The edited-for-publication version of the following paper originally appeared in *Forth Dimensions*, **XVIII #1**, pp 31-35 (1996).
Forth Dimensions is the magazine of the Forth Interest Group P.O. Box 2154, Oakland, California 94621 USA
E-mail: office@forth.org
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/Linux/doof-X.Y.Z.tgz. (Instead of X.Y.Z you have to include the version number. Watch for newer versions.)

Does Late Binding Really Have To Be Slow?

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

Introduction

Today's software tool is object-oriented programming, as many programmers will agree with me on this matter. No surprise that many Forth dialects already have some sort of OOP support either built into the kernel [2,3] or as an add-on feature [1]. On the other hand many from the Forth community will argue that the overhead involved in virtual method calls and field accesses is unacceptable in time critical applications. This paper is directed towards them, and its main goal is to make object-oriented techniques more attractive for those who like the "close to silicon" approach.

In this paper the implementation details of my Object-oriented Forth model (1) [3] will be presented. In this model overhead involved in virtual method calls and field accesses is very low. Actually on the Intel486(TM) microprocessor there is no overhead involved in virtual method calls and field accesses. The only overhead is in object instance access. In other words once an object has become active all virtual method calls take exactly the same time as any other call and field accesses take as much time as an ordinary variable access. Because the usefulness of this implementation technique is not limited to Forth, most of the following ideas have already been published [4], here the emphasis will be on our language-specific advantages.

An Object as a "Mini Universe"

I was reading an old article in Byte [5] which complained about the instructions and clock cycles spent on ordinary housekeeping in any program. The problem is that function parameters are passed on the stack so the caller has to push them onto the stack and the callee has to fish them out(2). In a well-factored program this parameter passing can be significant. Of course for us Forth programmers the painful part is not parameter passing (because we use the stack for our temporaries anyway) but the need to rearrange the stack. A piece of code using only global variables would be much cleaner and faster because it would not have to fiddle around with the parameters. The problem is that such a routine would be extremely inflexible and not at all reusable.

Considering all these I immediately thought about OOP. And object can have its own world where variables (the fields of the object) are all in some way at a fixed address so they do not have to be passed as parameters. All a routine (a method in this case) has to know who (3) is the object calling it. Explicit parameter passing is required only in the communication with the outside world.

In object-oriented languages objects and their methods -- either virtual or static -- constitute an alternative way of factoring. Not only common instruction sequences (4) can be factored out, but data structures and pieces of code handling them together; which can lead to a much cleaner program design (5).

Methods or Messages?

I have to explain at this point why I prefer the term "virtual method" to "message". The latter implies that one object communicates with another and only as a special case can the sender and the receiver of a

message be the same object.

The word "method" only indicates that the routine is somehow special in the sense that it operates on an object instance and assumes certain properties of that object (e.g. its memory layout). In other words a method is a piece of code which implements a certain particular kind of behavior of a class of objects. The more specific term "virtual method" means a method whose identity (the actual piece of code) is not known when the program is compiled (6) so it must be determined at run time. Because in my OOF methods are often used for factoring out reusable functionality inside an object I consider the term "message" to be rather misleading.

An "Object-oriented Architecture"

The above ideas might look good on paper but to learn more about them we need an implementation. When I started to experiment with my OOF I wanted to comply with the following conditions:

- The system must have efficient support for both field accesses and virtual method calls.
- The compiler has to be simple, so the efficiency of the OOP support must not depend on optimization.
- The details of OOP implementation must be expressed in the language so that the user of the system can tailor the high level layer to his or her individual taste(7).

As a working hypothesis let us consider a microprocessor which has built in support for the above mentioned kind of object-oriented behavior. We need one register to contain the base address of the active object; all field addresses will be relative to this base address. Another register is needed to hold the address of the *Virtual Method Table* or **VMT** which contains the addresses of the virtual method instances(8). Our theoretical microprocessor needs a couple of instructions to load these registers, and to read their contents and save them on a stack.

So the basic ideas are clear but the next problem is that the whole thing has to be implemented somehow in hardware. Making an object-oriented microprocessor is far beyond our facilities so I had to emulate the above behavior on an existing one. Our PC-s are based on the Intel486(TM) chip which does not have very many registers but enough to spare two for OOP support. This microprocessor also has quite flexible addressing modes so once the base address of an object is in a register, one instruction is enough to read or write one of its fields. The same can be said about the **VMT** pointer and virtual method calls.

In my Dynamic Object-oriented Forth or DOOF (the implementation of my OOF model for the Linux operating system) I have used the following register assignment:

eax : TOS=The **T**op **O**f **S**tack.
edi : PSP=The **Pa**rameter **S**tack **P**ointer(9).
esp : RSP=The **R**eturn **S**tack **P**ointer.
ebp : LSP=The **L**ocals' **S**tack **P**ointer.
ebx : OBJ=The **OBJ**ect Pointer.
esi : VMT=The **V**irtual **M**ethod **T**able Pointer.

Examples

Listing One (10) shows a sample object which is just a point on the screen (actually a character position on a character screen). This simple object has two fields which are its X and Y coordinates in a Cartesian coordinate system. A Point can show itself and hide (delete) itself.

Listing One

```
Objects  DEFINITIONS     \  Objects  is  the  base  of  the  Class  hierarchy.
3  Class  Points         \  A  new  class  with  3  new  virtual  methods.
Points                   \  The  detailed  description  of  the  new  class.
Field  X                 \  The  X  coordinate  of  the  Point.
Field  Y                 \  The  Y  coordinate  of  the  Point.
Method  Show  (  --  )   \  How  to  show  a  Point.
Method  Hide  (  --  )   \  How  to  hide  a  Point.
Method  Move  ( dY  dX  -- )  \  How  to  move  a  Point  a  given  distance.

As Init use: ( Y X  -- ) X ! Y ! ;M  \ Initialization of a new object instance.
```

```
:  GotoXY  (  --  )  X  @  Y  @  AT-XY  ;  \  This  is  a  static  method.

As  Show  use:  GotoXY  [CHAR]  *  EMIT  ;M

As  Hide  use:  GotoXY  SPACE  ;M

As  Move  use:  Hide  X  +!  Y  +!  Show  ;M

10  40  Obj  P                      \  Create  a  new  object  instance.

:  Test  ( dY  dX  -- )  P { Move } ; \  A  sample  word  using  our  object.
```

These are the most basic virtual methods or types of behavior a Point can manifest. One example of a more complicated kind of behavior is that a Point can move itself a specified distance in the X and Y direction. A good example for field accesses is **GotoXY** which is a static method. As we will see later there would be no performance penalty if we implement it as a virtual method. DOOF generated the following sequence of Intel486(TM) instructions (11) for **GotoXY**:

```
        sub         edi,4               ; Make room on the stack.
        mov         [edi],eax           ; Save top of stack.
        lea         eax,[ebx].X         ; Get the address of the first field.
        mov         eax,[eax]           ; @
        sub         edi,4               ; Make room on the stack.
        mov         [edi],eax           ; Save top of stack.
        lea         eax,[ebx].Y         ; The address of the next field.
        mov         eax,[eax]           ; @
        call        AtXY                ; An ordinary call of a Forth word.
        ret                             ; Return.
```

Not quite a piece of highly efficient code but so far no optimization has been used (12). A more intelligent compiler would output something like the following:

```
        sub         edi,8              ; Make room for two items on the stack.
        mov         [edi+4],eax         ; Save top of stack.
        mov         eax,[ebx].X         ; Get the value of the first field.
        mov         [edi],eax           ; Save the value on the stack.
        mov         eax,[ebx].Y         ; The value of the next field.
        call        AtXY                ; An ordinary call of a Forth word.
        ret                             ; Return.
```

This code is concise and efficient, but the problem is that the compiler has to be smart to produce this kind of output. With present compiler technology this optimization is possible [6,8] (13). Nevertheless the first version of the code can be produced by any dumb compiler.

If we take a closer look at either version of the code we can see that the number of instructions to access a field of the object is the same as the number of instructions needed to access a global variable (with the same smartness of the compiler) (14).

So far so good, but OOP is not only about field accesses. The quality of an OOP implementation depends on the efficiency of virtual method calls. The only one virtual method instance in our example which calls other virtual methods is **Move**, which has been compiled to the following sequence of instructions by DOOF:

```
        call        [esi].Hide          ; Call Hide.
        sub         edi,4               ; Make room on the stack.
        mov         [edi],eax           ; Save top of stack.
        lea         eax,[ebx].X         ; Get the address of the first field.
        mov         edx,[edi]           ; Move value to a register.
        add         [eax],edx           ; Add to the cell at the address.
        mov         eax,[edi+4]         ; Move next top item to tos register.
        add         edi,8               ; Adjust stack pointer.
        sub         edi,4               ; Make room on the stack.
        mov         [edi],eax           ; Save top of stack.
        lea         eax,[ebx].Y         ; Get the address of the next field.
        mov         edx,[edi]           ; Move value to a register.
        add         [eax],edx           ; Add to the cell at the address.
        mov         eax,[edi+4]         ; Move next top item to tos register.
```

```
        add         edi,8               ; Adjust stack pointer.
        call        [esi].Show          ; Call Show.
        ret                             ; Return.
```

Again this is rough code generated by a dumb compiler. On the other hand what we have been interested in is clearly visible in the above list: the virtual method calls are single instructions (15). In other words a virtual method call is exactly as expensive as any other call as long as our program does not change the active object instance.

Changing the Active Object Instance

The word **Test** in our example changes the active object and then calls one of its virtual methods. The machine code generated by DOOF reads(16):

```
        sub         edi,4               ; Make room on the stack.
        mov         [edi],eax           ; Save top of stack.
        mov         eax,offset P        ; Push the address of the object
                                        ; onto the stack.
        push        esi                 ; Save the VMT Pointer.
        push        ebx                 ; Save the Object Pointer.
        mov         ebx,eax             ; Load Object Pointer from the stack.
        mov         esi,[ebx-4]         ; Load VMT Pointer with the
                                        ; new object's VMT.
        mov         eax,[edi]           ; Reload top of stack.
        add         edi,4               ; Adjust stack pointer.
        call        [esi].Move          ; Call virtual method.
        pop         ebx                 ; Restore Object Pointer.
        pop         esi                 ; Restore VMT Pointer.
        ret                             ; Return.
```

Of course the fiddling with the stack makes our code somehow obscure again. A smarter compiler could emit something like the following:

```
        push        esi                 ; Save the VMT Pointer.
        push        ebx                 ; Save the Object Pointer.
        mov         ebx,offset P        ; Load Object Pointer from the stack.
        mov         esi,[ebx-4]         ; Load VMT Pointer with the
                                        ; new object's VMT.
        call        [esi].Move          ; Call virtual method.
        pop         ebx                 ; Restore Object Pointer.
        pop         esi                 ; Restore VMT Pointer.
        ret                             ; Return.
```

From the second version of the code which can be generated from **Test** we can clearly see that the overhead in changing the active object is significant (six housekeeping instruction for only one useful virtual method call). On the other hand if the virtual method calls other virtual methods (as in our case) or accesses a couple of fields in the active object before the program switches to another object, the code can be significantly smaller and faster than in implementations on as a parameter on the stack(17) [4].

Multiple Inheritance and Other Bells and Whistles

My OOF has always been intended to be a low level language which provides the advantages of late binding with no or only minor performance hit. In such a low level concept more complicated constructs like multiple inheritance and operator overloading have no place. Of course most of them can be implemented on the top of the features provided by OOF. For example multiple inheritance can be provided by aggregating several objects together, and for every method the compiler can present the appropriate part of the object [10]. By making the Class **Classes** which is the special vocabulary type representing classes of objects [3], a part of the class hierarchy, DOOF provides a way to derive more sophisticated tools to create new types (18).

Operator overloading is more troublesome. To meaningfully provide operators which can do arithmetic on different types of numbers we have to provide a way for the operator to check the type of its operands. Traditionally items on Forth's stack are typeless, so this checking cannot be done without substantially modifying the internals of our traditional model of a Forth engine. The resulting language is still Forth but the overhead of the runtime type checking can make it extremely unattractive for the designers of time

critical applications.

In arguments about Forth, the complete lack of predefined data structures is listed as one of the major weaknesses of the language. After I have read some C++ literature [9,10], and I have seen the complaints about C's array concept and its incompatibility with C++ objects(19) I have started to think otherwise: Our *greatest weakness* can be our biggest strength(20) because we do not have to take away anything from the language to adapt OOP. Plain old-fashioned Forth is the best starting point to make some kind of Forth++(21). We do not have array concepts to be incompatible with more carefully designed data aggregates needed for an object-oriented system. Forth -- very luckily -- has remained a low level tool, a kind of portable assembler, for most of its users. My OOF model has been designed to live peacefully with this approach.

Conclusions

In this paper a fast and effective way of implementing OOP has been presented. The benefits of the technique are most useful in applications where speed is critical. The suggested technique provides the benefits of object-oriented design without the need of a sophisticated compiler. Although the examples are Intel-specific the presented argument will hold true on most modern microprocessors.

The usefulness of this approach is not limited to one particular programming language, nevertheless those languages which encourage extensive factoring will benefit mostly from it. In Forth where the average definition is only a couple of words long(22) the implicit parameters passed from method to method with no extra cost can make a program based on OOP even faster than its counterpart implemented by using more conservative techniques(23).

References

1. D. Pountain *Object-oriented Forth*, Academic Press Limited, London (1987).
2. M. Dahm, *Object-oriented Forth*, *Forth Dimensions*, Vol. XIV No. 1. 16-22 (1992 May).
3. A. Zsoter, "An Assembly Programmer's Approach to Object-oriented Forth", *Forth Dimensions*, Vol. XVI No. 6, 11-17 (1995). The HTML version of this paper is available at http://www.forth.org/literature/andras.html
4. A. Zsoter, "Implementation of object-oriented programming via register based pointer", *Journal of Microcomputer Applications* 18, 279-285 (1995).
5. P. Wilson, "The CPU Wars. An overview of the microprocessor battlefield, and how it got that way", *Byte* Vol. 13. 213-234 (May 1988).
6. D. Watson, *High-Level Languages and Their Compilers.* Addison-Wesley Publishers Limited (1989).
7. P. A. Steenkiste. "Advanced Register Allocation", In *Topics in Advanced Language Implementation*, edited by P. Lee, MIT Press (1991).
8. T. Pittman and J. Peters, *The Art of Compiler Design, Theory and Practice*, Prentice-Hall, Inc. (1992).
9. B. Stroustrup *The C++ Programming Language* Addison-Wesley Publishing Company (1994).
10. M. A. Ellis and B. Stroustrup *The Annotated C++ Reference Manual* Addison-Wesley Publishing Company (1995).

Trademarks Mentioned

Intel486(TM) is a trademark of Intel Corporation.

Glossary

Objects *(--)*
The base class of all classes.
Class *(Methods "Name" --)*
Creates a new class(24)with *Methods* new slots for virtual methods in its VMT.
< | *(-- End)*

Performs the function of **DEFINITIONS** followed by pushing the offset of the end of an object instance (which is the same as the offset of the first of its new fields to be) onto the stack.

|> *(End --)*
Makes *End* (the first unused offset) the new default size of the class.

Field *(o1 "Name" -- o2)*
Creates a word *Name*. If **Name** is executed later it pushes onto the stack the address of the cell which is *o1* away from the base address of the active object.

Method *("Name" --)*
Creates a word *Name*. This word is the application programmer's interface to a new virtual method. If this word is executed later it will call the corresponding virtual method in the active object's **VMT**. The method indexes are assigned automatically as long as there are enough free slots in the class's **VMT**.

As *("Name" -- Index)*
Pushes the index of the virtual method identified by *Name* onto the stack.

use: *(C: Index -- use-sys)*
Starts a headerless definition which will be the body of a virtual method instance of the class.

;M *(C: use-sys --)*
Finishes a definition started by **use:** .

{ *(Obj --)*
Pushes the active object onto the return stack and then selects *Obj* as the active object.

} *(--)*
Reactivates the object which was previously pushed onto the return stack by **{**.

Init *(args* --)*
This is the name of a "virtual method" (25) which initializes the active object with *args**. The latter can be any number of arguments required by the type of the object.

Obj *(args* "Name" --)*
Creates an object with the Forth name *Name* and initializes it via **Init**.

Notes

1. To avoid confusion I will refer to this model as "my OOF" because other Object-oriented Forth implementations [2] are also called OOF.
2. Of course on CPUs with huge register files and with register windows stack accesses can usually be avoided.
3. We consider objects as animate entities.
4. Or, in a more Forthish parlance, word sequences.
5. Here I have to do what many Forthists would regard as heresy and recommend a C++ textbook [9]. The usefulness of design principles suggested in that book are far beyond the scope of one particular programming language.
6. If the type of an object is known at compile time a smart enough compiler can do the binding statically. If a method with identity known at compilation time is small enough a compiler can even inline it [9].
7. I used Forth as an Assembly language of a hypothetical machine with Object-oriented architecture.
8. I will call an abstract entity corresponding to a "kind of behavior" or a "kind of message the objects of a class answer to" a virtual method. The actual routines corresponding to the actual behavior of the objects of a given class will be referred to as virtual method instances in this paper.
9. Unlike in the MS-DOS version of the program, the parameter stack is not the same as the machine stack. This makes the system slower because the only efficient stack handling instructions of the x86 family of microprocessors use the (e)sp register. So any operation which changes the number of items on the stack has to increment or decrement the parameter stack pointer explicitly, using an extra instruction. On the other hand linking with existing C and C++ code as well as implementing most of the system in C++ are much easier in this way.
10. Because the OOP-support words are non-standard words there is a glossary at the end of this paper which explains them.
11. DOOF, just like OOF, does not use threading. It generates machine code.

12. Because no optimization has yet been implemented in DOOF.

13. Of course it will not help us if we have to make a Forth implementation which runs on a machine with very limited resources.

14. A variable access usually means pushing the address of the variable onto the stack and then executing a @ . In either version of the compiled program the instruction sequence would be similar to the above except that, instead of [ebx+offset], a simple constant address would be used.

15. Unfortunately on architectures with less flexible addressing modes this is not always the case. Depending on the flexibility of indirect calls the overhead can be very small (zero instructions as on the Intel architecture, one instruction on CPU-s which do not have indirect calls but can use a register as target address, and it can take even longer on very simple microprocessors where indirection is not quite supported).

16. The **VMT** pointer is stored in the cell immediately before the object. In this way the body of the object is not interrupted with an extra field and the address of the object can be passed to routines written in languages other than Forth as ordinary data structures.

17. Of course if everything is an object, and even ordinary integer arithmetic is implemented via method calls, my argument will not hold true. On the other hand such a system should only be implemented for good reasons, e.g. in certain AI programs. The presented technique is intended to improve efficiency not to replace sanity in the program design.

18. I am in trouble to put it into proper words. As traditional Forth is an untyped language we have not developed a terminology for such kind of concepts as *the type of a type*.

19. For further details see the notes at the end of x8.2.4 in reference [10].

20. This statement sounds very Taoist, no wonder that many programmers outside the Forth community consider Forth a religion rather than a programming language.

21. The term Forth++ appears more and more often in Forth discussion but there is no implementation I am aware of which would actually have the name Forth++.

22. Or at least we should strive for adopting a programming style with short definitions.

23. This issue has yet to be examined. Of course I am talking about equal functionality. If a program needs a certain level of indirection (e.g. I/O primitives have been vectored in many Forth implementation) OOP is definitely a good alternative.

24. In both existing implementations of the model described in this paper classes are represented as specialized vocabularies which have a **VMT**.

25. A more detailed description of the "fake virtual method" **Init** can be found in [3].