How to Make LISP Go Faster than C

Didier Verna

didier@lrde.epita.fr
http://www.lrde.epita.fr/˜didier

Version 1.4 – June 13, 2006

Introduction

Myths and legends...

- "LISP is slow" ... **NOT !** (it's been 20 years)
- Why is LISP fast ?
  - **Smart compilers** (⇒ native machine code)
  - **Static typing** (types known at compile-time)
  - **Safety levels** (compiler optimizations)
  - **Efficient data structures** (arrays, hash tables *etc.*)
- Demonstration:
  - Comparative C and LISP benchmarks
  - 4 simple image processing algorithms
  - Pixel storage and access / arithmetic operations
- ⇒ **Equivalent performance**
  (LISP 10% better in some cases)

Table of contents

Experimental conditions

- **The algorithms:** the "point-wise" class
  - Pixel assignment / addition / multiplication / division
  - Parameters: image size / type / storage
  - Presented: $800 * 800$ `int` / `float` images
- **The protocol**
  - Debian GNU Linux / 2.4.27-2-686 packaged kernel
  - Pentium 4 3GHz / 1GB RAM / 1MB level 2 cache
  - Single user mode / SMP off (no hyperthreading)
  - Measures on 200 consecutive iterations

C code sample

The `add` function

```
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;
}
```

- *Gcc* 4.0.3 (Debian package)
- Full optimization: `-O3 -DNDEBUG` plus inlining
- *Note:* inlining should be almost negligible

Results (seconds)

Time is of the Essence

Fully optimized inlined C code

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

- **Surprise:** integer division should be costly
- “Constant Integer Optimization” (with inlining)
- **Do not neglect inlining !**

First shot at Lisp code

CMU-CL 19c (CVS)

The add function, take 1

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

- Common-Lisp's standard `simple-array` type

- **Interpreted version:** 2300x
- **Compiled version:** 60x
- **Optimized version:** 20x

Untyped code ⇒ *dynamic* type checking !

Typing mechanisms

- **Typing paradigm:**
  - **Type information** (COMMON-LISP standard)
    Declare the *expected* types of LISP objects
  - **Type information is optional**
    Declare only what you know; give hints to the compilers
  - Both a *statically* and *dynamically* typed language
- **Typing mechanisms:**
  - **Function arguments:**

    ```
    (make-array size :element-type 'single-float)
    ```
  - **Type declarations:**
    Function parameter / freshly bound local variable
  - ...

Typed LISP code sample
Declaring the types of function parameters

The `add` function, take 2

```
(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)))))
```

- `simple-array`'s . . .
- of `single-float`'s . . .
- unidimensional.

Object representation

Why typing matters for performance

- Dynamic typing ⇒ objects of any type (worse: any size)
- LISP variables don't carry type information: objects do

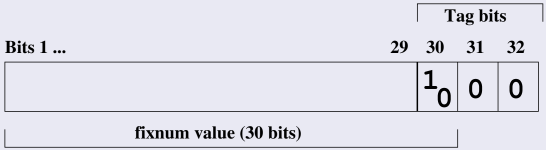
The "boxed" representation of LISP objects

- **Dynamic type checking is costly !**
- **Pointer dereferencing is costly !**

The benefits of typing
2 examples

- **Array storage layout:**
  - Homogeneous arrays of a known type
    $\Rightarrow$ native representation usable
  - Specialization of the `aref` function
  - “Open Coding”
- **Immediate objects:**
  - Short (less than a memory word)
  - Special “tag bits” (invalid as pointer values)
  - $\Rightarrow$ Encoded inline

Unboxed `fixnum` representation



Example: optimizing a loop index

`(dotimes (i 100) ...)`

Disassembly of a `dotimes` macro

```
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
```

Activating optimization

- "Qualities" (COMMON-LISP standard): between 0 and 3
- `safety`, `speed` *etc.*
- Global or local declarations in source code (no compiler flag)

Global qualities declaration

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

- **Safe code:** declarations treated as assertions
- **Optimized code:** declarations trusted

Results

And here comes a little surprise…

C and LISP comparative performance

	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

- Identical performances from C and LISP
- C better at integer division
  (no "constant integer optimization" in LISP compilers)
- **Surprise:** LISP 10% faster at floating-point division

Type inference

Static typing is not as easy as it seems...

- **What to do when not all types are provided ?**
  - What about the type of `i` and `size` ?
  - What about the type of `(* fixnum fixnum)` ?
- $\Rightarrow$ **Figure out at run-time**
  - Stay dynamically typed
  - Use boxed representations
- $\Rightarrow$ **Infer the missing types** ...but
  - Type inference systems of various behavior and quality
  - COMMON-LISP standard too weak about type declarations

Example of type inference

multiply excerpt

```
;; ...
(declare (type (simple-array fixnum (*)) to from))
(declare (type fixnum val))
;; ...
(setf (aref to i) (the fixnum (* (aref from i) val))))))
```

- `(* fixnum fixnum)` $\neq$ `fixnum` in general . . . but
  - `to` declared as an array of `fixnum`'s
  - So the multiplication **has** to return a fixnum
- Sadly, not all type inference systems are *that* smart (*e.g.* Allegro)
  - Need for further explicit type information
  - Type declarations for intermediate values: `the`

Conclusion

- **Optimizing LISP code:**
  data structures, type declarations, optimization
- **Today's compilers are smart:**
  performance can be equivalent to (or better than) C

- Typing can be cumbersome
  (source code annotation)
- Difficult to provide both correct and minimal information
  (weakness of the COMMON-LISP standard)

Perspectives

- **Low level:** try other compilers / architectures (and compiler / architecture specific optimization settings)
- **Medium level:** try more sophisticated algorithms (neighborhoods, front-propagation)
- **High level:** try different levels of genericity (dynamic object orientation, static meta-programming)

- **Do not restrict to image processing**

In greater detail...

For the interested reader

- **Beating C in Scientific Computing Applications**
  On the Behavior and Performance of Lisp, Part I.
  Verna, D. (2006). In *Third European* LISP *Workshop at ECOOP, Nantes, France*.
  `http://lisp-ecoop06.bknr.net/`.

Logo by Manfred Spiller