Higher Order Functions Considered Unnecessary for Higher Order Programming

Joseph A. Goguen
Programming Research Group, Oxford University

Abstract

It is often claimed that the essence of functional programming is the use of functions as values, i.e., of higher order functions, and many interesting examples have been given showing the power of this approach. Unfortunately, the logic of higher order functions is difficult, and in particular, higher order unification is undecidable. Moreover (and closely related), higher order expressions are notoriously difficult for humans to read and write correctly. However, this paper shows that typical higher order programming examples can be captured with just first order functions, by the systematic use of parameterized modules, in a style that we call parameterized programming. This has the advantages that correctness proofs can be done entirely within first order logic, and that interpreters and compilers can be simpler and more efficient. Moreover, it is natural to impose semantic requirements on modules, and hence on functions. A more subtle point is that higher order logic does not always mix well with subsorts, which can nonetheless be very useful in functional programming by supporting the clean and rigorous treatment of partially defined functions, exceptions, overloading, multiple representation, and coercion. Although higher order logic cannot always be avoided in specification and verification, it should be avoided wherever possible, for the same reasons as in programming. This paper contains several examples, including one in hardware verification. An appendix shows how to extend standard equational logic with quantification over functions, and justifies a perhaps surprising technique for proving such equations using only ground term reduction.

1 Introduction

Following an introduction to the OBJ language, this paper gives some examples showing how higher order functions can be avoided by using sufficiently powerful parameterized modules. I do not consider higher order functions harmful, useless, or unbeautiful; but I do claim significant advantages for avoiding higher order functions whenever possible, and I claim that they can be avoided quite systematically in functional programming, by using parameterized programming instead. Of course, higher order logic is useful in many areas, particularly the foundations of mathematics (e.g., type theory), extracting programs from proofs, describing proof strategies (e.g., LCF tactics), and the semantics of traditional programming languages (e.g., Scott-Strachey); but it too should be avoided whenever possible, and the appendix develops some techniques for reasoning about first order functions that often make it possible to do so.

1.1 Parameterized Programming

A major advantage of functional programming over traditional imperative programming is that it can yield better structured programs [57]. However, a language with sufficiently powerful parameterized modules can achieve highly structured programs without higher order functions. In particular, the examples given below show that typical higher order functional programming techniques are easily carried out with OBJ's parameterized programming, in ways that seem to me even more structured and flexible. Code is broken into highly parameterized, mind-sized, internally coherent

*This paper was written in 1987 while the author was at the Computer Science Laboratory at SRI International.
modules, and then new programs are constructed from old ones by instantiating, transforming and combining these modules. This paper argues that first order parameterized programming includes the essential power of higher order programming, and even offers certain advantages.

Parameterized programming is a general and powerful technique for software design, production, reuse and maintenance. This approach involves abstraction through two kinds of module: objects to encapsulate executable code, and in particular to define abstract data types; and theories to specify both syntactic structure and semantic properties of modules. Each kind of module can be parameterized, where actual parameters are modules, and can also import other modules. Interfaces of parameterized modules are defined by theories, and thus include semantic as well as syntactic constraints. For parameter instantiation, a view binds the formal entities in an interface theory to actual entities in a module, and also asserts satisfaction of the theory by the module. Views are first class citizens that can be named, can import modules, and can even be parameterized. This integration of objects, theories and views provides a powerful wide-spectrum capability. A software design is represented as a hierarchy of modules with associated views, and a software system is actually constructed from its components when the design (i.e., the code) is executed. In particular, module expressions allow complex instantiations of generics, and include commands that transform already defined modules; they can be seen as generalizing the UNIX make command. Maintenance is facilitated by editing and then re-executing such designs. Reusability is enhanced by the flexibility of the parameterization, composition and transformation mechanisms. Default views can greatly reduce the effort of defining views.

All these ideas are illustrated in OBJ, a wide-spectrum, first order functional programming language that is rigorously based upon order sorted (conditional) equational logic. This logic provides a notion of subtype that supports many useful features, including multiple representation, overloading, coercion, multiple inheritance, and exception handling. This rigorous semantic basis allows a declarative, specificational style of programming, eases system design and implementation, and facilitates program verification. Moreover, logical specifications can be directly executed. These points are illustrated in several examples, including a simple hardware verification example.

1.2 Some History

OBJ was originally designed in 1976 by Joseph Goguen as a language for "error algebras" [29], an attempt to extend algebraic abstract data types to handle errors and partial functions in a simple, uniform way. This first design also used ideas from Clear [7, 9] for parameterized modules. Initial implementations of OBJ were done from 1977 to 1979 at UCLA by Joseph Tardo as OBJO and OBJT [85, 46] using error algebras plus an "image" construct for parameterization. David Plaisted implemented OBJI by enhancing OBJT, during 1982-83 at SRI; improvements included (an efficient form of) matching modulo associativity and/or commutativity, hash coded memo functions, and a highly interactive environment [44]. OBJ2 [21, 22] was implemented during 1984-85 at SRI by Kokichi Futatsugi and Jean-Pierre Jouannaud, following a design led by Joseph Goguen and José Meseguer, based on order sorted algebra [43, 40, 36] rather than on error algebra; and OBJ2 provided Clear-like parameterized modules, theories and views, although not in full generality.

The latest version, OBJ3, is available from SRI International, and was developed (using Kyoto Common Lisp) by Joseph Goguen, José Meseguer, Timothy Winkler, Claude and Hélène Kirchner, and Aristide Megrelis; the implementation team was led by José Meseguer. Although OBJ3 has a syntax quite close to that of OBJ2, it has a different implementation based on a simpler and more efficient operational semantics for order sorted algebra [59]. Also, it provides much more

---

1 The Ada [75] notion of a generic package provides only part of what is needed. In particular, Ada generic packages provide no way to document the semantics of interfaces, although this feature can greatly improve the reliability of software reuse and can also help retrieve the right module from a library (as discussed in [31]). Also, Ada provides only very weak facilities for combining modules; for example, only one level of module instantiation is possible at a time.
sophisticated module expressions, including default views, for which Timothy Winkler deserves special credit. OBJ can be seen as an implementation of Clear for conditional order sorted logic.

Other implementations of OBJ include UMIST-OBJ from the University of Manchester Institute of Science and Technology [16], Abstract Pascal from the University of Manchester [60], MC-OBJ from the University of Milan [14], and a Franz Lisp OBJ2 from Washington State University [83]. The first two are written in Pascal and the third in C. UMIST-OBJ is available in Britain as a commercial product, called Obj-Ex, and another variant, called Axis [20], is available from Hewlett-Packard Labs in Bristol, UK. In addition, we are extending OBJ in the directions of relational and object-oriented programming, to languages called Eqlog [39] and FOOPS [41], respectively.

The experimental OBJ systems implemented so far have been used for many applications, including debugging algebraic specifications [44], rapid prototyping [37], defining programming languages in a way that immediately yields an interpreter (see [45] and the elegant work of Peter Mosses [71, 72]), specifying software systems (e.g., the GKS graphics kernel system [18], an Ada configuration manager [23], the Macintosh QuickDraw program [73], and OBJ in itself [16]), and hardware specification, simulation and verification (see [32, 84] and Section 3.2). Many of these applications were produced under an experiment sponsored by the British Alvey Project, and will be collected with some more recent work in a book on the practical use of OBJ [35]. OBJ is also being combined with Petri nets, thus allowing structured data in tokens [2], and is one language for programming a massively parallel machine that executes rewrite rules directly [61, 42]; in fact, we believe that OBJ on such a machine should greatly out-perform a conventional language on a conventional machine, by direct concurrent execution of rewrite rules; however, FOOPS offers some further advantages.

2 Aspects of OBJ

This section is a rather lengthy, but still incomplete and informal, introduction to OBJ. Readers already familiar with OBJ should skip directly to Section 3 and Appendix A. Readers who are already familiar with some other functional programming language should at least skim this section, because OBJ embodies basic design choices that are quite different from those of other programming languages, including other current functional programming languages:

1. It is rigorously based on deduction in order sorted equational logic, which provides a precise semantics for exception handling, multiple inheritance, overloading, and multiple representations for data abstractions. It uses strong sorting with retracts to ease parsing.

2. It supports parameterized programming, as sketched above and expanded below.

3. It supports user-defined evaluation strategies for each operation separately, rather than imposing a global order of evaluation; this allows both eager and lazy evaluation, as well as more complex options; also, efficient default evaluation strategies are computed by simple strictness analysis if the user does not provide an explicit strategy.

4. It has rewriting modulo attributes, including associative, commutative, identity and idempotent.

OBJ is a logical programming language in the sense that it is based on inference in a precise logical system, namely conditional order sorted equational logic; see [68] for more on logical programming, and [43] for more on order sorted equational logic. As has been well argued by advocates of Prolog, this confers certain important benefits: program simplicity and clarity (which can greatly ease program understanding, debugging and maintenance); separation of logic and control; and identity of program logic with proof logic. In such a language, a high level description of what a program does actually is a program; that is, one can execute it. Other logical programming languages include pure Prolog [17], pure Lisp [67], and CDS [4]; such languages can also be considered
as efficiently executable specification languages. Some other languages that are based on algebraic semantics include Larch [51], Aspegique [5], Obscure [62] and Act One [19]; it seems fair to say that they have all been significantly influenced by OBJ.

Higher order functional programming languages like Hope [12], Miranda [86], ML [53] and Haskell [56] can be seen as either based on rewrite rules, or else on higher order equational logic, although they tend to have some impure features for efficiency or convenience; for example, ML has assignment and exceptions, while Miranda has ad hoc coercions among various kinds of numbers, as well as lazy pattern matching. Standard ML has a powerful parameterized module facility inspired in part by Clear’s. Guttag, Horowitz and Musser [52] describe a system for the symbolic execution of algebraic abstract data types, and Levy and Sirovich [63] describe the TEL system for specifying semantics with equations. Other related systems include those due to Hoffmann and O’Donnell [55, 74], Lucas and Risch [64], and Prywes [78], all of which are first order, and the elegant work of Backus [1], which is higher order functional programming for a fixed set of rewrite rules and data types.

This section provides an intuitive introduction to features of OBJ that are needed for understanding our higher order programming examples. Some important topics are thus omitted, including user-definable evaluation strategies, details of OBJ semantics, and default views.

2.1 Strong Sorting

To avoid the confusion associated with the many different uses of the word “type,” we shall instead use the word “sort” from here on in connection with the many-sorted equational logic based approach of OBJ. Among the advantages of strong sorting are: to catch meaningless expressions before they are executed; to separate logically and intuitively distinct concepts; and to enhance readability by documenting these distinctions. With a modern (e.g., structural) editor, it is little trouble to insert sort declarations; many could even be inserted automatically by a compiler or a smart editor.

Ordinary unsorted logic offers the dubious advantage that anything can be applied to anything; for example,

\[
\text{first-name(not(age(3 * false))) iff birth-place(temperature(329))}
\]

is permissible. Although beloved by Lisp and Prolog hackers, unsorted logic is too permissive. However, many sorted logic is too restrictive, since it does not support overloaded function symbols, such as _+_ for integer, rational, and complex numbers (where the underbar character _ serves as a placeholder for arguments). Moreover, strictly speaking, an expression like (-4 / -2)! does not parse in many sorted logic (assuming that factorial only applies to natural numbers), since (-4 / -2)! parses as a rational rather than a natural. This problem can be solved by extending order sorted algebra with retracts, which provide sufficient expressiveness while still banishing truly meaningless expressions, as discussed in Section 2.3 below.

2.2 Operation and Expression Syntax

It seems worth some extra implementation effort and processing time to support syntax that is as flexible, informative, and close to users’ intuitions and to standard usage as possible. OBJ users can define any syntax they like for operations, including prefix, postfix, infix, and most generally, mixfix, to customize it for any given problem domain; this is similar to ECL [15]. 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 (or more precisely, a unique parse of least sort; see Section 2.3). The argument and value sorts of an operation are declared at the same time as its syntactic form. We distinguish two cases. The first is the usual parenthesized-prefix-with-commas functional form. For example,

\[
\text{op f : S1 S2 -> S3 .}
\]
indicates that \( f(X, Y) \) has sort \( S_3 \) when \( X \) has sort \( S_1 \) and \( Y \) has sort \( S_2 \). The general mixfix case uses place-holders, indicated by an underbar character, as in the prefix declaration

\[
\text{op top_} : \text{Stack} \rightarrow \text{Int}.
\]

for \( \text{top} \) as used in expressions like \( \text{top push}(A, B) \). Similarly, the “outfix” form of the singleton set operation, as in 4, is declared by

\[
\text{op \{\}_} : \text{Int} \rightarrow \text{Set}.
\]

and the infix form for addition, as in \( 2 + 3 \), is

\[
\text{op \_+\_} : \text{Int} \text{ Int} \rightarrow \text{Int}.
\]

while a mixfix declaration for conditional is

\[
\text{op if\_then\_else\_fi} : \text{Bool} \text{ Int} \text{ Int} \rightarrow \text{Int}.
\]

Between the : and the \( \rightarrow \) in an operation declaration comes the \textbf{arity} of the operation, and after the \( \rightarrow \) comes its \textbf{value sort} (also called “co-arity”); the (arity,value sort) pair is called the \textbf{rank} of an operation.

Operations with the same arity and value sort but with different forms can be declared together, for example

\[
\begin{align*}
\text{ops zero one :} & \rightarrow \text{S} \\
\text{ops \_+\_\_\_\_} : & \text{S S} \rightarrow \text{S}.
\end{align*}
\]

Parentheses are required in the second case, to mark the boundary between the two forms.

The following simple object for bit strings illustrates some basic OBJ syntax:

\[
\text{obj BITS is sorts Bit Bits}.
\]

\[
\begin{align*}
\text{ops 0 1 :} & \rightarrow \text{Bit} \\
\text{op nil :} & \rightarrow \text{Bits} \\
\text{op \_\_\_} : & \text{Bit Bits} \rightarrow \text{Bits}.
\end{align*}
\]

\text{endo}

A typical expression using the syntax of this object is \( 0 \ . \ 1 \ . \ 0 \ . \ \text{nil} \).

\subsection{2.3 Subsorts}

To handle cases where things of one sort are also of another sort, e.g., all natural numbers are also rational numbers, and cases where expressions may have several different sorts, we use \textbf{order sorted algebra}. This approach involves imposing a partial ordering on the set of sorts, e.g., \( \text{Nat} < \text{Rat} \), meaning \( \text{Nat} \leq \text{Rat} \) (we use \( < \) instead of \( \leq \) for typographical convenience). Then \textbf{multiple inheritance} is supported, since a given sort can have more than one distinct supersort, and operation overloading arises by restricting functions to subsorts. The \textbf{signature} of an object consists of the sorts, subsort relation, and operations defined in it, including their form, arity, and value sort.

Two happy facts are that order sorted algebra is only slightly more difficult than many sorted algebra, and that essentially all results generalize from the many sorted to the order sorted case without complication. Although this paper omits all technical details, order sorted algebra is a rigorous mathematical theory. Order sorted algebra originated in 1978, and is treated comprehensively in [43] and summarized in [40]. Some alternative approaches have been nicely developed by Gogolla [24, 25], Wadge [87], Reynolds [79], and others.
OBJ directly supports *subsort polymorphism*, which is operator overloading consistent under subsort restriction. By contrast, languages like ML [53] and Hope [12] support *parametric polymorphism* [70], following ideas of Strachey. OBJ's parameterized modules provide a similar capability in a different way.

A term over an order sorted signature is considered *well-formed* if it has a unique parse of lowest sort; [43] and [40] show that this occurs under certain mild and natural assumptions. Sometimes subexpressions are not of the expected sort, and must be "coerced" to it. This is trivial from a subsort to a supersort; for example, if the operation + is only defined for rationals, then \(2 + 2\) is fine even though 2 is a natural number, because \(\text{Nat} < \text{Rat}\). It is less trivial the other way; for example, consider \((-4 / -2)\) which is only defined for natural numbers. At parse time, we cannot know whether the subexpression \((-4 / -2)\) will turn out to be a natural number, so the parser must consider it a rational; in fact, the expression \((-4 / -2)\) does not parse in the conventional sense. However, we can "give it the benefit of the doubt" by having the parser insert a *retract*, a special operation symbol (denoted \(\text{r:rat} \to \text{nat}\) in the example below) that lowers the sort, and is removed at run time if the subexpression really is a natural, but otherwise remains behind as an informative error message. Thus, the parser turns the expression \((-4 / -2)\) into the expression \((\text{r:rat} \to \text{nat}(-4 / -2))\) which at runtime becomes first \((\text{r:rat} \to \text{nat}(2))\) ! and then \((2)\) !, using the (automatically provided) key equation

\[
\text{r:rat} \to \text{nat}(X) = X
\]

where \(X\) is a variable of sort \(\text{Nat}\). [36] describes the mathematical and operational semantics of retracts.

Exceptions have both inadequate semantic foundations and insufficient flexibility in most programming and specification languages. Algebraic specification languages sometimes use partial functions, which are simply undefined under exceptional conditions. Although this approach can be developed rigorously, as in [58], it is unsatisfactory in practice because it does not allow error messages or error recovery. For some time, we have been exploring rigorous approaches that allow users to define their own exception conditions, messages, and handling. Unfortunately, the original OBJ/OBJJ error algebra approach [29] sometimes lacks initial models [76], but our current order sorted algebra approach seems entirely satisfactory. Using subsorts, we can give a somewhat better representation for bit strings than that in the previous subsection:

```obj
obj BITS1 is sorts Bit Bits .
   subset Bit < Bits .
   ops 0 1 : \to Bit .
   op _ _ : Bit Bits \to Bits .
endo
```

A typical expression using this syntax is \(0 \ 1 \ 0\).

### 2.4 Semantics

OBJ has both an abstract denotational semantics based on order sorted algebra, and a more concrete operational semantics based on order sorted rewriting.

#### 2.4.1 Operational Semantics

Equations are written declaratively and interpreted operationally as rewrite rules, which replace substitution instances of lefthand sides by the corresponding substitution instances of righthand sides. We can illustrate computation by term rewriting with a simple \textsc{List-Of-Int} object. (The protecting \textsc{Int} line below indicates that the \textsc{Int} module, which provides the integers, is imported; module importation is discussed in Section 2.5 below.)
obj LIST-0F-INT is sort List .
  protecting INT .
subsorts Int < List .
op _ _ : Int List -> List .
op length_ : List -> Int .
var I : Int .
var L : List .
eq length I = 1 .
eq length(I L) = 1 + length L .
endo

A reduce command is executed until reaching a term to which no further rules can be applied, called a normal (or reduced) form. (Most functional programming languages require users to declare constructors such that a term is reduced if and only if it consists entirely of constructors. OBJ does not make any use of such constructors, thus achieving greater generality; however, constructor declarations could certainly be used to aid with compiler optimization.) For example,

reduce length(17 -4 329).

when evaluated in LIST-0F-INT gives

result Int: 3

by the following sequence of rewrite rule applications

length 17 -4 329 =>
  1 + length -4 329 =>
  1 + (1 + length 329) =>
  1 + (1 + 1) =>
  1 + 3 =>
  3

where the first step uses the second equation, which has the lefthand side length I L matching I to 17 and L to -4 329. The second step also uses this equation, but now matching I to -4 and L to 329; this match works by regarding the integer 329 as a List, since Int is a subsort of List. The third step simply uses the first rule, and the last step uses the built-in arithmetic\(^2\) of INT.

Let us now consider a more sophisticated integer list object with associative and identity attributes:

obj LIST-0F-INT1 is sorts List NeList .
  protecting INT .
subsorts Int < NeList < List .
op nil : -> List .
op _ _ : NeList List -> NeList [assoc].
op head_ : NeList -> Int .
op tail_ : NeList -> List .
var I : Int .
var L : List .
eq head(I L) = I .
eq tail(I L) = L .
endo

\(^2\)OBJ optionally allows users to define functions with Lisp code; this has been used to provide efficient implementations for the various kinds of numbers.
Then

```
reduce 0 nil 1 nil 3 .
```

is executed by applying the identity axiom modulo associativity, as follows

```
0 nil 1 nil 3 =>
0 1 nil 3 =>
0 1 3
```

Similarly, we have

```
reduce head(0 1 3) .
*** result Int: 0
```

```
reduce tail(0 1 3) .
*** result NilList: 1 3
```

```
reduce tail(nil 0 1 nil 3) .
*** result NilList: 1 3
```

where results are shown as comments, after ***. One can also explicitly name a module to be used as the context for evaluation, as in

```
reduce in BOOL : true and false .
```

The identity attribute is implemented by adding rules, rather than by pattern matching modulo identity. A subtle point is that sometimes extra rules are needed. For example, the special case `head 1 = 1` of the first equation in `LIST=OF=INT1` with `L = nil` must be added to the OBJ rulebase. Also, sometimes it is necessary to generate so-called “associative extension” rules [59].

OBJ has a built-in polymorphic binary infix `Bool`-valued operation `==` on every sort, to tell whether or not two ground expressions are equal. This is computed by checking syntactic identity of the normal forms of two expressions. For example, `==` on `Bool` is just `_iff_`. The operation `==` really is equality on a sort provided that the rules for expressions of that sort are Church-Rosser and terminating with respect to the given evaluation strategy, since these conditions guarantee that normal forms will be reached. The negation `_=/=` of `==` is also available. Finally, the conditional

```
if_then_else.fi : Bool S S => S
```

is provided for every defined sort `S`.

OBJ also allows conditional equations, with syntax

```
ceq (exp1) = (exp2) if (exp3) .
```

where `(exp3)` is `Bool`-valued, meaning operationally that the rewrite is applied only if the condition evaluates to `true`.

### 2.4.2 Denotational Semantics

Whereas the operational semantics of a programming language shows how computations are done, its denotational semantics should give precise meanings to programs in a conceptually clear and simple way that supports proving properties about them. The denotational semantics of OBJ is algebraic, as in the algebraic approach to abstract data types [48, 37, 89, 50]; that is, the denotation of an object is an algebra, a collection of sets with functions among them. In a logical programming language like OBJ, the already established proof theory of the underlying logical system applies directly to programs, and complex formalisms like Scott-Strachey denotational semantics and Hoare axiomatic semantics are not needed. **Initial algebra semantics** [28, 37, 69] takes the unique (up
to isomorphism) “initial” algebra as the “most representative” model of the equations (there may of course be many other models), i.e., as the representation-independent standard of comparison for correctness. [10] shows that an algebra is \textbf{initial} if and only if it satisfies these two properties:

1. \textbf{no junk}: every element can be named using the given constant and operation symbols; and

2. \textbf{no confusion}: all equations true of the algebra can be proved from the given equations.

If the rule set is Church-Rosser and terminating, then the rewrite rule operational semantics agrees with initial algebra semantics (see [30, 88]). Order sorted algebra, and thus OBJ, provides a completely general programming formalism, in the sense that any partial computable function can be defined, according to an as yet unpublished theorem of José Meseguer; [3, 69] give similar results for total computable functions.

2.5 \textbf{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 many modules, it is helpful to keep explicit track of the hierarchical structure of module dependence, showing exactly which modules use which others. The collection of other modules used by a given module, together with the dependence relations among them, constitute the immediate context of the given module. Whenever a module uses sorts or operations declared in another module, that other module must be explicitly imported and also must have been defined earlier in the program. A program developed in this way has the abstract structure of a hierarchy, or more precisely, an \textbf{acyclic graph}, of abstract modules.\footnote{Such a hierarchy differs from a Dijkstra-Parnas hierarchy of abstract machines because higher level modules are not \textit{implemented} by lower level (less abstract) machines; rather, higher level modules \textit{include} lower level modules.} More exactly, a directed edge in an acyclic graph of modules indicates that the higher (target) module \textbf{imports} the lower (source) module, and the \textbf{context} of a given module is the subgraph of other modules upon which it depends, i.e., the subgraph of which it is the top. Parameterized modules can also occur in such a hierarchy, and are treated in essentially the same way as unparameterized modules. (This discussion is a bit oversimplified, since OBJ environments must reflect not only submodule relations, but also the more general view relations that may hold among modules.)

OBJ has three modes for importing modules, called \textbf{using}, \textbf{extending} and \textbf{protecting}. By convention, if a module $M$ imports a module $M'$ that imports a module $M''$, then $M''$ is also imported into $M$; that is, “importing” is a \textit{transitive} relation. The meaning of these three import modes is related to initial algebra semantics, in that an importation of module $M'$ by $M$ is:

1. \textbf{protecting} if $M$ adds no new data items of sorts from $M'$, and also identifies no old data items of sorts from $M'$ (no junk and no confusion);

2. \textbf{extending} if $M$ identifies no old data items of sorts from $M'$ (no confusion); and

3. \textbf{using} if there are no guarantees at all.

\textbf{using} is implemented by copying the imported module’s text, without copying the modules that it imports; if desired, these can also be copied, by listing them after the \textbf{using} keyword.

2.6 \textbf{Parameterization}

The basic building blocks of parameterized programming are theories, views and module expressions, each of which can be parameterized; the resulting capabilities go well beyond (for example) those of Ada generic modules. As described above, an \textbf{object} encapsulates extensible code. On the other
hand, a **theory** defines the interface of a parameterized module, that is, the structure and properties required of an actual parameter for meaningful instantiation. A **view** expresses that a certain module satisfies a certain theory in a certain way (note that a module can satisfy a given theory in more than one way); that is, a view describes a binding of an actual parameter to a requirement theory. **Instantiation** of a parameterized module to an actual parameter, using a particular view, yields a new module. **Module expressions** describe complex interconnections of modules, possibly adding, renaming, or modifying functionality. All these topics are treated in greater detail below.

### 2.6.1 Theories

Theories express semantic properties of modules and module interfaces. This and the next subsection discuss requirement theories and views, respectively. In general, OBJ theories have the same structure as objects; in particular, theories have sorts, subsorts, operations, variables and equations, can import other theories and objects, can be parameterized, and can have views. The difference is that objects are executable, while theories just define properties. Semantically, a theory has a **variety** of models, all the (order sorted) algebras that satisfy it, whereas an object has just one model (up to isomorphism), its initial algebra.

Now some example theories. The first example is the trivial theory TRIV, which requires nothing except a sort, here designated Elt.

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

The next theory is an extension of TRIV, requiring that models also have a given element of the given sort, here designated *.

```plaintext
th TRIV* is extending TRIV .
  op * : -> Elt .
endth
```

Of course, this enrichment is equivalent to

```plaintext
th TRIV* is sort Elt .
  op * : -> Elt .
endth
```

which may seem clearer.

Next, the theory of pre-ordered sets (which are like partially ordered sets but without the anti-symmetric law). Its models have a binary infix **Bool-valued operation <=** that is reflexive and transitive.

```plaintext
th PREORD is sort Elt .
  op _<=_ : Elt Elt -> Bool .
  vars E1 E2 E3 : Elt .
  eq E1 <= E1 = true .
  ceq E1 <= E3 = true 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.

```plaintext
th EQV is sort Elt .
  op _eq_ : Elt Elt -> Bool .
  vars E1 E2 E3 : Elt .
  eq (E1 eq E1) = true .
  eq (E1 eq E2) = (E2 eq E1) .
  ceq (E1 eq E3) = true if (E1 eq E2) and (E2 eq E3) .
endth
```
Finally, the theory of monoids, which will later serve as a parameter requirement theory for a general iterator that in particular gives sums and products over lists.

```plaintext
th MONOID is sort M .
  op e : -> M .
  op _.*_ : M M -> M [assoc id: e] .
endth
```

The possibility of expressing *semantic* properties, such as the associativity of an operation, as part of the interface of a module is another aspect where parameterized programming has an advantage over traditional functional programming. For example, one can certainly write a (second order) function to iterate any given binary function (such as integer addition) over lists, but traditional functional programming cannot state the requirement that the binary function must be associative.

### 2.6.2 Views

A module can satisfy a theory in more than one way, and even if there is a unique way, it can be arbitrarily difficult to find. We therefore need a notation for describing the particular ways that modules satisfy theories. For example, nat can satisfy PREORD with the usual “less-than-or-equal” ordering, but “divides” (which is div in OBJ) and “greater-than-or-equal” are also possible; each of these corresponds to a different view. Thus, an expression like SORTING[NAT], where SORTING has requirement theory PREORD would be ambiguous in the absence of definite conventions for default views.

More precisely now, a view v from a theory T to a module M, indicated with the notation v: T => M, consists of a mapping from the sorts of T to the sorts of M preserving the subset relation, and a mapping from the operations of T to the operations of M preserving arity, value sort, and the (meaning of) whatever attributes assoc, comm, id: and idem are present, such that every equation in T is true of every model of M. (A view from one theory to another is what logicians call a theory interpretation [8].) The mappings of sorts and operations are expressed in the respective forms

```plaintext
  sort S1 to S1’
  sort S2 to S2’
  ...
  op o1 to o1’
  op o2 to o2’
  ...
```

where o1, o1’, o2, etc. may be operation forms, or forms plus value sort, or forms plus value sort and arity, as needed for disambiguation; moreover, o1’, o2’ etc. can be derived operations (i.e., terms with variables). Thus, each mapping can be considered a set of pairs. These two sets of pairs together are called a **view body**. The syntax for defining a view at the top level of OBJ adds to this names for the source and target modules, and possibly a name for the view. For example,

```plaintext
  view NATD from PREORD to NAT is
    sort Elt to Nat .
    op _<=_ to _div_ .
endv
```

defines a view called NATD from PREORD to NAT using the divisibility relation.

When there is an obvious view to use, it is annoying to have to write out that view in full detail. **Default views** allow writing simple module expressions like P[NAT, INT] wherever possible, by capturing the intuitive notion of “the obvious view;” see [33] for details.

11
2.6.3 Parameterized Modules

Let us now consider some parameterized modules. First, a simple parameterized LIST object, "abstracting" the previously given LIST-OF-INT object.

```plaintext
obj LIST[X :: TRIV] is sorts List NeList .
  subsorts Elt < NeList < List .
  op nil : -> List .
  op head_ : NeList -> Elt .
  op tail_ : NeList -> List .
  var X : Elt .
  var L : List .
  eq head X L = X .
  eq tail X L = L .
end
```

Modules can have more than one parameter. For example, the notation [X :: TH1, Y :: TH2] indicates two parameters, and if the two theories are the same, we can just write [X Y :: TH]. Parameterized theories are also allowed, such as vector spaces over a field F.

Even though the code is very similar to that for LIST, it seems worth doing STACK as well, since it is well-known and has been done in many different formalisms; in fact, it provides a very good illustration of the power of order sorted algebra.

```plaintext
obj STACK[X :: TRIV] is sorts Stack NeStack .
  subsorts Elt < NeStack < Stack .
  op empty : -> Stack .
  op push : Elt Stack -> NeStack .
  op top_ : NeStack -> Elt .
  op pop_ : NeStack -> Stack .
  var X : Elt .
  var S : Stack .
  eq top push(X,S) = X .
  eq pop push(X,S) = S .
end
```

This seems about as simple a program as one could desire.

2.6.4 Instantiation

This subsection discusses instantiating the formal parameters of a parameterized module with actual modules. This construction requires a view from each formal parameter requirement theory to the corresponding actual module. The result of such an instantiation is to replace each requirement theory by its corresponding actual module, using the views to bind actual names to formal names, without producing multiple copies of shared submodules. For example, assuming that we are given a parameterized object SORTING[X :: PREORD], we can form

```plaintext
make SORTING-NATD is SORTING[NATD] endm
```

using the explicit view NATD, while

```plaintext
make NATLIST is LIST[NAT] endm
```

uses the default view from TRIV to NAT to instantiate the parameterized module LIST with the actual parameter NAT. Similarly, we might have
make REAL-LIST is LIST[REAL] endm

where REAL is the field of real numbers, using a default view from TRIV to REAL, or

make REAL-VSP is VECTOR-SP[REAL] endm

using the default view from FIELD to REAL. More interestingly

make STACK-OF-LIST-OF-REAL is STACK[LIST[REAL]] endm

uses two default views. (Note that Ada does not allow such a complex module expression, and would require using two steps.) In general,

make M is P[A] endm

is equivalent to

obj M is protecting P[A]. endo

where A may be either a module or a view.

Module composition in parameterized programming is more powerful than the purely functional composition of traditional functional programming, in that a single module instantiation can compose many different functions all at once. For example, a generic complex arithmetic module CPXA can be easily instantiated with any of several real arithmetic modules as actual parameter:

- single precision reals, CPXA[SP-REAL],
- double precision reals, CPXA[DP-REAL], or
- multiple precision reals, CPXA[MP-REAL].

Each instantiation involves substituting dozens of functions into dozens of other functions. While something similar is also possible in higher order functional programming by coding up modules as records, it is much less natural. Furthermore, with parameterized programming, the logic can be first order, so that understanding and verifying the code can be simpler. Moreover, semantic declarations are allowed at module interfaces (given by requirement theories), and module expressions allow many useful transformations and combinations other than application.

Our approach to parameterization was inspired by the Clear specification language [7, 8]. In fact, OBJ can be regarded as an implementation of Clear. In particular, the notion of view was developed in collaboration with Rod Burstall for use in Clear. Clear's approach was in turn inspired by some ideas in general system theory [27]. A key idea is the use of colimits of diagrams of theories to determine the result of module expression evaluation. Although colimits are beyond the scope of this paper, they give a precise foundation for parameterized programming, and moreover, a foundation that is independent of the particular choice of an underlying logical system, by making use of "institutions" [34]. Any logical programming language (in the sense made precise in [68]) can be given the features for parameterized programming described in this paper. This includes Eqlog, FOOPS and FOOPlog, as well as OBJ, so that the various combinations of functional, relational and object oriented programming are all covered.

Environments for ordinary programming languages are assignments of names to values (perhaps with indirection), but environments for parameterized programming languages must also include relations between modules. Section 2.5 already discussed the submodule inclusion relation that arises from module importation, giving an acyclic graph structure. Views must also be stored in environments, with source and target explicitly indicated, giving rise to a general graph structure. If submodule inclusions are seen as views, then the submodule hierarchy appears as a subgraph of the view graph.
There is an interesting further generalization of instantiation. First, notice that any parameterized module can be seen as a view \( p : R \rightarrow B \) from the requirement theory \( R \) (or the sum of all requirement theories, if there are more than one) into the body \( B \), which necessarily already includes \( R \). For example, \( \text{STACK}[x : : \text{TRIV}] \) is just the inclusion view

\[
\text{STACK} : \text{TRIV} \Rightarrow \text{STACKBODY},
\]

where the \( \text{STACKBODY} \) code is the same as given above for \( \text{STACK} \), except for replacing the name \( \text{STACK}[x : : \text{TRIV}] \) by just \( \text{STACKBODY} \). Then, given any binding view \( b : R \Rightarrow A \) to an actual module \( A \), we can form the instantiation \( p[b] \), which substitutes \( A \) into \( B \) after translation by \( p \) of \( R \); more precisely, the result of the application is given by what is called a “pushout” in category theory, as developed in the semantics of Clear [7, 9]. With this technique, a single body can be parameterized in many different ways. Thus, Ada’s idea to separate the “body” and “specification” (really, interface) parts of modules was good, but it is much more flexible if views are added.

### 2.6.5 Module Expressions

Module expressions not only permit defining, constructing and instantiating complex combinations of modules, they also permit modifying modules in various ways, thus making it possible to use a given module in a wider variety of contexts, and to improve the efficiency of existing code. The major combination modes are instantiation and sum. Among possible modifications are:

1. **extend** a module, by adding to its functionality;
2. **rename** some of its external interface;
3. **restrict** a module, by eliminating some of its functionality;
4. **encapsulate** some existing code;
5. **modify** the code inside a module.

This approach to program transformation [6, 81, 80] provides a broad range of program transformations right inside of programs, and it also easily takes account of data structure. Although module importation can be seen as a special case of parameter instantiation, it is more convenient to treat it separately; see Section 2.5. It is worth mentioning that modules may also have internal states; although this feature is neither discussed in this paper nor so far implemented, [26, 41] and [38] give further information on our approach to this important issue.

The simplest module expressions are the constants, including the built-in data types \( \text{BOOL}, \text{NAT}, \text{INT}, \text{QID}, \text{ID} \) and \( \text{FLOAT} \), plus any user-defined unparameterized modules available in the current environment. The theory \( \text{TRIV} \) is also built in, as are the n-ary parameterized \( \text{TUPLE} \) modules, which form n-tuples of sorts for any \( n > 1 \). All the requirement theories of \( \text{TUPLE} \) are \( \text{TRIV} \). For example, \( \text{TUPLE}[	ext{INT}, \text{BOOL}] \) is a module expression whose principle sort consists of pairs of an integer and a truth value. Another example is \( \text{TUPLE}[	ext{LIST}[	ext{INT}], \text{INT}, \text{BOOL}] \).

**Renaming** uses a view body (i.e., a sort mapping and an operation mapping) to create a new module from an old one. A renaming is applied to a module expression postfix following \* and modifies the syntax of module expression by applying the pairs that are given. To enrich a module expression, we need only import it into a module and then add the desired sorts, operations and equations; thus, we really do not need explicit enrichment transformations for module expressions. For example, we can use renaming to modify the \( \text{PREORD} \) theory, and then enrich it, as follows:

\[
\begin{align*}
\text{th} \ EQU & \text{ is using } \text{PREORD} \ast (\text{op } \_\<=\_ \text{ to } _\text{eq}_{\_}) \ . \\
\text{vars} E1 \ E2 : \text{Elt} \ . \\
\text{eq} (E1 \ \text{eq} E2) & = (E2 \ \text{eq} E1) \ . \\
\text{endth}
\end{align*}
\]
Another important module building operation creates a new module that adds, sums or combines all the information in its summands. (There are actually three modes for the summand modules, just as there are for imported modules; the default is extending.) An important issue here is sharing submodules that are imported by more than one summand. For example, in the sum A + B, both A and B probably protecting import BOOL, and they may also protect or extend NAT, INT and other modules. The sum should contain only one copy of such multiply imported modules. It is also very useful to sum views; the source of the sum view is the sum of its sources, and the target of the sum view is the sum of its targets. (See [33] for more details.)

3 Higher Order Programming and Verification

This section argues that higher order functions are not needed for higher order programming. The first subsection shows how some typical higher order programming techniques can be accomplished in first order logic with parameterized programming, and it also suggests some advantages of this approach. The second subsection gives a hardware verification example.

3.1 Some Examples

Higher order logic is useful in many areas, including the foundations of mathematics (e.g., type theory), extracting programs from correctness proofs of algorithms, describing proof strategies (as in LCF tactics [49]), modeling traditional programming languages (as in Scott-Strachey semantics), and studying the foundations of the programming process. Perhaps the main advantage of higher order programming over traditional imperative programming is its capability for structuring programs (see [57] for some cogent arguments and examples). However, a language with sufficiently powerful parameterized modules does not need higher order functions. We do not oppose higher order functions as such; however, we do claim that they can lead to unnecessarily complex programs, and that they can and should be avoided in programming languages. We further claim that parameterized programming provides an alternative basis for higher order programming that has certain advantages. In particular, the following shows that typical higher order functional programming examples are easily coded as OBJ programs that are quite structured, flexible and rigorous. Moreover, we can use theories to document any semantic properties that may be required of functions.

One classic functional programming example is motivated by the following two instances: (1) \( \text{sigma} \) adds a list of numbers; and (2) \( \text{pi} \) multiplies them. To encompass these and similar examples, we want a function that applies a binary function recursively over suitable lists. Let’s see how this example looks in vanilla higher order functional programming notation. First, a polymorphic list type is defined by something like

\[
\text{type list}(\text{T}) = \text{nil} + \text{cons}(\text{T}, \text{list}(\text{T}))
\]

and then the function we want is defined by

\[
\begin{align*}
\text{function iter} : & (\text{T} \to (\text{T} \to \text{T})) \to (\text{T} \to (\text{list}(\text{T}) \to \text{T})) \\
\text{axiom iter}(f)(a)(\text{nil}) &= a \\
\text{axiom iter}(f)(a)(\text{cons}(c, \text{list})) &= f(c)(\text{iter}(f)(a)(\text{list}))
\end{align*}
\]

so\(^4\) that we can write

\[
\begin{align*}
\text{sigma} & (\text{list}) \Rightarrow \text{iter}(\text{plus})(0)(\text{list}) \\
\text{pi} & (\text{list}) \Rightarrow \text{iter}(\text{times})(1)(\text{list})
\end{align*}
\]

---

\(^4\) Most people find the rank of \( \text{iter} \) rather difficult to understand. It can be simplified by uncurrying with products, and convention also permits omitting some parentheses; but these devices do not help much. Actually, we feel that products are more fundamental than higher order functions, and that eliminating products by currying can be misleading and confusing.
For some applications of \texttt{iter} to work correctly, \( f \) must have certain \textit{semantic} properties. For example, if we want to evaluate \( \texttt{pi}(\texttt{list}) \) with as many multiplications as possible in parallel, then \( f \) must be associative. (The algorithm first converts \texttt{list} into a binary tree, and then does all the multiplications at each tree level in parallel.) Associativity of \( f \) implies the following “homomorphic” property, which is needed in the correctness proof:

\[
\text{(H) } \texttt{iter}(f)(a) \left( \texttt{append}(\texttt{list})(\texttt{list}') \right) = f(\texttt{iter}(f)(a)(\texttt{list}))(\texttt{iter}(f)(a)(\texttt{list}'))
\]

for \( \texttt{list} \) and \( \texttt{list}' \) of the same type. Furthermore, if we want the empty list \texttt{nil} to behave correctly in property (H), then \( a \) must be an identity for \( f \).

Now let's do this example in OBJ. First, using mixfix syntax \_\_\_ for \( f \) improves readability somewhat; but much more significantly, we can use the requirement theory \texttt{MONOID} to assert associativity and identity axioms for actual arguments of a generic iteration module:

\begin{verbatim}
obj \texttt{ITER[M :: MONOID]} is protecting \texttt{LIST[M]}.
  op \texttt{iter} : \texttt{List} \to \texttt{M}.
  var \texttt{X} : \texttt{M}.
  var \texttt{L} : \texttt{List}.
  eq \texttt{iter} (\texttt{nil}) = \texttt{e}.
  eq \texttt{iter} (\texttt{X} \ \texttt{L}) = \texttt{X} \ast \texttt{iter} (\texttt{L}).
endo
\end{verbatim}

where \( \texttt{e} \) is the monoid identity. Note that \texttt{LIST[M]} uses the default theory view \texttt{TRIV => MONOID}. (This code uses an associative \texttt{List} concatenation, but it is also easy to write code using a \texttt{cons} constructor in OBJ.)

We can now instantiate \texttt{ITER} to get our two examples. First,

\begin{verbatim}
make \texttt{SIGMA} is \texttt{ITER[NAT*]} endm
\end{verbatim}

sums lists of numbers, while

\begin{verbatim}
make \texttt{PI} is \texttt{ITER[NAT*]} endm
\end{verbatim}

multiplies lists of numbers, where the view \texttt{NAT*} views \texttt{NAT} as a monoid under addition, while \texttt{NAT*} view \texttt{NAT} as a monoid under multiplication. These seem impressively clear and concise programs; moreover, they are written in a rigorous first order logic. Moreover, they are executable:

\begin{verbatim}
red in \texttt{SIGMA} : \texttt{iter}(1 \ 2 \ 3 \ 4) .
red in \texttt{PI} : \texttt{iter}(1 \ 2 \ 3 \ 4) .
\end{verbatim}

which of course give the expected results, 10 and 24, respectively.

Any valid instance of \texttt{ITER} has the property (H), which in the present notation is written

\[
\texttt{iter}(\texttt{L} \ \texttt{L}') = \texttt{iter}(\texttt{L}) \ast \texttt{iter}(\texttt{L}')
\]

and it is easy to prove this by induction, using OBJ to do the computations; note that this implies that (H) also holds for every instantiation of \texttt{ITER}. It is natural to state this fact with a theory and view, as follows:

\begin{verbatim}
th \texttt{HOM[M :: MONOID]} is
  protecting \texttt{LIST[M]}.
  op \texttt{h} : \texttt{List} \to \texttt{M}.
  var \texttt{L} \ \texttt{L}' : \texttt{List}.
  eq \texttt{h} (\texttt{L} \ \texttt{L}') = \texttt{h}(\texttt{L}) \ast \texttt{h}(\texttt{L}') .
endth
\end{verbatim}

\begin{verbatim}
view \texttt{ITER-IS-HOM[M :: MONOID]} from \texttt{HOM[M]} to \texttt{ITER[M]} is endv
\end{verbatim}
This view is parameterized, because property (H) holds for all instances; to obtain the appropriate assertion for a given instance \( \text{ITER}[A] \), just instantiate the view with the same actual parameter module \( A \). Since semantic requirements on argument functions cannot be stated in a conventional functional programming language, all of this would have to be done \textit{outside} of such a language. But 

OBJ can not only assert the monoid property, it can even prove that this property implies property (H), using methods described in [32].

Some have argued that it is actually much easier to use higher order functions and type inference to get declarations and instantiations automatically. However, the notational overhead of encapsulating a function in a module is really only a few keywords, and these could even be generated automatically by a structural editor from a single keystroke; moreover, this overhead can often be shared among many function declarations. There is also some overhead due to variable declarations. However, it can be reduced to almost nothing by two techniques: (1) let type inference give a variable the highest possible sort; and (2) declare sorts “on the fly” with a qualification notation. (We have not implemented this for OBJ3, because explicit declarations can save human program readers much effort in doing type inference.) Sort and operation declarations are needed in any approach, but our notation for them could be slightly simplified, if someone thought it worth the trouble. However, our view has been that the crucial issue is to make the structure of large programs as clear as possible; thus, tricks that slightly simplify notation for small examples are of little importance, and are of negative value if they make it harder to read large programs.

On the other hand, our notation for instantiation can often be significantly simplified, for example, if non-default views are needed, or if renaming is needed to avoid ambiguity when there is more than one instance of some module in a given context. For example,

\[
\text{make } \text{ITER-NAT} \text{ is } \text{ITER[view to NAT is op \(_*\) to \(_+\). endv] endm}
\]

is certainly more complex than \( \text{iter} \text{(plus)} \text{(0)} \). However, we could just let \( \text{ITER[(_+_.)\text{NAT}]} \) denote the above module, and we could go a bit further and let \( \text{iter[(_+_.)\text{NAT}]} \) denote the \text{iter} function itself, with the effect of creating the module instantiation that defines it, unless it is already present. Indeed, this is essentially the same notation used in functional programming, and it avoids the need to give distinct names for distinct instances of \text{iter}. Let us call this \textit{abbreviated operation notation}. It can also be used when there is more than one argument; note that the expression \( \text{iter[(_+_.)\text{NAT}]} \) uses default view conventions so that \text{El} maps to \text{Nat} (rather than \text{Bool}), and \text{e} maps to 0. (The abbreviated operation notation has not yet been implemented in OBJ, but the abbreviated view notation has been, and indeed is used in the next example below.)

An alternative is to model polymorphism within order sorted algebra; here one could declare certain parameterized objects to be polymorphic within some syntactic scope, and obtain the usual kind of polymorphism with a first order logic. However, I am not sure that this is worth the trouble, because it is rare to need many different instantiations of the same function symbol that cannot be handled by very simple module expressions.

For a second example, let us define the traditional function \text{map}, which applies a unary function to a list of arguments. Its interface theory requires a sort and a unary function on it (more generally, we could have distinct source and target sorts, if desired).

\[
\text{th FN is sort } S . \\
\text{op } f : S \rightarrow S . \\
\text{endth}
\]

\[
\text{obj MAP[F :: FN] is protecting LIST[F] .} \\
\text{op map : List \rightarrow List .} \\
\text{var X : S .} \\
\text{var L : List .} \\
\text{eq map(nil) = nil .} \\
\text{eq map(X L) = f(X) map(L) .}
\]
endo

We can now instantiate MAP in various ways. The following object defines some functions to be used in examples below.

```plaintext
obj FNS is protecting INT .
  op sq_ : Int -> Int .
  op dbl_ : Int -> Int .
  op _*3 : Int -> Int .
  var N : Int .
  eq dbl N = N + N .
  eq N *3 = N * 3 .
  eq sq N = N * N .
endo
```

Our first instantiation of this uses a view FN => FNS that maps f to sq_, using an abbreviated notation:

```plaintext
make TEST1 is MAP[(sq).FNS] endm
```

Now a sample reduction

```plaintext
reduce map(0 1 -2 3) .
*** result NeList: 0 1 4 9
```

Next, some further reductions using views with operation abbreviation notation:

```plaintext
reduce in MAP[(dbl).FNS] : map(0 1 -2 3) .
*** result NeList: 0 2 -4 6
reduce in MAP[(*3).FNS] : map(0 1 -2 3) .
*** result NeList: 0 3 -6 9
```

The following module does another classical functional programming example, applying a given function twice; some instantiations are also given.

```plaintext
obj 2[F :: FN] is
  op 2x : S -> S .
  var X : S .
  eq 2x(X) = f(f(X)) .
endo
```

```plaintext
reduce in 2[(sq).FNS] : 2x(3) .
*** result Int: 81
reduce in 2[(dbl).FNS] : 2x(3) .
*** result Int: 12
reduce in 2[2[(sq).FNS]* (op 2x to f)] : 2x(3) .
*** result Int: 43046721
```

Let us consider this last example more carefully. Since 2 applies f twice, the result function 2x of the first instantiation applies sq_ twice, i.e., raises to the 4th power; then the second instantiation applies that twice, i.e., raises to the 16th power. The renaming is given to prevent syntactic ambiguity of 2x but could be avoided by using qualification.

To summarize, the difference between parameterized programming and higher order functional programming is essentially the difference between programming in the large and programming in
the small. Parameterized programming does not just combine functions, it combines modules. This parallels one of the great insights of modern abstract algebra, that in many important examples, functions should not be considered in isolation, but rather in association with other functions and constants, along with the axioms that they satisfy, and with their explicit sources and targets. Thus, the invention of abstract algebras (for vector spaces, groups, etc.) parallels the invention of program modules (for vectors, permutations, etc.); parameterized programming makes this parallel more explicit, and also carries it further, by introducing theories and views to document semantic requirements on function arguments and on module interfaces, as well as to assert provable properties of modules (such as property (H) above). As we have already noted, it can be more convenient to combine modules than to compose functions, because a single module instantiation can compose many conceptually related functions at once, as in the complex arithmetic (CPAX) example mentioned in Section 2.6.4. On the other hand, the notational overhead of theories and views is excessive for applying just one function. However, this is exactly the case where our abbreviated view and operation notations can be used to advantage. And we should not forget that it can be much more difficult to reason with higher order functions than with first order functions; in fact, the undecidability of higher order unification means that it will be very difficult to mechanise certain aspects of such reasoning. Also, it is much easier to compile and interpret first order programs. It is worth noting that Poigné [77] has found some significant difficulties in combining subsorts and higher order functions, and we hope to have been convincing that subsorts are very useful. Finally, note the experience of many programmers, and not just naive ones, that higher order notation can be very difficult to understand and to use.

3.2 Hardware Specification, Simulation and Verification

This subsection develops a computer hardware verification example. The crucial advantage of using a logical programming language here is that reductions really are proofs, because programs really are logical theories. The following propositional calculus decision procedure object is also an excellent example of software reuse, since its original form was written years before we thought of using it for hardware verification [44]:

```plaintext
obj PROP is sort Prop.
    protecting TRUTH & QID.
subsorts Id Bool < Prop.

op _and_ : Prop Prop -> Prop [assoc comm prec 2].
op _xor_ : Prop Prop -> Prop [assoc comm prec 3].
vars p q r : Prop.
eq p and false = false.
eq p and true = p.
eq p and p = p.
eq p xor false = p.
eq p xor p = false.
eq p and (q xor r) = (p and q) xor (p and r).

op _or_ : Prop Prop -> Prop [assoc comm prec 7].
op not_ : Prop -> Prop [prec 1].
op _implies_ : Prop Prop -> Prop [prec 9].
op _iff_ : Prop Prop -> Prop [assoc prec 11].
eq p or q = (p and q) xor p xor q.
eq not p = p xor true.
eq p implies q = (p and q) xor p xor true.
eq p iff q = p xor q xor true.
```

19
endo

Here and and xor are constructors, subject to the first group of equations, while the second group introduces derived operations. The attribute prec n means that the operation it follows has precedence n, where lower precedence means tighter binding. The declaration Id Bool < Prop prepares the way for overloading all the Boolean operations, and also includes identifiers among the propositions for use as "propositional variables."

The code below first defines time for use in bit streams, which are functions from Time to Prop.

A requirement theory LINE is defined, and then a NOT gate using it. The object F introduces the variables t and f0, which are a "generic" time and input stream, respectively. Finally, two NOT gates are composed and applied to F, using renaming to avoid syntactic ambiguities, and some rather nice default views. Note that an expression of the form t iff t' reduces to trueiff t and t' reduce to the same thing.

Three extended equations are actually proved, the first of which was described informally above. In more detail, this assertion has the form

\[ P \models_{\Sigma} (\forall \Phi) r \]

where \( P \models s \) means "s is satisfied by the initial algebra of P," where \( \Sigma \) is the union of the signatures of the OBJ objects PROP and TIME, \( P \) is the union of their equations, \( \Phi \) is the signature containing three functions \( f_0, f_1, f_2 \) from Time to Prop, and \( r \) is \((s_1 \land s_2) \Rightarrow s_3\), where

\[
\begin{align*}
s_1 &= (\forall t)(f_1(s t) = \text{not} f_0(t)), \\
s_2 &= (\forall t)(f_2(s t) = \text{not} f_1(t)), \\
s_3 &= (\forall t)(f_2(s t) = f_0(t)).
\end{align*}
\]

Because some readers may be surprised to see equations with second order quantifiers proved using just ground term reduction, some basics needed for the correctness of this verification technique are given in Appendix A; details may be found in [32].

obj TIME is sort Time .
  op 0 : -> Time .
  op s_ : Time -> Time .
endo

th LINE is
  protecting TIME + PROP .
  op f : Time -> Prop .
endth

obj NOT[L :: LINE] is
  op g : Time -> Prop .
  var T : Time .
  eq g(0) = false .
  eq g(s T) = not f(T) .
endo

obj F is
  extending TIME + PROP .
  op t : -> Time .
  op f0 : Time -> Prop .
endo
make 2NOT is NOT[NOT[F]*(op g to f)]*(op g to f2) endm

reduce f2(s t) iff f0(t) .
*** result Bool: true

reduce f2(s t) iff not f(t) .
*** result Bool: true

reduce f(s t) iff not f0(t) .
*** result Bool: true

Note that parameterized modules make the code much more readable than it would be without them. These techniques seem equally effective for more difficult examples of hardware specification, simulation and verification, as discussed in [32]. Parameterized programming is attractive for this application, because there can be many instances of just a few kinds of basic gates.

4 Summary and Discussion

This paper has shown that higher order functions are not needed for typical higher order programming techniques, and in fact has shown that there are some advantages to using first order parameterized programming instead, including greater flexibility and the possibility of imposing semantic requirements on the arguments of functions. Moreover, Poigné [77] has found some significant difficulties to combining subsorts with higher order functions, and because this paper has argued that subsorts can be very useful, that can be seen as another argument against higher order functions. Also, it can be much more difficult to reason about properties of higher order functions; in fact, the undecidability of higher order unification means that it can be very difficult to mechanize certain aspects of such reasoning. Moreover, it should be easier to compile, optimize, and interpret purely first order programs. Finally, note the experience of many programmers, and not just naive ones, that higher order notation can be very difficult to understand and to use. Waxing a bit philosophical, we may say that ordinary computation (manipulating bits according to already given instructions) is inherently first order, whereas mathematics is inherently higher order (we can always reason about our reasoning).

Appendix A presents a useful extension of equational logic to quantification over functions, and in particular justifies a perhaps surprising technique for proving second order quantified equations using just ground term reduction. This gives a powerful calculus for first order reasoning about first order functions, and I think it may capture much of the reasoning that is actually needed for functional programming.

I think we can conclude from all this that it is better to “factorize” code with parameterized modules than with higher order functions, and in fact, that it is better to avoid higher order functions whenever possible. From this, one could conclude that the essence of functional programming cannot be the use of higher order functions, and therefore must be the lack of side effects. However, I feel that the true essence may well be having a solid basis in equational logic, because this not only avoids side effects, but more importantly, it supports simple equational reasoning about programs and transformations, as needed for powerful programming environments.

Instead of seeing parameterized programming as a way to supplant higher order logic, we can see it as an interesting direction in which to generalize higher order logic, since the calculus of views must confront issues beyond those formalized in the λ-calculus, including the following:

1. The basic “types” (which are the unparameterized modules, including BOOL, NAT, MONOID and PREORD, as well as whatever a user chooses to define) denote not just classes of functions, but categories of models (order sorted algebras in the case of OBJ).
2. Similarly, parameters range not over classes of functions, but over classes of modules, and these classes are subject to semantic constraints (e.g., equations).

3. Modules include both theories and objects.

4. Parameterized modules represent functors between classes of models.

5. Views are an entirely new feature, not found in the λ-calculus.

These points perhaps deserve some elaboration. First, they suggest it might be awkward to “code up” parameterized programming into some form of denotational semantics (e.g., in the style of [65]) or type theory (e.g., in the style of Pebble [11], PX [54] or Martin-Löf’s type theory [66]). Even if we had such an encoding, it would not be the sort of notation that programmers should have to deal with in practice, but would be somewhat like trying to program with Gödel numbers; however, it could be valuable in theoretical studies. (Of course, one can code up λ-calculus or type theory in OBJ, but that is quite a different issue.) Moreover, such an encoding of parameterized programming would not emphasize what seem to be the really fundamental entities: just as types play a secondary role as indices for functions in the typed λ-calculus, and objects play a secondary role as indices for morphisms in category theory, so it may be that modules play a secondary role as indices to views in parameterized programming.

Because we claim first order proof theory as a major advantage for OBJ, it is interesting to see how far parameterization can be pushed without endangering this asset. It is possible to achieve the equivalent of parameters that are themselves parameterized through the nesting of parameterized modules. This is a special case of what type theory calls “dependent types.” See [26] for further discussion. Whether there are significant applications for some of the more elaborate possibilities that are allowed by type theory remains unclear. It also seems interesting to inquire whether we can find a suitable categorical semantics, in terms similar to the Cartesian closed category characterization of the λ-calculus (of course, the semantics of Clear [8] has already shown how to do everything that this paper needs using colimits of theories). Seeley’s locally Cartesian closed categories [82] seem relevant, as do Cartmell’s S-categories [13], since the extending hierarchy of parameterized module inclusions is preserved under instantiation; see also his hierarchical categories. There is also some interesting recent work by John Gray on dependent abstract data types. Altogether, this seems a promising area for future research.

Acknowledgements

I wish to thank: Professor Rod Burstall for his extended and on-going collaboration on Clear and its foundations, which inspired the parameterization mechanism of OBJ; Dr. José Meseguer for his invaluable contributions to every aspect of OBJ including its theoretical foundations, its implementation, and its applications; Timothy Winkler for his many suggestions concerning the design and theory of OBJ; Professor Jean Pierre Jouannaud for his efforts to educate me on the theory and practice of rewrite rules; Dr. Kokichi Futatsugi for his work on programming methodology using OBJ; and Victoria Stavridou for her efforts to use OBJ for hardware specification and verification. I also thank José Meseguer and Timothy Winkler for their very valuable comments on drafts of this paper.

The research reported in this paper has been supported in part by grants from the Science and Engineering Research Council, the National Science Foundation, and the System Development Foundation, as well as contracts with the Office of Naval Research and the Fujitsu Corporation.

A Second Order Quantifiers for First Order Equations

This appendix generalizes the standard case of equational logic, which only quantifies over constants, to permit quantification over arbitrary function symbols. Although this is a kind of second
order quantification, it should be seen as taking first order equational logic to its limit, rather than
as an incursion into the second order realm; what is essential is that the terms themselves are first
order. We will see that this generalization can be very useful. However, the mathematics is an easy
extension of the standard case; indeed, it is hard to see why it has not been thought of before. This
appendix includes some new results justifying the use of ground term reduction to prove equations
with second order quantifiers. The result is a powerful first order calculus for reasoning about (first
order) functions, which I believe is more satisfactory than trying to use the λ-calculus or some other
more general (and thus less powerful) tool.

Unlike the body of the paper, some familiarity with the basics of universal algebra is probably
needed to read this appendix, e.g., [47, 69] Although OBJ is actually based on order sorted
equational logic, the following discussion uses unsorted equational logic for expository simplicity.

A signature $\Sigma$ is a family of sets, for $n = 0, 1, 2 \ldots$ An element of $\Sigma_n$ is a function symbol
of arity $n$, and in particular, elements of $\Sigma_0$ are constant symbols. Given signatures $\Sigma$ and $\Phi$,
their union is defined by

$$(\Sigma \cup \Phi)_n = \Sigma_n \cup \Phi_n.$$ 

A $\Sigma$-algebra is a set $A$ and an interpretation function for $\Sigma$ into $A$, i.e., a family of functions
$i_n: \Sigma_n \rightarrow [A^n \rightarrow A]$ that interpret the function symbols in $\Sigma$ as actual functions on $A$. Since $A^0$
is some one-point set, say $*$, for $c \in \Sigma_0$ we can identify $i_0(c)$ with $i_0(c)(*)$, a point in $A$. Generally, we
write just $f$ for $i_n(f)$ in $A$.

Given $\Sigma$-algebras $A$ and $B$, a $\Sigma$-homomorphism $h: A \rightarrow B$ is a function $h: A \rightarrow B$ such that

$$h(f(a_1, \ldots, a_n)) = f(h(a_1), \ldots, h(a_n))$$

for each $f$ in $\Sigma_n$ and in particular, such that $h(c) = c$ for each $c$ in $\Sigma_0$.

Given a signature $\Sigma$, we let $T_\Sigma$ denote the $\Sigma$-algebra of all ground $\Sigma$-terms. Recall that $T_\Sigma$
is initial in the sense that given any other $\Sigma$-algebra $A$, there is a unique $\Sigma$-homomorphism from $T_\Sigma$
to $A$.

We now define a $\Sigma$-equation to consist of a signature $\Phi$ of variable symbols (disjoint from $\Sigma$), plus a pair of $(\Sigma \cup \Phi)$-terms. We write such equations abstractly in the form

$$(\forall \Phi) \ t = t'$$

and concretely in the form

$$(\forall f, g, x, y) \ t = t'$$

where the arities of $f, g, x, y$ can (presumably) be inferred from their uses in $t$ and $t'$.

An example of the power of this kind of equation arises in a denotational style semantics for
expressions, where one would normally have to write equations

$$(\forall e, e') \ [e + e'](\rho) = [e](\rho) + [e'](\rho)$$

$$(\forall e, e') \ [e \cdot e'](\rho) = [e](\rho) \cdot [e'](\rho)$$

$$(\forall e, e') \ [e \times e'](\rho) = [e](\rho) \times [e'](\rho)$$

$$\ldots$$

instead of the following much simpler equation which quantifies over the binary function symbol $\star$,

$$(\forall e, e', \star) \ [e \star e'](\rho) = [e](\rho) \star [e'](\rho)$$
(This equation actually has a slightly different meaning from the finite set of equations given above, since it asserts the homomorphic property for any possible *; but we can also get the other semantics by using a conditional equation.)

In the standard case, only $\Phi_0$ can be nonempty, and so $\Phi$ can be identified with a set $X$ of (standard) variables. In this case, the union signature is written $\Sigma(X)$, and such standard equations are written abstractly in the form

$$(\forall X) \; t = t'$$

where $t, t'$ are $\Sigma(X)$-terms, and concretely in the form

$$(\forall x, y, z) \; t = t'.$$

Given a $\Sigma$-algebra $A$ and also an interpretation $f: \Phi \rightarrow A$ of the variable symbols in $\Phi$ into $A$, there is a unique extension of $f$ to a $(\Sigma \cup \Phi)$-homomorphism, $f^\#: T_{\Sigma \cup \Phi} \rightarrow A$, by the initiality of $T_{\Sigma \cup \Phi}$ where $A$ is regarded as a $(\Sigma \cup \Phi)$-algebra by using $f$ to extend the interpretation function $i$ of $A$ from $\Sigma$ to $\Sigma \cup \Phi$. Then a $\Sigma$-term $t$ with variables in $\Phi$ is just an element of $T_{\Sigma \cup \Phi}$ and a $\Sigma$-algebra $A$ satisfies the $\Sigma$-equation $(\forall \Phi) \; t = t'$ iff for any interpretation $f: \Phi \rightarrow A$, we have that $f^*(t) = f^*(t')$ in $A$; in this case we write

$A \models (\forall \Phi) \; t = t'.$

A $\Sigma$-algebra $A$ satisfies a set $E$ of $\Sigma$-equations iff it satisfies each $e$ in $E$, and in this case we write

$A \models E$.

A presentation $(\Sigma, E)$ consists of a signature $\Sigma$ and a set $E$ of $\Sigma$-equations. Any OBJ program $P$ defines a presentation $(\Sigma, E)$ where both $\Sigma$ and $E$ are finite and (at present) $E$ is standard; the details of how $P$ yields $(\Sigma, E)$, which involve theories, views, colimits, etc., need not concern us here; we simply identify $P$ with its presentation $(\Sigma, E)$ and ignore the concrete syntax of OBJ. (Of course, the OBJ program really defines an order sorted presentation, but we here are restricting attention to the unsorted case.) Since OBJ is both a programming language and a specification language, it admits two kinds of program $P$:

1. objects, whose intended semantics is a standard model for $P$; and
2. theories, whose intended semantics is the variety of all models for $P$.

The second case is generally appears in an auxiliary role, because we are usually interested in defining particular data structures and particular functions over them. A basic intuition for equational logic is that standard models are initial models. Reduction techniques cannot be sufficient to prove all properties of initial models, and in particular, should be supplemented with induction techniques.

Now writing

$E \models (\forall \Phi) \; t = t'$

to mean that

$A \models (\forall \Phi) \; t = t'$

for every $A$ such that

$A \models E$

we have

**Theorem 1:** Given disjoint signatures $\Sigma$ and $\Phi$, given a set $E$ of $\Sigma$-equations, and given $t, t'$ in $T_{\Sigma \cup \Phi}$ then


24
\[ E \models \forall \Phi \ t = t' \text{ iff } E \models \Sigma \cup \Phi \ (\forall \emptyset) \ t = t' \]

where \( \emptyset \) denotes the empty signature.

**Proof:** Each condition is equivalent to the condition

\[ f^*(t) = f^*(t') \text{ for every } \Sigma \cup \Phi \text{-algebra } A \text{ satisfying } E, \]

where \( f^*: T_{\Sigma \cup \Phi} \rightarrow A \) is the unique homomorphism. \( \square \)

It is pleasing that this proof is so simple, and is based entirely on the semantics of satisfaction, rather than on any particular choice of rules of deduction.

It now follows that if we view \( E \) as **rewrite rules** and if \( E \) reduces \( t \) and \( t' \) to the same value, then \( E \models \forall \Phi \ t = t' \). This helps to justify the hardware proof in Section 3.2; the full details may be found in [32].

The moral of this appendix is that, not only are higher order functions unnecessary for higher order programming, but higher order logic is also unnecessary for reasoning about functional programs. More detail can be found in [32], including a completeness theorem, some induction principles, and techniques for verifying generic modules.

**References**


based on unpublished notes handed out at the Symposium on Algebra and Applications, Stefan Banach Center, Warszawa, Poland.


[23] Christopher Paul Gerrard. The specification and controlled implementation of a configuration
management tool using OBJ and Ada. In Joseph Goguen, Derek Coleman, and Robin Gallimore,
To appear.

Proceedings, Ninth CAAP (Bordeaux), pages 139-153. Cambridge University Press, 1984. Also
Forschungsbericht Nr. 169, Universität Dortmund, Abteilung Informatik, 1983.

editor, Recent Trends in Data Type Specification, volume Informatik-Fachberichte 116, pages
89–103. Springer-Verlag, 1985. Selected papers from the Third Workshop on Theory and
Applications of Abstract Data Types.


[27] Joseph Goguen. Mathematical representation of hierarchically organized systems. In E. At-

International Symposium on Category Theory Applied to Computation and Control, pages 234–
249. University of Massachusetts at Amherst, 1974. Also in Lecture Notes in Computer Science,

First IFIP Working Conference on Formal Description of Programming Concepts, pages 21.1–

[30] Joseph Goguen. How to prove algebraic inductive hypotheses without induction: with appli-
cations to the correctness of data type representations. In Wolfgang Bibel and Robert Kowal-
ski, editors, Proceedings, Fifth Conference on Automated Deduction, pages 356–373. Springer-


[32] Joseph Goguen. OBJ as a theorem prover, with application to hardware verification. In V.P.
Subramanyan and Graham Birnwhistle, editors, Current Trends in Hardware Verification and

editors, Software Reusability, Volume I: Concepts and Models, pages 159–225. Addison-Wesley,
1989.

[34] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and
programming. Journal of the Association for Computing Machinery, to appear. Report ECS-
LFCS-90-106, Computer Science Department, University of Edinburgh, January 1990; prelimi-
nary version, Report CSLI-85-30, Center for the Study of Language and Information, Stanford
University, 1985, and remote ancestor in “Introducing Institutions,” in Proceedings, Logics of
Programming Workshop, Edward Clarke and Dexter Kozen, editors, Springer-Verlag Lecture
Notes in Computer Science, Volume 164, pages 221-256, 1984.


Contents

1 Introduction ......................................................... 1
   1.1 Parameterized Programming .................................. 1
   1.2 Some History .................................................. 2

2 Aspects of OBJ .................................................. 3
   2.1 Strong Sorting ................................................. 4
   2.2 Operation and Expression Syntax .............................. 4
   2.3 Subsorts ......................................................... 5
   2.4 Semantics ....................................................... 6
      2.4.1 Operational Semantics .................................. 6
      2.4.2 Denotational Semantics ................................ 8
   2.5 Hierarchical Structure ........................................ 9
   2.6 Parameterization .............................................. 9
      2.6.1 Theories .................................................. 10
      2.6.2 Views ..................................................... 11
      2.6.3 Parameterized Modules ................................. 12
      2.6.4 Instantiation ............................................. 12
      2.6.5 Module Expressions ..................................... 14

3 Higher Order Programming and Verification .................. 15
   3.1 Some Examples ............................................... 15
   3.2 Hardware Specification, Simulation and Verification ....... 19

4 Summary and Discussion .......................................... 21

A Second Order Quantifiers for First Order Equations ......... 22