ICOT Technical Report: TR-097

TR-097

Principles of OBJ2

Kokichi Futatsugi (ETL),
Joseph A. Goguen, Jean-Pierre Jouannaud
(SRI International)
and José Meseguer (Stanford Univ.)

December, 1984

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

Table of Contents

Principles of OBJ2[1]

Kokichi Futatsugi[2], Joseph A. Goguen, Jean-Pierre Jouannaud[3], and José Meseguer

SRI International, Menlo Park CA 94025
and
Center for the Study of Language and Information, Stanford University 94305

1 Introduction

OBJ2 is a functional programming language with an underlying formal semantics that is based upon equational logic, and an operational semantics that is based upon rewrite rules. Four classes of design principles for OBJ2 are discussed briefly in this introduction, and then in more detail below: (1) modularization and parameterization; (2) subsorts; (3) implementation techniques; and (4) interaction and flexibility. We also trace OBJ history, current status, and future plans, and give a fairly complete OBJ bibliography. Most example code has actually been run on our current OBJ2 interpreter.

1.1 Modules and Generics

A key OBJ2 principle is the systematic use of parameterized (generic) modules. Encapsulating related code makes it more reusable, and generics are even more reusable, since they can be "tuned" for a variety of applications by choosing different parameter values; moreover, debugging, maintenance, readability and portability are all enhanced. The interface declarations of OBJ2 generics are not purely syntactic, like Ada's[4]; instead, they may contain semantic requirements that actual modules must satisfy before they can be meaningfully substituted. This can prevent many subtle bugs. An unusual feature of OBJ2 is the commands that it provides for modifying and combining program modules; thus, (a form of) program transformation is provided within the language itself. A key principle here is the systematic use of **module expressions** for describing and creating complex combinations of modules; see

[1]Supported in part by Office of Naval Research Contract No. N00014-82-C-0333, National Science Foundation Grant No. MCS8201380 and a gift from the System Development Foundation.

[2]Electrotechnical Laboratory, 1-1-4 Umezono, Sakura, Niihari, Ibaraki 305, Japan; work performed while on leave at SRI.

[3]CRIN, Campus Scientifique, BP 239, 54506 Vandoeuvre-les-Nancy Cedex, France; work performed while on leave at SRI and CSLI.

[4]Ada is a registered trademark of the U.S. Government (Ada Joint Program Office) and is defined by [DoD 83].

Section 2.5. This provides a level above that of conventional programming languages, in which previously written code is described by theories, and also is manipulated by module expressions to produce new code.

1.2 Subsorts

OBJ2 is strongly typed[5]. As in many other languages, users can introduce their own sorts and operations, thus supporting user defined ADTs. However, OBJ2 also permits users to declare that some sorts contain (or are contained in) others. This supports a simple yet powerful form of polymorphism, as well as exception (error) definition and handling (recovery), partially defined operations, and multiple inheritance (in the sense of object-oriented programming); see Section 3. In addition, it permits us to write very simple and elegant code for standard ADTs like Lists, Stacks and Tables, as shown below.

1.3 Implementation Techniques

The current implementation is a Maclisp interpreter employing several novel techniques for efficient term rewriting, including forms of tail recursion, hash-coded structure sharing (see Section 4.4), and user defined built-ins (see Section 4.5). The task of implementing OBJ2 has been greatly eased by our Command Interpreter Generator (CIG), essentially a powerful application generator for interactive menu-driven systems; see Section 4.1. The CIG allows a strict separation of OBJ2's high level interactive syntax from the functions that actually do the work, and also helps bring up prototypes very rapidly, facilitating experiments with OBJ2's syntax and semantics without having to reimplement a complex interactive system. An interesting methodological point is our use of OBJ to design the new implementation. We specified the database and the rewrite rule engine in both OBJ1 and OBJ2, and we also specified the interface between these two major components (but not in OBJ2). It was instructive to debug our OBJ2 design on a real example. It seems doubtful that we could have implemented such a complex and novel system in one year without using such techniques.

[5]Hereafter we generally use the word "sort" instead of "type", trying to avoid confusion among the many different ways that "type" has been used (major exceptions are use of the word "typechecking" and of the phrase "abstract data type", abbreviated ADT).

1.4 Interaction and Flexibility

OBJ2 is highly interactive and flexible. Program entry is driven by structured menu choice, rather than by keyword entry and subsequent parsing. All syntactic and many semantic errors are detected at program entry time; the user is then given a diagnostic message and a chance to try again. The OBJ2 interpreter provides much more help than most compilers even attempt. Implementing this principle is greatly aided by the CIG.

Users can define their own abstractions, with any desired syntax, and then use these abstractions as if they were built in; in fact, users can even give efficient implementations in the underlying Lisp. OBJ's "mixfix" (or "distfix") syntax permits prefix, postfix, infix, "outfix" (as in `{ x }` or `| A |`), and most generally, distributed fix operations with keywords and arguments in any desired order (`if_then_else_fi` is non-trivially mixfix). In addition to efficient implementation, user definable built-ins can also provide sophisticated I/O packages, such as window management, if the underlying hardware supports it. (Section 4.5 gives more detail on built-ins.)

2 Modules and Generics

The only top level OBJ2 entities are **modules** (which are either objects or theories) and **views** (which relate theories to modules, see Section 2.4); **objects** contain executable code, while **theories** contain nonexecutable assertions. Thus, executable code and nonexecutable assertions are both modularized, and are closely integrated with each other. Moreover, the mathematical semantics of theories and objects is elegantly unified since both are **data theories** in the sense of [Goguen & Burstall 84]; there is not space for details here, but the essential intuition is that the notion of theory generalizes to permit regarding certain subtheories as objects (i.e., they have initial interpretations) while others may have any interpretation.

2.1 Objects: Syntax and Semantics

OBJ2's basic entity is the **object**, which is a module (possibly parameterized) encapsulating executable code; objects generally introduce new sorts of data and new operations upon that data. An object has three main parts: (1) a **header**, containing its name, parameters, interface requirements, and imported module list; (2) a **signature**, declaring its new sorts, subsort relationships, and operations; and (3) a **body**, containing its code, consisting of equations and sort constraints (these are described in Section 3.1). The following `BITS` object introduces two new sorts `Bit` and `Bits` (for `Bit` lists) with some relevant operations; these lists have a S-expression-like syntax with `.` and `nil`. OBJ2 keywords and keysymbols are in *italics*.

```
obj BITS is
  extending NAT .
  sorts Bit Bits .
  ops 0 1 : -> Bit .
  op nil : -> Bits .
  op _._ : Bit Bits -> Bits .
  op length_ : Bits -> Nat .
  var B : Bit .
  var S : Bits .
  eq : length nil = 0 .
  eq : length B . S = inc length S .
endo
```

Each line in this example begins with an OBJ2 keyword. The first line with keyword *obj* gives the name **BITS** of the object. The second line indicates that the built-in object **NAT** (for natural numbers) is imported by **BITS**. The keyword "*extending*" indicates that a certain static check is performed (see Section 2.2 for more detail). The *ops* line declares the two Bit constants, the next *op* line declares the empty string `nil`, the next a "cons" operation (it adds a Bit to some Bits) with "dot" syntax, and the next a length operation using the sort **Nat** from **NAT**. An operation declaration in OBJ2 consists of: a **form**, indicating the distribution of keywords and arguments (underbars indicate argument places); an **arity**, which lists argument sorts; and a **coarity**, which is the output sort. Finally, variables of sorts **Bit** and **Bits** are declared and used in equations which give semantics for the **length** function.

OBJ2's basic user-level naming conventions are as follows: modules must have globally unique names; sort names must be unique within their module, and may be qualified by a module name for disambiguation; an operation name consists of its form, arity and coarity, and (in a future implementation) may be qualified by a module name; note that sort names used in arity and coarity can be qualified by module.

OBJ2 code is executed by interpreting equations as **rewrite rules**: the lefthand side is regarded as a pattern to be matched within an expression (variables can match any subexpression of appropriate sort); when a match occurs, the subexpression matching the lefthand is rewritten to the corresponding substitution instance of the righthand side. This process continues until there are no more matches; then the expression is said to be **reduced** or **in normal form**. Writing

```
eval length 329 . 666 . nil ende
```

at OBJ2's top level causes calculation of the reduced form **2** using the rules in **BITS**.

The Church-Rosser and termination properties imply that expressions will always have unique reduced forms; we have found that experienced programmers usually write rules that satisfy these properties. Term rewriting may seem a specialized computational paradigm, but in fact it is completely general: any computable function over any computable data types can be so realized [Bergstra & Tucker 80a, Bergstra & Tucker 80b]. This generalizes to operations that are associative, commutative, idempotent and/or have an identity [Jouannaud & Kirchner 84]; then implementation is by matching modulo the given operation attributes.

OBJ2's mathematical semantics is so-called initial algebra semantics (IAS) [Goguen, Thatcher & Wagner 78]. IAS provides a standard interpretation characterized by the properties of: (1) having "no junk", meaning that all data values are denoted by terms in the available operation symbols; and (2) having "no confusion", meaning that two terms denote the same data value if and only if they can be proved equal (with standard equational reasoning) from the given equations [Burstall & Goguen 82].

2.2 Hierarchy of Modules

OBJ2 modules can import other modules in three different ways, "*using*", "*protecting*" and "*extending*". These define three different restrictions on preserved properties of imported modules, and thus define three corresponding partial orders (i.e., hierarchies) among modules. The *using* hierarchy is the most general, and embeds the other two.

An importation may fail to preserve the "no confusion" property for the data elements of the imported module. For example, if we define the integers modulo 3 by importing the integers (defined with `0` and successor `s`) by adding the equation `s s s 0 = 0`, then the integer data elements become confused. Similarly, the "no junk" property may be violated by operations that create new data elements of imported sorts. For example, a `Bool`-valued operation `p` can create "Boolean junk" like `p(97)` if a user gives incomplete equations for `p`. "*Protecting*" is the most restrictive relation, indicating that both the "no confusion" and "no junk" properties are preserved, and thus the imported module remains unchanged. "*Extending*" is an easy-to-check sufficient condition for "no confusion", requiring that the operations defined in an imported module do not occur as topmost symbols on the lefthand side of a new equation; it can only be used for objects; and it supports separate compilation, since a given operation is defined once and for all by the module that declares it, and can therefore be computed from just the information in that module. "*using*" does not guarantee anything, it just copies the imported module.

2.3 Parameterized Objects and Theories

Modules group together the data and operations used for particular problems such as sorting, matrix manipulation, or graphics. Parameterized modules maximize reusability by permitting "tuning" to fit a variety of applications. For example, using (italicized) square brackets to separate the module name from parameters, SORTING*[*X*]* might sort lists over any ordered set X, and MATRIX*[*n,R*]* might provide the usual $n \times n$ matrix operations for scalars from R. The parameter X of SORTING ranges over (partially) ordered sets, i.e., sets with an irreflexive (i.e., X < X is false) transitive relation. The parameter n of MATRIX ranges over natural numbers, while R ranges over rings[6]. Thus, OBJ2 supports *semantic* interface requirements, whereas Ada's purely syntactic interfaces cannot exclude actuals that would produce unexpected or bizarre behavior. An OBJ2 module interface is described by **a requirement theory**, giving both syntax and axioms for the interface, with an object being an admissible actual only if it satisfies the axioms. The requirement theory for SORTING is given by

```
th POSET is
  protecting BOOL .
  sort Elt .
  op _<_ : Elt Elt -> Bool .
  vars E E' E'' : Elt .
  eq : E < E = false .
  ceq : E < E'' = true if (E < E' and E' < E'') .
endth
```

(OBJ2 code for SORTING is given in Section 2.5 below.) More simply, here is the requirement theory for an interface that only requires designating a sort from an actual (with no axioms):

```
th TRIV is
  sort Elt .
endth
```

A parameterized object may have one or more requirement theories; these are given in (italicized) square brackets after its name. Since more than one parameter may be subject to the same requirements, different instances of the same theory may be needed. For example, the following TABLE object has two TRIV requirement theories, INDEX and VAL, with Elt.INDEX and Elt.VAL their corresponding sorts, thus illustrating OBJ2's qualified sort name convention:

```
obj TABLE[INDEX :: TRIV, VAL :: TRIV] is
  protecting BOOL .
```

[6]Values with addition, difference, multiplication, and constants 0, 1, such that addition and multiplication are associative and commutative with identities 0, 1 respectively, difference is inverse to addition, and multiplication distributes over addition.

```
    sorts Table ErrVal .
    subsorts Elt.VAL < ErrVal .
    op empty : -> Table .
    op put : Elt.VAL Elt.INDEX Table -> Table .
    op _[_] : Table Elt.INDEX -> ErrVal .
    op undef : Elt.INDEX -> ErrVal .
    vars I I' : Elt.INDEX .
    var V : Elt.VAL .
    var T : Table .
    eq : put(V,I,T)[ I' ] = if I == I' then V else T [ I' ] fi .
    eq : empty [ I ] = undef(I) .
  jbo
```

In the first equation, "==" is a built-in equality such that for two terms t, t′ of the same sort, t == t′ is **true** if they have the same reduced form, and is **false** otherwise. == implements the decision procedure for equality that is associated with every object, and **if_then_else_fi** is a polymorphic conditional; both these are provided for every sort by the built-in BOOL object. The supersort ErrVal of Elt.VAL accomodates error messages for lookups of an index where no value is stored (Section 3 explains sub- and super- sorts). The form of the operation **put** has no underbars, and therefore gets a standard parentheses-with-commas syntax, e.g., **put(true,13,T1)** for TABLE[INT,BOOL].

2.4 Views

Instantiating a parameterized object means providing actual objects satisfying each of its requirement theories[7]. In OBJ2, the actual objects are provided through **views**, which *bind* required sorts and operations to those actually provided (i.e., views map the sort and operation symbols in the formal requirement theory to those in the actual object[8]) in such a way that all axioms of the requirement theory are satisfied. For example,

```
  view INT-DESC of INT as POSET is
    sort Elt to Int .
    vars X Y : Elt .
    op : X < Y to : Y < X .
  endview
```

views INT with descending order as a poset; in "*op* : X < Y *to* : Y < X" the first "<" is the one from POSET while the second "<" is from INT. Instantiating a parameterized object with a view

[7]Parameterized objects can also be seen as "object generators."

[8]More generally, a view can map an operation in the formal to an expression in the actual, as illustrated in the following example.

is indicated by replacing the parameter part with the view name. Thus, instantiating SORTING[P :: POSET] to sort integers in descending order is indicated by SORTING[INT-DESC]. Note that there can be more than one view from a theory to an actual; for example, another instantiation of SORTING with INT uses a view that maps "<" in POSET to the (irreflexive) divisibility relation in INT.

Sometimes a view can be inferred from a partial description, using syntactic similarities between the theory and the object. We call such a view description an **abbreviated view**; it is a **default view** if abbreviated to nothing [Goguen 84]. For example, there is a default view of INT as POSET that maps Elt to INT and _<_ : Elt Elt -> Bool to the usual ordering on integers. To instantiate a parameterized object with a default view, it suffices to place the actual in the parameter part. Thus SORTING[INT] sorts integers in ascending order; at the top level of OBJ2, one might write

make SORTING-INT *is* SORTING[INT] *endm*

to create SORTING-INT. Intuitively, applying a parameterized object corresponds to "editing" its text according to a view; but this is highly disciplined edit, with a formal semantics like that of Clear [Burstall & Goguen 77, Burstall & Goguen 80]. In this semantics, views correspond to theory morphisms, and parameterized objects have an associated theory inclusion from their requirement theory to their body. The pushout of that theory inclusion along the view gives a new theory whose initial algebra is the desired instantiation (see [Mac Lane 71] or [Goguen & Burstall 84] for the definition of pushout and [Goguen & Burstall 84] for more detail on instantiating parameterized objects).

2.5 Module Expressions

Parameterized programming permits building complex modules from simple ones by applying parameterized objects to actuals that are themselves instantiations of other parameterized objects and so on recursively; a similar approach using module construction operations occurs in OBJT, OBJ1 and the Hisp specification language [Futatsugi & Okada 82, Futatsugi & Okada 80]. OBJ2 **module expressions** are like ordinary arithmetic expressions, except that their arguments are modules rather than numbers. For example, if ID is a built-in identifier object, and if our library contains LEX[X :: POSET] providing lexicographic ordering for lists of elements from an ordered set X, and also contains a parameterized sorting object SORTING[P :: POSET], then the module expression SORTING[LEX[ID]] lexicographically sorts lists of phrases (i.e., lists of lists of identifiers). This expression uses several default conventions. In LEX[ID], a default view maps the sort of LEX's requirement theory to the sort Id and maps its ordering to the built-in ordering for identifiers. Another default view maps the sort Elt of SORTING's requirement theory to the List sort of LEX[ID], and maps its ordering to the lexicographic ordering from LEX[ID]. Here is the code for SORTING (the parameterized LIST module is given in Section 3.1):

```
obj SORTING [ELT :: POSET] is
  extending LIST[ELT] .
  op sorted : List -> Bool .
  op sort : List -> List .
  vars E E' : Elt .
  var L L' L'' : List .
  eq : sorted(nil) = true .
  eq : sorted(E) = true .
  eq : sorted(E E' L) = (E == E' or E < E') and sorted(E' L) .
  ceq : sort(L E L' E' L'') = sort(L E' L' E L'') if E' < E .
  ceq : sort(L) = L if sorted(L) .
jbo
```

(At the time of writing, OBJ2's module instantiation facility was not quite up to this and the following example; but we expect it will be by the time the paper is presented.)

If M is a module expression, then M * (...) is another, with sorts and operations renamed according to (...). For example, we rename the operation sort of SORTING to lex-sort in the module expression SORTING[LEX[ID]] * (op sort to lex-sort). Module expressions also occur in "definitions" inside of modules, with the effect of renaming the principal sort of the module expression, as in

```
obj SYMBOLTABLE is
  extending ID .
  extending INT .
  dfn Env := STACK[TABLE[ID.INT] * (sort Int to Loc, sort Table to Layer)] .
jbo
```

(The **principal sort** of a module is the first new sort introduced in it; or if there is none, the principal sort of its first imported module, and so on recursively.) Replacing the "*dfn*" by "*extending*" and renaming of sort Stack to Env would yield the same operational semantics and hierarchy of modules.

3 Subsorts

One sort of data is often contained in (or contains) another, e.g., the natural numbers are contained in the integers, which are contained in the rationals; then the sort Nat is a **subsort** of Int, and Int is a subsort of Rat (or Int is a **supersort** of Nat, etc.), written Nat $<$ Int $<$ Rat. Moreover, an operation may restrict to subsorts of its arity and coarity and still be "the same" operation. For example, each addition operation `_+_ : Rat Rat -> Rat`, `_+_ : Int Int -> Int`, `_+_ : Nat Nat -> Nat` is a restriction of the preceding one. OBJ2's very flexible subsort mechanism provides the following, all within the framework of an initial algebra semantics:

1. Such overloaded operations provide a simple but powerful polymorphism[9] (see Sections 3.2 and 4.2).
2. Multiple inheritance in the sense of object-oriented programming permits one sort to be a subsort of two (or more) others, each having various defined operations; then all these operations are inherited by the subsort. For example, we might have **RegisteredVehicle** < **Vehicle** and **RegisteredVehicle** < **TaxedObject**, with say a **speed** operation on **Vehicles** and a **tax-amount** operation on **TaxedObjects**; both these operations are inherited by items of sort **RegisteredVehicle**.
3. The familiar difficulties for ADTs with operations that are "partial" (such as **tail** for lists and **push** for bounded stacks) disappear by viewing the operations as total on the right subsorts (see Sections 3.1 and 3.2).
4. Errors can be treated in several styles, without need for special syntactic or semantic "error handling" mechanisms (see Sections 3.3 and 3.1).

3.1 Partial Operations and Sort Constraints

The following specification for a parameterized LIST object introduces a subsort **NeList** of nonempty lists to make the (traditionally partial) head and tail operations total. Here "*assoc*" indicates that an operation is associative, and "*id:* **nil**" indicates that it has **nil** as an identity.

```
obj LIST[X :: TRIV] is
  sorts NeList List .
  subsorts Elt < NeList < List .
  op nil : -> List .
  op __ : NeList NeList -> NeList [assoc]
  op __ : List List -> List [assoc id: nil]
  op head : NeList -> Elt .
  op tail : NeList -> List .
  var L : NeList .
  var E : Elt .
  eq : head(E L) = E .
  eq : tail(E L) = L .
jbo
```

[9]Parameterized objects provide yet another polymophism, since a generic object's operations are available to all its instances.

A more subtle kind of partial operation is illustrated by "**push**" in a bounded stack. Unlike LIST, where nonempty lists are generated by concatenation, the stacks for which "**push**" is ok have no natural expression as a set generated by constructors (e.g., by **push** itself); but they can be characterized by the equational condition of having a length not exceeding the bound. Such characterizations are called **sort constraints** in OBJ2 (the word "declarations" was used in [Goguen 78] for the unconditional case) and have the general form

```
    as <Sort> : f(X1,...,Xn) if <Condition> .
```

An initial algebra semantics for sort constraints is given in [Goguen & Meseguer 84a] and an operational semantics is given in [Goguen, Jouannaud & Meseguer 85]. BOUNDED-STACK uses the following requirement theory

```
th NAT* is
  protecting NAT .
  op bound : -> Nat .
endth
```

NAT* is a somewhat subtle theory especially constructed so that giving an interpretation of it corresponds exactly to giving a natural number as interpretation of the constant **bound**. Now the example:

```
obj BOUNDED-STACK[X :: TRIV, Y :: NAT*] is
  sorts NeStack Stack ErrStack .
  subsorts NeStack < Stack < ErrStack .
  op empty : -> Stack .
  op push : Elt ErrStack -> ErrStack .
  op top_ : NeStack -> Elt .
  op pop_ : NeStack -> Stack .
  op length : Stack -> Nat .
  var E : Elt .
  var S : Stack .
  as NeStack : push(E,S) if length(S) < bound .
  eq : length(empty) = 0 .
  eq : length(push(E,S)) = inc length(S) .
  eq : top push(E,S) = E .
  eq : pop push(E,S) = S .
jbo
```

(This example has not been run since sort constraints require subtleties in the parser that are not yet implemented.) Overflow of such a stack during computation produces a runtime parse error (see Section 3.3).

3.2 Logic of Subsorts

Although subsorts are very expressive, their logic is very simple, and indeed can be reduced to standard equational logic [Goguen & Meseguer 84a, Goguen, Jouannaud & Meseguer 85]. This is both theoretically and practically important, since both standard algorithms for term rewriting and theorem-proving (e.g., the Knuth-Bendix algorithm) and the large literature on algebraic data types can be applied without modification. In the OBJ2 system, subsort notation is available at the user level, but is translated into (lengthier) internal standard equational forms. The key idea is to view a subsort pair $s<s'$ as a unary operation $c_{s,s'} : s \to s'$ called a **coercion** from s to s'. Appropriate equations ensure that $c_{s,s'}$ is injective and that the subsort relation is transitive. Also, whenever an operation symbol (such as "+") is defined for both s and s' with $s<s'$, the following "morphism equation" expresses that the results must be the same at both levels,

$c_{s,s'}(x) +s' c_{s,s'}(y) = c_{s,s'}(x +s\ y)$.

("+s" and "+s'" are new operation symbols for the two levels.)

This defines a translation from OBJ2's "order-sorted algebra" notation [Goguen 78] to standard equational notation. In particular, if E is a set of equations defining an OBJ2 object, we obtain a translation E^* to which the above equations have been added. There is an equivalence between the class of order-sorted algebras satisfying E and the class of standard algebras satisfying E^* under which the initial algebras of each class correspond [Goguen & Meseguer 84a]. Thus we can translate back and forth without losing information. This translation involves mapping an overloaded operation in the order-sorted algebra (such as + above) to one of its forms in the standard algebra (e.g., +s' or +s). This translation is done by choosing the operation with the smallest coarity such that the resulting expression, called the **lowest parse**, is well-formed. Simple syntactic conditions on signatures ensure uniqueness of lowest parse [Goguen, Jouannaud & Meseguer 85].

Assuming that the order-sorted equations in E are Church-Rosser, it turns out that new equations may have to be added to E^* in order to get an equivalent set of Church-Rosser standard equations [Goguen, Jouannaud & Meseguer 85]. Thus OBJ2 performs a specialized Knuth-Bendix completion[10] on the equations E^*. Even when an operation has been declared associative (or associative-commutative) for both a subsort s and a supersort s', we can still do rewriting modulo associativity (or associativity-commutativity) with the completed E^* using a standard associative (or associative-commutative) matching algorithm [Goguen, Jouannaud &

[10] The usual problems of orienting rules and of non-termination do not arise here.

Meseguer 85]. However, the correctness of this last reduction is rather subtle, since the associativity axioms at the two levels interact with morphism equations to generate additional equations having an associative flavour.

3.3 Error Handling and Recovery

It is widely admitted that handling erroneous and meaningless expressions in an algebraic setting reduces to handling subsorts and partial operations. However, quite a number of different approaches have been taken, including the following:

1. Error Sorts -- providing special sorts in which to put special error values:
   a. Error Algebras -- our original approach [Goguen 77] involved disjoint "ok" and "error" sorts for each standard sort; we have abandoned this approach and do not discuss it further here, except to note that it was implemented in OBJT and OBJ1.
   b. Explicit Error Supersorts -- this is one of several approaches supported by OBJ2; it involves a new supersort of each standard sort; for example, **List** $<$ **ErrList** with **tail : List -> ErrList** and **tail(nil)** $=$ **tailess** where **tailess** is an error message of sort **ErrList** not of sort **List**.
2. Partial Operations -- here operations are defined on only part of what is normally considered their domain; for example, **tail** is only defined for non-empty lists:
   a. Partial Algebras -- here one attempts to generalize the theory of abstract data types to algebras involving partially defined operations; it has been found that the mathematical theory is far more complex than one would like, and we will not discuss this approach further here.
   b. Domain Subsorts -- here one restricts operations to the domain on which they are meaningful by defining a subsort for this domain; for example, in OBJ2 we could define **NeList** $<$ **List** to be the sort of non-empty lists and then have **tail : NeList -> List**; OBJ2 also has sort constraints.
   c. Sort Constraints -- these give the effect of partial operations with domain defined by a condition, as in the **BOUNDED-STACK** example.
3. Recovery Operations -- these are operations from a sort that may contain errors to one that doesn't:
   a. Retracts -- these are left inverses to coercions, permitting contingent parsing of expressions which would otherwise be ill-formed, as described below.
   b. Error Handlers -- more general recovery operations than retracts are possible in OBJ2, but are not discussed here.

The choice among the options supported by OBJ2 is largely a matter of style and taste; moreover, certain of these choices are very closely related to one another. For example, OBJ2 automatically provides both a coercion and a retract for every subsort pair; thus, both error supersorts and domain subsorts have associated retracts. One consideration in making a choice is what kind of explicit error messages are desired.

Let us now discuss retracts in more detail. Ill-formed terms of the order-sorted algebra are translated to well-formed terms of an extended standard algebra having new operation symbols $r_{s',s}$ called **retracts** that are associated with each subsort pair $s<s'$. The semantics of retracts is given by new equations in E^* expressing that coercions are right inverses of retracts. Ill-formed order-sorted terms that "might" become well-formed after reduction get the benefit of the doubt at parse time by using retracts to fill gaps between actual sorts and required sorts. For example, in the object `TABLE[INT,BOOL]`, the expression

`put(put(true,1,empty)[ 1 ],5, put(false,2,empty))`

is temporarily accepted by the parser as the term

`put(`$r_{\mathtt{Errval,Elt.Val}}$`(put(true,1,empty)[ 1 ]), 5,put(false,2,empty))` .

Doubt arises since `put(true,1,empty)[ 1 ]` can be an **ErrVal** value that is not an **Elt.Val**, but the expression is vindicated after reduction. On the other hand, the expression

`put(put(true,1,empty)[ 2 ],5, put(false,2,empty))`

is also temporarily accepted by the parser as the term

`put(`$r_{\mathtt{Errval,Elt.Val}}$`undef(2),5,put(false,2,empty))`

but since this is already reduced, it is returned as is, with the retract operation serving as a very informative error message. This kind of runtime typechecking, together with the polymorphism provided by subsorts and parameterization, gives much of the syntactic flexibility of untyped languages with all the advantages of strong typing.

4 Implementation Principles

In trying to make OBJ2 convenient to use, easy to implement, and fairly efficient, we have developed several new implementation techniques.

4.1 Command Interpreter Generator

Given two files, one specifying command syntax and the other containing basic system functions, the Command Interpreter Generator (CIG) compiles a system interpreter. The command specification is in effect a graph of menu choices, given as S-expressions that describe the command hierarchy with associated function calls. Each command has four parts: open key, body, subcommand part, and close key. For the **POSET** theory in Section 2.3, the open key "*th*" starts the theory creation command, and "*endth*" closes it; its body is "**POSET** *is*"; and its subcommand part is a sequence of commands one level lower, given by the six lines starting with "*protecting* BOOL ." For example, the fourth of these lines defines a new operation, with "*op*" as its open key and "_<_ : Elt Elt -> Bool ." as its body; both its subcommand part and its close key are empty. The top level of OBJ2 can be seen as a single command whose subcommand set contains object, theory, and view commands[11]; without using italics for keywords, this might look as follows:

```
;;;OBJ2 top level command specification
(()                     ;open keys
 ()                     ;close keys
 ()                     ;body part
 (th-obj-view%c)        ;subcommand processing function
 ())                    ;exit function

;;;subcommand specification for top level
(th-obj-view%c          ;function name for subcommand
 ()                     ;init function for subcommand
 ("prompt")             ;prompt spec
 (* ((th theory)        ;open keys for th command
     (endt endth ht)    ;close keys for th command
     (th-name%c)        ;theory name processing function
     (th-parts%c *)     ;subcommand processing function for theory command
     (theory%ex))       ;exit function for th command
    ((obj ob object)    ;open keys for obj command
     (endo jbo bo endobj) ;close keys for obj command
     (obj-name-pm%c)    ;obj name/parameter processing function
     (obj-parts%c *)    ;subcommand processing function for obj command
     (object%ex))       ;exit function for obj command
    ((view vw)          ;open keys for view command
     (endv endview wv endwv) ;close keys for view command
     (view-name-part%c) ;view name part processing function
     (view-parts%c *)   ;subcommand processing function for view command
     (view%ex)))        ;exit function for view command
 ())                    ;final function for subcommand

;;;subcommand spec for obj commands
(obj-parts%c            ;function name for this subcommand
 ()
 ("prompt")             ;prompt spec
 (* ...
    ((sort sorts so)    ;open key for sort command
```

[11] The real OB2 syntax is more complex, since communication with the reduction engine, file system, editor (Emacs), and Maclisp evaluator also occur at the top level.

```
   ()
   ...)
   ...
  ((op operator operation) ;open keys for op command
   ()
   ...)
  ((var vars)            ;open keys for var command
   ()
   ...)
  ((eq equation)         ;open keys for eq command
   ()
   ...)
  ...)
())
...
```

The first S-expression specifies the top level command, while subsequent S-expressions specify its subcommand parts. The CIG generates one Lisp function for each S-expression in this sequence. The generated functions combine with the basic functions that process body parts to give a highly interactive interpreter. The system generates an informative prompt[12] and the user should then supply a body or keyword. At each point in the interaction, the menu of possible commands is automatically available. The CIG also supports inputing bodies through "prompt/reply" interaction. (Unfortunately, there is not room here for a detailed explanation of CIG syntax.)

The present OBJ2 command structure only requires choosing an arbitrary sequence of subcommands from the fixed subcommand set of each command. However, a CIG can in principle handle many other kinds of command-subcommand control; in particular, the present CIG also supports choosing just one subcommand from the subcommand set of a command. Note that there are natural translations between the CIG's syntactic formalism and more familiar BNF. The Appendix gives BNF syntax for OBJ2. Our CIG runs in Maclisp, but Thierry Billoir has also implemented it on the Symbolics 3600, providing mouse-sensitive pop-up menus for command choice.

[12] OBJ2 prompts with the sequence of previous open keys.

4.2 Parser

The OBJ2 parser translates order-sorted expressions (the left- and right-hand sides and conditions of equations, sort expressions in sort constraints, intermediate terms from the reduction engine, and user queries) into sorted expressions in standard tree form, more specifically into S-expressions of internal operation symbols. For example, consider the following

```
sorts Int Rat .
subsorts Int < Rat .
op _+_ : Int Int -> Int [assoc comm]
op _*_ : Int Int -> Int [assoc comm 1]
op _-_ : Int Int -> Int .
op gcdOf_and_ : Int Int -> Int .
op _+_ : Rat Rat -> Rat [assoc comm]
op _*_ : Rat Rat -> Rat [assoc comm 1]
op _/_ : Int Int -> Rat .
vars A B C D : Int .
vars X Y Z : Rat .
```

where *comm* in italic brackets indicates that the operation is commutative, and a number in italic brackets gives the parsing precedence (operations without numbers have precedence 0). The parser distinguishes + on Int from + on Rat by decorating the operation with the sort, and we will write +i and +r respectively (internal names for operations are a bit more complex in the real implementation). The coercion from Int to Rat and the retract from Rat to Int are represented by c\i\r and r\r\i respectively; this retract permits parsing a term with an Int operation applied to a Rat expression. Then we have the following sample lowest parses:

1. A + B + C => (+i A (+i B C));
2. X + Y + Z => (+r X (+r Y Z));
3. A - B - C => (- A (- B C)) or (- (- A B) C),
   a parse error since it is ambiguous;
4. A + B * C => (+i A (*i B C));
5. A + B + X => (+r (c\i\r (+i A B)) X); and
6. gcdOf A + B and C / D => (gcdOf (+i A B) (r\r\i (/ C D))).

A binary associative operation is parsed as right associative, whereas a similar nonassociative operation would have an ambiguous parse, e.g., (1), (2), (3); commutative declarations have no effect on parsing; see [Goguen, Jouannaud & Meseguer 85] for more detail. Operations of lowest precedence are tried first at the top level of an expression (see (4)), and the parser gives expressions the lowest possible sort; thus, A + B is (+i A B) rather than (+r (c\i\r A) (c\i\r B)). Possible retracts must be considered in parsing expressions to be reduced, e.g. (6) (see Section 3.2).

The parser uses recursive descent backtracking, since it may be necessary to try all possibilities in checking that there is exactly one meaningful lowest parse; thus, parsing can be expensive for complex expressions. The parser stores partial results in tables to avoid redoing work. For example, in parsing A + B * A + B * A + B, the parse of B * A is stored and later looked up. This has a large effect for expressions like A - A - A - A. The parser also stores ambiguous expressions (those with several parses) and all their parses in a table; it can then return a single result indicating ambiguity, such as (*error* A - A - A - A); thus, we get

```
(1)  (*error* A - A - A - A)   with parses
          (- A (*error* A - A - A))
          (- (- A A) (- A A))
          (- (*error* A - A - A) A)
(2)  (*error* A - A - A)   with parses
          (- A (- A A))
          (- (- A A) A)
```

4.3 Database and Module Expression Evaluator

The OBJ2 database has four kinds of identifiers: module (theory or object) and view identifiers, sort identifiers, operation identifiers, and equation identifiers. Modules and views are basic units manipulated by the OBJ2 language; but sorts, operations and equations cannot be manipulated without affecting modules since every sort, operation and equation belongs to some module. Since module and view names are unique (Section 2.1), they can be used as identifiers in the database. A module's parameter names are local, and are identified by internally created global names. Since the sort names given by users are also local, database sort identifiers annotate the user sort name with the module name. An operation identifier is annotated by its form, its **rank** (the list of arity and coarity sorts), and its module. Equation identifiers are created from the top operation of the lefthand side and a key to distinguish equations having the same such operation. All information that may be needed later is retained in properties of these identifiers. For example, all operations belonging to a module are retained as a property of the module identifier under "m!ops", and the extend-hierarchy structure of modules is retained in a list of module identifiers on "m!extends" and "m!extended" properties.

The OBJ2 database supports reusability of modules. Thus, a parameterized module is reused by instantiating its parameter theories with appropriate views, without affecting the original module. A new module can also be created by renaming, thus giving different names to sorts and operations, and reusing most other parts of the old module. Evaluating a module expression often requires creating new modules with different sort names, operation forms or parameter values. This is implemented by representing modules as lambda expressions with their sort names, operation forms, and parameter modules as arguments. By applying such lambda expressions, new modules are efficiently created in the database. This method is also used to store already constructed modules into files so that whole systems can later be reused.

4.4 Rewrite Rule Engine

OBJ2 uses equations as rewrite rules, assuming that they terminate and are Church-Rosser. Since rewriting can use any combination of associative, commutative, identity and idempotent pattern matching, the rewrite engine is parameterized by the kind of rewriting used, by attaching properties to both sides of each equation; then (for example) commutative matching is used when a lefthand side has a commutative operation. When a righthand side is (for example) associative, its instances are put in normal form, i.e., flattened. Each property has such a "normalization," systematic use of which greatly speeds up the OBJ1 matching process implemented by Plaisted. OBJ2 rewriting is also parameterized by the **E-strategy** of each operation, a list of natural numbers telling the order to try reductions. For example, **`if_then_else_fi`** has strategy `(1 0)`; the initial `1` means first reduce the first subterm; the following `0` means reduce at the top as long as possible after that.

The rewrite engine's top level function, Objval, determines if a given term is already reduced by checking its representation. If it is not, Objval checks if another occurrence of the same term has already been computed and stored in a hash table (used for results of computations with top symbol having the "*saveruns*" attribute, which can be set by users). Of course, there may be collisions in the hash table, in which case only the most recent computation is retained. Finally, if the term must be computed from scratch, Reduce is called by Objval with the E-strategy of the top operation, and the starting term is overwritten by the reduced one, to avoid recomputing shared subterms. Depending on the E-strategy, Reduce splits into the following cases:

1. If the strategy is empty, then Reduce is called again with the remaining occurrences where reductions must be performed, as indicated in 3.d. If there are no such occurrences, then Reduce returns that result.
2. For non-empty E-strategies, the first ocurrence of the strategy is processed and if it is not 0, then Objval is called with the corresponding subterm as argument before reducing the whole term with the remaining part of the strategy.

3. Finally, if the first occurrence is 0, then reduction starts at the top, and the following are tried successively:
   a. Coercion and retract-coercion rules, to keep terms in normal form; e.g., terms to be reduced must be lowest parses of some OBJ2 expression for the Church-Rosser property to hold. These rules are not in the database, but the rewrite engine knows their form and uses them. The rewritten term will be in normal form because coercions and retracts first reduce their arguments; hence, no recursive call is needed here.
   b. Morphism rules, as discussed in Section 3.2, are also used to keep terms in normal form, and are built into the rewrite engine. If one applies, then Objval must be called recursively, since a new operation is on top.
   c. Built-in operations, defined by equations whose righthand side is Maclisp code; see Section 4.5. The result of a built-in operation is either a built-in constant or else an OBJ2 expression that must be parsed and then evaluated recursively by Objval.
   d. Finally, standard rules are attempted according to their topmost lefthand side operation. Actually, these rules are stored into two different data structures: those that don't change the top operation are tried first until none can be applied, using a ring that is searched for new rules to apply as soon as the current rule fails. This efficiently implements "tail recursive calls"; for example, the second rule of

      eq : `X + 0 = X .`
      eq : `X + s Y = (s X) + Y .`

      has + on top of both sides; so an expression with + on top is repeatedly matched with the second rule until it fails. When the ring has been searched without finding a rule that applies, the remaining rules are applied until one is found (or fails). If there is a match, then a new operation is on top, and Objval is called again; otherwise, the term cannot be reduced any more on top. However, reduction may now be possible on some immediate subterms because of reductions by rules in the ring: these are the new reductions that must be tried once the current strategy has been exhausted.

Let us emphasize some points:
1. E-strategies may lead to a complex search of the tree, very much like what happens with evaluation of parse trees by attribute grammars.
2. E-strategies need not be given by the user, but instead can be automatically generated at parse time using simple heuristics that try to avoid useless computations: for example, + in the above example has strategy `(2 0 1)`, using the heuristic that a binary operation with

no rules that look inside its first argument should begin by evaluating its second argument, and then reduce at the top; the final `1` causes complete reduction of non-ground terms with `+` remaining on top; note that left reducing a ground term with `+` on top is useless, since `+` is completely defined with respect to a set of constructors. On the other hand, retracts, coercions and most built-in operations (the exception being control structure built-ins like **`if_then_else_fi`**) have a call-by-value strategy that reduces their subterms before trying to reduce at the top.

3. E-strategies do not implement an optimal strategy, if one exists, but they are very efficient in practice: the ratio between attempted matches and sucessful matches is usually around 2/3, which is really impressive. In particular, use of E-strategies and the tail recursion ring makes a much larger difference in the efficiency of term rewriting than does the choice among data structures as discussed in [Hoffman & O'Donnell 84].
4. As an application of theory in [Goguen, Jouannaud & Meseguer 85], we can reduce expressions that are not well-formed at parse time but may have a well-formed normal form. This occurs, for example, if we try to use `s s s s 0 / s s 0` as an integer (`0` is zero and `s` is the successor function) somewhere in a term. It is apparently a rational, but since integers are a subsort of rationals, it may really be an integer after reduction. This problem is handled by the retract operations mentioned in Section 3.3 as follows: a retract from rationals to integers is put on top of the above subterm to indicate the hoped-for sort. Then, at reduction time the subterm reduces to `s s 0` with a coercion to the rationals since the result is supposed to be rational. Then a retract-coercion rule is applied to delete these two extra operations, permitting the rest of the computation to proceed. [Goguen, Jouannaud & Meseguer 85] proves soundness and completeness of this technique: if the starting term cannot be parsed correctly but its equivalence class contains a correct term, then the normal form will be computed, assuming that the initial rules are Church-Rosser.
5. Last, but not least, all this was easy to implement and provides well-structured code, since choice of the next subterm to reduce is not part of the rewrite engine itself.

4.5 User Defined Built-Ins

A difficult problem for logic programming is to maintain purity and yet provide efficiency and input/output; most languages simply compromise purity. OBJ2 adopts an approach in which logical purity can be maintained (at some cost in specification and theorem proving) or can be compromised (if needed to satisfy the practical requirements of a programming project). This approach permits users to create new "built-in" functions or objects, by implementing them in the underlying Lisp; they are also documented with OBJ2 syntax and (optionally, but encouraged) with equations. This approach provides OBJ2's built-in objects for Booleans, integers, natural numbers, identifiers, lists, arrays, etc. Thus, we get the efficiency of compiled Lisp along with a precise mathematical description of what these data types do. Although the algebraic specification is not executed, it can be useful for documentation and theorem proving. Built-ins are also useful for implementing I/O, including graphics. Built-ins will later permit "compiling" operations defined by equations into Lisp functions (as in Hope and ML) without changing the rewrite engine, since these operations can be seen as "built-in" once their Lisp code is generated. Note that a built-in operation may produce a built-in constant or an OBJ2 expression. Thus, built-in definitions must combine OBJ2 and Lisp syntax.

In order to verify that a given Lisp implementation of a built-in does in fact satisfy its specification (a standard OBJ2 object), we need a precise notion of "implementation". The literature includes a number of these [Goguen, Thatcher & Wagner 78, Ehrich 82], and the following seems adequate for the present purpose (it does not take account of states, for which see [Meseguer & Goguen 84, Goguen & Meseguer 82a]; however, the current OBJ2 does not have objects with states). Let A and B be objects with signatures Σ and Σ', reachable over their respective signatures. Then an **implementation** of A by B is a view v from the theory with signature Σ and no equations to the theory with signature Σ' and no equations (in the general sense mentioned previously, that maps Σ-operations to Σ'-expressions) such that there is a (necessarily unique and surjective) Σ-homomorphism to A from the Σ-reachable part of B viewed by v as a Σ-algebra; in symbols, $\mathrm{reach}_{\Sigma}(B^{v}) \rightarrow A$. It can be proved that implementations in this sense compose.

5 Past, Present and Future

OBJ2 is a fruition of fifteen years research in algebraic semantics[13]. In 1972, the basic principle of "initial algebra semantics" (IAS) was developed, and in 1973 applied to the denotational (or "attribute") semantics of context-free languages [Goguen 74]; this was later [Goguen, Thatcher, Wagner & Wright 77] related to abstract syntax, the Scott-Strachey approach, and structural induction. By 1973 we realized that concrete data types could usefully be viewed as many-sorted algebras [Goguen 73]. In 1974, IAS was applied to ADTs by

[13]This paragraph is not intended to survey all work related to OBJ or even all work contributing directly to OBJ; instead, it only sketches the direct line of development. Any survey of closely related work would have to include the important work of Zilles [Zilles 74] and Guttag [Guttag 75] on ADTs, the work on Hope by Burstall, MacQueen, Sanella and others [Burstall, MacQueen & Sanella 80], the work of Hoffman and O'Donnell [Hoffman & O'Donnell 82] on rewrite rule languages, and the survey of Huet and Oppen [Huet & Oppen 80] on rewrite rule theory. [Meseguer & Goguen 84] surveys a good deal of relevant work in algebraic semantics.

ADJ [Goguen, Thatcher & Wagner 78]. Two awkward limitations of this first approach were addresed by subsequent work: parameterization and errors (including partial operations). In 1977, the Clear specification language [Burstall & Goguen 77] was introduced as a way of structuring complex specifications into simpler parts; a major principle here is the use of so-called "colimits" of specifications, following earlier work on structuring general systems [Goguen 71, Goguen & Ginali 78]. Clear's powerful and precise mechanism for parameterized specifications inspired OBJ2's generic module mechanism. 1977 also saw the first draft of OBJ, in connection with a proposed algebraic semantics for errors [Goguen 77]. This language was subsequently implemented by Joseph Tardo, first as OBJ0 and later as OBJT [Goguen & Tardo 79]. The original motivation for this work was the observation that algebraic specifications published in the literature were very often wrong, so that some way of testing them was needed. The theorem that links rewrite rules with equational ADT specifications using IAS is that the normal forms of a set of terminating Church-Rosser rewrite rules give the initial algebra of the rules viewed as a set of equations; this result is the theoretical basis for OBJ and appears for example in [Goguen 80]. David Plaisted's OBJ1 [Goguen, Meseguer & Plaisted 83] is a significant improvement of OBJT, including his clever efficient implementation of associative-commutative rewriting, and many powerful and convenient interactive features, such as a spelling checker. The current OBJ2 implementation has profited from all of this.

Two improvements of OBJ2 to be implemented soon are: a "universal" sort U -- this will give a tremendous flexibility in programming style, since it supports arbitrary mixtures of typed and untyped code; and a new value * for E-strategies, indicating lazy evaluation -- this will support infinite data structures and processes. A more distant improvement will permit states in objects, following the theory in [Goguen & Meseguer 82a].

Although we originally thought of OBJ as a vehicle for testing algebraic ADT specifications, we soon came to think of it as a general purpose executable specification language, suitable for rapid prototyping [Goguen & Meseguer 82b] and for programming language semantics [Goguen & Parsaye-Ghomi 81]. However, our recent more efficient implementations and the expected arrival of parallel machines lead us to regard OBJ as an ultra high level programming language that can be executed extremely rapidly on suitable architectures, and that is especially suitable for "fifth generation" applications (such as expert systems). OBJ is a "logic programming" language in the sense that its basic statements are equations, and the interpretation of those equations (in algebras representing the underlying data types) agrees with the logic of equations plus the initiality principle. Moreover, a basic assumption of the rewrite rule implementation (namely, the Church-Rosser property) implies that multiple processors can be set to work concurrently, without the user having to explicitly schedule them or their interactions, with the final result guaranteed independent of the order of execution; also any available nonoverlapping rewrites can be carried out simultaneously. Recent work [Goguen & Meseguer 84b] on Eqlog explores combining the equational logic (and other powerful features) of OBJ with the Horn clause logic (and other powerful features) of Prolog. Two other research directions that we hope to pursue for rewrite rule languages are graphical programming methods and parallel architectures.

Future work will also explore integration with the theorem proving capabilities of the REVE system [Lescanne 83], and compilation techniques for rewrite rule based languages. Integration with REVE will permit OBJ2 to perform many validation checks on objects, including completeness checks using Thiel's algorithm [Thiel 84] and consistency checks using the so-called inductive completion algorithm [Huet & Hullot 82]. On the other hand, OBJ2 will provide REVE with a powerful language for specifying complex hierarchical theories, supporting the "hierarchical inductive completion algorithm" of [Kirchner H. 84]. In turn, this algorithm will permit checking OBJ2's *protecting* property. These techniques can be used to prove that actuals satisfy the requirement theories of parameterized modules. Thus, OBJ and REVE together constitute a very powerful environment that integrates programming, specification and verification[14]. There is little doubt that this will raise interesting new issues in each of these fields.

6 References

[Bergstra & Tucker 80a]
Bergstra, J. A., and Tucker, J. V.
A Characterization of Computable Data Types by Means of a finite equational specification method.
In J. W. de Bakker and J. van Leeuwen (editors), *Lecture Notes in Computer Science, Volume 81: Automata, Languages and Programming, Seventh Colloquium*, . pages 76-90. Springer-Verlag, 1980.

[Bergstra & Tucker 80b]
Bergstra, J. A., and Tucker, J. V.
Algebraic Specifications of Computable and Semicomputable Data Structures.
1980.
To appear in *Theoretical Computer Science*; 24pp.

[14] See also the Iota System [Nakajima & Yuasa 83].

[Burstall & Goguen 77]
Burstall, R. M. and Goguen, J. A.
Putting Theories together to Make Specifications.
Proceedings, Fifth International Joint Conference on Artificial Intelligence 5:1045-1058, 1977.

[Burstall & Goguen 80]
Burstall, R. M., and Goguen, J. A.
The Semantics of Clear, a Specification Language.
In *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, , pages 292-332. Springer-Verlag, 1980.
Lecture Notes in Computer Science, Volume 86.

[Burstall & Goguen 82]
Burstall, R. M. and Goguen, J. A.
Algebras, Theories and Freeness: An Introduction for Computer Scientists.
In *Proceedings, 1981 Marktoberdorf NATO Summer School*, . Reidel, 1982.

[Burstall, MacQueen & Sanella 80]
Burstall, R. M., MacQueen, D. and Sanella, D.
HOPE: an Experimental Applicative Language.
In *Conference Record of the 1980 LISP Conference*, , pages 136-143. Stanford University, 1980.

[DoD 83]
United States Department of Defense.
Reference Manual for the Ada Programming Language.
ANSI/MIL-STD-1815 A.
1983

[Ehrich 82]
Ehrich, H.-D.
On the Theory of Specification, Implementation and Parameterization of Abstract Data Types.
Journal of the Association for Computing Machinery 29:206-227, 1982.

[Futatsugi & Okada 80]
Futatsugi, K. and Okada, K.
Specification Writing as Construction of Hierarchically Structured Clusters of Operators.
In *Proceedings, IFIP Congress 80*, , pages 287-292. IFIP Press, 1980.

[Futatsugi & Okada 82]
Futatsugi, K. and Okada, K.
A Hierarchical Structuring Method for Functional Software Systems.
In *Proceedings, 6th International Conference on Software Engineering*, , pages 393-402. IEEE Press, 1982.

[Goguen 71] Goguen, J.
Mathematical Foundations of Hierarchically Organized Systems.
In E. Attinger (editor), *Global Systems Dynamics*, , pages 112-128. S. Karger, 1971.

[Goguen 73] Goguen, J. A.
Some Remarks on Data Structures.
1973.
Abstract of 1973 Lectures at Eidgenoschiche Technische Hochschule, Zurich.

[Goguen 74] Goguen, J. A.
Semantics of Computation.
In *Proceedings, First International Symposium on Category Theory Applied to Computation and Control*, , pages 234-249. University of Massachusetts at Amherst, 1974.
Also published in Lecture Notes in Computer Science, Vol. 25., Springer-Verlag, 1975, pp. 151-163.

[Goguen 77] Goguen, J. A.
Abstract Errors for Abstract Data Types.
In *IFIP Working Conference on Formal Description of Programming Concepts*, . MIT, 1977.
Also published by North-Holland, 1979, edited by P. Neuhold.

[Goguen 78] Goguen, J. A.
Order Sorted Algebra.
Technical Report, UCLA Computer Science Department, 1978.
Semantics and Theory of Computation Report No. 14.

[Goguen 80] Goguen, J. A.
How to Prove Algebraic Inductive Hypotheses without Induction: with applications to the correctness of data type representations.
In W. Bibel and R. Kowalski (editors), *Proceedings, 5th Conference on Automated Deduction*, , pages 356-373. Springer-Verlag, Lecture Notes in Computer Science, Volume 87, 1980.

[Goguen 84] Goguen, J. A.
Parameterized Programming.
Transactions on Software Engineering SE-10(5):528-543, September, 1984.
Originally appeared, *Proceedings, Workshop on Reusability in Programming*, edited by Biggerstaff, T. and Cheatham, T., ITT, pages 138-150, 1983; also, revised version as Technical Report CSLI-84-9, Center for the Study of Language and Information, Stanford University, September 1984.

[Goguen & Burstall 84]
Goguen, J. A. and Burstall, R. M.
Introducing Institutions.
In E. Clarke and D. Kozen (editor), *Proceedings, Logics of Programming Workshop*, , pages 221-256. Springer-Verlag, 1984.
Lecture Notes in Computer Science, volume 164.

[Goguen & Ginali 78]
Goguen, J. A. and Ginali, S.
A Categorical Approach to General Systems Theory.
In G. Klir (editor), *Applied General Systems Research*, , pages 257-270. Plenum, 1978.

[Goguen & Meseguer 82a]
Goguen, J. A. and Meseguer, J.
Universal Realization, Persistent Interconnection and Implementation of Abstract Modules.
In *Proceedings, 9th International Colloquium on Automata, Languages and Programming*, . Springer-Verlag, 1982.
Lecture Notes in Computer Science.

[Goguen & Meseguer 82b]
Goguen, J. and Meseguer, J.
Rapid Prototyping in the OBJ Executable Specification Language.
Software Engineering Notes 7(5):75-84, 1982.
Proceedings of Rapid Prototyping Workshop.

[Goguen & Meseguer 84a]
Goguen, J. and Meseguer, J.
Order-Sorted Algebra: Partial and Overloaded Operations, Errors and Inheritance.
Technical Report, SRI International, Computer Science Lab, 1984.
Given as lecture 'Logic of Subsorts and Polymorphism' at Seminar on Types, Carnegie-Mellon University, June 1983.

[Goguen & Meseguer 84b]
Goguen, J. and Meseguer, J.
Equality, Types, Modules and (Why Not?) Generics for Logic Programming.
The Journal of Logic Programming 1(2):179-210, 1984.
Also appears in *Proceedings, 1984 Logic Programming Symposium*, Upsala, Sweden, pp. 115-125; and Report CSLI-84-5, Center for the Study of Language and Information, Stanford University, March 1984.

[Goguen & Parsaye-Ghomi 81]
Goguen, J. A. and Parsaye-Ghomi, K.
Algebraic Denotational Semantics using Parameterized Abstract Modules.
In J. Diaz and I. Ramos (editors), *Formalizing Programming Concepts*, , pages 292-309. Springer-Verlag, Peniscola, Spain, 1981.
Lecture Notes in Computer Science, volume 107.

[Goguen & Tardo 79]
Goguen, J. A. and Tardo, J.
An Introduction to OBJ: A Language for Writing and Testing Software Specifications.
In *Specification of Reliable Software*, , pages 170-189. IEEE Press, 1979.

[Goguen, Jouannaud & Meseguer 85]
Goguen, J., Jouannaud, J.-P. and Meseguer, J.
Operational Semantics of Order-Sorted Algebra.
Summary presented at IFIP WG2.2, Boston, June 1984.
1985
Submitted for publication.

[Goguen, Meseguer & Plaisted 83]
Goguen, J. A., Meseguer, J., and Plaisted, D.
Programming with Parameterized Abtract Objects in OBJ.
In D. Ferrari, M. Bolognani and J. Goguen (editors), *Theory and Practice of Software Technology*, , pages 163-193. North-Holland, 1983.

[Goguen, Thatcher & Wagner 78]
Goguen, J. A., Thatcher, J. W. and Wagner, E.
An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types.
Current Trends in Programming Methodology IV:80-149. 1978.
Original version, IBM T. J. Watson Research Center Technical Report RC 6487, October 1976.

[Goguen, Thatcher, Wagner & Wright 77]
Goguen, J. A. , Thatcher, J. W., Wagner, E. and Wright, J. B.
Initial Algebra Semantics and Continuous Algebras.
Journal of the Association for Computing Machinery 24(1), January, 1977.

[Guttag 75]
Guttag, J. V.
The Specification and Application to Programming of Abstract Data Types.
PhD thesis, University of Toronto, 1975.
Computer Science Department, Report CSRG-59.

[Hoffman & O'Donnell 82]
Hoffman, C. M. and O'Donnell, M. J.
Programming with Equations.
ACM Transactions on Programming Languages and Systems 1(4):83-112, 1982.

[Hoffman & O'Donnell 84]
Hoffman, C. and O'Donnell, M.
Pattern Matching in Trees.
Journal of the Association for Computing Machinery 29(1):68-95, 1984.

[Huet & Hullot 82]
Huet, G., Hullot, J.M.
Proofs by Induction in Equational Theories with Constructors.
Journal of the Association for Computing Machinery 25(2):239-266, 1982.

[Huet & Oppen 80]
Huet, G. and Oppen, D.
Equations and Rewrite Rules: A Survey.
In R. Book. (editor), *Formal Language Theory: Perspectives and Open Problems*, . Academic Press, 1980.

[Jouannaud & Kirchner 84]
Jouannaud, J.P. and Kirchner, H.
Completion of a Set of Rules Modulo a Set of Equations.
In *Proceedings, 11th Symposium on Principles of Programming Languages*, , pages 83-92. ACM, 1984.
to appear, *SIAM Journal of Computing*.

[Kirchner H. 84] Kirchner, H.
A General Inductive Completion Algorithm and Application to Abstract Data Types.
In *Proceedings, 7th Conference on Automated Deduction*, , pages 282-302. Springer-Verlag, 1984.
Lecture Notes in Computer Science, Volume 170.

[Lescanne 83] Lescanne, P.
Computer Experiments with the REVE Term Rewriting Systems Generator.
In *Proceedings, Symposium on Principles of Programming Languages*, . ACM, 1983.

[Mac Lane 71] Mac Lane, S.
Categories for the Working Mathematician.
Springer-Verlag, 1971.

[Meseguer & Goguen 84]
Meseguer, J. and Goguen, J. A.
Initiality, Induction and Computability.
In M. Nivat and J. Reynolds (editors), *Algebraic Methods in Semantics*, . Cambridge University Press, 1984.
To appear; also available as SRI CSL Technical Report 140, December 1983.

[Nakajima & Yuasa 83]
Nakajima, R. and Yuasa, T.
The IOTA Programming System.
Springer-Verlag, 1983.
Lecture Notes in Computer Science, Volume 160.

[Thiel 84] Thiel, J.J.
Stop Losing Sleep over Incomplete Data Type Specification.
In *Proceedings, 11th Symposium on Principles of Programming Languages*, . ACM, 1984.

[Zilles 74] Zilles, S.
Abstract Specification of Data Types.
Technical Report Report 119, Computation Structures Group, MIT, 1974.

7 Appendix: OBJ2 BNF

This appendix does not use italics for keywords, nor does it give syntax for evaluation, file manipulation, etc.

Syntactic Conventions

```
<Name>          nonterminal symbols
|               alternative separator
exp...          one or more exp's
exp,...         one or more exp's separated by ","
{{exp}}         zero or one exp,
                  except that {{.}} indicates either period or <cr>
{e1,...,en}     choice of exactly one of e1,...,en
{_exp_}         exp in one line; i.e., no <cr>
(exp)           parentheses for syntactic grouping of expressions
"(" ")"         parentheses as terminal symbols
--- text <cr>   comment
```

Theory and Object Commands

```
<ThCmd> ::= {th,theory} <ThName> is <ThPart>... {endt,endth,ht}
<ThPart> ::= <ProtectingCmd> | <ExtendingCmd> | <UsingCmd> | <DefineCmd>
           | <SortsCmd> | <SubsortsCmd> | <VariablesCmd> | <SortConstrCmd>
```

```
             | <OpCmd> | <OpsCmd> | <EqCmd> | <CondEqCmd>

<ObjCmd> ::= {obj,ob,object} <ObjName> {{<ParameterDecl>}} is
             <ObjPart>... {endo,jbo,bo,endobj}
<ParameterDecl> ::= [ (<ParameterKey> :: <ThName>),... ]
<ObjPart> ::= <ThPart> | <BuiltinEqCmd>

<ProtectingCmd> ::= {protecting,pr} <ModExp> .
<ExtendingCmd> ::= {extending,ex} <ModExp> .
<UsingCmd> ::= {using,us} <ModExp> .

<DefineCmd> ::= {df,dfn,define} <SortKey> := <ModExp> .
--- the principal sort of the module <ModExp> is renamed to <SortKey>, i.e.
---   df SortKey := ModExp
--- is equivalent to
---   extending <ModExp> * "(" sort <PrincipalSort> to <SortKey> ")"

<SortsCmd> ::= {sort,sorts,so} {_<SortKey>..._} {{.}}
<SubsortsCmd> ::= {subsorts,ss} (<Sort>... <)... {_<Sort>..._} {{.}}
<Sort> ::= <SortKey> | <SortKey>.<ModName>
<ModName> ::= <ThName> | <ObjName>

<OpCmd> ::= {op,operator,operation} <OpForm> : {{<Arity>}} -> <sort> <OpClose>
<OpsCmd> ::= {ops,operators,operations} <OpForm>... : {{<Arity>}} ->
             <sort> <OpClose>
<Arity> ::= <Sort>...
<OpClose> ::= . | {[,at,attr,attribute} <Attribute>...
                       {],ta,enda,endat,rtta}
<Attribute> ::= assoc | associative | comm | commutative
             | identity: <OpForm> | id: <OpForm>
             | idmpt | idem | idempotent | <Strategy> | <Precedence>
             | saveruns | sr
<strategy> ::= "(" <NatNum>... ")"
<Precedence> ::= <IntNum>
<NatNum> ::= --- 0 1 2 3 ...
<IntNum> ::= --- 0 1 -1 2 -2 3 ...

<VariablesCmd> ::= {var,vars} <VarName>... : <Sort> {{.}}

<SortConstrCmd> ::= as <Sort> : <SortExp> if <Condition> .
<SortExp> ::= <SimpleTerm>
--- <SimpleTerm> is an expression composed only
--- of one operator symbol and variables (e.g.,
--- push Elm to Stack, add Elm to Set, etc.)
<Condition> ::= <Term>
```

```
<EqCmd> ::= {eq,equation} {{<Sort>}} : <Lhs> = <Rhs> .
<BuiltinEqCmd> ::= {beq,bq,builtineq} {{<Sort>}} : <Lhs> = <Sexp> .
<CondEqCmd> ::= {cq,ceq,condeq} {{<Sort>}} : <Lhs> = <Rhs> if <Condition> .
<Lhs> ::= <Term>
<Rhs> ::= <Term>

<MakeCmd> ::= {make,mk} <ObjName> is <ModExp> {endm,endmk,km,ekam}
--- make <ObjName> is <ModExp> endm
--- is equvalent to
--- obj <ObjName> is using <ModExp> . endo
```

View Commands

```
<ViewCmd> ::= {vw,view} <ViewName> of <ObjName> as <ThName> is <ViewPartCmd>...
                {endv,endview,endvw,wv}
<ViewPartCmd> ::= <SortPairCmd> | <VariablesCmd> | <OpExpCmd>
<SortPairCmd> ::= {so,sort} <Sort> to <Sort>
<OpExPairCmd> ::= {op,operator,operation} {{<Sort>}} : <OpExp> to
                    {{<Sort>}} : <Term>
<OpExp> ::= <SimpleTerm>
```

Module Expressions

```
<ModExp> ::= <ModName> | <ModName> [ <ModExp>,... ]
              | <ModExp> <Rename> | (<ModExp> +)... <ModExp>
<Rename> ::= * "(" (<RenamePartCmd>{{,}})... ")"
--- M1 + M2 * (op f to g)
--- should be parsed as
--- M1 + (M2 * (op f to g))

<RenamePartCmd> ::= <SortPairCmd> | <OpPairCmd>
<SortRenameCmd> ::= {so,sort} <Sort> to <SortKey>
<OpRenameCmd> ::= {op,operator,operation} <Operator> to "("<OpForm>")"
<Operator> ::= "("<OpForm>")" | "(""("<OpForm>")" : {{<Arity>}} -> <CoArity>")"

--- Note that the following nonterminals are not defined here: <ThName>
--- <ObjName> <ViewName> <ParameterKey> <SortKey> <OpForm> <Term> <SimpleTerm>
```