Methods in objects2: Duck Typing and Performance

M. Anton Ertl[*]
TU Wien

Abstract

The major new feature of objects2 is defining methods for any class (like in Smalltalk): this means that we can have two classes that are unrelated by inheritance, yet react to the same messages and can be used in the same contexts; this is also known as duck typing. This paper discusses the implementation of method dispatch for these general selectors as well as the more restricted class selectors of the original `objects.fs`, and compares the memory and execution time costs of these method selector implementations: Unhashed general selectors are as fast as class selectors (down to two instructions), but can consume a lot of memory (megabytes of dispatch tables for large class hierarchies); hashed general selectors are significantly slower (≥ 43 cycles), but consume less memory. Programmers don't need to choose a selector implementation up front; instead, it is easy to switch between them later, on a per-selector basis.

1 Introduction

My `objects.fs` package provided Java-inspired facilities for defining methods: Essentially the programmer defines a method selector for a certain class or interface, and can then only define methods for this selector for descendent classes of that class, or (for interface selectors) for classes implementing this interface.

One disadvantage of this approach and the way it was implemented in `objects.fs` was that passing an object of the wrong class to a selector was not detected. Detecting this would be useful for debugging, and also useful for, e.g., implementing proxies that pass on every not-understood method call to another object.

Also, some people argued that Smalltalk-style methods, which can be defined for any class, would be useful. They would allow the use of *duck typing*: A type is defined as a set of selectors, and every class/object for which methods are defined for these selectors, has this type.

[*]Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; `anton@mips.complang.tuwien.ac.at`

At first the implementation of these features seemed to me too expensive in run-time, and the benefits did not appear to be very significant. But eventually I learned about more efficient implementation techniques as well as additional uses for these features, so I set out to devise objects2, which provides these features (as well as backwards compatibility with `objects.fs`).

In this paper I look at the basic syntax (Section 2), at various method dispatch techniques (Section 3 and their performance (Section 5), and at the minimally invasive ways offered by objects2 for selecting between dispatch techniques (Section 6). It also discusses (Section 4) how to implement the current object pointer and measures the resulting execution time (Section 5).

What this paper does not discuss whether you should use Smalltalk-style methods and duck typing or use than Java/C++-like methods. Objects2 gives you both options (with a very easy transition between them, and the choice available per selector), and it is up to you to decide which one you want or need to use. This paper also does not give a general documentation of objects2; the documentation comes with the package.

2 Defining methods

Figure 1 shows an example program that defines three classes: A, it's child A1, and the unrelated class B (apart from the common ancestor class `object`, which is unavoidable in objects2).

It also defines two method selectors: `foo` and `bar`; there are two method definitions for each of these selectors. The first method definition for a name defines the selector (a Forth word with that name), any further method definition just defines the method (an anonymous colon definition) and makes it the method that the selector calls for the current class and it's children.[1]

The `some-A1 foo` call demonstrates that A1 inherits the foo-A method from A. The `some-A bar` example demonstrates that objects2 reports if a selector is invoked for a class for which no method is

[1]This kind of conditional definition is very unusual in Forth; it is due to the fact that we we want to optionally use duck typing, so we don't want to have to define the selector beforehand, as was done in `objects.fs`).

```
object class
:: foo ." foo-A" ;;
end-class A

A class \ child class of A
:: bar ." bar-A1" ;;
end-class A1

A  heap-new constant some-A
A1 heap-new constant some-A1

object class
:: foo ." foo-B" ;;
:: not-understood ( sel-xt obj -- )
  ( sel-xt ) some-A1 swap execute ;;
end-class B

B  heap-new constant some-B

some-A1 foo \ prints foo-A
some-B  foo \ prints foo-B
some-A  bar \ method not understood
some-B  bar \ prints bar-A1
```

Classes

Selectors	object	A	A1	B
not-understood	nu-obj	nu-obj	nu-obj	nu-B
foo	udfoo	foo-A	foo-A	foo-B
bar	udbar	udbar	bar-A1	udbar

Figure 1: An example program and its class×selector matrix

defined for the selector (in contrast to `objects.fs`, which would just blindly try to `execute` some xt, with unpredictable results).

Finally, the `some-B bar` call demonstrates the `not-understood` feature: Any call to a selector for a class for which no method is defined results in a call to the `not-understood` method for this class. By default (i.e., inherited from `object`), `not-understood` just produces an error report (as demonstrated by `some-A bar`), but you can define your own method to deal with not-understood messages, and this is done here: The `not-understood` method for B just invokes the original selector (which is passed as xt) for `some-A1`, which eventually prints `bar-A1`.

Objects2 has a bunch of other features (e.g., for defining instance variables), but they do not play a role for the issues discussed in this paper, so they are not discussed here.

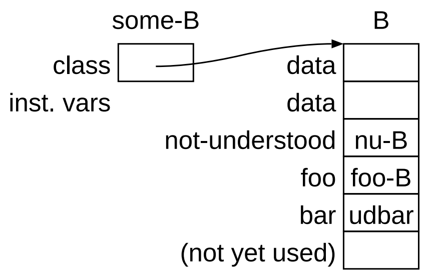

Figure 2: Object `some-B` and it's class `B`. `Some-B` has no instance variables

3 Dispatch Techniques

3.1 Unhashed general selectors

If any selector can be called with an object of any class, we have to implement a class×selector matrix. Figure 1 also shows the matrix for the example program. Some entries are defined directly by the programmer (e.g., the entry for `foo`×`A`), some are defined by inheritance (e.g., `foo`×`A1`); the rest gets the xt of the word `ud`*`selector`* (short for `undefined-`*`selector`*) which calls `not-understood` for the class and passes it the xt of *`selector`*.

In practice, instead of creating one big matrix, we store each column in the data of its class (see Fig. 2). Each object starts with a pointer to this class data. So the code for dispatching an unhashed method is:

```
: unhashed-selector ( u-offset "name" -- )
  create ,
does> ( ... object -- ... )
  ( object selector-body )
  @ over @ + ( object xtp ) @ execute ;
```

We cannot resize the class after objects of the class have been created: resizing might require moving the class data, i.e., updating the class pointers in the objects; since we do not track objects, we cannot do that. Therefore, we specify in advance how many unhashed selectors there are (see Section 6).

VFX Forth translates a call to such a selector into:

```
MOV  EDX, 0 [EBX]
ADD  EDX, [<selector-body>]
CALL 0 [EDX]
```

The memory access to the selector body cannot be optimized away by VFX, because the user is allowed to change the offset there at any time. However, it is possible to define selectors in a way that avoids that problem:

	Classes	Selectors	Cells
Minos	129	364	46956
GlForth	29	79	2291

Figure 3: Memory consumption of unhashed general selectors

```
: do-unhashed-selector ( object offset -- )
 over @ + ( object xtp ) @ execute ;

: unhashed-selector ( u-offset "name" -- )
 >r : r> postpone literal
 postpone do-unhashed-selector postpone ; ;
```

VFX compiles a call to such a selector into

```
MOV  EDX, 0 [EBX]
CALL [EDX+<u-offset>]
```

Unfortunately, this version is only fast on compilers that inline calls and optimize the result, like VFX.

Figure 3 shows the memory consumption of the dispatch tables of unhashed general selectors. For large programs the size of the dispatch tables can become a problem, because it grows approximately quadratically with the size of the program.

3.2 Hashed general selectors

Larger programs have more classes and more selectors, and usually the matrix is sparsely populated, i.e., most matrix entries point to ud*selector*. To save memory, we can use a hash table for looking up all the entries that are not ud*selector*; if no entry is found in the hash table, we call undef (a generic variant of ud*selector*), which eventually calls not-understood. As a key into the hash table, we can use an integer computed from a class index and a selector index: the class indices are spread so far apart that a class index can be just added to the selector index to get a unique key. Fig. 4 shows a hash table for our example.

The code for the hashed dispatch is:

```
does> ( ... object -- ... )
 @ ( ... object sel-id )
 over object-class @ class-base @ +
 ( object key )
 tuck hash-multiplier um* +
 ( key object hash )
 table-mask and 2* cells meth-hash-table +
 rot begin ( object table-entry key )
  over @ over = if \ right class/selector?
   drop cell+ @ execute exit then
  over @ 0= if
   nip undef exit then
  swap cell+ cell+ swap
 again ;
```

key	value
B::foo	foo-B
A1::foo	foo-A
B::not-understood	nu-B
A::foo	foo-A
A::not-understood	nu-obj
object::not-understood	nu-obj
A1::not-understood	nu-obj
A1::bar	bar-A1

Figure 4: A hash table for our example program

First this computes the key, then this key is hashed with a simple hash function, then we perform linear probing in the hash table, until the key matches our class/selector pair (then we execute the method), or until we find an empty entry (then we call undef). Note that the search loop is typically iterated very few times (ideally 0 times).

Whether the hashed or the unhashed version is preferred depends on the memory and run-time requirements of the application. E.g., if we assume that each selector in Minos has four methods on average, and that these methods are inherited to four classes on average, then we have 5824 entries in our hash table. We need either an 8K entry (16K cell) hash table with a 71% load factor (which may be slow), or a 16K entry (32K cell) hash table with a 36% load factor, but that does not save much memory compared to unhashed selectors.

Objects2 gives you the option of using the unhashed access for the most frequently used selectors (also useful for selectors that have methods for most classes), and hashing for the rest; see Section 6. By using unhashed selectors for the most frequent selectors, the relatively high load factor of the smaller hash table becomes acceptable, because only infrequent selectors are hashed; also, there are now fewer entries in the hash table, so the load factor is reduced somewhat.

```
class-selector u
class-selector v
class-selector w
class-selector x
object class
  :: u ." A-u" ;
end-class A
object class
  :: v ." B-v" ;
end-class B
A class
  :: w ." A1-w" ;
end-class A1
A class
  :: x ." A2-x" ;
  :: u ." A2-u" ;
end-class A2
```

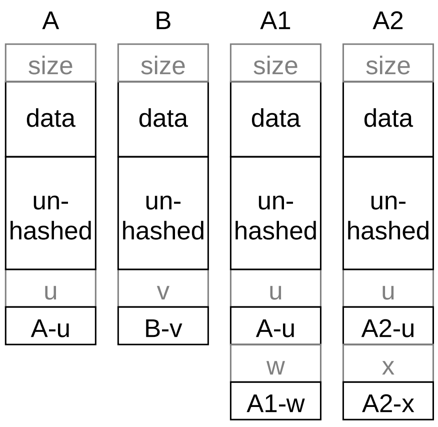

Figure 5: Class selectors. Different selectors (u and v, w and x) have the same index (optional checking data in gray)

3.3 Class selectors

Consider the following restriction: A selector can only be used on a specific class and its descendents. This means that two selectors for two non-overlapping classes (i.e., where neither class is descended from the other) can use the same index, resulting in densely populated dispatch tables (see Fig. 5) and lower memory consumption. We call these selectors *class selectors*.

We define classes starting with the most ancestral ones, and define all class selectors before we define child classes; this allows a very simple management of the selector indices: Every class has a current maximum selector index; defining a new class selector increases the maximum, thus creating an index for the new class selector. A child class inherits the maximum from its parent (and the parent's maximum stays the same from then on).

The dispatch code for class selectors without checking is the same as for unhashed selectors. The difference is in the index management and in the resulting restrictions: We have a limited number of unhashed selectors (the number is specified when loading objects2, see Section 6), whereas the class selectors are unlimited, but must satisfy the class selector restriction. In objects2 the indices of the class selectors start right after the indices of the unhashed selectors.

Using a class selector on an object of the wrong class will call the wrong method, or whatever is found at the class selector's offset from the start of the class; if we are lucky, we get a crash right away, if we are unlucky, the program does something we don't want.

We can have class selectors that check whether they are invoked for the right class: In addition to the method xt, we store the body address of the selector and the size of the class structure (shown in gray in Fig. 5). The selector then checks that its offset is within the class, and that what is stored right before the method is its body address. If not, the selector can produce an error (useful if checking is turned off after debugging) or perform not-understood processing. The selector code for the latter case is:

```
does> ( ... object -- ... )
 ( object sel-body )
 dup last-class-selector !
 tuck @ over object-class @
 ( sel-body object offset class )
 2dup class-size @ u< if
  ( sel-body object offset class )
  + rot over @ = if ( object p )
   cell+ @ execute exit
  then
  drop
 else
  2drop nip
 then ( object )
 last-class-selector @ cell+ @
 message-not-understood1 ;
```

With the parameters above (364 selectors, each with 4 methods that are inherited to 4 classes on average), class selectors consume 5824 cells of dispatch tables without checking and 11648 cells with checking. However, to work around the class selector restriction, programmers are likely to create deeper inheritance hierarchies and define selectors higher in the hierarchy, so the memory savings of class selectors are probably less than would be expected from the simplistic calculation above. The extreme variant of this would be to have a common ancestor class for all classes and define all selectors there, so all selectors are available for all classes, but with the same memory consumption as unhashed general selectors (for the unchecked version; checking is not necessary in this case).

```
interface
  selector i1
  selector i2
end-interface I
interface
  selector j1
  selector j2
end-interface J

object class
  implements I
  :: i2 ." A-i2" ;
end-class A
object class
  implements J
  :: j1 ." B-j1" ;
end-class B
B class
  implements I
  :: i1 ." B1-i1" ;
end-class B1
```

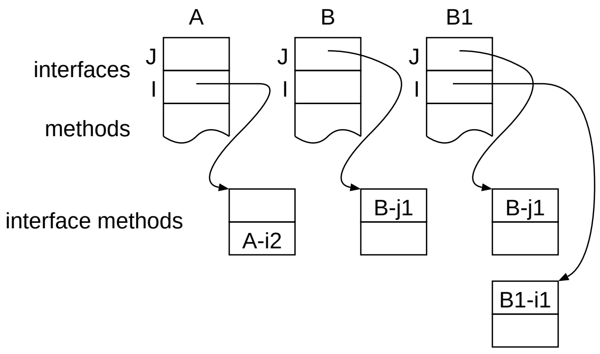

Figure 6: Interfaces and their implementation

3.4 Interface selectors

The original `objects.fs` did not have general selectors, so it included interfaces to make it possible to go beyond the limitations of class selectors: An interface is a set of selectors; you can define any class to support an interface, and the selectors of the interface can then be called for the class and its descendents. Figure 6 shows an example. The code for (unchecked) interface dispatch is:

```
does> ( ... obj -- ... )
 ( obj sel-body )
 2dup @ ( obj sel-body object if-offset )
 swap @ + @
 ( obj sel-body if-table )
 swap cell+ @ + @ execute ;
```

Here the selector stores (first cell) the offset of the interface from the class pointer, and (second cell) the offset of the method from the interface pointer, and uses both offsets to access the xt of the method.

Every interface requires a cell in every class (not just those that implement the interface); there can be several selectors per interface, so interfaces are somewhere between class selectors and unhashed general selectors in functionality and memory consumption.

Objects2 has general selectors, and interfaces do not appear to add enough to justify the additional complexity, so objects2 emulates interfaces with general selectors (for compatibility with `objects.fs`).

3.5 Monomorphic selectors

Sometimes a programmer defines a method (and, implicitly, a selector) to keep the program flexible, but does not define another method for the selector for now. Then the selector is actually used monomorphically, and dispatch can be very simple:

```
does> ( ... object -- ... )
 @ execute ;
```

In other words, a monomorphic selector is a deferred word (this version does not check that only descendents of the class for which the method was defined are passed to the selector).

4 Implementing the current object pointer

Apart from method dispatch, there is another interesting implementation issue:

Like `objects.fs`, objects2 has a current object `this`. `This` is set on method entry from the top of stack, and is visible inside the method.

This sounds like an ideal use for locals (in particular, `(local)`), but there is one catch: Standard programs must not use more than one locals definition per colon definition. And unfortunately there are Forth systems like VFX that rely on and enforce this restriction. So if we use locals for `this`, the programmer cannot use locals inside methods.

The other alternative is to define `this` as a value. The disadvantages are that this approach requires hardening against exceptions which may be difficult in some cases, and multi-tasking would require a user value (or user variable), which may be slower than global values.

Another property of the value implementation is that it allows us to access instance variables from outside methods; this has benefits in debugging, and can also be used to access instance variables from ordinary colon definitions, which can be used for factoring or for converting non-object-oriented code to object-oriented code. This kind of usage also has dangers, and some may prefer a local `this` because it prevents this usage.

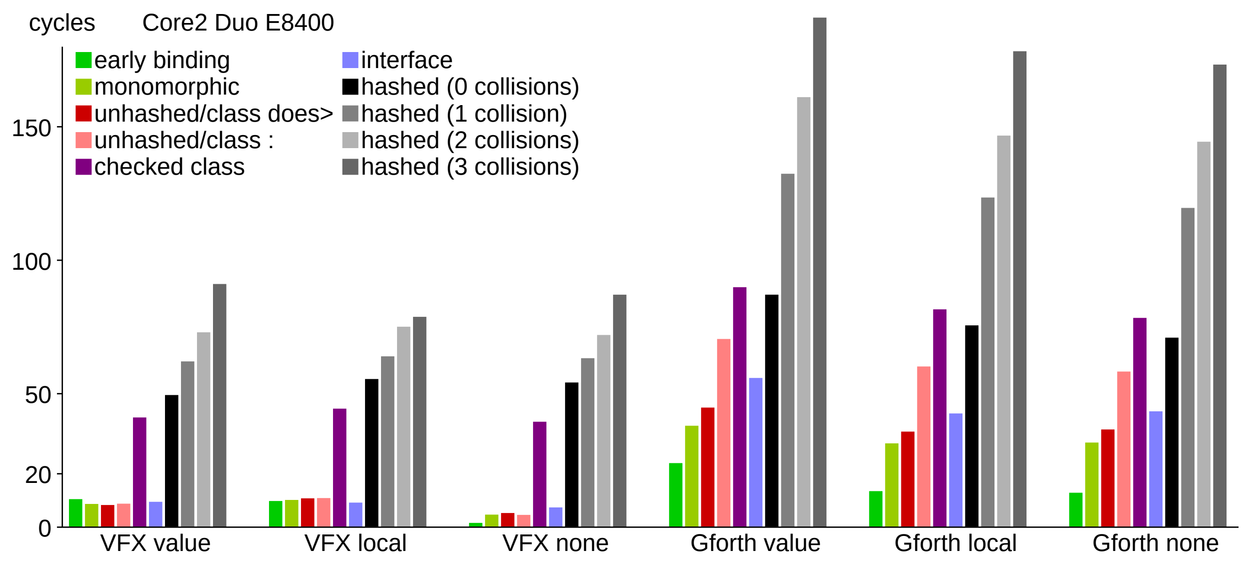

Figure 7: Time for one method call (plus overhead) in the micro-benchmark, varying dispatch code, Forth system, and `this` implementation

5 Execution time results

To compare the run time of various method dispatch (and current object pointer) techniques, I wrote a microbenchmark. It's a simple loop whose body calls the same selector 10 times in a row, and the object for which it is called is always the same (the method pushes that on the stack). I.e., caches should be hit and indirect branch prediction should be optimal.[2] The called method just increments the top-of-stack and pushes an object on the stack, but of course it does the handling of `this` (except for the *none* variant, which just `drop`s the object from the stack and pushes the object again). The shown times include the loop overhead around the selector calls. We also compare with *early binding*, where the method is compiled directly into the loop (and VFX inlines it) instead of going through some selector code. For the hashed selector, four timing variants are measured (by initializing the hash table appropriately): with 0, 1, 2 and 3 collisions (for this particular lookup) when probing the hash table; for a load factor of 50% (the value I recommend), most lookups should have zero or one collision.

Figure 7 shows the times in cycles per iteration, on a 3GHz Core2 Duo E8400. Two different Forth systems are used: VFX, an analytic native-code compiler that produces fast code for straight-line code; and Gforth, a system with a simpler code generation strategy (concatenate C-compiler-generated code fragments). Also, the two variants of implementing `this` are compared with each other, and for perspective, we also compare with not having `this` and (in this case) just dropping the object passed into the method.

[2]That's not realistic, but indirect branch prediction should affect every technique in the same way and cache misses should be relatively rare for frequently-executed code; and rarely executed code does not have a significant influence on performance.

All dispatch techniques except checked class selectors and hashed selectors have about the same performance on VFX, except that early binding is particularly fast for the *none* case, because VFX manages to optimize most of the loop body away: it inlines all the calls, and then VFX optimizes nearly all the dropping and pushing of the object away, leaving just the increments.

Interestingly, even though the VFX code for the unhashed selector using `:` looks much better than when using `does>`, this is not reflected consistently in the timing data.

Looking at the other dispatch techniques, on VFX a value `this` costs about 3–4 cycles more than no `this` and a local `this` costs about two more cycles. With more substantial methods, an out-of-order CPU like the Core 2 Duo will probably overlap the `this`-handling overhead with other code, reducing the cost for `this` even further.

The checked class selelector and the hashed selector are quite a bit slower on VFX: 30–35 cycles slower for the checked class selector, 43–50 cycles slower for the hashed selector with zero collisions. Each collision adds 9–10 cycles on average. The difference from the other selectors is surprisingly large, especially given the high speed of the other selectors. I believe this is mainly due to the fact that VFX's register allocation is limited to straight-line code (the other selectors all perform straight-line code). Hardware optimizations in the CPU might also play a role, even though the benchmark was modified so that the loop stream detector [Int12]

```
\ 3 unhashed selectors
   3 constant objects2-unhashed-selectors
\ hash table size: 2048
  11 constant objects2-hash-table-shift
\ warn if >1200 methods in hash table
1200 constant objects2-max-occupation

require objects2.fs

\ declare three selectors, such that they
\   are unhashed
selector draw
selector foo
selector bar
\ declare class and monomorphic selectors
class-selector baz
monomorphic-selector flip

\ load class libraries
require graphical.fs \ graphical class
require wine.fs      \ class about wine

\ load application code
require bla.fs
require blubb.fs
```

Figure 8: Choosing the implementation of selectors

should not come into play.

Unlike VFX, Gforth shows differences in performance between early binding, monomorphic, unhashed, and interface selectors, with the unhashed selector (implemented with `does>`) being 21–24 cycles slower than early binding. Class selectors using `:` as shown are significantly slower, because Gforth does not inline. The extra cost for checked class selectors is 40–45 cycles, and hashed dispatch without collisions costs 32-40 cycles more than unhashed dispatch, and each collision adds 34 cycles on average.

In Gforth a local `this` is faster than a value `this` by about 10 cycles, and not dealing with this is another 0–12 cycles faster.

Comparing unchecked class and interface selectors (from `objects.fs`) with unhashed and hashed selectors (new in objects2), we see that the added flexibility of the objects2 selectors either costs space (for the unhashed selector) or time and not as much space (hashed selector). Whether these costs are acceptable and whether the flexibility is worth the cost depends on the application and its environment. One of the features of objects2 is that it is easy to switch between these different selector variants, on a per-selector basis, as discussed in the following section.

6 Optimizing Dispatch

Objects2 offers the choice of using unhashed or hashed general selectors, class selectors, or monomorphic selectors. Moreover, you can make the choice on a per-selector basis, in a minimally invasive way: You do not need to change the class or method definitions, which may be libraries which you may not want to change. Instead, you can specify in the load file of the application which selectors use which dispatch implementation. The rest of this section describes this feature.

By default selectors are general selectors. The first n selectors are unhashed, the rest is hashed. You can determine the unhashed selectors by setting n before loading `objects2.fs` and then declaring the n selectors that you want to be unhashed.

You can also set the number of classes and the hash table size to reduce the memory consumption to the necessary amount or to allow more than the default number of classes and hash table entries.

You can also declare a selector as a class selector or monomorphic selector in the load file.

Here is an example of how a load file might look:

Note that these selector declarations happen outside any class, they just influence what the later method definitions do. Actually, implementation-wise, these “declarations” define a selector word of the desired kind, and a later method definition essentially just defines the method and inserts the xt into the appropriate table for this selector and class; in case of a class selector the first method definition inside a class also sets the root class for this selector, and every other method definition has to be in a descendent class of that class.

7 Missing language features

There are two language features that would be useful for implementing objects2 and which Forth systems usually provide in some way, but which are not standardized:

`body> ( addr -- xt )` would allow getting the xt of the selector for not-understood processing. But since this is not standardized, every selector has to store its xt in an extra field. Absence cost: one extra cell per selector.

`>definer ( xt -- definer )` would allow to check if a word has been defined as a selector or not, and which kind of selector. But since this word is not standardized, objects2 maintains several linked lists, one for each kind of selector, and if it needs to know the information, it searches these linked lists. Absence cost: another extra cell per selector, more CPU consumed by linear searches.

8 Related work

There is a large body of work on implementing object-oriented languages in general and method dispatch in particular [ACFG01, DH96, VH96, AGS94, ZCC97]. Ducournau [Duc11] presents a very good survey, but it helps to be familiar with some of the implementation techniques in order to understand this survey. The present work does not introduce any new techniques; instead, it makes a few of the existing techniques available to Forth programmers in a way that allows them to switch between different techniques easily, as appropriate for the application.

There have also been a number of object-oriented Forth extensions. Rodriguez and Poehlman [RP96] list 23, and since then more have been introduced, including `objects.fs` [Ert97]. This paper focuses on the main feature where objects2 differs from `objects.fs`: Allowing selectors to be used on arbitrary classes; it discusses the implementation of this feature and presents performance data.

A relatively recent entry in the collection of object-oriented Forth extensions is FMS (Forth meets Smalltalk). Like objects2, it supports defining methods for a given selector for any class. The implementation is not much documented, but seems to be based on a compressed table. I do not understand the table format enough to evaluate its space consumption. The dispatch code is relatively long compared to the variants shown above, so I expect it to take at least as much time as objects2's hashed dispatch with 0 collisions. A more substantial comparison is future work.

9 Conclusion

Fast, small, flexible (duck typing): Pick any two.

Fast, small: Class selectors, monomorphic selectors

Fast, flexible: Unhashed general selectors

Small, flexible: Hashed general selectors

Objects2 allows you to choose between these method selector implementations. Moreover, you can choose separately for each selector, and you can change the choice easily, in many cases not even touching the actual class code.

References

[ACFG01] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. Efficient implementation of java interfaces: Invokeinterface considered harmless. In *Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '01)*, pages 108–124, 2001.

[AGS94] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-method dispatch using compressed dispatch tables. *ACM SIGPLAN Notices*, 29(10):244–244, October 1994.

[DH96] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in C++. In *Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96)*, pages 306–323, 1996.

[Duc11] Roland Ducournau. Implementing statically typed object-oriented programming languages. *ACM Computing Surveys*, 43(3):Article 18, April 2011.

[Ert97] M. Anton Ertl. Yet another Forth objects package. *Forth Dimensions*, 19(2):37–43, 1997.

[Int12] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, April 2012. Order number 248966-026.

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

[VH96] Jan Vitek and R. Nigel Horspool. Compact dispatch tables for dynamically typed object oriented languages. In Tibor Gyimóthy, editor, *Compiler Construction (CC'96)*, pages 309–325, Linköping, 1996. Springer LNCS 1060.

[ZCC97] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient dynamic dispatch without virtual function tables. the SmallEiffel compiler. In *Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '97)*, pages 125–141, 1997.