Method dispatch in Oforth

M. Franck Bensusan
http://www.oforth.com

Abstract

Oforth is a Forth dialect that implements Object Oriented Programming as a built-in mechanism. For methods, it provides a full dynamic binding : two classes that are unrelated (ie Object is their common parent) can implement methods with the same name and the method to execute is resolved at runtime. Furthermore, classes are never "closed" and it is possible to extend a class with new methods at any moment.
As many core words are implemented as methods, method dispatch must be as fast as possible, while, if possible, limiting the memory used.
This paper discusses the implementation of method dispatch in Oforth : classic virtual tables are used to cache code addresses but they are allocated and constructed at runtime, while methods are executed. This is done without suffering much performance penalties.

1 Introduction

Oforth is a Forth dialect that implements a full OOP model. Many core word, like #+, #-, ... are implemented as methods so method dispatch must be as fast as possible. There are two more constraints to be addressed : dynamic binding and non-closed classes. "Dynamic binding" means that all classes can implement all methods, whatever their position in the hierarchy and the selection of the method to run will occur at runtime, according to the top of stack. "Non-closed" classes means that we can always add a new methods to an existing class. For instance, we can create the Integer class, then create the Float class, then add the “>float” method to the Integer class.

With theses constraints, it is not possible to create, at compile time, a definitive virtual table for each class with a pointer to this VT stored in each object. We have to adjust the virtual tables at runtime.

In this paper, we look at the syntax of messages, class definition and method definitions (2), the dispatch message mechanism implemented (3), optimizations that occur at compile time (4), some discussion about the performances and memory cost (5), and some discussions for future work (6).

There have been many works on method dispatch in the general programming language literature ([DUC11] for instance) and some work in the Forth community ([RP96], [ERT12]). This paper is not intended to expose new ideas on this subject : its objective is to expose the dispatch method used in Oforth and what choices have led to this implementation.

2 Messages, classes and methods

Messages are represented by words created in the dictionary. They can be "ticked", executed, ... as classic words. You will almost never create a new message without its first method, but, if necessary (forward definition for instance), you can do it using :

```
message: foo
```

A class is also a word in the dictionary. It is created by sending the #new: message to the Class class (a meta-class) :

```
Object Class new: A
```

This creates a new word, A, in the dictionary with Object as its parent. Oforth only supports single-inheritance. Using A word will push the class on the stack.

Once a class is created, methods can be added :

```
A Class new: A1

A1 method: foo
   "Foo for A1 :" . self . ;

Object Class new: B

B method: foo
   "Foo for B :" . self . ;

A method: bar
   "Bar for A :" . self . ;

A virtual: foo2
   "to be redefined" abort ;

A1 method: foo2
    "Redefined: " . self . ;

#bar .s
[1] (Message) #bar
```

If messages (here words foo, bar and foo2) were not created yet, they are created when the first method corresponding to the message is created.

All methods call (whether they are virtual or not) have dynamic binding, according to object on top of stack. Calling a method is just like calling a word, but the object that will receive the message have to be pushed on the stack first. One important rule is that this TOS is removed from the stack when calling the method, and stored on the return stack. In order to push this TOS (called the method receiver) on the stack in the method's body, the self word can be used. For instance, this is how the previous words are called on objects :

```
A1 new foo
Foo for A1 : aA1 ok

B new foo
Foo for A2 : aB ok

A new dup bar foo2
Bar for A : aA [console:1] #Exception : to
be redefined

A1 new dup bar foo2
Bar for A : aA1 Redefined:  aA1 ok
```

Methods can't be redefined into subclasses unless they are declared as virtual (here foo2, for instance). Non virtual methods correspond to final methods in Java : they can't be redefined in subclasses. Declaring a method as virtual can have impact on optimizations during compilation (see chapter 4).

Ticking a word is done using the # word and not ' (' is dedicated to characters). No space is needed between # and the name. So #bar will push the word bar (here a message) on the stack, or compile a literal into the current definition when compiling.

There is no word such as "end-class". The #bar method is added to A after A1 and B are declared. This allows to extend a class whenever we want, but this also adds constraints on the dispatch mechanism as the list of messages a particular class can respond to is never fixed once for all.

Furthermore, many core words are implemented as methods. The number of messages a class may respond to can be very important and this also adds constraints to the dispatch performances.

3 Dispatch mechanism

3.1 Object's tag field

Instead of associating an index with each message, Oforth uses an "orthogonal" mechanism : an index is associated with each class. In the first slot of each object, a tag is stored, which includes its class index (attributes are stored after this field). On 32bits systems, the class index is present in the 12 least-significant bits of the tag field :

```
0xnnnnnIII
```

Here, the class index value is III. By the way, this means that, on a 32bits Oforth systems, we can't declare more than 4095 classes (the Object class index is 1).

Other information in the tag field is not used for the dispatch mechanism and is not discussed here. Figure 1 shows the tag field stored into each object.

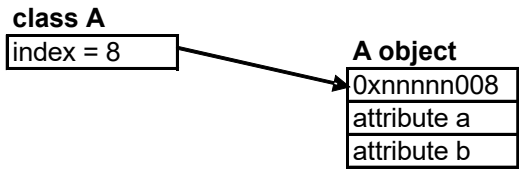

Figure 1 : tag field

3.2 One virtual table by message : the MVT.

As indexes are associated with each class, messages hold the virtual tables : the Message Virtual Table (MVT). Each slot of the MVT contains the address of the method's code to execute for the class corresponding to the index slot. The first slot of a MVT (index 0) holds its size.

When a message is created, it points to an empty virtual table (a static slot with value 0).

Figure 2 shows a MVT for message foo. At index 8, we find the code address of the method to be executed for objects of class A.

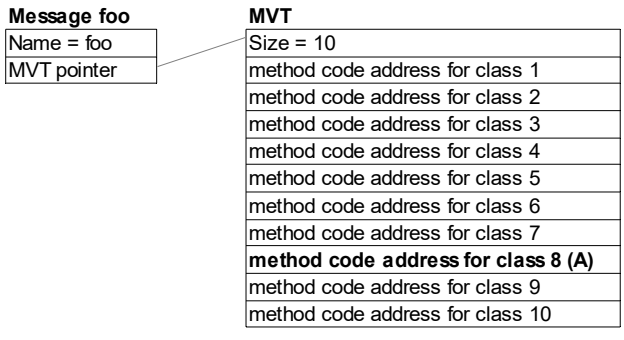

Figure 2 : a Message Virtual Table

3.3 Sending a message

As it is not possible to populate the MVT when classes are declared, everything must be handled at runtime, when messages are sent.

Listing below is the assembler code (x86 32bits) that is executed to send a message (in register r1). The method to execute is retrieved according to the Top Of Stack (TOS) class :

```
func(runMessage)
   test $1, TOS                  (1)
   jne LcallMethodInteger        (1)
   test TOS, TOS                 (1)
   je  LcallMethodNull           (1)

   movl (TOS), r0                (2)
   andl 0x00000FFF, r0           (2)

   cmpl IDClass, r0              (3)
   je LcallMethodClass           (3)

   movl virtualTable(r1), r2     (4)

   cmp r0, *r2                   (5)
   jl reallocMVT                 (5)

   movl (r2, r0, 4), r3          (6)
   jmp *r3                       (6)
```

Registers used are macros to map CPU registers. For x86 CPU, register allocation is :

```
#define r0          %eax
#define TOS         %ebx
#define r1          %ebp
#define r2          %ecx
#define r3          %edx
```

virtualTable(r1) is the field offset of the MVT pointer in the message objects.

Steps that occur during the runMessage function are :

(1) If TOS is a primitive integer or null, TOS is the value itself (and not a pointer) and we can't retrieve the tag field value from those objects. Class index (r0) is set manually before going to (step 4)

(2) Otherwise, we retrieve the class index (in r0) from TOS tag field.

(3) If it is the index of Class meta-class, we have to search for a class method to execute and the dispatch is done using another mechanism (see 3.6).

(4) The MVT associated with the message is retrieved (in r2)

(5) The MVT size is checked. If the size is smaller than the TOS class index, the MVT is reallocated (see 3.4).

(6) An indirect jump to the MVT slot value corresponding to the class index is performed.

3.4 MVT dynamic setting and reallocation

When a message is created, it points to an empty static MVT of size 0 (one static slot with 0 value). So no memory is consumed until the message is actually performed.

When the message is performed, if the MVT size is smaller than TOS class index (this will always be the case if the MVT is the static empty MVT), a new MVT of greater size is allocated and all its slots are populated with the address of a function named "polymorphic", then we go back to the dispatch mechanism. At this point, the MVT is larger enough and we can retrieve the value of the slot corresponding to the class index and run the "polymorphic" function. The purpose of this function is to retrieve the code address to be executed for class r0 and to adjust the slot value with thus value. This is done only once and, the next time, the slot will hold this calculated address code of the method to run and will jump directly to this address.

Figure 3 shows the MVT for foo message just after its first execution for class A. If it is executed again for class A, the method code is now performed. If it is performed for another class (index 5 for instance), the "polymorphic" function will update the slot 5 with the address of method code to run for class “5”.

The same code (step 6 in code 3.4) is used to adjust the MVT slots values (when their value is "polymorphic") and to run the method code (when their value is the method code address).

Also, as the MVT pointer is always accessed when a polymorphic call is performed for a message (step 4), we can extend classes by adding new methods without needing to adjust objects already created.

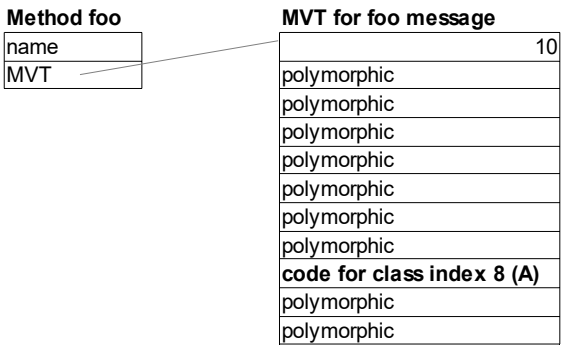

Figure 3 : MVT after executing foo for class A

3.5 The "polymorphic" function

The "polymorphic" function job is to retrieve the code address to execute for the message and TOS, and store its value in the message's MVT.

Each message in the dictionary has a linked list of all the methods declared and each method has two attributes : the class and the code address. A message is a word (with a name), but a method is not a word.

The "polymorphic" function starts with the TOS class and tries to retrieve into the linked list a method for this class. If not found, it retrieves the superclass of this class and searches again until it finds a method or it reaches null (null is the superclass of Object).
The algorithm is actually a little more complex, as it takes into account Properties (at each level, the search is done for the class and its properties).

If a method is found, the MVT slot is updated with its code address. Otherwise (ie the superclass null is reached), the virtual #doesNotUnderstand message is executed for TOS (default behavior, at Object level, is to raise a "does not understand" exception, but it can be redefined for a particular class).

3.6 Dispatch for class methods

For class methods, the dispatch mechanism is different. The correct implementation is also retrieved at runtime but without MVT : each time, the search is done through the hierarchy to retrieve the correct code to run : each class is searched one by one in the order of the inheritance until a method is found. This (slow) dispatch search has been chosen as it will not be used a lot because of optimizations that occur during compilation (see next chapter) : class methods call will mostly be optimized. This will save memory, as no MVT is allocated for class methods.

4 Optimizations during compilation

Those optimizations are handled by the #compile method implemented for messages words. If an optimization is possible, the polymorphic call is reduced to a procedure call.

4.1 When TOS is self

When the last word compiled is self, the receiver will be pushed on the stack. For instance :

```
A method: foo
   self bar ;
```

In this case, we know the type of the TOS object when #bar is performed (here A or one of its subclasses). So, at compile time, we search for a method to run. If we find a non-virtual method, it is the one that will be performed at runtime, so we can optimize by compiling a direct call to this method's code.

If the method found is virtual, no optimization is possible.

4.2 When TOS is a literal

When the last instruction is a push of a literal on the stack (Integer, Float, String, Word, ...), we also know the type of TOS at compile time and we can optimize by compiling a direct call.

A literal can be word, including a class. So this optimization will often apply when performing a class method :

```
: test
    120 Array newSize ;
```

Here the call to message #newSize will be optimized as we know that TOS will be the Array class. That is why there is no MVT allocated for class methods (see 3.6 dispatch for class methods).

4.3 When the message is declared for Object

If the message to compile is declared at the Object level and is not virtual, a direct call is compiled. It is the case for messages like #apply, #detect, #include?, ...

4.4 Otherwise...

If the type of TOS can't be detected at compile time and no optimization is possible, a polymorphic call is generated by calling the runMessage described before. Using the message word (its "name token"), the code generated in the current definition is :

```
movl message, r1
call runMessage
```

5 Performances and memory used

5.1 Performances considerations

When a polymorphic call is performed, 4 memory accesses and one indirect jump are executed :
- Access to the tag field of TOS (step 2).
- Access to the message's MVT (step 4).
- Access to the MVT's size (step 5).
- Access to the code to run and jump (step 6)

Memory access for step 2 and 6 are mandatory : we need to access the object to retrieve an information about its type, and we need to retrieve the MVT slot value. Furthermore, on modern processors, access to the field tag probably cache the object's attributes that can be accessed in the method.

Memory access for steps 4 and 5 are not strictly necessary but needed if we want to re-allocate the MVT at runtime and have extendable classes. On modern processors, (5) may cache the access to method's code address (6).

Two other mechanisms are used to optimize performances :

1) A static MVT of size 0 is associated with the newly created message. So there is no need to check if the MVT is null : we directly check if the MVT size is greater than the index (step 5).

2) Polymorphism is calculated by a function ("polymorphic") whose code address initializes the MVT slots. So the same jump to the slot address (step 6) allows to calculate the method to run (the first time) and to directly run the method code (the next times). There is no need to check if the slot value is empty or not.

5.2 Memory used

Main objective for the dispatch mechanism is performances, but it allows to save some memory compared to a "n classes x m messages" matrix :

1) Newly created messages don't allocate virtual table. A MVT is created only if the message is sent at runtime. This is important as many core words are messages and not all declared messages are used at runtime.

2) MVT size are calculated at runtime and won't be greater than necessary (the max index of the class that receives the message).

3) MVT are "by message" and not "by class", so there are many “small” MVT instead of few big virtual tables (one by class).

4) There is no MVT for class methods so no memory is allocated.

Nevertheless, there may still have lot of unused slots in a MVT. It could be interesting to implement other mechanisms (hold also a minimum index or hash MVT, ...). Those mechanisms have not been implemented yet.

5.3 Benchmarks

The following (simplistic) benchmark tests the various cases. On modern processors, everything will be in cache (particularly the message and its MVT) and branch prediction will apply.

Tests have been run on a core i7-4720 HQ 2,6Ghz on Windows 10.

```
: f ( -- )  ;

Object virtual: m   ;
Float method: m ;

Object Class new: A
A method:      m   ;
A classMethod: m   ;

A Class new: B

: em | i | #[ loop: i [ ] ]     bench . ;

: fc | i | #[ loop: i [ f ] ]   bench . ;

: mf | i | #[ loop: i [ 5.0 m ] ] bench . ;

: mb | i b |
   B new ->b
   #[ loop: i [ b m ] ] bench . ;

: mi | i j |
   10 ->j
   #[ loop: i [ j m ] ] bench . ;

: ca | i | #[ loop: i [ A m ] ] bench . ;

: cm | i cl |
   A ->cl
   #[ loop: i [ cl m ] ] bench . ;
```

Results are :

	Total (ms)	Calls (ms)	/ direct call
1000000000 em	1690	0	0,00
1000000000 fc	1970	280	1,00
1000000000 mf	1970	280	1,00
1000000000 mb	2250	560	**2,00**
1000000000 mi	2530	840	**3,00**
1000000000 ca	1975	285	1,02
1000000000 cm	3378	1688	**6,03**

The first column is the total time in milliseconds. The second column is the total time for calls (ie subtracting the time spent for the empty loop). The third column is the cost of a the polymorphic call compared to a direct call.

#em is the benchmark for an empty loop.

#fc calls an empty function (f) n times. Of course, here, a direct call is compiled.

#mf test calls a message that is optimized during compile time into a direct call (because TOS is a literal). It runs in the same time as #fc

#mb calls a message that will not be optimized during compile time. It uses the dispatch code described in (3.3). This test shows that a polymorphic call takes twice the time compared to a direct call.

#mi calls a message that will not be optimized during compile time. It uses the dispatch code described in (3.3), but as the receiver is an integer, it uses the special case for integers/null (step 1). In this case, the polymorphic call takes three times the time for a direct call. This is probably the result of explicit jumps that breaks CPU optimizations.

#ca calls a class method on class A. It is optimized (because TOS is a literal) and runs in the same time than a direct call.

#cm calls a message on class A that will not be optimized. The code to run is searched each time in the hierarchy, without VTM. Those calls are 3 times slower than MVT dispatch for methods and 6 times slower that a direct function call.

6 Future work

The main purpose of this dispatch implementation is to keep high performances while allowing extended classes.

Nevertheless, future work may be interesting on alternative mechanisms for MVT storage.

In a typical application :
- Some messages will be implemented only for one class and a big MVT will be allocated for only one pertinent slot (this is the worst pattern).
- Some messages will be implemented only at Object level and the MVT will be very small.
- Some messages will be in-between (#read, #+, #size , #<<, #log, ...) and the MVT will be partially filled.

In order to handle the first pattern, a possibility would be to have 2 sizes for each MVT : the minimum class index and the maximum class index. This would complexify a little the rumMessage function with a performance penalty, but save a lot of space when a message is implemented only for a few classes.

Other mechanisms have also been discussed in the literature ([ERTL11]) and could be implemented and benchmarked in a future work.

7 Conclusion

Oforth implements a full dynamic binding for method dispatch for methods, associated with extendable classes.

During compile time, some optimizations occur to reduce, when possible, messages call to direct call.

Virtual tables are "by message" and not "by class". They are not defined at compile time but calculated and reallocated at runtime, while messages are performed. This allows to save some memory (unused messages don't use MVT) and to extend classes.

This is done without suffering much performances penalties as the same code is used to manage MVT and to call methods : everything is done by calling to the addresses stored in the MVT slots.

8 References

[RP96] Bradford J. Rodriguez and W. F. S. Poehlman. A survey of object-oriented Forths. SIGPLAN Notices, pages 39–42, April 1996.

[DUC11] Roland Ducournau. Implementing statically typed object-oriented programming languages. ACM Computing Surveys, 43(3):Article 18, April 2011.

[ERT12] M. Anton Ertl. Methods in objects2: Duck Typing and Performance. 28th EuroForth Conference 2012.