Lisp-in-Lisp: High Performance and Portability

Rodney A. Brooks
Richard P. Gabriel
Guy L. Steele Jr.

1. Introduction

Lisp remains the primary programming language for artificial intelligence research, and, as new computer architectures are developed, the timely availability of a high-quality Lisp on these machines is essential.

Until recently every Lisp implementation has had either a relatively large assembly language core or extensive microcode support. We have been working on a Common Lisp [Steele 1982] for the S-1 Mark IIA super-computer being developed at LLNL, producing an implementation that has high performance, exploits the complex architecture of the S-1, and which is almost entirely written in Lisp [Brooks 1982b]. We call such a Lisp a *Lisp-in-Lisp*.

2. Implementing Lisp

There are a variety of approaches for implementing Lisp on a computer. These approaches vary both with the architectural support for Lisp by the target computer and with the philosophy of the implementors. All of these approaches, so far, require a compiler; we claim that the most portable approach relies most heavily on its compiler—since one wants to compile as much of the runtime system as possible, the coverage of the compiler and the efficiency of the code it produces are critical. Thus, the performance of a portable Lisp depends on the performance of its compiler.

At one end of the architectural support scale, the computer can support user microcode that implements a Lisp-oriented machine. In such an architecture, primitives, such as CONS, can be written in microcode and the Lisp compiler can simply place this instruction in the generated instruction stream rather than either a sequence of instructions that perform the operation or a function-call to a hand-written subroutine for the operation. Often the microcode emulates a register-free stack machine, and compilers for this type of architecture can be fairly simple. Some examples of this sort of architectural support are the Dolphins and Dorados built by Xerox.

At the other end of the scale are stock-hardware architectures where the instruction set is fixed, and this instruction set, along with the addressing modes, are designed to provide good support for a variety of languages. Typically these architectures have a number of registers and support one or more stacks. Registers are normally constructed from faster technology than the rest of the computer (excepting any cache), and they are physically, as well as logically, closer to the ALU and other functional units of the CPU. Therefore compiler-writers for high-performance Lisp implementations on these machines face a register-allocation problem.

Most compilers for stock hardware are structured so that a small number of primitives are directly compiled into assembly language, and all the other constructs of the Lisp are expressed in terms of those primitives. Each of the primitives needs a 'code generator' which produces the instruction stream that implements the primitive. In addition, a body of code is needed to emit code to move data from place to place; the code generators are parameterized to use these functions to emit code to move data to canonical locations when the actual locations are not usable directly.

Some examples of this level of architectural support are the Vax 11/780 and personal computers based on MC68000 microprocessors.

Along this spectrum lie computers such as the S-1 and the PDP-10. The PDP-10 has only a few features convenient to Lisp, but these have proven useful. The partitioning of the 36-bit word into 18-bit halves, with the addresses being 18 bits means that a CONS cell occupies a single word. The instruction classes HLR and HRR implement exactly CAR and CDR. The simple stack structure and instructions serving them implement well the spartan function-calls that are used.

The S-1 is farther along this spectrum towards microcoded machines: the 36-bit word contains 5 bits for tagging, and there is a Lisp function-call instruction. The tagging is supported by several instructions.

In interpreter-centered systems all of the primitives are hand-coded as subroutines that the interpreter calls. The compiler simply emits code that calls these functions. The remainder of the compiler mostly implements LAMBDA application—in particular, binding. In some ways, these traditional, interpreter-centered systems are like current microcoded machines: the hand-coded subroutines function as instructions implementing an abstract Lisp machine.

Those are the current extreme points of architectural support; in the future even more architectural support is possible, such as direct execution of Lisp.

The extreme points of philosophy derive from different values placed on the perceived efficiency versus the clarity of code (and the concomitant ease of writing it). Implementing a large portion of the Lisp system in assembly language or in microcode provides an excellent low-level programmer an opportunity to exploit the architecture completely. Microcoded machines often allow type-checking to occur in parallel with other operations; this can allow type-checking in more circumstances without efficiency loss than might be possible in a stock-hardware implementation, and the microcoded implementation can be made 'safer.'

With a Lisp-in-Lisp, the other philosophical extreme, efficiency is partially forfeited for ease of writing, ease of certification, and ease of portability. The observation is that Lisp programmers have already accepted the compiler as sufficiently efficient for their own code, and with a Lisp-in-Lisp they only sacrifice some interpreter and storage-allocation speed. As compiler technology advances this sacrifice may diminish.

3. Our Implementation

Our implementation is centered on three major tools; a compiler, an assembler and a cold-loader. All of these are written in a common subset of Common Lisp (our target Lisp) and MacLisp.

The Lisp system is created by compiling all of the functions that define it, assembling them, and constructing the binary image of an initial system on a disk file.

Our compiler [Brooks 1982a] is a sophisticated optimizing compiler based on the Rabbit Compiler [Steele 1978] and on the Bliss-11 compiler [Wulf 1975].

Since the kernel of the implementation manipulates data tags and generates pointers, the compiler must open-code special *sub-primitives.* For example, there are sub-primitives to perform arithmetic on 36-bit quantities, and to set and retrieve tag and data fields—turning them into FIXNUMs.

The code generators in the compiler have no detailed knowledge of the computer's 89 addressing modes. An optimizing assembler takes care of such details; it performs branch/skip optimizations and outputs various literals that must be created upon loading.

The cold-loader builds an initial core image in a disk file. It first builds an internal data structure for a skeleton core image which includes a stack with a stack frame for a start up function, a vector of initial register values, copies of special atoms such as NIL and T, and a kernel of the tables needed for storage management. It writes the associated word values into the output core-image file. Then it uses the same linking loader (compiled in MacLisp) that is used in the target Lisp environment, with file-output routines substituted for memory-writing routines, to load assembled Lisp files into the initial core-image. The storage-allocation routines of the linking loader access the simulated data structure rather than the in-core tables they access when used in the target Lisp. In this way the linking loader, a stream-based I/O system, a lexical interpreter, a reader, a printer, some debugging tools, and part of the storage-allocation system and garbage-collector interfaces and tables are loaded into the initial core-image disk file.

4. Advantages

There are two major advantages to Lisp-in-Lisp: the first concerns the ease of writing correct Lisp code for the system; the second concerns portability.

With Lisp-in-Lisp, difficult pieces of code can easily be written correctly. Typically the most difficult code to write and debug (or prove correct) is the garbage collector and storage-allocation routines. The garbage collector, since it operates on data not available to normal Lisp code, has traditionally been written in assembly language. A Lisp-in-Lisp system defines a set of sub-primitives that operate on pointers directly, and the compiler open-codes them.

Implementing sub-primitives as compiler code generators has three advantages. First, the template of code that is correctly supplied will be correctly applied in more situations than were anticipated by the writer of the code generator. The code-generator writer produces a template or an abstraction of the code sequence needed to perform the action; the compiler, and the register allocator in particular, can then supply the addressing modes and data transfers needed to correctly apply the abstract sequence of actions in any situation.

Second, routines written in a high-level language can be proven and debugged more easily. Relatively large-scale modifications can be made without worry about early commitment to storage layout or register assignments.

Third, code can be tested within an existing Lisp environment using its programming tools and with the knowledge that it is a correct language and environment. Thus errors in the implementation design and code generators can be isolated from errors in the Lisp code, the latter being eliminated while testing in the existing Lisp system.

There is a third spectrum in addition to the architectural support and philosophical outlook spectra mentioned above—the portability spectrum. Along this spectrum we claim that the microcode-supported Lisp systems and the large runtime-supported Lisp systems occupy one end (the least portable end) and Lisp-in-Lisp systems occupy the other end (the most portable end).

The task of writing microcode is comparable to the task of writing a large runtime system; when portability is needed, a compiler that assumes code generators for sub-primitives can be more easily portable than one that produces code for a stack machine. For a compiler that produces code for a stack machine, each abstract-machine instruction emitted requires microcode or macrocode support, or else each such instruction must be expanded into native machine code (as CMACROS in PSL are expanded).

If each abstract-machine instruction is expanded as part of the last, code-producing pass of the compiler, then register-allocation decisions depend only on earlier register-allocation decisions. This renders high quality register-allocation and, hence, high performance difficult to achieve.

In a Lisp-in-Lisp environment the large runtime system is written in Lisp and only the components needed to piece that system together (the sub-primitives) are hand-coded as simple code generators. Therefore the portability of a Lisp-in-Lisp system is the highest at the end of this spectrum.

Portable Standard Lisp (PSL) [Griss 1982] is the closest to the S-1 Lisp in terms of being a true Lisp-in-Lisp. The S-1 implementation has a more advanced compiler and is, therefore, of higher performance. InterLISP-D [Burton 1980] [Moore 1976] is a Lisp-in-Lisp system where the sub-primitives required are written in microcode. InterLisp-Vax uses the InterLisp-D Lisp-in-Lisp code, but it demonstrates compiler/architecture mismatch [Masinter 1981] [Gabriel 1982] [Gabriel 1983].

T [Rees 1982] uses an early version of the S-1 compiler, adapted to the Vax.

5. An Example

The S-1 compiler's internal language is an expression language—a graph easily derived from the tree structure of the standard internal representation for Lisp code. Each node in the graph represents a Lisp language construct, and backpointers to other nodes may be present to link interesting pairs of nodes (such as lambda-binding nodes and variable-reference nodes). There is no commitment to any particular architecture in this scheme. Moreover, the register-allocation is table-driven, and no part of the compiler assumes any registers exist.

The following function illustrates the advantages and style of Lisp-in-Lisp. It demonstrates the efficiency of the S-1 compiler by comparing the output of the compiler for a common runtime function with a hand-coding of that function. +& takes any number of fixnums and returns their sum. Ignoring type-checking of arguments, the following code provides the definition for the interpreter:

```
(DEFUN +& (&REST NUMBERS)
       (DO ((NUMS NUMBERS (CDR NUMS))
            (SUM 0 (+& SUM (CAR NUMS))))
           ((NULL NUMS)
            SUM)))
```

where the use of +& in the function definition is open-coded.

This code is easily seen to be correct whereas the assembly language coding of it may be difficult to do correctly, especially on an architecture as complex as the S-1.

The compiler-produced code for this function comprises 14 instructions. One tests that the function was called properly; two instructions CONS up the &REST argument into a list; two instructions move into registers the initial values for NUMS and SUM; three instructions manage the loop and perform the computation; five instructions set the tag field for the return value and return from the function; and one instruction is a jump from the beginning of the code to the end test, which appears at the bottom of the loop code.

A hand-coded version of this function displays one major optimization: It eliminates the call that listifies the &REST argument by taking the arguments off of the stack directly. We do not see an easy way that the compiler can perform this optimization in general, at the moment.

6. Difficulties

The key to the success of a Lisp-in-Lisp portable to a variety of machines is the existence of a portable compiler. With such a compiler the job of writing code generators and tables should be much less than the job of building or microcoding a computer on one hand, and much less than writing a large runtime system on the other.

Assembly language code may need to be written in addition to the Lisp code. In S-1 Lisp the CONSing routines have been hand-coded for efficiency and are accessed through a simpler function-call than normal Lisp functions.

Our approach demands an existing Lisp implementation that is compatible with the one being developed. We use MacLisp [Moon 1974].

In the absence of a portable compiler, a desirable component for a Lisp-in-Lisp implementation is a compiler that produces efficient code. One can start with a simple, but correct, Lisp compiler and move to a highly optimizing compiler later.

7. Statistics

Our compiler understands 15 special forms and open-codes 316 Lisp primitives, 10 of which are subprimitives that user-level code would not normally use. The initial environment is 10,300 lines of assembly language code; 413 lines—about 4%—are hand-written.

We could have written the whole Lisp system in Lisp and written enough code generators to have every instruction generated by the compiler. We hand-coded certain portions in assembly language for three reasons:

— Efficiency. To avoid the overhead of a fully general Lisp function-call for such common functions as CONS, LIST, LIST*, and all of the number-CONSers (they must interact with the storage allocation data structures so they cannot be open-coded), a special, fast procedure-call mechanism is provided, and the CONSers themselves are carefully hand-optimized. The compiler compiles CONS, for instance, as a fast procedure call. Other, less common, CONSing functions (e.g. CONS-IN-AREA) are completely written in Lisp. The CONSers account for 111 lines of code.

— Single use. Certain primitive operations, most notably those that interface with the operating system I/O facilities, need only be referred to by one piece of Lisp code. Rather than write code generators for these primitives it is just as easy to write the code directly. These primitives open, close, and delete files; transfer ascii characters and quarter words to and from buffers and thence to and from files; and transfer ascii characters to and from the terminal. These account for 148 lines of code. Trap handlers also fall under the single-use category and account for 87 lines of code.

— Long code sequences. Certain common operations lead to identical and lengthy code sequences. The fast procedure-calling mechanism developed for the CONSers is used to replace these sequences by jumps to single copies of them. There are two classes of such code sequences. S-1 Lisp uses deep-binding with caching [Gabriel 1982]. 34 lines of code provide functions to lookup, bind and cache special variables on the stack. The function-return interface for multiple values requires vectors and lists to be unbundled onto the stack and into registers in a particular way; 33 lines of code provide these functions.

8. Conclusions

We have implemented a Lisp-in-Lisp on a complex-instruction-set computer. The bulk of development time has been put into the optimizing compiler, which has been written with a clean partition between the machine-independent and machine-dependent parts. The remainder of the Lisp system required very few man-months to write and debug.

The speed of modern computers minimizes the need to squeeze every ounce of efficiency from them; the need to produce correct, understandable, and modifiable Lisp implementations rapidly is increasing. As more architectures become available to the AI community the methodology we have used in this project should become more widespread.

References

[**Brooks 1982a**] Brooks, R. A., Gabriel, R. P., Steele, G. L. *An Optimizing Compiler For Lexically-Scoped Lisp*, Proceedings of the 1982 ACM Compiler Construction Conference, June, 1982.

[**Brooks 1982b**] Brooks, R. A., Gabriel, R. P., Steele, G. L. *S-1 Common Lisp Implementation*, Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, August 1982.

[**Burton 1980**] Burton, R. R., Masinter, L. M., Bobrow, D. G., Haugeland, W. S., Kaplan, R. M., Sheil, B. A., Bell, A. *Overview and Status of InterLISP-D (Dorado and Dolphin)*, Proceedings of the 1980 ACM Symposium on Lisp and Functional Programming, August 1980.

[**Gabriel 1982**] Gabriel, R. P., Masinter, L. M. *Performance of Lisp Systems*, Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, August 1982.

[**Gabriel 1983**] Gabriel, R. P., Griss, M. L. *Lisp on Vax: A Case Study*, in preparation.

[**Griss 1982**] Griss, M. L., Benson, E. B., Maguire, G. Q. *PSL: A Portable Lisp System*, Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, August 1982.

[**Masinter 1981**] Masinter, L. **Interlisp-VAX: A Report**, Department of Computer Science, Stanford University, STAN-CS-81-879, August 1981.

[**Moon 1974**] Moon, David. **MacLisp Reference Manual, Revision 0**, M.I.T. Project MAC, Cambridge, Massachusetts, April 1974.

[**Moore 1976**] Moore II, J S. *The InterLISP Virtual Machine Definition*, Tech. Rept. CSL 76-5, Xerox Palo Alto Research Center, Palo Alto, Ca., Sept. 1976

[**Rees 1982**] Rees, J. A., Adams, N. I. *T: A Dialect of Lisp or, LAMBDA: the Ultimate Software Tool*, Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, August 1982.

[**Steele 1978**] Steele, Guy Lewis Jr., *RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization)* Technical Report 474, Massachusetts Institute of Technology Artificial Intelligence Laboratory, May 1978.

[**Steele 1982**] Steele, Guy Lewis Jr. et. al. *An Overview of Common Lisp*, Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, August 1982.

[**Wulf 1975**] Wulf, William; Johnsson, Richard K., Weinstock, Charles B., Hobbs, Steven O., and Geschke, Charles M. **The Design of an Optimizing Compiler**, Programming Language Series, Volume 2, American Elsevier, New York, 1975.