BORON- Yet Another Object Oriented Forth

Steven M. Lewis
Dept. of Biomedical Engineering
U. Southern. Cal.
Los Angeles CA 90089

ABSTRACT

A system of object oriented extensions to FORTH are described. Objects support inheritance, subclasses, instance variables, both early and late binding and may be intermingled with standard FORTH code. In the absence of specific messages, objects invoke default operations. A uniform means of handling structures of multiple objects is also described. Early binding generates code similar to FORTH code with little run time penalties.

Introduction

Object oriented programming differs from conventional programming in placing knowledge about methods for manipulatiing data structures within the structures themselves. Methods are selected by sending a message to the structure. The process generates more readable code and places a lower burden on the programmer.

A significant advantage of object oriented programming come when the same message evokes similar behaviors from a number of different objects. For example, consider a collection of FORTH VARIABLES, FVARIABLES, STRINGS, LISTS and other structures. To print the contents of each, one needs to write: ?; F@ F. ; COUNT TYPE ... , a separate code for each object. As OBJECTS the message PRINT: will cause all types of objects to print themselves.

A number of object oriented versions of FORTH have been described ([Duff 83],[Duff 86], [Click 86],[Pountain 86]). At least one commercial system, NEON™ (Krya Systems) is a version of FORTH with extensive object extensions. BORON is generic FORTH code for OBJECTS which, while retaining compatability with other systems, gives the programmer substantial control over the operation of objects.

As with all object oriented systems, BORON introduces three new data types: **CLASSES**, **INSTANCES** and **MESSAGES**. A CLASS is a defining word used for creating INSTANCES. Classes contain operations for building INSTANCES together with a list of all allowable operations on an instance of that class. Each possible operation is selected by a MESSAGE. Each INSTANCE is a state sensitive, immediate word. On execution the INSTANCE reads the message, looks up the operation and, in immediate mode, executes the operation. In compile mode the operation is compiled. I will use the word 'invokes' for these two operations.

Major decisions in the design of an object oriented system relate to the operation of INSTANCES and MESSAGES, the operation of objects when no message is received and whether interpretation of messages is done at compile time (early binding) or run time (late

binding). An additional consideration is knowledge of the structure of the specific FORTH implementation used. I attempted to write a generic system with minimal assumptions about the FORTH interpreter, vocabulary structure or form of the compiled code. This sacrifices elegance for portability. All system dependencies are handled by defining a few words which operate in a system dependent manner at the beginning of the system.

In the absence of a message in the current class, the address of the object is passed to the **PARENT** class for interpretation. In most standard object oriented systems, the unmodified address is passed to the PARENT. I felt that this was unnecessarily restrictive. While passing the object address is the default operation for a class, each class has the ability to modify the data before passing it to the PARENT. For example, a POINTER to an ARRAY invokes @ to get the address of the array before passing the message to ARRAY for interpretation.

Messages are associated with a CFA to invoke. This may be a single FORTH word (ADD-ACTION), a generated, headerless word (:MT message .... MT;) or a section of code which saves the base address on a stack for availability to all objects in the message (:M message .... M;) The first two methods place the object address on the parameter stack and operate with no run time overhead.

EXAMPLE 1

```
( Create class VALUE                                          )
( Value is used like variable of the system - default @ )
CLASS: VALUE  <PARENT SUPER-CLASS
  <DEFAULT @              ( default action is to @ data          )
  -> ADD-ACTION !         ( message -> causes ! a single word    )
  :MT ++ 1 SWAP +! MT;    ( make headerless word for message ++  )
  :M ++@ ++ SELF SELF M; ( SELF is current object, @ is default )
  . . .                   ( more actions                         )
;CLASS                    ( end of class definition              )

( Create a class VALUE[], an indexed VALUE, messages not          )
( related to the indexing are passed  to VALUE with element addr )
CLASS: VALUE[] <PARENT VALUE
   <CREATE-ACT CREATE-INDEX ( indexed variable )
   <DEFAULT INDEX ( index,base -- addr get addr of indexed elem)
   ( Add index methods here .....                    )
;CLASS                     ( end of class definition     )

10 VALUE[] MYDATA[]  ( create a 10 element array. I use [] in the
                       name as a reminder that this is an array )

: TEST  -> 5 MYDATA[]  3 MYDATA[] ;
compiles to:
   lit 5
    MYDATA[]     ( base address                                )
    INDEX        ( array default message -> is for VALUE )
    !            ( action for message -> in VALUE         )
    lit 3
    MYDATA[] INDEX ( same as above )
    @              ( default for VALUE since there is    )
                   ( no message, default actions compile.)
```

The system provides for the possibility of decoding messages at run time. Because of the time penalties involved, this is not recommended unless the message and/or the class of the instance are not available at compile time.

Users have access to all elements of the class structure. The parent, default action, creation action, methods and structure of a class may all be altered by the user. Three relations are possible for classes: **SUPER, PARENT** and **SIBLING.** Super creates an identical copy of the parent class without methods. Additional methods and data may be added. PARENT simply declares the destination of unresolved messages. SIBLING creates an identical copy of a class structure, usually as a prelude to modification. For example, if STACK is a class supporting stack operations (push,pop,init,^top where the later gets the address of the top element), then:

```
:CLASS VSTACK <SIBLING STACK <PARENT VALUE <DEFAULT ^TOP
;CLASS
```

will create a class accessing all stack operations but in the absence of a stack operation, supporting all VALUE operations on the top element.

All structures consisting of collections of similar elements, arrays, stacks, lists amd queues share a set of methods with similar effects, although the details are quite different from one class to another. START: initializes an internal pointer to point to the first element. It returns true unless the structure is empty. CURRENT: read the last message on the stack and apply it to the currently addressed element. NEXT: advance the pointer to the next element. NEXT: fails if either there are no further elements or the word BREAK is used. Failure initializes the internal pointer. BREAK stores the address of the instance addressed in CURRENT-VALUE.

```
EXAMPLE 3
40 STACK S1  ( create a stack, S1                                    )
( NOTE if S1 is an array, list or queue the same code works          )
: SUMS1 ( <>--n sum all elements in stack S1 )
    0   START: S1          ( initialize offset                      )
    BEGIN   WHILE GET: CURRENT: S1 ( get current data                )
       + NEXT:             ( -- f true if next is valid element )
   REPEAT ;

( A more complex use, here the operation and the structure   are )
both arguments. Thus, 0 ' [+] S1 MAPSTACK is the same as  SUMS1  )
*STACK  *SS  ( create a  pointer to a stack                  )

: MAPSTACK ( operation, ^stack -- <> apply the operation to
    all elements on the stack )
   P-> *SS              ( store stack address in *SS, P-> is )
                        ( a  message to a pointer to store   )
   >R                   ( operation to return stack          )
   START: *SS           ( <>-- f  start a structural access  )
   BEGIN   WHILE GET: CURRENT: *SS ( fetch current)
        R@ execute      ( perform that operation on it       )
        NEXT: *SS       ( -- f true if next is valid element )
   REPEAT R> DROP ;
```

Implementation

Messages are immediate words which place their address on the message stack. All INSTANCES begin by compiling their address as a literal. The current message is then read. If the message is in a list within the class called the action list, the associated action is fetched and compiled. If not, the default action is compiled and the message is passed to the PARENT class and the process is repeated. In the absence of a message, a default message is sent which always returns a NOOP. Uninterpretable messages cause an abort.

In addition to existing object oriented systems, other approaches offer much of the flavor of object oriented programming. The TO concept ([DOW 83]) stores methods as multiple CFAs in each instance. This approach wastes memory when the number of possible methods is large compared with the alternative of placing methods for a class of objects in a single location and simply pointing each object to that location. Earlier, [Lew 85] I proposed an extension to the TO concept placing responsibility for invocation of the correct code in a an immediate, state sensitive object. This paper describes how that concept can be extended to a full object oriented system.

[CLI 86]Click, C and P. Snow, Object oriented programming in FIFTH, JFAR 4:199-202, 1986.

[DUFF 84]Duff C.B., A group construct for field words, JFAR 2:83-88,1984.

[DUFF 86]Duff, C.B., Development of a threaded, object oriented language, JFAR 4:133-154, 1986.

[DOW 83]Dowling, T, The QUAN concept expanded, JFAR 1:69-72, 1983.

[LEW 85]Lewis, S.M. Should variable be an immediate, state sensitive word, JFAR 3:53-62, 1985.

[POU 86]Pountain, D. Object-oriented FORTH, Byte 11:227-233, 1986.