Abstract: This paper outlines a research programme in algebraic engineering. It starts with a review of classical algebraic specification for abstract data types, such as integers, vectors, booleans, and lists. Software engineering also needs abstract machines, recently called “objects,” that can communicate concurrently with other objects, and that have local states with visible “attributes” that are changed by inputs. Hidden algebra is a new development in algebraic semantics for such systems; its most important results are powerful hidden coinduction principles for proving behavioral properties, especially behavioral refinement.

1 Introduction

In view of the title of this conference, I should confess to being an algebraic engineer in (perhaps) the following four different senses:

1. I use algebra to build real software systems.
2. I build huge algebras to help build software systems.
3. I build software tools to help deal with these huge algebras.
4. I build new kinds of algebra, to get better results in building software.

All this is within the general framework of universal (also called “general”) algebra, as pioneered by Birkhoff [2] and Tarski (among others). Although I’m far from the only one doing such things, this survey focuses on the work with which I am most familiar, from my research groups at Oxford and at UCSD, and with the CafeOBJ project. The UCSD group is putting much related material on the web, including sample hidden algebraic proofs, with tutorial background information, remote proof execution, and Java applets to illustrate specifications, properties, and proof ideas; see www.cs.ucsd.edu/groups/tatami, and for further background, the papers [24, 23, 30] which are available from www.cs.ucsd.edu/users/goguen (along with many others). Some other related work is discussed briefly in Section 4.

1.1 Notes on the State of the Art

Software development is very difficult. To understand it better, we can distinguish among designing, coding and verifying (i.e., proving properties of) a program. Most of the literature addresses code verification, but this can be exceedingly difficult in practice, and moreover, empirical studies have shown that little of the cost of software arises from errors in coding: most comes from errors in design and requirements [3]. Moreover, many of the most important programs are written in obscure and/or obsolete languages, with complex ugly semantics (such as Cobol, Jovial and Mumps), are...
very poorly documented, are indispensable to some enterprise, and are very large, often several million lines, sometimes more. Therefore it is usually an enormous effort to verify real code, and it isn't usually worth the trouble. I like to call this the semantic swamp; it is a place to avoid. Moreover, programs in everyday use usually evolve, because computers, operating systems, tax laws, user requirements, etc. are all changing rapidly. Therefore the effort of verifying yesterday's version is wasted, because even small code modifications can require large proof modifications; proof is a discontinuous function of truth.

The above suggests we should focus on design and specification. But even this is difficult, because the properties that people really want, such as security, deadlock freedom, liveness, ease of use, and ease of maintenance, are complex, not always formalizable, and even when they are formalizable, may involve subtle interactions among remote parts of the system. However, this is an area where mathematics can make a contribution.

It is well known that most of the effort in programming goes into debugging and maintenance (i.e., into improving and updating programs) [3]. Therefore anything that can be done to ease these processes has enormous economic leverage. One step in this direction is to "encapsulate data representations"; this means to make the actual structure of data invisible, and to provide access to it only via a given set of operations which retrieve and modify the hidden data structure. Then the implementing code can be improved without having to change any of the code that uses it. On the other hand, if client code relies on properties of the representation, it can be extremely hard to track down all the consequences of modifying a given data structure (say, changing a doubly linked list to an array), because the client code may be scattered all over the program, without any clear identifying marks. This is why the so-called year 2000 problem is so difficult.

An encapsulated data structure with its accompanying operations is called an abstract data type. The crucial advance was to recognize that operations should be associated with data representations; this is exactly the same insight that advanced algebra from mere sets to algebras, which are sets with their associated operations. In software engineering this insight seems to have been due to David Parnas [47], and in algebra to Emmy Noether [51]. Parallel developments in software engineering and abstract algebra are a theme of this paper.

It turns out that although abstraction as isomorphism is enough for algebras representing data values (numbers, vectors, etc.), other important problems in software engineering need the more general notion of behavioral abstraction, where two models are considered abstractly the same if they exhibit the same behavior. The usual many sorted algebra is not rich enough for this: we have to add structure to distinguish sorts used for data values from sorts used for states, and we need a more general, behavioral, notion of satisfaction; these are developed in Section 3.

In line with our general discussion of software methodology above, we don't want to prove properties of code, but rather properties of specifications. Often the most important property of a specification is that it refines another specification, in the sense that any model (i.e., any code realizing) the second is also a model of the first. Methodologically, a refinement embodies a set of closely related design decisions for realizing one set of behaviors from another. In line with the discussion of the previous paragraph, we want to prove behavioral properties and refinements. Behavioral refinement is much more general than ordinary refinement, and many of the enormous variety of clever implementation techniques that so often occur in practice require this extra generality.

---

[2] Empirical studies show that real software development projects involve many false starts, redesigns, prototypes, patches, etc. [7]. Nevertheless, an idealized view of a project as a sequence of refinements is still useful as a way to organize and document verification efforts, often retrospectively.
1.2 Overview of this Paper

Section 2 gives a brief overview of classical algebraic specification theory for abstract data types, which may be thought of as realms of unchanging Platonic values, such as integers, booleans, and lists; this exposition follows [19] and [21]. Following [27], Section 3 explains why software engineering also needs abstract machines, recently called “objects,” and why their behavioral properties are important. It then gives a brief overview of the hidden algebra approach to behavioral specification, including its most significant proof technique, which is called coinduction. We use the notation of OBJ3 [26, 34] for some simple examples.

Dedication  After the Algebraic Engineering ’97 conference, I learned that it would be dedicated to Prof. John Rhodes, in honor of his sixtieth birthday. When I was a graduate student at Berkeley, I had the pleasure of taking classes from John, and of working for his company, the Krohn-Rhodes Research Institute. In Aizu, I had the pleasure of giving the opening lecture of the conference, as the warm up act to John’s first talk, and now I have the pleasure of dedicating this paper to him, hoping that many new generations of students will benefit from his inimitable, effervescent and effective teaching style.

2 Algebraic Specification of Abstract Data Types

This section first motivates abstract data types from the viewpoint of software engineering, gives a precise definition for this concept, and states some of its basic properties, especially that an abstract data type is uniquely determined by its specification as an initial algebra, and that abstract data types are indeed abstract. Problems in computing science, such as proving the correctness of a compiler, usually involve more elaborate data structures than integers and Booleans, such as queues, stacks, arrays, or lists of stacks of integers. We usually want proofs about software to be independent of how these underlying data types happen to be represented; for example, we are usually not interested in properties of the decimal or binary representations of the natural numbers, but instead are interested in abstract properties of the abstract natural numbers.

2.1 Signature, Algebra and Homomorphism

Our syntax for many sorted algebra permits overloaded operation symbols\(^3\). This has some interesting applications. Following suggestions I made 30 years ago, it is now usual in computer science to base many sorted algebra on many sorted sets: Given a set \( S \), whose elements are called \textit{sorts}, an \textit{\( S \)-sorted set} \( A \) is a family of sets \( A_s \), one for each \( s \in S \).

\(\textbf{Definition 2.1} \) An \textit{\( S \)-sorted signature} \( \Sigma \) is an \(( S^* \times S)\)-sorted set \( \{\Sigma_{w,s} \mid \langle w, s \rangle \in S^* \times S\} \). The elements of \( \Sigma_{w,s} \) are called \textit{operation symbols} of \textit{arity} \( w \), \textit{sort} \( s \), and \textit{rank} \( \langle w, s \rangle \); in particular, \( \sigma \in \Sigma_{w,s} \) is a \textit{constant symbol}. \( \Sigma \) is a \textit{ground signature} iff \( \Sigma_{w,s} \cap \Sigma_{w,s'} = \emptyset \) whenever \( s \neq s' \) and \( \Sigma_{w,s} = \emptyset \) unless \( w = [s] \).

By convention, \( |\Sigma| = \bigcup_w \Sigma_{w,s} \) and \( \Sigma' \subseteq \Sigma \) means \( \Sigma'_{w,s} \subseteq \Sigma_{w,s} \) for each \( w, s \). Similarly, \textit{union} is defined by \( (\Sigma \cup \Sigma')_{w,s} = \Sigma_{w,s} \cup \Sigma'_{w,s} \). A common special case is union with a ground signature \( X \), for which we use the notation \( \Sigma(X) = \Sigma \cup X \).

\(\textbf{Note} \) This allows a symbol to have more than one distinct rank.
Definition 2.2 A \( \Sigma \)-algebra \( A \) consists of an \( S \)-sorted set also denoted \( A \), plus an interpretation of \( \Sigma \) in \( A \), which is a family of arrows \( i_{s_1 \ldots s_n, s} : \Sigma_{s_1 \ldots s_n, s} \to [A^{s_1 \ldots s_n} \to A] \) for each rank \( \langle s_1 \ldots s_n, s \rangle \in S^k \times S \), which interpret the operation symbols in \( \Sigma \) as actual operations on \( A \). For constant symbols, the interpretation is given by \( i_{\emptyset, s} : \Sigma_{\emptyset, s} \to A_s \). Usually we write just \( \sigma \) for \( i_{\emptyset, s}(\sigma) \), but if we need to make the dependence on \( A \) explicit, we may write \( \sigma_A \). \( A_s \) is called the carrier of \( A \) of sort \( s \).

Given \( \Sigma \)-algebras \( A, A' \), a \( \Sigma \)-homomorphism \( h : A \to A' \) is an \( S \)-sorted arrow \( h : A \to A' \) such that \( h_i(\sigma_A(a_1, \ldots, a_n)) = \sigma_{A'}(h_i(s_1), \ldots, h_i(a_n)) \) for each \( \sigma \in \Sigma_{s_1 \ldots s_n, s} \) and \( a_i \in A_{s_i} \) for \( i = 1, \ldots, n \), and such that \( h_i(c_A) = c_{A'} \) for each constant symbol \( c \in \Sigma_{\emptyset, s} \). □

2.2 Term, Equation and Specification

Given an \( S \)-sorted signature \( \Sigma \), the \( S \)-sorted set \( T_\Sigma \) of (ground) \( \Sigma \)-terms is the smallest set of lists of symbols that contains the constants, \( \Sigma_{\emptyset, s} \subseteq T_\Sigma, s \), and such that given \( \sigma \in \Sigma_{s_1 \ldots s_n, s} \) and \( t_i \in T_{\Sigma, s_i} \), then \( \sigma(t_1 \ldots t_n) \in T_{\Sigma, s} \). We view \( T_\Sigma \) as a \( \Sigma \)-algebra by interpreting \( \sigma \in \Sigma_{s, s} \) as just \( \sigma \), and \( \sigma \in \Sigma_{s_1 \ldots s_n, s} \) as the operation sending \( t_1, \ldots, t_n \) to the list \( \sigma(t_1 \ldots t_n) \). Then \( T_\Sigma \) is called the \( \Sigma \)-term algebra. Note that because of overloading, terms do not always have a unique parse. The following is the key property of this algebra:

Theorem 2.3 (Initiality) Given a signature \( \Sigma \) with no overloaded constants\(^4\) and a \( \Sigma \)-algebra \( M \), there is a unique \( \Sigma \)-homomorphism \( T_\Sigma \to M \). □

The \( \Sigma \)-term algebra \( T_\Sigma \) serves as a standard model for a specification \( P = (\Sigma, \emptyset) \) with no equations. For example, if \( \Sigma \) is the signature for the natural numbers with just zero and successor (0 and \( \mathbb{S} \)), then \( T_\Sigma \) is the natural numbers in Peano notation, and if \( \Sigma' \) adds the operation symbols \( + \) and \( \times \), then \( T_{\Sigma'} \) consists of all expressions formed using these symbols (with the right arities); these are simple numerical expressions.

But there are also many examples that need equations, such as the natural numbers with addition. For this we need the following:

Definition 2.4 A \( \Sigma \)-equation consists of a ground signature \( X \) of variable symbols (disjoint from \( \Sigma \)) plus two \( \Sigma(X) \)-terms of the same sort \( s \in S \); we may write such an equation abstractly in the form \( \forall X \ t = t' \) and concretely in the form \( \forall x, y, z \ t = t' \) when \( X = \{ x, y, z \} \) and the sorts of \( x, y, z \) can be inferred from their uses in \( t \) and in \( t' \). A specification is a pair \( (\Sigma, E) \), consisting of a signature \( \Sigma \) and a set \( E \) of \( \Sigma \)-equations. □

Conditional equations can be defined in a similar way, but we omit this here. Given \( \Sigma \) and a ground signature \( X \) disjoint from \( \Sigma \), we can form the \( \Sigma(X) \)-algebra \( T_{\Sigma(X)} \) and then view it as a \( \Sigma \)-algebra by forgetting the names of the new constants in \( X \); let us denote this \( \Sigma \)-algebra by \( T_\Sigma(X) \). It has the following universal freeness property:

Proposition 2.5 Given a \( \Sigma \)-algebra \( A \) and an interpretation \( a : X \to A \), there is a unique \( \Sigma \)-homomorphism \( \overline{a} : T_\Sigma(X) \to A \) extending \( a \), in the sense that \( \overline{a}_s(x) = a_s(x) \) for each \( x \in X_s \) and \( s \in S \). □

Definition 2.6 A \( \Sigma \)-algebra \( A \) satisfies a \( \Sigma \)-equation \( \forall X \ t = t' \) iff for every \( a : X \to A \) we have \( \overline{a}_s(t) = \overline{a}_s(t') \) in \( A_s \), written \( A \models_\Sigma \forall X \ t = t' \). A \( \Sigma \)-algebra \( A \) satisfies a set \( E \) of \( \Sigma \)-equations iff it satisfies each one, written \( A \models_\Sigma E \). We may also say \( A \) is a \( P \)-algebra, and write \( A \models P \) where \( P = (\Sigma, E) \). The class of all algebras that satisfy \( P \) is called the variety defined by \( P \). Given sets \( E \) and \( E' \) of \( \Sigma \)-equations, let \( E \models E' \) mean \( A \models E \) for all \( E \)-models \( A \). □

\(^4\)Actually, every signature \( \Sigma \) has an initial term algebra, but when \( \Sigma \) has overloaded constants, terms must be annotated by their sort; we will use the same notation \( T_\Sigma \) for this case.
The following simple result is much used in equational theorem proving, but is rarely stated explicitly. Its proof is very simple because it uses the semantics of satisfaction rather than some particular rules of deduction, and because it exploits the initiality of the term algebra. We have found this typical of proofs in this area; commutative diagrams and other universal properties also help give elegant conceptual proofs, though we have omitted all other proofs from this paper.

**Fact 2.7 (Lemma of Constants)** Given a signature $\Sigma$, a ground signature $X$ disjoint from $\Sigma$, a set $E$ of $\Sigma$-equations, and $t, t' \in T_\Sigma(X)$, then $E \models (\forall X) t = t'$ iff $E \models (\forall \emptyset) t = t'$.

**Proof:** Each condition is equivalent to the condition that $\pi(t) = \pi(t')$ for every $\Sigma(X)$-algebra $A$ satisfying $E$ and every $a : X \to A$. □

**Theorem 2.8** $T_{\Sigma,E} = T_{\Sigma}/\cong_E$ is an initial $(\Sigma, E)$-algebra, where $\cong_E$ is the $\Sigma$-congruence relation generated by the ground instances of equations in $E$. □

The following result shows that satisfaction of an equation by an algebra is an “abstract” property, in the sense that it is independent of how the algebra happens to be represented. This is fortunate, because these are usually the properties in which we are most interested. This result implies that exactly the same equations are true of any one initial $P$-algebra as any other.

**Theorem 2.9** Given a specification $P = (\Sigma, E)$, any two initial $P$-algebras are $\Sigma$-isomorphic; in fact, if $A$ and $A'$ are two initial $P$-algebras, then the unique $\Sigma$-homomorphisms $A \to A'$ and $A' \to A$ are both isomorphisms, and indeed, are inverse to each other. Moreover, given isomorphic $\Sigma$-algebras $A$ and $A'$, and given a $\Sigma$-equation $e$, then $A \models e$ iff $A' \models e$. □

The word “abstract” in the phrase “abstract algebra” means “uniquely defined up to isomorphism”; for example, an “abstract group” is an isomorphism class of groups, indicating that we are not interested in properties of any particular representation, but only in properties that hold for all representations; e.g., see [39]. Because Theorem 2.9 implies that all the initial models of a specification $P = (\Sigma, E)$ are abstractly the same in precisely this sense, the word “abstract” in “abstract data type” has exactly the same meaning. This is not a mere pun, but a significant fact about software engineering.

Another fact suggesting we are on the right track is that any computable abstract data type has an equational specification; moreover, this specification tends to be reasonably simple and intuitive in practice. The following result from [46] somewhat generalizes the original version due to Bergstra and Tucker [1] ($M$ is **reachable** iff the unique $\Sigma$-homomorphism $T_\Sigma \to M$ is surjective):

**Theorem 2.10 (Adequacy of Initiality)** Given any computable reachable $\Sigma$-algebra $M$ with $\Sigma$ finite, there is a finite specification $P = (\Sigma', E')$ such that $\Sigma \subseteq \Sigma'$, such that $\Sigma'$ has the same sorts as $\Sigma$, and such that $M$ is $\Sigma$-isomorphic to $T_P$ viewed as a $\Sigma$-algebra. □

We do not here define the concept of a “computable algebra”, but it corresponds to what one would intuitively expect: all carrier sets are decidable and all operations are total computable functions; see [46]. What this result tells us is that all of the data types that are of interest in computer science can be defined using initiality, although sometimes it may be necessary to add some auxiliary functions. All of this motivates the following fundamental conceptualization, which goes back to 1975 [32, 31]:

**Definition 2.11** The **abstract data type** (abbreviated **ADT**) defined by a specification $P$ is the class of all initial $P$-algebras. □
The importance of initiality for computing developed gradually. The term “initial algebra semantics” and its first applications (including Knuthian attribute semantics) appear in [17], while its first application to abstract data types is in [32]; a more complete and rigorous exposition is given in [31]. More on initiality can be found in [33] and [46]; the latter especially develops connections with induction and computability. Results on the adequacy of initiality were first given by Bergstra and Tucker [1]. See [18] for more historical information about this early period, and [26, 21] for more recent results, examples and references.

2.3 OBJ Notation

OBJ gives a notation for expressing both initial and loose specifications, and this notation has been implemented in a way that permits proving things about such specifications [34, 26, 21]. OBJ modules that are to be interpreted loosely begin with the keyword theory (or th) and close with the keyword endth. Between these two keywords come declarations for sorts and operations, plus (as discussed later) variables and equations. For example, the following OBJ code specifies the theory of automata:

\begin{verbatim}
th AUTOM is
  sorts Input State Output .
  op s0 : -> State .
  op f : Input State -> State .
  op g : State -> Output .
endth
\end{verbatim}

Any number of sorts can be declared following sorts (or equivalently, sort), and operations are declared with their arity between the : and the ->, and their sort following the ->.

The keyword pair obj...endo indicates that initial semantics is intended. For example, the Peano natural numbers are given by

\begin{verbatim}
obj NATP is
  sort Nat .
  op 0 : -> Nat .
  op s_ : Nat -> Nat .
endo
\end{verbatim}

which uses “mixfix” syntax for the successor operation symbol: in the expression before the colon, the underbar character is a placeholder, showing where the operation’s arguments should go; hence successor has prefix syntax here.

All the OBJ3 code in this paper is executable, and (once a suitable definition for the module DATA is added) executing it actually proves the simple result about flags discussed later; the OBJ output from this paper is given in Appendix A.

3 Hidden Algebra

Initial semantics works very well for data structures like integers, lists, booleans, vectors and matrices, but is more awkward for situations that involve a state, i.e., an internal representation that is changed by commands and never viewed directly, but only through external “attributes.” For example, it is usually more appropriate to view stacks as machines with an encapsulated (invisible) internal state, having “top” as an attribute. Although initial models exist for any reasonable specification of stacks,
real stacks are more likely to be implemented by a model that is not initial, such as a pointer plus an array. This implies that a new notion of implementation is needed, different from the simple notion of initial model. Moreover, in considering (for example) stacks of integers, the sorts for stacks and for integers must be treated differently, since the latter are still modeled initially as data. Although these issues have been successfully addressed in an initial framework (e.g., [31]), it is really better to take a different viewpoint.

Hidden algebra explicitly distinguishes between “visible” sorts for data and “hidden” sorts for states. It makes sense to declare a fixed collection of shared data values, bundled together in a single algebra, because the components of a system must use the same representations for the data that they share, or else they cannot communicate.\(^5\)

**Definition 3.1** Let \( D \) be a fixed data algebra, with \( \Psi \) its signature and \( V \) its sort set, such that each \( D_v \) with \( v \in V \) is non-empty and for each \( d \in D_v \) there is some \( \psi \in \Psi \|_{w^*} \) such that \( \psi \) is interpreted as \( d \) in \( D \); we call \( V \) the visible sorts. For convenience, assume \( D_v \subseteq \Psi \|_{w^*} \) for each \( v \in V \). □

The above concerns semantics; but the prudent verifier needs an effective specification for data values to support proofs, and it is especially convenient to use initial algebra semantics for this purpose, because it supports proofs by induction. We now generalize the notion of signature:

**Definition 3.2** A hidden signature (over a data algebra \((V, \Psi, D)\)) is a pair \((H, \Sigma)\), where \( H \) is a set of hidden sorts disjoint from \( V \), and \( \Sigma \) is an \( S = (H \cup V) \)-sorted signature with \( \Psi \subseteq \Sigma \), such that

1. \( \Sigma \) cannot add any new operations on data items.
2. Every operation in a hidden signature is either a method, an attribute, or else a constant. Equations about data (\( \Psi \)-equations) are not allowed in specifications; any such equation needed as a lemma should be proved and asserted separately, rather than being included in a specification. The following example may help clarify this definition; it is the simplest possible example where something beyond pure equational reasoning and induction is needed.

**Example 3.3** We specify flag objects, where intuitively a flag can be either up or down, with methods to put it up, to put it down, and to reverse it:

\[
\text{th FLAG is sort Flag .}
\]
\[
\text{pr DATA .}
\]
\[
\text{ops (up_.) (dn_.) (rev_.) : Flag -> Flag .}
\]

\(^5\)In practice, there may be multiple representations for data with translations among them, and representations may change during development; but our simplifying assumption can easily be relaxed.
op up? : Flag \rightarrow \text{Bool}.
var F : Flag.
eq \text{up F = true.}
eq \text{up F = false.}
eq \text{up F = not up? F.}
endth

Here \text{FLAG} is the name of the module and \text{Flag} is the name of the class of flag objects. The operations \text{up, dn} and \text{rev} are methods to change the state of flag objects, and \text{up?} is an attribute that tells whether or not the flag is up; all have prefix syntax. □

If \Sigma is the signature of \text{FLAG}, then \Psi is a subsignature of \Sigma, and so a model of \text{FLAG} should be a \Sigma-algebra whose restriction to \Psi is \text{D}, providing functions for all the methods and attributes in \Sigma, and behaving as if it satisfies the given equations. Elements of such models are possible states for Flag objects. This motivates the following:

**Definition 3.4** Given a hidden signature \((H, \Sigma)\), a hidden \Sigma-algebra \text{A} is a (many sorted) \Sigma-algebra \text{A} such that \text{A}\mid_{\Psi} = \text{D}. □

We next define behavioral satisfaction of an equation, an idea introduced by Reichel [48]. Intuitively, the two terms of an equation ‘look the same’ under every ‘experiment’ consisting of some methods followed by an ‘observation,’ i.e., an 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:

**Definition 3.5** Given a hidden signature \((H, \Sigma)\) and a hidden sort \text{h}, then a hidden context of sort \text{h} is a visible sorted \Sigma-term having a single occurrence of a new variable symbol \text{z} of sort \text{h}. A context is appropriate for a term \text{t} iff the sort of \text{t} matches that of \text{z}. Write \text{c[t]} for the result of substituting \text{t} for \text{z} in the context \text{c}.

A hidden \Sigma-algebra \text{A} behaviorally satisfies a \Sigma-equation \((\forall X) \text{t} = \text{t}'\) iff for each appropriate \Sigma-context \text{c}, \text{A} satisfies the equation \((\forall X) \text{c[t]} = \text{c[t]}'\); then we write \text{A} \equiv_{\Sigma} \((\forall X) \text{t} = \text{t}'\).

A model of a hidden theory \text{P} = \((H, \Sigma, E)\) is a hidden \Sigma-algebra \text{A} that behaviorally satisfies each equation in \text{E}. Such a model is also called a \((\Sigma, E)\)-algebra, or a \text{P}-algebra, and then we write \text{A} \equiv \text{P} or \text{A} \equiv_{\Sigma} \text{E}. Also we write \text{E} \equiv_{\Sigma} \text{E}' implies \text{A} \equiv_{\Sigma} \text{E} for each hidden \Sigma-algebra \text{A}. □

**Example 3.6** Let’s look at a simple Boolean cell \text{C} as a hidden algebra. Here, \text{CFlag} = \text{CBool} = \{\text{true, false}\}, \text{up F = true, dn F = false, up? F = F}, and \text{rev F = not F}.

A more complex implementation \text{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 \text{HFlag} = \{\text{up, dn, rev}\}^*, the lists over \{\text{up, dn, rev}\}, while \text{HBool} = \{\text{true, false}\}, \text{up F = up ~ F, dn F = dn ~ F, rev F = rev ~ F, while up? F = true, up? F = false, and up? rev ~ F = not up? F}, where \text{~} is the concatenation operation. Note that \text{C} and \text{H} are not isomorphic. □

For visible equations, there is no difference between ordinary satisfaction and behavioral satisfaction. But these concepts can be very different for hidden equations. For example,

\text{rev rev F = F}

is strictly satisfied by the Boolean cell model \text{C}, but it is not satisfied by the history model \text{H}. However, it is behaviorally satisfied by both models. This illustrates why behavioral satisfaction is often more appropriate for computing science applications.
Previously we gave a semantic definition of an abstract data type as an isomorphism class of initial algebras for some specification; equivalently, by Theorem 2.10, we can define it to be an isomorphism class of computable algebras. The hidden analog of this defines an abstract machine to be a class of all hidden algebras that satisfy some hidden specification; an alternative that is often useful in practice restricts attention to the reachable models. (Although we are really only interested in the semicomputable models, there is no point in complicating the formal definition with this condition.)

3.1 Coinduction

Induction is a standard technique for proving properties of initial (or more generally, reachable) algebras of a theory. Principles of induction can be justified from the fact that an initial algebra has no proper subalgebras satisfying the same signature and equations [21, 46]; final (terminal) algebras play an analogous role in justifying reasoning about behavioral properties with hidden coinduction.

Before describing the final algebra, I want to note that its use is not precisely dual to that of the initial algebra for abstract data types. The semantics of a hidden specification is not the final algebra, but rather is the variety of all hidden algebras that satisfy the spec; in fact, final algebras do not even exist in general. However, their existence for certain signatures, with no equations, plays an important technical role.

Given a hidden signature $\Sigma$ without generalized hidden constants (recall these are hidden operations with no hidden arguments), the hidden carriers of the final $\Sigma$-algebra $F_\Sigma$ are given by the following "magic formula," for $h$ a hidden sort:

$$F_{\Sigma,h} = \prod_{v \in V} [C_{\Sigma[z_h]}v \rightarrow D_v] ,$$

the product of the sets of functions taking contexts to data values (of appropriate sort). Elements of $F_{\Sigma}$ can be thought of as 'abstract states' represented as functions on contexts, returning the data values resulting from evaluating a state in a context. This also appears in the way $F_{\Sigma}$ interprets attributes: let $\sigma \in \Sigma_{hw,v}$ be an attribute, let $p \in F_{\Sigma,h}$ and let $d \in D_w$; then we define $F_{\Sigma,\sigma}(p,d) = p_v(\sigma(z_h,d))$: i.e., $p_v$ is a function taking contexts in $C_{\Sigma[z_h]}v$ to data values in $D_v$, so applying it to the context $\sigma(z_h,d)$ gives the data value resulting from that experiment. Methods are interpreted similarly; see [27] for details.

Definition 3.7 Given a hidden signature $\Sigma$, a hidden subsignature $\Phi \subseteq \Sigma$, and a hidden $\Sigma$-algebra $A$, then behavioral $\Phi$-equivalence on $A$, denoted $\equiv_\Phi$, is defined as follows, for $a, a' \in A_s$:

(E1) $a \equiv_\Phi s a'$ iff $a = a'$

when $s \in V$, and

(E2) $a \equiv_\Phi s a'$ iff $A_c(a) = A_c(a')$ for all $v \in V$ and all $c \in C_{\Phi[z]}v$

when $s \in H$, where $z$ is of sort $s$ and $A_c$ denotes the function interpreting the context $c$ as an operation on $A$, that is, $A_c(a) = \theta_a^*(c)$, where $\theta_a$ is defined by $\theta_a(z) = a$ and $\theta_a^*$ denotes the free extension of $\theta_a$.

When $\Phi = \Sigma$, we may call $\equiv_\Phi$ just behavioral equivalence and denote it $\equiv$.

For $\Phi \subseteq \Sigma$, a hidden $\Phi$-congruence on a hidden $\Sigma$-algebra $A$ is a $\Phi$-congruence $\simeq$ which is the identity on visible sorts, i.e., such that $a \simeq a'$ iff $a = a'$ for all $v \in V$ and $a, a' \in A_v = D_v$. We call a hidden $\Sigma$-congruence just a hidden congruence. □

The key property is the following:

Theorem 3.8 If $\Sigma$ is a hidden signature, $\Phi$ is a hidden subsignature of $\Sigma$, and $A$ is a hidden $\Sigma$-algebra, then behavioral $\Phi$-equivalence is the largest behavioral $\Phi$-congruence on $A$. □
This result is not hard to prove (a simple but very abstract proof is given in [27]). The proof generalizes the well known construction of an abstract machine as a quotient of the term algebra by the behavioral equivalence relation (usually called the Nerode equivalence in that context) [46], and uses the existence of final algebras, which are proved to exist in [27].

Theorem 3.8 implies that if \( a \simeq a' \) under some hidden congruence \( \simeq \), then \( a \) and \( a' \) are behaviorally equivalent. This is the technique that we call coinduction; see [25, 41] for a number of variations. In this context, a relation may be called a \textbf{candidate relation} before it is proved to be a hidden congruence. Probably the most common case is \( \Phi = \Sigma \), but the generalization to smaller \( \Phi \) is useful, for example in verifying refinements.

\textbf{Example 3.9} Let \( A \) be any model of the \textbf{FLAG} theory in Example 3.3, and for \( f, f' \in A_{\text{Flag}} \), define \( f \simeq f' \) iff \( \text{up}\ f = \text{up}\ f' \) (and \( d \simeq d' \) iff \( d = d' \) for data values \( d, d' \)). Then we can use the equations of \textbf{FLAG} to show that \( f \simeq f' \) implies \( \text{up}\ f \simeq \text{up}\ f' \) and \( \text{dn}\ f \simeq \text{dn}\ f' \) and \( \text{rev}\ f \simeq \text{rev}\ f' \), and of course \( \text{up}\ f \simeq \text{up}\ f' \). Hence \( \simeq \) is a hidden congruence on \( A \).

Therefore we can show \( A \models (\forall F:\text{Flag}) \text{rev rev } F = F \) just by showing \( A = (\forall F:\text{Flag}) \text{up}\ F \text{rev rev } F = \text{up}\ F \). This follows by ordinary equational reasoning, since \( \text{up}\ \text{rev rev } F = \text{not}(\text{not}(\text{up}\ F)) \). Therefore the equation is behaviorally satisfied by any \textbf{FLAG}-algebra \( A \).

It is easy to do this proof mechanically using \textsc{OBJ3}, since all the computations are just ordinary equational reasoning. We set up the proof by opening \textbf{FLAG} and adding the necessary assumptions; here \( R \) represents the relation \( \simeq \):

\begin{verbatim}
openr FLAG .
op _R_ : Flag Flag -> Bool .
var F1 F2 : Flag .
eq F1 R F2 = (up F1 == up F2) .
ops f1 f2 : -> Flag .
close
\end{verbatim}

The new constants \( f1, f2 \) are introduced to stand for universally quantified variables, by the Lemma of Constants, and \( == \) is \textsc{OBJ3}'s built-in equality test (what it actually does is reduce its two arguments and check whether the results are identical). The following shows \( R \) is a hidden congruence:

\begin{verbatim}
open .
eq up F1 = up F2 .
red (up f1) R (up f2) .  *** should be: true
red (dn f1) R (dn f2) .  *** should be: true
red (rev f1) R (rev f2) .  *** should be: true
close
\end{verbatim}

where \texttt{red} is a command that tells \textsc{OBJ} to “reduce” the subsequent term, i.e., to apply equations as left-to-right rewrite rules, until a term is obtained where no rule applies.

Finally, we show that all \textbf{FLAG}-algebras behaviorally satisfy the equation with:

\begin{verbatim}
red (rev rev f1) R f1 .
\end{verbatim}

All the above code runs in \textsc{OBJ3}, and gives \texttt{true} for each reduction, provided the following lemma about the Booleans is added somewhere,

\begin{verbatim}
eq not not B = B .
\end{verbatim}

where \( B \) is a Boolean variable. I think this proof is about as simple as could be hoped for\(^6\). \( \square \)

Much more can be said about doing coinductive proofs, just as there is a rich lore about doing inductive proofs; see [27] for more information.

\(^6\)Actually, the third reduction is unnecessary, but it is more trouble to justify its elimination than it is to ask \textsc{OBJ} to do it; see [27].
3.2 Nondeterminism

Since nondeterminism takes some extra effort for abstract data types, it is perhaps surprising that
it is already an inherent facet of hidden algebra, as illustrated in the following very simple example:

\texttt{th C is pr DATA .}
\begin{itemize}
  \item \texttt{op c : \rightarrow Nat .}
\end{itemize}
\texttt{endth}

Here \(c\) has some natural number value in every model, and every number can occur; each model
chooses exactly one. However, there can also be arbitrary junk in models, so it makes sense to restrict
to reachable models; then the choice of a value for \(c\) completely characterizes a model. It is also easy
to restrict the choice of a value for \(c\), by adding equations like the following:

\begin{align*}
  \text{eq } c &= 1 = \text{true} . \\
  \text{eq } 2 &= c = \text{true} . \\
  \text{eq } \text{odd}(c) &= \text{true} . \\
  \text{eq } \text{prime}(c) &= \text{true} . \\
  \text{eq } c &= 1 \text{ or } c &= 2 = \text{true} .
\end{align*}

The last equation suggests a rather cute way to specify nondeterministic choice in hidden algebra:

\texttt{th CH is pr DATA .}
\begin{itemize}
  \item \texttt{op _ | _ : Nat Nat \rightarrow Nat .}
  \item \texttt{vars N M : Nat .}
  \item \texttt{eq N | M \equiv N \text{ or } N | M \equiv M = \text{true} .}
\end{itemize}
\texttt{endth}

Here every model is a “possible world” in which some choice of one of \(N, M\) is made for each pair \(N, M\). It is not hard to prove that this choice function is idempotent, i.e., satisfies the equation

\begin{equation*}
  \text{eq } N | N \equiv N .
\end{equation*}

However, the commutative and associative properties fail for some models (the reader is invited to
find the appropriate models) and hence for the theory.

Neither example of nondeterminism above involves state, which is the most characteristic feature
of hidden algebra, so we really should give an example of nondeterminism with a hidden sort. For
some reason, vending machines are very popular for illustrating various aspects of systems, especially
nondeterminism and concurrency. The spec below describes perhaps the simplest vending machine
that is not entirely trivial: when you put a coin in, it nondeterministically gives you either coffee
or tea, represented say by true and false, respectively; and then it goes into a new state where it is
prepared to do the same again. In this spec, \texttt{init} is the initial state, \texttt{in(init)} is the state after one
coin, \texttt{in(in(init))} is the state after two coins, etc., while \texttt{out(init)} is what you get after the first
coin, \texttt{out(in(init))} after the second, etc.

\texttt{th VCT is sort St .}
\begin{itemize}
  \item \texttt{pr DATA .}
  \item \texttt{op init : \rightarrow St .}
  \item \texttt{op in : St \rightarrow St .}
  \item \texttt{op out : St \rightarrow Bool .}
\end{itemize}
\texttt{endth}
As before, it is easy to restrict behavior by adding equations like

\[ cq \text{ out}(in(in(S))) = \text{not out}(S) \text{ if out}(S) = \text{out}(in(S)) \]

which says that you cannot get the same substance three times in a row.

For examples like this, it is also interesting to look at the final algebra \( F \), for the signature without the constant \( \text{init} \): according to the “magic formula,” it consists (up to isomorphism) of all Boolean sequences—i.e., it is the algebra of (what are called) \( \text{traces} \); in fact, contexts are the natural generalization of traces to a non-monadic world. Since there is a unique (hidden) homomorphism \( M \to F \) for any model \( M \) of \( \text{VCT} \), the image of \( \text{init} \) under this map characterizes the behavior of \( M \). This simple and elegant situation holds for nondeterministic concurrent systems in general. (More information about nondeterminism and final models can be found in [27].)

The approach to nondeterminism in hidden algebra is quite different from that which is traditional in automaton theory: in hidden algebra, each possible behavior appears in a different possible world, whereas a nondeterministic automaton includes all choices in a single model. The possible worlds approach corresponds to real computers, which are always deterministic, and must simulate nondeterminism, e.g., using pseudo-random numbers. Chip makers don’t make nondeterministic Turing machines or automata; in fact, if they could then \( P = \text{NP} \) wouldn’t be a problem!

### 3.3 Behavioral Refinement

The simplest view of behavioral refinement assumes a specification \((\Sigma, E)\) and an implementation \( A \), and asks if \( A \equiv_\Sigma E \); the use of behavioral satisfaction is significant here, because it allows us to treat many subtle implementation tricks that only “act as if” correct, e.g., data structure overwriting, abstract machine interpretation, and much more.

Unfortunately, trying to prove \( A \models_\Sigma E \) directly dumps us into the semantic swamp mentioned in the introduction. To rise above this, we work with a specification \( E' \) for \( A \), rather than an actual model\( F \). This not only makes the proof far easier, but also has the advantage that the proof will apply to any other model \( A' \) that (behaviorally) satisfies \( E' \). Hence, what we prove is \( E' \equiv E \); in semantic terms, this means that any \( A \) (behaviorally) satisfying \( E' \) also (behaviorally) satisfies \( E \); and very significantly, it also means that we can use hidden coinduction to do the proof. The method is just to prove that each equation in \( E \) is a behavioral consequence of \( E' \), i.e., a behavioral property of every model (implementation) of \( E \). More details and some examples are given in [27].

### 3.4 The Object Paradigm

Objects have local states with visible local “attributes” and “methods” to change state. Objects also come in “classes,” which can “inherit” from other classes, and objects can communicate concurrently with other objects in the same system. This paradigm has become dominant in many important application areas. We have already seen how to handle most of it with hidden algebra. Aspects of concurrency and inheritance are treated in [22, 27]. A full treatment of inheritance requires the use of order sorted algebra for subclasses [29]; this is another kind of algebra invented to help deal with software.

---

Some may object that this maneuver isolates us from the actual code used to define operations in \( A \), preventing us from verifying that code. However, we contend that this isolation is actually an advantage, since only about 5% of the difficulty of software development lies in the code itself [3], with much more of the difficulty in specification and design; our approach addresses these directly, without assuming the heavy burden of a messy programming language semantics. But of course we can use algebraic semantics to verify code if we wish, as extensively illustrated in [26]. Thus we have achieved a significant separation of concerns.
4 Summary and Related Work

This paper has presented hidden algebra as an extension of the classical initial algebra approach to abstract data types, and as a natural next step in the evolution of algebraic specification. Of course, no one would invent a method like coinduction for examples as simple as our flag example; this was chosen for expository simplicity. Many much more complex examples have been done, including correctness proofs for an optimizing compiler and for a novel communication protocol [35]; these can be viewed in our website. Hidden algebra first appeared in [20], and was subsequently elaborated in several papers, including [22, 41, 6]; an important precursor was work by Goguen and Meseguer on what they called “abstract machines” [28]. The rapidly growing literature on hidden algebra includes [12, 27, 40, 8, 15]. Coinduction seems to give proofs that are about as simple as possible, but more experience is needed before this can be said with complete certainty. The closely related area of coalgebraic semantics also uses coinduction, and also has a rapidly growing literature, including [49, 36, 37, 38]. However, it seems that coalgebra has difficulty in treating builtin data types, nondeterminism, and concurrency.

From the beginning of computer science almost sixty years ago, researchers have worked on program verification, starting with von Neumann and Turing. Now there are many different schools, there are thousands of papers, and hundreds of books. Most of this is far from rigorous, which is sad considering the topic. Perhaps the most recent really rigorous book is one that I wrote with Grant Malcolm [26]; it aims at making the best possible use of computers for proofs, and in fact is “executable” in that all its proofs run. John Reynolds has written an excellent book in a more traditional style [50].

Some early history of initial algebra semantics for abstract data types was given at the end of Section 2.2; see also [18]. It seems that algebraic specification may now be entering a golden age, in which new techniques are bringing old goals to fruition in unexpected ways, and are also opening new horizons from which exciting new goals seem reachable. We have discussed hidden algebra and its cousin coalgebra. Another important development is rewriting logic [42, 43], a weakening of equational logic providing an operational semantics that is ideal for rapid implementation of many algorithms, e.g., in term rewriting [10], as well as for describing and comparing the many kinds of concurrency [44]; rewriting logic has been efficiently implemented in the Maude system [45, 9]. The CafeOBJ system should also be mentioned [16, 14]; it provides industrial strength implementations of rewriting logic, as well as of ordinary order sorted equational logic, hidden sorted equational logic, and all their combinations [13]! The designs for both Maude and CafeOBJ are heavily indebted to that of OBJ3 and its predecessors, and indeed, can be considered extensions of OBJ3. There is also exciting new work in term rewriting [11] (which is the basis for implementing systems like OBJ3, Maude and CafeOBJ), for example in France around Prof. Jean-Pierre Jouannoud, on induction and termination proofs [5], including the spike [4] and CiME systems. The issues discussed in this paper seem to be of increasing importance for computer science, and I think we can look forward to continuing progress.

References


A OBJ3 Output

Below is the output that OBJ3 produces when it executes this paper (there is a little program that extracts the executable code from the paper, passes it to OBJ3, and puts the result in a file):

```
Welcome to OBJ3

OBJ3 version 2.04oxford built: 1994 Feb 28 Mon 15:07:40
Copyright 1988,1989,1991 SRI International
1997 Aug 26 Tue 5:44:40
```

```
 th AUTOM
[obj NATP

***> DATA is an invisible module:
[obj DATA

 th FLAG

***> prove rev rev F = F :
[op - R _ : Flag Flag -> Bool .
 [var F1 F2 : Flag .
 [eq F1 R F2 = ( up? F1 == up? F2 ) .
 [ops f1 f2 : -> Flag .
 [close

 op

 [eq up? f1 = up? f2 .
 [reduce in FLAG : up f1 R up f2
```
The true results above indicate that OBJ3 has in fact done the computations that constitute the proof.