Parameterized Programming

JOSEPH A. GOGUEN

Abstract—Parameterized programming is a powerful technique for the reliable reuse of software. In this technique, modules are parameterized over very general interfaces that describe what properties of an environment are required for the module to work correctly. Reusability is enhanced by the flexibility of the parameterization mechanism proposed here. Reliability is further enhanced by permitting interface requirements to include more than purely syntactic information. This paper introduces three new ideas that seem especially useful in supporting parameterized programming: 1) *theories*, which declare global properties of program modules and interfaces; 2) *views*, which connect theories with program modules in an elegant way; and 3) *module expressions*, a kind of general structured program transformation which produces new modules by modifying and combining existing modules. Although these ideas are illustrated with some simple examples in the OBJ programming language, they should also be taken as proposals for an Ada[1] library system, for adding modules to Prolog, and as considerations for future language design efforts. OBJ is an ultra-high level programming language, based upon rewrite rules, that incorporates these ideas, and many others from modern programming methodology.

Index Terms—Adaptability, interfaces, logic programming, methodology, modularization, OBJ, parameterized programming, program library, programming, program transformation, reliability, reusability.

I. Introduction

BOTH the costs and the demands for software are enormous; moreover, they are rapidly escalating. One promising approach for dealing with this situation is to *reuse* software to the maximum possible extent. This paper presents a technique called parameterized programming that can greatly extend the opportunities for reusing software modules. The paper also describes some language features that help to support parameterized programming, and then illustrates the use of this technique with some simple examples in the OBJ language.

The main work of this paper occurs in Section V, which discusses our parameterization concepts. Although these ideas are illustrated with some simple examples in the OBJ programming language, they should also be taken as proposals for an Ada library system (as developed in [18]), for adding modules to Prolog (as developed in [22]), and as considerations for future language design efforts, especially OBJ2. Sections II, III, and IV provide the background in OBJ needed to understand the examples; in general, these follow [26]. Section VI presents some conclusions, and the appendixes present some details arising from Section V.

A. Parameterized Programming

It is easier to reuse program parts than to reinvent them, provided that the time needed for program understanding is less than the time needed for program writing, and provided that the access time for the needed program parts is sufficiently small; then total programming time and also debugging time are reduced. The basic idea of *parameterized programming* is to maximize program reuse by storing programs in as general a form as possible. One can then construct a new program module from an old one just by instantiating one or more parameters. For this to work, we need a suitable notion of parameterized module, along with the capability for instantiating the parameters of such modules, and the capability for encapsulating existing code into modules. The Ada notion of a generic package provides much of what is needed [10]; of course, we also need that the module is actually available from a library in some more general form.

A feature not provided by the Ada generic package that contributes especially to the reliable use of the parameterized programming paradigm is to provide a careful definition of module interfaces, describing all of the resources needed by the module. Then correct instantiation of the formal parameter of a module is equivalent to placing that module into an environment in which it is guaranteed to function properly. (In the following, we will interchangeably use the metaphors of instantiating a parameter and interfacing with a module with an environment.) This kind of semantic documentation of module interfacing is also helpful in retrieving the right module from a library. Ada itself also provides no capability for reusing separately compiled program parts that have been developed in a top-down manner; this is because of limitations to the use of Ada's **separate** clause. Views and theories provide the formal apparatus to accomplish these goals, as described below and in [18].

It often happens that there is a software part that we want to reuse, but it is not in exactly the right form. One way to get it into the right form is to apply some program transformations. An important enhancement of the parameterized programming paradigm permits parameterized modules to be modified, either before or after instantiation, so that they can be fitted to a wider variety of applications. Among possible modifications are the following 1) *enrich* a module, by adding to its functionality; 2) *rename* some of the interface of a module; and 3) *restrict* a module, by eliminating some of its

Manuscript received August 1, 1983; revised May 14, 1984. This work was supported in part by a gift from the System Development Foundation and by the Office of Naval Research under Contract N00014-80-0296.

The author is with SRI International, Menlo Park, CA 94025 and the Center for the Study of Language and Information, Stanford University, Stanford, CA 94305.

[1] Ada is a registered trademark of the U.S. Government (Ada Joint Program Office) and is defined by [10].

0098-5589/84/0900-0528$01.00

functionality. All three of these can be accomplished with module expressions, guaranteeing the satisfaction of (selected) program properties given in the form of theories.

As an example of parameterized programming, consider a parameterized module LEX[X] that provides a lexicographic ordering on lists of X's where the parameter X can be instantiated to any set with a designated ordering relation.[2] Thus, if ID is a module that provides identifiers (and in particular, words) with their usual (lexicographic) ordering, then LEX[ID] provides a lexicographic ordering on sequences of words (and thus, for example, on book titles). Similarly, LEX[LEX[ID]] provides a lexicographic ordering on sequences of phrases (such as might be used in sorting a list of book titles), by instantiating the ordering that LEX[X] requires with the one that LEX[ID] provides, namely lexicographic ordering. Another example would be a module SLIST[Y] that (for example, bubble) sorts lists of Y's, for Y any set with a designated ordering relation; in particular, letting Y be LEX[LEX[ID]] gives a program SLIST[LEX[LEX[ID]]] that sorts lists of book titles.

Let us look at this example, a little more closely now, to see how theories and views are used. The formal parameter requirement theory for LEX is POSET, the theory of partially ordered sets: in order that LEX[M] be meaningful for some module M, it is necessary that there be a view of M as a poset, i.e., it is necessary to designate a sort and binary relation from M that satisfy the partial order axioms; in the case of ID, the lexicographic ordering relation is provided by such a view, LEX[ID] in turn provides another lexicographic ordering that corresponds to a view of it as a POSET. It is this view that makes it legal to instantiate LEX[X] with LEX[ID] to get LEX[LEX[ID]]. The notation LEX[X::POSET] is used to indicate that any actual parameter of LEX must satisfy POSET, i.e., must have a view as a poset. These views can be defined in advance of their use, and thus generally do not need to be mentioned at instantiation time. Often, as in this example, it suffices just to mention the actual parameter since there is a unique **determined view** of that module as the required theory (see Appendix B for details).

An important observation is that, in addition to having parameters, modules may also simply *use* other modules, that is, rely upon them being just as they are. Although this can be seem as a special case of parameter instantiation, it is more convenient to treat it separately; see Section II.

B. Language Features to Support Parameterized Programming

Not every programming environment will support parameterized programming equally well; the following are some features that seem especially helpful in supporting this technique. It is not necessary that all of these actually be features of the language; it suffices that they can be implemented as part of an environment for the language. This is the approach that we would advocate to provide fully parameterized programming for Ada; see [18]. Notice that many of these features can also be seen as desirable for supporting software reusability in general. Moreover, any feature that supports parameterized programming in general also supports what is sometimes called programming-in-the-large, i.e., program design; this is certainly the case with the features listed below.

1) Modularity: Breaking a program into parts each of which is "mind-sized" and has a natural function, maximizes conceptual clarity, modifiability, and ease of understanding; all these properties enhance resuability.

2) Hierarchical Structure: Particularly when modules are being instantiated and then reused in other modules, it is helpful to keep explicit track of the hierarchical structure of program development, showing which modules make use of which others.

3) Libraries: In order to make the most effective use of parameterized programming, it is necessary to actually get ahold of the module that can be instantiated to do what one wants. This will require an appropriate library facility.

4) Strong Typing: This helps catch inconsistencies of a design as it is developed, and prevents confusion between logically distinct concepts. Moreover, it can serve as the basis for error detection and exception handling by "abstract errors" [13], and is also useful in supporting overloaded mixfix syntax. All these contribute to reusability.

5) Parameterization: In order to maximize their reusability, software modules should be as highly parameterized as possible; this means that by substituting different parameters, a given module can be reused in many different ways. These parameters need not be just numbers or some other indexes that simply designate a particular member of a family of modules, but rather could be collections of other software modules; that is, the actual parameter of a parameterized module could be an environment. This leads to much greater reusability than mere indexical parameterization.

6) Requirements for Parameters: Reliability is a potential problem for parameterized modules if correctness of the module depends critically upon certain requirements being satisfied by the environment in which the module is used. To achieve reliability, it is necessary to know exactly which instantiations of the parameters are going to work. Reliable parameterized programming requires specifying the interface properties that must be satisfied in order for the module to work correctly.

7) Theories and Views: Similarly, it is very helpful to be able to declare properties of modules that can be relied upon when they are used as parameters to another module. The purpose of a theory is to declare such properties, and the purpose of a view is to indicate how a given module satisfies a given theory.

8) Information Hiding: In order to ensure that a program does not depend upon the way in which some abstraction is actually implemented, it should be possible to "hide" details of the implementation; this means ensuring that only operations in the declared interface can be used by other modules

[2] In general, a module can define one or more data structures, and can provide one or more operations upon those structures, possibly making use of other data structures and operations that are provided by other modules. For example, **LEX[X]** might provide both a binary relation, **1st1 < 1st2** meaning that **1st1** comes earlier in the lexicographic order than **1st2**, and a unary predicate, **lex(1st)**, indicating that the list **1st** is lexicographically ordered. A module may also have internal states, although we do not discuss this issue in this paper.

[43]. This principle of information hiding, also called abstraction, includes the notion of abstract data type.

9) Module Modification: Sometimes it is necessary to modify a module before it can be reused, for example, to change the syntax or semantics of some operations, to add some new functionality, or to delete some old functionality. Providing language features to accomplish such tasks adds significantly to the power of parameterized programming and significantly enhances reusability.

10) Simplicity: An economical and conceptually coherent syntax and semantics will maximize one's intuitive grasp of program text. Programs are then easier to understand and to read, and thus easier to reuse. One way to achieve this simplicity is to base the denotational semantics of a language on some simple logical system, implemented by a correspondingly simple operational semantics. This means that the specification and execution levels of program comprehension are identified, so that a logical axiomatization leads directly to a pattern of computation. This is the key idea of so-called "logic programming" as, for example, in Prolog [8], [48].

11) Formal Semantics: A simple underlying semantics for a language will greatly enhance the understandability of programs. It is also essential for program verification, and for the retrieval of modules from a library using semantic keys.

12) Interactive Program Development: Interactive programming will maximize the ease of actually doing parameterized programming. This includes configuration and version management, interactive structured editing, and running simple test cases.

The advantages of modularity and abstraction (including data abstraction) lie primarily in the control of detail; this is particularly important in view of the numerous changes that inevitably accompany large development efforts. The requirements associated with parameterized modules provide a kind of high level documentation; this can reduce the possibilities for misunderstanding, and moreover, can also facilitate program debugging, maintenance and reuse, as well as library access.

C. The OBJ Project

This paper illustrates parameterized programming with some simple examples written in the OBJ programming system now under development at SRI. The version used in this paper builds upon the earlier experimental OBJT and OBJ1 implementations; it has much more powerful and descriptive mechanisms for program parameterization, and replaces the error algebra approach to exception handling with a simpler approach based upon subsorts. OBJ was originally designed by Goguen [13], and first implemented by Tardo [25], [47] as OBJT. OBJ1 was implemented at SRI by D. Plaisted and runs under TOPS-20 or TENEX [26]. Both OBJT and OBJ1 are interpreters written in UCI-Rutgers LISP; it would be much more efficient to write the most critical parts of an OBJ interpreter in machine code, and better still to write an OBJ compiler. We believe that OBJ would run very fast indeed on a machine that could execute rewrite rules directly, and that such a combination would exceed the performance of more conventional languages on conventional machines to the extent that the rewrite rule machine could support the excecution of rewrite rules in parallel. The extensive use of structure sharing and (at the user's discretion) of hash addressing for selected terms, render OBJ1 already competitive for some applications, in particular, for rapid prototyping [21]. This permits the functional behavior of a system to be observed early in its development cycle, which can lead to significant improvements and clarifications in the requirement, specification, and design phases, thus making it possible to deliver better code sooner. OBJ has been used in quite a number of different applications, including database systems [25], modest programming languages [21], [24] and some simple secure message and secure operating systems.

An OBJ program is a sequence of "objects," each of which may define one or more new sorts of data, together with associated operations that may create, select, interrogate, store, or modify data. Such an object may use existing objects with their sorts of data and operations. The object concept includes both "types" in the programming language sense (that is, a domain of values of variables together with operations that access those values) and algorithms. These ideas are in close conformity with methodology espoused for example by Jackson [36]. OBJ may be the only programming language to use this kind of abstraction as its fundamental mechanism for structuring programs.

Here is a summary of some main features of OBJ: 1) strong typing, with subtypes; 2) user definable abstract objects (including abstract data types); 3) user definable "mixfix" syntax, with overloaded operations; 4) parameterized abstract objects; 5) libraries; 6) exception raising and handling that can define tight and informative boundaries for acceptable computations; 7) theories, that declare properties of modules; 8) views, that connect theories to modules; 9) full associative pattern matching, that can be used to define pattern driven demons; 10) commands for modifying objects, making it possible to apply a broad range of (data type based) program transformations right inside of programs; and 11) powerful interactive programming and debugging aids, such as an editor that helps you get expressions to parse before permitting them to be executed.

OBJ is based upon a simple logical system, namely equational logic; moreover, these high level descriptions of what a program does *actually are* the program; that is, one can execute them. Thus, OBJ is a "logic programming language," as are Prolog [9], pure LISP [40], and CDS [3]. As has been amply demonstrated by Prolog, this confers certain important benefits: program transparency (which eases program modifiability); separability of logic from control; and identity of program logic with proof logic (which eases program proving and program understanding).

Another language based upon rewrite rules is Hope [7]. Affirm [31] contains a system for the symbolic execution of abstract data types, and TEL [38] is another interesting early system based on rewrite rules. Still another related system is described by Lucas and Risch [39]. The elegant work of Backus [1] is also related. Hoffman and O'Donnell [32] have been developing an efficient execution algorithm for a special class of rewrite rules.

II. Hierarchical Structure

Conceptual clarity and ease of understanding are greatly facilitated by breaking a program into modules, each of which is mind-sized and has a natural function. This in turn greatly facilitates both debugging and reusability. When there are a significant number of modules, it is helpful to keep track exlicitly of the hierarchical structure of module dependence, showing exactly which modules make use of which others. The collection of other modules used by a given module, together with the dependence relations among them, consistute the immediate environment of the given module.

In order to make this structure as explicit as possible, whenever a module uses data or operations from another module, that other module should be explicitly mentioned and also have been defined earlier in the module sequence of the program. A program developed in this way has the explicit structure of a hierarchy, or more precisely, an *acyclic graph*, of abstract modules.[3] An *environment* is just such an acylic graph structure, and the *context environment* of a given module is the subgraph of other modules upon which it depends. It is important to notice that both parameterized and instantiated modules can occur in such a hierarchy, and are treated in essentially the same way. (The main difference is that only fully instantiated modules can be executed or compiled.)

In addition to its basic function of representing one aspect of program structure in an especially clear and convenient manner, the module hierarchy structure can be used for a number of other particular purposes. For example, it can be used to maintain multiple mutually inconsistent structures as subhierarchies. This is useful for keeping available more than one way to do the same or related things, such as a family of partially overlapping system designs. It can also be used to keep information from different sources in different places, and to maintain multiple inconsistent worlds. This could be useful for exploring the consequences of various mutually inconsistent assumptions, in the context of an environment of shared assumptions. Hierarchical structure can also be used to reflect access properties of a physically distributed database; OBJ can also be useful in this context for describing the different data representations used at different sites, and even for providing ways to translate among them [17].

The most basic part of the syntax of the OBJ language is concerned with this hierarchical structure. An OBJ object begins with the keyword OBJ (or OBJECT) and ends with ENDO (or ENDOBJ or JBO or TCEJBO[4]). The name of the object being defined must be placed immediately after OBJ keyword; following this name comes the list of names of the objects that are *used* in this object, separated from the object name by a "slash" symbol, /, that can be pronounced "over." Of course, if the current object depends upon no other objects, then the slash and name-list can be omitted. Moreover, if an object is mentioned in the used object list, then there is no need to mention any of the objects that it uses, as all these are already included by convention (that is, "used" is considered a *transitive* relation, so that a given object may use any object that is used by any object that it uses).[5] After this comes the body of the object, details of which are discussed later in this paper.

For unparameterized objects, the name is a simple identifier, such as STACK-OF-INT, PHRASE, OBJ14 or CARD. Parameterized objects have more complex names, as discussed in Section V. Optionally, the name of the object can be repeated after the object ending keyword; this enhances readability in the case of nested objects (discussed in Section II-C). For example

```
OBJ PHRASE / LIST
...
JBO PHRASE
```

A. Built-Ins

A programming language will usually provide, for efficiency if for no other reason, a number of built-ins, for example, basic data types such as integer. In a sufficiently powerful language supporting user definable modules, it is not necessary to provide any built-ins because anything desired can be defined as needed. But of course building in the most frequently needed modules can make a large difference in both efficiency and convenience.

OBJ has built-in objects TRUTH, BOOL, NAT, INT, and ID. TRUTH provides the two truth values T and F that are used by the built-in equality and conditional operations (see Section III-B). Because of this, is is assumed that every object uses TRUTH whether or not it is explicitly mentioned following the slash after the object name. The built-in object BOOL is an enrichment of TRUTH that provides the syntax and semantics expected for Booleans (e.g., infix associative AND and OR, prefix NOT), as do NAT and INT for natural numbers and integers, respectively. ID provides identifiers, with only the operations of equality and lexicographic order built-in. These identifiers must begin with the apostrophe symbol, e.g., 'A, 'B, '1040, and 'VERYLONGIDENTIFIER.

B. Libraries

Clearly, in order to reuse software, it must first be available. This requires not only that the code itself can be obtained, but also that it is clear what the code is supposed to do, and what kind of environment is needed for the code to do it correctly. Libraries provide the means for storing and retrieving modules. Issues of documentation are discussed later. The important issue of preserving a module's relationship to its context is considered here.

[3] This hierarchy differs from a Dijkstra–Parnas hierarchy of abstract machines because higher level modules are not *implemented* by lower level "less abstract" machines; rather, higher level modules are elaborations or *enrichments* of previous lower level modules. (Note that in this paper, the roots of the hierarchy graph are visualized as being at the bottom; these root modules embody the built-in capabilities of OBJ, such as its basic data types.)

[4] With tongue-in-check deference to popular convention in programming language design, OBJ admits ending keywords that are the backward spelling of the corresponding beginning keywords; alternatively, one may think of TCEJBO as Polish for OBJECT. A more helpful uniform convention in OBJ is that an ending keyword can always be of the form END ⟨X⟩ where ⟨X⟩ is the first letter of the corresponding beginning keyword.

[5] This default convention can be modified by information hiding, as discussed in Section III-D. An alternative convention is used in HISP [11].

The basic OBJ library unit is the **file**, which may contain one or more objects; a file may also contain top level OBJ commands. The basic way of processing such a library file in OBJ is to execute everything in it. In the case of objects, this means checking for internal consistency, as well as consistency with the current environment. Thus, the context of a given object can be preserved, by storing the other modules that it depends upon earlier in the same file.

OBJ provides a number of library commands. The IN command reads in and executes an arbitrary sequence of files; for example,

```
IN LIBRARY62 MYSYS TEST4 ENDI
```

reads in the OBJ files LIBRARY62, MYSYS, and TEST4 adding to the current environment whatever objects they contain, and executing whatever commands they contain. The GET command will get an arbitrary sequence of objects from a named file and add them to the current environment. For example,

```
GET STACK ARRAY FROM ALIBRARY ENDG
```

will get the objects STACK and ARRAY from ALIBRARY.

The fact that files can contain other top level OBJ commands as well as object definitions makes for great convenience and flexibility in actually using OBJ. For example, one can expand storage, call for a PHOTO to be taken, and execute some test cases, in addition to defining a particular multiobject environment. It seems especially appropriate that test cases should be stored along with their associated objects.

More elaborate library facilities, making use of views and theories for the retrieval of Ada code, are described in [18].

C. Nested Modules

It is very convenient to be able to nest modules within one another. This permits local modules, with local names, local data types, and local operations (however, note that some information hiding is required to accomplish this, see Section III-D). In the context of parameterized programming, note that all the parameters and imported (i.e., used) modules of an enclosing module are available for use in every nested module. Nesting taken together with parameterization and information hiding give a very potent generalization of the traditional notion of a block. OBJ's syntax for nesting is an obvious one.

```
OBJ ⟨name1⟩ / ⟨name-list1⟩
  ...
  OBJ ⟨name1.1⟩ / ⟨name-list1.1⟩
    ...
  JBO
  OBJ ⟨name1.2⟩ / ⟨name-list1.2⟩
    ...
  JBO
    ...
  JBO
```

III. Expression Syntax

We believe that it is worth some extra implementation effort and processing time to support syntax that is as flexible, as informative, and as close to users' intuitions as possible. In particular, users should be able to define prefix, postfix, infix, or more generally, **mixfix**[6] operations, in order to get a syntax that is maximally appropriate to a given problem domain.

Obviously, there are many opportunities for ambiguity in parsing such a syntax. OBJ's convention is that an expression is *well-formed* if and only if it has *exactly* one parse.[7] In keeping with the interactive nature of the language, the OBJ parser provides information about the difficulties that it encounters, and helps the user to correct them before attempting to parse them again.

A. Sorts and Subsorts

We believe that a programming language should have a strong but flexible type system; however, to avoid the confusion associated with the many uses of the word "type" we shall instead speak of "sorts" from here on to refer to the division of data into sets and subsets of items of the same kind. Among the advantages of *strong sorting* are:

- it helps to catch meaningless expressions before they are executed;
- it supports overloading in syntax; and
- when the notion of subsorts is added, we get
  - -- the utility of coercions without the associated confusion found in many programming languages,
  - -- as well as very convenient forms of error handling, and even
  - -- backtrack programming (see [26]).

We have been experimenting with sort systems in OBJ for some time, trying to find an approach that raises minimal difficulties; our original approach based on "error algebras" is given in [13], and has been elaborated by Plaisted [45]. An alternative is given by Reynolds in [46]. Still another approach is based upon polymorphism as introduced by Milner [41] and implemented in ML [29] and Hope [7].

This paper uses an approach based on the notion of a *subsort*. For example, the sort NAT is a subsort of the sort INT; moreover, a module may declare the sort INT to be a subsort of LIST-OF-INT. Then, the integer 2 can be coerced to the list with just one element, 2. The form of an OBJ subsort declaration is

⟨*sort1*⟩ < ⟨*sort2*⟩

which means semantically (that is, in terms of models) that the set of things of ⟨*sort1*⟩ is a *subset* (not necessarily a proper subset) of the things of ⟨*sort2*⟩. The form

⟨*sort-list1*\,⟩ < ⟨*sort-list2*\,⟩

can also be used, meaning that each sort in <*sort-list1*> is < each sort in <*sort-list2*>, and that the elements of the two lists are separated by commas.[8]

[6] This term is due to P. Mosses. The word "distfix" is also used, an abbreviation for "distributed fix."

[7] An elaboration required by subsorts is discussed in Section III-A.

[8] The metasyntactic notation ⟨*a-listn**x*⟩ indicates the *n*'th list of *a*'s, with *x*'s as separators; if there is no *x* there the separator is assumed to be a space; also a "carriage return" can always be used as a separator instead of an *x* or space.

The OBJ syntax skeleton for the concepts so far introduced is

```
OBJ <name> / <name-list>
   SORTS <sort-list>
   SUBSORTS <subsort-decl-list\;>
   ...
JBO
```

and is illustrated by

```
OBJ LIST-OF-INT / INT
   SORTS LIST
   SUBSORTS INT < LIST
   ...
JBO
```

OBJ checks for cycles of subsorts, and complains if it finds any. But OBJ does not complain about ambiguities introduced by the subsort relation; it simply regards each expression as having the *smallest* possible sort in the partial ordering of sorts induced by the given subsort relation. Foundations for this approach are given in [14] and [22]. A fashionable way to describe this feature is to say that OBJ's type system implements "multiple inheritance"; OBJ has had this feature from its earliest days [13].

Subsorts give an elegant way to treat errors. We illustrate this with a bounded stack example.[9] Really, this should be a parameterized object, with both the elements to be stacked and the maximum depth of the stack as parameters; however, because we have not yet introduced the machinery for parameterization, we take the concrete case of stacks of INT's of depth less than 10 000. The sort STACK includes expressions to describe the results of overflowing this bound, as well as correct stacks in the subsort OK-STACK , and nonempty OK-STACKS in the deeper subsort NE-STACK. The expression following SORT-DECLS defines the overflow condition; its semantics is described in [14].

```
OBJ BSTACK-OF-INT / INT NAT
   SORTS STACK
   SUBSORTS NE-STACK < OK-STACK < STACK
   OPS
      EMPTY : STACK
      PUSH : INT STACK - > STACK
      POP : NE-STACK - > OK-STACK
      TOP : NE-STACK - > INT
      DEPTH : OK-STACK - > NAT
   VARS
      I : INT ; S : OK-STACK
   SORT-DECLS
      (AS  NE-STACK : PUSH (I, S)  IF  DEPTH (S) < 10000)
   EQNS
      (DEPTH (EMPTY) = 0)
      (DEPTH (PUSH (I, S)) = INC (DEPTH (S)))
      (POP (PUSH (I, S)) = S)
      (TOP (PUSH (I, S)) = I)
ENDO
```

[9] This example draws on earlier versions due to Jouannaud and Meseguer, which in turn drew on [14].

If desired, DEPTH could be made a hidden operator; see Section III-D. Notice that without resorting to such relatively untractable devices as partial functions, we get POP and TOP defined only on nonempty stacks; moreover, we get appropriate error messages when they are applied to overflown stacks.

B. Operation Syntax

This subsection discusses one way of supporting user definable mixfix syntax. It seems clear that the sorts of the arguments and value of an operation should be defined at the same time that its syntactic *form* is defined. We distinguish two cases. The first provides a simple way to get the usual functional notation. For example

```
F : S1 S2 - > S3
```

indicates the usual parenthesized-prefix-with-commas notation, as in F(X, Y) of sort S3 for X of sort S1 and Y of sort S2. (It is obligatory to use commas as separators in well-found expressions using this syntactic form.)

The second case uses place-holders, indicated by an underbar,_, as in the prefix declaration

```
TOP_ : STACK - > INT
```

for **TOP** as used in expressions like TOP PUSH (A, B). Similarly, the "outfix" form of the set singleton operation, as in { 4 }, is indicated by

```
{_} : INT - > SET
```

and the infix form for addition, as in **2 + 3**, is

```
_ + _ : INT INT - > INT
```

while a mixfix declaration for conditional is

```
IF _ THEN _ ELSE _ FI : BOOL INT INT - > INT
```

One such conditional operation is provided for each built-in sort, and also for each user declared sort.

Between the : and the - > in such a syntax declaration comes the *arity* of the operation, a list of sorts each of which must have been previously declared; after the - > comes the *value sort* of the operation. The general format for these syntactic declarations is

⟨*form*⟩ : ⟨*sort-list*⟩ - > ⟨*sort*⟩

where a ⟨*form*⟩ is a non-empty string of ⟨*identifier*⟩'s and underbars, having exactly as many underbars in the ⟨*form*⟩ as sorts in the ⟨*sort-list*⟩. The form having just one underbar is excluded, as this corresponds to a subsort declaration, for which the notation already given is preferred (although previous versions of OBJ took the opposite point of view). Constants are declarations with empty arity. The *rank* of an operation consists of its arity plus its value sort. Operations that have the same rank but different forms can be declared together, for example

```
ONE, ZERO : - > S
```

and

```
_+_,_*_ : S S - > S
```

The *signature* of an object consists of its sorts, its subsort relation, and its operations, where each operation has a form, arity, and value sort.

Although strong sorting often serves to prevent expressions that use overloaded operators from being ambiguous, sometimes it does not suffice; one may then qualify expressions. For example, one might write (X (IS-IN. BAG) SET1 UNION SET2) to indicate that the expression is supposed to be a natural number (as used for bags) rather than a truth value (as used for sets).

C. Attributes

It is convenient to consider certain properties, such as associativity, commutativity and identity, as *attributes* of an operation. Although such properties are closely associated with the syntax of a given operation, they are also in part semantic properties.

In OBJ, such attributes are declared in parentheses after the syntax declaration of an operation. Binary infix operations can have an ASSOCIATIVE attribute (which can be abbreviated ASSOC); for example,

```
_OR_ : BOOL. BOOL -> BOOL (ASSOC)
```

indicates that OR is an associative binary infix operation on truth values. This means that:

- the parser does not require full parenthesization--for example the term (T OR (F OR T)) can be written (T OR F OR T);
- the deparser will omit unnecessary parentheses; and
- we also get the effects of an associativity equation.

An identity attribute can be declared for a binary infix operation. For example, in

```
_OR_ : BOOL, BOOL -> BOOL (ASSOC ID: F)
```

the attribute ID: F gives us the effects of the identity equations (B OR F = B) and (F OR B = B).

Binary infix operations can also be given the COMMUTATIVE attribute, abbreviated COMM. This is equivalent to adding a commutative equation, and is implemented by sorting arguments according to lexicographic ordering. Finally, operations can be IDEMPOTENT, abbreviated IDMPT.

For each sort S, there is a built-in *equality* operation with syntactic form

```
_==_ : S, S -> BOOL
```

Any binary BOOL-valued operation can be given the attribute (EQUALITY), which makes it equal to the built-in equality. For example,

```
IFF : BOOL BOOL -> BOOL (EQUALITY)
```

Similarly, any BOOL-valued binary operation can be given the attribute LEX. This declares it equal to the built-in lexicographic ordering operation on its arity sort. All of OBJ's built-in objects except TRUTH have such a built-in lexicographic ordering, denoted $<$; for user defined objects, it is determined by the order in which operations are declared, with earlier declared operations being earlier in the ordering.

An integer precedence attribute can be given to the parser; the higher the integer, the more binding the operation. For example, the built-in object INT has

```
_+_ : INT INT -> INT (ASSOC 5)
```

and

```
_*_ : INT INT -> INT (ASSOC 8)
```

so that the expression A + B * C is parsed as expected, as A + (B * C).

D. Information Hiding

A basic problem in programming is to control the complexity of large programs. One of the most useful ideas of modern programming methodology is intended to address just this problem. It is variously called "information hiding," "abstraction," and "encapsulation" [43], [44]. For example, the representation of an abstract data type in an Ada package may be hidden. Information hiding goes a step beyond modularization to permit declaring that some of the information inside a module *cannot* be assessed outside that module. Using this tactic ensures the important property that if it is desired to change a data representation, all that is required is to reimplement all the operations that the module provides, using the new representation; one does not have to search through the entire program for subtle uses of the old representation because there cannot be any; this is because only the operations that the module actually exports can be used outside of it.

There are several different approaches for hiding information in a strongly sorted modularized language. For example, one can declare operations and/or sorts to be hidden; one can use the convention that everything is visible unless declared hidden, or one can take the opposite view. One also may or may not permit a sort visible at one level of nesting to become hidden at some enclosing level. When an operation is declared *hidden*, an expression containing that operation is considered syntactically correct *only inside* the module where it is declared; in particular, no other module can use it, not even a module that depends upon the one in which the operation is declared.

In OBJ, sorts and operations are visible unless declared otherwise. Hidden sorts are declared after the keyword H-SORTS, and hidden operations after H-OPS. When a sort is declared hidden, every operation that involves that sort is automatically considered hidden, and therefore does not need to be so declared explicitly. If a sort or operation is visible in some nested object, and if it is desired at a given level of nesting to prevent it from being exported to further enclosing levels, this can be accomplished by redeclaring that sort or operation as hidden at the given level. However, a sort or operation that is hidden cannot be redeclared visible at an enclosing level.

Sometimes one may want to hide more than half of the sorts or operations of a module; then it is more convenient to list what is to be visible than to list what is to be hidden. This is accomplished by listing the sorts to be visible following the keyword V-SORTS, and similarly V-OPS for visible operators. One can also have mixed cases, such as H-SORTS with V-OPS.

E. Interactive Syntax

It is a considerable help in programming if simple semantic checks can be run at entry time, rather than only after com-

pilation. This is one motivation for structured editors and similar facilities. In fact, by the time an OBJ program gets to the stage where it can be executed, it has survived a great deal more checking than most compilers even try to provide.

In OBJ, when an expression fails to parse or has more than one parse, it is immediately reported along with any diagnostic information generated by a spelling checker that looks for and reports tokens (and token sequences, for the mixfix case) that have not been defined. The implementation is incremental and interactive: assuming that OBJ is reading from a file (using the IN_ENDI command), when an error is detected, comments are inserted into the file (using OBJ's ***_*** comment syntax) and the user may subsequently edit that file, working onward from the object where the error was first detected. When the editor (which is EMACS) is exited, OBJ again tries to execute these objects. Objects that were previously accepted are left unchanged; however there are commands to undo and edit any desired objects, or even whole files. Moreover, there is a command (at the operating system level) to resume executing the session that was last exited.

OBJ has a HELP facility, and also a PHOTO command to record in a designated file the objects and runs of a given session. (Many details and several commands have been omitted in this discussion; see [26].) In summary, user experience has shown that OBJ1's sophisticated interactive user interface tremendously improved programmer productivity over the previous version, by reducing the time required to find spelling and other syntax errors, and by eliminating the need to reprocess objects upon which an object containing a corrected error depended.

IV. Rewrite Rules

So far we have considered only declarations, either concerned with interface specification or else with expression syntax. But every programming language must have an operational component. OBJ's is rather unusual, being based upon rewrite rules. These are written declaratively as equations, but are interpreted as rules for replacing one subexpression by another. This is a completely general programming formalism, because Bergstra and Tucker [2] have shown that any computable function can be realized in this manner; see also [23]. It is especially convenient for defining abstract data types because of all the research that has been done on algebraic approaches to data abstraction ([15], [23], [27], [30], [50]), and it also seems to be a natural formalism for expressing the production rules used in expert systems; in fact, we believe that rewrite rules are a simple and natural way of describing any kind of nonnumerical processing.

For a simple example of how equations are interpreted as rewrite rules,

```
POP (PUSH (I, S)) = S
```

is used as a rule to rewrite the term TOP (POP (PUSH (2, PUSH (1, EMPTY))) to the term TOP (PUSH (1, EMPTY)) by replacing POP (PUSH (2, PUSH (1, EMPTY))) by PUSH (1, EMPTY). I and S in this rule are *variables*, I of sort INT and S of sort STACK. OBJ requires that all such variables be declared before they are used.

The syntactic form of a rewrite rule in OBJ is

$(\langle exp1 \rangle = \langle exp2 \rangle)$

where $\langle exp1 \rangle$ and $\langle exp2 \rangle$ are both well formed OBJ expressions, possibly using previously declared variables. In addition, there are *conditional rewrite rules*, of the form

$(\langle exp1 \rangle = \langle exp2 \rangle$ IF $\langle bexp \rangle)$

where $\langle bexp \rangle$ is an expression of sort BOOL. Such a rule can be thought of as a "pattern driven demon" that fires only when its $\langle bexp \rangle$ is true.

A. Some Examples

Let us illustrate this with a simple LIST-OF-INT object.

```
OBJ LIST-OF-INT / INT NAT BOOL
   SORTS LIST
   SUBSORTS INT < LIST
   OPS
      _._ : LIST LIST -> LIST (ASSOC ID: NIL)
      EMPTY?_ : LIST -> BOOL
      LENGTH_ : LIST -> NAT
      VARS L : LIST
           I : INT
      EQNS
         (LENGTH NIL = 0)
         (LENGTH I = 1)
         (LENGTH I . L = INC (LENGTH L))
         (EMPTY? L = L = = NIL)
JBO
```

We now evaluate some expressions from this LIST-OF-INT object. An expression E to be evaluated is presented to OBJ in one of the three forms

```
(E)
RUN E NUR
RUN E ENDR
```

The value is computed by matching the expression with the lefthand sides of equations, and then replacing the matched subexpression with the corresponding substitution instance of the righthand side, i.e., evaluation proceeds by applying the rewrite rules. For example,

```
RUN LENGTH 17 . NIL . -4 ENDR
```

gives

```
AS NAT: 2
```

by the following sequence of rewrite rule evaluations

```
(LENGTH 17 . NIL . -4) =>
(LENGTH 17 . -4) =>
(INC (LENGTH -4)) =>
INC (1) =>
2
```

where the first step is by identity property of NIL, and the second uses the rule with lefthand side (LENGTH I . L), matching I to 17 and L to -4; this last match works because the integer -4 can be regarded as a LIST since INT is a subsort of LIST. It is now possible to apply the equation (LENGTH I = 1) to this

singleton list, with I = -4. Then finally the built-in operations for NAT give the answer 2.

For another example,

```
RUN EMPTY? 17 . NIL . (3 . 4) ENDR
```

gives

```
AS BOOL: F
```

by the following sequence of reductions

```
(EMPTY? 17 . NIL . (3 . 4)) = >
(EMPTY? 17 . 3 . 4) = >
(17 . 3 . 4 = = NIL) = >
F
```

where the first step uses the ASSOC and ID attributes, and the last step uses the built-in equality on LIST-OF-INT.

Here are two further test cases for LIST-OF-INT, illustrating the effect of ASSOC:

```
RUN 1 . 2 . (3 . 4) NUR
```

gives

```
AS LIST-OF-INT: 1 . 2 . 3 . 4
```

and

```
RUN 3 . 4 . ((3 + 7) . (3 * 8)) NUR
```

gives

```
AS LIST-OF-INT: 3 . 4 . 10 . 24
```

Now an example of how a rewrite rule operational semantics provides a decision procedure for a theory of practical interest, the propositional calculus; this decision procedure is due to Hsiang [33] and was programmed in OBJ by D. Plaisted [26]. The rules in the object PROPC below reduces valid propositional formula in the connectives OR, IMP, NOT, EQUIV and AND, to VALID, and reduce all other formulas to a canonical form in the connectives +, * and NOT (note that + here is exclusive or).

```
OBJ PROPC / ID
  SORTS S
  SUBSORTS ID < S
  OPS
    _+_ : S S - > S (ASSOC COMM 10 SAVERUNS ID:
      ZERO)
    _*_ : S S - > S (ASSOC COMM IDMPT 11 SAVERUNS
      ID: VALID)
    _OR_ : S S - > S (ASSOC 6)
    _IMPLIES_ : S S - > S (4)
    NOT_ : S - > S
    _EQUIV_ : S S - > S (2)
    _AND_ : S S - > S (11)
  VARS X Y Z : S
  EQNS
    (X AND Y = X * Y)
    (X OR Y = X * Y + X + Y)
    (X IMPLIES Y = X * Y + X + VALID)
    (X EQUIV Y = X + Y + VALID)
    (NOT X = X + VALID)
    (X * ZERO = ZERO)
    (ZERO * X = ZERO)
    (X + X = ZERO)
    (X * (Y + Z) = X * Y + X * Z)
    ((X + Y) * Z = X * Z + Y * Z)
JBO
```

Giving an operation the SAVERUNS attribute causes the results of evaluations of terms headed by this operation to be saved. Thus the work of reduction is not repeated if the term appears again. The user may give any operations that he wishes the SAVERUNS attribute. OBJ uses hashing to implement this very efficiently; this is an area in which term-rewriting systems have an advantage over unification based systems like Prolog. SAVERUNS also causes OBJ to use structure sharing for common subexpression, which can greatly reduce storage requirements in some problems. Both the idea and the implementation are due to D. Plaisted.

Now some sample runs in the context of the PROC object.

```
RUN ('A IMPLIES 'B) EQUIV ((NOT 'B) IMPLIES (NOT 'A))
  ENDR
AS S: VALID

RUN (NOT ('A OR 'B)) EQUIV ((NOT 'A) AND (NOT 'B))
  ENDR
AS S: VALID

RUN ('C OR ('C AND 'D)) EQUIV 'C ENDR
AS S: VALID

RUN 'A EQUIV (NOT 'B) ENDR
AS S: 'A + 'B

RUN 'A * 'B + 'C + 'B * 'A ENDR
AS S: 'C
```

B. Order of Evaluation

It is necessary to consider certain further points before accepting that an object really does what it is supposed to do. First, we need to consider the order that rewrite rules will be applied, and whether or not that order makes any difference; a set of rules with the desirable property that the order of application of the rules does not matter, is said to be Church-Rosser. Secondly, we need to know that the process of applying rewrite rules will actually terminate; when it always does so, the set of rules is said to be *finite terminating*. [34] and [35] give good general discussions of term rewriting systems; see also [23], [37], [42].

In the case of the propositional calculus decision procedure given in Section IV-A, the rules have been shown by [33] to be Church-Rosser and finite terminating *modulo* the equations that correspond to the attributes ASSOC and COMM; the ID: attribute should be taken as abbreviating an identity equation for this purpose. Secondly, we should show that the OBJ implementation using this particular combination of rewrite rules and attributes is really correct. The difficulties here center on the distributive law, and are discussed in [26]. Here, we just note that it is necessary to give both versions of the distributive law, even though there is a commutative law

for *. This is because OBJ's COMM attribute is implemented with lexicographically ordered lists of values, so that one cannot be sure which of the two distributive laws may apply to a given case.

V. Parameterization

The basic building blocks of parameterized programming are parameterized modules. Three concepts that go beyond Ada generic packages and still seem quite practical are theories, views, and module expressions. *Theories* are used to define the properties required of an actual parameter for it to be meaningfully substituted for the formal parameter of a given parameterized module. *Views* are used to express that a given module satisfies a given theory in a particular way (this is necessary because it is possible for some modules to satisfy some theories in more than one distinct way). *Module expressions* are used to modify modules, by adding, deleting or renaming functionality. The final basic ingredient is the *instantiation* of a parameterized module to an actual parameter, using a particular view; this results in creating a new module. Our approach to parameterization is inspired by the Clear specification language [4], [5].[10]

A. Theories

The purpose of a theory is to express properties of a module (or module interface) as a whole. This subsection considers theories, while the next two subsections consider how theories are related to modules by formal parameters and by views, respectively. This is in direct contrast to the usual "assertions" of program verification, which talk about the state changes that occur when a statement (or sequence of statements) is executed.

In general, OBJ theories can have whatever structure objects can have; the difference is that objects are executable, while theories define unexecutable properties. In particular, theories can use other theories, can use objects, can be parameterized, and can even have views. We next give some examples of OBJ theories. They all express properties that parts of a software environment ought to satisfy for a given module to perform correctly.

The first example is the trivial theory TRIV, which requires nothing of a model except that it have a sort; this sort is designated ELT in the theory.

```
TH TRIV
   SORTS ELT
ENDTH
```

The next theory is an enrichment of TRIV, requiring that models also have a given element of the given sort; this element is designated * in the theory.

```
TH TRIV * / TRIV
   OPS * : - > ELT
ENDTH
```

[10] In particular, the notion of view was developed in collaboration with R. M. Burstall for use in Clear, although it has not yet been published in that connection. Clear's approach was in turn inspired by some ideas in general system theory [12], [20], suggesting the systematic use of colimits.

This enrichment is equivalent to the following:

```
TH TRIV *
   SORTS ELT
   OPS * : - > ELT
ENDTH
```

Next, the theory of partially ordered sets (some would call these quasi-ordered sets, because there is no antisymmetric law). Its models have a binary infix BOOL-valued operation < that is reflexive and transitive.

```
TH POSET / BOOL
   SORTS ELT
   OPS _<_ : ELT ELT - > BOOL
   VARS E1 E2 E3 : ELT
   EQNS
      (E1 < E1 = T)
      (E1 < E3 = T IF E1 < E2 AND E2 < E3)
ENDTH
```

The theory of an equivalence relation also has a binary infix BOOL-valued operation; it is denoted EQ and is reflexive, symmetric and transitive.

```
TH EQV / BOOL
   SORTS ELT
   OPS _EQ_ : ELT ELT - > BOOL
   VARS E1 E2 E3 : ELT
   EQNS
      (E1 EQ E1 = T)
      (E1 EQ E2 = E2 EQ E1)
      (E1 EQ E3 = T IF E1 EQ E2 AND E2 EQ E3)
ENDTH
```

Finally, the theory of monoids. This will serve as a parameter requirement theory for an iterator that will yields sums and products over lists in Section V-D.

```
TH MONOID
   SORTS M
   OPS _*_ : M M - > M (ASSOC ID : I)
ENDTH
```

B. Parameterized Modules

Theories are the requirements that the actual parameters of a parameterized module must satisfy in order for an instantiated module to behave as desired. A theory must have been previously defined in the program before it can be used in this way. For example, here is part of a parameterized LIST object.

```
OBJ LIST [X : : TRIV] / NAT BOOL
   SORTS LIST
   SUBSORTS X < LIST
   OPS
      _._ : LIST LIST - > LIST (ASSOC ID : NIL)
      EMPTY? : LIST - > BOOL
      LENGTH _ : LIST - > NAT
   ...
ENDOBJ
```

In general, an object (or a theory) may have more than one parameter; this is indicated with the notation

```
[X : : TH1 ; Y : : TH2]
```

Here both theories, TH1 and TH2, and their corresponding formal parameters, X and Y respectively, may involve more than one sort, and of course may involve many operations among their various sorts.

Here is an example of a parameterized theory, the theory of vector spaces over a field F.

```
TH VECTOR-SP [F : : FIELD]
   SORTS V
   OPS
      _+_ : V V -> V (ASSOC COMM ID : O)
      _*_ : F V -> V
   VARS F F1 F2 : F
        V V1 V2 : V
   EQNS
      ((F1 + F2) * V = (F1 * V) + (F2 * V))
      ((F1 * F2) * V = (F1 * (F2 * V)))
      (F * (V1 + V2) = (F * V1) + (F * V2))
ENDTH
```

The instantiation of parameterized objects is discussed in Section V-D.

C. Views

The purpose of a view is to show explicitly how a given module (either object or theory) satisfies another given theory. It may not suffice to write LEX[NAT] to define a module giving a lexicographic ordering to LIST's of NAT's, because there are many different order relations that could be used on the natural numbers. The most obvious is the usual "less-than-or-equal," but "divides" and "greater-than-or-equal" are other possibilities. The purpose of a view of NAT as a POSET, indicated POSET => NAT, is to indicate just which ordering is to be used; the three choices of ordering mentioned above correspond to three different views of NAT as POSET.

More precisely, a *view* of an object A as a theory T consists of a mapping from the sorts of T to the sorts of A that preserves the subsort relation, and a mapping from the operations of T to the operations of A that preserves arity, value sort and certain attributes, including ASSOC, COMM, ID: and IDPT (if they are present), such that every equation in T is provable of every model of A.[11] The mapping of sorts is expressed in the form

```
SORTS (S1 IS: S1')
   (S2 IS: S2')
   ...
```

and the mapping of operations is expressed in the form

```
OPS (Op1 IS: Op1')
   (Op2 IS: Op2')
   ...
```

[11] Technically, a view is a theory morphism in the sense of [5].

where Op1, Op1', Op2, etc. may be either operation forms, or forms plus arity and value sort, if that is needed for disambiguation.

Thus, each mapping consists of a set of pairs. These two sets of pairs together are called a *view body*. The syntax for defining a view adds to this names for the source and target theory or object, and a name for the view. For example,

```
VIEW NATD IS: NAT AS: POSET BY:
   SORTS (ELT IS: NAT)
   OPS (< IS: DIVIDES)
ENDV
```

defines a view called NATD of NAT as a POSET; we may also use the notation NATD: POSET => NAT.

Every object has a default view as TRIV using the first sort in its ⟨*sort-list*⟩ (or the first sort of the first object that it is built upon if it doesn't have a first sort itself, and so on backward recursively); this sort is called the *principal sort* of the object (or theory). A *determined view* is one that will be used unless another is explicitly provided instead; many determined views are *default views*, determined by the *default rules* for omitting parts of view bodies that are given informally below, and more formally in Appendix II. For example, the default view TRIV => NAT is

```
VIEW NATV IS: NAT AS: POSET BY:
   SORTS (ELT IS: NAT)
   OPS (< IS: <) ENDV
```

More generally, when there is only one sort in the source theory T and the SORTS line of a view is omitted, it is assumed that ELT is paired with the principal sort of T. For example, in

```
VIEW NATD IS: NAT AS: POSET BY:
   OPS (< IS: DIVIDES) ENDV
```

the default (ELT IS: NAT) is assumed.

There is also a default convention for OPS, namely correspondences of the form (Op IS: OP) can be omitted. For example, under this convention, the default view of NAT as POSET has (ELT IS: NAT) and (< IS: <). Thus, according to these conventions, the default view of NAT as a MONOID is

```
VIEW NAT* IS: NAT AS: MONOID BY:
   SORTS (M IS: NAT)
   OPS (* IS: *)
      (I IS: 1)
ENDV
```

where we know that 1 is an identity for * in NAT because the ID: attribute is preserved by views. The following is a non-default view of NAT as a MONOID.

```
VIEW NAT + IS: NAT AS: MONOID BY
   OPS (* IS: +) (I IS: O)
ENDV
```

Actually, (I IS: O) could be omitted, again by preservation of the ID: attribute.

Next, a view that involves a derived operation:

```
VIEW NATG IS: NAT AS: POSET BY:
  VARS N1 N2 : NAT
  OPS (N1 < N2 IS: N2 < N1) ENDV
```

More generally, one might have any expression (of the correct sort) on the right-hand side of the IS:; it is also necessary to declare the variables that are used in these derived expressions.

There is a useful generalization of the default rule that permits omitting an operation pair of the form (OP IS: OP), namely a pair (OP IS: OP') can be omitted provided that the arity and sort of OP' equal the translations (under the SORTS mapping) of those of OP, and OP' is the *only* operation (in its object or theory) having that particular arity and sort.

Sometimes it may be impossible to tell which if several operations or sorts with the same name, but from different objects or theories, is actually meant. For this purpose, *qualifiers* can be used to disambiguate the expression involved by providing the additional information of the object or theory where an operation is declared. For example

```
OBJ UNION [X Y : : TRIV *]
  SORTS UNION
  SUBSORTS (ELT . X), (ELT . Y) < UNION
    VIEW AS: TRIV * BY: (* IS: (* . UNION)) ENDV
  OPS * : - > UNION
  EQNS
    ((* . X) = (* . Y))
    ((* . UNION = (* . X))
ENDO
```

Qualifiers can also be applied to mixfix operations, as in

```
X (IS-IN . M47) FILE
```

where M47 is some module.

It is sometimes desirable to include views inside objects, as in

```
OBJ LEXL [X : : POSET] / LIST [X]
    VIEW AS: MONOID BY OPS (* IS: .) ENDV
  OPS _ < _ : LIST LIST - > BOOL
  . . .
ENDO
```

Once a view as T is included inside an object A, then the determined view of A as T becomes the included one; see the more detailed discussion in Appendix B.

D. Instantiation

To actually use a parameterized module, it is necessary to instantiate it with an actual parameter. This subsection considers how this might be done, and in particular introduces a number of convenient conventions.

The MAKE command applies a parameterized object to an actual, by use of a view; if the name of an object is used instead, the default view of that object as the requirement of the parameterized object is used if there is one. For example,

```
MAKE NATLIST IS: LIST [INT] ENDM
```

uses the default view TRIV = > NAT to instantiate the parameterized module LIST with the actual parameter NAT. Similarly, we might have

```
MAKE RAT-LIST IS: LIST [RAT] ENDM
```

where RAT is the field of rational numbers, using a default view TRIV = > RAT; also

```
MAKE RAT-VSP IS: VECTOR-SP [RAT] ENDM
```

uses a default view FIELD = > RAT, and

```
MAKE RAT-LIST-LIST IS: LIST [LIST [RAT]] ENDM
```

uses two default views.

To illustrate the case where an explicit view is used, let P[X : : POSET] be a parameterized object. Then we can form

```
MAKE P-NATD IS: P [NATD] ENDM
```

using the view NATD from Section V-C.

It is sometimes desirable to use (after the slash) module expressions to identify objects (or theories) used in the body and in the actual parameter. For example,

```
OBJ LEXL [X : : POSET] / LIST [X]
  OPS _ < < _ : LIST LIST - > BOOL
  VARS L L' : LIST
    E E' : ELT
EQNS
  (E < < NIL = F)
  (NIL < < L = NOT L = = NIL)
  (E . L < < E' . L' = IF  E = = E'  THEN L < < L' ELSE
  E < E' FI)
ENDO
```

in which LIST[X] uses the default view of X as TRIV, and provides lists of the actual parameter's principal sort. More generally, one might have

```
OBJ P1[X : : TH1] / P2[VIEW]
```

with VIEW : TH2 = > TH1, where TH2 is the requirement theory of P2. The most general case is to use a list of module expressions; see Section V-E and Appendix A.

Now here is a very powerful module for defining iterators:

```
OBJ ITER [M : : MONOID] / LIST[M]
  OPS ITER : LIST - > M
  VARS E : M ; L : LIST
  EQNS (ITER (NIL) = I)
    (ITER (E ; L) = E * ITER (L))
ENDO
```

where LIST[M] uses the default view TRIV = > MONOID; note that I is the identity of M. We now use this object for two rather interesting examples.

```
MAKE SIGMA IS: ITER[NAT+] ENDM
```

sums a list of numbers, while

```
MAKE PI IS: ITER[NAT*] ENDM
```

multiplies a list of numbers. We think that these are impressively concise and clear programs for these functions. Note that this approach avoides the complexities of higher order operations, but still has an equivalent power and is moreover modular.

Similarly,

MAKE NATLEX IS: LEX[NAT] ENDM

uses the default view of NAT as POSET to give a lexicographic ordering on lists of natural numbers, and

MAKE NATLEXD IS: LEX[NATD] ENDM

orders lists of NAT's using the divisibility ordering on NAT's. Similarly,

MAKE PHRASE IS: LEX [ID] ENDM

uses the lexicographic ordering $<$ given on ID to give a lexicographic ordering on lists of identifiers, and thus in particular on titles of books; and

MAKE PHRASE-LIST IS: LEX [PHRASE] ENDM

uses the lexicographic ordering on PHRASE to give a lexicographic ordering on lists of book titles. This is really a very concise program for what is really a rather complex functionality.

E. Module Expressions

An important addition to parameterized programming is a capability for modifying parameterized modules in various ways. This makes it possible to apply a given module in a wider variety of circumstances. Among the possible modifications are: to restrict a module, by eliminating some of its functionality; to rename parts of the external interface of a module; and to enrich a module by adding to its functionality. These operations make possible a broad range of (data type based) program transformations right inside of programs. No other programming language that we know has such features as commands in the language itself. (Earlier versions of OBJ had a similar capability, but as theories were not used, it was somewhat dangerous, because there was no explicit statement of what properties the result might have.)

Module expressions are expressions which define new modules out of old ones by combining and modifying the old ones according to a specific set of operations. The simplest case is that of the *constant* expressions, which are just built in. These include BOOL, NAT, INT, ID, and REAL. The application of an $n$ary parameterized module to $n > 1$ actuals provides a way of forming module expressions that is already familiar. In general, the n actuals must be views from the n requirement theories that come along with the parameterized module; but these can be supplanted by default or other determined views, and therefore (sometimes) by just the names of the actual modules to be used.

Another built in n-ary module constructor is the n-ary TUPLE constructor, which can form an n-tuple of modules for each $n > 1$; all n of its requirement theories are TRIV. Thus, for example, TUPLE[INT, BOOL] is a module expression whose principle sort consists of pairs of an integer and a truth value. Another is TUPLE[LIST[INT], INT, BOOL].

We next consider an image construction, which uses a ⟨*view-body*⟩, that is, a sort mapping and an operation mapping, to create a new module (object or theory) from an old one, by renaming the parts of the old one according to the instructions in the view body. The syntactic form for this construction is

⟨*mexp*⟩ * ⟨*view-body*⟩

What this does is to create a new copy of ⟨*mexp*⟩, with its syntax modified as indicated in ⟨*view-body*⟩.

Functionality can be deleted from a module by the HIDE construction. The syntax is

HIDE SORTS ⟨*sort-list*⟩ OPS ⟨*op-list*⟩ IN ⟨*mexp*⟩.

It might also sometimes be more convenient to indicate which sorts and/or operations are to be visible. This could be done with the syntax

VISIBLE SORTS ⟨*sort-list*⟩ HIDE OPS
⟨*op-list*⟩ IN ⟨*mexp*⟩.

and its obvious variants.

Finally, we consider ways of adding functionality. The simplest is just to "add" together a number of modules, using the syntax

⟨*mexp*⟩ + · · · + ⟨*mexp*⟩

and a more complex possibility is

ENRICH ⟨*mexp*⟩ WITH SORTS ⟨*sort-list*⟩ SUBSORTS
⟨*subsort-decl-list*⟩ OPS ⟨*op-list*⟩ VARS ⟨*var-decl-list*⟩
EQNS ⟨*eqn-list*⟩ ENDEN

However, since this construction can be hard to read, it is recommended that it be avoided if possible, and it is not included in the syntax for module expressions given in Appendix A.

VI. Denotational Semantics of OBJ

Whereas the operational semantics of a programming language is useful for showing how computations are actually carried out, the denotational semantics of a language is useful for giving precise meanings to programs in a conceptually clear and simple way; in addition, it permits the already established proof theory of the underlying logical system to be utilized in proving properties of programs. In the case of OBJ, the denotational semantics is algebraic, as in the well-known algebraic approach to abstract data types; thus, the denotation of an OBJ object is an *algebra*, that is, a collection of sets with functions among them; and the well-known proof theory for equational logic can be used for proving properties of these functions.

This is not the place for a detailed explanation of the algebraic approach to abstract data types; for this, see, for example, [15], [27], [30], [50]. The basic idea is that algebraic equations involving the operations in the signature should *completely* define the results of executing the operations. The initial algebra approach [27], [28], uses as a model the unique (up to isomorphism) most representative algebra which satisfies the equations (there may of course be many other models). It is known that this *initial* algebra always exists; moreover, it provides a representation-independent standard of comparison for correctness, that is, it provides an abstract algebraic seman-

tics. [6] (see also [23]) shows that a model is initial if and only if it has the following properties:

1) no junk: all elements are namable using the given constant and operation symbols; and

2) *no confusion:* all propositions true of the model can be proved using the given propositions as axioms.

Under certain mild conditions, the rewrite rule operational semantics agrees with initial algebra semantics. These conditions are just that 1) there are no infinite sequences of rewrites (finite termination), and 2) all sequences of rewrites give the same final result (Church-Rosser). This is proved in [16]; see also [49]. A formal semantics for procedure application can be given by using colimits; see [19].

VII. Conclusions

We hope to have shown that parameterized programming is a powerful technique for the reliable reuse of software. In this technique, modules are parameterized over very general interfaces that describe exactly what properties are required of an environment for the module to work correctly. Reusability is enhanced by the flexibility of the parameterization mechanism, which allows other modules as parameters. Reliability is enhanced by permitting interface requirements to include more than purely syntactic information. This paper has introduced three new ideas that seem especially useful in supporting parameterized programming 1) *theories*, which declare global properties of program modules and interfaces; 2) *views*, which connect theories with program modules in an elegant way; and 3) *module expressions*, which produce a new modules by modifying existing modules. The latter can be considered a kind of generalization of program transformations that permits certain specific kinds of transformations that are semantically well-behaved, to be combined in a structured manner. These ideas have been illustrated with some simple examples in the OBJ programming language, but should also be taken as proposals for an Ada library system, for adding modules to Prolog, and as considerations for future language design efforts. OBJ is an ultra-high level programming language, based upon rewrite rules, that incorporates these three ideas, and many others from modern programming methodology.

Appendix A
Module Expressions

The purpose of this Appendix is to give syntax for some OBJ constructions concerned with views and hiding. The syntactic variables used are indicated in italics between ⟨ · · · ⟩'s. Thus, ⟨*mexp*⟩ for module expression, ⟨*actual*⟩ for the actuals to a parameterized module application, plus obvious compounds like ⟨*sort-list*⟩. The rules defining module expressions are

1) ⟨*mexp*⟩ ⇒ ID, INT, NAT, BOOL, REAL

2) ⟨*mexp*⟩ ⇒ TUPL [⟨*actual*⟩$_1$, · · · ,⟨*actual*⟩$_n$], for $n > 1$

3) ⟨*mexp*⟩ ⇒ ⟨*n-ary-mod*⟩ [⟨*actual*⟩$_1$, · · · , ⟨*actual*⟩$_n$)], for $n > 0$ (These actuals must satisfy additional semantic conditions.)

4) ⟨*mexp*⟩ ⇒ ⟨*mexp*⟩ * ⟨*viewbody*⟩

5) ⟨*mexp*⟩ ⇒ ⟨*viewname*⟩ (⟨*viewname*⟩ must name a view.)

6) ⟨*viewbody*⟩ ⇒ SORTS ⟨*sort-pair-list*⟩ OPS ⟨*op-pair-list*⟩

7) ⟨*mexp*⟩ ⇒ HIDE SORTS ⟨*Sort-list*⟩ OPS ⟨*Op-list*⟩ IN ⟨*mexp*⟩ | VISIBLE SORTS ⟨*Sort-list*⟩ OPS ⟨*Op-list*⟩ IN ⟨*mexp*⟩ | etc.

8) ⟨*mexp*⟩ ⇒ ⟨*mexp*⟩ + ⟨*mexp*⟩ + · · · ⟨*mexp*⟩ (There must be at least 2 ⟨*mexp*⟩'s.)

9) ⟨*sort-pair-list*⟩ ⇒ ⟨*sort-name*⟩ IS : ⟨*sortname*⟩ · · ·

10) ⟨*op-pair-list*⟩ ⇒ ⟨*opname*⟩ IS : ⟨*op1name*⟩ · · ·

Note that views are defined at the top interactive level of OBJ, rather than in module expressions.

Two remarks regarding the implementation of module expressions: since they really are expressions, they can be evaluted with a stack mechanism, just like arithmetic expressions; of course, the values are not numbers, but module bodies. The resulting action on the database can be just to add whatever the module expression says to add, without trying to generate a disjoint copy. This means that sometimes we will get ambiguous parse errors, because there are two versions of the same operator. (It would be possible to check whether this is happening as operators are being added, but the error message could not be made very informative in any case because modules created during the evaluation of ⟨*mexp*⟩'s do not have names.)

Appendix B
View Calculus

We assume familiarity with the notion of a many-sorted signature; see for example [6], [19], [23]. We also assume that any given operation op in a signature Σ has a set attr(op) of attributes, chosen from ASSOC, COMM, IDMP and ID :.

Definition 1: Let Σ and Σ' be signatures. Then a *view* ϕ: $\Sigma \to \Sigma'$ is a pair (f, g) where f: $S \to S'$ and $g_{w,s}$: $\Sigma_{w,s} \to \Sigma'_{fw,fs}$. For ϕ in $\Sigma_{w,s}$ let us write $\alpha(\phi) = w, s$, the *rank* of ϕ.

There is, of course, an extra condition on views between modules, namely preservation of all equations and constraints.

Fact 2: If $\phi = (f, g)$: $\Sigma \to \Sigma'$ and $\theta = (f', g')$: $\Sigma' \to \Sigma''$ are views, then so is their *composition*, $\phi o \theta$: $\Sigma \to \Sigma''$, defined to be $(f \circ f', g \circ g'_f)$.

We distinguish among the following kinds of views.

1) *Explicit:* declared and named at the top level.

2) *Abbreviated:* computed from a partial description, using the "default" rules.

3) *Default:* abbreviated to nothing.

4) *Internal:* defined inside the target object (there can be at most one per target module).

5) *Determined:* the view with the given source and target that will be used in the absence of an named explicit view.

Note that there may not be any determined view for a given source and target.

Definition 3: The *principal sort* of a module is either:

1) the first new sort introduced in the module (if any) or

2) the principal sort of the first parameter theory (if any) or

3) the principal sort of the first module among the modules used.

Note that these rules are ordered.

Fact 4: Each module has a unique principal sort.

From here on, we will assume that signatures come with a designated principal sort.

Let us use the following notation for a view (f, g): two sets of ordered pairs, written

SORTS ($s1$ IS: $s1'$) ($s2$ IS: $s2'$) $\cdots$

OPS (op1 IS: op1') (op2 IS: op2') $\cdots$

Definition 5: An *abbreviated view* of ϕ consists of subset of the pairs defining ϕ, obtained according to the following rules:

1) a pair (s IS: s') can be omitted if both s, s' are principal sorts;

2) any pair of the form (s IS: s) or (op IS: op) can be omitted;

3) a pair (op IS: op') can be omitted if there is a unique op''-in $\Sigma'_{f\alpha(\text{op})}$ having the same attributes as w; i.e., if $\alpha(\text{op}') = f\alpha(\text{op})$ and $\text{attr}(\text{op}) \subseteq \text{attr}(\text{op}')$.

Note that these rules are ordered, so that, for example, 1) takes priority over 2).

Proposition 6: If θ is an abbreviation of two views ϕ and ϕ', then $\phi = \phi'$. In fact, ϕ can be reconstructed from ϕ by the following rules:

1) if the principal sort p of Σ does not appear as first element of any sort pair, then add (p IS: p') where p' is the principal sort of Σ';

2) if a sort s in S does not appear as first element of any sort pair, and if s is also in S', then add (s IS: s);

3) if op is in Σ and does not appear as first element of any operation pair, and if op is in Σ', then add (op IS: op);

4) if op is in Σ and there is just one op' is $\Sigma'_{f\alpha\text{op}}$ with $\text{attr}(\text{op}) \subseteq \text{attr}(\text{op}')$, then add (op IS: op').

Proposition 7: If ϕ: $\Sigma \to \Sigma'$ and θ: $\Sigma' \to \Sigma''$ are default views, then so is $\phi \circ \theta$.

Definition 8: Given modules M and M', a view from M to M' is *determined* if it satisfies the following ordered set of rules:

1) if there is an internal view $M \Rightarrow M'$ then it is the determined view;

2) if there is a default view $M \Rightarrow M'$ then it is the determined view;

3) if there is one only explicit view $M \Rightarrow M'$ then it is the determined view.

Acknowledgments

I wish to thank Prof. R. M. Burstall for his initial encouragement, many subsequent helpful comments, and especially his collaboration in the development of Clear, which inspired the parameterization mechanism described in this paper; Dr. J. Meseguer for numerous suggestions about OBJ, for many examples in OBJ, and for help with its theoretical foundations, particularly the theory of subsorts; Prof. D. Plaisted for many suggestions about the design of OBJ, and for the implementation of OBJ1 from which the version described here is descended; Dr. J. Tardo for his pioneering implementation of OBJT; Dr. Thatcher, Dr. Wagner, and Dr. Wright for their initial help with theoretical foundations; and Dr. P. Mosses for several useful suggestions. The present OBJ2 team consists of J. Meseguer, J.-P. Jouannaud and K. Futatsugi, all of whom have contributed important ideas.

References

[1] J. Backus, "Can programming be liberated from the von Neumann style?" *Commun. ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[2] J. A. Bergstra and J. V. Tucker, "Algebraic specifications of computable and semicomputable data structures," *Theoretical Comput. Sci.*, to be published.

[3] G. Berry and P. L. Currien, "The applicative language CDS: Its denotational and operational semantics," in *Algebraic Methods in Semantics*, M. Nivat and J. Reynolds, Eds. Cambridge, England: Cambridge Univ. Press, 1984, to be published.

[4] R. M. Burstall and J. A. Goguen, "Putting theories together to make specifications," in *Proc. Fifth Int. Joint Conf. Artificial Intell.*, vol. 5, pp. 1045–1058, 1977.

[5] —, "The semantics of Clear, a specification language," *Proc. 1979 Copenhagen Winter School on Abstract Software Specification* (Lecture Notes in Computer Science, vol. 86). New York: Springer-Verlag, 1980, pp. 292–332.

[6] —, "Algebras, theories and freeness: An introduction for computer scientists," in *Proc. 1981 Marktoberdorf NATO Summer School*. Reidel, 1982.

[7] R. M. Burstall, D. MacQueen, and D. Sanella, "HOPE: An experimental applicative language," in *Conf. Rec. 1980 LISP Conf.*, Stanford Univ., Stanford, CA, 1980, pp. 136–143.

[8] A. Colmerauer, H. Kanoui, and M. van Caneghem, "Etude et réalisation d'un système prolog," Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille II, France, Tech. Rep., 1979.

[9] —, "Etude et réalisation d'un système prolog," Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille II, France, Tech. Rep.,1979.

[10] *Reference Manual for the Ada Programming Language.* U.S. Dep. Defense, ANSI/MIL-STD-1815 A.

[11] K. Futatsugi, "Hierarchical software development in HISP," in *Computer Science and Technologies 1982*, T. Kitagawa, Ed. Amsterdam, The Netherlands: North-Holland, 1982, pp. 151–174.

[12] J. Goguen, "Mathematical foundations of hierarchically organized systems," in *Global Systems Dynamics*, E. Attinger, Ed. S. Karger, 1971, pp. 112–128.

[13] J. A. Goguen, "Abstract errors for abstract data types," in *IFIP Working Conf. Formal Description Programming Concepts*, M.I.T., Cambridge, MA, 1977, P. Neuhold, Ed. New York: North-Holland, 1979.

[14] —, "Order sorted algebra," Dep. Comput. Sci., Univ. California, Los Angeles, Semantics and Theory of Computation Rep. 14; also, *J. Comput. Syst. Sci.*, to be published.

[15] —, "Algebraic specification," in *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, MA: M.I.T. Press, 1979, pp. 370–376.

[16] —, "How to prove algebraic inductive hypotheses without induction: With applications to the correctness of data type representations," in *Proc. 5th Conf. Automated Deduction* (Lecture Notes in Computer Science, vol. 87), W. Bibel and R. Kowalski, Eds. New York: Springer-Verlag, 1980.

[17] —, "Merged views, closed worlds and ordered sorts: Some novel database features in OBJ," SRI International, Menlo Park, CA, Tech. Rep., 1982; also to appear in *SIGMOD*.

[18] —, "Suggestions for using and organizing libraries for Ada program development," Tech. Rep. prepared for Ada Joint Program Office, SRI International, Menlo Park, CA, 1984.

[19] J. A. Goguen and R. M. Burstall, "Introducing institutions," in *Proc. Logics of Programming Workshop* (Lecture Notes in Computer Science, vol. 164), E. Clarke and D. Kozen, Eds. New York: Springer-Verlag, 1984, pp. 221–256.

[20] J. A. Goguen and S. Ginali, "A categorical approach to general systems theory," in *Applied General Systems Research*, G. Klir, Ed. New York: Plenum, 1978, pp. 257–270.

[21] J. Goguen and J. Meseguer, "Rapid prototyping in the OBJ executable specification language," *Software Engineering Notes*, vol. 7, no. 3, pp. 75–84, 1982; also in *Proc. Rapid Prototyping Workshop*.

[22] —, "Equality, types and generics for logic programming," Center for the Study of Logic and Information, Stanford Univ., Stanford, CA, Tech. Rep. CSLI-84-5, Mar. 1984; see also *Proc. 1984 Logic Programming Symp.*, Upsala, Sweden.

[23] —, *An Initiality Primer.* Book in preparation.

[24] J. A. Goguen and K. Parsaye-Ghomi, "Algebraic denotational

semantics using parameterized abstract modules," in *Formalizing Programming Concepts* (Lecture Notes in Computer Science, vol. 107), J. Diaz and I. Ramos, Eds. Peniscola, Spain: Springer-Verlag, 1981, pp. 292–309.

[25] J. A. Goguen and J. Tardo, "An introduction to OBJ: A language for writing and testing software specifications," in *Specification of Reliable Software.* New York: IEEE Press, 1979, pp. 170–189.

[26] J. A. Goguen, J. Meseguer, and D. Plaisted, "Programming with parameterized abtract objects in OBJ," in *Theory and Practice of Software Technology*, D. Ferrari, M. Bolognani, and J. Goguen, Eds. Amsterdam, The Netherlands: North-Holland, 1982, pp. 163–193.

[27] J. A. Goguen, J. W. Thatcher, and E. Wagner, "An initial algebra approach to the specification, correctness and implementation of abstract data types," in *Current Trends in Programming Methodology*, R. Yeh, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1978, pp. 80–149; see also original version, IBM T. J. Watson Research Center, Yorktown Heights, NY, Tech. Rep. RC 6487, Oct. 1976.

[28] J. A. Goguen, J. W. Thatcher, E. Wagner, and J. B. Wright, "Abstract data types as initial algebras and the correctness of data representations," in *Computer Graphics, Pattern Recognition and Data Structure.* New York: IEEE Press 1975, pp. 89–93.

[29] M. J. Gordon, R. Milner, and C. P. Wadsworth, *Edinburgh LCF* (Lecture Notes in Computer Science, vol. 78). New York: Springer-Verlag, 1979.

[30] J. V. Guttag, "The specification and application to programming of abstract data types," Ph.D. dissertation, Dep. Comput. Sci., Univ. Toronto, Toronto, Ont., Canada, Rep. CSRG-59, 1975.

[31] J. V. Guttag, E. Horowitz, and D. R. Musser, "Abstract data types and software validation," *Commun. ACM*, 1978.

[32] C. M. Hoffman and M. J. O'Donnell, "Programming with equations," *ACM Trans. Programming Languages Syst.*, vol. 1, no. 4, pp. 83–112, 1982.

[33] J. Hsiang, "Refutational theorem proving using term rewriting systems," Ph.D. dissertation, Univ. Illinois, Urbana.

[34] G. Huet, "Confluent reductions: Abstract properties and applications to term rewriting systems," *J. Ass. Comput. Mach.*, vol. 27, pp. 797–821, 1980; see also preliminary version in *18th IEEE Symp. Foundations Comput. Sci.*, 1977.

[35] G. Huet and D. Oppen, "Equations and rewrite rules: A survey," in *Formal Language Theory: Perspectives and Open Problems*, R. Book, Ed. New York: Academic, 1980.

[36] M. A. Jackson, *Principles of Program Design.* New York: Academic, 1975.

[37] J. P. Jouannaud, "Confluent and coherent equational term rewriting systems," in *Proc. 5th CAAP* (Lecture Notes in Computer Science). New York: Springer-Verlag, 1983.

[38] G. Levy and F. Sirovich, "TEL: A proof-theoretic language for efficient symbolic expression manipulation," IEI, Tech. Rep., Nota Interna B77-3, Feb. 1977.

[39] P. Lucas and T. Risch, "Representation of factual information by equations and their evaluation," IBM Research, Yorktown Heights, NY, Tech. Rep., 1982.

[40] J. McCarthy, M. Levin, *et al.*, *LISP 1.5 Programmer's Manual.* Cambridge, MA: M.I.T. Press, 1966.

[41] R. Milner, "A theory of type polymorphism in programming," *J. Comput. Syst. Sci.*, vol. 17, no. 3, pp. 348–375, 1978.

[42] M. J. O'Donnell, *Computing in Systems Described by Equations* (Lecture Notes in Computer Scicnce). New York: Springer-Verlag, 1977.

[43] D. L. Parnas, "Information distribution aspects of design methodology," *Inform. Processing*, vol. 71, 1972.

[44] —, "A technique for software module specification," *Commun. ACM*, vol. 15, 1972.

[45] D. Plaisted, "An initial algebra sematics for error presentations," SRI International, Menlo Park, CA, 1982.

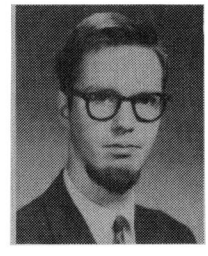
[46] J. Reynolds, "Using category theory to design implicit conversions and generic operators," in *Semantics Directed Compiler Generation* (Lecture Notes in Computer Science, vol. 94). New York: Springer-Verlag, 1980, pp. 211–258.

[47] J. Tardo, "The design specification and implementation of OBJT: A language for writing and testing abstract algebraic program specifications," Ph.D. dissertation, Dep. Comput. Sci., Univ. California, Los Angeles, 1981.

[48] M. H. van Emden and R. A. Kowalski, "The semantics of predicate logic as a programming language," *J. Ass. Comput. Mach.*, vol. 23, no. 4, pp. 733–742, 1976.

[49] M. Wand, "Algebraic theories and tree rewriting systems," Dep. Comput. Sci., Indiana Univ., Bloomington, Tech. Rep. 66, 1977.

[50] S. Zilles, "Abstract specification of data types." Computation Structures Group, Massachusetts Inst. Technol., Cambridge, Tech. Rep. 119, 1974.



Joseph A. Goguen received the B.A. degree from Harvard University, Cambridge, MA, and the Ph.D. degree from the University of California at Berkeley.

He has taught at Berkeley, Chicago, Edinburgh, and U.C.L.A., where he was a full Professor of Computer Science until 1981. He is currently a Senior Computer Scientist at SRI International, Menlo Park, CA; he is also associated with the Center for the Study of Language and Information at Stanford University, Stanford, CA, and is Managing Director of Structural Semantics. His research interests include software engineering and programming methodology, programming language and environment design, logic programming, abstract data types, specification and design languages, Ada library systems, semantics of natural and artificial languages, discourse analysis, and semiotics.