Inside Interlisp: Two Implementations

L. Peter Deutsch
Xerox Palo Alto Research Center
November 26, 1978

1. The Interlisp virtual machine

Interlisp and Standard Lisp (Griss, Utah) are the only Lisp dialects for which anything like a comprehensive functional specification exists. The Interlisp Virtual Machine (VM) document speaks for itself: only the highlights appear just below.

The non-technical qualities of Interlisp are unique and deserve mention. Interlisp has an exhaustive and well-organized reference manual, which is available on-line to answer questions about particular functions or (to a lesser extent) topics. It is only through amazing amounts of labor expended on this manual that the proliferation of features in Interlisp has remained usable.

1.1 Data types

Interlisp provides list cells, litatoms (symbols), fixed-size integers, fixed-size floating point numbers, character strings, arrays of pointers or integers, hash tables, and environments as predefined data types. The user can define his own packed record types: each type consists of any combination of pointers and non-pointers in a fixed layout. All the primitive operations are supposed to check the type of their operands, but Interlisp-10 omits the check on many "read" accesses (such as CAR and CDR) for efficiency.

Literal atoms are unique over the entire system -- there is no OBLIST. They may not have printnames which would read in as numbers. Each atom has an associated global value (NOBIND means none), property list (initially NIL), and function definition cell (initially NIL). CAR and CDR of litatoms are formally undefined (except for NIL). Litatoms and strings are essentially interchangeable for character operations, except that one cannot change the printname of a litatom. NIL is a litatom whose CAR and CDR are both fixed at NIL: attempts to change them are reported as errors. NIL also serves as "false"; anything other than NIL is "true". T is an ordinary litatom whose global value is T. NIL and T may not be rebound.

Some integers are implemented by "boxing"; programs can share pointers to boxes and smash values into boxes. (Like many such functional anomalies, this one crept in as an implementation artifact and was allowed to become official for efficiency reasons. It has caused a lot of obscure trouble.) Interlisp provides separate integer and floating point arithmetic operations, as well as the familiar PLUS etc.: these operations always fix or float their operands respectively, and compile more efficiently.

Strings are likewise implemented through descriptors, and some facilities exist which smash descriptors. The substring operation creates a new descriptor which shares the string characters with the old one, so replacing characters through one descriptor can affect characters accessed through another.

Arrays in Interlisp-10 may contain a pointer part and a non-pointer part, although the VM only requires the existence of homogenous arrays. The access functions must check the index to see which part it lies in.

Hash tables use EQ to compare keys. The garbage collector will remove entries from hash tables if the key is no longer referenced by anything else.

1.2 Environments and the interpreter

Environments (pointers to stack frames) again involve visible descriptors. Interlisp allows one to move freely up the call or access chain, to return to any frame, to evaluate expressions in any context, or to examine or alter the value (or name!) of any variable in any frame. The "spaghetti stack" concept requires that when control returns to a frame that has more than one reference to it, the temporary values (but not the argument variables) of the frame are copied.

The interpreter is required to leave its working variables in stack frames in a specified form, so that DWIM can examine and even alter them (e.g. to provide better recovery from parenthesis errors).

All errors in Interlisp send control to a user-definable function which can resume the computation cleanly. There is a higher-level facility for attempting a computation and regaining control at the point of the attempt if an error occurs.

Interlisp-10 uses shallow binding for variables. The user can declare variables as purely local (not accessible outside the function that binds them): this is the default for block compilation, but not for ordinary compilation, and it is not supported by the interpreter.

1.3 File system

Interlisp supports a somewhat implementation-independent file interface based on the Tenex file system. Files may be opened by name, which may include supplying defaults for unspecified fields (e.g. directory, version). There is a somewhat confusing set of rules for determining the referent of an incomplete file name. I/O may only be done to a file while it is open. Functions are available for deleting, renaming, and reading and setting attributes of files (e.g. length), and for scanning through the directory system.

Interlisp-10 supports communication through the ARPANET through the file system.

1.4 Symbolic I/O

Interlisp supports a complex table-driven I/O system for symbolic data. Input, and to a much lesser extent output, is controlled by an object called a *readtable* which specifies things like what characters (if any) have the functions of brackets or atom separators, what characters are to be treated as read-time macros, etc. Symbolic I/O also keeps track of horizontal line position and inserts end-of-line characters automatically, and will truncate excessively deep or long list structures if desired. Unfortunately, the continued accretion of features in this area has led to an unwieldy specification in the VM (a full third of the VM document), which still leaves users needing to add their own facilities.

Packages supply the ability to do more highly formatted output (e.g. centered, right-justified, with font changes and super/subscripting, in fixed fields, etc.) and to print circular or re-entrant structures preserving the exact topology.

1.5 The terminal

Interlisp provides substantial user control over line editing at the terminal through an object like a read table called a *terminal table.* Again, the features have accumulated over the years, but still fall short of more sophisticated requirements. The basic model of input is a teletype with a single-line buffer in which editing can occur through backspacing. Buffering and echoing can be disabled. Any character can be designated as an interrupt character of one of three types: hard (the interrupt takes effect immediately, but computation may be disrupted), soft (the interrupt may have to wait, but computation is guaranteed unaffected), or flag (a variable is set to T and nothing else happens).

1.6 Functional changeability

In Interlisp-10, the VM is implemented entirely in PDP-10 assembly language. Changing it is tricky -- implementing spaghetti stacks took over a person-year. While it is obviously unfair to compare a second implementation with a first, we think the Bytelisp approach described below is a much more promising one for future Interlisp implementations.

2. The Bytelisp virtual machine

In addition to Interlisp-10, whose underlying implementation machine is the PDP-10 augmented by Tenex or TOPS-20, PARC has another implementation of Interlisp on two personal computers, the Alto and the Dorado. This implementation, which we refer to as Bytelisp, relies on a virtual machine which is substantially less hardware-dependent than the PDP-10 implementation, and therefore of interest in our system comparison.

The Bytelisp virtual machine has an address space of 2↑24 16-bit words, of which only 2↑22 words are presently used by Lisp, divided into 256-word pages. Its instruction set is similar to that described in my paper "A Lisp Machine with Very Compact Programs", presented at IJCAI in 1973. This instruction set is based on 8-bit bytes: most instructions are only one byte long, and there are a total of about 80 different instructions. In addition to the instruction set, which is implemented in microcode, the Bytelisp machine has a support kernel of about 40 SUBRs written in a conventional minicomputer instruction set for such things as floating point and double-precision arithmetic, implementing a disk file system, and SYSIN and SYSOUT. Everything else -- including CONS, the garbage collector, and all I/O above the uninterpreted byte level -- is written in Lisp. The instruction set includes instructions for uninhibited access to the virtual memory, whose use is confined to the level of the system that implements the Interlisp VM. We believe that once we have completed the arduous task of verifying that our implementation of the VM agrees with the current Tenex implementation (which, thanks to poor documentation and lack of management, has diverged in numerous minor ways from the VM document over the past few years), we will be able to run essentially all of the support packages and tools, and all user programs that do not contain PDP-10 assembly code, without change: we attained a state very close to this for a brief period in early 1977, when we were able to run the Boyer-Moore theorem prover, part of the Interlisp version of Reduce, and part of the Dendral system on the Alto.

Bytelisp provides essentially no functional capability beyond that of the Interlisp VM, other than binary I/O and a primitive for operating on bit arrays which we have found useful for display graphics.

2.1 Functional changeability

The level of the Bytelisp system which implements the Interlisp VM consists of about 1.5K (32-bit) microinstructions and 50K (16-bit) non-Lisp instructions. 10K of the non-Lisp instructions are the operating system, which provides disk files, a time-of-day clock, and a few other miscellaneous services. 20K are initialization code which duplicates Lisp code -- we needed it on the Alto for performance reasons and will discard it on the Dorado. The remaining 20K include floating point, double-precision arithmetic, operating system interfaces, SYSOUT, access to the display, and the virtual memory system. A more integrated redesign of the non-Lisp part of the system could probably reduce its size from 30K to 20K. This is the minimum amount of code that would require rewriting if one wanted to move Bytelisp to other hardware.

The commitment of Bytelisp to addressing 16-bit words is a fairly strong one in the Lisp code that implements the VM, although there are primitive functions (and opcodes) for fetching and storing entire 24-bit pointers. Considerable mechanical work would be required to move Bytelisp to a machine like the VAX, which addresses bytes, or to the Jericho, which addresses 32-bit words. No code above the VM level is supposed to know the addressing grain.

While essentially any aspect of the VM is easily changed by modifying the Lisp code that implements it, the consequences for the system and user code above it are difficult to assess. Changes that do not affect functional capability are easier to contemplate: for example, we are considering changing from dynamic to shallow binding, and we might consider changing from a quantum map to typed pointers. Neither of these changes would be enormously difficult to make. More fundamental changes, such as replacing the Interlisp "spaghetti stack" with a heap structure, would have significant effects on the instruction set and would be harder.

3. Support: tools

It is worth pointing out that the philosophy of Interlisp is to provide a programmer interface to any tool available to the user at the terminal, and to provide a maximum degree of parametrization (via global flags and property lists) for every system facility. This has been successful -- perhaps too successful, in that whenever a facility has been found insufficiently general, it has almost always been extended by adding another flag, rather than attempting a redefinition of the facility.

3.1 Editing, filing, translating

Interlisp provides an integrated S-expression editor, for both programs and data. The editor includes a macro facility and the ability to call the interpreter (and be called from programs).

Interlisp has a prettyprinter which is capable of distinguishing different categories of names (e.g. programmer-defined from system functions, local from free variables) by changing font, underlining, etc.

Interlisp includes an integrated filing package which keeps track of what objects (functions, variable initialization, datatype definitions, etc.) have been changed and need to be dumped out on files. In conjunction with Masterscope, this package can do some startling things: for example, when a user has edited a compiler macro, the file package warns him of functions which need to be recompiled because they use the macro.

Interlisp provides infix syntax for many common operations, and a powerful iteration statement including parallel iteration, user-defined sequencing behavior, and ability to combine the generated values in user-defined ways (e.g. search, sum, form a list). This is integrated with the prettyprinter -- a program can be input in either infix or S-expression form, and output in either form independent of how it was input. Unfortunately, this facility is implemented by intercepting interpreter errors and breaking apart undefined atoms looking for infix operator characters. This causes it to behave in ill-defined and sometimes counter-intuitive ways.

3.2 Debugging, session management

When a run-time error occurs, Interlisp provides a variant of the normal executive that allows the user to look around the stack and to resume execution in a variety of different ways. Computation can often continue with little loss of state even if a function with an activation on the stack on the stack was edited.

The user can insert breakpoints at the entry to any function, possibly conditional, or at any place within the definition of an (interpreted) function.

The Interlisp "advising" facility provides a structured way of patching functions at their entries and exits. It is widely used to accomplish minor changes to functions pending incorporation of those changes in the system.

Interlisp saves the inputs and outputs of the last N interactions with the user (N settable), and provides facilities for reviewing, re-executing, grouping, and naming interaction events.

Most system functions are undoable: they save enough information to undo their effects. This includes all operations in the editor. The conventions for undoable functions are simple enough that users can easily write undoable functions of their own. Interlisp does its best to replace all functions in typed-in expressions with their undoable versions, so that some kinds of changes not normally undoable are undoable if they result directly from user input.

Interlisp is the original home of the DWIM facility, which automatically corrects certain kinds of spelling and minor syntactic errors. This idea has been progressively extended -- DWIM now can correct not only function and variable names, but file names, command names for the editor and the Interlisp executive, etc.

As mentioned above, the Interlisp manual is kept on-line, indexed by function names and by a wide variety of topics. The user can ask for information about a function or topic and Interlisp will print the relevant section of the manual. It does surprisingly well in selecting what to print, considering that the manual is essentially free-form text.

3.2 Analysis and monitoring

Masterscope is a unique Interlisp tool that maintains a data base of facts about programs (who calls whom, who binds what, etc.) and can answer quite complex questions about them. It is integrated with the editor and file package, so it knows when things need re-analyzing. It has an option to store the data base on an external file, so that information is available even for functions not currently loaded into the system. My personal experience is that Masterscope is especially valuable when one wants to make changes that affect interface data structures.

Interlisp contains packages for monitoring the consumption of resources (CONSes, computation time, page faults, or any user-defined quantity), either during the computation of a single expression, or continuously for a selected set of functions.

4. Support: packages

The list of packages below is bound to be incomplete, since new packages are written (and find their way into the system) on a continuous basis.

A number of packages exist for maintaining data structures on files. The Boyer-Moore fast search algorithm is implemented as a package, as is a facility for storing hash tables on external files. Packages exist for communication over the ARPANET and for coordinating the assignment of names in a multi-person project.

Interlisp provides a package containing all the usual transcendental functions. An ambitious package for statistical analysis has been implemented at PARC.

Several packages for manipulating displays are being developed in various places.

5. Performance

5.1 Space & speed

Interlisp-10 is notorious among Lisp users for its poor performance. Factors of 2 and 3 in both speed and code space compared to other dialects have been quoted. In fact, it appears that this apparent poor showing is due to a combination of factors, few of which are inherent in Interlisp's functional capabilities. Also, the speed differences vary greatly depending on the exact application.

Many users do not realize that Interlisp provides a "block compiler" which produces tighter calling and binding sequences at the sacrifice of being able to break or advise functions within the "block", or see the names of the internally bound variables. In one instance of unfavorable comparison between Interlisp-10 and UCI Lisp, for example, recompilation with the block compiler actually produced about a 20% advantage for Interlisp.

The present Interlisp-10 compiler and calling conventions have evolved over the years without much systematic scrutiny, and the Interlisp maintenance group at BBN have consistently felt that they did not have the resources for major reimplementation. Thus, for example, a design has existed on paper for about 3 years which would cut the cost of an ordinary function call and return roughly in half. Similarly, the compiler produces atrocious code for arithmetic, and even when block compiling has no facilities for passing unboxed numbers between functions. Likewise, a simple change in the stack format would reduce the cost of CAR and CDR of variables in non-block-compiled code by one instruction. There are many examples of such economies: I would guess that Interlisp-10 could come within 15% of any other dialect in both space and time with an investment of effort that would be extremely modest compared to the effort already invested in the Interlisp system.

Bytelisp uses a very compact instruction set and a CDR-coded list cell representation. The use of a quantum map, rather than type codes in pointers, slows down type testing by one memory reference (which would almost always be in the cache on a machine that had one). The use of deep, rather than shallow, binding slows the system down somewhat -- we have no hard numbers yet, but are working on getting them. (We are also considering switching to shallow binding -- the work required is substantial but not overwhelming.) The absence of instructions for unboxed arithmetic slows down arithmetic operations, since type tests are required, but since all integers from -2↑10 to +2↑16-1 have "immediate" representations, boxing is rarely required.

5.2 Fixed limits

The major limitation in Interlisp-10 is the PDP-10 18-bit address space. Interlisp now uses a software swapping scheme for code which effectively removes this limitation (at a significant cost in time for functions which are allowed to be swapped). However, the limit remains for data structures. Interlisp's requirement that space be allocated permanently to data types a page at a time also wastes about half a page per data type; this is not significant. Within the 18-bit limit, only stack space has a fixed amount allocated to it, which has caused problems in some applications: the other data types all acquire space as needed.

Of Bytelisp's 22-bit address space, a good deal is permanently allocated to various data types: 64K for stack, 1M for code and arrays, 512K each for strings and print names, and enough space for 32K atoms. Another 1M is shared between fixed-size types -- lists, integers, floating point numbers, user-defined data types, etc. -- as needed. The architecture of the system allows all these numbers to be increased, except for the stack and atom space, up to a grand total of 16M words. Relatively minor changes would allow doubling the stack size; more significant changes would remove the limit completely. Increasing the number of atoms to 64K would require minor work; further increases would require very substantial changes.