Overview

- In a Lisp development environment, the compiler has several tasks that are much more important than getting the maximum unsafe execution speed on highly tuned benchmarks.
- Many Lisp programmers find it difficult to get even moderately good performance; help is available.
- Programs can be written in a more maintainable way if high-level optimization in the compiler can be relied upon.

A Bit of History

Lisp programming environments and implementation techniques have changed quite a bit over the years:

- Early Lisp systems provided interpreters with consistently poor performance but what was then unusually good debugging facilities.
- Later Lisp systems added optimizations such as compilers and lexical variables, but...
- The user-hostility, poor debuggability and semantic irregularities of early Lisp compilers for stock hardware made their use unpopular.
- Special-purpose Lisp Machine hardware provided both a good programming environment and consistently good performance, but only for those who could afford it.
- Today's stock workstation Lisp systems have caught up to Lisp machines in peak performance, but are still trying to catch up in performance consistency and programming environment quality.

The CMU Common Lisp project started in the early 80's during the Lisp Machine era. The Lisp Machine was our model of what a Lisp programming environment ought to be, and from the beginning our system was a windowing system with a safe compiler. Work on the new compiler (Python) was begin in 86.

Why use Lisp?

Programmer productivity is the only reason:

- If your goal is to have a program that runs as fast as possible, then Lisp is probably the wrong language.
- Lisp increases programmer productivity by saving work during development and maintenance.
- This productivity is gained at the cost of time and space inefficiency.

Program Development Tasks

In an incremental Lisp development environment, all programming activities tend to take place concurrently, but can still be aided by activity-specific tools:

- design
- coding
- *debugging*
- tuning
- maintenance

Debugging

For several reasons, there has been a trend toward compiling Lisp programs during development and debugging. Many implementations provide some source-level debugging capabilities in compiled code, but there are serious drawbacks:

- Missing functionality, such source location and printing.
- Excessive performance penalty, especially in numeric code.

CMU CL avoids these limitations, since it maintains source location information, and does optimizations such as register allocation without affecting debuggability.

Debugger Example

```
(defstruct my-struct
  (slot :unknown :type (or integer (member :unknown))))

(defun source-demo (structs)
  (dolist (struct structs)
    (when (oddp (my-struct-slot struct)) (return struct))))

(source-demo
 (list (make-my-struct :slot 0)
       (make-my-struct :slot 2)
       (pathname "foo")))

Type-error in source-demo:
  #p"foo" is not of type my-struct

Restarts:
  0: Return to Top-Level.

Debug  (type H for help)

(source-demo (#s(my-struct slot 0) #s(my-struct slot 2) #p"foo"))
6] l
struct  =  #p"foo"
structs  =  (#s(my-struct slot 0) #s(my-struct slot 2) #p"foo")

6] (my-struct-slot (cadr structs))

2
6] source 1

(oddp (#:***here*** (my-struct-slot struct)))
6] edit
```

Run-Time Errors

Debugger capabilities are important, but the *time* that the debugger is invoked is also very important. There is only so much a debugger can do after a program has already crashed and burned.

Debuggers for other languages have caught up to and even surpassed Lisp, but Lisp programs are still easier to debug because:

- Run-time errors are more informative, and
- Run-time errors are signalled much sooner after the inconsistency arises.

Python improve the quality and timeliness of run-time error messages through strict type checking.

Type Check Example

```
(deftype limit () '(unsigned-byte 16))
(deftype a-kinds ()
  '(member :up :down :left))

(defstruct a
  (kind :up :type a-kinds)
  (value 0 :type limit)
  (next nil :type (or a null)))

(defun find-it (lim kind in)
  (declare (type limit lim)
           (type a-kinds kind)
           (type (or a null) in))
  (do ((current in (a-next current)))
      ((null a) nil)
    (when (and (eq (a-kind current) kind)
               (> (a-value a) lim))
      (return current))))
```

- It is impossible to create or modify an A with an invalid slot value. Both of these forms signal an error:

  ```
  (make-a :kind :strange)
  (setf (a-value a) -1)
  ```

- Slot types can be trusted, so this loop compiles safely with no type checks in the loop.

Compile-Time Errors

Lisp has been dismissed as a language for "serious" programming because it lacks compile-time type checking. Python’s type inference catches type errors:

```
(defmacro addf (x n)
  ‘(+ ,x (the single-float ,n)))

(defun type-error (x)
  (addf x 3))
```

Giving compiler warnings like this:

```
In: defun type-error
  (addf x 3)
--> +
==>
  (the single-float 3)
Warning: This is not a single-float:
  3
```

Dynamic Type Checking

In addition to detecting static type errors, Python also detects simple dynamic type errors like:

```
(defun foo (x arg)
  (ecase x
    (:register-number
     (and (integerp arg) (>= x 0)))
    ...))
```

```
In: defun foo
  (>= x 0)
--> if <
==>
  x
Warning: This is not a (or float rational):
  :register-number
```

- This sort of type inference enables low-cost strict type checking.
- Compile-debug cycles to fix trivial type errors are reduced.
- Not all (or even most) programs can be statically checked in Common Lisp, so run-time errors are still possible.
- Some errors can be detected without testing, but testing is always necessary.

Tuning and the Efficiency of Lisp

- Implementation inefficiency is a productivity problem, since programmers have to spend lot of time tuning their code.
- Modern Lisp compilers can generate good code, but convincing them to do it is hard.
- Unfortunately, implementations and programmers often sacrifice desirable properties (such as debuggability and maintainability) for efficiency.
- Some degree of trade-off is inevitable, but implementation technology has much more impact than is generally recognized.
- You *can* have your cake and eat (most) of it too.

Optimization in Breadth

In a big language like Common, good compiler algorithms and good implementation can still be defeated in the details. Python has a lot of operation-specific knowledge:

- There are over a thousand special-case rules for 640 different operations.
- There are 14 flavors of function call and three representations for multiple values.
- The + function has 6 transforms, 8 code generators and a type inference method.

Type inference rules:

```
(+ (the (integer 5 47) x) 3) ;  is (integer 8 50).
(logand x #xFFFF)            ;  is (unsigned-byte 16).
(aref (the base-string x) i) ;  is base-character.
```

Transformations:

```
(typep x ’(and (satisfies foo-p) (integer 3 7)))
  ==>
(and (foo-p x) (integerp x) (<= 3 x 7))

((lambda (&rest x) (apply #’fun x))
 a b c)
  ==>
(fun a b c)
```

Accountability

It is hard to write efficient Lisp code because:

- The cost of a generic operation can vary over a wide range depending on the exact operation performed.
- The cost of simple operations varies widely depending on the compiler's amount of type knowledge.
- The workings of the compiler are (rightly) totally mysterious to most programmers.

Accountability is an important user-interface concept that Lisp compilers should pay more attention to. In a compiler, accountability means that efficiency notes:

- are needed whenever an inefficient implementation is chosen for non-obvious reasons.
- should convey the seriousness of the situation.
- must explain precisely what part of the program is being referred to.
- must explain why the inefficient choice was made (and how to enable efficient compilation.)

Efficiency Note Example

Consider this example of "not good enough" declarations:

```
(defun eff-note (s i x y)
  (declare (string s) (fixnum i x y))
  (aref s (+ i x y)))
```

which gives these efficiency notes:

```
In: defun eff-note
  (+ i x y)
==>
  (+ (+ i x) y)
Note: Forced to do inline (signed-byte 32) arithmetic (cost 3).
      Unable to do inline fixnum arithmetic (cost 2) because:
      The first argument is a (integer -1073741824 1073741822),
      not a fixnum.

  (aref s (+ i x y))
--> let*
==>
  (kernel:data-vector-ref array kernel:index)
Note: Forced to do full call.
      Unable to do inline array access (cost 5) because:
      The first argument is a base-string, not a simple-base-string.
```

Numbers in Python

Python devotes a great deal of effort to optimizing numeric code, since:

- Even Lisp programs make substiatial use of numbers (especially fixnums.)
- Most common numeric operations are in the hardware, so efficient code is possible.
- It is difficult to generate good numeric code for Common Lisp.

Several Python features combine to make efficient numeric code much easier to attain:

- Type inference and the wide palette of code generation strategies mean that it takes fewer declarations to make your code efficient.
- Block compilation increases the range over which efficient numeric representations can be used.
- Python’s efficiency notes warn you whenever an numeric operation was not compiled efficiently.

Optimization in Depth

Efficient Lisp programming is hard because the world of efficient Lisp is a small world. In comparing the implementation of semantically identical constructs, we find some that are dramatically more efficient than others because the compiler happens to recognize them.

In order to provide consistent performance, an implementation must make it harder for a program to "fall off the edge of the world." Python does this in several ways:

- Programs are converted into a simple intermediate representation which naturally expresses Common Lisp semantics. Similar programs tend to be canonicalized.

- General compilation strategies are used to prevent minor changes (such as adding a return value) from causing dramatic performance degradation.

- Independent aspects of semantics are converted into independent operations, resulting in code that is no more general than it needs to be:
    - Error check generation is totally separate from operation semantics, so error checks can be separately optimized.

    - The many versions of function call provide various combinations of features.

Programming Tasks Revisited

Python enhances Lisp’s traditional tuning and debugging tools.

But what about the other programming tasks?

- *design*
- *coding*
- debugging
- tuning
- *maintenance*

How can a programming environment aid design, coding and maintenance?

Efficient Abstraction

Abstraction makes maintenance and incremental program design/development easier; this is why Common Lisp has such a rich variety of abstraction tools:

- Global functions
- Local functions
- First-class functional values
- Dynamic typing (ad-hoc polymorphism)
- Generic functions and inheritance
- Macros
- Embedded languages
- Language extensions

But... Programmers are often forced to avoid appropriate abstraction due to inefficiency (or the difficulty of efficient abstraction.)

Partial Evaluation

Partial evaluation is a generalization of constant folding. Instead of only replacing completely constant expressions with the compile-time result of the expression, partial evaluation transforms arbitrary code using information on parts of the program that are constant:

- Operation is constant:

```
(funcall #'eql x y) ==> (eql x y)
```

- Variables whose bindings are invariant:

```
(let ((x (+ a b)))
  x)
  ==>
(+ a b)
```

- Unused expression deletion:

```
(+ (progn (* a b) c) d)
  ==>
(+ c d)
```

- Conditionals that are constant:

```
(if t x y) ==> x
```

- Control flow simplification:

```
(tagbody (foo) (go T1) (dead) T1 (bar))
  ==>
(progn (foo) (bar))
```

Partial evaluation makes abstraction efficient by eliminating unnecessary generality.

Partial Evaluation Example

```
(declaim (inline find-in))
(defun find-in (next element list &key (key #'identity)
                     (test #'eql test-p) (test-not nil not-p))
  "Find Element in a null-terminated List linked by the accessor
   function Next.  Key, Test and Test-Not are the same as for
   generic sequence functions."
  (when (and test-p not-p)
    (error "Silly to supply both :Test and :Test-Not."))
  (if not-p
      (do ((current list (funcall next current)))
          ((null current) nil)
        (unless (funcall test-not (funcall key current) element)
          (return current)))
      (do ((current list (funcall next current)))
          ((null current) nil)
        (when (funcall test (funcall key current) element)
          (return current)))))
```

When inline expanded and partially evaluated, this transformation results:

```
(find-in #'foo-next 3 foo-list
         :key #'foo-offset :test #'<)
 ==>

(do ((current foo-list (foo-next current)))
    ((null current) nil)
  (when (< (foo-offset current) 3)
    (return current)))
```

Partial Evaluation and Meta-Programming

Meta-programming is writing programs by developing domain-specific extensions to the language. It is the most powerful abstraction mechanism I know of, and it is unique to Lisp.

...but meta-programming is hard because:

- the programmer must be a language designer, and
- good macro implementation requires experience and attention to detail.

Python can't design languages for you, but it can eliminate the need for some attention to detail:

- Partial evaluation makes inline functions as efficient as macros, so macros are only needed when functional abstraction is insufficient.
- Partial evaluation makes life easier for macro writers because they can freely introduce unnecessary conditionals and variable bindings.

Block Compilation

Block compilation assumes that global functions will not be redefined, allowing the compiler to see across function boundaries. In block compiled CMU CL:

- Control transfer is a direct jump.
- Argument syntax checking is safely eliminated.
- Keyword and optional argument parsing is done at compile-time.
- Locally optimized calling conventions are used.
- Numeric arguments and return values are passed in non-pointer registers.
- Entry points can be declared.
- Debuggability is maintained.

Benchmarks

Gabriel benchmarks run-times (w/o GC), normalized to Allegro 3.1.12 on a DECstation 5000 with 24meg of memory (>1 means CMU faster):

Benchmark	Unsafe	Safe
Tmem	0.91	1.72
Tak	1.02	1.74
Boyer	3.32	1.96
Browse	0.97	0.78
Ctak	1.06	1.25
Dderiv	0.79	1.28
Deriv	1.31	1.30
Destructive	1.50	2.00
Iterative-Div2	0.65	1.01
Recursive-Div2	0.90	0.70
Fft	1.55	33.04
Frpoly-Fixnum	1.03	1.09
Frpoly-Bignum	1.85	1.53
Frpoly-Float	0.88	0.85
Puzzle	1.08	5.71
Rtak	1.07	1.75
Takl	0.97	1.34
Stak	2.24	2.22
Run-Init-Traverse	2.22	2.88
Run-Run-Traverse	0.81	2.65
Triang	1.08	2.57
Geom means:	1.19	1.84

Summary

The biggest benefits of using CMU CL come when developing new code:

- Efficient source-level debuggers, strict type checking, and good compile-time and run-time errors ease debugging and maintenance.
- Accountability and high-level optimization make it easier to write efficient programs, making tuning less mysterious.
- High-level optimization allows greater use of abstraction, improving maintainability and reuse.
- Existing benchmarks show that Python's back end is as good or better than those of commercial Lisp implementations.

For a more personal slant on why I think you'll like CMU CL, here are two features that fall in the "I don't know how I lived without" category:

- The source command in the debugger, which shows the precise form that got a run-time error.
- Full type safety, which allows the Common Lisp type system to automatically do much of your data-structure consistency checking.