How to make LISP go faster than C

Didier Verna[*]

Abstract

Contrary to popular belief, LISP code can be very efficient today: it can run as fast as equivalent C code or even faster in some cases. In this paper, we explain how to tune LISP code for performance by introducing the proper type declarations, using the appropriate data structures and compiler information. We also explain how efficiency is achieved by the compilers. These techniques are applied to simple image processing algorithms in order to demonstrate the announced performance on pixel access and arithmetic operations in both languages.

Keywords: Lisp, C, Numerical Calculus, Image Processing, Performance

1 Introduction

More than 15 years after the standardization process of COMMON-LISP [5], and more than 10 years after people really started to care about performance [1, 4], LISP still suffers from the legend that it is a slow language. Although perfectly justified back in the 60's, this idea has become obsolete long ago: today, LISP can run as fast, or even *faster* than C.

If this false idea of slowness is still widespread today, it is probably because of the lack of knowledge, or misunderstanding of the 4 key factors for getting performance in LISP:

Compilers While LISP is mainly known to be an interpreted language, there are very good and efficient compilers out there, that can deliver not only byte-code, but also *native machine code* from LISP source.

Static Typing While LISP is mainly known for its dynamically typed nature, the COMMON-LISP standard provides means to declare variable types statically (hence known at compile-time), just as you would do in C.

Safety Levels While dynamically typed LISP code leads to dynamic type checking at run-time, it is possible to instruct the compilers to bypass all safety checks in order to get optimum performance.

Data Structures While LISP is mainly known for its basement on list processing, the COMMON-LISP standard features very efficient data types such as specialized arrays, structs or hash tables, making lists almost completely obsolete.

Experiments on the performance of LISP and C were conducted in the field of image processing. We benchmarked, in both languages, 4 simple algorithms to measure the performances of the fundamental low-level operations: massive pixel access and arithmetic processing. As a result, we got at least equivalent performance from C and LISP, and even a 10% improvement with LISP code in some cases.

In this paper, we present the algorithms used for benchmarking, and the experimental performance results we obtained. We also demonstrate how to properly tune the corresponding LISP code by introducing the proper type declarations, using the appropriate data structures and compiler information, in order to get the announced performance.

2 Experimental Conditions

2.1 The Algorithms

Our experiments are based on 4 very simple algorithms: pixel assignment (initializing an image with a constant value), and pixel addition / multiplication / division (filling a destination image with the addition / multiplication / division of every pixel from a source image by a constant value). These algorithms, while very simple, are pertinent because they involve the fundamental atomic operations of image processing: pixel access and arithmetic processing.

[*]EPITA Research and Development Laboratory, 14–16 rue Voltaire, F-94276 Le Kremlin-Bicêtre, France. Email: didier@lrde.epita.fr

The behavior of the algorithms is further controlled by additional parameters such as image size, image type (integers or floating point numbers), image representation (linear or multidimensional arrays) *etc.* For the sake of conciseness, only a few pertinent results are presented here. Nevertheless, people interested in the precise parameters combination and benchmark results we obtained can find the complete source code and comparative charts of our experiments at the author's website[1].

In the remainder of this paper, we consider $800 * 800$ integer or floating point images represented as 1D arrays of consecutive lines only.

2.2 The Protocol

The benchmarks have been generated on a Debian GNU/Linux[2] system running a packaged `2.4.27-2-686` kernel version on a Pentium 4, 3GHz, with 1GB RAM and 1MB level 2 cache. In order to avoid non deterministic operating system or hardware side-effects as much as possible, the PC was freshly rebooted in single-user mode, and the kernel used was compiled without symmetric multiprocessing support (the CPU's hyperthreading facility was turned off).

Also, in order to avoid benchmarking any program initialization side-effect (initial page faults *etc.*), the performances have been measured on 200 consecutive iterations of each algorithm.

3 C Programs and Benchmarks

For benchmarking the C programs, we used the GNU C compiler, GCC [3], version 4.0.3 (Debian package version 4.0.3-1). Full optimization is obtained with the `-O3` and `-DNDEBUG` flags, and by inlining the algorithm functions into the 200 iterations loop. It should be noted however that the performance gain from inlining is almost negligible. Indeed, the cost of the function calls is negligible compared to the time needed to execute the functions themselves (*i.e.* the time needed to traverse the whole images).

For the sake of clarity, a sample program (details removed) is presented in listing 1: the addition algorithm for `float` images.

Figure 1 presents the execution times for all C algorithms on both integer and floating point, $800 * 800$

```
void add (image *to, image *from, float val)
{
  int i;
  const int n = ima->n;

  for (i = 0; i < n; ++i)
    to->data[i] = from->data[i] + val;
}
```

Listing 1: `float` Pixel Addition, C Version

Algorithm	Integer Image	Float Image
Assignment	0.29	0.29
Addition	0.48	0.47
Multiplication	0.48	0.46
Division	0.58	1.93

Figure 1: Execution Times (s), C Implementation

images, and 200 consecutive iterations. These results will serve as a reference for comparison purpose with the upcoming LISP code.

The reader might be surprised by the performance of integer division, otherwise known as a costly operation. The explanation is that with inlining enabled, GCC is able to perform an optimization known as the "constant integer division" optimization [6], which actually removes the real division operation and replaces it by a multiplication and some additional (but cheaper) arithmetics.

4 First attempt at LISP Code

For testing LISP code, we used the experimental conditions described in section 2. We also took the opportunity to try several LISP compilers. For the sake of conciseness, benchmarks presented in the remainder of this paper are obtained with CMU-CL [4] version CVS 19c.

In this section, we describe our first attempt at writing LISP code equivalent to that of listing 1. This attempt is shown in listing 2.

```
(defun add (to from val)
  (let ((size (array-dimension to 0)))
    (dotimes (i size)
      (setf (aref to i) (+ (aref from i) val)))))
```

Listing 2: Pixel Addition, First LISP Version

The COMMON-LISP standard [5], besides the inevitable lists, provides more "modern" data types,

[1] `http://www.lrde.epita.fr/~didier/comp/research/`

[2] `http://www.debian.org`

[3] `http://gcc.gnu.org`

[4] `http://www.cons.org/cmucl`

such as structs, arrays and hash tables. Arrays allow you to store and access LISP objects according to a rectilinear coordinate system, and thus can be considered as the equivalent of `malloc`'ed or `calloc`'ed memory areas in C. The LISP function for creating arrays is `make-array`. In LISP, arrays can be multidimensional. On listing 2, you can see a call to `array-dimension` which retrieves the array's first rank size (no need to store this information in a data structure as in C). Remember that we are using 1D linear arrays to represent our images. The function for accessing array elements is `aref` and assignation is performed via `setf`. Unsurprisingly, `dotimes` is a macro performing a loop in a manner similar to the `for` language construct in C.

Running this function in a LISP interpreter shows that it is approximately 2300 times slower than the C version (system time and garbage collection excluded). The compiled version however is "only" 60 times slower than the equivalent C code. Finally, even with optimization turned on (see section 5.5), we are still 20 times slower than C.

To understand why we are getting this poor performance, one has to realize that our LISP code is *untyped*: contrary to C, the variables and function arguments we used could hold any LISP object. For instance, we use `array-dimension` on the function parameter `to`, but nothing prevents us from passing something else than an array; we perform arithmetic operations on the function parameter `val`, but nothing prevents us from passing something else than a number.

As a consequence, the compiled LISP code has to check *dynamically* that the variables we use are of the proper type with respect to the operations we want to apply to them. Our next step should then be to provide type information to the LISP compiler, just as we do in C.

5 Typing LISP Code

5.1 Typing mechanisms

The COMMON-LISP standard provides means to declare the expected types of LISP objects at compile time. It should be noted however that one is never *forced* to declare the type of a variable: types can be declared when they are known, or left unspecified otherwise. The LISP compilers are expected to do the best they can according to the information they have.

In a way, since the standardization of COMMON-LISP, it is not correct anymore to say that LISP is a dynamically typed language: it can be either dynamically or statically typed at the programmer's will.

There are different ways to specify types in COMMON-LISP. The first one is by passing specific arguments to functions. For instance, if you want to create an array and you know that this array will only contain single precision floating point numbers, you can pass the `:element-type` keyword parameter to the function `make-array` like this:

```
(make-array size :element-type 'single-float)
```

The next way to specify types is by means of "declarations": this mechanism is used to precise the type of a function parameter or a freshly bound variable. A type declaration should appear near the first occurrence of the variable it refers to. Listing 3 shows the next version of our addition algorithm, with type declarations issued.

```
(defun add (to from val)
  (declare (type (simple-array single-float (*))
                 to from))
  (declare (type single-float val))
  (let ((size (array-dimension to 0)))
    (dotimes (i size)
      (setf (aref to i) (+ (aref from i) val)))))
```

Listing 3: Pixel Addition, Typed LISP Version

As you can see, we have declared the (expected) type of the 3 function parameters: two arrays of `single-float` values (in COMMON-LISP, "simple array" denotes the equivalent of C arrays; there are other kinds of arrays that we will not describe here), and a `single-float` parameter. The `(*)` specification indicates that the arrays are 1D, but the actual size is unknown (at compile-time). 2D arrays would have been described with `(* *)` for instance.

A third way to provide type declarations will be described in section 6.2.

5.2 Objects Representation

To understand why type declarations matter for optimizing LISP performance, one has to realize the implications of dynamic typing a bit. Since LISP objects can be of any type (worse: of any *size*), the variables in LISP don't carry type information. Rather, the objects themselves are responsible for providing their own type. This means that most of the time, LISP objects contain type information plus a pointer to the real value. And obviously, pointer (de)referencing is causing a major performance loss.

Provided that type information is available, several techniques to avoid pointer representation can be used [1]. Two of them are described below.

5.3 Array Storage Layout

The first example deals with the way Lisp objects are stored in arrays: if the Lisp compiler knows that an array only holds floating point numbers for example, it can store the values directly in the array, in native machine format (just like in C), instead of storing pointers to them. A special version of the `aref` function can then be used (and even inlined) in order to access the values directly at the correct offset, instead of dereferencing pointers. This process of specializing functions instead of using generic ones is called "open-coding" in the Lisp community.

5.4 Immediate Objects

Our second example is that of "immediate objects", that is, objects that are small enough to be representable without indirection (on a single memory word). In a modern Lisp implementation, not all values are valid pointers to Lisp objects: typically, the 3 least significant bits are reserved for type information. In Common-Lisp, the standard integer type is called "fixnum". The Cmu-CL compiler [3] represents them as memory words ended by two zeroed bits. This effectively gives a 30 bits precision on a 32 bits machine. Most operations on fixnums can be performed directly; only a few of them require bit shifting, which is a small overhead anyway.

Now, let us examine the assembly code generated by Cmu-CL for a simple `(dotimes (i 100) ...)` loop. This is shown in listing 4.

The interesting lines appear with a gray background. They correspond respectively to the incrementation of the index, and the comparison with the upper limit (100). Here, we realize that the compiler has adapted the values in order to work directly on the shifted integer representation, hence, as fast as with "real" machine integers.

We see that in order to achieve good performance, the Lisp compilers try to, and usually *need* to be especially smart and implement all sorts of elaborated optimizations.

```
58701478:           .ENTRY FOO()
      90:           POP      DWORD PTR [EBP-8]
      93:           LEA      ESP,  [EBP-32]
      96:           XOR      EAX,  EAX
      98:           JMP      L1
      9A:  L0:      ADD      EAX,  4
      9D:  L1:      CMP      EAX,  400
      A2:           JL       L0
      A4:           MOV      EDX,  #x2800000B
      A9:           MOV      ECX,  [EBP-8]
      AC:           MOV      EAX,  [EBP-4]
      AF:           ADD      ECX,  2
      B2:           MOV      ESP,  EBP
      B4:           MOV      EBP,  EAX
      B6:           JMP      ECX
```

Listing 4: `Disassembly of a dotimes loop`

5.5 Optimization

In order to evaluate the gain from typing, we have to say a word about optimization first: the Common-Lisp standard defines "qualities" that the user might be interested in optimizing. The optimization level is usually represented by an integer between 0 and 3, 0 meaning that the quality is totally unimportant, 3 meaning that the quality is extremely important. Among those qualities are `safety` (that is, run-time error checking) and `speed` (of the object code).

Note that optimization qualities can be set globally, by means of *declamations*, but also on a per-function basis, by means of *declarations* similar to the type declarations we saw earlier. Here is how you would globally request fully safe and instrumented code for instance:

```
(declaim (optimize (speed 0)
                   (compilation-speed 0)
                   (safety 3)
                   (debug 3)))
```

When instructed to produce safe code, a modern Lisp compiler will treat type declarations as assertions and trigger an error when values are not of the expected type; this run-time checking process takes time. When instructed to produce fast and unsafe code however, the same compiler will "trust" the provided type declarations and activate all sorts of optimizations, like avoiding the use of pointer representation for Lisp objects when possible, open-coding functions *etc.*. As a consequence, the behavior is undefined if the values are not of the expected type, just as in C.

Our experiments show that completely safe Lisp code runs 30 times slower than the equivalent C code. On

	Integer Image		Float Image	
Algorithm	**C**	**Lisp**	**C**	**Lisp**
Assignment	0.29	0.29	0.29	0.29
Addition	0.48	0.48	0.47	0.46
Multiplication	0.48	0.48	0.46	0.45
Division	0.58	1.80	1.93	1.72

Figure 2: Execution Times (s), All Implementations

the other hand, figure 2 presents the results for fully optimized (unsafe) code. For reference, results from C code are also redisplayed.

In the case of integer images, we see that C and Lisp perform at *exactly* the same speed, apart from the case of division which is 3 times slower in Lisp. After further investigation and disassembly of the generated binary code, it appears that none of the tested Lisp compilers are aware of the constant integer division optimization that gcc is able to perform, so they use the slow `idiv` instruction of the `x86` cpu family. This is regrettable, but it shouldn't be too hard to improve those compilers on this particular case.

The case of floating point images, however, comes with a little surprise: we see that the first 3 algorithms run at identical, or slightly better (although not very significantly) in Lisp than in C. However, the division algorithm performs 10% faster in Lisp than in C. Other cases not presented here were also found in which Lisp performs significantly faster. These results should help us in getting C people's attention.

6 Type Inference

The mechanism by which typing helps the compiler optimize is not as simple as it seems. For instance, notice, on listing 3, that not all our variables were explicitly typed. Actually, two problems in this code were silently skipped until now: firstly, no information is provided about the array size, so the needed integer capacity for variables `size` and `i` is unknown. Secondly, the result of an arithmetic operation might not be of the same type as the operands.

When not all types are provided by the programmer, modern Lisp compilers try to *infer* the missing ones from the available information. Unfortunately (and this is a weakness of Common-Lisp), the standard leaves too much freedom on what to do with type declarations, so the type inference systems may differ in behavior and quality across the different compilers. As an illustration of this, two potential problems with typing are described below.

6.1 Loop Arithmetics

Without any type declaration provided, Cmu-CL is able to do inline arithmetics on the loop index of a `dotimes` macro, which explains why we did not provide an explicit type declaration for `i` in listing 3. However, when two `dotimes` loops are nested, Cmu-CL issues a note requesting an explicit declaration for the *first* index. On the other hand, no declaration is requested for the second one. The reason for this behavior is currently unknown (even by the Cmu-CL maintainers we contacted). In such a case, it is difficult to know if the bug lies in the type inference system or in the compilation notes mechanism. The only way to make sure that all possible optimizations are performed is to disassemble the suspicious code.

6.2 Arithmetic results

On the other hand, the type inference system of Cmu-CL has its strengths. Consider the integer multiplication algorithm, applied to integer images, in listing 5.

```
(defun mult (to from val)
  (declare (type (simple-array fixnum (*))
                 to from))
  (declare (type fixnum val))
  (let ((size (array-dimension to 0)))
    (dotimes (i size)
      (setf (aref to i) (* (aref from i) val)))))
```

Listing 5: Pixel Multiplication, Typed Lisp Version

It should be noted that the result of the multiplication of two fixnums might be bigger than a fixnum (a "bignum" actually). However, the type inference system of Cmu-CL notices that we store the result of this multiplication in an array declared as containing fixnums only. As a consequence, it assumes that we expect this multiplication to remain a fixnum, and continues to use optimized arithmetics.

Unfortunately, not all type inference systems are that smart. For instance, given the same code, the Allegro compiler (Acl)[5] will use a generic multiplication function which has the ability to return a bignum if needed, only at the cost of speed. Since we know that the result of the multiplication remains a fixnum, we have to tell that explicitly to Acl, by using another type declaration mechanism defined by the Common-Lisp standard, as illustrated by listing 6.

[5]`http://www.franz.com`

```
(defun mult (to from val)
  (declare (type (simple-array fixnum (*))
                 to from))
  (declare (type fixnum val))
  (let ((size (array-dimension to 0)))
    (dotimes (i size)
      (setf (aref to i)
            (the fixnum (* (aref from i) val))))))
```

Listing 6: Pixel Multiplication, Second Lisp Version

This is unfortunate, because it forces programmers to clutter the code with ideally non needed declarations, only for questions of portability.

7 Conclusion

In this paper, we explained how to achieve efficiency in Lisp by using the appropriate data structures, type declarations and optimization settings.

Since [1, 4], considerable work has been done in the Lisp compilers, to the point that equivalent Lisp and C code entails *strictly* equivalent performance, or even *better* performance in Lisp sometimes. We should also mention that when we speak of "equivalent C and Lisp" code, this is actually quite inaccurate. For instance, we are comparing a *language construct* (`for`) with a *programmer macro* (`dotimes`); we are comparing sealed function calls in C with calls to functions that may be dynamically redefined in Lisp *etc.*. This means that given the inherent expressiveness of Lisp, compilers have to be even smarter to reach the efficiency level of C, and this is really good news for the Lisp community.

Lisp still has some weaknesses though. We saw that it is not completely obvious to type Lisp code both correctly *and* minimally (without cluttering the code), and *a fortiori* portably. Compilers may behave very differently with respect to type declarations and may provide type inference systems of various quality. Perhaps the Common-Lisp standard leaves too much freedom to the compilers in this area.

8 Perspectives

From a low-level point of view, it would be interesting to extend our benchmarks to other compilers and architectures, and also to dig into each compiler's specific optimization options to see if performance can be improved even more.

From a medium-level point of view, benchmarking very simple algorithms was necessary to isolate the parameters we wanted to test (namely pixel access and arithmetic operations). The same experiments should now be conducted on more complex algorithms in order to figure out the impact on performance.

From a high-level point of view, we should also run performance tests at different degrees of genericity. Typically, it would be interesting to compare dynamic object orientation with C++ and Clos [2]. In a second step, the static meta-programming functionalities of the C++ templating system should be compared with the ability of Lisp to generate and compile new functions on the fly.

Finally, one should note that the performance results we obtained are not specific to image processing: any kind of application which involves numerical computation on large sets of contiguous data might be interested to know that Lisp has caught up with performance.

References

[1] Richard J. Fateman, Kevin A. Broughan, Diane K. Willcock, and Duane Rettig. Fast floating-point processing in Common-Lisp. *ACM Transactions on Mathematical Software*, 21(1):26–62, March 1995. Downloadable version at `http://openmap.bbn.com/~kanderso/performance/postscript/lispfloat.ps`.

[2] Sonja E. Keene. *Object-Oriented Programming in* Common-Lisp*: a Programmer's Guide to* Clos. Addison-Wesley, 1989.

[3] Robert A. Mac Lachlan. The python compiler for Cmu-CL. In *ACM Conference on* Lisp *and Functional Programming*, pages 235–246, 1992. Downloadable version at `http://www-2.cs.cmu.edu/~ram/pub/lfp.ps`.

[4] J.K. Reid. Remark on "fast floating-point processing in Common-Lisp". In *ACM Transactions on Mathematical Software*, volume 22, pages 496–497. ACM Press, December 1996.

[5] Guy L. Steele. Common-Lisp *the Language, 2nd edition*. Digital Press, 1990. Online and downloadable version at `http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html`.

[6] Henry S. Warren. *Hacker's Delight*. Addison Wesley Professional, July 2002. `http://www.hackersdelight.org`.