Code "bumming"for testing membership. A Lisp example

Richard Fateman

June 10, 2009

Abstract

Code "bumming" to get the smallest possible program, which was in the past often equivalent to getting the fastest code, is an intellectual challenge, even as its exact relevance can be questioned. Some code is better, even if it is longer, if it uses fewer jump instructions, or more of the conditional jump instructions fall through, or if subroutine calls are avoided. While it is easy to make generalizations about what makes a computer program fast, the devil is in the details, and each hardware and software system may have its own details. Nevertheless, we were provoked into trying to do some micro-optimizations, and show the results here, using Lisp, and a simple function that tests for membership in a list.

1 Motivation

In a large program in the core of the Lisp-based computer algebra system Macsyma[1], there are frequent uses of a program called `memq`. This is the name of a function from the original host system for Macsyma, Maclisp, and is no longer a separate built-in function in ANSI Common Lisp, but has been incorporated into a more general `member` function. The special case of `memq` can be easily defined in several ways. In fact, it was determined from program profiling of Maxima, that a substantial amount of time was spent in calling `memq` which in turn called (`member`) with an extra argument to use `eq` for testing equality, as is required by `memq`.

2 Clawing through simple code to get speed

One programmer suggested[2] textually substituting `member` for `memq`. This idea could be promoted for two reasons:

- It relieves the programmer of the intellectual burden of knowing about `memq` and `member` both.
- The program is relieved of one layer of subroutine call.
- If the Lisp compiler is clever enough to optimize the `member` call, perhaps it will be reduced to in-line code, and could be faster.

Alternatively, we might assume the compiler is not particularly clever (While there are certainly clever Lisp compilers, it is hardly a requirement!), and that we could do better by defining `memq` more carefully ourselves. From first principles, using the typical Lisp style of recursion, we can write this:

```
;; if element is in thelist, return the sublist with element first, else nil.

(defun memq(element thelist)
   (if (null thelist)  ; is the list empty?
        nil            ; if so, return nil
```

[1] or its open-source Maxima version

[2] in fact, went ahead and did the editing.

```
        (if (eq (first thelist) element)  ; is the first entry in the list the same as element?
            thelist                       ; if so, return that sublist
            (memq element (rest thelist))))) ; not found -- keep looking

;; examples: (memq 'b '(a b c)) --> (b c)
;;           (memq 'x '(s r v u w)) --> nil
```

You might object to this code on the basis that "recursion is slow", and rewrite it as a loop (see later). An easy compiler optimization (often used in Common Lisp, and *required* in the Scheme dialect of Lisp) changes this recursion into a loop anyway. Another objection might be that it has too many comments, obscuring the obviousness of the code.

For reference, here's the version of `memq` which is based on the fact that Common Lisp has a more general function. We have also shortened the names of the parameters.

```
(defun memq (a b) (member a b :test 'eq))
```

If you are eager to write a loop instead of using recursion, here's one that takes advantage of the pun that an empty list in Common Lisp is also the nil or false value. Thus `(while b ...)` can be used to check to see if `b` is the empty list.

```
(defun memq(a b)
     (while b (if(eq x (first b))(return b)(pop b))))
```

If you are unsure of exactly how `while` works, or what `pop` does, you can use primitives like `label` and `goto`, which is the language used to expand `while` into a lower level of code, before it is compiled.

```
(defun memq(a b)
    (prog ()
     thetag
      (if (or (null b)(eq x (first b)))(return b)(setq b (rest b)))
      (go thetag))))
```

Can we make this faster? One way is to unroll the loop so that the jump back to the beginning happens less often. Easily done like this:

```
(defun memq(a b) ;unroll the loop
(declare(optimize(speed 3)(safety 0)))
  (block nil
    (tagbody
     thetag
      (if (or (null b)(eq x (first b)))(return-from nil b)(setq b (rest b)))
      (if (or (null b)(eq x (first b)))(return-from nil b)(setq b (rest b)))
      (if (or (null b)(eq x (first b)))(return-from nil b)(setq b (rest b))) ;etc
      (go thetag))))
```

We have written a macro program so that we can write `(memq x y :unroll 3)` to generate such code. Indeed our benchmarking suggests a modest level of unrolling is probably a good idea.

Is there any way to make this any faster? As it turns out, the typical use of `memq` in Macsyma/Maxima has a high probability of returning after just a few comparisons (say 3), either because the element occurs early in the list, or the list in its entirety is fairly short. Given that typical usage, maybe we should not be calling the function `memq`, but we should be expanding it in-line. This can generally be done quite directly in Lisp. All Common Lisp compilers allow a function to be declared `inline`, and some (not all) expand such functions when possible. All Common Lisp compilers allow for macro definitions which are necessarily expanded in-line, and these can be quite simple. The simplest is to expect the Lisp compiler to do the right thing in inline expansion of `member`, at least when optimization is set to highest speed.

```
(defmacro memq (a b) `(member ,a ,b :test 'eq))
```

Here's another version, illustrating yet another looping construct, the Lisp `do`. It illustrates a bit more thoroughly that writing macros is different from writing ordinary programs in Lisp. A good way of checking on what happens is to try `macroexpand`. Here's a macro definition and what it expands into:

```
(defmacro memq(a b)
  (let ((h (gensym))(i (gensym))) ;set up 2 new variables
  `(do ((,h ,a)
    (,i ,b (rest ,i)))
       ((null ,i) nil)
  (if (eq ,h (first ,i))(return ,i)))))

;; now try it out, and see the expansion of gensym and do.

(pprint (macroexpand '(memq x y))) -->

(block nil
  (let ((#:g2119896 x) (#:g2119897 y))
    (tagbody
      #:Tag246
        (cond ((null #:g2119897) (return-from nil (progn nil))))
        (tagbody
            (if (eq #:g2119896 (first #:g2119897))
                (return #:g2119897)))
        (psetq #:g2119897 (rest #:g2119897))
        (go #:Tag246))))
```

A very elaborate version which also allows unrolling to be specified, but is otherwise the same as the above program is this:

```
(defmacro mmk (x y &key (unroll 1)) ;; memq with unrolling of loop :unroll times
  (labels ((k-copies(r k)(let ((ans nil))
                           (dotimes (i (max k 1) ans)(setf ans (append r ans)))))
           ) ;return a list of k copies of the list r appended, at least 1
  (let* ((elem (gensym))(thelist (gensym))
         (r (k-copies `((cond ((null ,thelist) (return-from nil nil))
                              ((eq ,elem (car ,thelist))
                               (return ,thelist)))
                        (setq ,thelist (cdr ,thelist)))
                      unroll)))
    `(block nil
       (let ((,elem ,x) (,thelist ,y))
         (tagbody
          thetag
          ,@r
          (go thetag)))))))
```

Yet another version can be written if we don't care about providng the semi-predicate aspect of `memq`. That is, `memq` is normally defined to return `nil` if the element is not present and otherwise returns *the sublist beginning with the found element*. A pure predicate would, in the case of a found element, just return *true* or `t`. If we are content with such a pure predicate, and furthermore, we have a small constant list to test

against, what we can do is expand the test into a construction as shown in this example, a sequence of compare/branch operations.

```
(macroexpand ' (memq x '(a b c d e f g h i j k)))) -->
  (cond ((or (eql 'a x) (eql 'b x) (eql 'c x) (eql 'd x) (eql 'e x)
             (eql 'f x) (eql 'g x) (eql 'h x) (eql 'i x) (eql 'j x)
             (eql 'k x))
         t)) ; using eql here with a quoted symbol is same as eq.

;or

(case x ((a b c d e f g h i j k) t)) ; which in AllegroCL expands to the above.

;e.g.
(defmacro memq(a b) `(case ,a ((,b) t))) ;2nd arg is list of consts.
```

Is any of this hacking worth the effort? To some extent it depends on how clever your compiler really is. If you have a byte-coded interpreter, it is unlikely that any code you write will be faster than the implementer's version of member. If you have a very clever compiler, it might similarly be the case that you won't do better than what is built in, because the compiler would take note of possible optimizations and do them for you. It is unusual in a language implementation for a user to be able to tell the compiler, in such detail, how to generate code. For Allegro 8.0, specifying the optimization generates a pretty good inline code loop, although we can do slightly better by unrolling.

```
(defmacro memq (a b)
 `(let ()(declare (optimize (speed 3)(safety 0))
         (member ,a ,b :test 'eq))))
```

How do we know all this?

In Common Lisp there is a command to disassemble any piece of code. Here we also show yet another version, which we think makes most of the jump instructions "fall through" so it is perhaps faster than the second version. We have inserted comments by hand.

```
(defun memq(a b)
  (declare(optimize (speed 3)(safety 0)))
  (prog ()
   top
    (if b (unless (eq a (first b)) (setf b (rest b))(go top)))
    (return b)))

(compile 'memq)
(disassemble 'memq) -->

;; disassembly of #<Function memq>
;; formals: a b

;; code start: #x26dd9f94:
   0: 3b fa       cmpl  edi,edx       ;test b for nil
   2: 74 0c       jz    16            ;usually drops through
   4: 8b 5a ef    movl  ebx,[edx-17]  ;(first b)
   7: 3b c3       cmpl  eax,ebx       ; (eq a (first b))?
   9: 74 05       jz    16            ; usually drops through
  11: 8b 52 f3    movl  edx,[edx-13]  ; (rest b)
```

```
  14: eb f0       jmp   0             ; loop back 14 bytes
  16: 8b c2       movl  eax,edx       ; return sequence
  18: f8          clc
  19: 8b 75 fc    movl   esi,[ebp-4]
  22: c3          ret
  23: 90          nop

;; compare to this

(defun memq (a b) (member a b :test 'eq))
(compile 'memq)
(disassemble 'memq) -->

;; disassembly of #<Function memq>
;; formals: a b

;; code start: #x26b9bc04:
   0: 3b fa       cmpl  edi,edx
   2: 75 07       jnz   11           ;usually branches
   4: 8b c7       movl  eax,edi      ;return sequence
   6: f8          clc
   7: 8b 75 fc    movl   esi,[ebp-4]
  10: c3          ret
  11: 8b 5a ef    movl   ebx,[edx-17] ;(first b)
  14: 3b c3       cmpl  eax,ebx      ;(eq a (first b)) ?
  16: 75 05       jnz   23           ;usually branches
  18: 8b c2       movl  eax,edx      ;return sequence
  20: f8          clc
  21: eb f0       jmp   7            ;go to end of ret
  23: 8b 52 f3    movl   edx,[edx-13]  ;(rest b)
  26: eb e4       jmp   0             ;usually branches
```

Not being an expert on intel i486 instructions, these are still fairly readable: `clc` is clear-carry bit, `cmpl` is compare long, `movl` is move long.

A generally more useful tool for figuring out optimizations is a profiler: someone (else) determined through profiling that `memq` was a "hot spot" in Maxima, leading to interest in speeding it up. It is not plausible to profile a 23-byte assembly language program with no function calls with the same kind of call-counting tool, and so looking at assembler and (see below) timing provide more information.

3 Caveat: Instructions are not all: memory allocation matters

Superior performance depends on fast memory access, and for all but relatively trivial programs, memory allocation and deallocation. A production Lisp system has as a requirement that memory be reclaimed in an orderly and efficient manner, and that one does not have a memory panic – that is, filling up all of memory, paging space, etc., unless in fact all memory has been allocated and is in use. Programs that run only so long as a benchmark is being run, and then are killed without looking at the allocations can give the illusion of being efficient by leaving fragmented or bloated memory; a real application would have to clean that up to continue computing. Thus any fair benchmark should look at memory high-water marks as well as final memory allocation before exit.

In the case of member-testing, no memory is really allocated, so this is not an issue.

4 Benchmarks on a particular computer and a particular Lisp

Our first benchmark looks for the 11th item of an 11-item list. This is repeated 100,000,000 times. Our first version takes about 2.2 seconds. We can speed it up, to 1.8 seconds by unrolling by 3, though oddly, unrolling by 2 slows it down slightly. Additional unrolling provides negligible speedup. If we observe in our optimization, that we have a constant (quoted) second argument, the time can be reduced to .82 seconds. If we do not need the semi-predicate and only true/false is needed, the time is reduced further to 0.5 seconds.

So it is possible to reduce the time by about a factor of 4.

The discrepancy between one version and another is smaller for a shorter list. Also the discrepancy is small if the element being sought is early on in the list. For a match at location 3, the initial run took about .6 seconds which can be reduced to .45 seconds (by 3-unrolling) and then to .22 (pure predicated, constant list.)

In this case we see an improvement of about a factor of 3.

The computer we used was a Pentium D 3.00GHz 3.0GB RAM, and Allegro Common Lisp 8.0. We were able to test two other Lisps, GCL 2.6.8 and SBCL 1.0.13, each also running on the same Windows XP system.

The simplest version in GCL is slightly faster than ACL, but unrolling the loop results in substantially *slower* times for 2 or 4: it doubles the time. However unrolling by 3 or 9 speeds it up slightly. SBCL was similar to GCL, and notably their compilers were clever enough to notice, in some of the test programs, that the value computed inside the loop is irrelevant and that it need not be computed. (SBCL was slightly cleverer, and noticed one more instance than GCL). They therefore removed it and in the process defeated the test by timing the "empty loop". Absent the optimization, SBCL timings were overall similar (within a few percent slower or faster) on our timing tests.

A slightly more elaborate program that defeats the optimization (by hiding the fact that the result is not used) changes the timing so that the results are comparable to Allegro, and appears to be about as tight as can be arranged. Unfortunately, GCL's `disassemble` function does not work; SBCL has a disassembler; its code is quite similar in size to Allegro, but with different arrangements of code.

5 Conclusions

While your mileage may vary, depending on the degree to which your main program is using `memq` as well as your hardware and software, we show that in some set of circumstances a factor of 3 or 4 is obtainable in this micro-benchmark. Using the built-in function is certainly not the best that can be done, if you know about your data. Lisp compiler-writers might improve the expansion of `(member ... :test)` according to the suggestions in this paper, if they are not already doing something similar.