Forth Meets Smalltalk

Charles B. Duff and Norman D. Iverson

Kriya Systems, Inc.
505 N. Lakeshore #5510
Chicago, IL 60611

Abstract

Forth has certain inadequacies that limit its use as a general-purpose production language, particularly for building large applications. The object-oriented approach of Smalltalk-80 is used as a model for extending the Forth compiler to create a hybrid language. The resulting system permits definition of classes and objects in the manner of Smalltalk-80, and can be fully intermixed with standard Forth.

Programming Environments

When evaluating a language for use in the production of commercial applications, two kinds of efficiency are of paramount concern: that of the produced code, and that of the programmer. The first point is most important in real-time and extremely compute-bound applications, while the second is relevant to any software vendor attempting to operate a viable business. While Forth has advantages over many other languages in both of these areas, it is the opinion of the authors that more emphasis has been placed on the efficiency of the language than that of the programmer in Forth's evolution thus far. As a developer of applications exclusively in Forth, it has been our experience that certain additions and extensions to the language environment can reduce the amount of programmer time spent in coding and debugging without adding excessive overhead.

This paper will describe Neon™, a language that we have developed for the Macintosh computer. Neon incorporates an indirect-threaded Forth kernal, but greatly extends it to provide more functionality for the programmer. We have focused our efforts on three areas:

Data Structures — such important language constructs as Strings, Arrays and Lists are not inherent in the standard Forth vocabularies. Many implementations of such data structures have been suggested in the Forth literature, but as long as they remain unstandardized, a Tower of Babel situation prevails that impedes transportability and programmer efficiency. CREATE DOES> is inadequate as a general facility for defining new constructs because it allows only one behavior per datum. In this paper we will describe a general, powerful facility for the construction of data structures that encourages reuse and standardization, but does not preclude extension.

Protection of local data — we have seen much maintenance time spent overcoming unforeseen side effects due to the fact that Forth offers minimal protection against global access to a datum. Such protection, while it eliminates certain tricks and shortcuts, supports a more secure development process, particularly when multiple programmers are involved.

Organization and Readability — while good naming can make Forth highly readable, far too

often many factors combine to make Forth programs difficult to comprehend, even by the programmer who wrote them. The worst offender in this area is the proliferation of stack operations that occurs in certain types of Forth words.

Guiding the Development Process

While it is undesirable for a language to be overbearing and too protective, it is beneficial to encourage programmers to use common techniques. More code can be shared, ease of maintenance improves, and many other benefits accrue from standardized approaches to programming. The most effective way to bring this about is to provide a construct that is so powerful and elegant, it makes little sense to solve a problem in a different way. In Forth, the most profitable area in which to exercise this kind of guidance would seem to be that of data structure definition. We will concentrate our discussion on that topic, and begin by proposing a set of design goals for an improved data structure process in Forth.

Guidelines for Building "Ideal" Extensible Data Structures

1. Realism — The language used in the computer domain should closely map the problem domain. As an example, Forth's stack manipulation words, such as ROT, SWAP and DUP have nothing to do with the nature of the problem being solved; they are artificial formalisms that owe their existence only to the relatively sparse nature of the Forth compiler. While this provides an efficient method for manipulation of parameters, it often severely compromises readability. [GLAS83] points out that constructs such as named arguments, that improve Forth's readability also facilitate the process of writing Forth code. Such techniques as procedural arguments can result in simpler, more easily understood code [LUO84]. Local variables (see [PER82], [BOW82]) are an important facility that impact realism as well as the issue of localization, discussed below.

On a larger scale, certain basic characteristics of a language greatly affect its capacity for realism. Procedural languages, such as C and Pascal, define symbolic templates for data and then allow the programmer to create a sequence of procedures or "recipes" that manipulate the data in various ways to produce a desired result. Both the sequential nature of these descriptions and the rather separate treatment of code and data take them a step away from the problem being solved, rather like telling someone else how to solve a problem. On the other hand, languages such as Simula and Smalltalk have arisen that allow the programmer to build models of phenomena. This approach is more concrete, easier to comprehend, and closer to the language of the problem [GOL83], [COX84], [WEG84]. These languages tend to bundle data with the access methods uniquely applicable to the data, in a common structure called an *object*.

Forth can exhibit characteristics of either approach because it is not a rigidly defined and static language. Glass points out that Forth exhibits many traits of a functional language, and can serve as a basis for a more usable and elaborate functional programming environment while preserving the efficiencies of the threaded architecture [GLAS84]. Our experience in using Forth as the basis for an object-oriented language supports this conclusion.

2. Transparency — Names for operations on data structures should be generic, and insensitive to the actual implementation of the data. [LEA83] provides an indication of the difficulty of achieving this in Forth; for instance, the words QPUT, QBPUT, QWPUT and QSPUT all accomplish stores to queues of differing data types. Ideally, the word PUT would suffice for all of the implementations.

3. Localization — Unless code and data is localized to the portion of a program in which it is relevant, negative side effects and interactions are likely to occur as a result of programmer mistakes that could be detected at compile time [MYE78]. It should be possible

to restrict the scope of names to a small subset of the application (such as a single procedure), and to apply this technique in an arbitrary hierarchy.

Vocabularies provide a partial, but flawed, capacity for localization in Forth. While they restrict the scope of names, their behavior is difficult to reconstruct from a long source listing because of the implicit nature of vocabulary linkage. Vocabularies are full-blown examples of modal behavior, changing the behavior of standard language elements without making the expected differences explicit. Solutions such as ONLY make vocabularies usable, but they remain impractical for the degree of localization analogous to Procedures in Pascal. Applying them at this level could easily demand 50 or 100 vocabularies in a large application, which would certainly present an interesting management problem for the unfortunate soul attempting to understand the source.

4. Independence — Ideally, two modules should minimize the amount and complexity of the data exchanged between them. Also, their logic should not rely upon implementation details of other modules. A good design emphasizes clear, simple data interfaces between modules. [MYE78] contains an excellent discussion of data coupling between modules.

5. Encapsulation — This is a more specific application of the last point. The implementations of access methods relevant to a given data structure should be hidden from the rest of the system, and logically bundled with the data structure itself. They should be available as generic operations, in the manner of [2] above. [COX84] discusses the message/object programming style of Smalltalk-80 as a paradigm for other languages to achieve encapsulation of access methods.

6. Aggregation — Just as we can build higher-level Forth words out of more primitive subordinates, we should be able to construct composite data structures out of existing ones. Traditional methods of defining data structures in Forth do not permit easy nesting or aggregation.

7. Inheritance — Often, a new data structure shares many characteristics with one previously defined. We should be able to inherit the data and access methods of a previous data structure without a great deal of effort. This has benefits in programming time as well as code bulk.

8. Efficiency — Clearly, it is important to preserve as much efficiency in code space and execution time as is practical. Ideally, a dynamic decision could be made by the programmer to trade time for space, or vice-versa. This flexibility is currently exhibited by Forth.

The Search for a Solution

After surveying the existing literature for approaches that would meet these requirements, we failed to find any that could satisfy all of them. [FOR81] demonstrates an effective method of achieving generic operators that was not designed to solve the general class of problems that we are concerned with. The work on multiple code-field structures (see [ROS82], [DOW83], [SCH83]) is very efficient, and attractive in terms of transparency, realism and encapsulation, but has disadvantages with respect to the other points. In particular, it explicitly uses some of the intimate details of a word's compiled structure, which violates the principle of independence. In addition, no facility for nesting structures is provided by this approach alone. Nevertheless, we feel that multiple-codefield structures offer many advantages over Forth's existing defining word compilers, and should be considered as an eventual replacement for CREATE DOES>. We made much use of these techniques in building the framework for the system that we will describe.

A Cross-Cultural Marriage

The object-oriented approach of Smalltalk-80 (see [COX84], [GOL83], [KRA83], [BYT81]) has benefits in all of the areas except efficiency, which is one of Forth's paramount

virtues. We felt that a marriage between the two languages could be very attractive if it could preserve efficiency in both time and space. Thus, we undertook to build an object-oriented environment on top of the Forth nucleus, preserving as much as possible the existing Forth interpreters and compiler. Our intent was to allow the object-oriented and conventional methods of writing Forth code to freely coexist, so that each could be exploited for the tasks to which it was suited.

The Smalltalk-80 language was developed at Xerox PARC, descending from the work of Alan Kay and the Dynabook project (see [KAY77], [GOL83]). It was originally conceived as a simulation language that would allow anyone to simulate real-world phenomena without being an experienced programmer. Smalltalk has grown into a highly sophisticated interactive environment, well-suited for education and research. It is built around a small, well-defined set of concepts that are consistently applied throughout the system. These include:

Objects — everything in Smalltalk is an object, which is an entity that associates a unique data area with a set of access methods for that data. An object has enough intelligence about its private data to manipulate the data in response to high-level *messages* from other objects. These messages assume no knowledge of the internal representation of the data or its manipulation techniques. This situation is analogous to someone asking an accountant to prepare their tax return—they need make only the high-level request "prepare my return", and provide some raw information about their finances. The accountant later delivers a finished product, without involving the requestor in the mechanical details of the process. Different accountants might have different internal methods, but this is transparent to the requestor (the message remains the same).

Classes — every object is an *instance* of (and is created by) a class. A class describes the data format and methods for all of the objects which it instantiates. Classes are hierarchically linked in a superclass/subclass relationship. A subclass inherits all of the data and methods of its superclass, and adds or overrides some specific to itself. In the above example, the word Accountant could describe the class of the individual (the object), who is an instance of that class. Class Accountant might describe in general terms the operations performed by any accountant, while subclasses of Accountant (such as Computerized Tax Accountant) would add the details of specific accounting methods. All subclasses of Accountant would share through inheritance the general methods defined in Accountant (such as how to use a general ledger).

Methods — a class contains a description of all of the different operations that it can perform. Each is referred to with a public name, called a *selector* (such as "Prepare Taxes"). When a method is invoked, the caller knows nothing about the internal implementation of that method. Two techniques are available for invoking methods. *Early binding* looks up the method for a given selector in a given class at compile time, and compiles the actual execution address of the method. This is very efficient, but requires that the class of an object be known at compile time, which is sometimes an unsuitable restriction. *Late binding* defers the lookup of the method until runtime. This results in complete independence of later modules from those compiled earlier, since the only form of linkage is selector name. Smalltalk-80 provides only the latter form of binding, sacrificing efficiency for generality.

Messages — Objects communicate via messages, which consist of untyped *parameters* and *selectors* that separate the parameters and identify the method desired. Messages minimize the level of coupling between objects, since they cannot depend upon implementation details. In distinction, each Forth word requires a unique name, because the dictionary is basically global. Several Forth words that perform a similar operation on different data types must have different names.

Named Instance Variables — These make up the private data of an object. These variables are themselves objects, which allows arbitrary nesting of structures. Each has a name, which is only available to the methods of the class and its subclasses. Objects not in the subclass chain of the class containing a named instance variable cannot access that variable directly.

Indexed Instance Variables — some objects contain a variable-length *indexed area* that can be both privately and publicly accessed. This is done by sending the object a message with an index and a selector denoting indexed access.

The scope of this paper does not permit a more complete exposition of the principles behind Smalltalk-80; for that see [GOL83].

Tradeoffs

A difficulty with Smalltalk is that it is poorly suited as a production language due to the runtime overhead of its bytecode interpreter, method lookup due to late binding, and extensive dynamic memory management (see [KRA83], esp. chapter 18). In general, Smalltalk is optimized for a highly interactive environment, rather than a turnkey application under which time and space are critical. Also, Smalltalk tends to hide the hardware from the programmer, making it inappropriate for many of the problems currently being solved by Forth. Smalltalk cannot consider any arbitrary data area as an object (such as an Operating System buffer); all objects must be created as instances of actual classes.

In our efforts to build a class/object environment over Forth, we decided to make certain tradeoffs for the sake of efficiency. Our system uses early binding as its default mode, which means that the class of an object must be known at compile time. This permits us to streamline the invocation of a method down to the bare essentials and avoid an expensive runtime method lookup. If the programmer encounters a situation in which late binding is preferred, code can be compiled to perform the method lookup at runtime. Secondly, we elected to retain Forth's static memory allocation technique, and build objects into the dictionary as words. This is fast and well-suited for production code. Our implementation associates code with each object that returns its address, which will allow us to transparently build objects in a dynamic memory pool if one is provided and managed by the system. This is the case on the Macintosh.

Thirdly, in Smalltalk, classes are objects themselves; we chose to forego elegance and build classes as special case words that have compile time and run time behavior. Any of these decisions could change provided that we find a way to minimize the cost in time and space.

Object-oriented Programming in Forth

When operating under the message/object paradigm, the task of programming becomes a very different process. Instead of coding sequences of operations upon data, the programmer creates classes, instances of the classes (objects), and sequences of message exchanges between the objects. The design process becomes one of identifying the objects in a system as aggregates of data and operations, rather than separately identifying data structures and procedures that use them (see [COX84] for more detail). The designer is naturally led to think in terms of building models or simulations rather than procedural solutions.

Classes may be instantiated in two ways: when used outside of a compiling definition, a class creates an object much as a defining word creates a "child". This is simply a Forth header (Name/Link fields), with a pointer to the owning class in place of its cfa field, and the data area for the class's named and indexed instance variables allocated immediately following the class pointer. The class contains a short sequence of assembly code at the location referenced by its objects, which allows objects to be executed as Forth words. When

used in this manner, an object returns the base address of its data. This prevents a dependency from existing between the Forth compiler and the object compiler, such as tick having to know where an object's data begins.

When the name of a class is used within the definition of another class (class compilation state), it does not fully instantiate itself as an object. Rather, it creates an instance variable dictionary entry in the class being compiled, so that when that class is instantiated, the other class will be instantiated as an instance variable within the newly created object. This nesting can be carried to any degree, which is an extremely powerful method for building data structures.

Within classes, inheritability is provided by constructing dictionaries very much like the Forth dictionary for instance variables and methods. Each class has its own Ivar and methods dictionaries, which are linked to those of its superclass (see Figure 1). Thus, when an instance variable or method is referenced within a method, the name is searched for through the superclass chain of the compiling class. This provides the strictly hierarchical inheritance capability of Smalltalk-80 and Simula.

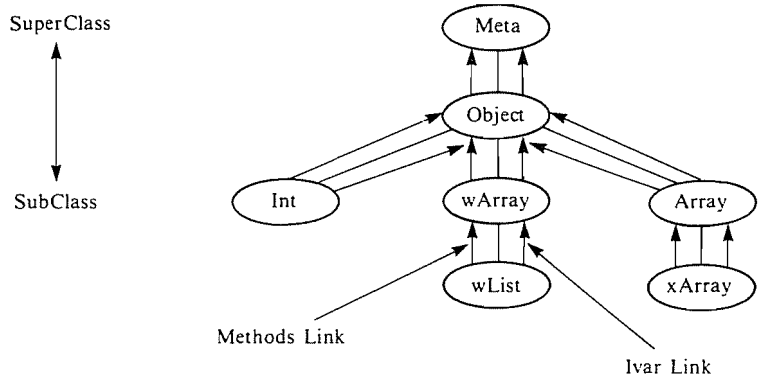

Figure 1. Methods and Instance Variable Links to Superclass.

Methods are referenced by using a selector before the name of an object or instance variable while compiling. All parameters to a method are assumed to be on the parameter stack at runtime. A selector is constructed with the defining word **SELECTOR**, and is immediate, so that its DOES> behavior actually occurs in compilation state. Selectors incorporate the entire interface between objects and the user, and are the means by which object and instance variable references are constructed in the dictionary. As previously mentioned, selectors will automatically perform early binding, unless preceded by the word **DEFER:**, which causes method lookup to be deferred until run time.

An Example

We will show some examples of the syntax in our class/object extensions, and then discuss our approach in implementing the various structures and their compilers. Figures 2–6 contain source for a series of classes that implement a simple word-game aid which assigns numerical values to letters and words.

Figure 2 shows the general form of a class definition. The word **:CLASS** is an alias for **:** and initiates compilation of a new class for **className**. The word **<SUPER** is used to indicate what superclass this class is to be derived from. If no particular class is to be

```
:CLASS className          ( begins compilation of and names a new class.)
   <SUPER superClass      ( identifies the superclass for this class )
   n <INDEXED             ( sets this class's indexed element width)

                        ( ** Instance Variable Definitions ** )
   10 Array Values        ( creates an Ivar named VALUES )

                        ( ** Define methods for this class ** )
   :M SSSS:               ( Start compilation of method for selector SSSS )
     <forth code>         ( Method code )
   ;M                     ( End compilation of this method )

   ...                    ( Repeat for other methods )

;CLASS                    ( end definition of this class )
```

Figure 2. Template for Class Construction.

inherited, class **Object** is used as the superclass. This provides certain general methods that are useful for any class. **<SUPER** builds the header for the new class, consisting of a normal Forth header plus some additional information (see Figure 7). This header is filled with header information from the superclass, thereby providing inheritance of its various attributes. The word **<INDEXED** accepts a value from the stack that determines the new class's indexed element width. This is automatically inherited from the superclass, but can be overridden if desired. If the indexed element width is 0, the class contains only named instance variables and has no indexed area.

A special class of Object Vectors can be created by using the **<Vectored** switch in the class definition. Object vectors are objects whose data elements are other objects. They can be defined as scalars, arrays, lists, or whatever else is desired, and operate by passing through deferred methods to the objects owned in their data areas. A complex structure such as an electronic mailbox could be implemented as an object list or array, passing through methods such as Print: to its owned elements. A composite picture could be built as an array of graphics objects, each with the capacity to draw itself in an appropriate manner.

Next follow the named instance variables that will define the private data for this class's objects. These instance variables are themselves created from existing classes, providing nested data structure definition; for example, class **POINT** contains two instance variables, X and Y, of class **INT** (integer variables). The methods within **POINT** can operate on either of the instance variables with all of the methods available to class **INT**. External objects are denied access to **POINT**'s Ivars except via methods that **POINT** explicitly defines to access them (localization).

Following the Instance Variable definitions are the methods for the class. Each method is delimited by **:M** and **;M**, and associates a body of Forth code with the selector immediately following **:M**. One of the design constraints in our system was the ability to freely intermix Forth code with object references. After the methods for the class are defined, the word **;CLASS** terminates this class definition.

Figure 3 shows the implementation of an integer variable class. This class has one instance variable, **Data**. The word **BYTES** is not itself a class, but a utility word to simply allocate private memory. It is used in the most primitive classes because other classes are not

```
( Define a class of 16-bit variable objects )
:CLASS   Int   <Super Object   ( Object is the root of all superclass chains)

      2 BYTES Data      ( Create a single Ivar that allocates 2 bytes of data)

( Define methods )
( — val )
  :M GET:    ^Base W@ ;M      ( Define fetch method — ^Base returns base addr of obj)

( val — )
  :M PUT:    ^Base W! ;M      ( Define store method )

( val — )
  :M +:      ^Base W+! ;M     ( Add n to contents of object )

          ( This method provides a right-to-left assignment, as in:   XX =: MyInt )
  :M =:   { addr -- }
          Get: Self addr W! ;M   ( Shows named input stack and use of Self)

;CLASS     (End definition of INT )
```

Figure 3. Definition of a Class of Integer Variable Objects.

yet available with which to build instance variables. In this case, **BYTES** is used to allocate the two bytes that comprise an **Int**'s data.

Int's methods follow, beginning with **GET:**, which returns the contents of the **Int**'s private data. The word **^BASE** leaves a pointer to the instantiated object's data area; Forth words can then be used to accomplish primitive operations on that data, such as fetches and stores. The =: method demonstrates the use of the named parameter stack, defining a single parameter, **addr**. The phrase **Get: Self** shows how a previously defined method can be used to access the object's own data. **Self** used as a message receiver always causes method lookup to begin within the currently defining class, and operates on the instantiated object's private data. **Super** can be used to initiate method lookup in the superclass rather than the defining class, to avoid resolving to a redefined method. **Self** and **Super** are implemented as pseudo-instance variables in class **Meta**, which is an artificially constructed class with **Object** as its sole subclass. When a new class definition is begun, the class pointers for **Self** and **Super** are pointed to the class being defined and its superclass, respectively, making them appear as instance variables for those classes.

Figure 4 details the construction of a class of 16-bit arrays, **wArray**. This class has no named instance variables; all of its data is allocated as indexed, variable-length, public data. The phrase **2 <INDEXED** sets the element width to 2, and the amount of data allocated will be determined at instantiation of the class. The methods **AT:** and **TO:** are defined for all indexed classes as the fetch and store primitives.

In Figure 5, the class **wList** is constructed as a subclass of **wArray**, inheriting **wArray**'s methods and indexed attributes. This example shows the creation of an instance variable, **Size**, of class **Int**. Names for instance variables need only be unique within the superclass chain, and can have the same names as Forth words without fear of collision.

Figure 6 shows **wList** instantiated to create an object, **Values**, that is 26 elements long. Various invocations of **wList** methods show runtime behavior of the class.

```
( Define a class of 16-bit array objects )

:CLASS  wArray  <Super Object

   2 <INDEXED  ( This class has 16-bit indexed data )
  ( wArray has no named Instance Variables )

  ( AT: and TO: are fetch and store selectors for indexed access )
  ( GET: and PUT: could be used, but this serves as a reminder that obj is indexed)

( ind -- val )
  :M AT:  ^Elem  W@ ;M  ( ^Elem returns address of element #n )

( val ind -- )
  :M TO:  ^Elem  W! ;M

( val ind -- )
  :M +TO:  ^Elem  W+! ;M

   ( Initialize all elements with a value )
  :M FILL: { val --} Limit: Self 0  ( Limit: is provided by Class Object )
           DO val I To: Self LOOP  ;M

;CLASS
```

Figure 4. Definition of a Class of Integer Array Objects.

```
( Define a subclass of wArray to solve word games )
( It will be an array holding number values for capital letters A-Z)
:CLASS   wList <Super wArray

      Int  Size  ( Ivar named Size — holds size of words to process)

( char — val )
  :M GET:  65 — At: Self ;M     ( leave value for char )

( Read next word in input stream, print its total value )
  :M SHOW: Bl Word 0 Get: Size 0 ( Shows a fetch of private Ivar)
           DO Here 1+ I + C@  ( Get next char in word )
              Get: Self + LOOP  ( Accumulate values on stack )
           ." The Value Is: " . ;M  ( Print total value )

( val — )
  :M SIZE:  Put: Size ;M  ( Set the value of private Ivar Size )

( redefine the FILL: method, which is inherited from superclass wArray )
( val0 val1 val2 ... valn -- )
  :M FILL:  Limit: Self 0
           DO Limit: Self I - 1+ To: Self
           LOOP ;M ( Fill array with values from stack )

;CLASS
```

Figure 5. Definition of a Subclass of wArray

```
( Instantiate wList ) OK
26 wList Values  ( creates a wList object named Values. 26 elements long) OK
( Load Values with a value for each character ) OK
2 3 7 8 4 1 8 0 4 5 2 6 8 4 3 7 8 5 6 2 1 5 9 6 7 3  Fill: Values OK
5 Size: Values  ( Tells Values to process words 5 characters long) OK
Show: Values  TRAMP
 Value Is: 24 OK
Show: Values  FRONT
 Value Is: 15 OK
```

Figure 6. Instantiation and Runtime Behavior of a wList.

Chronology of Class/Object Usage

We have identified 5 chronological phases in the construction of classes and objects and their use:

Sequence 0 — This is prehistoric from the user's point of view, and involves the definition of the specialized compiler extensions to Forth that make Class/Object structures possible.

Example: ::CLASS ... ... ;

Sequence 1 — (Definition Phase) Execution of **:Class ... ;Class** to define new classes that can inherit data and methods from other classes.

Example: :CLASS Int <Super Object ..

Sequence 2 — (Instantiation Phase) Execution of the defined classes to cause instantiation (object creation). Every defined class contains the cfa of an object builder (called **(BUILD)**) that executes when the class is executed. This code is capable of producing either an object or an instance variable dictionary entry, depending on whether a class is being currently compiled. In the case of public objects, they exist as normal words in the Forth dictionary with the exception that the cfa field contains a pointer to the object's class (see Figure 7). Because of this, objects cannot be executed directly, but must be preceded by a selector.

Example: Int Size

Sequence 3 — (Invocation Phase) Compilation of messages involving the created objects and selectors denoting their methods into either Forth words or other methods. Messages can also be executed directly from the Forth interpreter. The <DOES code for Selectors contains all of the logic for compiling messages in a variety of states.

Example: :.Size Get: Size . ;

Sequence 4 — (Run Phase) When executed, the compiled message causes the methods stack to be set up with the object's base address and length and any named stack parameters, and then the code for the correct method to be executed.

Example .Size 10 OK

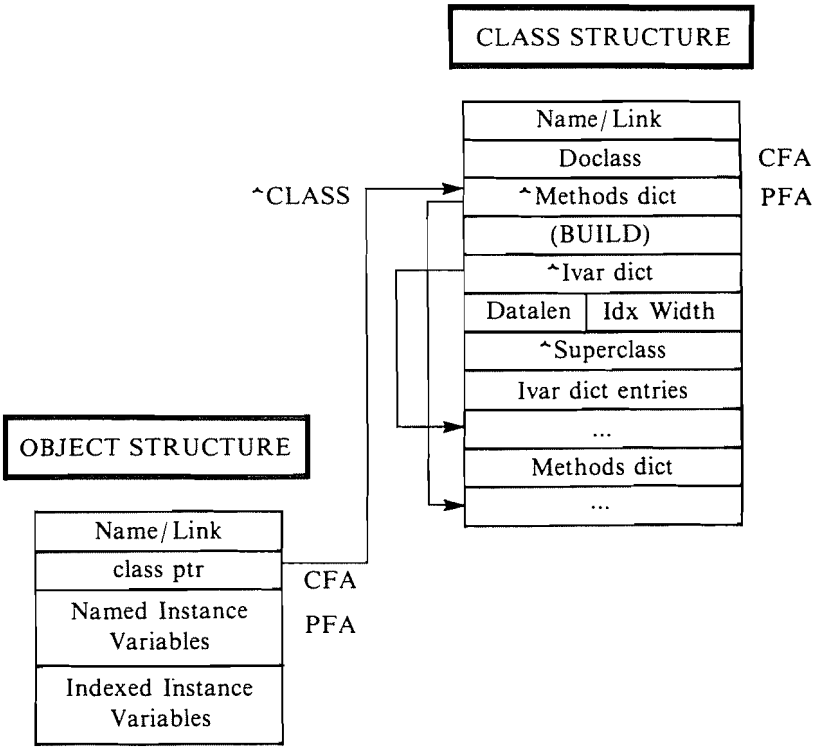

Figure 7. Data Organization in Classes and Objects.

Figure 7 describes the internal formats of objects and classes in the Forth dictionary. The cfa of a class points to the **DoClass** code, which causes the **(BUILD)** cfa to be executed and an object of Ivar thereby created. The Instance Variable and Methods Dictionaries are variable-length linked lists constructed in a similar manner to the Forth dictionary. Their entry formats are described in Figure 8.

Invocation Modes

A method is invoked by a statement of the form:

Selector: ObjectName

The selector pulls the objectname from the input stream, looks itself up in the object's superclass chain, and compiles an invocation for the method that was found. Methods can be invoked in a number of ways. Most of them can occur either in compile state or run state; SELECTOR is state-smart, making the difference transparent to the user. Object references can be divided into the following categories:

Type	:	:M	Run	Example	Comments
Public	X	X	X	**Get: MyObject**	Non-Ivar objects only
Private	—	X	—	**Get: MyIvar**	Ivars only
Class	X	X	X	**Get: wArray**	Requires addr on stack
Deferred	X	X	X	**DEFER Get:**	Requires object addr on stack

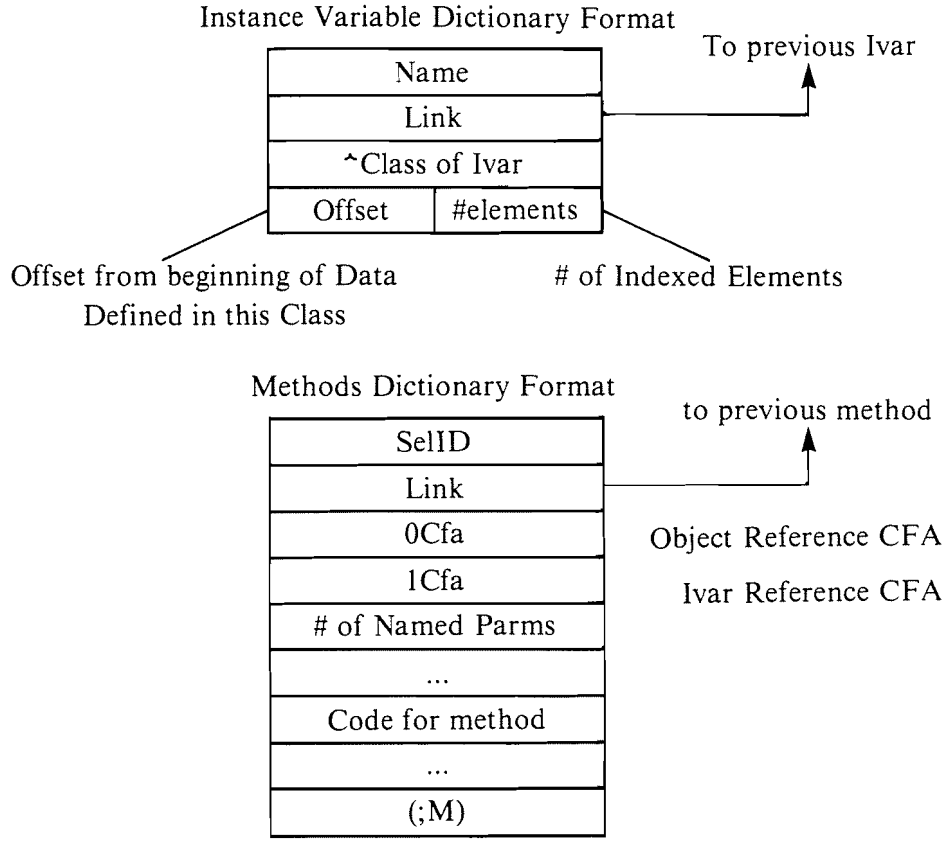

Figure 8. Methods and Instance Variable Dictionary Formats.

The public reference is compiled whenever a publicly defined (non-Ivar) object is used. It can occur within colon definitions, methods, or in run state. Private references are compiled for Instance Variable references only. They can only occur within methods. Class references are very powerful in that they allow *any* stacked memory address to be handled as an object, regardless of whether it was created as an object. Since the class of the object is specified in the reference, the selector knows where to look for the method, and doesn't have to find the class through the object. This type of reference requires that the address to be operated upon be on the stack at runtime. Class references were an important part of our design, because they allow us to use class methods on operating system data structures. On our original target system, the Macintosh, the sheer quantity of OS support made this consideration paramount.

Deferred references assume that the address of a true object will be on the stack at runtime, which is when final resolution of the method will occur within the object's superclass chain. This is also quite powerful, since it allows the calling code to be completely insensitive to the class of the object (assuming, of course, that the class has a method with the proper selector). For instance, a number of graphics objects could be kept in a list for display, and at runtime, the same selector would cause each to execute its own appropriate display method. The drawback of this method is the runtime overhead required to look up

the method. We have minimized this overhead by using a fixed-length, hashed internal representation for selectors.

```
: Display   Size: DisplayList 0
            DO   I   At: DisplayList DEFER: Draw:   LOOP ;
```

If it was known that each object in the list would be a rectangle, then the more efficient static-bound class reference could be used. This requires only that each element of **displayList** have 8 bytes of data that are organized as a rectangle's Points; it removes the requirement that each element actually be a Rect object (has a pointer to class Rect in its CFA):

```
: Display   Size: DisplayList 0
            DO   I   At: DisplayList   Draw: Rect      LOOP ;
```

The Methods Stack

We found that, in order to easily implement nested data structures and nested object references, a third stack would be of great help. Since we implemented it, the methods stack has proven so useful that we're not sure how we did without it before. As a fringe benefit, it expedited our addition of named input parameters to Forth, a feature that greatly improves readability. This facility allows the programmer to assign local, temporary names to up to 6 cells on the parameter stack, and then reference their values from within a method or Forth word:

```
:M SEARCH: {  address len argument \ bool --- pos t   OR   f}
      False —> bool   len 0
      DO address I + C@  argument =
        IF I True —> bool   Leave  THEN
      LOOP bool ;
```

In the above example, **address**, **len** and **argument** are named parameters that leave their values when referenced within the word or method. **Bool** is a local variable, existing only within the scope of the Search: method. Nothing is left on the stack unless the programmer explicitly puts something there, such as **I**. The top stack cell can be assigned to a local variable or input parameter with the —> operator. This example has eliminated all stack manipulation operations from the code, leaving a relatively direct translation of the algorithm. Compare this with the word salad that occurs in a standard Forth implementation:

```
: SEARCH   ( addr len arg --- pos t OR f)
    SWAP >R false rot rot R> 0
    DO over I + C@ over =
      IF Rot drop I true 2 swap leave
      THEN
    LOOP 2 drop ;
```

An incisive validation of our point occurred in the review process for this paper. Our original coding of the second routine was wrong, because we forgot that **addr** and **arg** were left on the stack. The reviewer noted this, but his proposed solution was also wrong! The fact that all of us are experienced Forth programmers points out that stack management errors are a permanent affliction, regardless of experience. Readability and therefore maintainability can be greatly improved through use of named parameters and local variables, because the programmer need not go through the exercise of recreating the stack layout each time the code is read. In our implementation, examples such as the one above have benchmarked faster than the equivalent Forth code, due to the fact that several stack words were eliminated.

Runtime Actions at Invocation

The other chief use to which the methods stack is put is in addressing object references at runtime. When a method is executed, it receives the base address of its object's data on the top of the methods stack. The second element is always the length of the object's named instance variable area, which is equal to the offset of its indexed area from the PFA of the object (see Figure 7). This allows any object to address both its named and its indexed area within a single method very efficiently.

When a method is compiled, it is given two code fields that are designed to handle public and private references. The compilation logic in SELECTOR is responsible for deciding which CFA will be compiled into the dictionary for a given reference. Figure 9 shows how the invocation of a method is compiled for each type of object reference.

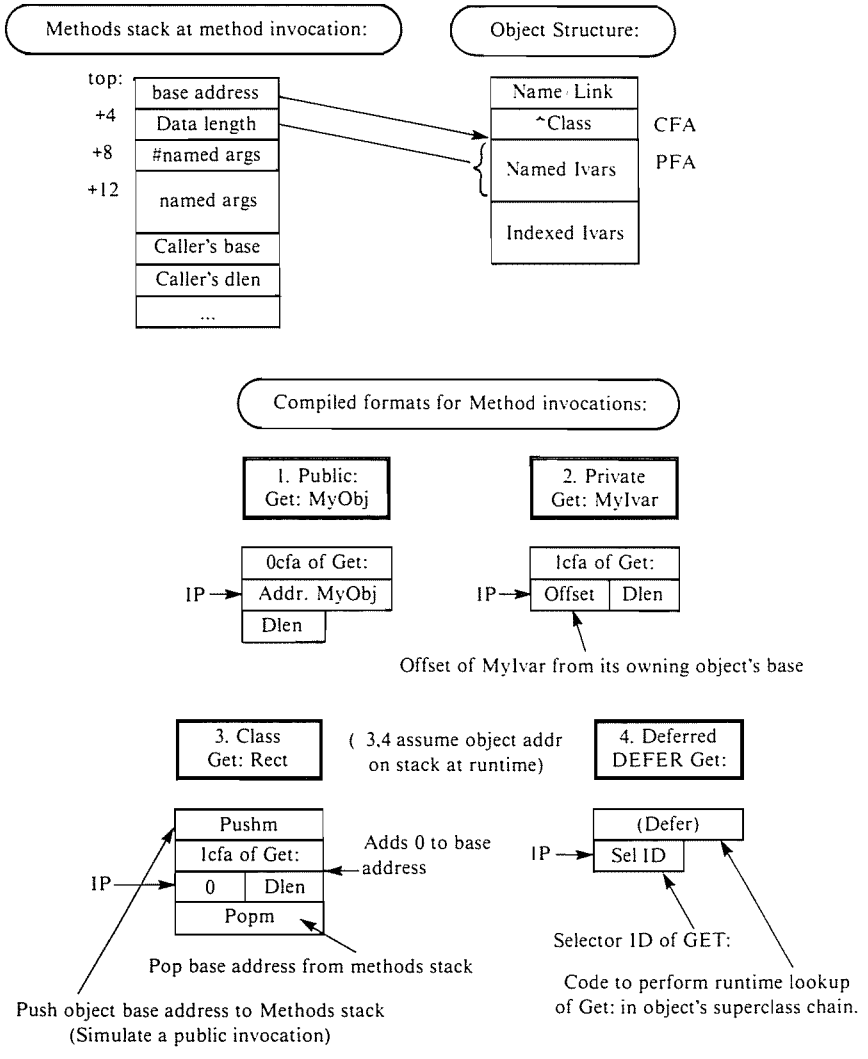

Figure 9. How the various invocations are compiled.

The 0CFA code always assumes that the base address of the object and its named data length are located at (IP) when the method is entered (this assumes a post-incrementing IP). 0CFA simply has to push the data length and the base address to the methods stack before executing a normal colon nest. The 1CFA code is designed for relative references, in which an offset is at (IP) rather than an absolute address. This allows classes to be easily nested, since each need only record its instance variables as offsets within its data area. 1CFA adds the offset at (IP) to the previous address on the methods stack, which was put there by a 0CFA in a public reference to the object, and stacks the data length and the new base address. In the case of a CLASS reference, the object's base address is on the parameter stack at runtime, and cannot be compiled in. PUSHM is compiled in first to move the base address to the mstack, and then the 1CFA and an offset of 0 causes the base address to appear on the top of the mstack when the method is executed. POPM cleans up the extra base address pushed by the initial PUSHM. This rather elaborate arrangement saves having to provide a third cfa for each method to handle occasional class references; that approach could be taken if it were desirable to trade space for speed, or if class references were to be a high-frequency item in an application.

Indexed Methods

Within a method, the current object's base address is always available on the top of the mstack, and can be placed on the parameter stack by COPYM or its alias **^BASE**. Indexed objects have a variable-length indexed area appended to the name Ivar area (see Figure 7). Indexed classes require a value on the stack for the number of indexed elements at instantiation time, as in:

```
10 Array Temperatures
```

which creates an array 10 elements long, each of which is 4 bytes wide. The element width is set when the class is compiled by passing an argument to the word **<INDEXED**. When the object is created, its indexed area is preceded by a 4-byte header that contains the width of each element and the number of elements allocated. This header can be located in an indexed class by adding the top two elements on the methods stack, which are the base address and the length of the named Ivar area. The word **^IXData** leaves this address on the parameter stack. Another word, **^ELEM**, is available to compute an element's address given its index. Because **^ELEM** uses the header, it can be used for any element width, and can optionally do range checking on the index. It could be tailored for a particular element width in a time-critical application.

Conclusions

We have used this system to build several applications, and experienced what we felt to be a significant productivity increase in those projects. The system that we have described succeeds in meeting the criteria outlined at the beginning, which leads us to believe that those criteria support a more effective programming process. Some of the effects that we observed are:

Because of the availability of generic operators and increased structuring of source, we have found it much easier to read another programmer's code and immediately understand its function. For instance, a **GET:** is always a **GET:**, whether the subject is a rectangle or an integer.

There is less inclination to extend the language in bizarre ways (although that capability remains), because powerful objects can be built using existing pieces. As classes are added, the number of methods that can be reused increases dramatically, and further reduces the necessity of reinventing the wheel. This effect is self-supportive, because programmers are led

to make more of an investment in the future when they experience the benefits of reuse so dramatically. [WEG84] provides an excellent discussion of reusability in building large systems.

While Forth untouched greatly encourages modular design, this system pushes the designer further into isolating the active agents or "players" in a problem, and building the application as a model of the real-world system (if there is one). Object-oriented systems are well-suited to creating authentic simulations of real-world behavior, due to the consolidation of data with the access methods relevant to it. Applications that we have built reflect this, and tend to read like a schematic drawing of the system. This close mapping of the problem and computer domains has many benefits for the designer as well as the maintenance programmer.

We found that the syntax of this language tends to fall into distinct non-RPN phrases, which tend to be easier to unravel than the strict RPN of unadulterated Forth. For instance, the sequence

```
Get: X  Get: Y  Put: NewPoint
```

as opposed to

```
X @ NewPoint ! Y @ NewPoint 2+ !
```

has much more meaning to the non-acolyte, and we have found that it makes quick scans easier for even the experienced Forth programmer. Previous work with QUAN and VALUE has demonstrated the benefits of this prefix notation for readability [SCH83], [DOW83]. Neon includes two predefined multiple-codefield structures, VALUE and VECT, that do the work of Forth's variables and execution variables in a more efficient manner.

The named parameter stack with local variables has a profound effect on the ease of construction and subsequent readability of any code that must manipulate several stack cells, particularly in loops. Most stack manipulation words can be eliminated, with the exception of an occasional DUP.

Judicious use of deferred binding results in code that is more easily maintained, because other objects are completely isolated from an object's implementation of its methods. Earlier methods can be modified without concern for updating later code that might depend upon specific behavior (see [COX84]).

We have casually observed that the high degree of reuse afforded by inheritance of methods seems to result in smaller source and object than that produced with conventional Forth. This effect is directly related to the complexity of the application, as reuse increases in large applications.

Using these extensions definitely adds a degree of runtime overhead, depending greatly upon how extensively features like deferred binding are used. In some applications, this could conceivably be a problem, but the option of using assembly code for time-critical sections remains. To help quantify the issue, we ran some benchmarks comparing object execution time to standard Forth and Neon's extended Forth (Figs. 10-11).

The first benchmark compares a simple 4-byte variable object, VAR, to a Forth VARIABLE and a Neon VALUE (multiple-cfa variable — see [SCH83]). A fetch operation was performed 10,000 times on each type of structure. It can be seen that the VAR loses about 50% in runtime to the other two. This is attributable to the fact that VAR's GET: method is not coded directly in assembler, but must nest to a code word. Also, a nested method call is more expensive than a nested Forth call due to the extra work of setting up the methods stack. VALUE runs at least 10% faster than its VARIABLE equivalent because only one cfa must be executed versus two.

The next example compares execution of the Eratosthenes' Sieve algorithm using standard Forth, Neon using named stack parameters, and two implementations using

objects. The first object implementation uses the AT: and TO: methods for array access, just as a developer would in prototyping an application. The second implementation removes the nest required by the reference to the AT: and TO: methods in the superclass, and uses the AT1 and TO1 primitves directly. These primitives are optimized for 1-byte array access, and can easily be substituted by the developer as an optimization step. The data indicate an overhead of 25% for the non-optimized class, while the optimized class performed the sieve about 10% faster than its Forth equivalent. Clearly, array access is the primary factor in this example.

Figure 10. Source for NEON benchmarks.

```
\ Benchmarks for NEON - CBD 11/02/84

\ Standard Byte Sieve of Eratosthenes : 3.86 Secs
8190 Constant Size
0 Variable Flags Size Allot

: Do-Prime Flags Size 1 Fill 0 Size 0
     DO Flags I + C@
       IF I Dup + 3 + Dup I +
          BEGIN Dup Size <
          WHILE 0 Over Flags + C! Over +
          REPEAT 2drop 1+
        THEN
     LOOP . ." Primes " cr ;


\ using local variables : 3.99 secs
: do-prime .n {\ lo hi -- }
    flags size 1 fill
    0 size 0
    DO flags I + c@
    IF I 2* 3 + Dup —> lo I + —> hi
     BEGIN hi size <
       WHILE 0 hi flags + c! lo ++> hi
       REPEAT 1+
      THEN
      LOOP . ." Primes " CR ;

selector prime:
selector bench:

\ Class example using inherited methods : 5.09 secs
\ First define superclass byte array

:CLASS barray >Super Object 1 >Indexed

  :M FILL: {val --} idxbase limit val fill ;M
  :M AT: at1 ;M
  :M TO: to1 ;M
;CLASS
```

Figure 10 continued.

```
:CLASS Sieve >Super Barray

  :M PRIME: {\ lo hi -- }
      1 fill: self
      0 size 0
      DO I at: Self
      IF I 2* 3 + Dup —> lo I + —> hi
       BEGIN hi Size >
         WHILE 0 hi to: Self   lo++ > hi
         REPEAT 1+
      THEN
      LOOP . ." Primes " CR ;M

;CLASS

\ Class example using optimized primitives : 3.55 secs
:CLASS FSieve >Super Object
  :M PRIME: {\ lo hi -- }
      1 fill: self
      0 size 0
      DO I stl
      IF I 2* 3 + Dup —> lo I + —> hi
       BEGIN hi Size >
         WHILE 0 hi to'l   lo++ > hi
         REPEAT 1+
      THEN
      LOOP . ." Primes " CR ;M

;CLASS
```

Figure 11. Benchmark Results.

Eratosthenes' Sieve

Routine	Seconds	Factor
Standard Forth	3.86	1.00
Named parameters	3.99	1.03
Standard class	5.09	1.31
Optimized class	3.55	0.92
	Fetch * 10,000	
Value	0.91	0.88
Variable @	1.03	1.00
Get: Var	1.50	1.45
Var Defer: Get:	12.80	12.42

These benchmarks are included to show the raw overhead of method nesting versus very lean Forth code. In complex problems with more sophisticated data structures, Forth solutions would be harder to optimize, and therefore an object-based implementation will compare more favorably in these cases. Methods code is as easy to optimize as is Forth, as demonstrated by the array example. One further step would be to allow direct implementation of methods in assembly code, a feature that we are likely to provide in a future version.

In our experience, the advantages relating to production time have far outweighed any drawbacks in performance, and this will no doubt be the case for many developers. Writing traditional Forth code requires substantial effort to ensure the integrity of the parameter stack, the proper application of operators to data structures, and definition of new data structures via defining words. Often, the source code bears little resemblance to the nature of the problem to be solved, and much time can be spent unraveling layers of artificial constructs instead of understanding the problem more fully.

Our experience has been that the use of the class/object paradigm and named parameter stack can greatly reduce the implementation effort required in a project. Once the initial analysis of a problem has been clearly stated, much of the work towards implementation has already been done. The construction of classes that embody the solution proceeds directly from a clear understanding of the problem. And because the foundation of this system is in Forth, the way is open for further refinement and tuning as more direct experience is gained.

Acknowledgements

We would like to thank the guest editor and the referees for their very useful suggestions with respect to this paper and the work that it describes.

References

[LEA83] R. Leary and D. McClimans, "Message Passing with Queues", *Journal of Forth Application & Research* Vol. 1 No. 2, 1983.

[SCH83] K. Schliesiek, "Multiple Code Field Data Types and Prefix Operators", *Journal of Forth Application & Research* Vol. 1 No. 2, 1983.

[ROS82] E. Rosen, "High Speed, Low Memory Consumption Structures", *FORML*, 1982.

[DOW83] T. Dowling, "The QUAN Concept Expanded", *Rochester Forth Applications Conference*, 1983.

[MYE78] G. Meyers, *Composite Structured Design*, Van Nostrand Reinhold, 1978.

[COX84] Cox, "Message/Object Programming", *IEEE Software* Vol. 1 No. 1, Jan. 1984.

[GOL83] A. Goldberg, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[KRA83] G. Krasner, *Small-80: Bits of History, Words of Advice*, Addison-Wesley, 1983.

[BYT81] *Byte Magazine*, Smalltalk Issue, August, 1981.

[KAY77] Alan Kay, "Microelectronics and the Personal Computer", *Scientific American*, Sept. 1977.

[WEG84] P. Wegner, "Capital-Intensive Software Technology", *IEEE Software*, Vol. 1 No. 3, July, 1984.

[FOR81] L. Forsley and G. Cholmondeley, "A Forth-Based Relational Data Language", *Rochester Forth Standards Conference*, 1981.

[GLAS83] H. Glass, "Towards a More Writable Forth Syntax", *Rochester Forth Application Conference*, 1983.

[LUO84] K. Luoto, "Procedural Arguments", *Forth Dimensions*, Vol. 6 No.2

[PER82] M. Perkel, "Turning the Stack Into Local Variables", *Forth Dimensions*, Vol. 5, No. 6.

[BOW82] S. Bowhill, "Fast Local Variables for Forth", *FORML Proceedings*, 1980.

[GLAS84] H. Glass, "A Threaded Interpretive System as the Kernal of a Functional Programming Environment", unpublished paper.

Mr. Duff received a B.A. from the University of Chicago. He has done graduate work in information engineering at the University of Illinois. Currently he is software manager at Kriya Systems, Inc. developing micro-computer products in Forth. He has also written a book for McGraw-Hill about the Apple MacIntosh. He is engaged in research regarding the extension of Forth to create new languages that are more suited for such tasks as simulation and artificial intelligence.

Mr. Iverson studied math at the University of Illinois. Currently he is senior software engineer at Kriya Systems, Inc. In addition to his interest in Forth, he has a background in statistical analysis and financial systems.

Manuscript received June 1984.