Performance and Genericity
the forgotten power of Lisp

Didier Verna

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

April 05 – ACCU 2008
Some Background
Which explains a lot…

- **Assistant professor in computer science**
  - Research on the performance and expressiveness of Common Lisp
  - Teaching (amongst other things) functional programming to imperative-biased students

- **Imperative-educated myself**
  - But I resisted

- **Member of the XEmacs core maintainers team**
  - 11 years
Why Lisp?
What else do you need?

- Functional, purely or not
- Imperative
- **Object-Oriented / MOP**
- Aspect- / Context-Oriented
- Declarative
- Reflexive (introspection / intercession)
- Macros
- Forms of pattern-matching, currying
- Strict evaluation or not
- Dynamically Typed, or not
- Lexically scoped, or not
- Interpreted / Byte-Compiled / Compiled, Embeddable
- No real difference between run-time and compile-time

Regular expressions, web servers, web clients, foreign-functions interfaces, OpenGL, multi-threading support etc. etc. etc.
From Simon’s keynote
Simon’s big picture... was almost complete
I’m a peaceful guy…

- Please continue using your favorite language
- Please continue *wishing* you could use your favorite language
- Please don’t feel aggressed
  “This is cool” ≠ “That is bad”

… BUT:

- Don’t you dare complaining about the parens
- Don’t you dare thinking that Lisp is dead
  *It doesn’t even smell funny*
  
  ▶ Old ≠ dead
  ▶ Old = *mature*
  ▶ Please *at least* have the decency to mention it!
Because if you can read this...

```cpp
template <template <class> class M, typename T, typename V>
struct ch_value_<M<tag::value_<T>>, V>
{ typedef M<V> ret; };

template <template <class> class M, typename I, typename V>
struct ch_value_<M<tag::image_<I>>, V>
{ typedef M<mln_ch_value(I, V)> ret; };

template <template <class, class> class M, typename T, typename I, typename V>
struct ch_value_<M<tag::value_<T>, tag::image_<I>>, V>
{ typedef mln_ch_value(I, V) ret; };

template <template <class, class> class M, typename P, typename T, typename V>
struct ch_value_<M<tag::psite_<P>, tag::value_<T>>, V>
{ typedef M<P, V> ret; };
```
... sure you can read that!

\begin{verbatim}
(template (template (class) (class M) (typename T) (typename V))
    (struct (ch_value_ (M (tag::value_ T)) V))
    (typedef (M V) ret)))

(template (template (class) (class M) (typename I) (typename V))
    (struct (ch_value_ (M (tag::image_ I)) V))
    (typedef (M (mln_ch_value I V)) ret)))

(template (template (class class) (class M) (typename T) (typename I) (typename V))
    (struct (ch_value_ (M (tag::value_ T) (tag::image_ I)) V))
    (typedef (mln_ch_value I V) ret)))

(template (template (class class) (class M) (typename P) (typename T) (typename V))
    (struct (ch_value_ (M (tag::psite_ P) (tag::value_ T)) V))
    (typedef (M P V) ret)))
\end{verbatim}
Performance

1. Experimental Conditions

2. C Programs and Benchmarks

3. Lisp programs and benchmarks
   - Raw Lisp
   - Typed Lisp
   - Results

4. Type inference
Genericity

5 Binary Methods non-issues
- Types, Classes, Inheritance
- Corollary: method combinations

6 Enforcing the concept – usage level
- Introspection
- Binary function class

7 Enforcing the concept – implementation level
- Misimplementations
- Strong binary functions
Subliminal slide
You didn’t notice...
Part I

Performance

Breaking the legend of slowness
Introduction
False beliefs

- Yobbo sez: “But Lisp is slow right?”
- Me: “How do you know that?”
- Yobbo replies *(choose your favorite answer)*:
  - Huh, it’s a well known fact
  - Well, that’s what I was told
  - Hmmm, last time I checked... (yeah, in 84)
Lisp is not slow

- It’s been 20 years
  - Smart compilers (⇒ native machine code)
  - Weak typing (types known at compile-time)
  - Safety levels (compiler optimizations)
  - Efficient data structures (arrays, hash tables etc.)
  - Today’s machines ≠ 1960’s machines

We need rock solid evidence:

- Comparative C and Lisp benchmarks (part 1: full dedication)
- 4 simple image processing algorithms
- Pixel storage and access / arithmetic operations
Table of contents

1 Experimental Conditions

2 C Programs and Benchmarks

3 Lisp programs and benchmarks
   - Raw Lisp
   - Typed Lisp
   - Results

4 Type inference
Experimental conditions

- **The algorithms**: the “point-wise” class
  - Pixel assignment / addition / multiplication / division
  - Soft parameters: image size / type / storage / access
  - Hard parameters: compilers / optimization level
  - More than 1000 individual test cases

- **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

```c
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
In terms of behavior

- **1D implementation** *slightly better* (10% ⇒ 20%)
- **Linear access faster** (15 ⇒ 35 times)
  - Arithmetic overhead: only 4x – 6x
  - Main cause: hardware cache optimization
- **Optimized code** faster (60%) in linear case, irrelevant in pseudo-random access
- **Inlining negligible** (2%)
**Results**

In terms of performance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Integer Image</th>
<th>Float Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>0.29</td>
<td>0.29</td>
</tr>
<tr>
<td>Addition</td>
<td>0.48</td>
<td>0.47</td>
</tr>
<tr>
<td>Multiplication</td>
<td>0.48</td>
<td>0.46</td>
</tr>
<tr>
<td>Division</td>
<td>0.58</td>
<td>1.93</td>
</tr>
</tbody>
</table>

- Not much difference between pixel types
- **Surprise:** integer division should be costly
  - “Constant Integer Optimization” (with inlining)
  - **Do not neglect inlining!**
First shot at Lisp code

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 source 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
  - ...
The add function, take 2

```lisp
(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...
- uni-dimensional.
- 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!
The benefits of typing
2 examples

- **Array storage layout:**
  - Homogeneous arrays of a known type
    ⇒ 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)
  ⇒ Encoded inline

### Unboxed `fixnum` representation

<table>
<thead>
<tr>
<th>Tag bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Bits 1 ...

fixnum value (30 bits)
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...
- uni-dimensional.
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:          ADD    EAX, 4
  9D:          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
The `add` function

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

- **CMU-CL (19c), SBCL (0.9.9), ACL (7.0)**
- **Full optimization:** `(speed 3), 0 elsewhere`
- **Array type:** 1D, 2D
- **Array access:** `aref, row-major-aref, svref`
Comparative results
In terms of behavior

≠ Plain 2D implementation much slower (2.8x ⇒ 4.5x)
= Linear access faster (30 times)
  ▶ Same reasons, same behavior…
= Optimized code faster in linear case, irrelevant in pseudo-random access
  ≠ Gain more important in Lisp (3x ⇒ 5x)
  ≠ Gain more important on floating point numbers
  ⇒ In Lisp, safety is costly
= Inlining negligible
  ≠ No “Constant Integer Optimization”
  ≠ Negative impact on performance (-15%), if any
  ⇒ Inlining still a “hot” topic (register allocation policies ?)
Comparative results
In terms of performance

Pseudo-random access

- Assignment: Lisp 19% faster than C
- Other: insignificant (5%)
- Exception: integer division
Comparative results
In terms of performance

Linear access

- **ACL**: poor performance
- **CMU-CL, SBCL**: strictly equivalent to C
- C wins on integer division, loses on floating-point one
Type inference

A weakness of Common Lisp . . .

- Static typing cumbersome (source code annotations)
  - Can we provide *minimal* type declarations . . .
  - . . . and rely on type inference ?

- Incremental typing by compilation log examination

- Unfortunately:
  - Compiler messages not necessarily ergonomic
  - Type inference systems not necessarily clever
Example of (missing) 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

- CMU-CL and SBCL ok, ACL not ok.
  - Need for further explicit type information
  - worse in ACL:
    declared-fixnums-remain-fixnums-switch
Conclusion

- **In terms of behavior**
  - External parameters: no surprise
  - Internal parameters: differences, attenuated by optimization

- **In terms of performance**
  - Comparable results in both languages
  - Very smart Lisp compilers (given language expressiveness)

- **However:**
  - Typing can be cumbersome
  - Difficult to provide both correct and minimal information (weakness of the Common Lisp standard)
  - Inlining is still an issue
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
Subliminal slide
You didn’t notice...
Part II

Genericity

a guided-tour through binary methods
Introduction
What are binary methods?

- **Binary Operation:** 2 *arguments* of the same *type*
  Examples: arithmetic / ordering relations (=, +, > etc.)

- **OO Programming:** 2 *objects* of the same *class*
  Benefit from polymorphism *etc.*

  ⇒ Hence the term **binary method**

- **However:** [Bruce et al., 1995]
  - problematic concept in traditional OO languages
  - type / class relationship in the context of inheritance
Table of contents

5 Binary Methods non-issues
   - Types, Classes, Inheritance
   - Corollary: method combinations

6 Enforcing the concept – usage level
   - Introspection
   - Binary function class

7 Enforcing the concept – implementation level
   - Misimplementations
   - Strong binary functions
The **Point** class UML hierarchy

```
  Point
   x, y : Integer
   equal (Point) : Boolean

  ColorPoint
   color : String
   equal (ColorPoint) : Boolean
```
The C++ Point class hierarchy

class Point
{
    int x, y;

    bool equal (Point& p)
    {
        return x == p.x && y == p.y;
    }
};

class ColorPoint : public Point
{
    std::string color;

    bool equal (ColorPoint& cp)
    {
        return color == cp.color && Point::equal (cp);
    }
};
But this doesn’t work…
Overloading is not what we want

Looking through base class references

```cpp
int main (int argc, char *argv[]) {
    Point& p1 = *new ColorPoint (1, 2, "red");
    Point& p2 = *new ColorPoint (1, 2, "green");

    std::cout << p1.equal(p2) << std::endl;
    // => True. #### Wrong!
}
```

- **ColorPoint::equal only overloads** `Point::equal` in the derived class
- From the base class, only `Point::equal` is seen
- What we want is to use the definition from the **exact** class
The C++ `Point` class hierarchy

```cpp
class Point {
    int x, y;

    virtual bool equal (Point& p) {
        return x == p.x && y == p.y;
    }
};

class ColorPoint : public Point {
    std::string color;

    virtual bool equal (ColorPoint& cp) {
        return color == cp.color && Point::equal (cp);
    }
};
```
But this doesn’t work either... Still got overloading, still not what we want

The forbidden fruit

```cpp
virtual bool equal (Point& p);
virtual bool equal (ColorPoint& cp); // #### Forbidden!
```

- **Invariance** required on virtual methods argument types
- **Worse:** here, the `virtual` keyword is *silently* ignored
- And we get an overloading behavior, as before
- **Why?** To preserve type safety
Why the typing would be unsafe
And lead to errors at run-time

Example of run-time typing error

In fact, a ColorPoint

Just a Point

bool foo (Point& p1, Point& p2)
{
  return p1.equal (p2);
}

But gets only a Point!
The ColorPoint implementation expects a ColorPoint argument (ex. accesses the color field)

The ColorPoint implementation expects a ColorPoint argument (ex. accesses the color field)
When subtyping a polymorphic method, we must
- supertype the arguments (contravariance)
- subtype the return value (covariance)

Note: C++ is even more constrained
- The argument types must be invariant

Note: Eiffel allows for arguments covariance
- But this leads to possible run-time errors

Analysis: [Castagna, 1995].
- Lack of expressiveness
  subtyping (by subclassing) ≠ specialization
- Object model defect
  single dispatch (not the record-based model)
Methods vs. Generic Functions
- C++ methods belong to classes
- CLOS generic functions look like ordinary functions (outside classes)

Single dispatch vs. Multi-Methods
- C++ dispatch based on the first (hidden) argument type (this)
- CLOS dispatch based on the type of any number of arguments

Note: a CLOS “method” is a specialized implementation of a generic function
The CLOS Point class hierarchy

(defclass point ()
  ((x : initarg : x : reader point-x)
   (y : initarg : y : reader point-y)))

(defclass color-point (point)
  ((color : initarg : color : reader point-color)))

;; optional
defgeneric point= (a b)

(defmethod point= ((a point) (b point))
  (and (= (point-x a) (point-x b))
       (= (point-y a) (point-y b))))

(defmethod point= ((a color-point) (b color-point))
  (and (string= (point-color a) (point-color b))
       (call-next-method)))
How to use it?
Just like ordinary function calls

Using the generic function

```
(let ((p1 (make-point :x 1 :y 2))
      (p2 (make-point :x 1 :y 2))
      (cp1 (make-color-point :x 1 :y 2 :color "red"))
      (cp2 (make-color-point :x 1 :y 2 :color "green")))
  (values (point= p1 p2)
           (point= cp1 cp2)))
;; => (T NIL)
```

- Proper *method* selected based on *both* arguments (multiple dispatch)
- Function call syntax, more pleasant aesthetically
  
  ```lisp
  (p1.equal(p2) or p2.equal(p1))
  ```
- => Hence the term **binary function**
To avoid code duplication:
  - C++: Point::equal()
  - CLOS: (call-next-method)

Applicable methods:
  - All methods compatible with the arguments classes
  - Sorted by (decreasing) specificity order
  - call-next-method calls the next most specific applicable method

Method combinations:
  - Ways of calling several (all) applicable methods (not just the most specific one)
  - Predefined method combinations: and, or, progn etc.
  - User definable
The C++ Point class hierarchy

class Point
{
    int x, y;

    bool equal (Point& p)
    {
        return x == p.x && y == p.y;
    }
};

class ColorPoint : public Point
{
    std::string color;

    bool equal (ColorPoint& cp)
    {
        return color == cp.color && Point::equal (cp);
    }
};
The CLOS Point class hierarchy

(defclass point ()
  ((x :initarg :x :reader point-x)
   (y :initarg :y :reader point-y)))

(defclass color-point (point)
  ((color :initarg :color :reader point-color)))

;; optional
(defgeneric point= (a b))

(defmethod point= ((a point) (b point))
  (and (= (point-x a) (point-x b))
       (= (point-y a) (point-y b))))

(defmethod point= ((a color-point) (b color-point))
  (and (string= (point-color a) (point-color b))
       (call-next-method)))
Applicable methods
There are more than one...

- **To avoid code duplication:**
  - **C++**: `Point::equal()`
  - **CLOS**: `call-next-method`

- **Applicable methods:**
  - All methods compatible with the arguments classes
  - Sorted by (decreasing) specificity order
  - `call-next-method` calls the next most specific applicable method

- **Method combinations:**
  - Ways of calling several (all) applicable methods (not just the most specific one)
  - Predefined method combinations: `and`, `or`, `progn` *etc.*
  - User definable
Using the `and` method combination

Comes in handy for the equality concept

---

The `and` method combination

```lisp
(defgeneric point= (a b)
  (:method-combination and)
)

(defmethod point= and ((a point) (b point))
  (and (= (point-x a) (point-x b))
       (= (point-y a) (point-y b))))

(defmethod point= and ((a color-point) (b color-point))
  (and (call-next-method)
       (string= (point-color a) (point-color b))))
)
```

⇒ In CLOS, the generic dispatch is (re-)programmable
Binary methods could be misused
Can we protect against it?

The `point=` function used incorrectly

```lisp
(let ((p (make-point :x 1 :y 2))
     (cp (make-color-point :x 1 :y 2 :color "red")))
  (point= p cp))
;; => T #### Wrong !
```

- `(point= <point> <point>)` is an applicable method (because a color-point is a point)
- The code above is valid
- And the error goes unnoticed
Using the function `class-of`

```
(assert (eq (class-of a) (class-of b)))
```

- **When to perform the check?**
  - In all methods: code duplication
  - In the basic method: not efficient
  - In a `before-method`: not available with the and method combination
  - In a user-defined method combination: not the place

- **Where to perform the check?** (a better question)
  - Nowhere near the code for `point=`
  - Part of the binary function concept, not `point=`

- ⇒ We should implement the binary function `concept`
  - A specialized class of generic function?
CLOS itself is object-oriented

- The CLOS MOP: a *de facto* implementation standard
- The CLOS components (classes *etc.*) are (meta-)objects of some (meta-)classes
- Generic functions are meta-objects of the standard-generic-function meta-class

⇒ We can subclass standard-generic-function

**The binary-function meta-class**

```lisp
(defclass binary-function (standard-generic-function) ()
  (:metaclass funcallable-standard-class))

(defmacro defbinary (function-name lambda-list &rest options)
  `(defgeneric ,function-name ,lambda-list (:generic-function-class binary-function)
    ,@options))
```
Calling a generic function involves:

- Computing the list of applicable methods
- Sorting and combining them
- Calling the resulting *effective* method

**compute-applicable-methods-using-classes**

- Does as its name suggests
- Based on the classes of the arguments
- A good place to hook

We can specialize it!

- It is a generic function . . .

---

**Specializing the c-a-m-u-c generic function**

```lisp
(defmethod c-a-m-u-c :before ((bf binary-function) classes)
  (assert (equal (car classes) (cadr classes))))
```
Binary methods could be misimplemented
Can we protect against it?

- We protected against calling
  \( (\text{point}= \text{<point>} \text{<color-point>}) \)
- Can we protect against *implementing* it?
- \text{add-method}
  - Registers a new method (created with \text{defmethod})
  - We can specialize it!
    - It is a generic function . . .

Specializing the \text{add-method} generic function

\[
\begin{align*}
(\text{defmethod} & \text{add-method} : \text{before} ((\text{bf binary-function}) \text{method})) \\
(\text{assert} & \text{(apply '#'equal (method-specializers method))}))
\end{align*}
\]
Binary methods could be forgotten
Can we protect against it?

- **Strong binary functions:**
  - Every subclass of `point` should specialize `point=`
  - Late checking: at generic function call time
    (preserve interactive development)

- **Binary completeness:**
  1. There is a specialization on the arguments’ exact class
  2. There are specializations for all super-classes

- **Introspection:**
  - Binary completeness of the list of applicable methods
  - `c-a-m-u-c` returns this!

**Hooking the check**

```lisp
(defun c-a-m-u-c ((bf binary-function) classes)
  (multiple-value-bind (methods ok) (call-next-method)
    ;; ...
    (values methods ok)))))
```
Is there a bottommost specialization?

Check #1

- \( \text{classes} = (\text{<exact> <exact>}) \)
- \( \text{method-specializers} \) returns the arguments \( \text{classes} \) from the \text{defmethod} call
- \( \Rightarrow \) We should compare \( \text{<exact>} \) with the specialization of the first applicable method

Check #1

\[
\begin{align*}
(\text{let*} \ ((\text{method} (\text{car methods})) \ \text{(class} (\text{car (method-specializers method)})))) \\
(\text{assert} (\text{equal} (\text{list class class) classes})) \\
;;; \ldots \\
)
\]
Are there specializations for all super-classes?

Check #2

- **find-method** retrieves a generic function’s method given a set of qualifiers / specializers
- **method-qualifiers** does as its name suggests
- **class-direct-superclasses** as well

```lisp
(labels ((check-binary-completeness class)
  (find-method bf (method-qualifiers method)
    (list class class))
  (dolist
    (cls (remove-if 
      #'(lambda (elt)
        (eq elt (find-class
          'standard-object)))
      (class-direct-superclasses class)))
    (check-binary-completeness cls)))
  (check-binary-completeness class)))
```
Conclusion

- Binary methods problematic in traditional OOP
- Multi-methods as in CLOS remove the problem
- CLOS and the CLOS MOP let you support the concept:
  - make it available
  - ensure a correct usage
  - ensure a correct implementation

- But the concept is implemented explicitly
  - CLOS is not just an object system
  - CLOS is not even just a customizable object system

CLOS is an object system designed to let you program new object systems
Lisp satisfies
Alive and kicking

- Lisp is a truly multi-paradigm programming language
  *Probably the most versatile of them*

- Lisp is the language of freedom
  *PPP: Permissive Programming Paradigm*

- Freedom means more ways to shoot yourself in the foot

- But also the ability to be extremely defensive if you want to
What’s the next challenge in computer languages?

- Not functional programming (we won)
  Threads are dead, long live Erlang!
- Dynamic vs. static languages
- Simon: “Be pure by default, impure when needed”
- Me: “Be dynamic by default, static when needed”


**Articles**


Books and stuff

- Practical Common Lisp (Peter Seibel)
- Structure and implementation of Computer programs [scheme] (Abelson, Sussman)
- Have a look at the link section on my website
Next Events

  http://prog.vub.ac.be/~pcostanza/els08/

- **5th European Lisp Workshop**, July 7 2008, Cyprus, co-located with ECOOP.  
  http://elw.bknr.net/2008

- **Next International Lisp Conference** ... 2009  
  MIT, Cambridge
Congratulations!
Remember? I’m a peaceful guy...

I've just heard that C++ is going to have lambda expressions... 48 years after Lisp!