Specifying, Programming and Verifying with Equational Logic

Joseph Goguen and Kai Lin
Dept. Computer Science & Engineering
University of California at San Diego
9500 Gilman Drive, La Jolla, CA 92093-0114, USA

Abstract In a logical programming language, a program is a theory over a formal logic, and its computation is deduction in that theory. As a consequence, specification, programming and verification all fit a single unified framework. The OBJ languages are logical programming languages, based on various extensions of first order equational logic. Everything in these languages is logic-based, including their powerful module systems, which provide a convenient language for software architecture. A new feature introduced in this paper is mutual coinduction, illustrated with inductive proof schemes. Two theoretical contributions are a new formalization of logical programming, and a new semantics for higher order modules.

1 Introduction

Programming is difficult, as shown by the fact that debugging a program usually takes more time than creating it; moreover, the difficulty of debugging increases non-linearly with program size. One reason for such phenomena is the astonishing complexity and subtlety of the semantics of most widely used programming languages, due mainly to the desire for high efficiency on conventional processors. But rapid increases in the power and flexibility of hardware, and in the need for greater reliability and security in applications, suggest that it may be valuable to consider alternative approaches, based on higher level languages with much simpler semantics, despite the undoubted inertia of tradition, and the difficulty of learning new languages and new paradigms.

This paper focuses on the OBJ family of languages, which have semantics based on various extensions of (first order) equational logic. The OBJ languages are **logical programming languages**, in which programs are theories, and computation is deduction, which makes it possible to do specification, programming and verification in a unified framework. This paper is mainly intended to introduce and motivate the material that it covers, rather than to provide a thorough mathematical exposition. Consequently, there are many references and several examples, but all proofs and many technical details are omitted.

Equational logic of course cannot do everything, but when it is applicable, it has some significant advantages resulting from its simplicity, including ease of learning and use, and the decidability of problems that are intractable in more complex logics; moreover, algorithms for these problems are often quite efficient, e.g., term rewriting, unification, narrowing, and Knuth-Bendix completion. Unsorted unconditional equational logic, which goes back to Whitehead [73], and later Birkhoff [5], is not expressive enough for most computer science applications, and has therefore been extended in many ways. The simplest extensions are to many sorted and conditional equational logic [41], which provide additional expressive power and support strong type checking. Two further extensions, to overloaded operations and subtypes, yield **order sorted equational logic** [43], which supports partial operations, exception handling, type conversion, multiple representations, and more [57]; the core members of the OBJ family are all based on this, including OBJ2, OBJ3 [48], CafeOBJ [15], and most recently, BOBJ [45, 34] which extends OBJ3 with hidden equational logic and higher order modules.

Two other ways to extend equational logic are partial algebra [11], which is used in CASL [61], and membership equational logic [56], which is used in Maude [13, 12]. A more radical extension introduces behavioral abstraction, which permits specifying systems with states, infinite data structures (such as streams), non-determinism, and concurrency, and which also allows verifying much more sophisticated properties of systems; our version of this is called **hidden algebra** [30, 39, 44, 66]. Behavioral logic is a diverse research area, including not just hidden algebra, but also the coherent hidden algebra of Diaconescu [16, 15], the observational logic of Bidoit and Hennicker [51, 4], and coalgebra, e.g., see the survey [53], and for recent results, [19]. These approaches fall into two broad categories, depending on whether or not a fixed data algebra is assumed for all models. [35, 68]; further details are given later.

Perhaps the two most important innovations are the module system which appears in all current OBJ family members, and the C4RW coinduction algorithm of BOBJ [35, 36, 45, 34]. OBJ3 and BOBJ have higher order module systems, though BOBJ goes further, and has also extended its coinduction algorithm to handle mutual coinduction; therefore this paper is somewhat focused on these two recent contributions, for which it provides an introduction and motivation in the next two subsections. It is noteworthy that together they give a useful platform for specifying and verifying complex systems, such as communication protocols. Section 2.5 discusses logical programming.

Dedication: This paper is dedicated, with respect and affection, to Prof. Dov Gabbay, whose vision of what logic could and should be has been an inspiration for many of us to pursue our dreams.

Acknowledgements: The work surveyed in this paper is the result of dedicated effort by many people over a long period of time, as citations in this paper make clear, and the new work reported here would have been impossible without their important contributions. Although it is infeasible to thank everyone, we would like to particularly thank Rod Burstall, Răzvan Diaconescu, Grant Malcolm, José Meseguer, and Grigore Roşu for their very valuable collaborations, and Till Mossakowski and Andrzej Tarlecki for their valuable comments on the new ideas in this paper.

1.1 Modularization

Modularization controls the complexity of large systems by composing them from parts; this eases both initial construction and later modification by making large grain structure explicit, and it also considerably facilitates reuse. Early designs for OBJ [26] called for a module system like that of the Clear specification language [6]. This approach was later improved and formalized as **parameterized programming**, which provides parameterized modules and (so called) views among its "first class citizens," where the latter say how to fit the syntax of a formal parameter to an actual parameter, including defaults when there is only one obvious choice; moreover, views can be parameterized, and **module expressions** compose modules, and in particular, can describe software architectures. Parameterized programming was first fully implemented in OBJ3 [48], following partial implementations in earlier versions of OBJ, and it appears in all current members of the OBJ family, including CafeOBJ [15], Maude [12], the European languages CASL [61] and ACTTWO [18], and of course BOBJ. Other languages that have been influenced by parameterized programming include Ada, ML, C++, and Modula, none of which has views, so that the syntax of an actual parameter module must contain the syntax of its formal parameter. Ada does not even allow instantiated parameterized modules to be used as actual parameters. ML [59] comes closest to fully implementing parameterized programming, but it lacks views, and in particular, it lacks the convenience of default views.

The Clear module system [6] has semantics based on the category of theories, where views are given by theory morphisms and module composition is given by colimit, inspired by an earlier category theoretic approach to general systems [25]. This semantics also applies to the OBJ family, and is here extended to higher order parameterized programming in Section 3.3.2, and illustrated with an inductive proof scheme.

It should not be thought that parameterized programming is limited to functional languages, let alone to algebraic specification languages; it can be implemented for almost any language, e.g., using techniques suggested for the LIL [27] extension of Ada, and implemented in Lileanna [71], which involve translating to intermediate compiled code (Dianna in the case of Ada) and then applying the compiler's backend optimization. Similar techniques can be used for the higher order case.

1.2 Behavioral Specification and Verification

The basic idea of **behavioral abstraction** is that an equation (or other axiom) need not actually be satisfied by all interpretations into a model of its free (or universally quantified) variables, but need only *appear* to do so with respect to a given set of "experiments," which consist of applying a sequence of state changing operations, and then one state observing operation. This idea, first suggested by Horst Reichel [64, 65], underlies the hidden algebra approach to behavioral specification and verification that is implemented in BOBJ and CafeOBJ. The weaker notion of satisfaction is important because many clever implementations used in practice only satisfy their specifications in this sense.

A **behavioral specification** consists of a signature Σ in which some operations are declared behavioral (this is defined in Section 2.4), a set of equations, some of which are behavioral, and a subsignature of Σ, called a **cobasis**, the operations in which can be used in experiments. Then **behavioral verification** attempts to determine if a given equation is behaviorally satisfied by all models that behaviorally satisfy the given specification. In contrast to the corresponding problem for ordinary satisfaction, there cannot exist any finite complete set of rules of deduction for behavioral satisfaction [8]. Nevertheless, there is an algorithm that is surprisingly useful in practice, as we will see. Moving from ordinary to behavioral logics allows much more natural treatment of systems with internal states, and also supports concurrency and non-determinism, which are essential for modern network based systems. Moreover, since the BOBJ implementation builds on order sorted algebra, classes, subclasses (inheritance), overloading, exceptions, and abstract data types, are also supported, making this approach suitable for non-trivial software engineering applications.

Although simulated computation with behavioral specifications is not feasible, specifying and verifying high level designs seems more useful in practice, because the bugs that are most difficult to find and correct typically arise at the design level; in fact, we argue that verification is a proper generalization of programming for the specification level. CafeOBJ [15] and Spike [3] also implement behavioral specification and verification, but circular coinductive rewriting is only implemented in BOBJ. Behavioral equivalence generalizes the notion of bisimilarity used in process algebra, for which there is a very large literature, including proof methods that are special cases of coinduction; we just mention Milner's CCS [58] and [62], where the notion of bisimilarity seems to have originated. Hidden algebra generalizes process algebra and transition system to include non-monadic parameterized methods and attributes, which can sometimes dramatically simplify verification.

2 Basic OBJ Features

OBJ began around 1974 as a notation for algebraic specification, and soon after was implemented in the OBJ0 system based on term rewriting; the first publication on OBJ [26] included early versions of parameterized programming and order sorted algebra. Subsequent implementations include OBJT [46], OBJ1 [40], OBJ2 [21], OBJ3 [48], and most recently, BOBJ [45, 34], which is highly portable since implemented in Java, and is used for the examples in this paper. CafeOBJ [15] and Maude [12] also share many basic design features. Readers already familiar with OBJ or equational programming may wish to skip Sections 2.1 to 2.3, but Section 2.4 introduces behavioral semantics, and Section 2.5 contains a new theory of logical programming.

2.1 Loose Semantics

The simplest semantics for used for OBJ is loose semantics, in which a theory specifies the variety of all algebras that satisfy its axioms. We begin with some basic concepts, notation and terminology from. Given a set S, an S-**sorted set** A is a family of sets A_s, one for each $s \in S$. The elements of S are called **sorts** and the notation $\{A_s \mid s \in S\}$ is used. A **signature** Σ is an $(S^* \times S)$-sorted set $\{\Sigma_{w,s} \mid \langle w, s\rangle \in S\}$. The elements of $\Sigma_{w,s}$ are called operation (or function) symbols of **arity** w, **sort** s, and **type** $\langle w, s\rangle$; in particular, $\sigma \in \Sigma_{[],s}$ is a constant symbol ($[]$ denotes the empty string). If σ has the type $\langle w, s\rangle$, we write $\sigma \colon w \to s$, and constants are written $c\colon \to s$ when $c \in \Sigma_{[],s}$.

Signatures are given in BOBJ by giving sorts after the keywords `sort` or `sorts`, and operations after the keywords `op` or `ops`. The **form** of an operation follows the `op` keyword, then a colon followed by a list of the sorts for arguments to that operation, followed by an arrow, followed by the value sort of the operation. Underbar characters serve as place holders within the form, to indicate where the arguments should go; the number of underbars and argument sorts should be the same. If there are no underbars but the argument sort list is non-empty, as with the `insert` operation below, the operation is assumed to have syntax that requires opening and closing parentheses, with commas between arguments, as in `insert(2, S)`.

```
sorts Elt Set .
op empty : -> Set .
op _in_ : Elt Set -> Bool  .
op insert : Elt Set -> Set .
```

Overloading is possible (and very helpful for readability) in this framework, since the same form can have more than one type. For example, the form `_in_` could also be an operation on lists, with type $\langle$`List List`$,$`Bool`$\rangle$, where `Bool` is the sort of Booleans, from the builtin module `BOOL`, which is imported by default into every other module. Σ is a **ground signature** iff $\Sigma_{[],s} \cap \Sigma_{[],s'} = \emptyset$ whenever $s \neq s'$ and $\Sigma_{w,s} = \emptyset$ unless $w = []$. Union is defined componentwise, by $(\Sigma \cup \Sigma')_{w,s} = \Sigma_{w,s} \cup \Sigma'_{w,s}$. A common case is union with a ground signature X, where we use the notation $\Sigma(X)$ for $\Sigma \cup X$.

A Σ-**algebra** A consists of an S-sorted set also denoted A, plus an **interpretation** of Σ in A, which is a family of arrows $i_{s_1...s_n,s}\colon \Sigma_{s_1...s_n,s} \to [A_{s_1} \times ... \times A_{s_n} \to A_s]$ for each type $\langle s_1...s_n, s\rangle \in S^* \times S$, which interpret the operation symbols in Σ as actual operations on A. For constant symbols, the interpretation is given by $i_{[],s}\colon \Sigma_{[],s} \to A_s$. Usually we write just σ for $i_{w,s}(\sigma)$, but if we need to make the dependence on A explicit, we may write σ_A. A_s is called the **carrier** of A of sort s. Given

Σ-algebras A and B, a Σ-**homomorphism** $h\colon A \to B$ is an S-sorted arrow $h\colon A \to B$ such that $h_s(\sigma_A(m_1, ..., m_n) = \sigma_B(h_{s_1}(m_1), ..., h_{s_n}(m_n))$ for each $\sigma \in \Sigma_{s_1...s_n,s}$ and all $m_i \in A_{s_i}$ for $i = 1, ..., n$, and such that $h_s(c_A) = c_B$ for each constant symbol $c \in \Sigma_{[],s}$.

A Σ-**congruence relation** $\sim$ on a Σ-algebra A is a S-indexed equivalence relation such that if $\sigma\colon s_1...s_n \to s$ and $a_i, b_i \in A_{s_i}$ with $a_i \sim b_i$ for $1 \leq i \leq n$, then $\sigma(a_1, ..., a_n) \sim \sigma(b_1, ..., b_n)$. Given a Σ-congruence relation $\sim$ on A, the **quotient Σ-algebra** $A/{\sim}$ is a Σ-algebra $A/{\sim}$ such that $(A/{\sim})_s$ is $A_s/{\sim_s}$ for any sort s and $\sigma([a_1], ..., [a_n]) = [\sigma(a_1, ..., a_n)]$ for any $a_i \in A_{s_i}$ and $1 \leq i \leq n$ and $\sigma\colon s_1...s_n \to s \in \Sigma$.

Given an S-sorted signature Σ, the S-sorted set T_Σ of Σ-**terms** is the smallest S-sorted set such that $\Sigma_{[],s} \subseteq T_{\Sigma,s}$ and given $\sigma \in \Sigma_{s_1...s_n,s}$ and $t_i \in T_{\Sigma,s_i}$ then $\sigma(t_1...t_n) \in T_{\Sigma,s}$. Notice that T_Σ is a Σ-algebra by interpreting $\sigma \in \Sigma_{[],s}$ as just σ, and $\sigma \in \Sigma_{s_1...s_n,s}$ as the operation sending $t_1, ..., t_n$ to the list $\sigma(t_1...t_n)$. Thus, T_Σ is called the Σ-**term algebra**. Note that because of overloading, terms do not always have a unique parse. Below is the key property of this algebra; proofs are generally omitted, but can be found in the literature.

Theorem 1 Given a signature Σ with no overloaded constants and a Σ-algebra A, there is a unique Σ-homomorphism $T_\Sigma \to A$. $\Box$

Given a signature Σ and a ground signature X disjoint from Σ, we can form the $\Sigma(X)$-algebra $T_{\Sigma(X)}$ and then view it as a Σ-algebra by forgetting the names of the new constants in X; let us denote this Σ-algebra by $T_{\Sigma(X)}$. It has the following universal **freeness** property:

Theorem 2 Given a Σ-algebra A and $a\colon X \to A$, there is a unique Σ-homomorphism $a\colon T_{\Sigma(X)} \to A$ extending a, in the sense that $\overline{a}_s(x) = a_s(x)$ for each $x \in X_s$ and $s \in S$; sometimes we will write just a instead of $\overline{a}$. $\Box$

A Σ-**equation** consists of a ground signature X of variable symbols (disjoint from Σ) plus two $\Sigma(X)$-terms of the same sort $s \in S$; we may write such an equation abstractly in the form $(\forall X)\ t = t'$ and concretely in the form $(\forall x, y, z)\ t = t'$ when $|X| = \{x, y, z\}$ and the sorts of x, y, z can be inferred from their uses in t and in t'. Similarly, a Σ-conditional equation consists of a ground signature X of variable symbols plus a set of pairs of $\Sigma(X)$-terms u_i, u'_i and t, t', each pair of the same sort and $1 \leq i \leq n$, written in the form $(\forall X)\ t = t'$ `if` $u_1 = u'_1, ..., u_n = u'_n$. Hereafter we use the word "equation" for both the conditional and unconditional cases. A **specification** or **theory** P is a pair (Σ, E), consisting of a signature Σ and a set E of Σ-equations.

If a Σ-equation e is $(\forall X)\ t = t'$ `if` $u_1 = u'_1, ..., u_n = u'_n$ and A is a Σ-algebra, we say A **satisfies** this equation, written $A \models_\Sigma e$, iff for any map $\theta\colon X \to A$, if $\theta(u_i) = \theta(u'_i)$ for $1 \leq i \leq n$, then $\theta(t) = \theta(t')$. Given a specification $P = (\Sigma, E)$, $A \models P$ iff $A \models_\Sigma e$ for every $e \in E$. Given a set of Σ-equations E, we define **provability** $\vdash_\Sigma$ for Σ-equations by the following:

1. $E \vdash_\Sigma (\forall X)\, t = t$
2. If $E \vdash_\Sigma (\forall X)\, t = t'$, then $E \vdash_\Sigma (\forall X)\, t' = t$
3. If $E \vdash_\Sigma (\forall X)\, t = t'$ and $E \vdash_\Sigma (\forall X)\, t' = t''$, then $E \vdash_\Sigma (\forall X)\, t = t''$
4. If $(\forall Y)\, t = t'$ `if` $u_1 = u'_1, ..., u_n = u'_n \in E$ and $\theta\colon Y \to T_{\Sigma(X)}$ and $E \vdash_\Sigma (\forall X)\, \theta(u_i) = \theta(u'_i)$ for $1 \leq i \leq n$, then $E \vdash_\Sigma (\forall X)\, \theta(t) = \theta(t')$.
5. If $E \vdash_\Sigma (\forall Y)\, t_i = t'_i$ and $t_i \in T_{\Sigma(X),s_i}$ for $1 \leq i \leq n$ and $\sigma\colon s_1...s_n \to s$, then $E \vdash_\Sigma (\forall X)\, \sigma(t_1, ..., t_n) = \sigma(t'_1, ..., t'_n)$.

Theorem 3 Soundness and Completeness If e is a Σ-equation, $E \models_\Sigma e$ iff $E \vdash_\Sigma e$. $\Box$

Example 1 A simple example of loose semantics is the theory of groups:

```
th GROUP is sort Elt .
  op e : -> Elt .
  op _-1 : Elt -> Elt [prec 5].
  op _*_ : Elt Elt -> Elt .
  vars X Y Z : Elt .
  eq X * e = X .
  eq X * (X -1) = e .
  eq (X * Y)* Z = X *(Y * Z).
end
```

The keyword `th` introduces theories with loose semantics; it is followed by the name of the module, `GROUP`, and closed by the keyword `end`. Declarations for variables and equations have the obvious keywords. The precedence "attribute" `[prec 5]` gives the operation `-1` a tight binding (in OBJ, lower precedence numbers indicate tighter binding). □

Order sorted signatures can also include subsort declarations. We do not develop the theory of order sorted algebra, but do note that all major results generalize, sometimes with minor modifications.

2.2 Term Rewriting

Many simple equations can be proved by **reduction**, also called **term rewriting**, which applies the available equations to a given term, until no equation can be applied. This subsection gives some basic theory.

Given a signature Σ and ground signatures X, Y of **variable symbols** (disjoint from Σ), a **substitution** θ is a S-sorted set $\{\theta_s \colon X_s \to T_{\Sigma,s}(Y)\}$. By Theorem 2, every such θ extends uniquely to a Σ-homomorphism $\overline{\theta} \colon T_\Sigma(X) \to T_\Sigma(Y)$. For any term $t \in T_{\Sigma,s}(X)$, let $\theta(t) = \overline{\theta}_s(t)$. Given a term $p \in T_{\Sigma,s}(X)$ and a term $t \in T_{\Sigma,s}(Y)$, we say p **matches** t if there exists a substitution θ such that $\theta(p)$ is syntactically the same as t.

Given a signature Σ and a ground signature X of variable symbols (disjoint from Σ), a Σ-**rewrite rule** is a pair of terms, written $l \to r$, such that l and r have the same sort and all variables in r also appear in l. A Σ-**rewrite system**, or **term rewriting system**, abbreviated **TRS**, R is a set of Σ-rewrite rules. A term t **rewrites to** a term t' using R, written $t \to_R t'$ or just $t \to t'$, iff there exists a rewrite rule $l \to r$ in R and a substitution θ such that t has a subterm $\theta(l)$ and t' can be obtained from t by replacing $\theta(l)$ with $\theta(r)$; the term $\theta(l)$ is called the **redex** of the rewrite. Let $\to_R^*$ be the reflexive and transitive closure of $\to_R$. R is **confluent**, also called **Church-Rosser**, iff whenever $t \to_R^* t_1$ and $t \to_R^* t_2$, then there exists a term t' such that $t_1 \to_R^* t'$ and $t_2 \to_R^* t'$. R is **terminating** iff there is no infinite rewriting $t_1 \to_R^* t_2 \to_R^* \ldots$. A **normal form** of t under R is a term t' such that t' cannot be written and $t \to_R^* t'$; we may write $[[t]]_R$ for the normal form of t under R. A TRS is **canonical** iff it is confluent and terminating. It can be shown that in a canonical TRS, every Σ-term has a unique normal form, called its **canonical form**. Note that reduction applies to ground terms only, which means that any variables desired should be introduced as new constants[1]. See [2] for a basic survey of one sorted term rewriting.

The OBJ languages use term rewriting to provide an operational semantics, by viewing equations as rewrite rules, i.e., by applying equations in the forward direction. Term rewriting for initial and loose

[1]This is justified by the so called Theorem of Constants, which says $P \models_\Sigma (\forall X)\ \varphi$ iff $P \models_{\Sigma \cup X} \varphi$, where φ is a Σ-sentence

theories is invoked with the command `red`, followed by a term (and a period). For example, given a module `INTSET` for sets of integers,

```
select INTSET .
red 3 in insert(1,insert(2,insert(3,empty))).
```

constructs the set $\{1,2,3\}$ and then tests whether 3 is in it, in the context of the module `INTSET`, which is made the module currently in focus by the `select` command. Here is the output:

```
reduce in INTSET : 3 in insert(1, insert(2, insert(3, empty)))
result Bool: true
rewrite time: 165ms       parse time: 4ms
```

For operations with attributes for associativity, commutativity, or identity, rewriting is done modulo those equations; details can be found in [48, 2] and many other places. The built in module `TRUTH`, which is included in `BOOL` and is by default imported into every other module, provides a polymorphic binary operation `==` which compares the normal forms of its two arguments. For example,

```
red insert(3,insert(3,empty)) == insert(3,empty) .
```

returns `true`, since the two canonical forms are identical; otherwise it returns `false`. If the TRS is canonical, then `true` is returned iff the two terms are provably equal, but if the TRS is non-terminating, reduction may go into an infinite loop, and if the TRS is not confluent, reduction could return `false` when the terms are nonetheless provably equal.

2.3 Initial Semantics

Given a specification (Σ, E), a natural congruence relation $\equiv_E$ can be defined directly from $\vdash_\Sigma$ by $t \equiv_E t'$ iff $E \vdash_\Sigma (\forall\emptyset)\, t = t'$, and we have the following important *initiality* results:

Theorem 4 Given a specification $S = (\Sigma, E)$, for any Σ-algebra A with $A \models S$, there exists a unique Σ-homomorphism from $T_\Sigma/\equiv_E$ to A. Given a set of E of Σ-equations, a Σ-algebra A is initial iff it has no junk (the Σ-homomorphism $T_\Sigma \to A$ is surjective) and no confusion (it satisfies *only* the equations that can be deduced from E). $\Box$

The **initial semantics** of a specification (Σ, E) is the class of its initial algebras. It can be shown that all the initial algebras of a specification are Σ-isomorphic. By Theorem 4, $T_\Sigma/\equiv_E$ is an initial algebra of (Σ, E). Because any element in $T_\Sigma/\equiv_E$ can be generated by operations, induction is valid for proving properties of initial algebras. Generally, more than one induction scheme is valid for a given specification.

Example 2 The module below defines natural numbers in Peano notation with five operations: the constant `0`, the successor function `s`, infix operations `+` and `*`, and `sum`(n) which computes $1+2+...+n$; the keyword `obj` indicates that it has initial semantics:

```
obj NATS is sort Nat .
  op 0 : -> Nat .
  op s_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat [ assoc comm prec 40 ] .
  op _*_ : Nat Nat -> Nat [ assoc comm prec 20 ] .
  op sum : Nat -> Nat .
  vars M N : Nat .
  eq 0 + M = M .
```

```
    eq s M + N = s(M + N) .
    eq 0 * M = 0 .
    eq s M * N = M * N + N .
    eq sum(0) = 0 .
    eq sum(s M) = s M + sum(M) .
  end
```

The operations `+` and `*` are declared associative and commutative and given precedence. The equations define the non-constructor operations recursively over the constructors[2], `0` and `s`, and rewriting proceeds by matching modulo the equations for the attributes. (The `assoc` and `comm` attributes actually do more than the corresponding equations: they enable parsing and pattern matching modulo those equations.) These numbers differ from those of BOBJ's built in module `NAT`, which use Java integers and provides many additional operations. □

Induction is an essential aspect of theorem proving, and is valid for modules with initial semantics (but not loose semantics). Although not directly supported by the OBJ languages, inductive proofs can be still be done by the method of proof scores [22], as illustrated by the following:

Example 3 We prove the formula $1 + 2 + ... + n = n(n + 1)/2$, in the form

$(\forall \texttt{N}: \texttt{Nat})\ \texttt{sum(N)} + \texttt{sum(N)} = \texttt{N} * (\texttt{s N})$.

The first `red` command below checks the base case, the constant `n` stands for the universally quantified variable `N` in the formula, the equation introduces the inductive hypothesis, and the second `red` checks the inductive step. The operation `==` uses rewriting modulo attributes, and returns `true` iff its two arguments reduce to the same thing modulo the given attributes.

```
  open NATS .
    red sum(0) + sum(0) == 0 * (s 0) .
    op n : -> Nat .
    eq sum(n) + sum(n) = n * (s n) .
    red sum(s n) + sum(s n) == (s n) * (s s n) .
  close
```

Since both reductions return `true`, this proof succeeds.

Because this same pattern is followed in many other proofs, encapsulating it in a reusable module would be useful. But induction is second order, so this cannot be done with a first order module system; Section 3.3.1 will show how to do it using BOBJ's higher order modules and views. □

2.4 Hidden Algebra

Behavioral specifications characterize systems by how they behave in response to relevant experiments, rather than how they are implemented. Our hidden algebra formalization of this intuition distinguishes visible from hidden sorts, with equality being strict on visible sorts and behavioral on hidden sorts, in the sense of indistinguishability under experiments; thus hidden sorts are treated as black boxes, the state of which can only be observed and updated by certain specific operations. Therefore behavioral specifications impose fewer constraints on the semantics of modules, as a result of which some inference rules of ordinary equational reasoning are unsound, although a small modification restores soundness;

[2]A subsignature $\Pi \subseteq \Sigma$ is a signature of **constructors** for a theory P iff for every (ground) Σ-term t, there is a Π-term t' such that $P \vdash_\Sigma t = t'$.

another result of this extra freedom is that no finite set of inference rules can be complete for behavioral satisfaction [8]. Context induction [50, 3] and general coinduction [37, 39] are established proof techniques for behavioral properties, but both need creative human intervention. Circular coinduction [68] is a powerful circular coinductive rewriting algorithm implemented in BOBJ by C4RW, and used to automatically proved many behavioral properties [35, 36].

A **hidden signature** Σ is a signature with its sorts partitioned into visible sorts V and hidden sorts H. Operations in Σ with one hidden argument and a visible result may be called **attributes**, and those with one hidden argument and a hidden result called **methods**. A **hidden Σ-algebra** is just a Σ-algebra; elements of visible sort in a hidden Σ-algebra represent *data*, and those of hidden sorts represent *states*; the subalgebra of visible sorts and operations of visible sort is called the **data algebra**. A **behavioral specification** or **theory** is a triple (Σ, Γ, E) where Σ is a hidden signature, and Γ is a hidden subsignature of Σ, and E is a (finite) set of Σ-equations. The operations in Γ are called **behavioral**.

The definition of hidden algebra given above allows a loose interpretation for the data algebra, following the general approach of [35, 68]. However, this is not appropriate for some problems, for example if `true` and `false` become identified. This can be remedied by requiring every hidden algebra over a given signature to have a *fixed* data algebra, as in the original version of hidden algebra, or alternatively, by allowing so called data constraints, in the sense of [31], as additional sentences. Note that general results proved for the loose data approach will also apply to fixed data algebras, so there is no loss of generality in proceeding in this way.

Given a hidden signature Γ, a **Γ-context**, denoted $C[\Box]$, for sort s is a Γ-term in $T_\Gamma(\{\Box\} \cup Z)$ having exactly one special variable $\Box$ of the sort s, where Z is an infinite set of special variables different from $\Box$. If $C[\Box]$ is a Γ-context of sort s and $t \in \Sigma_s$, let $C[t]$ denote the result of substituting t for $\Box$. A Γ-context $C[\Box]$ for hidden sort s is called **Γ-experiment** if its sort is visible.

If Γ is a subsignature of a hidden signature Σ and A is Σ-algebra and $\sim$ is an equivalence on A, then an operation σ in $\Sigma_{s_1...s_n,s}$ is **congruent** for $\sim$ iff $\sigma_A(a_1, ..., a_n) \sim \sigma_A(a'_1, ..., a'_n)$ whenever $a_i \sim a'_i$ for $1 \leq i \leq n$. A **hidden Γ-congruence** on A is an equivalence relation on A that is congruent for each operation in Γ and is the identity on visible sorts. The Γ-congruence $\equiv^\Gamma_\Sigma$, called **behavioral equivalence**, on A is defined as follows: two data elements are equivalent iff they are equal, and two states are equivalent iff they cannot be distinguished by Γ-experiments,i.e., iff any experiment produces the same value when applied to them. The following is a basic result:

Theorem 5 Given a hidden subsignature Γ of Σ and a Σ-algebra A, $\equiv^\Gamma_\Sigma$ is the largest hidden Γ-congruence on A. $\Box$

An operation σ is **Σ-behaviorally congruent** for A (or simply **congruent**) iff it is congruent for $\equiv^\Gamma_\Sigma$. A hidden Σ-algebra A **Γ-behaviorally satisfies** a Σ-equation $e = (\forall X)\ t = t'$ `if` $u_1 = u'_1, ..., u_n = u'_n$, written $A \models\!\!\!\equiv^\Gamma_\Sigma e$, iff for any mapping $\theta\colon X \to A$, if $\theta(u_i) \equiv^\Gamma_\Sigma \theta(u'_i)$ for $1 \leq i \leq n$, then $\theta(t) \equiv^\Gamma_\Sigma \theta(t')$. If E is a set of Σ-equations, then $A \models\!\!\!\equiv^\Gamma_\Sigma E$ iff $A \models\!\!\!\equiv^\Gamma_\Sigma e$ for any $e \in E$. We say A **behaviorally satisfies** a behavioral specification $\mathcal{B} = (\Sigma, \Gamma, E)$ iff $A \models\!\!\!\equiv^\Gamma_\Sigma E$; we write $A \models\!\!\!\equiv^\Gamma_\Sigma \mathcal{B}$. Define $E \models\!\!\!\equiv^\Gamma_\Sigma e$ iff $A \models\!\!\!\equiv^\Gamma_\Sigma E$ implies $A \models\!\!\!\equiv^\Gamma_\Sigma e$ for every algebra A. Define $\mathcal{B} \models\!\!\!\equiv^\Gamma_\Sigma e$ iff $E \models\!\!\!\equiv^\Gamma_\Sigma e$.

Given a behavioral specification $\mathcal{B} = (\Sigma, \Gamma, E)$, the **provability relation** $\Vdash\!\!\!\vdash^\Gamma_\Sigma$ for Σ-equations is defined by the following rules:

1. Reflexivity: $E \Vdash\!\!\!\vdash^\Gamma_\Sigma (\forall X)\ t = t$.
2. Symmetry: If $E \Vdash\!\!\!\vdash^\Gamma_\Sigma (\forall X)\ t_1 = t_2$, then $E \Vdash\!\!\!\vdash^\Gamma_\Sigma (\forall X)\ t_2 = t_1$.
3. Transitivity: If $E \Vdash\!\!\!\vdash^\Gamma_\Sigma (\forall X)\ t_1 = t_2$ and $E \Vdash\!\!\!\vdash^\Gamma_\Sigma (\forall X)\ t_2 = t_3$, then $E \Vdash\!\!\!\vdash^\Gamma_\Sigma (\forall X)\ t_1 = t_3$.

4. Substitution: If $(\forall Y)\, t = t'$ `if` $u_1 = u'_1, ..., u_n = u'_n$ in E and $\theta\colon Y \to T_\Sigma(X)$ and $E \Vdash^\Gamma_\Sigma$ $(\forall X)\ \theta(u_i) = \theta(u'_i)$ for $1 \le i \le n$, then $E \Vdash^\Gamma_\Sigma (\forall X)\ \theta(t) = \theta(t')$.
5. Congruence:
   (a) If $E \Vdash^\Gamma_\Sigma (\forall X)\ t = t'$ where $t, t' \in T_{\Sigma \cup X, v}$ and $v \in V$, and $t_1, ..., t_{i-1}, t_{i+1}, ..., t_n \in T_{\Sigma\cup X}$, then $E \Vdash^\Gamma_\Sigma (\forall X)\ \sigma(t_1, ..., t_{i-1}, t, t_{i+1}, ..., t_n) = \sigma(t_1, ..., t_{i-1}, t', t_{i+1}, ..., t_n)$.
   (b) If $E \Vdash^\Gamma_\Sigma (\forall X)\ t_i = t'_i$ for $1 \le i \le n$ and σ is congruent operation in Σ, then $E \Vdash^\Gamma_\Sigma$ $(\forall X)\ \sigma(t_1, ..., t_n) = \sigma(t'_1, ..., t'_n)$.

Define $\mathcal{B} \Vdash (\forall X)\, t = t'$ iff $E \Vdash^\Gamma_\Sigma (\forall X)\, t = t'$. These rules specialize those of ordinary equational deduction by considering all sorts visible. Note that (5b) only applies to congruent operations. If all operations are congruent, then ordinary equational deduction is sound for behavioral satisfaction. The following expresses soundness with respect to both equational and behavioral satisfaction, generalizing a result in [15] that equational deduction is sound when all operations are congruent.

Theorem 6 If $\mathcal{B} \Vdash (\forall X)\, t = t'$, then $\mathcal{B} \models\!\!\!\equiv^\Gamma_\Sigma (\forall X)\, t = t'$ and also $E \models (\forall X)\, t = t'$. □

General coinduction [37, 39, 66] can be used to prove that a Σ-equation $(\forall X)\, t = t'$ is behaviorally satisfied by a behavioral specification $\mathcal{B}$ by the following steps:

- Define a binary relation R on terms (R is called the **candidate relation**).
- Show that R is a hidden Σ-congruence.
- Prove that $t\ R\ t'$.

Soundness of general coinduction follows directly from Theorem 5. Its major problem is that it requires human creativity to define an appropriate candidate relation. **Context induction** [50] can also be used to prove behavioral properties, using well-founded induction on the context structure to show that it is valid for all experiments. But in many real examples, context induction is not trivial and requires extensive human input, for example, in the form of inductive lemmas that can be difficult to discover and difficult to prove [24].

It often happens that some experiments are unnecessary in a context induction, because the equations imply that some experiments are equivalent to others. A similar but dual situation occurs in abstract data type theory when all the elements can be generated from a subset of operations, called the constructors, generators, or basis (when induction is involved). A general definition of cobasis is introduced in [67], and a simplified version can be given as follows: a **cobasis** Δ is a subset of operations in Γ that generates enough experiments, in the sense that no other experiment can distinguish any two states that cannot be distinguished by these experiments.

The denotational semantics of a behavioral module is the class of all algebras (i.e., implementations) that behaviorally satisfy specifications, and their operational semantics is given by behavioral rewriting. Behavioral modules in BOBJ are defined between the keywords `bth` and `end`. Sorts in behavioral modules are considered hidden unless declared with the keyword `dsort`, for visible sorts in behavioral modules. Similarly, operations in behavioral modules are considered congruent unless given the attribute `ncong`.

Example 4 A Behavioral Theory of Sets

```
bth BSETNAT is sort Set .
  pr NATS .
  op empty : -> Set .
  op _in_ : Nat Set -> Bool  .
```

```
    op insert : Nat Set -> Set .
    vars N1 N2 : Nat . var S : Set .
    eq N1 in empty = false .
    eq N1 in insert(N2, S) = N1 == N2 or N1 in S  .
  end
```

The first equation gives the result of observing `empty` with `_in_`, and the next equation gives the results of observing `insert` with `_in_`. □

The most important difference between this behavioral theory and the initial theory for sets in Example 6 is that this theory does not have the equation `insert(E1, S) = S if E1 in S` . Although the other equations look the same, they are methodologically different. Data theories are usually designed with respect to constructors, but behavioral theories are designed with respect to observors. For example, `empty` and `insert` are constructors of the data theory `SET`, i.e., all ground sets can be created with them; and then all other operations can be defined based on the terms generated by these constructors.

We recommend designing a behavioral theory by selecting some operations as a cobasis to generate the behavioral equivalence relation, and then defining other operations with respect to these basic observers. For example, in `BSETNAT` above, the operation `_in_` is the unique observer in the cobasis, so that two sets are behaviorally equivalent iff they always return the same visible results under the observation of `_in_`, i.e., iff they have the same elements. Then for example, the traditional implementation of sets as lists with possible repetitions is behaviorally correct.

2.5 Logical Programming

Our claim that the OBJ languages are rigorously based on versions of equational logic is best demonstrated by defining the notion of logical programming language, and then showing how the various OBJ computations fit that definition. To be fully formal would require formalizing the notion of "a logic," including both deductive and model theoretic aspects. Such a formalization was sketched in the main paper on institutions [31], was carried further in a somewhat different way by Meseguer in [55], and was recently more fully realized in [60]. Here we leave that notion informal, assuming that a logic $\mathcal{L}$ has notions of signature Σ, Σ-sentence (with $Sen(\Sigma)$ the set of all these), Σ-model (with $Mod(\Sigma)$ the class of all these), Σ-satisfaction $\models_\Sigma$ and Σ-deduction $\vdash_\Sigma$ such that deduction is **sound**, i.e., such that $P \vdash_\Sigma e$ implies $P \models_\Sigma e$ where P is a set of Σ-sentences, e is a Σ-sentence, and $P \models_\Sigma e$ means that $M \models_\Sigma P$ implies $M \models_\Sigma e$ for all Σ-models M. Readers familiar with institutions with proofs [60] will see how to use that notion to fully formalize the above. Although it is less clear how to formalize the meta-logic $\mathcal{L}'$ introduced below, it suffices to let it be just the ordinary language of mathematics, applied to $\mathcal{L}$; in particular, it allows talk about proofs in $\mathcal{L}$.

A **program** P of a logical programming language over a logic $\mathcal{L}$ is a theory over $\mathcal{L}$, i.e., a set of Σ-sentences; a **query** is a sentence in $\mathcal{L}'$ of the form $(\exists X)\ q(X)$ where X is a set of variable symbols; and an **answer** to such a query is an assignment a from X to terms in $\mathcal{L}'$ such that $q(a)$ is in $Sen(\Sigma)$ and $P \vdash_\Sigma q(a)$, where $q(a)$ denotes the result of substituting $a(x)$ into q for each x in X; this is sound with respect to the intended models by assumption.

We now consider some examples. The first is **loose semantics** for (say) many sorted (or order sorted) equational logic, where signatures declare sorts (with subsorts) and operations, where sentences are equations, models are algebras, and satisfaction and deduction are as usual. Queries are of the form $(\exists p)\ \overline{p} = e$ where p is a variable for proofs over P, e is an equation, and $\overline{p}$ denotes the equation that

p proves. Here we have **query completeness** in the sense that a query $(\exists X)\ q(X)$ has an answer a with $P \vdash_{\Sigma} q(a)$ iff $P \models_{\Sigma} q(a)$ (this generalizes the query completeness notion of [55] to our setting).

Our second example is **initial semantics** for many sorted (or order sorted) equational logic, where the relation $P \models_{\Sigma} e$ is restricted to initial models for P, and where queries have the form $(\exists p)\ \overline{p} = e$, and where deduction allows induction as well as equational reasoning. Query completeness holds here, although there is no algorithm that can realize it.

Our third example is **pure logic programming**, in the sense used in the Prolog community (e.g., [54]). Here signatures Σ declare relation symbols, Σ-sentences are Horn clauses, and Σ-queries have the form[3] $(\exists X)\ R(X)$ where R is a conjunction of relations applied to variables, and deduction is resolution. Then a suitable Herbrand theorem (e.g., see [42]) implies that we can use either loose or initial semantics, and that query completeness holds. All this extends to many sorted Horn clause logic with equality [42].

Our fourth example is **term rewriting** over equational theories P that are terminating. Here sentences are rewrite rules, queries have the form $(\exists t')\ t = t'$ where t' is reduced with respect to P, and deduction is term rewriting with P. This is not query complete over loose semantics, unless P is also Church-Rosser and therefore canonical, in which case initial semantics also applies.

Our fifth example is **behavioral semantics** as implemented by BOBJ's C4RW algorithm [45, 34]. Here programs are signatures (that include congruence declarations for some operations) with sets of rewrite rules and a cobasis declaration; satisfaction is behavioral; deduction is C4RW, shown sound in [45]; and queries have the form $(\exists p)\ \overline{p} = e$ where e is a behavioral equation and p is a C4RW proof. This is neither query complete nor reducible to initial semantics.

The first and fifth examples would not usually be called programming, because instead of computing a value, they try to prove an assertion. However, we claim that verification is the proper analog of programming for the level of specifications. Moreover, C4RW is surprisingly efficient when it terminates, e.g., [45] – which it may not, just as with ordinary programming. Note that when C4RW fails to return `true`, the equation tested could still be valid, due to the necessary incompleteness of behavioral deduction. On the other hand, the first example is very far from being efficient, so that its computations, which essentially are blind searches, should not properly be called programming; in fact, the OBJ languages do not implement this, but rather allow users to construct proofs by hand. But if the program is canonical, then the validity of equations can be decided by checking whether or not the two terms have the same reduced forms, using the built in `==` operation; this can be seen as based on initial semantics. We conclude that Meseguer's initiality requirement in [55] is reasonable for those computations that are ordinarily called programming, because fixed data structures such as integers are likely involved, though this is not necessarily the case, e.g., for term rewriting proofs of equational identities in the theory of groups. However, query completeness is less reasonable, and neither requirement is appropriate if we wish to capture all of the semantics of OBJ family languages. Nevertheless, these two notions usefully enrich our understanding of the nature of computation.

We finally remark that the logical "existential" or "query" semantics sketched here is not limited to first order languages. For example, it also applies to pure functional programming languages, such as Haskell [52], though we omit the details, some of which can be found in [55]. Moreover, it applies to databases and to brokers in service oriented architectures.

[3]It would be more consistent with our other examples to ask for proofs of $R(X)$ instead of just the substitution. Note that asking for a proof as output is just asking for a trace of a computation.

3 Modularization

The module systems of parameterized programming go well beyond those of standard programming languages. We believe that views are not just a syntactic convenience, but are necessary for realizing the full potential of module parameterization. For example, we speculate that the lack of views explains the confusing multiplicity of semantics that have been given for ML functors ("functor" is ML terminology for parameterized module, see [72]), as well as its awkward treatment of sharing.

3.1 Parameterization and Views

Given signatures Σ, Σ' with sorts S, S', then a **signature morphism** $\Sigma \to \Sigma'$ is a pair (f, g) where $f\colon\ S \to S'$, and g is an $(S^* \times S)$-indexed function $g_{w,s}\colon\ \Sigma_{w,s} \to \Sigma'_{f(w),f(s)}$. A **view**, or **theory morphism**, from a theory $T = (\Sigma, E)$ to a theory $T' = (\Sigma', E')$ is a signature morphism $v\colon\ \Sigma \to \Sigma'$ such that if $(\forall X)\ t = t'$ is an equation in E, then $E' \vdash (\forall \overline{X})\ \overline{v}(t) = \overline{v}(t')$ where $\overline{X}_{f(s)} = X_s$ for any sort $s \in \Sigma$ and $\overline{v}\colon\ T_{\Sigma(X)} \to T_{\Sigma'(\overline{X})}$ is the Σ-homomorphism induced by v; we may write $v\colon\ T \to T'$. The OBJ languages do not check semantic correctness of views, but only their syntax; therefore users should check the semantics.

Example 5 A Simple View

```
    view V from GROUP to INT is
      sort Elt to Int .
      op (_ -1) to (-_) .
      op (_*_) to (_+_) .
   end
```

View syntax is straightforward, except that when items are omitted, the system tries to figure out those missing items; the resulting views are called **default views**, see [48] for details. □

A **parameterized specification** or **parameterized theory** is a pair (T_1, T_2) of specifications such that T_1 is included in T_2; we call T_1 the **parameter** or **interface theory** and T_2 the **body**. In Example 6 below, T_1 is `ELT` and T_2 is `SET`. Instantiation of (T_1, T_2) with an actual parameter P requires a view $T_1 \to P$ to describe the binding of actual to formal parameters; often a default view can be used. Following ideas developed for the Clear specification language [7, 31], the instantiation is given by a colimit construction.

Example 6 A Parameterized Initial Theory of Sets The initial theory `SET` below allows us to form sets of elements from any collection with an equality relation defined on it satisfying the law of identity, given in its interface theory `ELT`. Parameterization of a module `M` by an interface `I` is indicated with the notation `M[X :: I]`, where `X` is the formal parameter of the parameterized module.

```
   th ELT is sort Elt .
      op eq : Elt Elt -> Bool .
      var E : Elt .
      eq eq(E, E) = true .
   end

   dth SET[X :: ELT] is sort Set .
      op empty : -> Set .
      op _in_ : Elt Set -> Bool  .
```

```
    op insert : Elt Set -> Set .
    vars E1 E2 : Elt . var S : Set .
    eq E1 in empty = false .
    eq E1 in insert(E2, S) = eq(E1, E2) or E1 in S .
    eq insert(E1, S) = S if E1 in S .
  end
```

The following tells BOBJ to instantiate SET with the builtin module INT of integers, and call the result INTSET:

```
  dth INTSET is
    pr SET[INT] .
  end
```

This uses a default view from ELT to INT and pr (for "protecting") indicates a importation. □

Two additional features from parameterized programming are renaming and sums of modules. The first allows selected sorts and operations to be renamed within a module; this can be very helpful when reusing modules in new contexts. The sum just combines two or more modules, taking proper account of any shared submodules that may have arisen through importation. The syntax of these features is illustrated in the following:

```
  dth NATS+INT is
    pr NATS *(sort Nat to Peano, op 0 to zero) + INT .
  end
```

Here a sort and constant of NATS are renamed to avoid conflict with those of INT, and the two modules are then combined; the parser can determine whether the sort of any given term is Peano or Int, even though the operations _+_ and _*_ are overloaded.

Example 7 Behavioral Theory of Streams The behavioral specification STREAM declares infinite streams parameterized by the "trivial" interface theory TRIV, which only requires that some sort be designated.

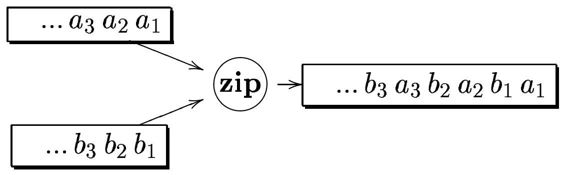
```
  th TRIV is sort Elt . end

  bth STREAM[X :: TRIV] is sort Stream .
    op head_ : Stream -> Elt .
    op tail_ : Stream -> Stream .
    op _&_ : Elt Stream -> Stream .
    var E : Elt . var S : Stream .
    eq head(E & S) = E .
    eq tail(E & S) = S .
  end
```

The operation _&_ inserts an element into the head of a stream, and head and tail respectively return the first element, and the stream after removing its first element. The next specification adds an operation which "zips" two streams together by taking elements from them alternately:

```
   bth ZIP[X :: TRIV] is pr STREAM[X] .
     op zip : Stream Stream -> Stream .
     vars S S' : Stream .
     eq head zip(S,S') = head S .
     eq tail zip(S,S') = zip(S', tail S) .
  end
```

The picture below shows the application of zip to two input streams:



The command red does behavioral rewriting in the context of a behavioral theory. For example,

```
open ZIP[NAT] .
  ops ones twos : -> Stream .
  vars S S' : Stream .
  vars N M  : Nat .
  eq head ones = 1 .
  eq tail ones = ones .
  eq head twos = 2 .
  eq tail twos = twos .
  red head tail tail zip(ones, twos).
close
```

We will use these behavioral theories in later examples. □

3.2 Behavioral Views

Behavioral parameterized theories can use any kind of theory as their interfaces, but the interfaces of non-behavioral theories must not be behavioral theories, i.e., behavioral theories are only allowed as interfaces for other (parameterized) behavioral theories. Given behavioral theories $\mathcal{B}_i = (\Sigma_i, \Gamma_i, E_i)$ for $i = 1, 2$, let the set of visible sorts and the set of hidden sorts in $\mathcal{B}_i$ be V_i and H_i, respectively. Then a **behavioral view** from $\mathcal{B}_1$ to $\mathcal{B}_2$ is a signature morphism $v\colon \Sigma_1 \to \Sigma_2$ such that: (1) $v(s) \in V_2$ for any sort $s \in V_1$; and (2) for any equation $(\forall X)\, t = t'$, if $\mathcal{B}_1 \models (\forall X)\, t = t'$, then $\mathcal{B}_2 \models (\forall \overline{X})\, \overline{v}(t) = \overline{v}(t')$ where $\overline{X}_{v(s)} = X_s$ for any sort $s \in \Sigma_1$ and $\overline{v}\colon T_{\Sigma_1(X)} \to T_{\Sigma_2(\overline{X})}$ is the homomorphism induced by v.

Notice that this definition of behavior views requires verifying all behavioral properties of the source module, which is impossible in practice. It is sufficient to define a signature morphism v from $\mathcal{B}_1$ to $\mathcal{B}_2$ such that (1) all translated equations of $\mathcal{B}_1$ are behaviorally satisfied by $\mathcal{B}_2$; and (2) the image of a cobasis of $\mathcal{B}_1$ under v is a cobasis of $\mathcal{B}_2$. This is because it then follows that any behavioral property of $\mathcal{B}_1$ is also behaviorally satisfied by $\mathcal{B}_2$. In practice, the condition (2) above can be satisfied by making some operations non-congruent.

3.3 Higher Order Parameterized Programming

Since [29] shows that first order parameterized modules give essentially all the programming power of higher order functional languages but with a first order logic, one may ask what higher order parameterized modules can add to this. The answer is that they add an architectural level of structural description and reuse that goes well beyond that of first order modules, as shown by the example below.

As already mentioned, all current OBJ family languages have first order parameterized modules and views, including Maude and CafeOBJ, as well as CASL. OBJ3 for some time has had formal parameters that are parameterized by previously introduced formal parameters [38], but BOBJ is the only language that provides higher order views. A very different approach to higher order modularization based on the λ-calculus and type theory, appears in Extended ML [70], in ASL [69], and in its extension ASL+FPC [1]. Recent SML/NJ releases of ML [59] include higher order parameterized modules, based on higher order parameterized signatures, but without views. C++ allows higher order parameterized templates, but these amount to little more than macro expansions with type checking. Larch [23] has a parameter passing mechanism that can simulate some uses of first order views, but it does not support views as reusable first class citizens. A new semantics for higher order parameterized programming is given in Section 3.3.2. We first illustrate the main ideas with an inductive proof scheme written in BOBJ.

3.3.1 A Reusable Induction Scheme

This subsection illustrates higher order parameterized programming by defining a reusable induction scheme[4] that builds on one in [38], which in turn built on [74]. The interface module `NIND` below requires constructors for basic Peano induction for natural numbers; it is the interface for the induction scheme, which will be instantiated with actual modules which specify the inductive problem to be solved. Because `NIND` has initial semantics, allowable actuals must also have initial semantics for their two operations that correspond to those in this module.

```
obj NIND is sort Term .
  op c : -> Term .
  op f : Term -> Term .
end
```

We now define terms over `NIND` by introducing a new constant symbol `x` of sort `Term`; this will be the induction variable.

```
obj TERM [X :: NIND] is
  op x : -> Term .
end
```

Because `TERM` has `NIND` as its interface, instanting it with an actual module `A` gives a module that defines terms over the operations of `A`. For example, if the formal parameter of `TERM` is instantiated with `NATS`, then `sum(x) + sum(x)` is one of the resulting terms. This module has initial semantics because we want its models to contain all and only terms in the variable `x`.

The interface theory below calls for two terms, for the left and right sides of an equational goal:

```
th GOAL [X :: NIND] [T :: TERM[X]] is
  ops l r : -> Term .
end
```

Because its first keyword is `th` (for theory), this module has loose semantics, which allows its two constants to be instantiated arbitrarily. It has two formal parameters, the first with interface `NIND`, with the second, `TERM[X]`, dependent[5] on the first. The two constants, `l` and `r`, represent the left and right sides of a goal.

A module with its interfaces separated into groups with brackets can be partially instantiated by providing actual modules for the parameters in the first group, with result a module parameterized by the remaining parameters, and having the partial instantiation as its body. Now we define the induction scheme:

[4]The modules in this example are generated by hand, but the Kumo system generates (first order) inductive proof schemes automatically [33].

[5]This gives the effect of what is called a **dependent type** in type theory.

```
th SCHEME [P :: NIND, G :: GOAL[P]] is
  us B is G[(c).(TERM[P])] .
  let base = l.B == r.B .
  us H is G[(x).(TERM[P])] .
  eq l.H = r.H .
  us C is G[(f(x)).(TERM[P])] .
  let step = l.C == r.C .
  let proof = base and step .
end
```

The second interface of SCHEME is a parameterized module having GOAL[P] as its interface. The first line of the body of SCHEME instantiates G with TERM[P] by mapping the symbol x in its formal parameter TERM[P] to c in the actual parameter TERM[P], which is denoted (c).(TERM[P]) using the "dot" qualification convention; the result is then renamed B (for base) and is imported, where the keyword "us" indicates importation without requiring initiality or any other constraints to be satisfied. The equation base = l.B == r.B is well defined because l and r are constants of the sort Term.G[(c).(TERM[P])] in B. Similarly, the H and C importations are for the induction hypothesis and the inductive step. Lines 2,3,5 of the body of SCHEME correspond to lines 1,3,4 of the body of the above "open", except that SCHEME must be instantiated before the proof can be executed, and the two cases to be checked are conjoined into one by "and".

In more detail, to do an inductive proof using SCHEME, we first instantiate its first formal parameter P with an actual module containing appropriate functions over its constructors, then we instantiate its second formal parameter G with two terms over that, say defined by an actual module M. Then B is calculated as M[(c).(TERM[P])], and all the operations in G are replaced by operations from M. More precisely, a view from G[(c).(TERM[P])]) to M[(c).(TERM[P])] is created for replacing the operations of B in SCHEME. So to apply SCHEME to NATS, we first define a view from NIND to NATS,

```
view NINDV from NIND to NATS is
  op f to s .
end
```

which will instantiate the first parameter of SCHEME to NATS. Notice that the mappings sort Term to Nat and op c to 0 need not be stated here, since they are inferred by the default view mechanism. On the other hand, op f to s is needed because there are two unary operations in NATS. Once SCHEME is instantiated with NATS, its second parameter becomes G[NINDV], which we instantiate with

```
view SUMV from GOAL[NINDV] to GOAL[NINDV] is
  op l to (sum(x) + sum(x)) .
  op r to (x * s x) .
end
```

so that the complete instantiation is accomplished with the command

```
make SUM-PROOF is SCHEME[NINDV, SUMV] end
```

This evaluates the module expression in its body and gives it the name SUM-PROOF. In the evaluation, B is calculated as GOAL[NINDV][(0).TERM[NINDV]], using the view from G[P][(c).TERM[P]] to GOAL[NINDV][(0).TERM[NINDV]] that BOBJ automatically generates, with the body

```
 op l to sum(0) + sum(0) .
 op r to 0 * s 0 .
```

Since SUMV maps l to sum(x) + sum(x), when the (still parameterized) module GOAL[NINDV] is instantiated with TERM[NINDV], then x is mapped to 0, so that l is mapped to sum(0) + sum(0). Under the above view, the equation base = l.B == r.B becomes base = sum(0)+ sum(0) == 0 * s 0. Similar work is done for the modules H and C. Thus SUM-PROOF contains

```
    eq base = sum(0) + sum(0) == 0 * s 0 .
    eq sum(n) + sum(n) = n * s n .
    eq step = sum(s n) + sum(s n) == s n * s s n .
```

and then the whole proof can be checked with just one command,

```
    red proof .
```

for which BOBJ returns `true` after execution. However, users often want to see more detail, which can be accomplished by first giving the command

```
    set trace on .
```

Actually, there is a simpler way to instantiate using a so-called *in-line view*:

```
    make SUM-PROOF is SCHEME[NINDV, view to GOAL[NINDV] is
                                    op 1 to sum(x) + sum(x) . end] end
```

and `NINDV` could also be replaced by an in-line view.

Of course, higher order modules can do much more than this.....

3.3.2 Semantics for Higher Order Modules

This section sketches a semantics for higher order modules, based on the categorical general systems theory of [25, 32], particularly its higher order capability, the importance of which was emphasized (to Goguen) by Gregory Bateson in the early 1970s. This section assumes familiarity with category theory (for which see [20, 63] among many other sources), and necessarily begins rather abstractly. The intention is to develop an approach that is independent of any particular linguistic basis, and that in particular transcends the *ad hoc* peculiarities of the many architecture description languages that have been proposed. The approach also applies to mainstream imperative programming languages, e.g., by using underlying concrete institutions like those proposed in [47]. A semantics for the Maude module system in [17] is rather similar, but applies only to the first order case, and is formulated as an institution the signatures of which are diagrams of theories.

The most basic construction is the category $\mathbb{D}(\mathbb{C})$ of diagrams over a category[6] $\mathbb{C}$, which has objects functors $\mathbb{B} \to \mathbb{C}$ from a variable base category $\mathbb{B}$ to the fixed target $\mathbb{C}$, with morphisms from $a\colon \mathbb{B}_1 \to \mathbb{C}$ to $b\colon \mathbb{B}_2 \to \mathbb{C}$ being a functor $f : \mathbb{A} \to \mathbb{B}$ plus a natural transformation $\alpha\colon f;b \Rightarrow a$ (where ";" denotes composition), with the evident identities, and with composition $(f,\alpha);(g,\beta) = (f;g,\ (f*\beta);\alpha)$ where (g,β) is a morphism from b to $c\colon \mathbb{B}_3 \to \mathbb{C}$. Then $\mathbb{D}(\mathbb{C})$ is cocomplete if $\mathbb{C}$ is[7]; it will be convenient to write $\downarrow D$ for the colimit of D in $\mathbb{D}(\mathbb{C})$. Also, note that there is a natural injection $\mathbb{C} \to \mathbb{D}(\mathbb{C})$, for which we will use the notation $\lceil \_ \rceil$, and that $\downarrow \_$ is right adjoint to $\lceil \_ \rceil$.

Since $\mathbb{D}$ can be applied to any category, we can form $\mathbb{D}(\mathbb{D}(\mathbb{C}))$, which we denote by $\mathbb{D}^2(\mathbb{C})$ or just $\mathbb{D}^2$; now we can iterate to form $\mathbb{D}^n$, with $\mathbb{D}^0 = \mathbb{C}$ by convention; moreover, we can form the colimit in $\mathbb{C}at$ of the sequence of natural injections

$$\mathbb{D}^0 \to \mathbb{D}^1 \to \mathbb{D}^2 \to \mathbb{D}^3 \to \ldots\ldots.$$

since $\mathbb{C}at$ is cocomplete; denote this colimit $\mathbb{D}^\infty$. Also, there is a functor $Colim\colon \mathbb{D}^2 \to \mathbb{D}$ which computes the diagram of colimits of a diagram of diagrams[8]; substituting $\mathbb{D}^{n-1}(\mathbb{C})$ for $\mathbb{C}$gives also $Colim\colon \mathbb{D}^{n+1} \to \mathbb{D}^n$. Similarly, let $\lceil \_ \rceil$ denote any injection $\mathbb{D}^n \to \mathbb{D}^{n+1}$, let $\downarrow \_$ denote any colimit

[6] It helps to handle submodule sharing if $\mathbb{C}$ is a category with inclusions in the sense of [9, 10].
[7] This follows from its being the indexed category $[\_, \mathbb{C}]$ using general results about indexed categories.
[8] It can be obtained by extending the colimit functors on each $[\mathbb{B}, \mathbb{C}]$ to all of $\mathbb{D}(\mathbb{C})$.

functor $\mathbb{D}^{n+1} \to \mathbb{D}^n$, let $\downarrow^2$ denote $\downarrow\downarrow$ and more generally, let $\downarrow^k\colon \mathbb{D}^{n+k} \to \mathbb{D}^n$. Finally, let $\downarrow^\infty$ denote the map $\mathbb{D}^\infty \to \mathbb{C}$ induced by all the maps $\downarrow^n\colon \mathbb{D}^n \to \mathbb{C}$. Then it is not hard to see that $Colim\lceil D\rceil = D$, that $\downarrow\lceil D\rceil = D$, and that $\downarrow Colim\ D = \downarrow^2 D$, among other such identities[9].

To apply this machinery to higher order modules, we substitute for $\mathbb{C}$ a category $\mathbb{T}$ of theories (which for BOBJ would involve constraints in the sense of [31] for initial semantics); $\mathbb{T}$ contains the basic, or zeroth order, modules. First order modules lie in $\mathbb{D}(\mathbb{T})$, second order modules in $\mathbb{D}^2(\mathbb{T})$, an so on; from now on, we write just $\mathbb{D}, \mathbb{D}^2$, etc. The functor $\downarrow^\infty$ computes the zeroth order specification of a system built by composing higher order modules. Note that the sum operation (+) on theories is just coproduct, and that the same applies to diagrams of any order. We do not give a categorical semantics for renaming (the $*$ operation) because it is easier (almost trivial) to define it operationally, and in any case, colimits do not need to care much for names, since they keep careful track of where names come from.

Parameterization and instantiation are more interesting. Parameterized programming [28, 48] defines a parameterized module to be a theory inclusion $i\colon P \to B$ where P is the parameter theory and B is the body (see Section 3.1). This works well, but it does not capture the idea that the inclusion itself is a module. However, we can do this with above machinery by encapsulating i, i.e., by viewing it as a diagram M, i.e., as an object in $\mathbb{D}$. This shift of level is part of a much richer, software architecture oriented point of view, in which module instantiation appears as a kind of module interconnection, instead of a perhaps *ad hoc* seeming pushout: let A be an actual parameter for M, i.e., let there be given a "fitting morphism" $f\colon P \to A$; then the instantiation of M by A is indicated by the module interconnection diagram $n\colon \lceil A\rceil \to M$ in $\mathbb{D}^2$, where the functor component of n maps the one object of the category that underlies $\lceil A\rceil$ to the object underlying P in M, and the natural transformation of n is f.

Of course, much more can be done, by making use of more complex diagrams of higher orders. Sockets, pipes, connectors, ports, adaptors, channels – the entire zoo of contemporary software architecture is naturally modeled in this formalism, without needing to bring in any additional *ad hoc* features. The final chapters of a recent book [20] by José Fiadeiro contain much that is relevant to this topic, though with a different semantics; in particular, it provides excellent motivation for higher order parameterized modules, with many examples from software engineering. It seems likely that an alternative approach can be developed based on John Gray's Cartesian closed category of sketches [49].

We now return briefly to the induction scheme of Section 3.3.1. The module `TERM` is a first order parameter theory for the second order module `GOAL`, which in turn is a parameter theory for the third order module `SCHEME`, which has further structure arising from its internally defined modules `B`, `H`, and `C`. The modules `TERM`, `GOAL` and `SCHEME` are also all parameterized by `NIND`, but instantiating `NIND` with `NAT` using the view `NINDV` and taking the colimit still yields a third order module, because `SCHEME` is still parameterized by the second order (but now partially instantiated) module `GOAL`. The next step instantiates `GOAL` with itself, using a tricky view `SUMV` that introduces the terms to be proved equal. Now taking the colimit collapses to a zeroth order theory in which the computations can take place, triggered by a single command. (It is possible to draw some helpful diagrams for all this, but there isn't sufficient room in this paper.)

It is interesting to notice that techniques like those used for higher order modules can also be used to define higher order data types. We illustrate this with a simple example instead of giving a general construction. Let $L\colon \mathbb{T} \to \mathbb{T}$ be the functor which sends a theory T to the theory $T + \texttt{LIST}[T]$, and construct the sequence

[9]However, these only hold up to isomorphism.

$$\mathbb{L}^0 \to \mathbb{L}^1 \to \mathbb{L}^2 \to \mathbb{L}^3 \to ....$$

where $\mathbb{L}^0 = T$, $\mathbb{L}^1 = L(T)$, $\mathbb{L}^2 = L(L(T))$, etc., with the evident inclusions. Then its colimit includes elements, lists, lists of lists, lists of lists of lists, etc. It seems there may be an amusing analogy here with continued fractions that is worth further exploration.

4 Circular Coinductive Rewriting

Behavioral rewriting [15] is to behavioral deduction what standard rewriting is to standard equational deduction, a simple but useful proof method. Based on the notion of cobasis, a more powerful proof method called **circular coinduction** is introduced in [68]. A enriched behavioral deduction system can be got by adding the following rule: Suppose Δ is a cobasis of a behavioral specification $\mathcal{B} = (\Sigma, \Gamma, E)$ and $<$ is a well founded partial order on Γ-contexts which is preserved by the operations in Γ. For any terms t_1 and t_2 in $T_\Sigma(X)$, if for any $\delta \in \Delta$ and for appropriate variables W, $\mathcal{B} \Vdash^{\Gamma}_{\Sigma} (\forall X)(\forall W)\, \delta(t_1, W) = c[\theta(t_1)]$ and $\mathcal{B} \Vdash^{\Gamma}_{\Sigma} (\forall X)(\forall W)\, \delta(t_2, W) = c[\theta(t_2)]$ and $c < \delta$, or else $\mathcal{B} \Vdash^{\Gamma}_{\Sigma} (\forall X)(\forall W)\, \delta(t_1, W) = u$ and $\mathcal{B} \Vdash^{\Gamma}_{\Sigma} (\forall X)(\forall W)\, \delta(t_2, W) = u$ for some Γ-term u, then $\mathcal{B} \Vdash^{\Gamma}_{\Sigma} (\forall X)\, t_1 = t_2$. Circular coinductive rewriting proves behavioral equalities by combining behavioral rewriting with circular coinduction [35]; it also strengthens the duality with induction by allowing coinductive hypotheses to be used in proofs.

BOBJ provides a limited operational semantics for behavioral modules, by applying equations as behavioral rewrite rules. Because of non-congruent operations, ordinary rewriting is not in general sound, as illustrated by the following behavioral theory with a non-congruent operation:

Example 8 Nondeterministic Stacks The following behavioral variant of a stack theory illustrates one way that nondeterminism can arise in hidden algebra specifications:

```
bth NDSTACK is sort Stack .
  protecting NAT .
  op push _ : Stack -> Stack [ncong] .
  op top _ : Stack -> Nat .
  op pop _ : Stack -> Stack .
  var S : Stack .
  eq pop push S = S .
end
```

The operation `push` places a nondeterministically chosen natural number on the stack's top. Even for behaviorally equivalent stacks `S1` and `S2`, `push(S1)` and `push(S2)` may insert different natural numbers onto `S1` and `S2`; therefore `push(S1)` and `push(S2)` may be distinguishable by the attribute `top`, so that `push` should be declared non-congruent. The equation in this specification says that a stack is not behaviorally changed by pushing a new element and then popping it. Notice that `push(pop(push(S))) == push(S)` is not behaviorally satisfied, although `pop(push(S))` and `S` are behaviorally equivalent. However, ordinary rewriting will reduce `push(pop(push(S)))` to `push(S)`. □

Behavioral rewriting is invoked with the command `red`, which handles non-congruent operations properly. A term $C[\,\theta(l)\,]$ **behaviorally rewrites** to $C[\,\theta(r)\,]$, where $C[\,\Box\,]$ is a context and $l \to r$ is a rewrite rule, iff one of the following is satisfied:

1. The redex does not have a non-trivial context.
2. All operations from the top of C down to $\Box$ are congruent.

3. The context of the redex has a subcontext D such that all the operations from the top of D to $\square$ are congruent and D has a visible sort.

For example, `push(pop(push(S)))` cannot be reduced to `push(S)`, because the context `push(`$\square$`)` doesn't satisfy the conditions above.

Behavioral rewriting can prove simple behavioral properties, but more powerful methods are needed to verify more difficult behavioral properties. Unlike general coinduction [39] and context induction [3], conditional circular coinductive rewriting with provides a powerful way to prove behavioral properties, without intensive human intervention. The C4RW algorithm also includes very useful capabilities for automatic cobasis discovery and for case analysis; the algorithm is described in detail and proved correct in [68], and is also described and then illustrated with a correctness proof for a non-trivial version of the alternating bit protocol in [34].

4.1 Mutual Coinduction

Behavioral operations can mutually depend on each other. In this case, behavioral properties $\{P_1, ..., P_n\}$ may also depend on each other, in such a way that no P_i can be proved by itself, but they can all be proved together. The BOBJ syntax for this is:

```
    cred  (<goal>) ... (<goal>) .
```

where the goals may be conditional. Mutual circular coinductive rewriting is in the C4RW algorithm. The example below uses operations `odd` and `even` which take a stream and return streams respectively formed by the elements in the odd and even positions of the argument stream. Thus odd($e_1\, e_2\, e_3\, e_4\, e_5\, e_6\, e_7\, e_8$ ...) is $e_1\, e_3\, e_5\, e_7$ ..., while even($e_1\, e_2\, e_3\, e_4\, e_5\, e_6\, e_7, e_8$ ...) is $e_2\, e_4\, e_6\, e_8$ ....

```
    bth ODD-EVEN[ X :: TRIV ] is pr ZIP[X] .
      var S : Stream .
      ops (odd_) (even_) : Stream -> Stream .
      eq head odd S  = head S .
      eq tail odd S  = even tail S .
      eq head even S = head tail S .
      eq tail even S = even tail tail S .
   end
```

This module imports `ZIP`, and all its operations are behavioral since they all preserve the intended behavioral equivalence, which is 'two streams are equivalent iff they have the same elements in the same order.' The property `zip(odd S, even S) = S` is proved by circular coinduction with:

```
   cred zip(odd S, even S) == S .
```

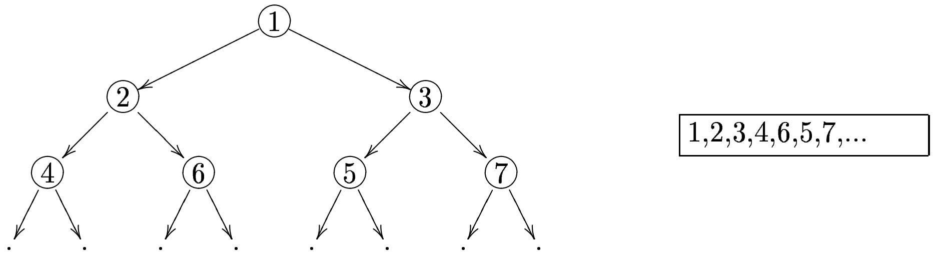
We next introduce a behavioral module for infinite binary trees:

```
    bth TREE[ X :: TRIV ] is sort Tree .
      op root_ : Tree -> Elt .
      ops left_ right_ : Tree -> Tree .
      op make : Elt Tree Tree -> Tree .
      var E : Elt .  vars T1 T2 : Tree .
      eq root  make(E, T1, T2) = E .
      eq left  make(E, T1, T2) = T1 .
      eq right make(E, T1, T2) = T2 .
    end
```

Given a tree, the operations `root`, `left` and `right` respectively return its root, its left subtree, and its right subtree; these three operations are a cobasis of `Tree`. The operation `make` takes an element and two trees to create a new tree. Next, define an operation `t2s`, which transforms trees into streams, and an operation `s2t` which does the inverse:

```
bth TREE-STREAM [ X :: TRIV ] is
  pr TREE[X] + ODD-EVEN[X] .
  op t2s_ : Tree -> Stream .
  op s2t_ : Stream -> Tree .
  var T : Tree . var S : Stream .
  eq head t2s T  = root T .
  eq tail t2s T  = zip(t2s left T, t2s right T) .
  eq root s2t S  = head S .
  eq right s2t S = s2t tail odd S .
  eq left s2t S  = s2t even S .
end
```

In converting a tree to a stream, `t2s` first outputs its root, and then interleaves streams from its left and right subtrees, as illustrated in the following:



To build a tree from a stream, `s2t` uses the head of the stream as root, and uses the stream got by selecting all elements at odd positions except the first to build the right subtree, and all elements at even positions to create the left subtree. In the following, two behavioral properties are first proved and then introduced as lemmas for proving that `s2t` and `t2s` are inverse operations:

```
open .
  vars S S1 S2 : Stream .
  var T : Tree .
  cred even zip(S1, S2) == S2 .
  cred zip(even S, even tail S) == tail S .
  cred even t2s T == t2s left T .
  eq even zip(S1, S2) = S2 .             *** lemma
  eq zip(even S, even tail S) = tail S . *** lemma
  eq even t2s T = t2s left T .           *** lemma
  cred s2t t2s T == T .
  cred t2s s2t S == S .
close
```

The following defines functions `f` and `g` on trees and streams respectively, which will turn out to be identity functions. For any tree `T`, `left(f(T))` is defined by first taking the left subtree of `T`, and then converting it to a stream, and then applying `g` to the stream (since `g` is an identity function, this will be just the original stream), and then transforming the stream back to the tree, and finally applying `f` to this tree. Other cases are similar.

```
bth SETUP [ X :: TRIV ] is pr TREE-STREAM [X] .
  op f_ : Tree -> Tree .
  op g_ : Stream -> Stream .
  var T : Tree . var S : Stream .
  eq root f T  = root T .
  eq left f T  = f s2t g t2s left T .
  eq right f T = f s2t g t2s right T .
  eq head g S  = head S .
  eq tail g S  = g t2s f s2t tail S .
end
```

This way of defining f and g are not so unusual in practice. Sometimes it is hard to define an operation on a given sort directly, but we can transform it to an element on another sort, then use the operations on that sort, and finally transform the result back to the original sort. Now we prove f and g are identity functions in the following:

```
set cobasis of TREE-STREAM .
open .
  eq s2t t2s T = T .
  eq t2s s2t S = S .
  cred ( f T == T ) ( g S == S ) .
close
```

This produces the following BOBJ output:

```
c-reduce in SETUP :
   f T == T
   g S == S
using cobasis of SETUP:
   op head _ : Stream -> Elt [prec 15]
   op root _ : Tree -> Elt
   op left _ : Tree -> Tree
   op right _ : Tree -> Tree
   op tail _ : Stream -> Stream [prec 15]
---------------------------------------
handled: f T == T
reduced to: f T == T
add rule (C1) : f T = T
-----------------------------------------
handled: g S == S
reduced to: g S == S
add rule (C2) : g S = S
-----------------------------------------
target is: f T == T
expand by: op root _ : Tree -> Elt
reduced to: true
     nf: root T
-----------------------------------------
target is: f T == T
expand by: op left _ : Tree -> Tree
deduced using (C1, C2) : true
     nf: left T
-----------------------------------------
target is: f T == T
expand by: op right _ : Tree -> Tree
deduced using (C1, C2) : true
     nf: right T
```

```
----------------------------------------
target is: g S == S
expand by: op head _ : Stream -> Elt [prec 15]
reduced to: true
     nf: head S
----------------------------------------
target is: g S == S
expand by: op tail _ : Stream -> Stream [prec 15]
deduced using (C2, C1) : true
     nf: tail S
----------------------------------------
result: true
c-rewrite time: 327ms     parse time: 3ms
```

In this proof, two new "circularity" rules, C1 and C2, are added in the first two steps. The third step expands the goal `f(T) == T` by using `root`, and then the new goal is proved by behavioral rewriting directly. The next step gets a new goal by expanding the same goal using `left`, and the output above shows that both sides of this new goal reduce to `left(T)` by behavioral rewriting. The following shows the steps of this proof, using both C1 and C2:

```
      left f T               (using left f T = f s2t g t2s left T )
  ⟶  f s2t g t2s left T    (using C1, f T = T )
  ⟶  s2t g t2s left T      (using C2, g S = S )
  ⟶  s2t t2s left T        (using s2t t2s T = T )
  ⟶  left T
```

The circularity `g S = S` which is used in the proof could not be proved if `f T == T` were the only coinductive goal.

Example 9 Fibonacci and Other Streams We can define a generalized Fibonacci function by

$$fib(n+2) \quad = \quad fib(n+1) + fib(n)$$

where $fib(0)$ and $fib(1)$ may be given any values. Using `STREAM` as defined above, the following defines a stream `fib` of generalized Fibonacci numbers:

```
bth FIBO-STREAM is pr STREAM[NAT] .
  vars M N : Nat .
  op fib : Nat Nat -> Stream .
  eq head fib(M, N) = M .
  eq tail fib(M, N) = fib(N, M + N) .
end
```

Note that for any natural numbers `M` and `N`, the first and second elements of the stream `fib(M, N)` are the first and second arguments of `fib`. All other elements in the stream `fib(M, N)` are the sums of the two prior elements. Now suppose the function g is defined on natural numbers by:

$$\begin{array}{lcll} g(n+2) & = & g(n+1) + g(n) & \text{if } n \text{ is even} \\ g(n+2) & = & g(n-1) & \text{if } n \text{ is odd} \end{array}$$

where $g(0)$ and $g(1)$ can again be given any values. The following defines streams of these numbers:

```
bth G-STREAM is pr ODD-EVEN[NAT] .
  vars M N : Nat .
  op g : Nat Nat -> Stream .
  eq head g(M, N) = M .
  eq head tail g(M, N) = N .
```

```
      eq head tail tail g(M, N) = M + N .
      eq head tail tail tail g(M, N) = M .
      eq tail tail tail tail g(M, N) = g(M + M + N, M + N) .
   end
```

If $fib(0) = g(0) = M$ and $fib(1) = g(1) = N$, then the property $fib(n+2) = g(2n+1)$ can be expressed in the query

```
    tail tail fib(M,N) == tail odd g(N, M) .
```

However, this cannot be proved directly, because it generates infinitely many new proof tasks. Moreover, the property only covers the situation where g is applied to odd natural numbers, so we need a property covers even natural numbers, namely $fib(n) = g(2n)$. These two properties each need the other, and though neither can be proved by itself, they can be proved together by the following:

```
   open FIBO-STREAM + G-STREAM .
     vars M N : Nat .
     cred ( tail tail fib(M, N) == tail odd g(N, M) )
          ( fib(M, N) == even g(N, M) ) .
   close
```

□

5 Conclusions

Many theoretical and practical innovations have developed with the OBJ family of languages, some of which are listed below. We feel that this strongly supports the view that theory and practice should be pursued together, since each raises new ideas for the other, and each can test and support the validity of the other. Sometimes we implemented things that we did not yet have theory for, sometimes we puzzled over how to implement an existing theory, and often we struggled to extend both theory and implementation to cover some phenomenon of practical interest. Certainly both higher order parameterization and mutual coinduction fall into this area. Some other innovations, most of which have already been discussed somewhere in this paper, are: overloaded many sorted algebra, order sorted algebra, retracts, membership equational logic (in Maude [56]), hidden algebra, circular coinductive rewriting, parameterized programming, higher order parameterized modules, institutions, efficient term rewriting modulo equations (in Maude [13]), Grothendieck institutions (for the semantics of CafeOBJ [14]), initial algebra (and model) semantics, e.g. for constraint logic programming [42], and reflective equational programming (in Maude [17]). Many of these seem underexploited, and since the intellectual mines of OBJ are probably not yet exhausted, we may perhaps expect to see more interesting or unusual logical contributions from this area of research in the future.

References

[1] David Aspinall. Type checking parameterised programs and specifications in ASL+FPC. In Martin Wirsing, Dirk Pattinson, and Rolf Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT'02*, pages 129–144. Springer, Lecture Notes in Computer Science, volume 2755, 2003.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge, 1998.

[3] Narjes Berregeb, Adel Bouhoula, and Michaël Rusinowitch. Observational proofs with critical contexts. In *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 38–53. Springer, 1998.

[4] Michel Bidoit and Rolf Hennicker. Constructor-based observational logic. Technical Report LSV–03–9, Laboratoire Spcification et Verification, CNRS de Cachan, March 2003.

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

[6] 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.

[7] Rod Burstall and Joseph Goguen. An informal introduction to specifications using Clear. In Robert Boyer and J Moore, editors, *The Correctness Problem in Computer Science*, pages 185–213. Academic, 1981. Reprinted in *Software Specification Techniques*, Narain Gehani and Andrew McGettrick, editors, Addison-Wesley, 1985, pages 363–390.

[8] Samuel Buss and Grigore Roşu. Incompleteness of behavioral logics. In Horst Reichel, editor, *Proceedings, Coalgebraic Methods in Computer Science (CMCS'00)*, volume 33 of *Electronic Notes in Theoretical Computer Science*, pages 61–79. Elsevier Science, March 2000.

[9] Virgil Emil Căzănescu and Grigore Roşu. Weak inclusion systems. *Mathematical Structures in Computer Science*, 7(2), 1997.

[10] Virgil Emil Căzănescu and Grigore Roşu. Weak inclusion systems, part 2. *Journal of Universal Computer Science*, 6(1):5–21, 2000.

[11] Maura Cerioli, Till Mossakowski, and Horst Reichel. From total equational to partial first order. In *State of the Art in Algebraic Specification*. IFIP, to appear 1998. Chapter 3.

[12] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[13] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In José Meseguer, editor, *Proceedings, First International Workshop on Rewriting Logic and its Applications*. Elsevier Science, 1996. Volume 4, *Electronic Notes in Theoretical Computer Science*.

[14] Răzvan Diaconescu. Grothendieck institutions. *Applied Categorical Structures*, 10:383–402, 2002.

[15] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. World Scientific, 1998. AMAST Series in Computing, Volume 6.

[16] Răzvan Diaconescu and Kokichi Futatsugi. Behavioural coherence in object-oriented algebraic specification. *Journal of Universal Computer Science*, 6(1):74–96, 2000.

[17] Francisco Duran and José Meseguer. Structured theories and institutions. *Theoretical Computer Science*, 309:357–380, 2003.

[18] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*. Springer, 1990. EATCS Monographs on Theoretical Computer Science, Vol. 21.

[19] José Fiadeiro, Neil Harman, Markus Roggenbach, and Jan Rutten (Eds.). *Algebra and Coalgebra in Computer Science*. Springer, 2005. Lecture Notes in Computer Science, vol. 3629.

[20] José Luiz Fiadeiro. *Categories for Software Engineering*. Springer, 2004.

[21] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Proceedings, Twelfth ACM Symposium on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[22] Kokichi Futatsugi, Joseph Goguen, and Kazuhiro Ogata. Verifying design with proof scores. In Bertrand Meyer, editor, *Proceedings, Verified Software: Theories, Tools, Experiments.* Springer, 2005. Workshop celebrating 150th anniversary of Eidgenoschische Hochschule Zürich, 10-13 October 2005.

[23] Steven Garland. LP – the Larch Prover: version 3.1b, 1999. MIT, Laboratory for Computer Science, `http://www.sds.lcs.mit.edu/larch/LP`.

[24] Marie-Claude Gaudel and Igor Privara. Context induction: an exercise. Technical Report 687, LRI, Université de Paris-Sud, 1991.

[25] Joseph Goguen. Mathematical representation of hierarchically organized systems. In E. Attinger, editor, *Global Systems Dynamics*, pages 112–128. S. Karger, 1971.

[26] Joseph Goguen. Abstract errors for abstract data types. In Eric Neuhold, editor, *Proceedings, First IFIP Working Conference on Formal Description of Programming Concepts*, pages 21.1–21.32. MIT, 1977. Also in *Formal Description of Programming Concepts*, Peter Neuhold, Ed., North-Holland, pages 491–522, 1979.

[27] 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.

[28] 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.

[29] 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. University of Texas at Austin Year of Programming Series; preliminary version in SRI Technical Report SRI-CSL-88-1, January 1988.

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

[31] 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.

[32] Joseph Goguen and Susanna Ginali. A categorical approach to general systems theory. In George Klir, editor, *Applied General Systems Research*, pages 257–270. Plenum, 1978.

[33] Joseph Goguen and Kai Lin. Web-based support for cooperative software engineering. *Annals of Software Engineering*, 12:25–32, 2001.

[34] Joseph Goguen and Kai Lin. Behavioral verification of distributed concurrent systems with BOBJ. In Hans-Dieter Ehrich and T.H. Tse, editors, *Proceedings, Conference on Quality Software*, pages 216–235. IEEE Press, 2003.

[35] Joseph Goguen, Kai Lin, and Grigore Roşu. Circular coinductive rewriting. In *Automated Software Engineering '00*, pages 123–131. IEEE, 2000. Proceedings of a workshop held in Grenoble, France.

[36] Joseph Goguen, Kai Lin, and Grigore Roşu. Behavioral and coinductive rewriting. In *Proceedings, Rewriting Logic Workshop, 2000.* Elsevier, 2001. Electronic Notes on Theoretical Computer Science, Volume 36, at `www.elsevier.nl/locate/entcs/volume36.html`.

[37] Joseph Goguen and Grant Malcolm. Hidden coinduction: Behavioral correctness proofs for objects. *Mathematical Structures in Computer Science*, 9(3):287–319, June 1999.

[38] Joseph Goguen and Grant Malcolm. More higher order programming in OBJ3. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, pages 397–408. Kluwer, 2000.

[39] Joseph Goguen and Grant Malcolm. A hidden agenda. *Theoretical Computer Science*, 245(1):55–101, August 2000. Also UCSD Dept. Computer Science & Eng. Technical Report CS97–538, May 1997.

[40] Joseph Goguen and José Meseguer. Rapid prototyping in the OBJ executable specification language. *Software Engineering Notes*, 7(5):75–84, December 1982. Proceedings of Rapid Prototyping Workshop.

[41] Joseph Goguen and José Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11(3):307–334, 1985.

[42] 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.

[43] 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 exist from as early as 1985.

[44] Joseph Goguen and Grigore Roşu. Hiding more of hidden algebra. In Jeannette Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 – Formal Methods*, pages 1704–1719. Springer, 1999. Lecture Notes in Computer Sciences, Volume 1709, Proceedings of World Congress on Formal Methods, Toulouse, France.

[45] Joseph Goguen, Grigore Roşu, and Kai Lin. Conditional circular coinductive rewriting with case analysis. In Martin Wirsing, Dirk Pattinson, and Rolf Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT'02*, pages 216–232. Springer, Lecture Notes in Computer Science, volume 2755, 2003.

[46] Joseph Goguen and Joseph Tardo. An introduction to OBJ: A language for writing and testing software specifications. In Marvin Zelkowitz, editor, *Specification of Reliable Software*, pages 170–189. IEEE, 1979. Reprinted in *Software Specification Techniques*, Nehan Gehani and Andrew McGettrick, editors, Addison Wesley, 1985, pages 391–420.

[47] Joseph Goguen and William Tracz. An implementation-oriented semantics for module composition. In Gary Leavens and Murali Sitaraman, editors, *Foundations of Component-based Systems*, pages 231–263. Cambridge, 2000.

[48] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer, 2000.

[49] John Gray. The category of sketches as a model for algebraic semantics. In John Gray and Andrezj Scedrov, editors, *Categories in computer science and logic*, volume 92 of *Contemporary Mathematics*, pages 109–135. American Mathematical Society, 1989.

[50] 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.

[51] Rolf Hennicker and Michel Bidoit. Observational logic. In *Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 1999.

[52] Paul Hudak, Simon Peyton Jones, Philip Wadler, Arvind, et al. Report on the functional programming language Haskell. *ACM SIGPLAN Notices*, 27, May 1992. Version 1.2.

[53] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, June 1997.

[54] John Wilcox Lloyd. *Foundations of Logic Programming.* Springer, 1987. Second edition.

[55] José Meseguer. General logics. In H.-DĖbbinghaus et al., editors, *Proceedings, Logic Colloquium 1987*, pages 275–329. North-Holland, 1989.

[56] José Meseguer. Membership algebra as a logical framework for equational specification. In Francisco Parisi-Presicce, editor, *Proceedings, WADT'97 – Workshop on Abstract Data Types*, pages 18–61. Springer, 1998. Lecture Notes in Computer Science, Volume 1376.

[57] José Meseguer and Joseph Goguen. Order-sorted algebra solves the constructor selector, multiple representation and coercion problems. *Information and Computation*, 103(1):114–158, March 1993. Revision of a paper presented at LICS 1987.

[58] Robin Milner. *A Calculus of Communicating Systems.* Springer, 1980. Lecture Notes in Computer Science, Volume 92.

[59] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT, 1997.

[60] Till Mossakowski, Joseph Goguen, Razvan Diaconescu, and Andrzej Tarlecki. What is a logic? In Jean-Yves Beziau, editor, *Logica Universalis*, pages 113–133. Birkhauser, 2005. Selected papers from the First World Conference on Universal Logic.

[61] Peter Mosses, editor. *CASL Reference Manual.* Springer, 2004. Lecture Notes in Computer Science, Volume 2960.

[62] David M.R. Park. *Concurrency and Automata on Infinite Sequences.* Springer, 1980. Lecture Notes in Computer Science, Volume 104.

[63] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* MIT, 1991.

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

[65] 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.

[66] Grigore Roşu. *Hidden Logic.* PhD thesis, University of California at San Diego, 2000.

[67] Grigore Roşu and Joseph Goguen. Hidden congruent deduction. In Ricardo Caferra and Gernot Salzer, editors, *Proceedings, 1998 Workshop on First Order Theorem Proving*, pages 213–223. Technische Universität Wien, 1998. (Schloss Wilhelminenberg, Vienna, November 23-25, 1998).

[68] Grigore Roşu and Joseph Goguen. Circular coinduction. In *Proceedings, Int. Joint Conf. Automated Deduction.* Springer, 2000. Sienna, June 2001.

[69] Donald Sannella, Stefan Sokolowski, and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. *Acta Informatica*, 29:689–736, 1992.

[70] Donald Sannella and Andrzej Tarlecki. Extended ML: an institution independent framework for formal program development. In David Pitt, Samson Abramsky, Axel Poigné, and David Rydeheard, editors, *Proceedings, Summer Workshop on Category Theory and Computer Programming*, pages 364–389. Springer, 1986. Lecture Notes in Computer Science, Volume 240.

[71] William Tracz. LILEANNA: a parameterized programming language. In *Proceedings, Second International Workshop on Software Reuse*, pages 66–78, March 1993. Lucca, Italy.

[72] Jeffrey Ullman. *Elements of ML Programming.* Prentice Hall, 1998.

[73] Alfred North Whitehead. *A Treatise on Universal Algebra, with Applications, I.* Cambridge, 1898. Reprinted 1960.

[74] Hirokazu Yatsu and Kokichi Futatsugi. Modular specification in CafeOBJ. In Kokichi Futatsugi, editor, *Proceedings, Workshop on Tenth Anniversary of OBJ2, Itoh, Japan, 21–22 October 1995.* Unisys, 1995.