

A Hidden Agenda

Joseph Goguen

*Dept. of Computer Science & Engineering, University of California at San Diego,
CA 92093-0114, USA*

Grant Malcolm

Dept. of Computer Science, University of Liverpool, Liverpool L69 3GL, UK

Abstract

This paper publicly reveals, motivates, and surveys the results of an ambitious hidden agenda for applying algebra to software engineering. The paper reviews selected literature, introduces a new perspective on nondeterminism, and features powerful *hidden coinduction* techniques for proving behavioral properties of concurrent systems, especially refinements; some proofs are given using OBJ3. We also discuss where modularization, bisimulation, transition systems and combinations of the object, logic, constraint and functional paradigms fit into our hidden agenda.

1 Introduction

Algebra can be useful in many different ways in software engineering, including specification, validation, language design, and underlying theory. Specification and validation can help in the practical production of reliable programs, advances in language design can help improve the state of the art, and theory can help with building new tools to increase automation, as well as with showing correctness of the whole enterprise. The utility, vitality and growing links with other areas all suggest the existence of a significant emerging field, that might be called ‘algebraic engineering’. Although its mathematical roots lie within the framework of universal (also called ‘general’) algebra, as pioneered by Noether, Birkhoff, Tarski and others, it is part of computer science. Its research agenda involves extending universal algebra to account for the realities of modern software, and this paper is largely focused on foundational aspects of that programme. However, our agenda also includes the broader task of providing real support for practical software engineering.

Today's software systems are often concurrent and distributed, with interfaces that encapsulate local states. These features are the core of what has come to be called the 'object paradigm', which may be described as comprising:

- 1 *objects*, with *local state*, plus attributes and methods;
- 2 *classes*, to classify objects through an *inheritance hierarchy*; and
- 3 *concurrent* distributed operation.

Some perhaps less basic but still important aspects include the following:

- 4 *encapsulation* and *distribution* of state;
- 5 *overloading* and *overwriting* of methods and attributes, including polymorphism and dynamic binding;
- 6 *nondeterminism*, which is closely related to concurrency and distribution;
- 7 *reactivity* and *message passing*; and
- 8 *abstract* (also called 'deferred') *classes*.

These are less basic because it can be argued that 4 is already implicit in 1, 5 in 2, 6 and 7 in 3, and 8 in the proof orientation of our programme. Items 1 and 4 constitute the older notion of abstract machine, which already supports data abstraction. The object paradigm adds support for code reuse through inheritance, and an affinity for concurrency. The hidden algebraic engineering described in the body of this paper pays particular attention to the object paradigm. However, we would emphasize that our results apply to ordinary programs and their components, since these can be regarded as abstract machines.

All this raises many difficult challenges, for both theory and practice. To better understand the situation, it is useful to distinguish among designing, coding and verifying (i.e., proving properties of) software systems. Most of the literature addresses code verification, but this can be exceedingly difficult in practice. Moreover, empirical studies have shown that little of the cost of software arises from coding errors: most of the cost of correcting programs comes from design and requirements errors [8]. Because many important programs are written in obscure and/or obsolete languages with complex ugly semantics, are very poorly documented, and are very large, often several million lines, it is usually an enormous effort to verify them, and it is very rarely worth the effort. We call this area the *semantic swamp*, and recommend that it be avoided. An additional problem is that programs in everyday use are evolving, because their context of use is evolving; this context includes computer hardware, operating systems, tax laws, user requirements, and much more. Therefore the effort of verifying yesterday's version is largely wasted, because even small code modifications can require large proof modifications: proof is

a discontinuous function of truth.

The above suggests that it is better to focus on *design and specification* than on verification. But these are also 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 formalisable, and even when they are formalisable, may involve subtle interactions among remote parts of the system. However, formal semantics can contribute to solving these problems.

It is well known that most of the effort in programming goes into debugging and maintaining (i.e., improving and updating) programs [8]. Therefore anything that can be done to ease these processes has enormous economic potential. One step in this direction is to ‘encapsulate data representations’; this means to make the actual 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 changed without having any effect on other code that uses it. If on the contrary, 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, and may use the representation in unexpected ways. This helps to explain why the so-called year 2,000 (‘Y2K’) problem is so difficult.

An encapsulated data structure with its accompanying operations is called an *abstract data type*. The crucial advance here 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 [83], and in algebra to Emmy Noether [98]; its systematisation was pioneered by Garrett Birkhoff [7], but see also [100].

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 must distinguish sorts used for data values from sorts used for states, and we need a behavioral notion of satisfaction; these are developed in Section 2.

In line with the general discussion of software methodology above, we want to prove properties of specifications, not properties of code. 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, this corresponds to verifying a *design*¹. In line with the previous paragraph, we want to verify behavioral properties and behavioral refinements; this is usually much easier than verifying the corresponding code. Behavioral refinement is much more general than ordinary refinement, and many of the enormous variety of clever implementation techniques that occur in practice require this extra generality; by contrast, refinement in so-called model-based methods (like Z) corresponds to reducing nondeterminism, and is inadequate for verifying many real designs. Section 3.5 describes techniques for verifying behavioral refinements, also showing that reducing nondeterminism is a special case of our notion of refinement.

The most important item on our agenda is to provide effective support for proving behavioral properties of systems, including refinement. We believe hidden algebra allows simpler proofs than other formalisms, because it can exploit algebraic structure that is discarded by most other approaches.

Hidden algebra developed with the object paradigm in mind [32,41], but it also supports combinations of the functional, logic, and object paradigms [33,47]; see Section 2.5 and Appendix B, and [43]. The key to combining different paradigms is to combine their underlying semantics at an appropriate level of abstraction, especially their logics, as advocated in [50]. Note that ordinary imperative programming is the special case of hidden algebra where the objects correspond to program variables; hence our work applies to traditional concurrent and sequential programming².

Because we use hidden sorts to specify classes, order sorted algebra [40,53,51] provides a natural way to handle inheritance; in addition, it supports partial functions, non-terminating systems, various kinds of subtype, error definition and recovery, coercions, overwriting, multiple representations, and more³. But for expository simplicity, this paper treats only hidden many sorted algebra. The module system of parameterized programming gives other forms of inheritance, plus the power of higher order functional programming in a first order setting which facilitates both proving and programming [30]. Because many object and logic languages provide only weak support for modularization, and because code reuse can be surprisingly hard in practice, our agenda includes

¹ Empirical studies show that real software development projects involve many false starts, redesigns, prototypes, patches, etc. [12]. Nevertheless, an idealized view of design as a sequence of refinements is still useful as a way to organize and document a project, often retrospectively [34].

² In particular, traditional pre- and post-condition techniques can be used, as extensively illustrated in our ‘executable’ algebraic text [44] on the semantics of imperative programs which shows how order sorted algebra and rewriting can add greatly to their effectiveness.

³ [40] discusses polymorphism, dynamic binding and overwriting, [51] discusses errors, coercions, etc., and [36] discusses partial functions.

powerful modularization techniques, as sketched in Appendix B.

Because constraints are inherent to hidden algebra, a new kind of logic programming can be obtained just by adding existential quantifiers. This constitutes a new item for our hidden agenda, combining the functional, object, logic, concurrent and constraint paradigms. What seems most surprising here is that it is actually *artificial* to exclude any of concurrency, nondeterminism, local states, objects, classes, inheritance, constraints, streams, existential queries, because they occur naturally in the hidden world.

The examples in this paper use OBJ3 to express theories and to do proofs, in a style where humans do the interesting high level work, while OBJ does the boring computations. Of course, much of the ‘interesting’ work could be automated, but this is certainly impossible in general. Some logical foundations for this approach are summarized in Appendix A. We assume familiarity with basic many sorted algebra and with OBJ. The relevant background appears in [44,37,58,61] and [78], among many other places. We try to avoid category theory, but in some cases (e.g, Section 3.9) its greater elegance and power are so compelling that we could not resist.

Section 2 introduces hidden algebra, with Section 2.3 giving necessary and sufficient conditions for a specification to be consistent, and Section 2.4 discussing nondeterminism. Hidden coinduction is introduced in Section 3; Sections 3.4 and 3.8 discuss the relation to labeled transition systems and bisimulation, and the generalization to multiple hidden arguments. Coinduction is applied to the correctness of refinements (i.e., implementations) in Section 3.5, and Section 4 gives some conclusions and directions for future research.

Because this paper is in part a survey, some overlap with other papers is inevitable, especially in Section 2, which contains the basic definitions. However, we have only given proofs that do not appear elsewhere, or else are very short, and most of Section 3 is new, especially Theorem 36 and Proposition 37, as well as the fundamental justification for coinduction, Theorem 26 in Section 3.9, and the relationships with colgebraic and state-based approaches in Sections 3.3 and 3.4. Appendix A is an overview of the logical relations that provide the foundation for theorem proving with systems based on term rewriting like OBJ, and Appendix B is an overview of our approach to modularity and to combining multiple paradigms.

1.2 A Little Literature

An adequate survey of the literature on concurrency would consume many volumes, and the literature on the object paradigm would surely take more; and then there are the logic, functional and constraint paradigms! Even worse,

each of these literatures seem to be growing so fast that no individual can keep up with them. The situation remains unreasonable even if we confine attention to work that is mathematically rigorous. Consequently, this subsection only considers research that seems especially closely related to ours, and that has influenced it in some way. Even this more modest task is impossible, and we beg forgiveness for our many sins of omission, and humbly request any who are sufficiently annoyed to send us their suggestions for improvement. Unfortunately, our own prior work is the the most closely related, so we must also beg the reader's indulgence for an overabundance of self-citation.

We begin with algebra. While algebra is certainly part of mathematics, computing science has made some contributions, including a simplification of notation for many sorted algebra [26], as well as initial algebra semantics (for abstract syntax and abstract data types) [26,58,59], order sorted algebra (giving a systematic approach to subtypes that also allows specifying partial recursive functions, partially defined functions, error definition and recovery, coercions and multiple representations [40,53,51]), and most recently within this tradition, hidden sorted algebra [32,41,42], or *hidden algebra* for short.

The hidden approach differs from earlier approaches in the ways that it addresses concurrency and nondeterminism, in its use of a fixed universe of data values, and in its use of behavioral satisfaction. The founding hidden paper is [32], which builds on earlier algebraic work on abstract data types [26,58,59], and is a natural extension of prior work by Goguen and Meseguer on (what they then called) abstract machines [48,78]. Hidden algebra also generalizes automaton theory, which again has a long tradition in computing science⁴; we would particularly mention the pioneering work of Eilenberg and Wright [23], which took a categorical approach to the 'tree automata' generalization of state transition automata; tree automata generalize the strong monadicism of traditional transition systems. The first systematic exposition of hidden algebra is [41], with many new ideas, including our approach to concurrency. The problem of combining hidden with ordered sorts was first solved in [10], although different solutions seem more appropriate for some purposes. Order sorted hidden algebra is further developed in [42] and [76], and [40] shows how to handle overwriting of methods using a special kind of signature. The development of coinduction began with [42,76] in the context of correctness proofs for refinements of objects and abstract machines.

The hidden approach uses behavioral satisfaction to get an algebraic treatment of state that abstracts away from implementation details. This elegant idea was introduced by Reichel [85] in the context of partial algebras; see also [86]. Behavioral equivalence of states, a generalization of bisimulation, appeared in

⁴ The authors thank Cris Calude for pointing out a very early treatment of behavioral equivalence in work of E.F. Moore [80] from 1956.

[48], which also first recognized the connection between tree automata and software engineering. Reichel’s notion of behavioral theory further developed in various directions, again mainly using partial algebras, e.g., see [6,21,22] and the survey [82].

In order to get the powerful module and type system of parameterized programming [54,27,29,28,31,60], it is necessary that the signatures (with their morphisms), models and axioms form what is called an ‘institution’ [39]. What we call ‘half institutions’ are used in [22], which claims that one cannot get a full institution for the object paradigm with behavioral satisfaction. However, this is because they fail to distinguish between fixed data values and hidden states, and between the use of hidden signature maps for vertical structure (refinements) and hidden signature morphisms for horizontal structure (module composition). In [76], we explain that these two kinds of signature arrows are appropriate for different purposes, and show that the one that should form an institution (the morphisms) does in fact do so (see also Section 3.5).

The first effective algebraic proof technique for behavioral properties was context induction, introduced by Hennicker [68] and further developed with Bidoit (e.g., [6]). Their research programme is similar to ours in several ways; however, their approach is more concerned with semantics than with proofs, and their context induction can be very awkward to apply in practice (e.g., -see [24] for a discussion of some of the difficulties). We have found that hidden coinduction eliminates the awkwardness of context induction.

Reichel’s seminal work on behavioral satisfaction was in part motivated by an insight on how to unify initial and final semantics [85]. Behavioral and final semantics were perhaps first advocated by Montanari *et al.* [25], though Wand [99] also made an early contribution. Finality is also used for treating states in [85,48,78,75], among many other places, including the present paper; there is some elegant more recent work by Reichel on co-algebraic semantics for the object paradigm [87]. Some sophisticated results on computability for initial and final algebras appear in [81]; both initiality and finality results compatible with the hidden paradigm were proved in [48]. This flood of work on finality and behavioral abstraction validates some intuitions expressed long ago by Guttag [64,65]. However, we wish to emphasize that hidden algebra does not embrace final semantics, but rather takes a *loose* approach, modulo protection of a given algebra of data values.

The hidden approach seems the first to address both concurrency and effective proof techniques for systems of objects within the algebraic tradition, although we should certainly mention rewriting logic [77], which is also in the algebraic specification tradition, and indeed, also builds on OBJ [61]. This elegant approach views actions as inference steps in a ‘rewriting logic,’ which is essentially equational logic without the law of symmetry. A strong point of

this approach is its ability to model many different approaches to concurrency in natural and simple ways. However, its treatment of objects (and messages), based on an associative-commutative (AC) ‘soupification’⁵ operation seems to raise substantial difficulties for proving properties of large systems, due to the computational complexity of AC unification, and the concrete level of this representation. In [18], behavioral specification is applied to rewriting logic, through the sophisticated semantic definition of behavioral satisfaction between rewriting logic models and (conditional) rules that arises from the very general ‘institutional’⁶ approach to behavior developed in [10].

So-called ‘process algebras’ (also called concurrency calculi), like CCS [79], ACP [4] and CSP [70], are typically presented using systems of equations. Equations are used very differently in process algebra than in hidden algebra. In process algebra, variables range over processes and the algebraic operators combine processes. Consequently, process algebra equations describe relations among ways of combining processes, rather than relations among methods and attributes, as in hidden algebra. The emphases of the three research groups differ: the CSP group has emphasized set theoretic semantics of processes as streams of actions, while the CCS group has been more concerned with equational axiomatization and decision procedures, and the ACP group has been more concerned with the use of their equations as rewrite rules; these different motivations and intuitions have led to different equations. As noted by Abramsky [1], the

veritable Babel of formalisms ... suggests that the current methodologies for concurrency are insufficiently constrained, or perhaps that some key ideas are still missing.

The lack of consensus on a suitable set of equations is discouraging, suggesting that these ‘laws of concurrency’ may not have the same status as ‘laws of nature’ in physics, despite occasional claims to the contrary. Process algebras treat an anemic⁷ special case that disallows parameterized methods and attributes, and has no explicit role for data types. Of course, process algebra can be generalized, and in fact, hidden algebra can be seen as one such generalization, which admits powerful algebraic proof methods, as opposed to tedious search through vast spaces.

⁵ This refers to the metaphor of a ‘rich soup’ of objects and messages; by contrast, in hidden algebra messages appear as terms, and do not get blended with objects.

⁶ This means it is independent of the underlying logic, and hence can be applied to order sorted algebra, rewriting logic, and many other institutions.

⁷ The technical term ‘anemic’ is defined in Section 3.4; it corresponds to the ‘strong monadicism’ mentioned above. For example, an integer stack `push` method must be represented as infinitely many transitions, one for each integer, instead of a single operation parameterized by integers.

Like process algebras, labeled transition systems (see [62]) correspond to an anemic special case of hidden algebra, where much of the structure that makes proofs easier has been discarded, and where there is no explicit place for data types used as attribute values. However, transition systems can be generalized to avoid some of these limitations, as shown in Section 3.4, where they are given additional structure to represent attribute values. Despite these limitations, we have been much inspired by the many deep insights that can be found in the mainstream of concurrency research. Abramsky [1] introduces interaction categories, a very elegant categorical approach to processes, having some similarities to our approach to concurrency that deserve further exploration. Our approach to composite systems of concurrent interacting objects is discussed in [45].

Approaches based on set-theoretic semantic models, such as Z [94] and CSP [70], can lead to very difficult proofs involving properties of sets and hence axioms for set theory⁸. Denotational semantic models (e.g., [3]) are no better in this respect, and as Abramsky [1] remarks, have only been really successful for functional languages; this should not be surprising, because denotational semantics is so strongly based on the λ -calculus. For the ambitious verifier, set-theoretic and denotational semantic definitions lead into a dangerous semantic swamp, infested with alligator-mouth-like \in s and/or sharkfin-like λ s.

There is also a distinguished tradition of research in *coalgebra*. One thread in this tradition seeks to show existence of final transition systems, which give rise to an abstract notion of bisimulation and can be used to give a semantics for process algebras [2,5]. Another thread views coalgebra as a variation on universal algebra [90], and applies it to functional programming [66,74,63], automaton theory [91,90], and the object paradigm [87,71–73,13]. An interesting recent development combines algebra and coalgebra to describe denotational and operational semantics [96].

Reichel [87] was the first to apply coalgebra explicitly to the object paradigm, and his basic construction can be seen as making precise the assertion that hidden algebra extends coalgebra with generalized constants [75,14]. It is precisely this extension that allows the treatment of nondeterminism advocated in this paper. In fact, it seems difficult to treat nondeterminism in a purely coalgebraic approach, because the obvious move of using power objects in the defining functor compromises the effectiveness of equational reasoning.

Hidden algebra is related to the elegant category theoretic approach to constraint logic programming of Diaconescu [16,17]: the fixed data universe of hidden algebra corresponds to the signature and model of the builtins, and

⁸ Of course, any algebraic laws that have been shown to hold can be used, but because any such rule set must be incomplete, such an approach is necessarily limited.

the other operations constitute an extension of the ‘logical’ signature; however, hidden algebras differ from constraint logic models, because the builtins are protected, whereas Diaconescu’s constraint logic allows confusion (in the technical sense of footnote 9) of data elements, so that disequalities may not hold; but disequalities are important for many real examples; for example, the alternating bit protocol will not be correct if its two bits are not distinct.

1.3 Acknowledgements

The research described in this paper has been supported in part by Fujitsu Laboratories Ltd., the CEC under ESPRIT-2 BRA Working Groups 6071, IS-CORE (Information Systems CORrectness and REusability) and 6112, COMPASS (COMPREhensive Algebraic Approach to System Specification and development), a contract under the management of the Information Technology Promotion Agency (IPA), Japan, as part of the Industrial Science and Technology Frontier Program ‘New Models for Software Architectures,’ sponsored by NEDO (New Energy and Industrial Technology Development Organization), and the CafeOBJ project, under management of the IPA as part of its Advanced Software Technology Programme. We thank the members of the ‘declarative’ group at Oxford for their ideas, comments, and support, without which the project described here would not be possible; in addition to ourselves, members include Corina Cîrstea, Răzvan Diaconescu, Lutz Hamel, Hendrik Hilberdink, Akira Mori, Simone Vegliani and James Worrell. We especially wish to acknowledge the contributions of Răzvan Diaconescu, who is co-author of two of the basic hidden papers, and who also made many helpful comments on drafts of this paper, and of José Meseguer, who is co-author of two of the most important pre-hidden papers, and who also made helpful comments on a draft of the paper. We are also grateful to Grigore Roşu for finding several bugs, and Uwe Wolter for useful comments.

Dedication

On behalf of both authors, Joseph Goguen makes the following dedication 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. At the Algebraic Engineering Conference in Aizu, I had the pleasure of giving the opening lecture, as a 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 style of teaching.

2 Hidden Algebra

There is an important distinction between data that is used for values (e.g., for attributes) and data that is used for internal states (e.g., of objects); the former have an immutable ‘Platonic’ character, while the latter have a mutable ‘Aristotelian’ character. Hidden algebra embodies this fundamental distinction by assigning the former to ‘visible’ sorts and the latter to ‘hidden’ sorts. States are ‘hidden’ in that they are only observed indirectly by performing ‘experiments,’ which more technically are ‘contexts,’ i.e., terms that return visible data values. Visible and hidden sorts are respectively treated in the next two subsections, which contain the most basic concepts of hidden algebra.

2.1 Visible Data Values

In order for the components of a system to communicate, they must use the same representations for any data that they share; this motivates fixing a single algebra of data values to be shared. (In practice, there could be multiple representations for data with translations among them, but our assumption can easily be relaxed to cover such cases.)

Definition 1 *Let D be a fixed data algebra, with Ψ 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 $\psi \in \Psi_{[],v}$ such that ψ is interpreted as d in D . For convenience, we assume that $D_v \subseteq \Psi_{[],v}$ for each $v \in V$. We call (V, Ψ, D) the **visible data universe**, and we call the sorts in V **visible sorts**. \square*

The condition $D_v \subseteq \Psi_{[],v}$ just says that we have fixed some notation (i.e., names) for data elements; this is quite usual for data types that are used for attributes (e.g., the numerals 0,1,2,3,... as names for natural numbers). Note that D can be any Ψ -algebra, and need not be a term model or an initial model.

Definition 1 has a semantic flavor; but verifiers need specifications for data values that support efficient proofs, and for this it is especially convenient to use *initial algebra semantics*, because it supports the proofs by induction⁹ that are so often needed for lemmas used in software proofs. For example, the following OBJ3 specification for the natural numbers is used in several later examples:

⁹ More precisely, the ‘no junk’ half of initiality supports induction over reachable algebras (see Definition 5), while the ‘no confusion’ half supports disequality proofs, which are often important in practice.

```

obj NAT is sort Nat .
  op 0 : -> Nat .
  op s_ : Nat -> Nat [prec 1].
  op _<_ : Nat Nat -> Bool .
  var N M : Nat .
  eq 0 < s N = true .
  eq N < N = false .
  eq s N < 0 = false .
  eq s N < s M = N < M .
endo

```

Here `NAT` is the name of the module and `Nat` is the name of the sort for natural numbers. The keyword pair `obj . . . endo` indicates initial algebra semantics. The underbar characters indicate where an argument goes, so that the successor operator `s_` has prefix syntax, and the inequality operator `_<_` has infix syntax. The rest of the OBJ3 syntax used here should be fairly self-evident; for more on OBJ3, see [44,61].

It is convenient for our examples in OBJ to use a fixed module `DATA` of data values, which gives the signature and axioms for D . The following says that `DATA` is just the natural numbers; this is adequate for most of this paper, noting that `NAT`, like all other modules in OBJ, imports the Booleans by default.

```

obj DATA is
  pr NAT .
endo

```

The (cumulative) signature of `DATA` is of course Ψ . Although Definition 1 requires that Ψ contains all the elements of the data algebra as constant symbols, it does no practical harm to relax this assumption in our OBJ examples. Hidden algebra simply assumes a fixed data universe: there is no requirement that this be given by a finite presentation, so if we begin with some theory P , such as `NAT`, we can simply take $\Psi_{[],s}$ to be the elements of the initial P -term algebra.

In general, one expects that data types used as values should be computable, so it is encouraging to recall that any computable algebra can be finitely specified using initial algebra semantics. Although some potential data algebras are *not* computable, such as the real numbers, even these can be captured with initial algebra semantics by using an uncountable number of constants and equations. Moreover, loose semantics can also be used, by explicitly giving any properties that are needed, and then noting that *any* Ψ -algebra D satisfying these properties could have been chosen as the fixed data algebra.

Definition 2 A hidden signature (over (V, Ψ, D)) is a pair (H, Σ) , where H is a set of **hidden sorts** disjoint from V , Σ is an $S = (H \cup V)$ -sorted signature with $\Psi \subseteq \Sigma$, such that

- S1 each $\sigma \in \Sigma_{w,s}$ with $w \in V^*$ and $s \in V$ lies in $\Psi_{w,s}$, and
- S2 for each $\sigma \in \Sigma_{w,s}$ at most one hidden sort occurs in w .

We may abbreviate (H, Σ) to just Σ . 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 $s \in V$. If $w \in V^*$ and $s \in H$, then $\sigma \in \Sigma_{w,s}$ is called a (**generalized**) **hidden constant**.

A **hidden theory** (or **specification**) is a triple (H, Σ, E) , where (H, Σ) is a hidden signature and E is a set of Σ -equations that does not include any Ψ -equations; we may write (Σ, E) or even E for short. \square

Condition (S1) expresses data encapsulation, that a hidden signature cannot add any new operations on data items. Condition (S2) says that methods and attributes act on single (states of) objects (however, this is relaxed in Section 3.8). Note that every operation in a hidden signature is either a data operation from Ψ , a method, an attribute, or a generalized hidden constant. Generalized hidden constants pick out initial states of the abstract machines that are being specified, and are quite different from methods and attributes; we refer to them as ‘generalized constants’ to emphasize this distinction, rather than to suggest that they really represent constants. Equations about data (i.e., Ψ -equations) are not allowed in hidden specifications; any such equation that might be needed as a lemma should be proved and asserted separately, not included in a specification. Definition 2 allows (what we have been calling) ‘parameters’ on attributes, methods and generalized constants, to take us beyond strong monadicism; these are just additional visible arguments. The definition also allows multiple hidden sorts, which are useful for (in the jargon of the object paradigm) *complex attributes*; these are class valued attributes.

The following should help to clarify Definition 2. It uses OBJ3 syntax for theories, where the keyword pair `th...endth` with ‘`pr DATA`’ indicates loose semantics ‘protecting’ DATA, as explained in detail a little later.

Example 3 The following OBJ3 theory specifies a cell that holds a single natural number:

```

th X is sort State .
  pr DATA .
  op init : -> State .
  op getx_ : State -> Nat .

```

```

op putx   : Nat State -> State .
var S : State .   var N M : Nat .
eq getx putx(N,S) = N .
endth

```

Here \mathbf{X} is the name of the module and \mathbf{State} is the name of the class of objects it defines, which are just cells that hold a natural number; i.e., the models of \mathbf{X} are ways to implement such cells, which for example are named by so-called ‘program variables’. The constant `init` can be seen as an initial state, the attribute `getx` returns the value in the cell, and method `putx` places a number in the cell; `getx` has prefix syntax. We could add an equation like

```
eq getx init = 0 .
```

to set the initial value of the attribute; of course, the initial value could be any other number, or it could be left undefined by just not giving such an equation (see the discussion in Section 2.4). \square

If Σ is the signature of \mathbf{X} , then Ψ is a subsignature of Σ , and a model A of \mathbf{X} should be a Σ -algebra whose restriction to Ψ is D , that provides functions for all the methods and attributes in Σ , and behaves as if it satisfies the given equations. Elements of such models are possible states for \mathbf{X} objects, i.e., for cells. This motivates the following:

Definition 4 *Given a hidden signature (H, Σ) , a **hidden Σ -algebra** A is a (many sorted) Σ -algebra A such that $A|_{\Psi} = D$. A **hidden Σ -homomorphism** is a (many sorted) Σ -homomorphism that is the identity on visible sorts. \square*

Notice that $D \subseteq \Sigma$ implies that $D \subseteq T_{\Sigma}|_{\Psi}$ and also that $g(d) = d$ for any $d \in D$. For any hidden algebra A and ground Ψ -terms t, t' (i.e., t and t' have no variables), because we are ‘given’ D , we can decide whether $A \models t = t'$ by checking whether $D \models t = t'$, by evaluating the terms in D , i.e., by checking whether $g(t) = g(t')$, where $g : T_{\Psi} \rightarrow D$ is the unique Ψ -homomorphism.

We next define behavioral satisfaction of an equation; intuitively, its two terms should ‘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 5 *Given a hidden signature (H, Σ) and a hidden sort h , then a **Σ -context** of sort h is a visible sorted Σ -term having a single occurrence of a new variable symbol z of sort h . A context is **appropriate** for a term t iff the sort of t matches that of z . We write $c[t]$ for the result of substituting t for z in the context c , and let $C_{\Sigma}[z]$ denote the V -indexed set of contexts with hidden variable z .*

A hidden Σ -algebra A **behaviorally satisfies** a Σ -equation $(\forall X) t = t'$ iff for each appropriate Σ -context c , A satisfies the equation $(\forall X) c[t] = c[t']$; then we write $A \models_{\Sigma} (\forall X) t = t'$, and we may drop the subscript Σ .

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

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

iff for every assignment $\theta : X \rightarrow A$, we have¹⁰

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

for all appropriate contexts c whenever

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

for $j = 1, \dots, m$ and all appropriate contexts c_j . As with unconditional equations, we write $A \models_{\Sigma} e$.

A **model** of a hidden theory $P = (H, \Sigma, E)$ is a hidden Σ -algebra A that behaviorally satisfies each equation in E . Such a model is also called a (Σ, E) -**algebra**, or a P -algebra, and then we write $A \models P$ or $A \models_{\Sigma} E$. Also we write $E' \models_{\Sigma} E$ iff $A \models_{\Sigma} E'$ implies $A \models_{\Sigma} E$ for each hidden Σ -algebra A . Finally, a hidden Σ -algebra A is **reachable** iff the unique Σ -homomorphism from the initial (term) Σ -algebra T_{Σ} is surjective. \square

Hidden algebra is a kind of a loose behavioral semantics over a fixed data algebra; contrary to [82], there is no competition between hidden semantics and initial algebra semantics, because they have different purposes.

Example 6 The following are some contexts for the hidden theory \mathbf{X} of Example 3:

$$\begin{aligned} c_1[z] &= \text{getx } z \\ c_2[z] &= \text{getx putx}(1, z) \\ c_3[z] &= \text{getx putx}(329, \text{putx}(1, z)). \end{aligned}$$

There are an infinite number of contexts, but they all begin with **getx**, because that is the only attribute. \square

We next give some models for the theory \mathbf{X} , each of which is a way of implementing a cell:

¹⁰ Here θ^* denotes the unique Σ -homomorphic extension of θ .

Example 7 *The simplest model C is a hidden algebra having a natural number as its state, $C_{\text{State}} = \omega$, $C_{\text{getx}}(N) = N$, $C_{\text{putx}}(N, M) = N$, and any value for C_{init} , say 7; this illustrates the nondeterminism discussed in Section 2.4.*

A more complex implementation H keeps complete histories of interactions, so that the action of $\text{putx}(N, S)$ is to concatenate N to the front of the list S of numbers. Then $H_{\text{State}} = \omega^$, the lists of natural numbers, while $H_{\text{init}} = []$, the empty list, $H_{\text{getx}}(S) = \text{head}(S)$ for $S \neq []$, $H_{\text{getx}}([])$ is say 23, and $H_{\text{putx}}(N, S) = N \frown S$, where \frown is the concatenation operation.*

Note especially that C and H are not isomorphic to each other, and moreover, that there are an infinite number of non-isomorphic variants of each. \square

For visible equations, there is no difference between ordinary satisfaction and behavioral satisfaction. But these concepts can be very different for hidden equations (i.e., equations whose terms are of hidden sort). For example,

$$(\forall N M : \text{Nat}) (\forall S : \text{State}) \text{putx}(N, \text{putx}(M, S)) = \text{putx}(N, S)$$

is strictly satisfied by the model C , but it is *not* satisfied by the history model H . However, it is *behaviorally* satisfied by both models. This shows that behavioral satisfaction is often more appropriate for computing science applications. (We will later use coinduction to prove that *every* \mathbf{X} -model satisfies this equation behaviorally.)

The simplest ways to reason about behavioral satisfaction are justified by:

Proposition 8 *In proving $E \equiv_{\Sigma} e$, the ordinary rules of equational deduction are valid, including substituting one behavioral equation into another, and of course symmetry and transitivity; visible equations can also be used in such proofs. \square*

Here ‘equational deduction’ refers to the form with explicit universal quantifiers that was introduced by Goguen and Meseguer [49], because otherwise difficulties can arise with models having empty hidden carriers. The result is easy to prove, and can be very useful. For example, if we want to prove

$$(\forall N M : \text{Nat})(\forall S : \text{State}) \text{getx putx}(N, \text{putx}(M, S)) = N$$

for the theory \mathbf{X} , we can give the following to OBJ, where the `red` command asks it to do term rewriting and return a normal form (if it finds one):

```
red getx putx(N, putx(M, S)) == N .
```

OBJ3 treats the variables as constants, and gives the correct result, `true`; see Appendix A. However something more powerful than reduction, such as coinduction, is needed to prove the equation

$$(\forall NM : \mathbf{Nat})(\forall S : \mathbf{State}) \text{putx}(N, \text{putx}(M, S)) = \text{putx}(N, S),$$

as discussed in Section 3.1.

2.3 Consistency

Unfortunately, it is easy to write theories that have no models. For example, if we add the equation

$$\text{eq } \text{putx}(N, S) = \text{putx}(N+1, S) .$$

to the theory \mathbf{X} of Example 3, then we can prove¹¹ that $1 = 0$, which contradicts the data protection of Definition 4. This motivates:

Definition 9 *A hidden theory is **consistent** iff it has at least one model with non-empty carriers.* \square

Some techniques for guaranteeing consistent specifications from [47] are summarized in Theorem 12 below, which uses the following concepts:

Definition 10 *A $\Sigma(X)$ -term is **local** iff the only visible operations in it are constants and variables (i.e., in D or in X). Let $L_{\Sigma, s}$ denote the set of local ground Σ -terms of sort s . An equation is **local** iff its left and right sides are local and its conditions (if any) are $\Psi(X)$ -terms; a set of equations is local iff each one is. A **constraint** is an equation such that both its terms have their top operations in Ψ .* \square

A non-trivial local equation cannot be a constraint. Constraints constrain the values of undefined terms over a theory, as discussed in some detail in Section 2.4, which shows how this relates to nondeterminism.

Definition 11 *A set E of Σ -equations is **D -complete** iff $D \models_{\Psi} (\forall \emptyset) t = t'$ implies $E \models_{\Sigma} (\forall \emptyset) t = t'$ for all Ψ -terms t and t' .* \square

Theorem 12 *If the equations E in a hidden theory are D -complete and are Church-Rosser and local as rewrite rules, then the theory is consistent.* \square

A proof may be found in [45]. Many examples in this paper can be shown consistent using this result. A sufficient condition for the Church-Rosser prop-

¹¹ Those unfamiliar with the Goguen-Meseguer explicit quantifier approach [49] may wish to note that what actually gets proved is the equation $(\forall S : \mathbf{State}) 1 = 0$, which is true of all models, including the one with empty hidden carrier, but which only implies the equation $(\forall \emptyset) 1 = 0$ if the hidden carrier is non-empty.

erty is that the equations are nonoverlapping¹². Once a specification has been shown consistent ignoring its nonlocal equations, the consistency of constraints can be considered separately; however, determining whether a set of constraints has a solution can be arbitrarily difficult, even unsolvable.

Example 13 *The following hidden theory for arrays can be proved consistent using Theorem 12:*

```

th ARR is sort Arr .
  pr DATA .
  op nil : -> Arr .
  op put : Nat Nat Arr -> Arr .
  op _[_] : Arr Nat -> Nat .
  var I J N : Nat .  var A : Arr .
  eq nil[I] = 0 .
  cq put(N,I,A)[J] = N if I == J .
  cq put(N,I,A)[J] = A[J] if not I == J .
endth

```

Here `nil` is the empty array, `A[I]` is the value of `A` at index `I`, and `put(N,I,A)` puts `N` at `I` in `A`. There are no hidden equations. \square

2.4 Nondeterminism

Theories of distributed programming need nondeterminism, because it is unrealistic to assume that the nodes of a network know what other nodes are going to be doing. Therefore any formalism intended to be useful for modern software engineering should treat nondeterminism in a simple and natural way. But most concurrency calculi treat nondeterminism in complex and unnatural ways, and there are sharp ongoing debates among the advocates of the various approaches, with no obvious resolution in sight, e.g., between angelic and demonic nondeterminism¹³.

Nondeterminism is inherent to the hidden paradigm; it arises whenever some attribute values are not determined by a specification. To understand this, it may help to view models as ‘possible worlds,’ where each possible combination of nondeterministic choices appears in a different world. However, this does not mean that more than one value can occur in a given world; on the contrary,

¹² For conditional equations, the left sides may overlap, but then the conditions must be disjoint.

¹³ That there are so many different kinds of nondeterminism in the standard approaches is a major cause for the Babel of mutually inconsistent ‘laws of concurrency’ mentioned in Section 1.2.

each model is deterministic, in that attributes only take one value at a time. However, a given hidden specification may have multiple models, in which the attributes have completely different values.

Definition 14 *Given a hidden theory $P = (H, \Sigma, E)$, a hidden ground Σ -term t is **defined** iff for every context c (of appropriate sort), there is some $d \in D$ such that $E \models c[t] = d$; otherwise, t is **undefined**. P is **lexic** iff all ground terms are defined. \square*

Note that ‘undefined’ simply means that a term is not constrained to be equal to any particular value, and is quite a different notion from the undefinedness of partial functions. Undefinedness is a property that holds at the level of hidden specifications; each hidden model will ‘define’ terms in a particular way. Thus if a term t is undefined, then for any context c , the term $c[t]$ will equal *some* data value in any hidden model.

Fact 15 *Given a hidden theory $P = (H, \Sigma, E)$, then:*

- 1 *A visible term t is defined iff $E \models t = d$ for some $d \in D$.*
- 2 *P is lexic iff all visible ground terms are defined.*
- 3 *P is lexic if it has no hidden (generalized) constants.*

\square

Having no undefined ground terms corresponds to Guttag’s notion of sufficient completeness [65]. However, not only do we not require this condition, but we claim that undefined terms are very useful in system development, and even at run time. Instead of having explicitly to say that something is ‘undefined,’ one simply does not define it; then it can have any value consistent with the given theory, and indeed, all possible combinations of values will occur among the models of the theory. Hidden algebra avoids the theological disputes about nondeterminism, and simply delivers a certain range of implementation freedom, in the form of a collection of possible worlds.

Example 16 *Consider the following simple theory with one hidden sort, one natural number valued attribute, one hidden constant, no equations, and the usual data (naturals and Booleans):*

```

th EX1 is sort H .
  pr DATA .
  op c :    -> H .
  op a : H -> Nat .
endth

```

There is exactly one undefined visible ground term here, namely $\mathbf{a}(c)$. Hence this theory calls for a nondeterministic choice of a natural number, and in-

deed (up to behavioral equivalence, as defined in Section 3) there is exactly one reachable Σ -algebra for each choice of a natural number for the attribute. There are also infinitely many non-reachable models; these worlds may have arbitrarily many other ‘unnamed’ (i.e., unreachable or ‘junk’) objects, each with a natural number attribute. If we add the constraint

```
eq a(H) < s s s s 0 = true .
```

then the nondeterminism is restricted so that (again up to behavioral equivalence) there are just four reachable models, each a world where the attribute of the object c has value 0, 1, 2 or 3. The unreachable models contain other objects, each of which has an attribute with value 0, 1, 2 or 3. \square

Things get more interesting when there are methods as well as attributes. Then the elements reachable from a given element of a hidden algebra are the *states* that can arise by applying methods to that element; a connected component of elements consists of all states for a single object. It is almost obligatory to test drive a new specification technology over a range of stacks, because most approaches have already done so; hence stacks are a convenient (but minimal) benchmark for comparing approaches. We first specify a non-deterministic stack. (Since this paper is limited to many sorted algebra, the handling of errors is weak; [40,53] do better, using order sorted algebra.)

Example 17 *Here the operation `push` nondeterministically puts a new natural number on top of a stack. This single operation thus corresponds to countably many nondeterministic transitions in a traditional state transition system.*

```
th NDSTACK is sort Stack .
  pr DATA .
  op empty : -> Stack .
  op push_ : Stack -> Stack .
  op top_  : Stack -> Nat .
  op pop_  : Stack -> Stack .
  var S : Stack .
  eq pop push S = S .
  eq pop empty = empty .
endth
```

Terms like `top push empty` are undefined, i.e., unconstrained or nondeterministic, and they can take any value. Each model of this specification is deterministic, and represents one possible way of resolving the nondeterminism.

Behavioral satisfaction of the first equation implies that whatever number is pushed on a stack stays there until it is popped; for example, it follows that

```
eq top pop push S = top S .
```

and that

$$\text{eq top pop pop push push S} = \text{top S} .$$

However, it is not true that

$$\text{top push pop S} = \text{top S} ,$$

because the new number pushed on S may be different from the old one.

The term `top empty` is also undefined, and hence can take any value. Of course, we could fix its value with an equation like

$$\text{eq top empty} = 0 .$$

Moreover, we could constrain `push` to just one of the four values 0, 1, 2, 3 by adding an equation like that in Example 16,

$$\text{eq top push S < s s s s 0} = \text{true} .$$

It is also possible to have several different nondeterministic `push` methods, each subject to different constraints. \square

Thus hidden semantics differs sharply from initial semantics, where terms like `top empty` would appear as new elements of sort `Nat`; it also differs from pure loose semantics, where such terms could be either new elements or else old data values.

Hidden algebraic nondeterminism can be used much as in the concurrent constraint programming paradigm: a specification describes the possible states of an object in isolation, but what states actually occur is co-determined with other objects through their interactions, expressed as constraints. For example, the specifications for an array and a pointer into it describe all their possible states separately, but when they are put together to implement a stack, many states are no longer reachable¹⁴. Thus, hidden algebra is naturally nondeterministic; we will see that it is also well suited to nonterminating (Example 34), concurrent, and reactive systems.

Example 18 *Here is a hidden version of the traditional stack theory with a non-unary deterministic push:*

```
th STACK is sort Stack .
  pr DATA .
  op empty : -> Stack .
```

¹⁴Details of this may be found in [45]; in that context, it is impossible for the array containing all 1's to occur.

```

op  push : Nat Stack -> Stack .
op  top_  : Stack  -> Nat   .
op  pop_  : Stack  -> Stack .
var  S : Stack .   var  I : Nat .
eq  top  push(I,S) = I .
eq  pop  empty    = empty .
eq  pop  push(I,S) = S .
endth

```

Here `top empty` is the only undefined ground term (up to equality). \square

Undefined values obstruct initial hidden algebras, as shown in Theorem 21 below. Recalling that $L_{\Sigma,s}$ denotes the set of local ground Σ -terms of sort s , note that any Σ -algebra M induces a hidden Σ -algebra structure on L_{Σ} which we denote L_M , by interpreting methods as term building operations, and interpreting an attribute $\sigma \in \Sigma_{w,v}$ by $(L_M)_\sigma(\ell) = M_\sigma(h_M(\ell)) = h_M(\sigma(\ell))$ for suitable $\ell \in (L_{\Sigma})_w$, where h_M is the unique Σ -homomorphism $T_{\Sigma} \rightarrow M$. Similarly, restricting h_M to local terms gives a unique hidden Σ -homomorphism $L_M \rightarrow M$ that we denote φ_M .

Proofs of the next three results may be found in [45].

Proposition 19 *Let $L_{\Sigma}[z]_s \subseteq C_{\Sigma}[z]_s$ denote the set of local Σ -contexts of sort s involving the variable z of hidden sort. Then a hidden Σ -algebra M behaviorally satisfies a hidden equation $(\forall X) t = t'$ iff it satisfies $(\forall X) c[t] = c[t']$ for every visible local context $c \in L_{\Sigma}[z]$. \square*

Proposition 20 *For a given hidden signature Σ , we have the following:*

- 1 *For any hidden Σ -homomorphism $f : M \rightarrow N$ and an unconditional equation e , if $N \models e$ then $M \models e$.*
- 2 *For any Σ -algebra M and equation e , if $M \models e$ then $L_M \models e$.*
- 3 *If a hidden theory has an initial model, then that initial model behaviorally satisfies any equation behaviorally satisfied by any other hidden model of the theory.*
- 4 *If either e is a ground equation or M is reachable, then $M \models e$ iff $L_M \models e$.*
- 5 *If there is a hidden Σ -homomorphism $f : M \rightarrow N$, then $L_M = L_N$.*

\square

Theorem 21 *A hidden theory $P = (H, \Sigma, E)$ has an initial model, denoted L_P , iff it is consistent and lexic. \square*

System development consists in part of progressively reducing implementation

freedom¹⁵, which may involve reducing nondeterminism, among other things. Reducing nondeterminism is consistent with software engineering practice, where all the operations in a program are deterministic, but at a given development stage many programs may still satisfy the specification. Thus, hidden nondeterminism is more appropriate for refinement than the forms usually found e.g., in process algebra. Nondeterminism can also remain right down to the implementation level, where any consistent value may be returned. For example, a set of constraints may be resolved only at run time, and in different ways at different times. Thus, the same notion of nondeterminism is useful for implementation freedom and for runtime choice.

2.5 More Hidden Satisfaction and the Logic Paradigm

It is easy to define hidden satisfaction for all the connectives of first order logic, and indeed of modal and other logics. For us, the most important of these is existential quantification, because it provides the existential queries that are the basis of our combined logic-object paradigm [47,43]:

Definition 22 *An (existential) Σ -query is a sentence of the form*

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

*and is **behaviorally satisfied** by a Σ -algebra A iff there is some assignment $\theta : X \rightarrow A$ such that*

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

for $j = 1, \dots, m$ and all appropriate contexts c_j . \square

Our combined logic-object paradigm uses a *hidden Herbrand theorem* to reduce reasoning over arbitrary models (e.g., of object oriented databases) to reasoning over a single ‘Herbrand universe’ term algebra, as in ordinary logic programming (see [50] for a precise statement of the corresponding result for equational logic programming). The following result is from [47]:

Theorem 23 *Given a consistent lexic hidden theory (Σ, E) , then an initial (Σ, E) -algebra G behaviorally satisfies a Σ -query q iff every (Σ, E) -algebra behaviorally satisfies q . \square*

¹⁵ However, real software development processes involve much more, including constantly evolving requirements and the resulting need to constantly evolve the software [34].

We can always choose a canonical term algebra [58] for G , and thus use narrowing to solve queries, as illustrated in [47]; of course, more efficient methods can be used in special cases. Applications of this paradigm [43] may involve queries to an object oriented database where the resulting object is not just retrieved, but actually created. For example, one might describe a holiday package (or a software package) that one wants, and then actually get the tickets and reservations (or the executable code) as the result of the query¹⁶.

The work in this section extends to Horn clause logic with equality, by applying a construction that reduces that logic to hidden equational logic (see [47], extending Diaconescu [15]). This gives a paradigm that unifies the object paradigm with equational logic programming and traditional Horn clause logic programming [50].

3 Behavior and Hidden 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 (e.g., [37,78]). We will see that final (terminal) algebras play an analogous role in justifying reasoning about behavioral properties with hidden coinduction. We first need the following:

Definition 24 *Given a hidden signature Σ , a hidden subsignature $\Phi \subseteq \Sigma$, and a hidden Σ -algebra A , then **behavioral Φ -equivalence** on A , denoted \equiv_{Φ} , is defined as follows, for $a, a' \in A_s$:*

$$(E1) \quad a \equiv_{\Phi, s} a' \quad \text{iff} \quad a = a'$$

when $s \in V$, and

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

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 θ_a is defined by $\theta_a(z) = a$ and θ_a^* denotes the free extension of θ_a .

When $\Phi = \Sigma$, we call \equiv_{Φ} just **behavioral equivalence** and denote it \equiv .

For $\Phi \subseteq \Sigma$, a **hidden Φ -congruence** on a hidden Σ -algebra A is a Φ -congruence \simeq which is the identity on visible sorts, i.e., such that $a \simeq_v a'$

¹⁶ Of course, you may wish to refine the constraints before your tickets are printed and your credit card is billed.

iff $a = a'$ for all $v \in V$ and $a, a' \in A_v = D_v$. We call a hidden Σ -congruence just a **hidden congruence**. \square

It is not hard to demonstrate the following:

Fact 25 *Given a hidden signature Σ and a hidden subsignature Φ :*

- 1 *any hidden Φ -congruence is a hidden $(\Phi \cup \Psi)$ -congruence;*
- 2 *$\Phi' \subseteq \Phi$ implies $\equiv_{\Phi} \subseteq \equiv_{\Phi'}$; and*
- 3 *behavioral Φ -equivalence is a hidden Φ -congruence.*

\square

However, the key property¹⁷ is:

Theorem 26 *If Σ is a hidden signature, Φ is a hidden subsignature of Σ , and A is a hidden Σ -algebra, then behavioral Φ -equivalence is the largest behavioral Φ -congruence on A . \square*

This result is the foundation for coinduction. Probably the most common case is $\Phi = \Sigma$, but the generalization to smaller Φ is useful, for example in verifying refinements. A simple but rather abstract proof of this theorem using so-called comma categories is given in Section 3.9, and [89] generalizes the result to multiple hidden arguments.

3.1 Coinduction

Theorem 26 implies that if $a \simeq a'$ under some hidden congruence \simeq , then a and a' are behaviorally equivalent. This justifies a variety of techniques for proving behavioral equivalence (see also [42,76]); all such techniques are varieties of coinduction. In this context, a relation may be called a **candidate relation** before it is proved to be a hidden congruence.

Example 27 *Let A be any model of the theory \mathbf{X} theory in Example 3, and for $s, s' \in A_{\text{State}}$, define $s \simeq s'$ iff $\text{getx } s = \text{getx } s'$ (and $d \simeq d'$ iff $d = d'$ for data values d, d'). Then the equation in \mathbf{X} gives us that $s \simeq s'$ implies $\text{putx}(N, s) \simeq \text{putx}(N, s')$ and of course $\text{getx } s \simeq \text{getx } s'$. Hence \simeq is a hidden congruence on A .*

Therefore we can show $A \models (\forall S : \text{State}) \text{putx}(N, \text{putx}(M, S)) = \text{putx}(N, S)$ by showing $A \models (\forall S : \text{State}) \text{getx } \text{putx}(N, \text{putx}(M, S)) = \text{getx } \text{putx}(N, S)$,

¹⁷ This elegant formulation appeared in a conversation between Grant Malcolm and Rolf Hennicker, for the special case where $\Phi = \Sigma$.

which follows by ordinary equational reasoning over \mathbf{X} . Therefore the equation is behaviorally satisfied by any \mathbf{X} -algebra A .

It is easy to do this proof mechanically with *OBJ3*, because it only uses ordinary equational reasoning. We set up the proof by opening \mathbf{X} and adding the necessary assumptions; here \mathbf{R} represents the relation \simeq :

```

openr X .
op _R_ : State State -> Bool .
var S1 S2 : State .
eq S1 R S2 = getx S1 == getx S2 .
ops s1 s2 : -> State .
ops m n : -> Nat .
close

```

The new constants $s1$, $s2$, m , n are introduced to stand for universally quantified variables (using the theorem of constants [44,37]). The line `openr X` indicates that the module \mathbf{X} is to be ‘opened with retention’ in the sense that the material added to it will be retained. The following shows that \mathbf{R} is a hidden congruence:

```

open .
  eq getx s1 = getx s2 .
  red putx(n,s1) R putx(n,s2) . ***> should be: true
close

```

Finally, we show that all \mathbf{X} -algebras behaviorally satisfy the equation with

```

red putx(m,putx(n,s1)) R putx(m,s1) .

```

All this code runs in *OBJ3*, and the reduction gives `true`. This proof seems about as simple as is possible, although it is atypical in that no lemmas about the data algebra were required. \square

The above is a simple example of what we call *hidden coinduction*, as explained further below. We now give some results to simplify such proofs. Suppose $\Sigma = \Gamma \cup \Delta$; the letters Γ and Δ are intended to suggest *generators* (also called *constructors*) and *destructors* (also called *selectors*), respectively¹⁸ [76]. In Example 27, Δ contains `getx` and Γ contains `putx`.

Corollary 28 *If $\Sigma = \Delta \cup \Gamma$ and if \equiv_{Δ} on a Σ -algebra A is preserved by Γ , then $\equiv_{\Delta} = \equiv_{\Sigma}$ on A . More generally, if $\Psi \subseteq \Phi = \Delta \cup \Gamma \subseteq \Sigma$ and \equiv_{Δ} is preserved by Γ , then $\equiv_{\Delta} = \equiv_{\Phi}$.*

¹⁸ Information on order sorted constructors and selectors appears in [51].

Proof: We show the more general result. By Fact 25, \equiv_{Δ} is a hidden $(\Delta \cup \Psi)$ -congruence that contains behavioral Φ -equivalence, since $\Delta \subseteq \Phi$. If \equiv_{Δ} is preserved by Γ , then it is a hidden $\Delta \cup \Gamma = \Phi$ -congruence, and the desired result follows from Theorem 26. \square

Verifiers want to do as little work as possible. Hence they do not want to bother with Ψ at all, and they do not want any overlap between Δ and Γ , i.e., they want to use Δ and Γ such that $\Phi = \Delta + \Gamma + \Psi$, where ‘+’ denotes disjoint union for operations and ordinary union for sorts. For the object paradigm, it is often natural to let Δ contain attributes and Γ methods; then we can give a simple syntactic definition for \equiv_{Δ} :

Proposition 29 *If $\Phi = \Delta + \Gamma + \Psi$ where Δ consists of attributes, if A is a Σ -algebra, and if we define $aR_{\Delta}a'$ iff $\delta(a, d) = \delta(a', d)$ for all $\delta \in \Delta$ and all $d \in A_w$ where w is the arity of δ , then $R_{\Delta} = \equiv_{\Delta}$. Thus, if R_{Δ} is preserved by Γ , then R_{Δ} is behavioral Φ -equivalence.*

Proof: (E2) of Definition 24 is equivalent to the definition of R_{Δ} because all operations in Δ are visible. \square

The above shows that \mathbb{R} as defined in Example 27 really is \equiv_{Δ} . Furthermore, if Γ consists of methods and Δ of attributes, and if the equations satisfy a certain common property, then R_{Δ} is automatically preserved by Γ :

Definition 30 *If $\Phi = \Delta + \Gamma + \Psi$ where operations in Δ are visible and in Γ are hidden, then a set E of Σ -equations is Δ/Γ -**complete** iff for all $\delta \in \Delta$, $m \in \Gamma$, there is some $t \in T_{\Delta \cup \Psi}(\{x\})$ such that using E we can prove*

$$\delta(d, m(d', x)) = t ,$$

with x of hidden sort h' , $\delta \in \Delta_{wh,v}$, $m \in \Gamma_{w'h',h}$, $d \in D_w$, and $d' \in D_{w'}$. \square

The following is a straightforward corollary to Proposition 29:

Proposition 31 *If $\Phi = \Delta + \Gamma + \Psi$ with operations in Δ visible and in Γ hidden, if A is a hidden Σ -algebra, and if E is Δ/Γ -complete, then R_{Δ} is preserved by Γ , and therefore R_{Δ} is behavioral Φ -equivalence. \square*

In the special case where equations involving Γ have the form

$$\delta(m(x)) = t ,$$

for all δ and m , with x of hidden sort h , $\delta \in \Delta$, $m \in \Gamma$, $t \in T_{\Delta \cup \Psi}(\{x\})$, it is easy to see that E is Δ/Γ -complete. (This result was suggested to us by Răzvan Diaconescu.)

To summarize, **hidden coinduction** is the proof technique where we define a relation, show it is a hidden congruence, and then show behavioral equivalence of two terms by showing that they are congruent. Many of the concepts and results of this subsection are generalized in [89] and [88].

The way we define the congruence relation in a coinductive proof can have a dramatic effect on how the proof applies to models. If the relation is defined inductively over some constructors, then given a model A , the congruence is only defined on the subalgebra $A_0 \subseteq A$ generated by those constructors in A ; this is the subalgebra that is reachable using those constructors. More specifically, the proof that such a candidate relation is a congruence might proceed by induction on the given constructors; in this case, what is proved is that the relation is a congruence on the subalgebra A_0 . Usually we don't care whether or not a behavioral equation is satisfied by unreachable states, because these states cannot occur when the machine is run. The restriction of hidden algebra to reachable models is further studied in [97].

3.2 Another Coinduction Example

A parallel connection (see Section 3.7) of two cells of the kind defined by the specification X of Example 3 can be specified as follows:

```

th XY is sort State .
  pr DATA .
  op init : -> State .
  ops (getx_) (gety_) : State -> Nat .
  ops putx puty : Nat State -> State .
  var S : State .
  vars M N : Nat .
  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 .
endth

```

The operations of the two cells are tagged by x , y , respectively. Note the nondeterminacy of values for the initial state `init`. The last two equations say that the operations of the cells do not interfere with each other, and intuitively it seems that the stronger noninterference assertion

$$\text{putx}(M, \text{puty}(N, S)) = \text{puty}(N, \text{putx}(M, S))$$

should also hold. This is another equation that does not strictly hold in all models of XY , but that does hold behaviorally, and can be proved using coin-

duction. To this end, let Δ contain `getx` and `gety`, and let Γ contain `putx` and `puty`. We then verify the equation using Proposition 29 and Corollary 28 as follows:

```

openr XY .
  op _R_ : State State -> Bool .
  var S1 S2 : State .
  eq S1 R S2 = getx S1 == getx S2 and gety S1 == gety S2 .
  ops s1 s2 : -> State .
  ops n m : -> Nat .
close
***> first show R is a congruence:
open .
  eq getx s1 = getx s2 .
  eq gety s1 = gety s2 .
  red putx(n,s1) R putx(n,s2) . ***> should be: true
  red puty(n,s1) R puty(n,s2) . ***> should be: true
close
***> now check the equation:
red putx(m,puty(n,s1)) R puty(n,putx(m,s1)) .

```

Since the last reduction yields `true`, the equational is behaviorally satisfied by all models of `XY`.

Of course, this example was chosen for expository simplicity, but coinduction has also been used in much more complex proofs, including correctness of a rather sophisticated optimizing compiler [67], and of a distributed concurrent truth maintenance protocol [55]. These and a number of other examples can be found on the web, at www.cse.ucsd.edu/groups/tatami/.

3.3 Finality

This paper does *not* advocate a final algebra semantics; instead, our semantics for hidden theories is a loose behavioral semantics with a standard interpretation for data. In practice, the best implementations are often neither initial nor final, but somewhere in between. However, final hidden algebras are important for our theoretical development, and in particular, they play a key role in justifying hidden coinduction. The construction of a final algebra F_Σ below follows [10], and should help our intuition to grasp what is going on.

Given a hidden signature Σ without generalized hidden constants (recall these are hidden operations with no hidden arguments), the hidden carriers of F_Σ are given by the following ‘magical formula’:

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

the product of the sets of functions taking contexts to data values (of appropriate sort).

Elements of F_{Σ} can be thought of as ‘abstract states’ represented as functions on contexts, returning the data values resulting from evaluating a state in a context; thus they are a kind of continuation. This also appears in the way F_{Σ} 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: Let $\sigma \in \Sigma_{hw,h'}$ be a method, let $p \in F_{\Sigma,h}$ and let $d \in D_w$; then $F_{\Sigma,\sigma}(p,d)$ must be an element of

$$F_{\Sigma,h'} = \prod_{v \in V} [C_{\Sigma}[z_{h'}]_v \rightarrow D_v] .$$

For $v \in V$ and $c \in C_{\Sigma}[z_{h'}]_v$, define

$$(F_{\Sigma,\sigma}(p,d))_v(c) = p_v(c[\sigma(z_h, d)]) ;$$

i.e., with a slight abuse of notation, given an abstract state p , the result we get from looking at $\sigma(p,d)$ in a context c is the same as the result that p gives in the context $c[\sigma(z_h, d)]$.

Theorem 32 *For any hidden signature Σ without generalized hidden constants, F_{Σ} is a final hidden Σ -algebra.*

Proof: *The preceding paragraph shows that F_{Σ} is a hidden Σ -algebra. Given another hidden Σ -algebra A , there is a hidden Σ -homomorphism $g : A \rightarrow F_{\Sigma}$ taking $a \in A_h$ to the family (over $v \in V$) of mappings $C_{\Sigma}[z_h]_v \rightarrow D_v$ that sends c to $A_c(a)$. It is straightforward to check that g is unique. \square*

Given a hidden Σ -algebra A , the unique Σ -homomorphism $A \rightarrow F_{\Sigma}$ takes a hidden state to all its observable behaviors; it can be thought of as evaluating all attributes for all states that can be reached from the given state.

Example 33 *For Σ the signature of \mathbf{X} in Example 3, the unique Σ -homomorphism from a \mathbf{X} -algebra A to F_{Σ} maps a state $s \in A_{\text{State}}$ into the set of all assignments of boolean values to all contexts, i.e., to each of the following:*

```

getx s
getx putx(0, s)          getx putx(1, s)          .....
getx putx(0, putx(0, s))  getx putx(0, putx(1, s))  getx putx(1, putx(0, s))
.....
.....

```

□

Example 34 *A class of infinite streams of numbers can be specified as follows:*

```

th STREAM is sort Stream .
  pr DATA .
  op hd_ : Stream -> Nat .
  op tl_ : Stream -> Stream .
endth

```

Here `hd` gives the first value in the stream, and `tl` gives the remainder of the stream. The final algebra for this theory is the set of infinite lists of numbers, and the unique homomorphism from a **STREAM**-algebra A to the final algebra F_Σ maps $a \in A$ to the infinite list of numbers

`hd a, hd tl a, hd tl tl a, ...`

□

We now show that Theorem 32 generalizes to give final models for all consistent lexic theories. Definedness is exactly what allows constants to be interpreted in the final model. First, we need the following:

Definition 35 *Given a hidden signature Σ , let Σ^\diamond denote Σ with all hidden constants removed. Given a Σ -algebra A , let A^\diamond denote A viewed as a Σ^\diamond -algebra. □*

Theorem 36 *If each equation in a hidden theory $P = (H, \Sigma, E)$ has at most one variable of hidden sort, then P has a final model, denoted F_P , iff it is consistent and lexic.*

Proof: *Suppose P is consistent and lexic, and for any P -algebra A , let $\varphi : A^\diamond \rightarrow F_{\Sigma^\diamond}$ be the unique hidden Σ^\diamond -homomorphism to the final algebra F_{Σ^\diamond} , made into a hidden Σ -algebra by interpreting generalized constants $\sigma \in \Sigma_{w,h}$ by $(F_{\Sigma^\diamond})_\sigma(d) = \varphi((A)_\sigma(d))$ for all $d \in D_w$; note that φ is a hidden Σ -homomorphism. Let F_A be the image of φ , i.e., factor φ as the composition of surjective $\varphi_0 : A^\diamond \rightarrow F_A$ and inclusive $\varphi_1 : F_A \hookrightarrow F_{\Sigma^\diamond}$. Because φ_0 is surjective, Lemma 37 below implies that $F_A \models E$. Now let F_P be the greatest subalgebra of F_{Σ^\diamond} that behaviorally satisfies E ; in fact, this is the union of all*

the images F_A for each hidden P -algebra A . For any equation in E with variables X , because at most one variable in X is of hidden sort, any assignment $\theta : X \rightarrow F_P$ is an assignment $\theta : X \rightarrow F_A$ for some A , and so $F_P \models E$. For any P -algebra A , we have already noted that F_A is a subalgebra of F_{Σ° that behaviorally satisfies E ; therefore it is contained in F_P , which shows that the domain of φ lies in F_P , which is therefore final. This concludes the ‘if’ direction of proof. The converse proof is like that of Theorem 21. \square

Proposition 37 *Given a surjective hidden homomorphism $h : A \rightarrow B$ and an equation e , then $A \models e$ implies $B \models e$. Given $\Delta \subseteq \Sigma$ and a hidden Σ -homomorphism $h : A \rightarrow B$, then $a \equiv_\Delta a'$ in A iff $h(a) \equiv_\Delta h(a')$ in B for all a, a' in A .*

Proof: Let e be of the form $(\forall X) t = t'$ if $t_1 = t'_1, \dots, t_m = t'_m$, and let $\theta : X \rightarrow B$ be such that for $i = 1, \dots, m$, $\theta^*(t_i) \equiv_B \theta^*(t'_i)$. Because h is surjective, there is some $\rho : X \rightarrow A$ such that $\theta = \rho \circ h$. Therefore $h(\rho^*(t_i)) \equiv_B h(\rho^*(t'_i))$, and so by the second assertion, $\rho^*(t_i) \equiv_A \rho^*(t'_i)$ for $i = 1, \dots, m$. If $A \models e$, this means that $\rho^*(t) \equiv_A \rho^*(t')$, so by the second assertion, $h(\rho^*(t)) \equiv_B h(\rho^*(t'))$, i.e., $\theta^*(t) \equiv_B \theta^*(t')$, and therefore $B \models e$.

For the second assertion, by definition, $a \equiv_\Delta a'$ is equivalent to $A_c(a) = A_c(a')$ for all $c \in L_\Delta[z]$, which is equivalent to $h(A_c(a)) = h(A_c(a'))$ for all $c \in L_\Delta[z]$, because h is the identity on visible sorts; moreover, because h is a homomorphism, this in turn is equivalent to $B_c(h(a)) = B_c(h(a'))$ for all $c \in L_\Delta[z]$, which is by definition $h(a) \equiv_\Delta h(a')$. \square

Dual to the construction of an initial algebra for a theory as a quotient by the congruence defined by its equations, the final hidden algebra F_P of a theory P is the greatest subalgebra of F_Σ that satisfies the equations¹⁹. The unique homomorphism to the final algebra can be thought of as mapping each state to all the observations that can be made upon it. We can get such a function for an arbitrary Σ -algebra A by forgetting that certain hidden elements of A are named by hidden constants in Σ . This motivates the following:

Proposition 38 *Two elements of a hidden Σ -algebra A are behaviorally equivalent iff they map to the same element under the unique Σ° -homomorphism $A^\circ \rightarrow F_{\Sigma^\circ}$ to the final Σ° -algebra F_{Σ° .*

Proof: This follows straightforwardly from the definition of behavioral equivalence and the ‘magic formula’ that defines F_Σ . \square

In other words, behavioral equivalence on an algebra is the kernel of the unique homomorphism to the final algebra [75]. The quotient under this equivalence

¹⁹ This exists because there is at least one model. This issue is given a categorical treatment in [88].

gives an algebra which is used in [10] to define behavioral satisfaction. The constructions for initial and final abstract machines given in [48] are very similar to those given here, and are perhaps the first in the literature.

3.4 Bisimulation and Transition Systems

This section considers the relationship between hidden theories and certain classes of labeled deterministic transition systems: we show that models of anemic hidden theories correspond to labeled transition systems that have been given an additional structure to handle attributes, and that bisimulation proofs are anemic coinduction proofs.

Traditional state transition systems and process algebras [62,4] consider systems with a single global state and just one kind of data; hence they correspond to specializing our hidden paradigm to just one hidden sort, say \mathbf{h} , and just one visible sort, say \mathbf{v} ; thus we may write D for $D_{\mathbf{v}}$. But to capture the traditional notion, we must discard even more algebraic structure: because only unary operations are allowed, we must replace each operation $\sigma \in \Sigma_{\mathbf{h}w,s}$ by a collection of operations $\sigma_d \in \Sigma_{\mathbf{h},s}$, one for each $d \in D_w$, so that $\sigma_d(x) = \sigma(x, d)$; that is, we must fix each visible sorted argument. This is possible because $D \subseteq \Psi_{[],\mathbf{v}} \subseteq \Sigma_{[],\mathbf{v}}$; finally we must also forget the algebraic structure on D . This impoverishment means that structural facts like $\text{pop push}(\mathbf{X}, \mathbf{S}) = \mathbf{S}$ cannot be expressed, and hence cannot play the useful role in verification that they should. However, impoverishing hidden algebra isn't enough: we must also enrich the traditional transition system with an additional structure for attributes.

Definition 39 *An anemic signature is a hidden signature with no hidden constants and with just two sorts, \mathbf{h} hidden and \mathbf{v} visible, such that each operation has at most one argument, and that argument is hidden²⁰.*

*Given an anemic signature Σ , then a Σ -transition system is (N, δ, α) , where N is a set (of **states**), $\delta : N \times \Sigma_{\mathbf{h},\mathbf{h}} \rightarrow N$ is a **state transition function**, and α is an **output function** $N \rightarrow [C_{\Sigma}[z]_{\mathbf{v}} \rightarrow D]$.*

*A **morphism of Σ -transition systems**, $h : (N, \delta, \alpha) \rightarrow (N', \delta', \alpha')$, is a function $h : N \rightarrow N'$ such that $\alpha = \alpha' \circ h$ and $h(\delta(n, \sigma)) = \delta'(h(n), \sigma)$. \square*

In traditional transition systems, what is visible is essentially the sequence of transitions, called a *trace* (more precisely, a trace is a sequence of transition labels). Attributes are not explicit, and are considered *inferable* from traces. For example, in a cookie vending transition system, the number of cookies

²⁰ It follows that Ψ contains only constants, i.e., $\Psi_{[],\mathbf{v}} = D$ and $\Psi_{w,s} = \emptyset$ otherwise.

left in the machine can be calculated from the number of `give-cookie` transitions in the trace. Our definition of transition system differs from tradition in having an output function that gives *attribute values* for states. Our definition also differs in that transitions are *total* and *deterministic*: in any state n , a transition σ leads to precisely one new state, namely $\delta(n, \sigma)$. This is not a problem, because as we have already argued, hidden nondeterminism is at least as powerful and elegant as standard approaches; the idea is that instead of a single nondeterministic machine, we will have a collection of ‘possible’ deterministic machines.

Given an anemic signature Σ and a Σ -algebra A , we get a Σ -transition system $(N_A, \delta_A, \alpha_A)$, where $N_A = A_{\mathbf{h}}$, $\delta_A(a, \sigma) = A_\sigma(a)$, and α_A is the unique homomorphism to the final Σ -algebra F_Σ . Also, any hidden Σ -homomorphism $h : A \rightarrow A'$ gives a morphism of Σ -transition systems $h : (N_A, \delta_A, \alpha_A) \rightarrow (N_{A'}, \delta_{A'}, \alpha_{A'})$. Moreover, every Σ -transition system (N, δ, α) gives rise to a hidden Σ -algebra N_δ^α , where $(N_\delta^\alpha)_{\mathbf{h}} = N$, and for methods $\sigma \in \Sigma_{\mathbf{h}, \mathbf{h}}$ we define $(N_\delta^\alpha)_\sigma(n) = \delta(n, \sigma)$, and for attributes $\sigma \in \Sigma_{\mathbf{h}, \mathbf{v}}$ we define $(N_\delta^\alpha)_\sigma(n) = \alpha(n)(\sigma(z))$. The following is straightforward:

Proposition 40 *The above defines a one-to-one correspondence between hidden Σ -algebras and Σ -transition systems. Moreover, every anemic signature Σ has a final Σ -transition system, $FT_\Sigma = (F_{\Sigma, \mathbf{h}}, \delta_\Sigma, 1)$, where 1 denotes the identity function on $F_{\Sigma, \mathbf{h}}$ and $\delta_\Sigma(f, \sigma) = (F_\Sigma)_\sigma(f)$, i.e., $\delta_\Sigma(f, \sigma)(c) = f(c[\sigma(z)])$. The unique Σ -transition morphism from (N, δ, α) to the final Σ -transition system is α . \square*

Finality of FT_Σ gives rise to the traditional bisimilarity relation for deterministic transition systems, in the same way that finality of F_Σ gives rise to the behavioral equivalence relation on hidden algebras:

Definition 41 *Given an anemic signature Σ , a **bisimulation** on a Σ -transition system (N, δ, α) is a relation $B \subseteq N \times N$ such that $(n_1, n_2) \in B$ implies:*

- 1 $\alpha(n_1) = \alpha(n_2)$.
- 2 $(\delta(n_1, \sigma), \delta(n_2, \sigma)) \in B$ for all $\sigma \in \Sigma_{\mathbf{h}, \mathbf{h}}$.

\square

The first condition says bisimulations are coherent with respect to observations given by α ; the second is the standard bisimulation condition for deterministic transition systems.

Example 42 *Let Σ be obtained from the signature of \mathbf{X} in Example 27 by converting the parameterized operation `putx`(n, s) into the infinite family of operations `putx.n`(s). Then one Σ -transition system is (ω, δ, α) , where $\delta(s, \text{putx.n}(c)) = n$, and where α is defined recursively on contexts as follows:*

$$\begin{aligned}\alpha(s)(\mathbf{getx}(z)) &= z \\ \alpha(s)(c[\mathbf{putx.n}(z)]) &= \alpha(\mathbf{n})(c)\end{aligned}$$

Now $B = \{(s_1, s_2) \mid \alpha(s_1)(\mathbf{getx}(z)) = \alpha(s_2)(\mathbf{getx}(z))\}$ is a bisimulation. Note that in the hidden Σ -algebra corresponding to this transition system, $\alpha(s)(\mathbf{getx}(z))$ corresponds to the attribute \mathbf{getx} in the state s . Example 27 shows that two states of a \mathbf{X} -model are behaviorally equivalent iff they have the same \mathbf{getx} -values; the relation \simeq of that example corresponds to the bisimulation B here, and the proof that \simeq is a hidden congruence shows that B is indeed a bisimulation. \square

The two conditions in Definition 41 state that a bisimulation is a Σ -congruence; in particular, the first condition ensures that we can extend a bisimulation to an S -sorted relation that is the identity on visible sorts. This means that every bisimulation on a Σ -transition system gives rise to a hidden congruence on the corresponding hidden algebra, and conversely, any hidden congruence on a hidden algebra gives rise to a bisimulation on the corresponding Σ -transition system. Many results about hidden congruences translate across this correspondence to give results about bisimulations. In the standard terminology, two states are called **bisimilar** iff they are related by some bisimulation; in other words, bisimilarity is a maximal bisimulation, just as behavioral equivalence is a maximal hidden congruence. Indeed, two states of a Σ -transition system are bisimilar iff they are behaviorally equivalent in that system viewed as a hidden Σ -algebra. Bisimilarity arises from morphisms to the final transition system in the same way that behavioral equivalence arises from homomorphisms to the final Σ^\diamond -algebra (cf. Proposition 38):

Proposition 43 *Given an anemic signature Σ and a Σ -transition system A , two states of A are bisimilar iff the unique morphism to the final transition system (i.e., the final algebra) maps those states to the same element. \square*

In other words, for a Σ -transition system (N, δ, α) , two states n and n' are bisimilar iff $\alpha(n) = \alpha(n')$. This is another way of looking at anemic coinduction: two states of a transition system are bisimilar iff those states are behaviorally equivalent in the transition system viewed as an algebra of a hidden signature (cf. Theorem 38).

Anemic hidden algebras and nondeterministic transition systems are related via an *adjunction*, giving a ‘best’ hidden algebra with the given behaviour for every nondeterministic transition system, and *vice versa*.

In an alternative approach, nondeterministic Σ -transition systems are like Σ -transition systems (N, δ, α) , except δ returns *sets* of states, i.e., each method is interpreted as an action that gives a nondeterministic choice of result states, or

possibly no result if δ returns the empty set. All Σ -transition systems are non-deterministic systems, if we think of δ as always returning a singleton set, so any hidden algebra gives a nondeterministic Σ -transition system, as above. To construct a hidden algebra from a nondeterministic transition system, note that hidden contexts (i.e., sequences of methods) form a transition system where δ takes a context c and a method σ in Σ and returns the singleton set $\{\sigma(c)\}$. For any nondeterministic Σ -transition system (N, δ, α) , we construct a hidden algebra whose carrier set is the set of functions from hidden contexts to N that preserve transitions, i.e., the set of functions f such that for all contexts c and methods σ , we have $f(\sigma(c)) \in \delta(f(c), \sigma)$. This algebra interprets a method σ as mapping a function f to the function $c \mapsto f(\sigma(c))$ for hidden contexts c . Transition preserving functions from contexts to N correspond to deterministic ‘paths’ through the nondeterministic transition system, so the hidden algebra we have constructed can be thought of as the largest deterministic subsystem of (N, δ, α) . Unfortunately, verification will in general be more difficult than in the hidden algebra setting, because uniformities that might be expressed by equations (or their lack) will be lost in the set theoretic representation.

To summarize, our main points have been that: (1) hidden algebra generalizes traditional transition systems to nonanemic signatures, capturing additional algebraic structure of methods, attributes and states; (2) bisimulation is the anemic special case of hidden coinduction; (3) the extra structure of hidden algebra makes verification easier; and (4) hidden nondeterminism is more graceful and verification-friendly than that of traditional transition systems.

3.5 Refinement

Hidden coinduction is applied to correctness proofs of refinements in [45]; in this section we summarize some of the main definitions and results to demonstrate the utility of coinduction for this purpose. The simplest view of refinement assumes a specification (Σ, E) and an implementation A , and asks whether $A \models_{\Sigma} E$; the generalization to behavioral satisfaction is significant here, as 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 described in the introduction. To rise above this, we work with a specification E' for A , rather than an actual model²¹. This not only makes

²¹ 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*. Empirical studies show that little of the

the proof far easier, but also has the advantage that the proof will apply to any other model A' that satisfies E' . Hence, what we prove is $E' \equiv E$; in semantic terms, this means that any A satisfying E' also satisfies E , but very significantly, it also means that we can use hidden coinduction to do the proof.

A more sophisticated view of refinement [48,92,68,82] allows the concrete implementation to rename or even identify some of the abstract sorts and operations, thus giving rise to a hidden signature map from the abstract to the concrete signature:

Definition 44 *A hidden signature map $\varphi : (H, \Sigma) \rightarrow (H', \Sigma')$ is a signature morphism $\varphi : \Sigma \rightarrow \Sigma'$ that preserves hidden sorts and is the identity on (V, Ψ) . A hidden signature map $\varphi : \Sigma \rightarrow \Sigma'$ is a **refinement** $\varphi : (\Sigma, E) \rightarrow (\Sigma', E')$ iff $\varphi A' \models_{\Sigma} E$ for every (Σ', E') -algebra A' . \square*

(In the above, $\varphi A'$ denotes A' viewed as a Σ -algebra.) It can be shown that φ is a refinement if all visible consequences of the abstract specification hold in the concrete specification [76]:

Proposition 45 *A hidden signature map $\varphi : (\Sigma, E) \rightarrow (\Sigma', E')$ is a refinement if $E' \models \varphi(c[e])$ for each $e \in E$ and each visible Σ -context c , where if e is the equation $(\forall X) t = t'$, then $c[e]$ denotes the equation $(\forall X) c[t] = c[t']$. \square*

Further consequences of Corollary 28 can be used to justify applying hidden coinduction for proving correctness of refinements; see [76,45] for details and examples. Correctness proofs for refinements involve showing that the concrete specification has the desired behaviour, and generally make use of the concrete equations.

We are often asked how our approach to refinement relates to the so-called model-based approaches of Z, VDM, etc., which follow Hoare [69] in defining refinement to be a relationship between two models, one ‘abstract’ and the other ‘concrete’, mapping variables in the concrete model to the abstract objects that they represent. We believe that the often complex structure of the concrete models, and their set-theoretic nature, make model-based correctness proofs unnecessarily difficult. Our approach instead uses *theories* at both the concrete and abstract levels, and does proofs in an axiomatic setting designed to facilitate reasoning. In particular, our notion allows stepwise refinement without choosing concrete representations for variables; such a choice corre-

difficulty of software development lies in the code itself (only about 5% [8]); much more of the difficulty lies in specification and design, and our approach addresses these directly, without assuming the heavy burden of a messy programming language semantics. Of course we can use algebraic semantics to verify code if we wish, as extensively illustrated in [44]. Hence what this maneuver actually achieves is a significant separation of concerns.

sponds to fixing a model, and it is good engineering practice to delay such a heavy commitment for as long as possible. Also, verifying correctness of subtle representation changes is often eased by behavioral satisfaction.

For us, an implementation is correct if it is a model of the concrete theory, and verifying this is easier than showing that it satisfies the abstract specification, because the structures are much closer together. The perhaps surprising fact that mappings go in opposite directions for specifications and for models is explained at an abstract level by the theory of institutions [39], thus demonstrating a natural duality between the model-based and the theory-based notions of refinement.

3.6 Examples of Refinement

We give an example illustrating how our notion of refinement encompasses nondeterminism reduction (as in model-based approaches – see the discussion in Section 2.4, and for further comparison with model-based approaches to refinement, see [45]). We show that the two cell theory \mathbf{XY} can be implemented by an enrichment of \mathbf{ARR} , a theory of (infinite) arrays; this means that every model of an array yields two independent cells. Because initial values for the two cells are not determined, but initial values for the array are determined, this refinement involves a reduction²² of nondeterminism, as well as a refinement of the data structure.

```

th XYA is pr ARR *(sort Arr to State) *(op nil to init).
  ops (getx_) (gety_) : State -> Nat .
  ops putx puty : Nat State -> State .
  var S : State .    var N : Nat .
  eq getx S = S[1] .
  eq gety S = S[2] .
  eq putx(N,S) = put(N,1,S).
  eq puty(N,S) = put(N,2,S).
endth

*** now the proof:
open .
  op s : -> State .
  op n : -> Nat .
  red getx putx(n,s) == n .
  red gety puty(n,s) == n .

```

²² Actually the nondeterminism is completely eliminated here; of course, it is easy to find examples of hidden refinement where nondeterminism is reduced but not completely eliminated.

```

    red getx puty(n,s) == getx s .
    red gety putx(n,s) == gety s .
close

```

Here the phrase `ARR *(sort Arr to State)` renames the sort `Arr` of the module `ARR` to be `State`, and `*(op nil to init)` further renames its operation `nil` to be `init`.

OBJ3 does a total of 28 rewrites for this proof. Each equation in the abstract theory (the one to be implemented) is strictly satisfied by the concrete theory, so that coinduction is not needed. Some refinement proof that do require coinduction can be found on the web, at www.cse.ucsd.edu/groups/tatami, and a coinduction proof that a parallel connection of a pointer with an array refines the stack theory appears in [45]; this one uses a total of 120 rewrites.

3.7 Concurrency

This section briefly summarizes work on concurrency in the hidden paradigm, originating with the notion of independent sum in [41], and further developing in [45] with the notion of concurrent connection. Concurrency is natural to the hidden paradigm, in that no order of execution is specified by hidden theories; in particular, concurrent execution is legal whenever it is possible.

Recalling the cell specification X of Example 3, let Y specify another such cell by everywhere replacing X and x by Y and y , respectively. A concurrent connection of X and Y should be a specification $X \parallel Y$ with one hidden sort, where the operations of X and Y have the same semantics as before, but do not interfere with each other. The specification XY of Section 3.2 does exactly this, combining X and Y , identifying their sorts, and adding equations to express noninterference.

This is a special case of a very general construction, which also provides for synchronizing objects. The main result so far says that there is a certain ‘universal characterization’ (in the sense of category theory) for the concurrent connection [41,13]; this is useful for proving general laws about it. Another result is that a concurrent system is deadlock free iff its concurrent connection is consistent; this is because deadlock means that the equations expressing synchronization do not have any models. Hidden coinduction is a powerful tool for proving properties of systems defined by concurrent connection, and communication protocols provide many suitable examples [97].

Roşu and Goguen show in [89] that the restriction to monadic operations (i.e., (S2) of Definition 2) is not necessary, and that many properties of hidden algebra still hold if this restriction is dropped. Roşu and Goguen also consider specifications where behavioral equivalence is determined by a fixed subsignature, as in Definition 24. In this section we briefly discuss the consequences of this, particularly with respect to the behavioral specification of nondeterministic behaviors.

The following definition from [89] uses a fixed data universe (V, Ψ, D) , as in Section 2.1.

Definition 46 *A behavioral specification is a tuple (H, Φ, Σ, E) , where H is a set of hidden sorts disjoint from V , and Φ and Σ are $(H \cup V)$ -sorted signatures such that $\Psi \subseteq \Phi \subseteq \Sigma$ and each $\sigma \in \Sigma_{w,s}$ with $w \in V^*$ and $s \in V$ lies in Ψ , and where E is a set of Σ -equations. The operations in $\Phi - \Psi$ are called **behavioral**. \square*

Behavioral specifications are like hidden specifications, but without condition (S2) of Definition 2, and with a fixed hidden subsignature Φ which is used to define behavioral equivalence, as in the CafeOBJ language [19]. Contexts can be defined for behavioral specifications as in Definition 5 (but they may contain non-monadic operations), and so behavioral equivalence \equiv_{Φ} can be defined as in Definition 24 as equality under all contexts built from the operations in Φ . This is used in the following

Definition 47 *A behavioral model of (H, Φ, Σ, E) is a Σ -algebra A such that $A|_{\Psi} = D$ and such that A Φ -behaviorally satisfies each equation in E , where A Φ -behaviorally satisfies an equation of the form*

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

iff for every assignment $\theta : X \rightarrow A$ such that $\theta^(t_i) \equiv_{\Phi} \theta^*(t'_i)$ for $i = 1, \dots, m$ we have $\theta^*(t) \equiv_{\Phi} \theta^*(t')$. \square*

One consequence of using a hidden subsignature Φ to define behavioral equivalence is that equational deduction is no longer sound (unless hidden coinduction or some other method has shown that \equiv_{Φ} is behavioral Σ -equivalence for all models of the behavioral specification). To see this, consider the nondeterministic stacks of Example 17, and suppose that the subsignature Φ contains only the operations **top** and **pop**. This means that two states of a stack are behaviorally Φ -equivalent iff their tops are equal, the tops of their pops are equal, and so on. Any model of the specification will satisfy the equation

`pop push empty = empty`

in all contexts built from `top` and `pop`, but not necessarily in contexts containing `push`. For example, the equation

$$\text{top push pop push empty} = \text{top push empty} \quad (2)$$

follows from the above equation by equational deduction; however, it is not Φ -behaviorally satisfied by all models of the behavioral specification. This is because (2) is just the first equation in the context `top push z`, which is not a Φ -context.

An explicit counterexample is given by a model A where A_{stack} is $\omega^* \times \omega$, i.e., pairs consisting of a list of numbers and a number. Then define `top`(l, n) to be the first element of l (and 0 if l is empty), `pop`(l, n) = (l', n), where l' is the tail of l (or the empty list if l is empty), `push`(l, n) = ($l', n + 1$), where l' is l with n added at the start, and `empty` = ($[], 0$). In this model, the left side of equation (2) is 1, while the right side is 0.

In terms of the discussion of nondeterminism in Section 2.4, this model ‘nondeterministically’ pushes a value onto a stack by ‘choosing’ ever larger numbers, by incrementing by one. It is *not* a hidden model in the sense of Definition 5, precisely because it does not satisfy equation (2); however, it is a behavioral model according in the sense of Definition 47. We can see therefore that Roşu and Goguen’s definitions allow a strictly larger class of models, and that these models allow a form of ‘internal’ nondeterministic choice, insofar as they need not satisfy (2). We can describe this equation intuitively as follows: Start with the empty stack, and then nondeterministically push some number onto the stack; call this number n . This gives us the right side of (2). Now pop this number from the stack, and again nondeterministically push some number onto the stack; the equation says that the number chosen must be n . By relaxing the definition of satisfaction of equations such as (2), Definition 47 allows models that are free to pick any number to push onto the stack the second time.

3.9 A Categorical Hat Trick

Those who are antagonistic to and/or ignorant about category theory should skip this subsection. The remaining readers will find here a very brief but elegant proof of Theorem 26. In a sense, this proof generalizes the well known construction of a minimal machine as the quotient of the term algebra by the behavioral equivalence relation (usually called the Nerode equivalence in that context) [78,75].

Given a hidden theory (Σ, E) and an (Σ, E) -algebra A , let $h : A^\circ \rightarrow F_{\Sigma^\circ}$ be the unique homomorphism to the final Σ° -algebra F_{Σ° . We can factor h as $i \circ j$ where j is surjective and i is injective. Now let \mathcal{C} denote the category of all factorizations $f \circ e$ of h where e is surjective. Then the factorization $i \circ j$ is final in \mathcal{C} and therefore has no proper quotients. This is the same as saying that the congruence defined by (the kernel of) j is maximal on A° . Thus we have proved Theorem 26 for the case $\Phi = \Sigma$; but the general case now follows, because any Σ -algebra is also a Φ -algebra.

4 Conclusions and Futures

Although the hidden agenda disclosed here is very ambitious, we hope to have given evidence that it is feasible to meet its goals, and indeed that much of the necessary groundwork has already been done. We admit it is surprising that the hidden approach is both more general and more effective (in regard to proofs) than the traditional process algebra and transition system approaches, but it really does seem that the ‘simplifications’ introduced by these approaches actually make many proofs more difficult. It is perhaps even more surprising that while we initially focused on the object paradigm, we could not avoid the constraint, logic, and concurrent paradigms, nor non-determinism and infinite data values. Moreover, by using the module system of parameterized programming (see Appendix B), we can obtain the power of higher order functional programming in a first order setting. Of course, a great deal of work must still be done to meet the challenge set by the enormous efforts that the more established approaches have already put into exploring applications and developing mechanical support.

We feel that hidden algebra is a natural next step in the evolution of algebraic specification, carrying forward the intentions of its founders in a simple and elegant way to the realities of modern software. Initial algebra semantics remains appropriate for data values, but hidden algebra allows us to also handle systems of objects (abstract machines), concurrency, constraints, streams, existential queries, and more; we wish to further explore this potent combination of paradigms, and apply it to further problems of real practical value.

We are experimenting with ways to organize hidden proofs as active websites, using HTML, Java, JavaScript, etc., produced by a proof assistant cum website generator called Kumo [56,55], which provides direct support for hidden coinduction and automatically generates an entire website for a proof, including executable OBJ3 proof scores, and links to background material and explanation pages. We are considering integrating Kumo with decision methods for special domains beyond canonical term rewriting theories, such as Presburger arithmetic. Another topic is traceability, which is very important when con-

structuring complex new proofs; we will explore use of the TOOR hypermedia tool [84] for this purpose.

We have generally talked as if hidden theories are only used for specification. But theories can be directly executed under certain restrictions on the form of equations, as already implemented in OBJ, Eqlog and FOOPS. Therefore hidden theories can be used directly for small and medium sized applications, as well as for prototyping of large applications. It would be worthwhile developing tool support for this.

This paper restricts attention to hidden many sorted algebra. The extension to hidden order sorted algebra is not really difficult, but it cannot be trivial, because it covers nonterminating systems, partial recursive functions, multiple inheritance, error definition and handling, coercion, overwriting, multiple representation, and more; many details appear in [76], but there is still more work to be done. We also wish to further explore connections with other approaches, including coalgebra and concurrent logic programming.

References

- [1] Samson Abramsky. Interaction categories and communicating sequential processes. In A. William Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 1–15. Prentice Hall, 1994.
- [2] Peter Aczel and Nax Mendler. A final coalgebra theorem. In D.H. Pitt *et al.*, editor, *Category Theory and Computer Science*. Springer, 1989. Lecture Notes in Computer Science, Volume 389.
- [3] Pierre America, Jaco de Bakker, Joost Kok, and Jan Rutten. Denotational semantics of a parallel object-oriented language. *Information and Computation*, 83(2):152–205, 1990.
- [4] Jan Baeten and W.P. Weijland. *Process Algebra*. Cambridge, 1990. Cambridge Tracts in Theoretical Computer Science, Volume 18.
- [5] Michael Barr. Terminal coalgebras in well-founded set theory. *Theoretical Computer Science*, 114:299–315, 1993.
- [6] Michel Bidoit, Rolf Hennicker, and Martin Wirsing. Behavioural and abstractor specifications. *Science of Computer Programming*, vol. 25, no. 2–3, 1995.
- [7] Garrett Birkhoff. On the structure of abstract algebras. *proceedings of the Cambridge Philosophical Society*, 31:433–454, 1935.
- [8] Barry Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.
- [9] Rod Burstall. Programming with modules as typed functional programming. *Proceedings, International Conference on Fifth Generation Computing Systems*, 1985.

- [10] Rod Burstall and Răzvan Diaconescu. Hiding and behaviour: an institutional approach. In A. William Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 75–92. Prentice Hall, 1994.
- [11] Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.
- [12] Graham Button and Wes Sharrock. Occasioned practises in the work of implementing development methodologies. In Marina Jirotko and Joseph Goguen, editors, *Requirements Engineering: Social and Technical Issues*, pages 217–240. Academic, 1994.
- [13] Corina Cîrstea. A Semantic Study of the Object Paradigm. Transfer thesis, Programming Research Group, Oxford University, 1996.
- [14] Corina Cîrstea. Coalgebra semantics for hidden algebra: parameterized objects and inheritance. Paper presented at the 12th Workshop on Algebraic Development Techniques, June 1997.
- [15] 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.
- [16] Răzvan Diaconescu. *Category-based Semantics for Equational and Constraint Logic Programming*. PhD thesis, Programming Research Group, Oxford University, 1994.
- [17] Răzvan Diaconescu. A category-based equational logic semantics to constraint programming. In Magne Haverdaen, Olaf Owe, and Ole-Johan Dahl, editors, *Recent Trends in Data Type Specification*, Lecture Notes in Computer Science, pages 200–222. Springer, 1996.
- [18] Răzvan Diaconescu. Foundations of behavioural specification in rewriting logic. In *Proceedings, First International Workshop on Rewriting Logic and its Applications. Asilomar, California, September 1996*. North-Holland, 1996.
- [19] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.
- [20] 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.
- [21] Hartmut Ehrig, Hans-Jörg Kreowski, Bernd Mahr, and Peter Padawitz. Algebraic implementation of abstract data types. *Theoretical Computer Science*, 20:209–263, 1983.
- [22] Hartmut Ehrig, Fernando Orejas, Felix Cornelius, and Michael Baldamus. Abstract and behaviour module specifications. Technical Report 93–25, Technische Universität Berlin, 1993.

- [23] Samuel Eilenberg and Jesse Wright. Automata in general algebras. *Information and Control*, 11:452–470, 1967.
- [24] Marie-Claude Gaudel and Igor Privara. Context induction: an exercise. Technical Report 687, LRI, Université de Paris-Sud, 1991.
- [25] V. Giarrantana, F. Gimona, and Ugo Montanari. Observability concepts in abstract data specifications. In *Proceedings, Conference on Mathematical Foundations of Computer Science*. Springer-Verlag, 1976. Lecture Notes in Computer Science, Volume 45.
- [26] Joseph Goguen. Semantics of computation. In Ernest G. Manes, editor, *Category Theory Applied to Computation and Control*, pages 151–163. Springer, 1975. Lecture Notes in Computer Science, Volume 25; also in Proceedings of Symposium at University of Massachusetts at Amherst, 1974, pages 234–249.
- [27] Joseph Goguen. Parameterized programming. *Transactions on Software Engineering*, SE-10(5):528–543, September 1984.
- [28] Joseph Goguen. Suggestions for using and organizing libraries in software development. In Steven Kartashev and Svetlana Kartashev, editors, *Proceedings, First International Conference on Supercomputing Systems*, pages 349–360. IEEE Computer Society, 1985. Also in *Supercomputing Systems*, Steven and Svetlana Kartashev, Eds., Elsevier, 1986.
- [29] 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.
- [30] Joseph Goguen. Higher-order functions considered unnecessary for higher-order programming. In David Turner, editor, *Research Topics in Functional Programming*, pages 309–352. Addison Wesley, 1990.
- [31] 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.
- [32] 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.
- [33] Joseph Goguen. An approach to situated adaptive software. In *Proceedings, International Workshop on New Models of Software Architecture*, pages 7–20. NEDO, 1993.
- [34] Joseph Goguen. Requirements engineering as the reconciliation of social and technical issues. In Marina Jirotko and Joseph Goguen, editors, *Requirements Engineering: Social and Technical Issues*, pages 165–200. Academic Press, 1994.

- [35] Joseph Goguen. Parameterized programming and software architecture. In *Proceedings, Reuse '96*, pages 2–11. IEEE Computer Society, April 1996. Invited keynote address.
- [36] Joseph Goguen. Stretching first order equational logic: Proofs with partiality, subtypes and retracts. In Maria Paola Bonacina and Ulrich Furbach, editors, *Proceedings, Workshop on First Order Theorem Proving*, pages 78–85. Johannes Kepler Univ. Linz, 1997. Schloss Hagenberg, Austria, October 1997; RISC-Linz Report No. 95–50; revised version to appear in *Journal of Symbolic Computation*.
- [37] Joseph Goguen. *Theorem Proving and Algebra*. MIT, to appear.
- [38] Joseph Goguen and Rod Burstall. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical Report CSL–118, SRI Computer Science Lab, October 1980.
- [39] 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.
- [40] Joseph Goguen and Răzvan Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4:363–392, 1994.
- [41] Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Hartmut Ehrig and Fernando Orejas, editors, *Proceedings, Tenth Workshop on Abstract Data Types*, pages 1–29. Springer, 1994. Lecture Notes in Computer Science, Volume 785.
- [42] Joseph Goguen and Grant Malcolm. Proof of correctness of object representation. In A. William Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 119–142. Prentice Hall, 1994.
- [43] Joseph Goguen and Grant Malcolm. Situated adaptive software: beyond the object paradigm. In *Proceedings, International Symposium on New Models of Software Architecture*, pages 126–142. Information-Technology Promotion Agency, 1995.
- [44] Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. MIT, 1996.
- [45] Joseph Goguen and Grant Malcolm. Hidden Coinduction: behavioral correctness proofs for objects. To appear in *Mathematical Structures in Computer Science*, 1999.
- [46] Joseph Goguen and Grant Malcolm. More higher order programming in OBJ3. In Joseph Goguen and Grant Malcolm, editors, *Algebraic Specification with OBJ: An Introduction with Case Studies*. To appear.
- [47] Joseph Goguen, Grant Malcolm, and Tom Kemp. A hidden Herbrand theorem. In C. Palamidessi, H. Glaser and K. Meinke, editors, *Principles of Declarative Programming*, pages 445–462. Springer, 1998. Lecture Notes in Computer Science, Volume 1490.

- [48] 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.
- [49] Joseph Goguen and José Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11(3):307–334, 1985.
- [50] Joseph Goguen and José Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In Douglas DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 295–363. Prentice Hall, 1986. An earlier version appears in *Journal of Logic Programming*, Volume 1, Number 2, pages 179–210, September 1984.
- [51] Joseph Goguen and José Meseguer. Order-sorted algebra solves the constructor selector, multiple representation and coercion problems. In *Proceedings, Second Symposium on Logic in Computer Science*, pages 18–29. IEEE Computer Society, 1987.
- [52] 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.
- [53] 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. Drafts exists from as early as 1985.
- [54] Joseph Goguen, José Meseguer, and David Plaisted. Programming with parameterized abstract objects in OBJ. In Domenico Ferrari, Mario Bolognani, and Joseph Goguen, editors, *Theory and Practice of Software Technology*, pages 163–193. North-Holland, 1983.
- [55] Joseph Goguen, Kai Lin, Akira Mori, Grigore Roşu, and Akiyoshi Sato. Distributed cooperative formal methods tools. In Michael Lowry, editor, *Proceedings, Automated Software Engineering*, pages 55–62. IEEE, 1997.
- [56] Joseph Goguen and Akira Mori. Semiotics, ProofWebs, and Distributed Cooperative Proving. To appear in *Proceedings, User Interfaces for Theorem Provers*, Sophie-Antipolis, France, 1997.
- [57] Joseph Goguen and Adolfo Socorro. Module composition and system design for the object paradigm. *Journal of Object Oriented Programming*, 7(9):47–55, February 1995.
- [58] Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Raymond Yeh, editor, *Current Trends in Programming Methodology, IV*, pages 80–149. Prentice Hall, 1978.

- [59] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95, January 1977.
- [60] Joseph Goguen and Will Tracz. An implementation-oriented semantics for module composition, 1997. Submitted for publication.
- [61] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Algebraic Specification with OBJ: An Introduction with Case Studies*. To appear. Also Technical Report, SRI International.
- [62] Robert Goldblatt. *Logics of Time and Computation* (2nd edition), CSLI Lecture Notes Number 7, Center for Studies in Language and Information, Stanford University, 1992.
- [63] Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Electronic Notes in Theoretical Computer Science*, 1, 1995.
- [64] John Guttag. *The Specification and Application to Programming of Abstract Data Types*. PhD thesis, University of Toronto, 1975. Computer Science Department, Report CSRG–59.
- [65] John Guttag. Abstract data types and the development of data structures. *Communications of the Association for Computing Machinery*, 20:297–404, June 1977.
- [66] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In D.H. Pitt, A. Poigne and D.E. Rydeheard, editors, *Category Theory and Computer Science*, pages 140–157. Lecture Notes in Computer Science, Volume 283. Springer, 1988.
- [67] Lutz Hamel. *Behavioural Verification and Implementation of an Optimizing Compiler for OBJ3*. PhD thesis, Oxford University Computing Lab, 1996.
- [68] Rolf Hennicker. Context induction: a proof principle for behavioural abstractions. In A. Miola, editor, *Proceedings, International Symposium on the Design and Implementation of Symbolic Computation Systems*, volume 429 of *Lecture Notes in Computer Science*, pages 101–110. Springer, 1990.
- [69] C.A.R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.
- [70] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [71] Bart Jacobs. Mongruences and cofree coalgebras. In Maurice Nivat, editor, *Algebraic Methodology and Software Technology (AMAST95)*, pages 245–260. Springer, 1995. Lecture Notes in Computer Science, Volume 936.
- [72] Bart Jacobs. Objects and classes, coalgebraically. In B. Freitag, Cliff Jones, C. Lengauer and H.-J. Schek, editors, *Object-Oriented with Parallelism and Persistence*, pages 83–103. Kluwer, 1996.

- [73] Bart Jacobs. Invariants, bisimulations and the correctness of coalgebraic refinements. Technical Report CSI-R9704, Computer Science Institute, University of Nijmegen, March 1997.
- [74] Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14, 1990.
- [75] Grant Malcolm. Behavioural equivalence, bisimilarity, and minimal realisation. In Magne Haveraaen, Olaf Owe and Ole-Johan Dahl, editors, *Recent Trends in Data Type Specifications*. Springer, pages 359–378, 1996. Lecture Notes in Computer Science, Volume 1130.
- [76] Grant Malcolm and Joseph Goguen. Proving correctness of refinement and implementation. Technical Monograph PRG-114, Programming Research Group, University of Oxford, 1994. Submitted for publication.
- [77] José Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
- [78] 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.
- [79] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Parts I and II. Technical Report ECS-LFCS-89-85 and -86, Computer Science Department, University of Edinburgh, 1989.
- [80] Edward F. Moore. Gedanken-experiments on sequential machines. In *Automata Studies*, pages 129–153. Princeton, 1956.
- [81] Lawrence Moss, José Meseguer, and Joseph Goguen. Final algebras, cosemicomputable algebras, and degrees of unsolvability. *Theoretical Computer Science*, 100:267–302, 1992. Original version from March 1987.
- [82] Fernando Orejas, Marisa Navarro, and Ana Sánchez. Algebraic implementation of abstract data types: a survey of concepts and new compositionality results. *Mathematical Structures in Computer Science*, 6(1), 1996.
- [83] David Parnas. Information distribution aspects of design methodology. *Information Processing '72*, 71:339–344, 1972. Proceedings of 1972 IFIP Congress.
- [84] Francisco Pinheiro and Joseph Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, pages 52–64, March 1996. Special issue of papers from ICRE '96.
- [85] Horst Reichel. Behavioural equivalence – a unifying concept for initial and final specifications. In *Proceedings, Third Hungarian Computer Science Conference*. Akademiai Kiado, 1981. Budapest.
- [86] 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.

- [87] Horst Reichel. An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science*, 5:129–152, 1995.
- [88] Grigore Roşu. A birkhoff-like axiomatizability result for hidden algebra and coalgebra. In *Proceedings of the First Workshop on Coalgebraic Methods in Computer Science (CMCS'98), Lisbon, Portugal, March 1998*, volume 11 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1998.
- [89] Grigore Roşu and Joseph Goguen. Hidden congruent deduction. Ricardo Caferra and Gernot Salzer, editors. In *Proceedings, International Workshop on First Order Theorem Proving*, Technische Universität Wien, pages 213–223, 1998. Schloss Wilhelminenberg, Vienna, November 23-25, 1998. Also, to appear in *Lecture Notes in Artificial Intelligence*, Springer, 1999.
- [90] Jan Rutten. Universal coalgebra: a theory of systems. Technical Report CS-R9652, CWI, 1996.
- [91] Jan Rutten and Daniele Turi. Initial algebra and final coalgebra semantics for concurrency. In Jaco de Bakker, Jan Willem de Roever, and Gregorz Rozenberg, editors, *Proc. REX Symposium 'A Decade of Concurrency'*, pages 530–582. Springer, 1994. Lecture Notes in Computer Science, Volume 803.
- [92] Donald Sannella and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications. *Acta Informatica*, 25:233–281, 1988.
- [93] Adolfo Socorro. *Design, Implementation, and Evaluation of a Declarative Object Oriented Language*. PhD thesis, Programming Research Group, Oxford University, 1994.
- [94] J. Michael Spivey. *Understanding Z*. Cambridge, 1988.
- [95] Will Tracz. Parameterized programming in LILEANNA. In *Proceedings, Second International Workshop on Software Reuse*, March 1993. Lucca, Italy.
- [96] Daniele Turi and Gordon Plotkin. Towards a mathematical operational semantics. In *Proceedings, Logic in Computer Science*, IEEE Computer Society Press, pages 280–291, 1997.
- [97] Simone Vegliani. *Integrating Static and Dynamic Aspects in the Specification of Open, Object-based and Distributed Systems*. PhD thesis, Oxford University Computing Laboratory, 1997.
- [98] Bartel van der Waerden. *A History of Algebra*. Springer, 1985.
- [99] Mitchell Wand. Final algebra semantics and data type extension. *Journal of Computer and System Sciences*, 19:27–44, 1979.
- [100] Alfred North Whitehead. *A Treatise on Universal Algebra, with Applications, I*. Cambridge, 1898. Reprinted 1960.

A Using OBJ3 for Hidden Algebraic Proofs

OBJ3 does just two kinds of computation: reduction with rewrite rules, and general equational reasoning; these are implemented by its `reduce` and `apply` commands, respectively. When combined with OBJ3's declarative capabilities, these computations support a surprisingly wide range of proof techniques, including reasoning for loose, initial, and hidden semantics, as discussed in the following subsections. Perhaps the easiest way to explain how this works is to explicate the various logical relations involved; we will see that there are many of them. (This discussion requires a slightly more sophisticated understanding of algebra than is assumed in most of the body of this paper.)

A.1 Loose Semantics

Under loose semantics, we have the beautiful situation described by Birkhoff's Theorem [7], that an equation is true for all models iff it is provable by equational deduction. In particular, anything provable by reduction is true of all models. Although the converse does not hold and reduction is incomplete, reduction has the tremendous advantage of being totally automatic. This is important because even fairly simple problems can require a few hundred deductions. Applying a non-rewrite equation requires user control, to choose values for variables and the precise point of application; this can get tedious. Even though OBJ3 does allow sophisticated combinations of application and rewriting (see Section 5.5 of [61]), it is clear that users will always prefer proofs by pure rewriting if they are possible.

Now let's introduce our first relation: let $t = t'$ mean that the equation can be deduced using equational reasoning; more explicitly, we can write $T \vdash (\forall X) t = t'$, where T is an equational theory. Here we are in the domain of general equational reasoning; note that OBJ3 also uses the equality sign to separate the two terms of an equation.

Second, $t \overset{*}{\rightarrow} t'$, or more explicitly $T \vdash (\forall X) t \overset{*}{\rightarrow} t'$, means that t rewrites to t' , where all equations in T must be rewrite rules. When a specification consists of rewrite rules, and when only rewriting is to be used for proofs, it would make sense to use the notation \rightarrow for equations in T instead of $=$, because then $\overset{*}{\rightarrow}$ is the transitive and reflexive closure of our third relation \rightarrow , which indicates one step rewriting. No special properties of T , such as Church-Rosser or termination, are needed for these relations, only that T consists of rewrite rules²³. This is the domain of Meseguer's rewriting logic [77]. In practice, OBJ

²³ OBJ3 is more general than this, because it supports rewriting modulo associativity, commutativity and/or identity.

specifications nearly always consist of rewrite rules, even though this is far from a theoretical necessity. The relation $=$ is the transitive, reflexive, symmetric closure of \rightarrow , and thus goes beyond rewriting logic.

Fourth, let $t == t'$ mean that t, t' both have normal forms under T , and that these forms are identical; if $t == t'$ is true, then t and t' rewrite to the same term, so that an equational proof for $t = t'$ exists; but if $t == t'$ is false, then we don't know whether or not $t = t'$ holds. Note also that $t == t'$ can fail to have a value because of non-termination.

To summarize, $t \rightarrow t'$ implies $t \xrightarrow{*} t'$, which implies $t == t'$ if t' has a normal form, and each of these implies $t = t'$. None of these implications can be reversed.

A.2 Initial Semantics

Under initial semantics, the intended interpretation of a specification T is the class of its initial models. For equational logic, we know that such models always exist, and moreover that they are all isomorphic [58]. Strictly speaking, OBJ3 makes no special computational provision for initial semantics; however it does allow users to declare their intention that certain specifications should be interpreted initially instead of loosely, by using the keyword pair `obj . . . endo` instead of the pair `th . . . endth`. Although it would make no computational difference at all if this convention were reversed, certain computations have a different significance under initial semantics than they would under loose semantics. In particular, if T is canonical, i.e., both Church-Rosser and terminating, then for t, t' ground terms, $t == t'$ iff $t = t'$ holds in the initial algebra; that is, $==$ gives a decision procedure for initial ground equational satisfaction, and it decides disequality as well as equality. But reduction cannot prove all the equations that are true in initial models; inductive proofs are more powerful, are valid, and are often necessary.

All of the proof methods discussed in the previous subsection for loose semantics are still valid for initial semantics, since they are valid for all models, and so certainly for initial models. Inductive proofs are also valid without any assumptions on the form of equations, although such assumptions may of course facilitate computations for the base and step cases of an induction.

We now introduce some more relations: let $t \cong t'$ mean $D \models (\forall X) t = t'$, where D is an initial T -algebra; we may also write this as $T \models (\forall X) t \cong t'$, or even (as in [37]) $T \models (\forall X) t = t'$. The important fact that reasoning techniques for loose semantics are still valid is expressed by the assertion $t = t'$ implies $t \cong t'$. So in particular, $t == t'$ implies $t \cong t'$, and also of course $t \xrightarrow{*} t'$ implies $t \cong t'$. But more than this, we can use ‘inductive lemmas’ in these loose deductions;

that is, if we have previously proved $t_1 \cong t_2$ using induction, and if $t_1 = t_2$ is a rewrite rule, then we can add $t_1 \rightarrow t_2$ to T in computing $t == t'$. Let us (somewhat informally, since it doesn't indicate exactly what set T' of rewrite rules is involved, except that $T \subseteq T' \subseteq Th(D)$, where $Th(D)$ denotes the set of all equations true of D) write $t \cong \cong t'$ for proofs done this way; and similarly, let us write $t \overset{*}{\rightsquigarrow} t'$ for proofs by reduction that may use inductive lemmas. Then we have $t \overset{*}{\rightsquigarrow} t'$ implies $t \cong \cong t'$ if t' has a normal form, and $t \overset{*}{\rightsquigarrow} t'$ and $t \cong \cong t'$ both imply $t \cong t'$. These relations represent the most convenient way to do many inductive arguments.

A.3 Hidden Semantics

A hidden specification is a loose protecting extension (because \models is defined in terms of \models) of an initially interpreted subtheory; let T denote the entire theory and let T_D denote its initial subtheory. Under this definition, hidden models are technically (loose) models of T such that when restricted to the signature Σ_D of T_D they are an initial model of T_D ; but because all initial models are isomorphic, there is no loss of generality if we assume, as in the body of this paper, that the restriction to Σ_D is some particular initial model D of T_D .

Now more relations: behavioral equality, $t \equiv t'$, is as in Definition 5; we may also write $T \models (\forall X) t \equiv t'$, or even $T \models (\forall X) t = t'$. Once again, it is important to note that loose reasoning is valid; in other words $t = t'$ implies $t \equiv t'$. Therefore $t == t'$ implies $t \equiv t'$, and $t \overset{*}{\rightarrow} t'$ implies $t \equiv t'$. Moreover, we can use 'behavioral lemmas' about hidden sorts and inductive lemmas about visible sorts in such loose deductions: if we have previously shown $t_1 \equiv t_2$ or $t_1 \cong t_2$, and if $t_1 = t_2$ is a rewrite rule, then we can add $t_1 \rightarrow t_2$ to T in computing $t \equiv t'$. That inductive assertions about the data subtheory can be used in behavioral proofs is expressed by saying $t \cong t'$ implies $t \equiv t'$, provided t, t' are of visible sort whenever we write $t \cong t'$. In fact, inductive lemmas about data are often needed in behavioral proofs. Let us (again somewhat informally, since it doesn't say exactly what rewrite rules are involved) indicate this kind of deduction by writing $t \equiv \equiv t'$, and let us similarly write $t \overset{*}{\Rightarrow} t'$ for proofs by reduction that may use behavioral and inductive lemmas. Then $t \overset{*}{\Rightarrow} t'$ implies $t \equiv \equiv t'$; these relations often represent the most convenient way to carry out behavioral proofs, often as part of some coinduction.

The official OBJ3 syntax does not allow declaring some sorts to be hidden. Although it would be easy to modify the syntax, there is no compelling reason to do so, because (1) just as with the distinction between `obj . . . endo` and `th . . . endth`, it would have no computational effect, and (2) it is easy to introduce a notational convention that serves the same purpose, which after all

is just to declare user intentions. In fact, we did introduce such a convention: that all new sorts declared in theory modules containing the line `pr DATA` are hidden, and all the sorts declared in `DATA` are visible.

A.4 Discussion

In doing coinductive proofs within the hidden conventions suggested above, there are just two kinds of sort, hidden and visible; therefore the two kinds of computation that OBJ3 provides (rewriting and general equational reasoning) are always valid, because they are valid for each kind of sort separately. However, because induction is only valid for visible (data) sorts and coinduction is only valid for hidden sorts, some mental discipline is necessary in using OBJ3 this way; it is necessary to keep track of the significance of computations based on conventions that do not actually effect the outcome of the computations. Under this discipline, many assertions that would otherwise have to be made outside OBJ can be seen as assertions within OBJ. The fact that nearly all the relations that we have so carefully distinguished in our theoretical discussion above, are denoted by the same symbol in OBJ3 syntax, namely `=`, makes more sense than seems reasonable at first, because of its convenience, and because of the inclusions among these relations discussed above.

Turning a bit towards philosophy, we note that a mechanical theorem prover does not know what it is doing (nor does any program, nor any computer); the meaning of a computation can only be supplied by a user, based on information about its context.

B Modularity, Inheritance and Multiple Paradigms

Our approach to modularity assumes that signatures, models and axioms satisfy certain natural conditions that define a so-called *institution*²⁴ [39,20,60]. This gives the powerful module facilities of *parameterized programming* [29,31], including generic modules that take other modules as parameters that may themselves be parameterized, module expressions that say how to interconnect and/or modify²⁵ modules taking proper account of their parameterization and of modules that they inherit, and much more [29].

²⁴ For this, we must use (hidden) signature morphisms [41], rather than (hidden) signature maps (see Definition 44); this is appropriate because signature morphisms are used for horizontal structure, which is what modularity provides, whereas signature maps are used for vertical structure ([38] introduced this distinction between vertical and horizontal structure).

²⁵ As does the phrase `*(sort Arr to State)` in Section 3.6.

An analogy with functional programming may help [9]: modules are like functions, with theories as their types; then evaluating a well-typed module expression yields a new module, just as evaluating a well-typed functional expression yields a new function. In fact, parameterized programming gives the power of higher order functional programming in a purely first order setting, which makes it easier to do proofs and to write programs [30].

Parameterized programming also allows integrating specifications with executable code, where the former is in a specification language and the latter is in a conventional programming language²⁶. In this case, executing a module expression builds new executable code with its specification; the code may be in both textual and compiled executable forms. This is implemented in LILEANNA [60], a module interconnection language for Ada, which extends parameterized programming with a information hiding like that in hidden algebra. LILEANNA [28,95,60] has also shown that this can have practical benefit for real problems (flight control software for helicopters). Parameterized programming was first implemented in OBJ [54,27,61], and builds on ideas from Clear [11]; it has influenced the module systems of the Ada, C++ and ML languages. Recent work on parameterized programming shows how it can be used for software architecture [35], and how it extends to higher order modules and views [46].

Hidden algebra handles the main features of the object paradigm, and even seems to clarify and improve things a bit, e.g., by cleanly separating data from objects, and classes from modules; we have also shown how to add powerful generic modules, behavioral types for specification, and a declarative programming style. These ideas have been implemented in the FOOPS language [52,57,93] and been shown useful, e.g., for implementing a multimedia requirements tracing system [84]. We will not discuss our approach to inheritance here, because it uses order sorted algebra, but it is natural and simple, with subclasses just subsorts having more attributes and methods [41]. The integration of functional and logic paradigms in the Eqlog language [50,16] has been extended to integrate the object, logic and functional paradigms, while adding the constraint paradigm [47].

²⁶ The idea of associating code, specifications, and possibly other software objects, such as test cases and documentation, comes from *hyperprogramming* [31].