Sleeping with the Enemy: Lisp and C in a Large, Profitable Real-time Application

John R. Hodgkinson
Gensym Corporation
125 Cambridge Park Drive
Cambridge MA 02140
(617) 547-2500
jh@gensym.com

Abstract

We examine the integration of Lisp and C in a commercial application, with a view towards the "proper place" of each language in large, practical applications in general. Topics addressed include portability, flexibility, efficiency, compactness, and user base of the two languages. We outline the pivotal role of Lisp-to-C translation tools in integration efforts. An attempt is made to dispel some half-truths about Lisp and inflated claims about C promulgated by interested parties.

I find it strange that the Lisp community can't work together to defeat a common enemy. The enemy is C. Not C used for its proper purpose, as a universal machine language, but C touted as a high-level programming language. Every claim made for C as a development language — portability, flexibility, efficiency, compactness, widespread use — needs qualification. This would be an interesting academic fact, but for one thing. C and its scions like C++ are winning the battle to dominate the programming landscape; indeed, in many areas they have already won. And during the fighting, C has promulgated a slew of half-truths and utter lies about its competitors, especially Lisp.

First, credentials. Gensym produces G2, an expert system shell for large real-time data applications in many industries, including but not limited to chemical, nuclear, aerospace, food processing, and traffic control. G2 is delivered on a plethora of workstation platforms, including Sparc, Vax and Decstation, HP 9000 series, IBM PowerStation, and 80386 machines. The actual G2 product is a binary image produced by compiling and linking C code. The original source code for G2 is a mixture of C and Lisp, with Lisp predominating. The Lisp section is translated into platform-independent C code before the compilation on the different platforms. The software necessary to build and maintain G2 is an integration of third-party Lisp-to-C translation software and in-house tools written in Lisp, C, and VMS and UNIX command scripts. In short, G2 is a product where Lisp and C both play a part.

Now, into the fray. C has won the hearts of decision-makers with its claims of portability. With C, we are told, one set of source code fits all platforms. No longer does a company have to hire experts who specialize in one or another arcane platform, just to deliver a needed application there.

They only have to hire C programmers, and schools are turning these out by the bucketful. Our effort to translate a sizable Lisp program into C has not borne out these all-too-prevalent platitudes. This is not to say that our uniform set of C sources gives us no edge in porting. Indeed, some ports to new platforms have occurred in a matter of person-days, where comparable ports previously took weeks, months, or were even deferred as not then worth the cost.

But every time we port, the C-is-portable myth takes another hit. First, translated C code can never be as readable as the original Lisp code. The first casualties are the many Lisp macros that encapsulate the abstractions and algorithms necessary to build and run G2. These simply cannot be expressed in C. The C macro facility, #define, is a limited text-to-text transformer, since C completely lacks the notion of programs as data. This may seem a mere theoretical ornament, until the first time you try to obtain the functionality of a Lisp macro in C. Even the simplest expansion, say an analogue of progn with an arbitrary number of forms to be sequentially executed, is beyond #define's ken.

And complicated macros like loop fare even worse. Loop is implemented in Lisp itself as a set of macros that provide a concise yet efficient way to express iteration paths, some of which can be quite complex. C has no comparable facility. The result is that we have to macroexpand our loops in Lisp first, then translate them into C. All the underlying machinery is now visible in the translated code, including previously hidden lexical variables, unwind-protects, and tagbody labels. The resulting code is hard to read, debug, maintain, and extend. The workaround, of course, is a preprocessor to C that establishes a new, readable special form. But the moral of the story is that C cannot be significantly extended in C, whereas it is a simple matter to extend Lisp in Lisp.

Experience also reveals the depressing fact that system calls are different on each platform, even those purportedly adhering to UNIX standards. Sections of our code that call out to the operating system, say for time information or to establish network connections, have to be compiler-switched by platform, no matter how similar the man pages are across those same platforms.

Aside from an inadequate macro facility and the lack of truly standard system calls, C compilers often have problems emulating the semantics of translated Lisp constructs. One of the most prevalent such construct in our code is (setf svref), assignment to a simple vector reference. The straightforward translation uses C array access after taking a fixed offset to cover the Lisp header of the array. Since ANSI C does not specify the order of execution when such constructs are nested, we cannot use this straightforward translation, but instead have to resort to a verbose series of temporary variables. As mentioned before, C macros are no help in cleaning this up. Even when ANSI does make a pronouncement, there can still be problems. Many C compilers have pure bugs in the comma operator for sequential evaluation, even though the specification is quite clear. We have also encountered problems with argument order for indirect function calls and initialization expressions for register variables.

Sometimes we solve these problems with the compiler vendor, sometimes by requesting an enhancement from the makers of our Lisp-to-C translator. For instance, one useful enhancement has been translator switches which relax certain Lisp semantic guarantees, such as left-to-right argument evaluation, where efficiency is crucial. But the most reliable solutions take the form of modifying Lisp code to ensure simple translations. We often do this by shadowing Lisp symbols for special forms with our own macros, which insert explicit temporary variable bindings. In other words, we often use Lisp to ensure the portability of C, not the other way around.

The preceding is not to be taken as condemning the Lisp-to-C translation approach. On the contrary, we feel it has allowed us to use the best of both worlds — advanced and widely available C compiler technology on the one hand, flexible Lisp abstractions on the other. But in the process we have not found C to be any more "portable" than the Lisp from which it is translated.

With the rallying cry of "portability!" muted, at this point C proponents next bring up the myth-encrusted issue of memory management. Lisp uses some form of voodoo memory management that pauses at unpredictable times and is unfit for any practical application, they say. Again, actual experience can shed some light on this claim. Lisp does contain a facility called garbage collection, which enables it to reclaim memory when that memory can no longer make a difference in future computations. Lisp does not enforce this facility on its users; if the user keeps pointers to allocated memory and reuses them where necessary, garbage collection need never happen. C on the other hand ignores the entire issue of memory management. If a user — say an X server, to take a random example — continually calls malloc() without a corresponding free(), the system will crash, predictably. C's lack of memory management encourages this kind of profligacy, even in large programs whose designers should know better.

It is worthwhile to note that present Lisp garbage collectors are not suited for real-time systems of any complexity. Briefly, any Lisp which contains arbitrary-length data structures (arrays are the most common example) needs to pause an arbitrary length of time during reclamation, even when the garbage collection is incremental. Misleading claims to the contrary, C has exactly comparable behavior: malloc() can take an arbitrary amount of time to de-fragment memory enough to find a contiguous block of the required length. Thus, in order to obey real-time constraints, our application eschews garbage collection altogether. It is entirely possible to write code in Lisp which does not trigger the Lisp garbage collector. In addition, the extensibility of Lisp allows us to manage memory in several convenient and efficient ways.

The simplest way to manage memory is as C does, with explicit allocation. This is the approach we take with Lisp symbols. We provide G2 users with a way to create a symbol, based on the Lisp intern function. Reclaiming a symbol is more problematic. Giving users an explicit unintern is not the solution, since they could unintern a symbol that is still in use elsewhere. Instead, we give users the ability to measure the amount of memory devoted to storing symbols, so that they can measure their run time growth, and we recommend that users pool symbols for future re-use in situations where creating symbols at run time is unavoidable. Incidentally, although Lisp typically does garbage-collect uninterned symbols, that ability is not necessary for our purposes.

Another memory-management method, this one unique to Lisp, is the dynamic-extent declaration. In implementations that obey this construct, we can inform the compiler that memory bound to a given variable will no longer be used outside of a given lexical scope. We are, however, concerned with all memory allocation, not just memory that is bound to a variable. We have therefore extended the Lisps we use to handle the allocation events in which we are interested. Where necessary, we have requested extensions of the dynamic-extent concept from our vendors, extensions which permit temporary allocation of floating-point numbers and infinite-precision integers within a lexical scope. On exit from that scope, the compiler arranges to dispose of any memory used.

When our needs are too specific to warrant vendor extensions, the extensibility of Lisp comes to the rescue again. We have added tools which permit explicit reclamation forms inside a lexical scope. And when all else fails, Lisp makes it easy to rewrite those facilities which drop memory references uncontrollably. We have completely rewritten the Lisp pathname facility to use reclaimable pathnames, and have rewritten parts of the stream and error-handling facilities not to allocate garbage-collectable memory.

All systems face issues of memory management. Lisp is a language which permits users to address these issues head-on where necessary, and otherwise allows them a sophisticated default behavior. C is a language which gives users unabstracted memory-allocation primitives and not much else.

C adherents have another arrow in their quiver: everybody "knows" that Lisp arithmetic is generic and therefore slow, certainly slower than a language as close to the underlying machine as C. A less contentious way of saying this, however, is that Lisp arithmetic lets us ignore the machine or not, depending on the needs of the application. When generic arithmetic is needed, Lisp provides it. When fast, type-specific arithmetic is required, that is available too, in a variety of ways. One way is through explicit type declaration. Lisp macros allow us to build entire suites of type-specific arithmetic operators. Forms using these macros expand into type-declaring forms which in turn give the necessary information to the compiler. Since in Lisp data is typed, not variables, the same variable may be treated generically or type-specifically at different branches in the code, as more is known about it.

Lisp-to-C translation provides another way that Lisp can access the underlying machine, and thereby improve performance. Lisp represents objects either as immediate data (in the case of limited-precision integers, characters, and the like) or as pointers (when larger amounts of memory are involved). Since a double-precision float is larger than the size of a pointer in most systems, Lisp usually represents floats as pointers to memory allocated elsewhere. This process is called "number-boxing." C supporters are correct in assuming that number-boxing can be inefficient, since unboxing a number to perform immediate arithmetic on it, then re-boxing it so that Lisp can use it again, can consume needless instructions. C supporters are incorrect, however, in assuming that Lisp contains no way around this problem. Given flexible translation software, we have written subprimitives that allow the mutation in place of an immediate double-precision float. Then we can establish lexical or dynamic scopes within which all floating-point arithmetic is immediate. In that way, we retain useful abstractions, but where necessary we can get as close to the machine as C does.

The chant of "Lisp is slow" has been muted, but "Lisp is fat" continues, although abated somewhat by Lisp-to-C translator technology. Lisp delivery methods admittedly lag those of C. The size of a Lisp image in virtual memory is excessive in some implementations, due to a hemispheric garbage collector. As well, Lisp workspaces saved on disc often seem larger than need be. For instance, the saved Lisp image we used to deliver, even after excluding facilities like the compiler and editor, was larger than the linked C image we now deliver by a factor of three.

But using a Lisp-to-C translator straight out of the box is not a complete panacea. An application linked directly from C binaries uses only those library component files which contain referenced functions. In many C applications, this can bring about a fair degree of automatic "tree-shaking." But G2 uses functions from nearly every facility in Lisp, obviating much of this gain. And although this is not the case in G2, Lisp applications often call functions by indirecting through the function cell of a symbol. This practice also increases image size and poses difficulties for tree-shaking programs, especially in those applications that haven't been written to funcall compiled functions wherever possible. Finally, the convenience of Lisp macros and inline declarations is here seen as a two-edged blade, since without metrics for the time-space tradeoff, inlining can consume more space than it's worth.

We have found that careful measurement at the machine level can go a long way towards alleviating many deliverability problems. Most C compilers offer profiling options, which provide extensive quantification of time spent in groups of function calls. As well, there are utilities which log the amount of space consumed by individual functions in C binaries. These standard tools allow us to effect time-space tradeoffs in a fine-grained way, sometimes platform by platform. The trick is not to deliver in C no matter what. Rather it is to use C where its strength lies, instead of forcing it into the uneasy role of high-level development language.

A final cautionary note. Although G2 is object-oriented, both in its internals and in its user model, it does not use CLOS. This is not because we don't admire CLOS, or because a production model would not suit our needs. On the contrary, we can see many advantages in the CLOS paradigm. For example, a standard, efficient meta-object facility would provide an ideal way to implement reclaimable objects. The reason we don't use CLOS is that it is still unproven technology. To us, a "proven" object system would (among other things) do methods dispatching efficiently. That is, it could establish that a generic function call is as fast as a normal function call in the degenerate case (where arguments are all type t), and no slower than a function call and a typecase clause with a static class hierarchy. Although C++ does not meet our needs either, since all it offers is static method combination, no one can dispute that it is winning the marketplace. CLOS is the first major paradigm that Common Lisp is attempting to standardize since its inception, but attaining acceptance as widespread as C++ will be an uphill battle.

In closing, imagine a world without Lisp, a world in which C has triumphed. The last AI application has been ported to C and its original source code thrown away. Car, cdr, and cons have fallen on the ash-heap of history. The last elegant CLOS prototype has been ported to C++. Things will go fine at first. Indeed, the economy will welcome the influx of new programmers needed to maintain the ported code. Programmers asked to extend the code, however, will have a harder time of it. They will marvel at some of the strange blocks of almost-repeated code resulting from long-forgotten macroexpansions, wonder at the clever indirect function calling through long-lost symbol function-cells, yearn for dynamic method combination and some control over metaclasses. They will despair of being able to express anything as cleverly in C. Let us hope one of them stumbles across that silver book in an argon-filled time-capsule. If Lisp did not exist, it would be necessary to re-invent it. Since Lisp does exist, let us take advantage of it.

References

Allard, James R., and Lowell B. Hawkinson, "Real-Time Programming in Common Lisp," Communications of the ACM, September 1991, vol. 34, no. 9.

Booch, Grady, Object-Oriented Design, Benjamin-Cummings Publishing, Redwood City, California, 1991.

Gabriel, Richard P., "Lisp: Good News, Bad News, How to Win Big." AI Expert, June 1991, vol. 6, no. 6.

Gabriel, Richard P., Performance and Evaluation of Lisp Systems, MIT Press, Cambridge, Massachusetts, 1985.

Gensym Corporation, G2 3.0 Reference Manual, Cambridge, Massachusetts, 1992.

Harbison, Samuel P., and Guy L. Steele Jr., The C Language, 3rd edition, Prentice Hall, Englewood Cliffs, N.J., 1991.

Keene, Sonya, Object-Oriented Programming in Common Lisp, Addison-Wesley Publishing, Reading, Massachusetts, 1989.

Kiczales, Gregor, Jim des Rivieres, and Daniel G. Bobrow, The Art of the Metaobject Protocol, MIT Press, Cambridge, Massachusetts, 1991.

Lee, Peter, ed., Topics in Advanced Language Implementation, MIT Press, Cambridge, Massachusetts, 1991.

Lieberman, H., and C. Hewitt, "A Real-Time Garbage Collector Based on the Lifetimes of Objects," Communications of the ACM, June 1983, vol. 26, no. 6.

Steele, Guy L. Jr., Common Lisp: The Language, 2nd Edition, Digital Press, Bedford, Massachusetts, 1990.

Stroustrup, Bjarne, The C++ Programming Language, Addison-Wesley Publishing, Reading, Massachusetts, 1987.