Forth Dimensions is the magazine of the Forth Interest Group
P.O. Box 2154, Oakland, California
94621 USA

The author can be accessed by e-mail h9290246@hkuxa.hku.hk.
(This address is valid only before the end of June 1996.)

The program can be downloaded from ftp://ftp.taygeta.com/pub/Forth/Reviewed/oof.zip. Watch out for newer versions.

An Assembly Programer's Approach to Object-oriented Forth

András Zsótér
Department of Chemistry
The University of Hong Kong
Pokfulam Rd., Hong Kong

Introduction

Some people use Forth as a computer language while other people -- including myself -- use it as a computer program for controlling their machine and other pieces of hardware. When I had to decide what to use in our project (a kind of laboratory automatation with robotics) I did not search for a computer language (I feel easy enough with Assembly and Pascal to do the job) but I searched for a system which gives me freedom to do whatever I want. I also preferred an interactive program to a compiler generating stand-alone applications. Naturally the solution is a Forth system. On the other hand during my previous pieces of work I used the OOP facilities of *Turbo Pascal* and I felt I would miss it if I had to use a language without it. As an obsessed Assembly programmer I decided to implement a version of Forth for myself and I ended up with a system which is very convenient to use if one wants to be sure all the time what is going on in it. Basically I shaped my Forth system after the old-fashioned FIG-Forth (1) with some modifications. Because the program runs on a 486 machine the most natural solution was to use its 32-bit protected mode. This way I do not have to worry about running out of address space. Also protected mode really means some protection against accidental mistakes and their consequences (2).

Definition of Compile Time Behaviours

Because I am very much concerned about the speed of my program I decided to generate native code. The technique I used for code generation is sometimes mentioned in the Forth litterature as nano-compiling [1]. While older Forth compilers used CFA to store the address of a machine code subroutine to be called when a word is being executed, I used an addtional one (CCFA=Compile Time Code Field Address) to store the address of the subroutine to be called when a word is being compiled. The user of the program can explicitely define the compile time action of a non-immediate word by using the words **C:** and **;C** (3). For example one might want to implement SWAP! in the following way:

```
: SWAP! ( Addr Data -- ) SWAP ! ; C: POSTPONE SWAP POSTPONE ! ;C
```

Alternatively more optimized machine code can be generated if someone has a more intimate knowledge about the system. Immediate words have the same routine for compile time and run time behaviour. This way the user has control over the code generation and one can specify explicitely which routines are to be inlined or to be substituted by more adequat machine instructions and which are to be left alone and treated as ordinary subroutines. One might complain that the old-fashioned state smart words can do the same and the previous example might have been coded as:

```
: SWAP! ( Addr Data -- )
    STATE @ IF  POSTPONE SWAP POSTPONE ! ELSE SWAP ! THEN ; IMMEDIATE
```

This is true as long as words such as **COMPILE** , **[COMPILE]** , **COMPILE,** and **POSTPONE** do not mess up everything (4). With the use of CCFA (5) the definitions of compiling words became almost trivial. **POSTPONE** *Name* generates a call to the CCFA routine of *Name*. **COMPILE,** interprets the top item on the stack as a CFA (``execution token'') and if the word is immediate (its CFA and CCFA point to the same address) **COMPILE,** generates a call to that address. If the word is not immediate **COMPILE,** simply calls the routine pointed to by CCFA (ie. compiles the runtime behaviour of the word into the new definition). **[COMPILE]** and **COMPILE** can be defined the following way:

```
: [COMPILE] ( -- ) '  COMPILE, ; IMMEDIATE

: COMPILE   ( -- ) ' LITERAL POSTPONE COMPILE, ; IMMEDIATE
```

One Word with Multiple Names

Forth programmers tend to ``factor out'' similar pieces of code in their programs. As the Forth language grew bigger and bigger, pieces of code appeared with the same effect under different names. If one uses words implemented by different programmers it is good sometimes to have all the usual names ready. The most commom Forth solution is the definition of the new name in the form:

```
: NewWord OldWord ;
```

The above solutution is ususally satisfactory, however if one wants *NewWord* to behave **exactly** as *OldWord* a more sophisticated definition is needed:

```
: NewWord OldWord ; C: POSTPONE OldWord ;C
```

If *OldWord* is immediate the definition is different:

```
: NewWord POSTPONE OldWord ; IMMEDIATE
```

To avoid all this trouble a new definition word **Alias (CFA --)** is provided. So from now on the above definition would be:

```
' OldWord Alias NewWord
```

This definition will work regardless of the immediacy of *OldWord*. At first sight this facility does not seem to be of much help but because of the OOP facilities, it is sometimes necessary to have a name ready in multiple vocabularies. Also Forth programmers name their words on a pragmatic basis (what the word is used for) and not on a semantic one (what is the effect of the word). If a piece of code has multiple usage the use of aliases can greatly increase the readability of the program.

What is an Object?

One of the most powerful features of my Forth implementation is that it is object-oriented. There are several problems with this utterance. Some people even argue that Forth in itself is a object-oriented language because of the **CREATE .... DOES>** capabilities. In my oppinion a real OOP is more sophisticated than that and polymorphism, inheritance and virtual methods (6) are necessary in a system to qualify it as an OOP language. In my interpretation an object is an entity which consist of data (residing at least partially in memory) and a set of methods to manipulate the data. I always considered an object to be quite independent from its environment. In order to make the latter explicit I introduced the idea of the active object. Only one object can be active at a time. The system has a pointer to the active object (7) which can be manipulated via the following words:

```
O!   ( Object --   ) Select object = Write the object pointer.
O@   ( -- Object   ) Query object  = Read the object pointer.
O>R  (R: -- Object ) Save object pointer to the return stack.
R>O  (R: Object -- ) Retrieve saved object pointer from the stack.
{    ( Object1 --  ) (R: -- Object2) This word combines the
                                     functionality of O>R and O! .
}    (R: Object -- ) The same as R>O .
```

As the term ``active'' already implies that an object is considered to be an ``animate'' entity, this means individual behaviour is attributed to each and every object instance, a reason why OOP is called ``programming in the active voice'' [2].

An object ``can access'' the application's memory in two ways. The first way is the same as we normally address the memory and the second one is when all addresses are relative to the object's base address. At first sight this latter way seems to make sense only for the fields of an object (so that they are represented as ``offsets'' from the starting address of the object) but it can be useful also for addressing other entities outside the object (8). In order to make the data stored in an object accessible for the traditional Forth memory operations such as **@** and **!** a new word is introduced. This new word is **^ (RelAddr -- AbsAddr)**. As it is clear from the stack effect comment this word transforms an address relative to the base of the active object to an absolute address. For completeness I added the reverse operation **-^ (AbsAddr -- RelAddr)** (a better notation would be a downward pointing arrow but that is not included in the common character sets). Using the **^** notation the fields of an object can be represented as offsets from the base address of the object. This is the less sophisticated way of accessing data in an object. In order to find out more about the behaviour of the objects we must take a look at their classes. While an object instance is an individual chunk of data with a set of methods to manipulate it, a class is a set of objects which share the same set of methods.

Vocabulary and Class hierarchy

In order to implement OOP I chose an old fashioned FIG-FORTH style vocabulary structure. I also kept the old system variables **CONTEXT** and **CURRENT** to hold the address of the search and definition vocabularies. I have defined the rules of searching so that not only the **CONTEXT** vocabulary is searched but if a name cannot be found in the **CONTEXT** vocabulary the search goes on with its ancestors (9). A class is a special vocabulary with late binding support. This means a class has a VMT (10) which contains the addresses of the virtual methods belonging to the class. There is one class called **Objects** which is the root of the object hierarchy or in other words the common ancestor of all classes. A new child class can be defined by using the word **Class (NewMethods --)**. For example the following line will define a new class *ClassA*:

```
4 Class ClassA
```

The header of *ClassA* will contain a VMT with four more entries than the parent class of *ClassA*. The VMT of the parent class will be copied to the child's VMT, thus the virtual methods will be inherited by default.

Methods

New method names can be assigned to the new entries in the VMT by using the definition word **Method**. The line below will define a word *NewMethod1* in the **CURRENT** class (remember that a class is just a special kind of vocabulary):

```
Method NewMethod1
```

The index of the first undefined entry in the VMT of the **CURRENT** class will be assigned to the method. Notice that the definition of the name of a new method does not specify the action performed by the method. The latter must be defined later by using the word **use: (MethodIndex --)**. The index corresponding to a method's name can be obtained by using the word **As (-- MethodIndex)**. So the definition of a method's body will look like:

```
As NewMethod1 use:  ;M
```

This even looks like an English sentence thus this notation makes the source more readable. Any methodname visible from the **CURRENT** class can be used. In this way not only the new methods can be defined but the old ones defined in the ancestor classes can also be re-defined.

One further advantage of this approach is that the compilation is entirely incremental. Method names and method bodies can be defined in any order. When a methodname appears in a definition the following rules determine what code will be generated for it:
-- If the **CONTEXT** class is an ancestor of the **CURRENT** one a ``static'' call is generated which means the binding is done at compilation time. The effect of this behaviour is similar to *Turbo Pascal*'s **AnAncestor.AMethod;** type of statement. If a method is mentioned not only by having its name specified but by having its type and name specified together (this makes sense only in a method of a successor type) then that method no longer identifies a ``series of routines'' but only one routine which is known at compile time.
-- If **CONTEXT** and **CURRENT** are the same or belong to different hierarchy the emitted code uses *late binding* which means a call to a routine with a certain index in the VMT of the currently active object is generated. Here is an example:

```
1 Class ClassA ClassA DEFINITIONS
Method M1 ( ???? )   ( The stack effect should be recorded here. )
As M1 use: <....> ;M ( Some action. )
1 Class ClassB ClassB DEFINITIONS
Method M2 ( ???? )
As M1 use: <.....> ;M
As M2 use: ClassA M1 ( Call the method M1 of the ClassA. )
           ClassB M1 ( Call the M1 method of the active object. )
                  ;M
```

``Static'' methods can also be defined. They are otherwise ordinary Forth words which operate on the active object (or in other words they belong to a certain class of objects which can use them to perform certain tasks). The concept of ``static'' methods does not add anything new to the OOP support but it arises as a side product of this implementation of classes and objects. Nevertheless ``static'' methods can be useful in ``factoring'' the virtual methods.

Obtaining the Standard Size and the Address of the Virtual Method Table of a Class

Every class has a standard size (stored in the header of the class). Also every class has a VMT table (as a part of the class header). The size of the **CONTEXT** class can be accessed by using the words **SizeOf (-- Addr)** or **[Size] (-- Addr)**. The address supplied in both cases is the address of the cell containing the standard size of the class. I am talking about ``standard size'' here because in some cases objects belonging to the same class can have different sizes (eg. arrays). It is the responsibility of the programmer to keep the size information recorded in the class header valid but some tools to facilitate this are provided in the program. The address of the VMT table can be obtained by **VMTof (-- VMT)** or by **[VMT] (-- VMT)** (11).

The Memory Layout of an Object

The only link between an object instance and its class is the address of the VMT table. This address is stored in every object in the cell immediately before the base address of the object. Yes, this means that the VMT occupies a *negative* offset. I have found that using a negative offset reduces the possibility of accidental errors when testing objects interactively. It is always tempting, especially during debugging, to access an object as if the latter was an ordinary variable. If the VMT address is stored at a negative offset objects really become similar to variables and other data structures defined by **CREATE ... DOES>** . Thus the first usable data field begins at the base address of the object. One method using the word **^** for accessing data inside the objects data area has already been mentioned. Another way of manipulating data inside the object is to use fields. The definition word **Field (Offset -- Offset+CELL)** can be used for creating fields with meaningful names. The following line will create a word *Year* and leave 12 (in my implementation) on the stack.

```
 8 Field Year
```

When the word *Year* is executed it will leave the base address of the active object plus 8 on the stack which is the absolute address of the field of the active object called *Year*. In order to make the declaration of fields even easier, two new words can be introduced:

```
: Fields     ( -- 1st-unused-offset ) SizeOf @ ;
: End-Fields ( 1st-unused-offset -- ) SizeOf ! ;
```

By using these new words the declaration of the new fields of a class will look like the following:

```
Fields
Field F1
Field F2
End-Fields
```

In this way the first new field of the class begins after the last field of the parent class (the size information is copied together with the VMT when a new class is declared) so that the fields of the parent are inherited. Also the size of the class is taken care of because **End-Fields** will store the offset of the first unused byte which is the same as the size of object's data area in bytes (12).

Constructors

In order to create instances of a given class one needs definition words. If the objects are allocated on the heap (13) the address of the VMT table still has to be assigned to it and its fields need to be initialized. Because objects belonging to one class can be located in different areas (eg. dictionary space and heap) I decided to implement a word which initializes the data area of an existing object. This word is called **Init** and it is implemented as a virtual method (14). If the objects have individual names and they are located in the Forth dictionary the simplest way to produce them is via definition words. One such definition word might be the following:

```
: Obj (  -- )
  VMTof CREATE , HERE SizeOf @ ALLOT { Init } DOES> ( -- Object) CELL+ ;
```

Notice that the effect is similar to that of **VARIABLE** in older Forth systems where an initial value had to be supplied.

An Example

To demonstrate the capabilities of my Forth system I created the example on Listing One. The base class *Numbers* has some methods -- *ways of behaviour* -- common to all numbers. This is an abstract class so it cannot be *instanciated* that means an object belonging to the class *Numbers* cannot be created (15). The two derived classes *Integers* and *Rationals* implement meaningful kinds of numbers. The latter two can be *instanciated*. After compiling the example we can define different kinds of numbers. The following line will create two rational numbers *R1* and *R2*:

```
Rationals 60 30 Obj R1 81 3 Obj R2
```

The world *Rationals* does not do anything but changes the **CONTEXT** class. In other words it specifies the type of the new object (**Obj** always uses **CONTEXT** to determine the type of the object to be created). Afterwards their values can be examined easily:

```
R1 . R2 .      1 / 2  1 / 27 Ok
```

(*R1* and *R2* are already normalized). An addition is also simple:

```
R1 R2 + .    29 / 54 Ok
```

In the example the value of *R1* **is** changed and it equals to 29/54. Objects belonging to the class *Integers* can be treated the same way:

```
Integers 20 Obj I1 50 Obj I2  Ok
I1 . I2 .   20 50 Ok
I1 I2 + .   70 Ok
```

In fact the same words (**.** , **+** **-** , ***** and **/**) can handle them (16).

The Accessibility of the Information

Information hiding is one of the usual features of an OOP language. The pioneers of OO-Forth spent a lot of effort on it [3]. On the other hand Forth is very often used by hardware developers, hackers and similar people who definitly will ignore such an effort. So I decided not to bother with it. One cannot really ``physically seal'' a piece of memory from experts. Also to let the user know what is going on ``behind the scenes'' saves a lot of trouble during debugging. This does not mean that I want to encourage hacking around the internal parts of an object -- which would render my whole effort spent on implementing this OOP support meaningless. I simply do not believe that a programmer who is not willing to use proper techniques can be forced to do so.

On the other hand the encapsulation is rather good in my model. In principle nothing from the outside can access the data area of an object; even the address of a field cannot be calculated. Only the object itself can ``make it known'' to the rest of the application. One advantage of Forth is that words are not split into cathegories as in other languages. There are no such things as ``operators'', ``keywords'' or ``identifiers''. This lack of differentiation means the programmer has more freedom to change the underlying implementation provided that the stack effect (thus the interface to the rest of the application) remains the same. Because of the latter property of Forth and because of the good encapsulation the programmer can

``hide'' the implementation details from the user in my Object-oriented Forth dialect even though the information would not be ``physically sealed''. To prevent accidental usage of words which are supposed to be ``factors'' or auxillary words one might create aliases of the usable methods in other vocabularies where the rest of the application's routines reside.

Conclusions

In this paper a dialect of Forth featuring OOP support and native code generation has been introduced. In this dialect the code generation especially the definition of compile time behaviours is controlled by the user. In this way one can decide which parts of the code are important and to be inlined or substituted by more efficient pieces of machine code and which are to be left alone. The main feature of the OOP support in this Forth is simplicity. It uses a vocabulary structure with some extras to implement classes with inheritance and late binding (thus polymorphism). Although most of the OOP support words do elementary manipulations (but what do you expect from an Assembly programmer) they can be used for building higher level interfaces tailored to individual taste and needs. The necessity of a good OO-Forth is obvious these days but the Forth community still does not have a standard. Although very many Object-oriented Forth implementations are available I found my dialect a very convenient one. The program size is small (the kernel part is about 32K -- 32-bit machine code, not threaded code) and the functionality is easy to understand. Because of the nano-compiler approach a programmer can easily keep in mind what is going on behind the scenes thus he/she has a better control over the system. The independent and ``animate'' object instances provide a better encapsulation thus facilitate an even more structured programming style than the usual OO-Forth dialects or *C++* and *Turbo Pascal*.

References

1. K.D. Veil and P.J. Walker 1994 Forth Nano-Compilers
   Forth Dimensions **XVI**, #2, July - August 1994, 34-37
2. Borland International Inc. 1992.
   Borland Pascal With Objects User's Guide
3. Dick Pountain *Object-oriented Forth*
   Academic Press Limited, London, 1987

Trademarks mentioned

Acknowledgements

I want to say thanks to the Hung Hing Ying Physical Sciences Research Fund for providing a grant to pay for my 486.

Listing One

```
FORTH DEFINITIONS

: GCD ( U1 U2 -- GreatestCommonDivisior )
  BEGIN
  2DUP <> WHILE                 ( If U1=U2 either will do. )
   2DUP MIN >R MAX R - R>       ( Substract the smaller from the greater )
  REPEAT                        ( Chech again if U1=U2. )
  DROP ;                        ( One of them is enough. )

Objects DEFINITIONS
5 Class Numbers Numbers DEFINITIONS
Method Add ( Number -- )
Method Sub ( Number -- )
Method Mul ( Number -- )
Method Div ( Number -- )
Method Print

0 Class Integers Integers DEFINITIONS
```

```
Fields
Field N
End-Fields

As Init use: ( N -- ) N ! ;M

As Add use: { N @ } N +! ;M

As Sub use: { N @ } NEGATE N +! ;M

As Mul use: { N @ } N @ * N ! ;M

As Div use: { N @ } N @ SWAP / N ! ;M

As Print use: N @ . ;M

Numbers DEFINITIONS
2 Class Rationals Rationals DEFINITIONS
Fields
Field NUM
Field DEN
End-Fields
Method Normalize ( -- )
Method Invert    ( -- ) ( 1/x )

As Init use: ( DEN NUM -- ) NUM ! DEN ! Normalize ;M

: (Add) ( num den -- )           ( A "factor" of Add )
        ( This is not the best way to add two rational )
        ( numbers but as an example it will do.        )
   DUP NUM @ * NUM !             ( NUM*den                )
   DEN @ SWAP OVER * DEN !       ( DEN*den => DEN         )
   * NUM +!                      ( num*DEN+NUM*den => NUM )
   Normalize ;

As Add use: { NUM @ DEN @ } (ADD) ;M

As Sub use: { NUM @ NEGATE DEN @ } (ADD) ;M

As Mul use:
   { NUM @ DEN @ }     ( Get the values of the other Rational. )
   DEN @ * DEN !       ( Multiply denominator by the other's denominator.)
   NUM @ * NUM !       ( Multiply numerator by the other's numerator.)
   Normalize ;M

As Div use:
   { NUM @ DEN @ }     ( Get the values of the other Rational. )
   NUM @ * SWAP        ( Multiply numerator by the other's denominator.)
   DEN @ * NUM ! DEN ! ( Multiply denominator by the other's numerator.)
   Normalize ;M

As Print use: NUM @ . ." / " DEN @ . ;M

As Normalize use:
   DEN @ NUM @           ( Get denominator and numerator. )
   2DUP XOR >R           ( A not quite ANSI way to determine the sign. )
   ABS SWAP ABS          ( Calculate the absolute value of both. )
   2DUP GCD              ( Calculate the GCD. )
   DUP >R / DEN !        ( Normalize the denominator. )
   R> /                  ( Normalize the numerator. )
   R> 0< IF NEGATE ENDIF ( Adjust the sign. )
   NUM ! ;M

As Invert use: NUM @ DEN @ NUM ! DEN ! Normalize ;M
```

```
Numbers DEFINITIONS

( An now some words that look useful to an ordinary Forth programmer. )

: . ( Number -- ) { Print } ;

: + ( Number1 Number2 -- Number1+Number2 ) SWAP { Add 0@ } ;

: - ( Number1 Number2 -- Number1-Number2 ) SWAP { Sub 0@ } ;

: * ( Number1 Number2 -- Number1*Number2 ) SWAP { Mul 0@ } ;

: / ( Number1 Number2 -- Number1/Number2 ) SWAP { Div 0@ } ;
```

Footnotes

1. At the present stage the program has an ANSI compatible mode which supports most of the features of the new standard.
2. I needed full control but I did not need too much operating system connection. So I implemented a V86 monitor which takes care of file I/O and other operating system connections by running DOS in a virtual 8086 machine and provides facilities to execute a 486 style (32 bit) protected mode program.
3. The default action is to generate a subroutine call to the body of the word.
4. For example, consider the effects of **POSTPONE SWAP!** in the latter case. Is this the semantics one might expect?
5. In ANSI FORTH terms CCFA would be called a ``compilation token''.
6. Virtual method, or in *Smalltalk* terminology, a message is a piece of code which is subject to late binding. As opposed to a static method which is unique and defined only in one class so it can be identified at compile time, a virtual method means a series of subroutines -- one for each member of a family of classes -- thus the actual routine can be chosen only at run time when the active object is known.
7. Besides the theoretical considerations mentioned above this approach has an implementation advantage. The object pointer can be very easily implemented by dedicating a CPU register for it. (In my implementation I used two registers one for keeping the address of the object and one for keeping the address its Virtual Method Table -- the VMT.) In this way the cost of the field accesses and method calls can be greatly reduced.
8. Consider a large database which consists of great many objects interconnected via pointers. When the database image is saved to disk and it has to be reloaded to a different address all the pointers will become invalid and they must be fixed. On the other hand if the pointers are realtive they will remain valid no matter what the physical address of the database is.
9. The term ancestor seems to me rather intuitive but for those who like definitions the following will do: Vocabulary **A** is the parent of vocabulary **B** if **B** was created with **A** being the **CURRENT** vocabulary. Vocabulary **X** is an ancestor of vocabulary **Y** if **X** is the parent of **Y** or **X** is an ancestor of the parent of **Y**. In other words **B** is a child of **A** and **Y** is a successor of **X**.
10. The term VMT as most of my terminology is borrowed from *Turbo Pascal* [2].
11. The difference between **[Size]** and **SizeOf** (also between **[VMT]** and **VMTof**) is the same as the difference between **[']** and **'**.
12. In machines which are not capable of addressing individual bytes the indication of the object's size in bytes can be meaningless. In my implementation it is the easiest way to go because on the 486 even in 32-bit mode, individual bytes are accessible.
13. My program has not yet have a built-in **Memory-Allocation** word set support but for the time being any standard definition of this word set will do. I have found Gordon Charlton's **ANS HEAP** to be useful.

14. Although **Init** is quite different from the virtual methods: if a virtual method with a certain name is defined in a class with a given stack effect it is supposed to have the same stack effect in all the successor classes. This is not true for **Init** and it is just an implementation trick to define this word as a ``virtual method''.

15. In reality the definition word **Obj** will create such an object but any attempt to call its methods will trigger an error message.

16. At first sight this ``sharing the operators'' might look close to the idea of *C++ Operator Overload* but the latter one implies a typed language. In our example these ``operators'' can work on any derived class of *Numbers* but always on two operands belonging to the *same class* while a *C++*-style operator should be able to handle such cases as multiplication of a rational number by an integer.