr DATA .
  sorts NeStack Stack .
  subsort NeStack < Stack .
  op push : Nat Stack -> NeStack .
  op empty : -> Stack .
  op pop_ : NeStack -> Stack .
  op top_ : NeStack -> Nat .
  var S : Stack . var N : Nat .
  eq pop push(N,S) = S .
  eq top push(N,S) = N .
endo
```

Notice that this specification involves a subsort `NeStack` of the basic sort `Stack` of stack states; this subsort is for those stack states that are non-empty. Thus, this specification uses order sorted equational logic to classify the possible states of objects. A model is an order sorted algebra with the appropriate data subalgebra, having the `push` and `pop` methods and the `top` attribute (plus the constant `empty`), and behaving as if it satisfies the two equations.

The following contexts are appropriate for terms of the class `Stack`:

```
c(z) = top z
c(z) = top pop z
c(z) = top pop pop z
```

These contexts respectively select the top, second and third elements on a stack, if they exist. (Strictly speaking, the above are not really contexts for `STACK`, but rather for the enrichment of `STACK` by retracts, as described for example in [22], and implemented in OBJ3.) □

We are now ready to define hidden order sorted signatures and algebras; these extend our previous definitions for the many sorted case by adding an ordering relation $\leq$ on sorts.

Definition 13: Given an order sorted signature $(V, \leq, \Psi)$ with set V of visible sorts and an order sorted Ψ-algebra D, then $(S, \leq, \Sigma, V, \Psi, D)$ is a **hidden order sorted signature** if $(V, \leq, \Psi) \subseteq (S, \leq, \Sigma)$ as order sorted signatures, and (S, Σ, V, Ψ, D) is a hidden many sorted signature, such that no visible sort is related (by $\leq$) to any hidden sort.

A **hidden order sorted specification** or **module** is a tuple $(S, \leq, \Sigma, V, \Psi, D, E)$ where $(S, \leq, \Sigma, V, \Psi, D)$ is a hidden order sorted signature and E is a set of Σ-equations. We may abbreviate this to just (Σ, D, E), as in the many sorted case.

A **hidden order sorted** $(S, \leq, \Sigma, V, \Psi, D)$**-algebra** is an order sorted $(S, \leq, \Sigma)$-algebra M that is also a hidden many sorted (Σ, D)-algebra. A **homomorphism** of hidden order sorted algebras $h\colon M \to M'$ is a homomorphism of order sorted algebras that is also a homomorphism of hidden sorted algebras. □

The condition that visible and hidden sorts are unrelated under $\leq$ has the intuitive meaning that nothing visible can be a subsort of something hidden, and *vice versa*.

3.2 Inheritance and Modularity

There are many opinions about what inheritance is or should be. We believe that much confusion has resulted from not distinguishing carefully between subclasses and imported modules, or more fundamentally, between classes and modules (see [26] for a more detailed discussion of this). We believe that each of these is needed, and that they are different. Similarly, there are many different notions of encapsulation, which can also become confused by failure to distinguish between modules and classes. In our opinion, the fundamental notion of inheritance is at the module level rather than at the class level. Also, we believe that modules should support the declaration of more than one class (examples showing the need for this are given in [26]).

The basic intuition about modules is that they should support the reuse or replacement of code [31]. This requires that the *properties* of an imported module should be preserved by its importing module: we want old classes to behave the same way in their new context as they did in their original context. In fact, this requirement is often not satisfied by code that is written in the usual object oriented languages.

The following example will help us get started on our quest to formalise these issues.

Example 14: The module `CFLAG` below introduces a new subclass `CFlag` of `Flag` having a new attribute `count` that counts `up`'s and `down`'s (but ignores `rev`'s):

```
obj CFLAG is
   pr DATA .
   sort Flag .
   op newf : -> Flag .
   ops (up_) (dn_) (rev_) : Flag -> Flag .
   op up?_ : Flag -> Bool .
   var F : Flag .
   eq up? up F = true .
   eq up? dn F = false .
   eq up? rev F = not up? F .
   eq up? newf = false .

   sort CFlag .
   subsort CFlag < Flag .
   op newc : -> CFlag .
   ops (up_) (dn_) (rev_) : CFlag -> CFlag .
   op count_ : CFlag -> Nat .
   var C : CFlag .
   eq count newc  = 0 .
   eq up? newc = false .
   eq count up  C = count C + 1 .
```

```
    eq count dn  C = count C + 1 .
    eq count rev C = count C .
  endo
```

The first part of this module is exactly the same as `FLAG`; in particular, the second line says that it has the same data algebra (which should include at least the Booleans and natural numbers). The rest of the module defines the new subclass `CFlag` with its new constant `newc`, new attribute `count`, and with new (overloaded) versions of all the methods in `FLAG`. Notice that the subsort relationship here between `Flag` and `CFlag` is used to introduce a new class of object, without disturbing the old class `Flag`. It is also interesting to notice that the Boolean cell algebra C cannot be extended with a `count` attribute that will make it a model of `CFLAG`, but the history algebra H can be so extended.

By using OBJ's module import notation, we can simplify the above code:

```
  obj CFLAG is
    pr DATA .
    pr FLAG .
    sort CFlag .
    subsort CFlag < Flag .
    op newc : -> CFlag .
    ops (up_) (dn_) (rev_) : CFlag -> CFlag .
    op count_ : CFlag -> Nat .
    var C : CFlag .
    eq count newc  = 0 .
    eq up? newc = false .
    eq count up  C = count C + 1 .
    eq count dn  C = count C + 1 .
    eq count rev C = count C .
  endo
```

The line[3] `pr FLAG` above indicates that the code for `FLAG` is imported in such a way that every model of `CFLAG` also gives a model of `FLAG` (as already noted, the converse of this does not hold). □

The claim that properties of `FLAG`-models should also hold for the `FLAG`-part of a `CFLAG`-model expresses the encapsulation of the module `FLAG` within the module `CLFAG`. To formalise this, notice that there is an *inclusion* from the signature of `FLAG` into the signature of `CFLAG`, and that this inclusion satisfies some rather special conditions, which are given in the following:

Definition 15: Given hidden order sorted signatures Σ and Σ', then a **hidden order sorted signature morphism** $\Phi\colon \Sigma \to \Sigma'$ is an order sorted signature morphism $\Phi = (f, g)\colon \Sigma \to \Sigma'$ such that:

(M1) $f(v) = v$ for each $v \in V$;

(M2) $g(\psi) = \psi$ for each $\psi \in \Psi$;

(M3) $f(H) \subseteq H'$ (where $H' = S' - V$ and S' is the sort set of Σ');

(M4) if $\sigma' \in \Sigma'_{w',s'}$ and some sort in w' lies in $f(H)$, then $\sigma' = g(\sigma)$ for some $\sigma \in \Sigma$; and

[3] This declaration differs from the previous declaration "`pr DATA`" in that `DATA` has an initial semantics, whereas `FLAG` has a loose behavioural semantics.

(M5) for any hidden sorts h, h', if $f(h) < f(h')$ then $h < h'$.

For the many sorted case, the condition (M5) is omitted. □

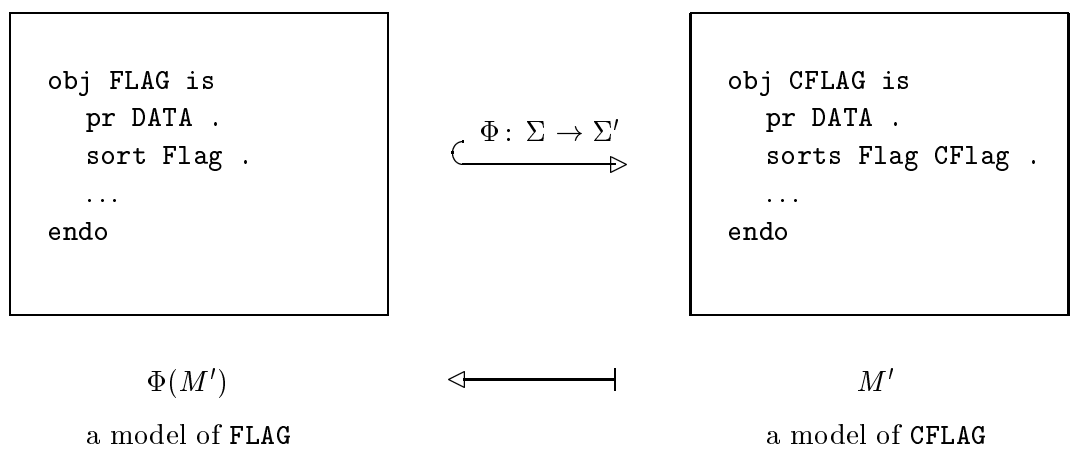
The first three conditions say that hidden sorted signature morphisms preserve visibility and invisibility for both sorts and operations, while the fourth and fifth conditions express the *encapsulation* of classes and subclasses, in the sense that no new methods or attributes can be defined on any imported class, and that any sublcass relation between images of hidden sorts comes from a relation between their sources. A morphism of modules, i.e., of hidden sorted specifications, must satisfy an additional condition:

Definition 16: Given hidden sorted specifications (Σ, D, E) and (Σ', D, E'), then a hidden sorted signature morphism $\Phi\colon (\Sigma, D) \to (\Sigma', D)$ is a **morphism** $\Phi\colon (\Sigma, D, E) \to (\Sigma', D, E')$ iff

(M6) $M' \models_{\Sigma'} E'$ implies $\Phi(M') \models_{\Sigma} E$,

for all Σ'-algebras M', where the **reduct** $\Phi(M')$ of M' to Σ is M' viewed as a Σ-algebra, just as in many sorted algebra. □

The diagram below may help to visualise this situation:



4 The Hidden Equational Institution

This section shows that the Satisfaction Condition (previously stated as Theorem 5 for overloaded many sorted algebra) also holds for hidden sorted algebra. This condition has methodological significance, because it implies that whatever we prove about the objects defined by some module P remains true in the context of some module P' that imports P. This property is crucial to the reuse of modules: if we define stacks and then use them in some new program, but they no longer behave the same way, then we will have to start from scratch in trying to understand the new code; we want to import not just code, but also *properties* of code.

Theorem 17: Given a hidden many sorted signature morphism $\Phi\colon \Sigma \to \Sigma'$, a Σ'-algebra M', and a Σ-equation $e = (\forall X)\ t_0 = t_1$, then

$$\Phi(M') \models_{\Sigma} e \text{ iff } M' \models_{\Sigma'} \Phi(e).$$

Proof: We have to show that for all appropriate contexts c and c',

$$\Phi(M') \models_\Sigma (\forall X)\ c(t_0) = c(t_1)$$

iff

$$M' \models_{\Sigma'} (\forall X')\ c'(\Phi(t_0)) = c'(\Phi(t_1)).$$

The "if" direction follows from Theorem 5, the Satisfaction Condition for ordinary many sorted equational logic, plus the fact that $\Phi(c(t_i)) = \Phi(c)(\Phi(t_i))$.

For the converse, conditions (S3) and (M4) of the definitions of hidden sorted signature and morphism imply that every context c' for $\Phi(e)$ is of the form $c''(\Phi(c))$ where $c'' \in T_{\Psi \cup \{z'\}}$ and c is a context for e. Therefore it suffices to show that

$$M' \models_{\Sigma'} (\forall X')\ \Phi(c)(\Phi(t_0)) = \Phi(c)(\Phi(t_1)).$$

But this follows from Theorem 5 plus the assumption, again using the fact that $\Phi(c(t_i)) = \Phi(c)(\Phi(t_i))$. □

We find it rather exciting that the conditions (S0–S3) and (M1–M4), which express encapsulation in the object paradigm, can be (and in fact, were) determined by the need to make the above proof of the Satisfaction Condition work. This derivation of the methodological principle of encapsulation from the metamathematical principle of the invariance of truth under change of notation seems to confirm the naturalness of both principles.

Note that the result and proof given above treat the many sorted case, following [15]. Although the result generalises to the order sorted case, the proof does not, and the interested reader should instead consult the proof of Burstall and Diaconescu in [5]. The result also generalises to conditional equations.

The next result makes precise our intuition that properties true of an imported module in isolation remain true of that module in its new context. To state this result, we must first explain what it means for an equation to be true of a module: it is just that the equation is satisfied by all correct implementations of the module. This semantic entailment relation is obtained from our basic behavioural satisfaction relation as follows:

Definition 18: Let $P = (\Sigma, D, E)$ be a specification and let M be a (Σ, D)-algebra. Then we let $M \models\!\!\equiv_\Sigma P$ mean that $M \models\!\!\equiv_\Sigma e$ for each $e \in E$. If e is a Σ-equation, then we let $P \models\!\!\equiv_\Sigma e$ mean that for any (Σ, D)-algebra M, if $M \models\!\!\equiv_\Sigma P$ then $M \models\!\!\equiv_\Sigma e$. □

Corollary 19: Let $P = (\Sigma, D, E)$ and $P' = (\Sigma', D, E')$ be hidden sorted specifications with $P \subseteq P'$ (i.e., with $\Sigma \subseteq \Sigma'$ and $E \subseteq E'$); let $\Phi\colon \Sigma \to \Sigma'$ denote the inclusion, and let e be a Σ-equation. Then

$$P \models\!\!\equiv_\Sigma e \text{ implies } P' \models\!\!\equiv_{\Sigma'} e.$$

Proof: First note that $\Phi(e) = e$ for any Σ-equation e, and thus that $\Phi(E) = E$. Now let M' be a Σ'-algebra satisfying E'. Because $E \subseteq E'$, we have $M' \models\!\!\equiv_{\Sigma'} E$, so that $\Phi(M') \models\!\!\equiv_\Sigma E$, by the Satisfaction Condition. Therefore $\Phi(M') \models\!\!\equiv_\Sigma e$ by hypothesis, and hence $M' \models\!\!\equiv_{\Sigma'} e$, again by the Satisfaction Condition. □

4.1 Institutions

We now have all the ingredients from hidden sorted algebra needed to form what is called an *institution*, which is a formalisation of the notion of "logic." These ingredients are as follows:

- signatures and their morphisms,
- Σ-sentences, for each signature Σ, and
- Σ-models, for each Σ,

such that

- signatures form a category, denoted $\mathbb{S}ign$,
- sentences are functorial on signatures,
- models are functorial on signatures, and
- the Satisfaction Condition holds,

$$\Phi(M') \models_{\Sigma} e \text{ iff } M' \models_{\Sigma'} \Phi(e),$$

where M' is a Σ'-model, e is a Σ-sentence, and $\Phi\colon \Sigma \to \Sigma'$ is a signature morphism.

A formalisation of logic that is adequate for Computing Science should encompass both syntax and semantics. On the one hand, *syntax is fundamental*, because we must say things in a finite amount of space: we manipulate finite texts, which (partially) describe real world entities, such as VLSI chips, bank balances, and computations, and we perform computations on such texts, such as proofs, translations, and type checks.

On the other hand, *semantics is fundamental*, because we are really interested in the *models*, rather than their descriptions; that is, we are really interested in chips, computations, etc.; we manipulate the syntactic representations only because we hope they correspond to the reality[4].

There is a reconciliation of these two views of what is fundamental that has been traditional since Tarski's classic semantic definition of truth for first order logic [37], based on the notion of *satisfaction*, seen as a binary relation between models and sentences (or more generally, sets of models and sets of sentences). Some such notion is needed for the basic notions of soundness and completeness of logical systems, because these notions depend in an essential way on the relationship between provability (which is syntactic) and satisfaction (which is semantic, i.e., concerns "truth" in Tarski's sense). In turn, these notions are fundamental to classical treatments of the adequacy of rules of deduction for logical systems: soundness and completeness with respect to an intuitively plausible class of models gives us confidence in some set of rules of deduction[5].

The theory of institutions allows us to discuss the pivotal relationship between specifications and models without committing to either side at the expense of the other. Institutions are much more

[4] Actually, this is a somewhat naive view, because we never really "have" the models, but only descriptions of models. Moreover, these descriptions generally use powerful idealistic constructions, such as power sets and dependent types, whose ultimate consistency cannot be resolved.

[5] There are also non-classical approaches to mathematics, e.g., the type theories of Martin-Löf, which do not appeal to notions of satisfaction, soundness and completeness, but rather regard their rules of inference as intuitively self-evident. There is room to doubt such claims, because of the contradictions that have been found in some such systems. However, it may be possible to construct appropriate set theories for such systems, and then construct appropriate models within such set theories.

abstract than Tarski's classical approach to truth, and also add another basic ingredient: signatures, and the possibility of translating sentences and models from one signature to another. The Satisfaction Condition, that truth is invariant under change of notation, is surely a very basic intuition for all classical logics.

Signatures are important for Computing Science because we consider not just "pure" logics, but rather "applied" logics, that use special "non-logical" symbols to capture the particularities of application areas, e.g., VLSI chip design for a fast Fourier transform. Category theory suggests treating not just the objects, which in this case are signatures, but also the relevant structure preserving morphisms, i.e., signature morphisms; it is these that enable us to consider how sentence, model, and satisfaction vary under change of notation.

Definition 20: An **institution** consists of

1. a category $\mathbb{S}ign$, whose objects are called **signatures**,
2. a functor $Sen\colon \mathbb{S}ign \to \mathbb{S}et$, giving for each signature a set whose elements are called **sentences** over that signature,
3. a functor $Mod\colon \mathbb{S}ign^{op} \to \mathbb{C}at$ giving for each signature Σ a category whose objects are called Σ-**models**, and whose arrows are called Σ-(**model**) **morphisms**, and
4. a relation $\models_\Sigma \subseteq |Mod(\Sigma)| \times Sen(\Sigma)$ for each $\Sigma \in |\mathbb{S}ign|$, called Σ-**satisfaction**,

such that for each morphism $\Phi\colon \Sigma \to \Sigma'$ in $\mathbb{S}ign$, the **Satisfaction Condition**

$$M' \models_{\Sigma'} Sen(\Phi)(e) \text{ iff } Mod(\Phi)(M') \models_\Sigma e$$

holds for each $M' \in |Mod(\Sigma')|$ and $e \in Sen(\Sigma)$. We may write $\Phi(e)$ for $Sen(\Phi)(e)$, and $\Phi(M')$ for $Mod(\Phi)(M')$. $\square$

A perhaps surprising amount of Computing Science can be done over any institution; [18, 9] and [34] give many results. For example, a version of Corollary 19 holds for any institution. For this paper, it is especially significant that the notions of module, generic (parameterised) module, sum (combination) of modules, importation (inheritance) of modules, etc. can all be defined (using colimits of specifications) over any institution whose signatures admit colimits (details are given in [6, 18, 10]). This includes all the institutions that interest us, such as hidden sorted equational logic over over a fixed Ψ and D. The following result (from [18]) supports this approach, and is also used in Section 7.

Theorem 21: If the category $\mathbb{S}ign$ of signatures of an institution $\mathcal{I}$ has all (finite) colimits, then so does the category $\mathbb{S}pec(\mathcal{I})$ of all specifications over $\mathcal{I}$. $\square$

Some researchers have argued that hidden sorted algebra is not really an institution, because without conditions like (M1–M4) the Satisfaction Condition does not hold. However, the most general possible morphisms are *not* necessarily the right ones. For example, although functions work well for sets, they are not the right morphisms for groups, because arbitrary functions do not preserve the group structure. Similarly for Banach spaces, morphisms should preserve all the relevant structure – not just the algebraic structure of vector spaces, but also the norm, in an appropriate sense. Thus, because the purpose of hidden sorts in hidden sorted algebra is to encapsulate states, we should expect to preserve the structure relating to encapsulation, and not just the operations in signatures.

5 Techniques for Verification

This section considers how to prove things about objects. The approach is to try to reduce reasoning about behavioural satisfaction to reasoning about ordinary satisfaction using ordinary equational deduction. The results in this section and the next are joint work with Tom Kemp, and follow [19].

Theorem 22: Ordinary equational reasoning (substituting equals for equals) is sound for behaviourally satisfied equations. □

But it is not complete. For example, we can prove

```
eq up? up dn F = true .
```

just by substitution, but we cannot prove

```
eq rev rev F = F .
```

in this way. The following result is originally due to Burstall and Diaconescu [5]; a more concrete proof is given in [19]:

Theorem 23: Given a hidden sorted Σ-algebra A, there is an ordinary Σ-algebra $\beta_\Sigma(A)$ such that

$$A \models\!\!\equiv e \text{ iff } \beta_\Sigma(A) \models e,$$

for any equation (or conditional equation) e. □

This result allows us to reason about $\beta_\Sigma(A)$ instead of A. Moreover, it extends to all first order sentences. Here is how $\beta_\Sigma(A)$ is constructed:

Given $a, a' \in A_s$ define $a \equiv_s a'$, and say a and a' are **behaviourally equivalent**, iff $c_A(a) = c_A(a')$ in A, for any s-sorted context c. Then $\equiv$ is a Σ-congruence relation on A, and so we can form the quotient $A/\equiv$, another Σ-algebra, denoted $\beta_\Sigma(A)$.

Unfortunately, it can be hard to see exactly what $\beta_\Sigma(A)$ looks like, and thus hard to use this result for practical reasoning. There is another construction that often yields a term algebra that is more convenient for reasoning. Let E be a set of Σ-equations, and let $\mathrm{Eq}(D)$ be the set of Ψ-equations satisfied by D. Now let Q be the Σ-congruence relation on T_Σ defined by

$$tQt' \text{ iff } (\forall\emptyset)\ t = t' \text{ is deducible from } E \cup \mathrm{Eq}(D).$$

Finally, let $T = T_\Sigma/Q$. This is almost what we want; it can be given the structure of a hidden sorted algebra under certain modest conditions.

Definition 24: A set E of Σ-equations is D-**protecting** iff

- for all $d, d' \in D$, if dQd' then $d = d'$, and
- for any $t \in T_{\Sigma,v}$ with $v \in V$, there is some $d \in D_v$ such that tQd.

□

Theorem 25: Let (Σ, D) be a hidden sorted signature and E a D-protecting set of Σ-equations, then:

(0) There is a hidden sorted Σ-algebra $G^D_{\Sigma,D,E}$ (denoted just G for short) that behaviourally satisfies E, and that is Σ-isomorphic to T_Σ/Q as an ordinary Σ-algebra.

(1) If G satisfies some (possibly conditional) Σ-equation then every reachable (Σ, D, E)-algebra behaviourally satisfies it.

(2) If G behaviourally satisfies some (possibly conditional) ground Σ-equation then every (Σ, D, E)-algebra behaviourally satisfies it.

(3) If all equations in E are unconditional, then (1) and (2) above can be made "iff" rather than "if ... then" assertions.

□

In general, proving behavioural satisfaction for G requires induction over all possible contexts.

Example 26: Suppose we want to establish that the equation

```
eq rev rev F = F .
```

is (behaviourally) satisfied by all models of `FLAG`.

The equations in `FLAG` are a canonical term rewriting system, so we can use the canonical (i.e., reduced) terms as (an algebra Σ-isomorphic to) G. Any appropriate context must be of the form $\texttt{up?}(c(z))$. Then we use induction on the form of c to prove that

$$\texttt{up?}(c(\texttt{rev rev } F)) = \texttt{up?}(c(F))$$

for all appropriate terms $c(z)$.

(0) The base case is $c(z) = z$, in which each side reduces to $\texttt{up? } F$.

(1) For the induction step, there are three cases,

  (a) $c(z) = \texttt{up } c'(z)$,
  (b) $c(z) = \texttt{dn } c'(z)$, and
  (c) $c(z) = \texttt{rev } c'(z)$,

  where in each case the equation to be proved is assumed for c'.

All four cases can be proved by reduction, because the two sides have the same canonical form. The transcript of an OBJ3 proof score for this is given in Appendix A. □

6 Existential Queries

Our motivation is to unify the object and logic paradigms, by allowing *queries* over (systems of) objects involving so-called "logical variables" that can range over objects. This will support a novel programming style, in which framing a query can activate methods that change the world so that a solution object actually comes to exist. In applications, this can be realised over an object oriented data base. Let us consider two examples:

- *A travel agency.* Consider a query about a holiday package, having constraints on cost, flight times, seat assignments, hotels, expected weather, and so on; then a solution to this query would be an actual holiday package, providing tickets, reservations, visas, etc. that satisfy the constraints (of course, subject to user approval before commitment).
- *A software factory.* Consider a query about an operating system, with constraints on number of disc drives, amount of core memory, external connections, etc.; then a solution would be a loadable image of an actual properly configured operating system, assembled from generic components in the software library.

One can easily imagine other applications where answering an active query over a domain specific objectbase actually creates new objects that satisfy the given constraints; for example, there are many such applications in the area of Computer Aided Design. This new paradigm differs from traditional logic programming in that permanent storage is permanently altered; for example, seats may be reserved in an airline database, or an executable core image of an operating system may be created. The particular way that this approach is based on logic means that it supports the powerful generic module facilities of parameterised programming, in the sense of [10, 12].

In order to know that this new paradigm for computing works as desired, we need a *hidden Herbrand theorem*, which reduces reasoning over arbitary implementations (e.g., of object oriented databases) to reasoning over a single term algebra (a "Herbrand universe"). The result below is joint work with Tom Kemp, and follows [19].

Definition 27: An (**existential**) Σ-**query** is a sentence of the form

$$(\exists X)\ t_1 = t'_1, \ldots, t_m = t'_m$$

and it is **behaviourally satisfied** by a Σ-algebra A iff there is some substitution $\theta\colon X \to A$ such that, for $j = 1, ..., m$,

$$\theta^*(c_j(t_j)) = \theta^*(c_j(t'_j))$$

for all appropriate contexts c_j. $\square$

The following is the basic result:

Theorem 28: If (Σ, D) is a hidden sorted signature and E is a D-protecting set of Σ-equations, then whenever G behaviourally satisfies a Σ-query q, so does every (Σ, D, E)-algebra. $\square$

When G is (isomorphic to) a canonical term algebra, we can use narrowing to solve queries, for example, over arbitrary implementations of object oriented databases (narrowing is described in [7], among other places, and the use of narrowing to solve queries is illustrated in [19].)

We note that the definitions and results in this and the previous section extend to Horn clause logic with equality, by applying a construction that reduces that logic to hidden sorted equational logic. This is described in [19], extending some earlier work of Diaconescu [8].

7 Concurrent Systems of Objects

To motivate the general constructions that are given below, we first consider a parallel connection of two simple objects:

Example 29: We begin by defining an integer cell X with the following OBJ3 code:

```
obj X is
  pr DATA .
  sort X .
  op init : -> X .
  op putx : Nat X -> X .
  op getx_ : X ->  Nat .

  var X : X .  vars M N : Nat .
  eq getx init      = 0 .
  eq getx putx(N,X) = N .
endo
```

It is interesting to notice that the equation

```
  eq putx(M,putx(N,X)) = putx(M,X) .
```

is a behavioural consequence of the above specification. Also, it is interesting that the integer-valued expressions in one hidden variable X (of sort `X`), i.e., the *contexts*, are the polynomials in the variable X. We may also call these *attribute expressions*; they form a polynomial ring $\mathbb{Z}[X]$. Next, let us define the cell Y in exactly the same way that we did X, but with `X` and `x` everywhere replaced by `Y` and `y`, respectively.

The parallel connection of X and Y should be described by a specification $X \parallel Y$ that has a single hidden sort, with all the operations of X and Y having the same semantics as before, in such a way that the operations in X and Y do not interfere with each other. Thus it seems intuitively clear that $X \parallel Y$ should be just the union of the specifications X and Y, with their sorts identified, and with some new equations to express the independence of X and Y:

```
obj XY is
  pr DATA .
  sort S .
  op init : -> S .
  ops putx puty : Nat S -> S .
  ops (getx_) (gety_) : S ->  Nat .

  var S : S .  vars M N : Nat .
  eq getx init = 0 .
  eq gety init = 0 .
  eq getx putx(N,S) = N .
  eq gety puty(N,S) = N .
  eq getx puty(N,S) = getx S .
  eq gety putx(N,S) = gety S .
endo
```

The last equations equations, plus

```
  eq putx(M,puty(N,S)) = puty(N,putx(M,S)).
```

(which is a behavioural consequence of the specification `XY`) express the independence of X and Y within their parallel connection. Notice that the attribute expressions for `XY` form a ring $\mathbb{Z}[X, Y]$ of polynomials in two variables, X, Y. Notice that the `init` methods from `X` and `Y` have been identified to a single method in `XY`. □

It is fairly straightforward to generalise the above construction of `XY` to the parallel connection of any set of specifications; Section 7.1 below gives the precise details, and also provides a suitable universal characterisation of the construction. But first, let us consider a somewhat more sophisticated example:

Example 30: Consider a simple manager for a class of objects, each of which is a cell that can hold one natural number. There is a method for updating the value in a cell, and an attribute that gives the value stored in a cell. We will use OBJ3's built-in module `QID` (for identifiers as strings with an initial quote symbol) to supply unique identifiers for objects.

```
obj NAT? is
  pr NAT + QID .
  sort Nat? .
  subsort Nat < Nat? .
  op bad : -> Nat? .
endo

obj MGR is
  pr NAT? .
  sort St .
  op init : -> St .
  op put : Id Nat St -> St .
  op get : Id St ->  Nat? .

  vars I J : Id .  var N : Nat .  var S : St .
  eq get(I,init)        = bad .
  eq get(I,put(I,N,S))  = N .
  cq get(I,put(J,N,S))  = get(I,S) if I =/= J .
endo
```

The attribute expressions for `MGR` form a ring $\mathbb{Z}[\mathtt{Id}]$ of polynomials in variables corresponding to the unique identifiers for the cells, where each cell has the following form:

```
obj CELL is
  pr NAT? .
  sort St .
  op init : -> St .
  op put  : Nat St -> St .
  op get_ : St ->  Nat? .

  var N : Nat .  var S : St .
  eq get init     = bad .
  eq get put(N,S) = N .
endo
```

This kind of cell is more sophisticated than the simple cells X and Y considered in Example 29 above, because of the error handling capability; in particular, there is an "inactive" state before a value is assigned. The next subsection defines a way to connect a number of copies of `CELL` in parallel to form a system that (in some appropriate sense) is the same as the system specified by `MGR`. The following equation, which together with the last equation of `MGR` expresses independence, is a behavioural consequence of `MGR`:

```
cq put(I,N,put(J,M,S)) = put(J,M,put(I,N,S)) if I =/= J .
```

Note that it would not be very difficult to enrich this specification with "meta level" methods, for example, to increment all currently active cells. □

It seems reasonable to expect that this approach will generalise to the dynamic creation and destruction of objects in more complex systems; the approach can also be considered a (weak form of) reflection, because it provides a "metaobject" for managing each class.

7.1 Universal Property for Parallel Connection

This subsection provides a universal characterisation for the parallel connection of hidden sorted specifications having just one hidden sort, which is denoted h. A basic semantic property of the parallel connection is that its components do not interact. Definition 31 below expresses this property precisely; in it, the notation $(\forall \underline{X})$ stands for $(\forall x_1)(\forall x_2)...(\forall x_n)$ where each x_i is a visible sorted variable and $n > 0$.

Definition 31: Given hidden sorted specifications P_1 and P_2 (with just one hidden sort), then hidden sorted specification morphisms of the form $P_1 \xrightarrow{\varphi_1} P \xleftarrow{\varphi_2} P_2$ are **independent** iff $\varphi_1(h) = \varphi_2(h)$, and for all $i, j \in \{1, 2\}$ with $i \neq j$, we have

$$P \models (\forall \underline{X})(\forall \underline{Y})(\forall S)\ \varphi_i(a_i)(\underline{X}, \varphi_j(m_j)(\underline{Y}, S)) = \varphi_i(a_i)(\underline{X}, S),$$

$$P \models (\forall \underline{X})(\forall \underline{Y})(\forall S)\ \varphi_i(m_i)(\underline{X}, \varphi_j(m_j)(\underline{Y}, S)) = \varphi_j(m_j)(\underline{Y}, \varphi_i(m_i)(\underline{X}, S)),$$

where S is an h-sorted variable, where m_k is a method symbol of P_k, and where a_i is an attribute symbol of P_i. In this case, we will write $\varphi_1 \perp \varphi_2$. □

Example 32: If X and Y are the integer cells in Example 29, but without the `init` methods, and if XY is specification `XY`, again without the `init` method, let ι_X and ι_Y be the insertion morphisms defined by $\iota_X(\texttt{X}) = \iota_Y(\texttt{Y}) = \texttt{S}$, $\iota_X(\texttt{get}) = \texttt{getx}$, $\iota_X(\texttt{put}) = \texttt{putx}$, $\iota_Y(\texttt{get}) = \texttt{gety}$, and $\iota_Y(\texttt{put}) = \texttt{puty}$. Then ι_X and ι_Y are independent. An equation of the first kind required by Definition 31 is

```
eq getx puty(N,S) = getx S .
```

while an equation of the second kind is

```
eq putx(M,puty(N,S)) = puty(N,putx(M,S)).
```

which is a behavioural consequence of the specification XY. □

The intuitive idea behind the following universal characterisation of the parallel connection of systems P_i is simply that it is initial among all specifications that have independent insertions of the P_i.

Definition 33: Given a collection $\mathcal{P} = \{P_i \mid i \in I\}$ of hidden sorted specifications (each having just one hidden sort), let $\mathbb{I}nd(\mathcal{P})$ be the full subcategory of co-cones φ over $\mathcal{P}$ such that $\varphi_i \perp \varphi_j$ for all $i, j \in I$ with $i \neq j$; i.e., $\mathbb{I}nd(\mathcal{P})$ is the category whose objects are families $\varphi_i\colon P_i \to P$ for $i \in I$ of specification morphisms such that $\varphi_i \perp \varphi_j$ for all $i, j \in I$ with $i \neq j$, and whose morphisms $\rho\colon \varphi \to \varphi'$ are the specification morphisms $\rho\colon P \to P'$ such that $\varphi_i; \rho = \varphi'_i$ for all $i \in I$.

Let us call an initial object in $\mathbb{I}nd(\mathcal{P})$ an **independent sum** or **parallel connection** of the P_i for $i \in I$, and let us denote it by $\|_{i \in I}\, P_i$ or by $\|\, \mathcal{P}$. □

Note that although the components P_i are required to have just one hidden sort, the candidates P for the independent sum are not required to satisfy this property. However, the proof of Theorem 35 below shows that in fact the independent sum of components having just one hidden sort also has just one hidden sort. But first, we should state the following, the proof of which is similar to proofs in [18]:

Theorem 34: The category of hidden sorted signatures has colimits. □

Now the main result of this subsection:

Theorem 35: Any collection $\mathcal{P} = \{P_i \mid i \in I\}$ of hidden sorted specifications having just one hidden sort has an independent sum.

Proof: For any specification P, let $\Sigma(P)$ denote its signature. Now define $(\Sigma(P_i) \xrightarrow{\iota_i} \Sigma)_{i\in I}$ to be the colimit of the cone $(\Upsilon \hookrightarrow \Sigma(P_i))_{i\in I}$ in the category of hidden sorted signatures, where Υ is the hidden sorted signature with hidden part consisting of just the sort h, and with no method or attribute symbols. Then we define the specification $\|\mathcal{P}$ to have Σ as its signature and to have

$$\bigcup_{i\in I} \iota_i(P_i) \;\cup\; \bigcup_{i\neq j\in I} (A_{ij} \cup M_{ij})$$

as its sentences, where

$$A_{ij} = \{(\forall \underline{X})(\forall \underline{Y})(\forall S)\ \iota_i(a_i)(\underline{X}, \iota_j(m_j)(\underline{Y}, S)) = \iota_i(a_i)(\underline{X}, S) \mid a_i \in A_i \text{ and } m_j \in M_j\}$$

and

$$M_{ij} = \{(\forall \underline{X})(\forall \underline{Y})(\forall S)\ \iota_i(m_i)(\underline{X}, \iota_j(m_j)(\underline{Y}, S)) = \iota_j(m_j)(\underline{Y}, \iota_i(m_i)(\underline{X}, S)) \mid m_k \in M_k\},$$

where A_k is the set of attributes of P_k and M_k is the set of methods of P_k for each $k \in I$. Now it is not difficult to see that $\iota_i \perp \iota_j$ for all $i \neq j \in I$.

Next, consider a co-cone $(P_i \xrightarrow{\varphi_i} P)_{i\in I}$ such that $\varphi_i \perp \varphi_j$ for all $i \neq j \in I$. Then we have to show that there is a unique specification morphism $\rho\colon \|\mathcal{P} \to P$ such that $\iota_i; \rho = \varphi_i$ for all $i \in I$.

$$\begin{tikzcd}
P_i \arrow[rr, "\iota_i"] \arrow[dr, "\varphi_i"'] & & \|\mathcal{P} \arrow[dl, "\rho"] \\
& P &
\end{tikzcd}$$

Because $\varphi_i(h) = \varphi_j(h)$ for all $i, j \in I$, we conclude that $(\Sigma(P_i) \xrightarrow{\varphi_i} \Sigma(P))_{i\in I}$ is a co-cone over $(\Upsilon \hookrightarrow \Sigma(P_i))_{i\in I}$. Therefore there is a unique hidden sorted signature morphism $\rho\colon \Sigma \to \Sigma(P)$ such that $\iota_i; \rho = \varphi_i$ for all $i \in I$.

Now it only remains to show that ρ is a specification morphism $\|\mathcal{P} \to P$. Let q be a sentence in $\|\mathcal{P}$. If $q \in \iota_i(P_i)$ for some $i \in I$, then $\rho(q) \in (\iota_i; \rho)P_i = \varphi_i(P_i)$. Therefore $P \models \rho(q)$. Also, if $q \in A_{ij}$ for some $i, j \in I$, then q is of the form

$$(\forall \underline{X})(\forall \underline{Y})(\forall S)\ (\iota_i(a))(\underline{X}, (\iota_j(m))(\underline{Y}, S)) = (\iota_i(a))(\underline{X}, S)$$

for some attribute symbol a of $\Sigma(P_i)$ and some method symbol m of $\Sigma(P_j)$. Therefore $\rho(q)$ is of the form

$$(\forall \underline{X})(\forall \underline{Y})(\forall S)\ (\varphi_i(a))(\underline{X}, (\varphi_j(m))(\underline{Y}, S)) = (\varphi_i(a))(\underline{X}, S).$$

Since $\varphi_i \perp \varphi_j$, we have $P \models\!\!\equiv \rho(q)$.

The case where $q \in M_{ij}$ for some $i, j \in I$ follows in a similar way. □

For example, if ι_X and ι_Y are the insertion morphisms defined in Example 32, then the diagram $X \xrightarrow{\iota_X} XY \xleftarrow{\iota_Y} Y$ satisfies the universal property for being an independent sum. Similarly, the specification `MGR` of Example 30 is (in some appropriate sense) the independent sum of a (potentially infinite) collection of `CELL` objects, indexed over their unique identifiers.

We expect that the construction and result of this section will generalise to the case where there may be more than one hidden sort in the component specifications.

7.2 Universal Property for Parallel Connection with Synchronisation

This subsection extends the previous subsection to situations where some methods and/or attributes are shared among the component specifications; this corresponds to what is called *synchronisation* in concurrency theory; it provides a very convenient way for communication among objects that would otherwise be independent. We first define the kind of situation in which synchronised parallel connection occurs.

Definition 36: Given a collection $\mathcal{P} = \{P_i \mid i \in I\}$ of hidden sorted specifications, then a **synchronisation** τ of $\mathcal{P}$ is a cone $(Q \xrightarrow{\tau_i} P_i)_{i \in I}$. The hidden sorted specification Q is called the **shared part** of the synchronisation.

Now let $P_1 \xleftarrow{\tau_1} Q \xrightarrow{\tau_2} P_2$ be a synchronisation τ of P_1 and P_2. Then hidden sorted specification morphisms of the form $P_1 \xrightarrow{\varphi_1} P \xleftarrow{\varphi_2} P_2$ are **independent over** τ iff $\tau_1; \varphi_1 = \tau_2; \varphi_2$, and for all $i, j \in \{1, 2\}$ with $i \neq j$, we have

$$P \models\!\!\equiv (\forall \underline{X})(\forall \underline{Y})(\forall S)\ \varphi_i(a_i)(\underline{X}, \varphi_j(m_j)(\underline{Y}, S)) = \varphi_i(a_i)(\underline{X}, S),$$

$$P \models\!\!\equiv (\forall \underline{X})(\forall \underline{Y})(\forall S)\ \varphi_i(m_i)(\underline{X}, \varphi_j(m_j)(\underline{Y}, S)) = \varphi_j(m_j)(\underline{Y}, \varphi_i(m_i)(\underline{X}, S)),$$

where S is an h-sorted variable, where m_k is a method symbol of P_k, where a_i is an attribute symbol of P_i, and where none of these symbols lie in the image of τ. In this case, we will write $\varphi_1 \perp^\tau \varphi_2$. □

The result below generalises Theorem 35 to synchronised parallel connections of hidden sorted specifications. Its proof follows the same lines as that of Theorem 35, and is therefore omitted.

Theorem 37: Given a collection $\mathcal{P} = \{P_i \mid i \in I\}$ of hidden sorted specifications (each having just one hidden sort) and a synchronisation τ of $\mathcal{P}$, let $\mathbb{I}nd^\tau(\mathcal{P})$ be the full subcategory of co-cones φ over $\mathcal{P}$ such that $\varphi_i \perp^\tau \varphi_j$ for all $i, j \in I$ with $i \neq j$.

Then $\mathbb{I}nd^\tau(\mathcal{P})$ has an initial object, called the **independent sum over** τ, or the **synchronised parallel connection over** τ, denoted $\|^\tau_{i \in I} P_i$ or $\|^\tau \mathcal{P}$. □

For example, the specification `XY` of Example 29 is the parallel connection of the specifications `X` and `Y` in that example, with synchronisation over the `init` method.

8 Further Research

It seems reasonable to expect that further aspects of concurrency could be treated in our hidden sorted framework in a natural way. However, we should also expect to find some difficult problems; in particular, it seems likely that combining object-valued logical variables with concurrency will raise some challenging theoretical problems concerning what may be an interesting new style for concurrent programming.

Non-determinism is another area that could be challenging. Notice that our way of describing objects with equational specifications is already non-deterministic, in the sense that there is no fixed starting state, and no fixed input-output relations; instead, we just describe the set of possible states, and how methods affect them. Then interconnecting objects with synchronisation may impose constraints that cut down on the possible states of the component objects. Combining this "internal non-determinism" with our constraint oriented approach to interconnection gives a situation similar to so-called "external non-determinism."

Another interesting problem is to characterise the interleaving parallel connection, which we might denote $X \otimes Y$, as opposed to the true concurrent connection that we have denoted $X \parallel Y$ and for which we have given a universal characterision above.

It is interesting to note that forming the so-called Lawvere algebraic theory (in the sense of [28] and [29]) of a hidden sorted specification gives rise to a structure generalising some traditional structures of process algebra, in that there is an action of the monoid of all method expressions (in the sense of FOOPS [21]) on the collection of all attribute expressions (again in the sense of FOOPS). This construction is similar to one used in [5], except that there only ground method and attribute expressions were used. Complex method and attribute expressions bring us closer to concurrent systems programming than the usual process algebra approaches, which only consider ground methods and attributes, ignoring the structure of data, and ignoring the relationships between methods and attributes that allow forming complex method expressions. On the other hand, traditional process algebra seems well suited to exploring concurrency as such, and it has been applied to some important issues that are not directly addressed in this paper, such as non-determinism and non-termination.

Finally, we expect soon to do some simple proofs of concurrent object systems using the 2OBJ metalogical framework theorem prover [24, 36] to support hidden sorted equational reasoning. This should provide a good test of how viable the approach suggested in this paper can be in practice.

Acknowledgements

We wish to thank Dr. Grant Malcolm for several important suggestions, and Ms. Frances Page for cheerfully typing many drafts, and for preparing the figure.

References

[1] Antonio Alencar. *OOZE: An Object Oriented Specification Language*. PhD thesis, Programming Research Group, Oxford University Computing Lab, to appear 1994.

[2] Antonio Alencar and Joseph Goguen. Specification in OOZE with examples. In *Object Oriented Specification Case Studies*, pages 158–183. Prentice Hall, 1993. Programming Research Group, Oxford University Computing Lab, Report PRG–TR–7–92, 1992.

[3] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice-Hall, 1990.

[4] Garrett Birkhoff. On the structure of abstract algebras. *Proceedings of the Cambridge Philosophical Society*, 31:433–454, 1935.

[5] Rod Burstall and Răzvan Diaconescu. Hiding and behaviour: an institutional approach. Technical Report ECS-LFCS-8892-253, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992. To appear in *A Classical Mind: Essays in Honour of C.A.R. Hoare*, Prentice Hall, 1993.

[6] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings, 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer, 1980. Lecture Notes in Computer Science, Volume 86; based on unpublished notes handed out at the Symposium on Algebra and Applications, Stefan Banach Center, Warsaw, Poland, 1978.

[7] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewriting systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 244–320. Elsevier Science, 1990.

[8] Răzvan Diaconescu. The logic of Horn clauses is equational. Technical Report PRG-TR-3-93, Programming Research Group, University of Oxford, 1993. Written in 1990.

[9] 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.

[10] Joseph Goguen. Principles of parameterized programming. In Ted Biggerstaff and Alan Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 159–225. Addison Wesley, 1989.

[11] 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.

[12] Joseph Goguen. Hyperprogramming: A formal approach to software environments. In *Proceedings, Symposium on Formal Approaches to Software Environment Technology.* Joint System Development Corporation, Tokyo, Japan, January 1990.

[13] Joseph Goguen. Proving and rewriting. In Hélène Kirchner and Wolfgang Wechler, editors, *Proceedings, Second International Conference on Algebraic and Logic Programming*, pages 1–24. Springer, 1990. Lecture Notes in Computer Science, Volume 463.

[14] Joseph Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, March 1991. Also, Programming Research Group Technical Monograph PRG–72, Oxford University, March 1989.

[15] Joseph Goguen. Types as theories. In George Michael Reed, Andrew William Roscoe, and Ralph F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991. Proceedings of a Conference held at Oxford, June 1989.

[16] Joseph Goguen. *Theorem Proving and Algebra.* MIT, 1994.

[17] Joseph Goguen and Rod Burstall. Introducing institutions. In Edward Clarke and Dexter Kozen, editors, *Proceedings, Logics of Programming Workshop*, pages 221–256. Springer, 1984. Lecture Notes in Computer Science, Volume 164.

[18] 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.

[19] Joseph Goguen and Tom Kemp. A hidden Herbrand theorem, 1993. Submitted to special issue of *Theoretical Computer Science*, edited by A. William Roscoe and Michael W. Mislove.

[20] Joseph Goguen and José Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In M. Nielsen and E.M. Schmidt, editors, *Proceedings, 9th International Conference on Automata, Languages and Programming*, pages 265–281. Springer, 1982. Lecture Notes in Computer Science, Volume 140.

[21] 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. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153–162, October 1986.

[22] 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. Also, Programming Research Group Technical Monograph PRG–80, Oxford University, December 1989, and Technical Report SRI-CSL-89-10, SRI International, Computer Science Lab, July 1989; originally given as lecture at *Seminar on Types*, Carnegie-Mellon University, June 1983; many draft versions exist, from as early as 1985.

[23] Joseph Goguen and Adolfo Socorro. Module composition and system design for the object paradigm. *Journal of Object Oriented Programming*, to appear 1994. Technical Report, Oxford University Computer Lab, 1993.

[24] 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.

[25] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, to appear 1993. Also to appear as Technical Report from SRI International.

[26] Joseph Goguen and David Wolfram. On types and FOOPS. In Robert Meersman, William Kent, and Samit Khosla, editors, *Object Oriented Databases: Analysis, Design and Construction*, pages 1–22. North Holland, 1991. Proceedings, IFIP TC2 Conference, Windermere, UK, 2–6 July 1990.

[27] Robert Goldblatt. *Topoi, the Categorial Analysis of Logic*. North-Holland, 1979.

[28] F. William Lawvere. Functorial semantics of algebraic theories. *Proceedings, National Academy of Sciences, U.S.A.*, 50:869–872, 1963. Summary of Ph.D. Thesis, Columbia University.

[29] Ernest Manes. *Algebraic Theories*. Springer, 1976. Graduate Texts in Mathematics, Volume 26.

[30] José Meseguer and Joseph Goguen. Initiality, induction and computability. In Maurice Nivat and John Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge, 1985.

[31] David Parnas. Information distribution aspects of design methodology. *Information Processing '72*, 71:339–344, 1972. Proceedings of 1972 IFIP Congress.

[32] Horst Reichel. Behavioural equivalence – a unifying concept for initial and final specifications. In *Proceedings, Third Hungarian Computer Science Conference*. Akademiai Kiado, 1981. Budapest.

[33] Horst Reichel. Behavioural validity of conditional equations in abstract data types. In *Contributions to General Algebra 3*. Teubner, 1985. Proceedings of the Vienna Conference, June 21-24, 1984.

[34] Donald Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Information and Control*, 76:165–210, 1988. Earlier version in *Proceedings, International Symposium on the Semantics of Data Types*, Lecture Notes in Computer Science, Volume 173, Springer, 1985.

[35] Adolfo Socorro. *Design, Implementation, and Evaluation of a Declarative Object Oriented Language*. PhD thesis, Programming Research Group, Oxford University, 1993.

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

[37] Alfred Tarski. The semantic conception of truth. *Philos. Phenomenological Research*, 4:13–47, 1944.

A Transcript of an OBJ3 Proof Score

The following is the actual output produced by OBJ3 when given a proof score for showing that the equation

```
eq rev rev F = F .
```

is behaviourally satisfied by `FLAG`, according to the method discussed in Example 26 above. (Unfortunately, it does require some familiarity with OBJ3 and its use for theorem proving; the necessary details can be found in [13], [16], and [25]. The module `PROPC` that is used here provides a decision procedure for the Booleans.)

```
ruby: obj
                      \||||||||||||||||||/
                    --- Welcome to OBJ3 ---
                      /||||||||||||||||||\
     OBJ3 version 2.02 built: 1992 Jul 11 Sat 18:44:17
         Copyright 1988,1989,1991 SRI International
                    1993 Apr 23 Fri 10:24:39
OBJ> in tasop-pf
==========================================
***> file: ~goguen/oo/papers/tasop-pf.obj
==========================================
in /users/goguen/obj/prop/propc
Reading in file : "/users/goguen/obj/prop/propc"
==========================================
***> this file is /users/goguen/prop/propc.obj
==========================================
***> decision procedure for the propositional calculus
==========================================
obj PROPC
Done reading in file: "/users/goguen/obj/prop/propc"
==========================================
obj DATA
==========================================
obj FLAG
==========================================
***> to show: for all c, F: up? c rev rev F = up? c F
==========================================
openr
==========================================
***> common notation
==========================================
vars F z : Flag .
==========================================
op f : -> Flag .
==========================================
ops ( LHS _ ) ( RHS _ ) : Flag -> Bool .
==========================================
```

```
ops ( c _ ) ( c' _ ) : Flag -> Flag .
==========================================
eq LHS F = up? c rev rev F .
==========================================
eq RHS F = up? c F .
==========================================
close
==========================================
open
==========================================
***> base case:
==========================================
var z : Flag .
==========================================
eq c z = z .
==========================================
reduce in FLAG : LHS f == RHS f
rewrites: 10
result Bool: true
==========================================
close
==========================================
***> induction steps:
==========================================
openr
==========================================
***> induction hypothesis:
==========================================
var F : Flag .
==========================================
eq up? c' rev rev F = up? c' F .
==========================================
close
==========================================
***> first up:
==========================================
open
==========================================
var z : Flag .
==========================================
eq c z = up c' z .
==========================================
reduce in FLAG : LHS f == RHS f
rewrites: 7
result Bool: true
==========================================
close
==========================================
```

```
***> second dn:
==========================================
open
==========================================
var z : Flag .
==========================================
eq c z = dn c' z .
==========================================
reduce in FLAG : LHS f == RHS f
rewrites: 7
result Bool: true
==========================================
close
==========================================
***> third rev:
==========================================
open
==========================================
var z : Flag .
==========================================
eq c z = rev c' z .
==========================================
reduce in FLAG : LHS f == RHS f
rewrites: 10
result Bool: true
==========================================
close
OBJ> q
Bye.
```

This only took about 4.6 seconds to run, and much of that was consumed by I/O. By including this detailed example, the authors hope to reinforce their basic contention that the hidden sorted algebra semantics for the object paradigm supports a simple calculational style of reasoning that is both more convenient and more efficient than currently known alternatives. Furthermore, it seems clear that it should be straightforward to totally automate this kind of proof using the 2OBJ theorem prover [36, 24].

Contents

1 Introduction **1**

2 Prerequisites and Preliminaries **2**
2.1 Overloaded Many Sorted Algebra 2
2.2 Congruence and Quotient 4
2.3 Equational Deduction 4
2.4 The Satisfaction Condition 5
2.5 Other Topics 5

3 Hidden Sorted Algebra **5**
3.1 The Order Sorted Case 9
3.2 Inheritance and Modularity 10

4 The Hidden Equational Institution **12**
4.1 Institutions 14

5 Techniques for Verification **16**

6 Existential Queries **17**

7 Concurrent Systems of Objects **18**
7.1 Universal Property for Parallel Connection 21
7.2 Universal Property for Parallel Connection with Synchronisation 23

8 Further Research **24**

A Transcript of an OBJ3 Proof Score **28**