Lisp v.s. RISC
or
What Common Lisp Implementors Really Want

Robert A. MacLachlan
Carnegie Mellon University
ram@cs.cmu.edu

January 14, 1993

Abstract

For more than 20 years computer architects have been trying to discover how to support Lisp programming, however no clear conclusion has been reached. This is largely due to the continual change in implementation tradeoffs that resulted from changes in hardware (microcode, virtual memory, byte addressability, RISC), compiler technology (register allocation, type inference, flow analysis) and the language itself (user defined types, object orientation, type declarations, programming style.) This paper explores how contemporary Lisp differs from other languages, and uses dynamic statistics collected from real applications compiled by a state-of-the-art Lisp compiler to develop realistic Lisp support recommendations for today's computer architects.

1 Introduction

This paper focuses on the implementation of Common Lisp[8], which is widely used, and is being standardized by ANSI. Lisp shares implementation concerns with other languages (such as Smalltalk), and Lisp is itself a family of languages. Broadly speaking, these languages are oriented toward exploratory programming[12], and tend to sacrifice run-time efficiency for ease of coding and incremental program modification. Exploratory programming languages are particularly applicable to research, development and rapid prototyping. In the following discussion, read "Lisp" to mean "Lisp and most exploratory programming languages."

As far as architectural interactions go, the most significant differences between Lisp and conventional languages such as C are:

- Type declarations are optional; the default type for variables and data structure elements is `T`, which is in effect a universal union type. The actual type (and representation) of type `T` values is not known at compile time, so these values must be represented in a standard format called a *descriptor.* A descriptor contains some *tag* bits which control the interpretation of the other bits and contain some information about the actual type. Descriptors are small enough to fit in a registers; larger objects are represented as pointers.

- Since compile-time type information is incomplete, the compiler may not be able to prove that the operands to type-specific operators are of the correct type. Often object types must be checked at run time to insure that type errors are eventually detected. When this checking is done, the implementation is said to be *safe*; when type checking is omitted for speed, the implementation is *unsafe*. Lisp allows programmer control of the speed/safety tradeoff so that exported interfaces and untested code can be safe, yet tested inner loops can still be fast.

- Function calls are dynamically linked, and functions can be incrementally recompiled at run-time. Function calls also support complex features such as optional and keyword arguments, multiple return values and object-oriented method dispatch. This complexity cannot be eliminated by the compiler, since the possibility of run-time redefinition prevents the compiler from knowing the protocol of the called function.

- Garbage collection is required, and heap allocation is extensively used. In order to support high allocation (and collection) rates with low collection overhead and unintrusive collection pauses, State-of-the-art Lisp implementations use copying generational garbage collectors.[5, 12] These algorithms place demands on the compiler and architecture both by moving objects at unpredictable times and by requiring some memory references to be trapped.

See [2] for an overview of contemporary Lisp systems

2 Object Representation

Lisp implementations for contemporary byte-addressable architectures use a *low-tag* representation for descriptors.[7] The low two (or three) bits of a word are used as tag bits. Some objects (such 30-bit integers) are stored into the descriptor itself, avoiding memory allocation and indirection overhead. The tag for fixed-precision integers is chosen to be zero so that the normal addition, subtraction, comparison and overflow detection mechanisms are directly usable. If the descriptor is a pointer to a heap object, then the upper bits of the descriptor are the upper bits of the pointer. Since low bits of the pointer are replaced with tag bits, heap objects must be allocated with 4 (or 8) byte alignment.

To load or store an object's contents, the tag bits must be removed. This is usually done by specifying the tag code as a negative offset in the memory reference. If there are two tag bits, and the hardware ignores the low bits in word memory references, then tag removal is unnecessary. However, Lisp implementors prefer unaligned accesses to be trapped, since this provides a degree of free type checking. If the run-time type differs from the declared type, then the actual tag often differs from the expected tag, and accesses cause an alignment trap.

Since the descriptor contains only a few bits of type information, most objects also have one or two header words which hold array dimensions, etc. In objects of user defined types (instances), the header contains a pointer to a run-time type descriptor. In order to do a complete type test on a descriptor, it is often necessary to examine both the tag bits and the header, so type checking may require memory references.

3 Related Work

In the early 80's, MSI TTL implementations of the Symbolics 3600 architecture offered the best Lisp cost/performance and the best programming environment.[6] Since then, mass-produced microprocessors have surpassed special-purpose Lisp architectures in cost/performance (though the programming environments are often still inferior.) The main reason for this "failure" is that state-of-the-art VLSI development is too expensive for small-volume designs to be cost-effective.

When RISC architectures were initially proposed in the 80's, a common criticism was that RISC architecture was not language independent, or more

specifically, that RISC machines were C machines. This lead a number of RISC advocates to explore the need for and the nature of support for other languages, and for Lisp in particular.[9, 11, 12] The main features explored were variants of arithmetic and memory reference that did tag checking. Function call performance was also improved via register windows and faster computed jumps. From a Common Lisp implementation perspective, the main shortcomings of this body of work are:

- Lisp implementation techniques, programming style and the language itself were all undergoing rapid change during this period. The Lisp implementations and benchmarks available for use in architecture research were outdated.

- Conclusions were based on dynamic instruction frequency statistics collected from functional simulation of small programs. The low level of data collection made it difficult to determine why a particular frequency was observed, since the linguistic function was unclear. Functional simulation also provided no information about cache, TLB and paging performance, yet anecdotal evidence suggests that Lisp memory system behavior is poor.

4 Why is Lisp so Slow?

Performance improvement is the main reason that has been offered for incorporating language-specific features in hardware. The unacceptable performance exhibited by many Lisp programs has provided a large part of the motivation for Lisp architectures, yet these studies have generally reduced run-time by only 20%–40%.[9, 12] Although helpful, the architectural approach has done little to address anecdotal reports of Lisp programs that are 5 or 50 times slower than they ought to be.

It has been said that "A Lisp programmer knows the value of everything and the cost of nothing"; the kernel of truth in this claim is that the high level and complexity of Lisp operations makes the implementation (and its efficiency) obscure. As long as there is a fairly direct mapping between language operations and machine operations, it is easy for programmers to develop an *efficiency model* which predicts program performance. When the mapping is complex and context-dependent (as in Lisp), there is no intuitive efficiency model, and performance becomes hard to predict.

5 Recent Compiler Technology

The CMU Common Lisp compiler[4] uses a new interpretation of type declarations which solves several problems with implementing Lisp on standard hardware. In effect, Lisp has discovered (optional) strong typing. Historically, Lisp compilers have suffered from two problems with declarations and run-time type checking:

- Declarations are considered to be true, and no type checking is done on declared values. Usually type checking can only be done by telling the compiler to ignore declarations and call a run-time routine for every operation. This makes use of declarations error-prone, since incorrect declarations can result in obscure run-time errors.

- Since (in general) Lisp operation semantics does depend on the run-time value of the operands, Lisp implementors have tended to view operand type checking and operation as in indivisible unit. However, this forces an irreducible type checking overhead.

In CMU Common Lisp, declarations help (rather than hurt) safety. In the default compilation mode, type declarations are verified; if necessary run-time type checks are emitted solely to test that a declaration is correct. Paradoxically, this approach gives an overall reduction in type checking, since it transfers the responsibility for type checking to the initialization code for variables and datastructures. Initialization and assignment are dynamically less common than reference, so fewer type checks are required.

Even when are no declarations, the compiler still unbundles type checking from operation semantics, allowing independent optimization of type checking. Depending on programming style, the reduction in type checking varies from modest (25%) to very large (complete elimination).

The largest performance gain from using CMU Common Lisp often comes from heeding efficiency diagnostics that the compiler displays. These efficiency notes make it easy for a programmer to verify that all operations have been open-coded as expected.

Languages in the Smalltalk family share Lisp's problems with dynamic type checking. Considerable work has been done on developing compilation techniques which minimize the overhead of run-time operand type dispatching. In particular, the compiler for Self[1] has demonstrated that rare general cases can often be handled by duplicating loops and moving the test before the loop. The main cost is increased code size and compilation time.

6 Data Collection

Empirical data was collected from the execution of four benchmarks under two test conditions (safe and unsafe) on two processors: SPARC and MIPS R3000. Dynamic operation frequency was determined by instrumenting the compiler for CMU Common Lisp.[3] Each basic block in the in the generated code is assigned a counter which is incremented at the start of the block. The compiler dumps auxiliary information about the code in each block so that the instruction-level meaning of a block count can be determined.

The summary of basic block contents is based on the Virtual OPerations (VOPs) in the compiler's intermediate representation. A detailed instruction breakdown is not recorded, but variations in the instruction sequence actually generated for a VOP are controlled for by recording a predicted cycle count based on the instructions generated. Since the instruction sequence for a VOP is generally fixed or at least highly predictable, conclusions about instruction set design can be drawn from this information.

This profiling approach has the advantage of causing only a modest slowdown in execution, permitting instrumentation of realistic programs. Also, the direct attribution of costs to intermediate operations makes it easy to determine what linguistic function instructions serve. The main inaccuracies are:

- Costs for loading stack and non-immediate constant arguments and saving stack results are charged to the VOP which operates on the values. This smears out the cost of non-register allocation, and makes it difficult to draw conclusions about the usefulness of register windows.

- Some VOPs contain embedded control structure, and these hidden blocks do not have counters. An estimate of the predicted average cost has been used in these cases; except for integer division none of these operations were frequent enough for the results to be substantially distorted.

6.1 The Benchmarks

The four benchmarks are: Richards, Cascor, Soar and Compile. Richards is a discrete-event simulation written in an object-oriented style; it is the only benchmark which is not a real program in daily use. Cascor is a float-intensive neural network simulator. Soar is an real AI toolkit used here to

Table 1: System Performance

	MIPS			SPARC	
Benchmark	*Size*	*Time*	*CPI*	*Time*	*CPI*
Richards	8k	0.5	1.0	0.8	1.8
Cascor	51k	79.6	1.4	122.0	2.8
Soar	383k	21.5	1.5	32.7	1.5
Compile	3755k	5.7	1.9	6.6	2.3

solve a toy problem. Compile is the CMU Common Lisp compiler.

7 System Performance Summary

Actual run times for uninstrumented benchmarks were collected on two workstations: a DECstation 5000/100 (MIPS) and a Sun4/65 SparcStation 1+. Both have a 25mhz clock rate, 24 megabytes of memory, and were running the Mach operating system. As the run-times in table 1 show, these machines are not equivalent, but of the available systems with the desired processors, they were the most similar. *time* is the run-time reported by the operating system, and *cpi* is a measure of the cycles per instruction, derived from the run-time, clock rate and executed instruction count. Presumably due to poorer memory system performance, the SS1+ is clearly slower on these benchmarks. Only a small part of the discrepancy can be accounted for by the implicit load interlocks on the SPARC.

8 Operation Frequencies

This section presents the most time-consuming operations of each benchmark when compiled unsafely on the MIPS. Unsafe compilation was chosen to clarify what each benchmark actually does (which may be obscured by checking overheads.) The MIPS compiler was used both because the R3000 is a "classic" RISC architecture and because the MIPS compiler backend for CMU Common Lisp is more mature.

Since there are over 500 different VOPs, reporting frequencies on a per-VOP basis provides too much detail to discern overall patterns. In this section, low-frequency VOPs are grouped into 27 classes; even after this reduction, most classes are so rare as to be uninteresting.

Table 2: Richards VOP Costs

VOP	Count	Cost	Percent
Load global	648,073	3.73	20.62%
`if-eq`	961,607	2.10	17.28%
`structure-ref`	1,093,353	1.66	15.54%
Function call/return	362,044	3.76	11.61%
Move	1,504,259	0.65	8.35%
Store global	204,952	3.00	5.25%
`structure-set`	347,354	1.52	4.52%
Simple type predicate	123,004	3.47	3.64%
`branch`	258,390	1.58	3.49%
Integer division	9,999	36.00	3.07%
Other	573,451	1.35	6.62%

8.1 Richards Results

Table 2 shows the cost breakdown for the unsafe Richards benchmark on the MIPS. *count* is the total number of VOPs executed in each VOP class. *cost* is the average cost in cycles of the VOPs executed (exclusive of cache misses, etc.) The cost for move VOPs is less than one because the register allocator satisfies many move operations without emitting any instructions. The cost for "function call/return" is misleading because function call is divided into three or four VOPs with differing costs. The actual cost for a function call in Richards is 12 cycles.

The "load global" and "store global" classes represent accesses to global variables. These references are relatively expensive in Lisp because the variable is represented by a heap-allocated object that must be loaded as a non-immediate constant. Lisp programmers often manually cache global values in local variables to avoid this overhead. Side-effect analysis of global variables would allow the number references to be reduced.

`branch` is an unconditional branch; in addition to its usual uses, `branch` also implements tail-recursive function calls. The cost of 1.58 reflects the fact that 58% of unconditional branch delay slots were not filled. `if-eq` is a conditional branch based on the equality of two registers (pointer identity.) The cost of 2.1 reflects branch delay noops as well as the cost of loading pointer constants. A "simple type predicate" is a conditional equality test of the tag bits in a descriptor and an immediate constant (three instructions

Table 3: Cascor VOP Costs

VOP	Count	Cost	Percent
Array read	103,091,109	2.88	22.25%
Load global	70,744,637	3.75	19.88%
Inline arith	161,941,099	1.29	15.65%
Inline compare less/greater	60,211,856	3.21	14.47%
Alien operations	4,557,819	20.89	7.14%
Array write	24,557,146	3.79	6.98%
Untagging	26,247,011	3.32	6.53%
`branch`	13,709,827	1.95	2.00%
Move	104,821,695	0.24	1.87%
`if-eq`	4,568,413	3.62	1.24%
Other	10,554,254	2.52	1.99%

+ delay.)

`structure-ref` and `structure-set` are the load and store operations for user-defined types. The cost for `structure-ref` (1.66) mainly reflects unfilled load delay slots due to indirection chains. In contrast, the cost for `structure-set` is greater than one because the stored value is often a constant which must be loaded.

8.2 Cascor Results

Table 3 shows the results for the unsafe MIPS Cascor benchmark. "Inline arith" is in this case single-float `+` and `*` and integer addition (for array indexing.) CMU Common Lisp does unusually well; some commercial implementations are 5x worse. Performance was actually within 25% of a C version of the same algorithm. Lisp float performance is terrible when many float values are heap allocated, so float-intensive Lisp programs are not common. The compiler uses type inference to determine expression types, then chooses to represent floats in float registers. Whenever the type is unknown, the descriptor representation must be used, and the float data is forced onto the heap.

8.3 Soar Results

Soar is a "classic" (i.e. old) Lisp program — it uses lists where it should use arrays, has no user-defined types, and is composed of many small functions.

Table 4: Soar VOP Costs

VOP	Count	Cost	Percent
Function call/return	12,668,884	5.02	20.80%
Move	67,951,847	0.56	12.50%
Integer multiplication	810,780	36.00	9.55%
Integer division	811,772	35.00	9.29%
`if-eq`	11,410,301	2.14	7.99%
Generic arithmetic	1,221,793	19.47	7.78%
`car/cdr`	13,022,497	1.52	6.49%
Tagging	1,624,489	9.99	5.31%
`branch`	6,296,285	1.59	3.27%
Complex type predicate	836,331	10.91	2.98%
Simple type predicate	1,668,546	4.96	2.71%
Inline `eql`	1,875,818	3.08	1.89%
Complex type check	982,393	5.63	1.81%
Simple type check	1,115,737	4.57	1.67%
Untagging	1,627,198	3.00	1.60%
Other	12,053,509	1.10	4.35%

It also turns out to spend 20% of its time doing integer multiplication and division, a fact unknown to the program's maintainer. It turns out that by changing the factor to a power of 2, these operations could easily be replaced with a shift/and sequence, however, no change was made when this was discovered because Soar is being reimplemented in C for more speed.

"Generic arithmetic" is used when numeric type declarations are absent. The compiler calls a runtime routine which verifies that the operands are fixed precision integers and does the operation. Any non-integer operands are passed through to a Lisp function whose activities would show up as other VOPs (in Soar all operands are integers.)

Soar is the most list-intensive of the benchmarks, and it only spends 6.5% of its time doing list operations. Most of the type testing and some of the function call overhead is due to calls to safely compiled Lisp library routines consisting of simple loops that should probably be inline expanded.

"Tagging" and "untagging" convert numbers (32-bit integers in Soar) to and from descriptors. The cost for tagging is exaggerated – most instructions in the sequence are rarely executed because the result normally can be represented as a fixed precision integer. Actually, in Soar all the values can

Table 5: Compile VOP Costs

VOP	Count	Cost	Percent
Function call/return	3,066,328	4.61	18.43%
Move	14,869,443	0.61	11.91%
Array type test	546,123	12.00	8.55%
`if-eq`	2,652,150	2.27	7.87%
Complex type predicate	422,368	9.24	5.09%
Generic arithmetic	219,444	16.75	4.80%
`structure-ref`	1,666,452	2.19	4.75%
Simple type check	705,971	4.21	3.88%
`branch`	1,762,318	1.44	3.31%
Simple type predicate	599,549	4.02	3.15%
Array read	874,683	2.74	3.13%
Array bounds check	466,361	4.27	2.60%
Inline compare less/greater	544,161	3.10	2.20%
`car/cdr`	841,927	1.74	1.91%
Allocation	76,239	18.23	1.81%
Inline `eql`	569,138	2.39	1.78%
Complex type check	162,095	7.91	1.67%
Load global	257,809	4.35	1.46%
`structure-set`	559,926	1.95	1.43%
List/string utility	37,642	29.00	1.42%
Other	4,759,645	1.42	8.83%

be so represented, but the CMU compiler avoids some generic arithmetic by doing 32-bit operations and then seeing if the final result fits back into 30 bits.

8.4 Compile Results

Table 5 shows that the compiler is less call-intensive than Soar, uses structures more than lists, and doesn't waste too much time on generic arithmetic (despite making considerable use of arrays.) The high frequency of "array type test" is anomalous, and suggests tuning opportunities. A "type check" VOP is another kind of type-testing conditional which traps when the test is unsuccessful. On the MIPS this is done by a conditional branch to an (uncounted) trap instruction.

Table 6: Safe System Performance

		MIPS		SPARC	
Benchmark	*Size*	*Time*	*CPI*	*Time*	*CPI*
Richards	37%	67%	1.0	50%	1.6
Cascor	41%	168%	1.3	161%	2.2
Soar	38%	18%	1.4	8%	1.6

9 The Cost of Safety

Table 6 is similar to table 1, but reports the increases size and run-time when the benchmarks are compiled with error checking code. There is a consistent increase in size, but the increase in run-time is much more variable. In these benchmarks, cycles per instruction is reduced in safe code because the checking code doesn't cause cache misses. Larger programs might show deterioration in instruction cache performance due to the increase in code size.

Soar shows the smallest performance degradation because its inner loops spend most of their time calling safe library routines and doing integer multiplication and division. Both Cascor and Richards make frequent global variable references, and in safe code global variable references must trap when the variable is not initialized. This increases run-time by 15%.

Type checking of user structures increases the Richards time by 42%. Since an indirection and pointer comparison is required, the cost is spread across the `if-eq`, `structurep` and `structure-ref` VOPs, with `if-eq` alone accounting for 20% of the increase.

Cascor shows the largest performance degradation; this is mostly due to dynamic array bounds checking and an array type check in the inner loop. A better type declaration would move the type check out of the loop, and reduce run-time by 28%. Cascor also spends 10x longer in the garbage collector due to increased heap allocation of floats, resulting in a 50% increase in run-time. This could also be eliminated by better declarations.

10 The SPARC Architecture

A comparison of performance on the MIPS and SPARC architectures is interesting because the SPARC incorporates tag checking operations designed for Lisp and Smalltalk.[10] As table 6 shows, the performance degradation

for safe code is smaller on the SPARC. On Soar, for example, the SPARC gains 3.9% in simple type checks because of its conditional trap instruction. And Richards gains 1.2% by using the tagged arithmetic instructions to eliminate 35% of the simple type checks. The SPARC also gains instructions on memory reference operations because no load noops are needed; this mainly improves code density.

CMU Common Lisp for the SPARC is not as well tuned as the MIPS version. It makes no attempt to use branch delay slots or to avoid load interlocks. This is less significant than it seems because the scheduler for the MIPS mainly eliminates load delays that correspond to non-interlocking loads. The SPARC register windows are also not used; several studies have found significant benefits from using register windows in Lisp and Smalltalk.[11, 12] Other Lisp implementations do use the SPARC's register windows, but there are unpleasant interactions with Lisp's non-local exit (exception) mechanisms because the register window pointer is not user accessible and the over/underflow handler is in the operating system.

With the current CMU compiler, the SPARC and MIPS architectures seem to be about even on safe code, since the MIPS tends to offset the SPARC type checking with a 4%–7% gain from saving an instruction in `if-eq`. The MIPS is that much faster in unsafe code, since the `if-eq` advantage applies, but the type checking operations don't.

11 Instruction Set Evaluation

Problems remain, but generally Lisp has benefited from RISC architectures, since they are in many ways more language-neutral:

- Because of Lisp's unusual calling convention and garbage collection, procedure call instruction are usually useless for Lisp.
- Because of Lisp's composition of datatypes via indirection rather than direct inclusion, favoring of linked lists over arrays, and garbage collection, complex addressing modes are usually useless.

Optimization of function call has traditionally been considered the Holy Grail of Lisp implementation, and great attention was given to function call in Lisp machine design. It is therefore interesting to note that a careful RISC encoding of Lisp's complex dynamically-linked call sequence takes no more time than on a Lisp machine. Moon[6] reports that the Symbolics 3600

took less than 20 cycles to execute a call; in CMU Common Lisp, the cost is 15 cycles (assuming cache hits.) There is a large space increase, however.

11.1 Code Size

Increase in code size is probably where RISC compares most poorly to past Lisp architectures. In the case of function call, the size has increased from 3 bytes to 48 (plus any register saving.) In the MIPS CMU Common Lisp image, function call accounts for 25% of the total code size.

Even ignoring stack-addressed byte codes (which may be 5x–10x more dense [13, 6]), significant gains are possible from denser instruction encoding. The IBM RT PC is a RISC architecture with 16 registers and 16-bit load and move instructions. RT CMU Common Lisp has 8.6 megabytes of code; when compiled for the MIPS R3000, the code size is 12.1 megabytes (40% larger.) On the R3000, a function which only calls another function is 57% larger.

C programs seem to exhibit only slight size increase from RT to MIPS. It is not clear why Lisp is so much more sensitive to code density. The largest contribution probably results from the inline expansion of function call and other 3–10 instruction sequences containing many loads, moves and equality comparisons that are not easily optimized away, but which can be densely encoded. A possible alternative to denser instruction encoding is ultra-fast (0 to 2 cycle) calls to "millicode" routines in the runtime system.

Poor memory locality in Lisp makes the code bloat problem even worse. A Lisp system is in effect a huge shared library; it exhibits poor locality because programs are linked against the library after the library has been built. Like CAD applications and EMACS, Lisp programs also tend to be interactive and to run for a long time. This prevents the operating system from using process destruction as a hint that code is no longer needed.

Dropping memory prices may eventually make code density irrelevant, but it hasn't happened yet. The combination of poorer code density with exploding functionality in programming environments has kept up with memory growth. Also, in comparison to disk latency, processor speed has greatly increased, making demand paging increasingly futile as a technique for exploiting code locality.

11.2 Conditional Instructions

Lisp makes extensive uses of conditionals, so some minor variations on conditional instructions can have a significant impact on Lisp performance:

- Pointer equality is an important operation. If `eq` takes two cycles rather than three, this speeds up Lisp 4%–6%.
- A trap conditional on an immediate equality/inequality test can be used for tag and argument count checking in safe code (savings 6% to 9%.)
- Many conditional tests are automatically generated by the compiler to check for exceptional cases. The outcome of these tests is highly predictable (>99%), so speculative execution features, annulled branch delay slots and branch prediction are useful.

11.3 Memory Instructions

Lisp implementations use constant offsets to subtract out tags and skip over headers. Ideally all memory referencing instructions should take constant offsets, but these uses are especially important:

- If load did not take a constant offset, the penalty would be large (30% or more), since explicit tag extraction would be required, and loading of non-immediate constants would become costly.
- Function call is helped by a computed branch instruction with an immediate offset (savings 1%.)
- Garbage collection makes it difficult to convert array references into pointer arithmetic. Loads and stores that compute the effective address by adding two registers and an immediate offset help array-intensive applications (savings load 5%, store 1.5%.)

11.4 Arithmetic Instructions

Arithmetic operations are important because they are frequent and potentially efficient:

- Programs that lack integer type declarations may spend 10%–80% of their time in runtime routines for generic arithmetic. Trapping tagged

add, subtract and compare instructions can eliminate this overhead. Explicit inline tag checks can reduce the overhead by more than half, but result in substantial code growth.

- Integer overflow detection is important even in programs with type declarations: it is difficult for the compiler to prove than arithmetic doesn't overflow, and Lisp semantics require graceful overflow into extended-precision arithmetic.

- When integer multiplication and division take many tens of cycles, a surprisingly large number of programs spend 3%–30% of their time doing these operations. Due to Lisp's emphasis on dynamically sized arrays, constant operands are less common than in other languages.

12 Conclusions About Lisp

- Improvements in compiler technology and changes in programming style (use of declarations) can reduce the overheads for type checking and generic arithmetic. These changes also allow efficient open-coding of floating-point and word-integer arithmetic, greatly improving numeric performance.

- The large semantic distance between Lisp operations and hardware instructions makes it difficult for programmers to develop an intuitive understanding of operation costs — the compiler must provide feedback to the programmer via efficiency notes.

- Modern, well-tuned Lisp programs often make less use of function call (18%), list operations (2%) and heap allocation (2%) than the contrived, small benchmarks used in previous studies. Greater use is made of user-defined types, arrays, and arithmetic.

- With good compilers and well-tuned programs, behavior is determined by what the program does, not by the language it is implemented in. Where Lisp programs do differ (larger, worse locality, more indirection intensive), these properties tend to be ones that all object-oriented programs are exhibiting to an increasing degree.

13 Conclusions About Architecture

- Some non-Lisp specific architectural features such as compare-and branch and conditional traps are quite useful. The argument for Lisp-specific instructions is generally not overwhelming.
- RISC architectures generally match Lisp better than CISCs because complex instructions are likely to have assumptions which make them useless for Lisp. Also, the prevalence of pointers makes complex addressing modes less useful.
- Type checking is not the most important thing. Instead, worry about that memory system: it is always too small and too slow.

References

[1] Chambers, C., and Ungar, D. Making pure object-oriented languages practical. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (1991), pp. 1–15.

[2] Layer, D. K., and Richardson, C. Lisp systems in the 1990s. *Communications of the ACM 34*, 9 (Sept. 1991), 49–57.

[3] MacLachlan, R. A. Cmu common lisp user's manual. Tech. Rep. CMU-CS-91-108, Carnegie Mellon University School of Computer Science, Feb. 1991.

[4] MacLachlan, R. A. The python compiler for cmu common lisp. In *ACM Conference on Lisp and Functional Programming* (Oct. 1992).

[5] Moon, D. A. Garbage collection in a large lisp system. In *ACM Conference on Lisp and Functional Programming* (1984), pp. 235–246.

[6] Moon, D. A. Architecture of the symbolics 3600. In *Proceedings of the International Symposium on Computer Architecture* (1985), pp. 76–83.

[7] Rees, J. A., and Adams, N. I. T: A dialect of lisp, or lambda: The ultimate software tool. In *ACM Conference on Lisp and Functional Programming* (1982).

[8] Steele Jr., G. L. *Common Lisp: The Language*, 2 ed. Digital Press, 1990.

[9] Steenkiste, P., and Hennessy, J. Tags and type checking in lisp: Hardware and software approaches. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems* (1987), ACM, pp. 50–59.

[10] Sun Microsystems Inc. *The SPARC Architecture Manual*, 1987.

[11] Taylor, G. S., Hilfinger, P. N., Larus, J. R., Patterson, D. A., and Zorn, B. G. Evaluation of the spur lisp architecture. In *Proceedings of the International Symposium on Computer Architecture* (1986), pp. 444–451.

[12] Ungar, D. M. *The Design and Evaluation of a High Performance Smalltalk System*. MIT Press, 1987.

[13] Wholey, S., and Fahlman, S. E. The design of an instruction set for common lisp. In *ACM Conference on Lisp and Functional Programming* (1984), pp. 150–158.