COGNITIVE AND INSTRUCTIONAL SCIENCES SERIES
CIS-5 (SSL-80-4) Revised

Papers on Interlisp-D

CONTRIBUTIONS BY
Richard R. Burton, Ronald M. Kaplan, Larry M. Masinter, B. A. Sheil, Alan Bell
William van Melle, Daniel G. Bobrow, L. Peter Deutsch and Willie Sue Haugeland

SEPTEMBER 1980; Revised JULY 1981

ABSTRACT

This report consists of five papers on Interlisp-D, a refinement and implementation of the Interlisp virtual machine [Moore, 76] which supports the Interlisp programming system [Teitelman *et al.*, 78] on the Dolphin and Dorado personal computers.

KEY WORDS AND PHRASES

Interlisp, programming environments, personal computing, system construction, measurement, optimization, compilation, input-output, graphics, user interfaces.

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

INTRODUCTION

Interlisp-D is both a revision and an implementation of the Interlisp virtual machine (VM) specification [Moore, 76] for the Dolphin and Dorado personal computers. It qualifies as an implementation of the VM by virtue of supporting both the Interlisp system software [Teitelman *et al.*, 78] and several large, independently developed, application systems, including the Mycin system for infectious disease diagnosis [Shortliffe, 76], the KLONE knowledge representation language [Brachman, 78] and the West tutoring system [Burton & Brown, 78]. It qualifies as a revision of the VM for several reasons. First, the VM was based mainly on an analysis of the structure of the PDP-10 implementation. Any thorough reimplementation was bound to uncover a host of oversights and viable alternatives and Interlisp-D did not disappoint us in this respect. Second, Interlisp-D is the first implementation of Interlisp in a personal computing environment and this raised some issues which did not apply to the time shared system on which the VM was based. Finally, in the interests of transportability, a deliberate attempt has been made throughout Interlisp-D to minimize dependencies on the software environment. One of the ways this was done was by implementing in Lisp many facilities that previous implementations had obtained from their environment. This effectively extended the VM "downwards" far below the level which was previously considered primitive.

Two of the papers that appear in this report were presented at the *1980 Lisp Conference.* They are reprinted here, with slight changes, so as to make them more widely available. The next two were originally prepared as documentation and appear here for the first time. The last, which is to appear in *SIGART Newsletter,* provides a more recent report on the system's status and probable future development. The papers are

Interlisp-D: Overview and status
A report on the implementation, its goals and techniques, and some reflections thereon.

Local optimization in a compiler for stack based LISP machines
A description of the optimizations used during compilation of Lisp into the special purpose Lisp instruction set and their observed effectiveness.

The Interlisp-D I/O system
An outline of the design of the I/O system, coded in Lisp, that provides most of the facilities that a Lisp implementation usually takes from the host operating system.

The Interlisp-D display facilities (Revised)
One of the goals of Interlisp-D is to make Interlisp available as a personal computing environment. Thus, it incorporates an extensive set of graphics facilities. This paper describes the design of the Interlisp-D low level graphics facilities, as they have emerged from the first year of experimentation. In addition to their use in Interlisp-D, we hope they will serve as a basis for other implementations' efforts to explore the use of graphics in Interlisp.

Interlisp-D: Further steps in the flight from time-sharing
A status report and description of ongoing and planned extensions, as of June 1981.

As documented in the first paper, the implementation of Interlisp-D was a major effort, which has taken a long period of time and included the efforts of a large number of people. The integration of Interlisp's programming support tools into a personal computing environment is a task of similar magnitude. Our hope is that these papers may make the path somewhat easier for future implementors.

REFERENCES

Brachman, R. *et al.*
KLONE Reference Manual. BBN Report No. 3848, 1978.

Burton, R. and Brown, J.
An investigation of computer coaching for informal learning activities. *International Journal of Man-Machine Studies*, 11, 1979, 5-24.

Moore, J.
The Interlisp virtual machine specification. Xerox PARC, CSL-76-5, 1976.

Shortliffe, E.
Computer-based medical consultations. American Elsevier, 1976.

Teitelman, W. *et al.*
The Interlisp reference manual. Xerox PARC, 1978.

Interlisp-D: Overview and Status

Richard R. Burton, Larry M. Masinter, Alan Bell, Daniel G. Bobrow,
Willie Sue Haugeland, Ronald M. Kaplan and B. A. Sheil

Abstract

Interlisp-D is an implementation of the Interlisp programming system on the Dolphin and Dorado, two large personal computers. It evolved from AltoLisp, an implementation on a less powerful machine. This paper describes the current status of Interlisp-D and discusses some of the issues that arose during its implementation. The techniques that helped us improve the performance included transferring much of the kernel software into Lisp, intensive use of performance measurement tools to determine the areas of worst performance, and use of the Interlisp programming environment to allow rapid and widespread improvements to the system code. The paper lists some areas in which performance was critical and offers some observations on how our experience might be useful to other implementations of Interlisp.

BACKGROUND

Interlisp is a dialect of Lisp whose most striking feature is a very extensive set of user facilities including syntax extension, error correction, and type declarations [Teitelman *et al.*, 78]. It has been in wide use on a variety of time shared machines over the past ten years.

AltoLisp

In 1974, an implementation of Interlisp for the Alto, a small personal computer, was begun at Xerox PARC by Peter Deutsch and Willie Sue Haugeland [Deutsch, 1973]. This AltoLisp implementation introduced the idea of providing a microcoded target language for Lisp compilations which modelled the basic operations of Lisp more closely than a general purpose instruction set. A similar instruction set was also implemented for Maxc, a microprogrammed machine running the TENEX operating system [Fiala, 1978].

The design of AltoLisp is presented in [Deutsch, 1978]. Its characteristics include a very large address space (24 bits); deep binding; CDR encoding [Bobrow & Clark, 1979]; transaction garbage collection [Deutsch & Bobrow, 1976]; and an extensive kernel implemented in a mix of microcode and Bcpl. Although AltoLisp was completed and several large Interlisp programs were run on it, its performance was never satisfactory, due principally to the limited amount of main memory and the lack of support in the processor architecture for either virtual memory management or byte code decoding. Interlisp-D is the result of transferring AltoLisp to an environment with neither of these limitations.

Interlisp-D

The Dorado [Lampson & Pier, 1980] is a large, fast, microcodable personal machine with 16-bit data paths. It has a large main memory (~1 megabyte) and hardware support for both

---

A revised version of a paper originally presented at the 1980 Lisp Conference, Stanford, Ca.

instruction decoding and virtual memory management. The Dolphin is a similar, but smaller and less powerful, machine.

Both machines have microcode to emulate the Alto, so the initial transfer of the running AltoLisp system to them was straightforward. Although the microcode to interpret the Lisp instruction set needed to be rewritten, the Bcpl runtime support system was transported with only minor changes. However, initial performance was far worse than would be expected from a simple consideration of machine features. We expected Dorado Interlisp-D to dominate Interlisp-10 running on a single user DEC KA-10, but in fact, some computations took 10 to 100 times longer. Our primary goal, then, became to improve the performance of the existing system. First, careful measurements were taken of the system doing a variety of tasks. Functions which took inordinate amounts of time were examined in detail. Additional microcode was written, and major portions of the Lisp code were redone.

The most surprising thing to us was that we obtained considerable performance improvements by moving large parts of the system from Bcpl into Lisp. This allowed us to use a number of programming tools in the Interlisp system, and allowed us to put more structure into the layers of the system's kernel. Interlisp-D is now supporting a user community. While speed ratios vary widely across different classes of computation, it appears that Dorado Interlisp-D runs more than five times faster than Interlisp-10 on a single-user DEC KA-10. *[Note: This figure is from August 1980;* cf. *remarks in* "Further steps...".*]*

THE "LISPIFICATION" OF INTERLISP-D

Much of the Interlisp system is written in Lisp itself, resting on a kernel not defined in Lisp. The Interlisp virtual machine specification [Moore, 1976] attempted to identify a set of kernel facilities which would support the full Interlisp system. This was done by carefully documenting those parts of the PDP-10 Interlisp system that were written in assembly language or imported from the operating system. This specification is quite large. AltoLisp reduced this kernel by implementing some of the VM facilities in Lisp; Interlisp-D accelerated this development. In addition to improving the transportability of the implementation, the move also improved performance, gave the implementors access to more a more powerful implementation language and programming tools, and limited the breadth of expertise required of system implementors.

Efficiency

Programs written in a higher level language are often less efficient than equivalent assembly language programs, because they cannot exploit known invariances and optimizations which would violate the strict semantics of the target language. Moving code from Lisp into the kernel has been a traditional way of improving the performance of Lisp systems. Substantial sections of the PDP-10 implementation of Interlisp, for example, are in machine code for this reason. When a large proportion of AltoLisp was moved from Bcpl into Lisp in order to improve memory utilization and aid modification, the speed of the system decreased by nearly a factor of three [Deutsch, 1978]. Thus, to improve Interlisp-D performance, we first looked for Lisp-coded sections of the system that could be incorporated into the Bcpl kernel. However, we soon discovered that the poor performance was due more to the design of the

algorithms in the kernel than to the language in which they were implemented. Since we did not wish to carry out a large-scale redesign in the limited Bcpl programming environment, we decided to go in the other direction: we would move code *out* of the extended Bcpl kernel and into Lisp so that we would be better able to change the algorithms. Specific targets for replacement were large sections of the Bcpl kernel with known performance problems whose functionality could easily be expressed in Lisp; one of the major areas was the I/O system.

Language power and tools

A primary reason for implementing the bulk of a programming system in itself is that one obtains the advantage of programming in a (presumably) more expressive and powerful language. In addition, we felt that the major modifications and tuning that would be necessary to provide adequate performance would be far more tractable in Interlisp. In Interlisp we had both a first rate programming environment and instrumentation tools, and we had no other system implementation language which had either. Our subsequent experience has sustained this view.

Linguistic uniformity

An important sociological benefit of having a programming system described in the language it implements is that the system's implementors and users share the same culture. Users can inspect the system code, comment on it, adapt it for their own purposes, and sometimes even change it. This involves the users of the system in its design and maintenance in a way that would not be possible if system construction took place in a different language culture. Specifically, the availability of the system source code allows the system to grow and adapt much more rapidly than environments in which a formal documentation phase is a prerequisite to the development and distribution of new facilities. In turn, the users can explore the behavior of the system "all the way to the edges", as there are no sharp language barriers. The value of this linguistic uniformity has been confirmed by its successful use in other language cultures, such as Smalltalk [Goldberg, 1980].

An example: the IO system

A high level language I/O system consists of both low level device handlers and device independent sequential and random access. In most Interlisp implementations, the entire I/O system, up to and including the functions defined in the virtual machine, is provided by the host operating system. In Interlisp-D, all of the logical I/O system and a substantial proportion of the device dependent code is written in Lisp. The logical I/O system implements the Interlisp user program I/O facilities and the underlying operations in terms of which these are implemented. These include sequential and random access operations (i.e., read and write a byte, query end of file, reposition file pointer, etc.), buffer management (both for system only and directly user accessible buffers) and a device independent treatment of file properties. The logical level is in turn implemented in terms of the notion of an I/O *device*. This is an object which provides a standard set of low level, device dependent functions, such as those to read and write a page, create and delete files, etc. Using this interface, the addition of a new device is simply a matter of writing a new set of these functions. The Interlisp-D I/O system design is described in [Kaplan *et al.*, 1980].

IMPLEMENTATION TECHNIQUES

Measurements

In tuning the performance of a program, it is crucial to be able to determine exactly where time is being spent. With a large body of code and limited manpower, it is not possible to "optimize everything." Our performance measurement system has proved invaluable in tracking down specific (and unforeseen) problems.

The measurement system was originally developed for AltoLisp by Deutsch and Haugeland. It operates in two stages. First, the computation of interest is run with *event logging* enabled. This produces a (very large) file of *log events*, which is later analyzed. The log events are put out by both the microcode and the run time support system and include time-stamped events for function call and return, entry and exit from the Bcpl routines, I/O activity, and other events of interest. Alternatively, the microcode can also collect counts of opcode frequencies and a frequency sample of the microcode PC.

Statistics gathering can be enabled at any time that Lisp is running. One can decide spontaneously to take measurements whenever performance unexpectedly degrades. Comparison of these measurements with those taken during a similar run that exhibited normal performance can be used to identify the source of intermittent performance problems. This technique was used, for example, to track down an intermittent slowdown in the code that handled stack frame overflow.

The analysis phase reads the log file and computes summary statistics from it. From call and return events, the time spent in individual functions can be computed, either including or excluding the time spent in the functions called by them. The accumulated times (including the times spent by called functions) locate the higher level functions which are the root of a large amount of time and which may be a candidates for redesign. The individual time (excluding called functions' times) are useful for isolating what improvement can be expected from optimizing or microcoding the body of that function.

Function performance data is presented in tables which show the number of times each function was called and the time spent in each function. For example:

function	#ofCalls	Time	%ofTime	PerCall
NTHCHC	1977	236702	10.6	119
\HT.FIND	1729	168492	7.6	97
LITLEN	2111	131708	5.9	61
LITBASE	2141	118902	5.3	56
...				

Tables such as this isolate very accurately those functions which are worth rewriting as well as identifying those which are not. In this example, NTHCHC, which calls both LITLEN and LITBASE, is an obvious candidate. In another run we discovered that 15 percent of the time was being spent adding one to a counter which had overflowed the small number range. This prompted a redesign of the large number arithmetic.

Additional controls on the analysis routines allow more specific questions to be answered. The analysis can be restricted to that part of the computation within any particular function. For example, only that part of the computation that takes place within READ can be analysed. The analysis can also be limited to a set of functions, in which case only these functions will appear in the table of results. Any time spent in a function not in the set will be charged to the closest bounding function that is.

The analysis routines extract from the log file useful information besides performance data. For example, the dynamic calling behavior is captured in the log, so one frequently useful technique is to list which functions have called (and been called by) other functions, and even how many arguments they were passed. The flexibility of the analysis routines combined with the wealth of information collected during the logging stage allows a given computation to be examined from many points of view.

Initialization

There are several areas that cause fundamental problems for the implementation of a language system in itself: memory management (which requires that the memory manager itself will not cause memory faults), stack overflow recovery (where the stack manager must itself have some stack), and initialization. Initialization is difficult because the initialization program must operate when the system is not in a well formed state. The problem in initialization can be characterized by the question: "If the compiled code reader is itself compiled code, who will read *it* in?"

Several methods of doing initialization suggest themselves. For example, the image can be initialized by a program written in some other language. This is the solution adopted in AltoLisp. Alternatively, if the interpreter is written in some other language, the compiled code reader can be run interpretively to read itself in. However, both of these solutions require a substantial amount of non-Lisp code either for storage allocation or for interpretation.

We adopted still another solution. The compiled code reader was modified to load code into an environment other than that in which it is running. The primitive functions that the loader uses to manipulate the environment (e.g., fetch and store into specified virtual memory locations) are replaced by functions that manipulate another memory image stored as a file. To begin with, an empty memory image file is created and then the "indirect" version of the compiled code reader is used to load the compiled files that constitute the lowest level of the system into this empty image. We thus avoid the potential problem of maintaining two different programs with knowledge of system data structures.

An appropriate programming environment

One of the advantages of writing most of the kernel in Lisp is that Interlisp provides a very powerful programming environment. Its attributes that we found particularly useful were:

Language features: The advantages of "data-less" or data-structure-independent programming have long been known: more readable code, fewer bugs, the ability to change

data structures without having to make major source program modifications. The Interlisp record package and data type facility encourages this good practice by providing a uniform and efficient way of creating, accessing and storing data symbolically, i.e., fields of data structures are referred to by name. Because the Interlisp-D implementation allows a large number of data types, we have felt free to give system data structures (such as file-handles, page buffers, read tables) their own data types. In addition, records could be overlaid on structures not under Lisp's control (e.g., the leader page of a disk file or the format of a network packet) to provide the same uniform access.

Cross compilation: We maintained an Interlisp-10 environment in which we could edit, compile and examine functions for the Dorado. The function and record definitions for the Dorado implementation were kept on property lists instead of defintion cells. This allowed us to work on functions such as READ and CONS without destroying the environment in which we were working.

Masterscope: Many of our improvements to AltoLisp involved massive changes throughout the many system source files. Interlisp's Masterscope program was an essential aid in determining what would be affected by a proposed improvement and in actually performing the necessary edits. Masterscope is an interactive program for analyzing and cross-referencing Lisp functions. It constructs a database of which functions call which other functions, where variables are bound, used, or set, and where record declarations are referenced. Masterscope utilizes the information in the database to interpret a variety of English-like commands. Our cross-compilation environment incrementally updated a database that was shared among all programmers on the project, so that with very little overhead the information in the database was kept consistent with the current state of the evolving system.

Masterscope was most helpful in planning and carrying out modifications to major system interfaces, which usually meant changing the numbers and kinds of arguments to various functions. We would first ask Masterscope to simply list the callers of those functions to give some estimate of the impact of the proposed change, much as one might use a static cross-reference program. We would then invoke the SHOW command, instructing Masterscope to locate in the source-file definitions of all the callers the expressions that actually called the interface functions. These expressions were gathered together and displayed as a group, so that we could verify our intuitions about what assumptions clients were making about the interface. In many cases, the rapid source-code exploration that Masterscope made possible revealed flaws in our redesign which otherwise would not have become apparent until much more effort had been expended. Having decided that our modification was acceptable, we used Masterscope's EDIT command to actually drive the editing. This caused Masterscope to load the definitions of all the client functions, call the Interlisp editor on each one, and position the editor at each of the expressions that needed to be changed. Masterscope, not the programmer, kept track of which functions had been changed and which still needed to be edited. When Masterscope finished the editing sequence, the programmer was sure that the changes had been made completely and consistently.

Our redesign of the I/O system [Kaplan *et al.*, 1980] is a good illustration of the power of this interactive tool. We completely replaced the lowest-level I/O interface, which involved

changes to approximately 40 functions on 15 source files. The major part of the revision was accomplished in response to a single EDIT WHERE ANY CALLS '(BIN BOUT ...) command, without ever looking at hard-copy source listings.

Rapid access to system sources: Our cross-compilation environment maintained a shared data base which allows the definition of any Lisp function to be retrieved for viewing or editing in a few seconds. The microcode and Bcpl can be "browsed" using the same interface. Rapid online access to system sources lessened the need to work from listings.

Levelling

One of the original motivations for having a large part of AltoLisp in Bcpl was the belief that it was important not to provide Lisp primitives that gave unrestricted access to the implementation data structures. This reasoning fails to discriminate between the system implementation and user program levels. Allowing system programs arbitrary access to memory locations does not at all imply that user level code has this access.

Failing to make the system/user distinction hurt AltoLisp in three ways. First, it provided one motivation for the large Bcpl kernel. Second, most of that part of the system which *was* written in Lisp was prohibited from manipulating underlying data structures except through overly general functional interfaces. Last, it discouraged the use of higher level structuring facilities (such as the record package) so that code that required any knowledge of system data structures tended to be written entirely in terms of low level primitives.

Using Lisp as a system implementation language requires very careful consideration of the layering of the system into levels of access and knowledge. Further, the precision that is needed cannot be obtained by simple binary discriminations but must be carefully considered for each piece of code. This presents a considerable challenge to the implementors' self restraint, as Lisp provides few facilities to enforce such a layering. Appropriate use of abstraction is essential if layering is to be preserved under the constant revision necessitated by intensive performance debugging.

Diagnostics

Development of the Lisp microcode was aided by a reasonably complete set of microcode diagnostics written in Lisp. Diagnostics are difficult because they are most useful when very little can be assumed *a priori* to work. It is also difficult to achieve complete coverage of all cases. In addition, extensive knowledge of the Lisp system was required to develop diagnostics. For example, every opcode needs to be tested when encountering page faults or stack overflows. Setting up a situation which will page fault or overflow the stack in the next opcode requires a very intimate knowledge of the implementation. Having undertaken several microcode revisions, development of a comprehensive set of diagnostics seems well worth the effort.

Important performance issues

While not strictly a technique, we feel that it is important to mention the major areas in which

performance has proved to be crucial. While some of these are undoubtedly specific to Interlisp-D, we feel that they deserve consideration by those who might be building similar Lisp systems.

The earlier intuition that the hardware assist for decoding byte opcodes was important was substantiated. Performance improved by nearly a factor of two when this was installed. Implementing the decoding and dispatch in microcode is conceding a large performance loss.

There are several parts of the system for which it seems important to have microcode support. When written in Lisp, the garbage collector seems to consume between 10-30% of the processor, although the figure varies widely over different computations. Further, in a system that uses deep binding, some form of microcode assist for free variable lookup is very desirable. A speedup factor of between two and four accompanied the introduction of microcode support for this in Interlisp-D. Statistics show that less than one percent of the execution time is now spent in free variable lookup.

Their heavy use in implementing system code almost mandates that the arithmetic functions have complete microcode support. Further, we found it to be critical to have a large range of small numbers (numbers without boxes), so that the performance critical, low level system code did not invoke Lisp's storage management.

WHY IS AN INTERLISP IMPLEMENTATION SO HARD?

The Dolphin/Dorado implementation of Interlisp took many times the expected effort to complete. Given the widespread intuition to the contrary, it is perhaps worthwhile to reflect on why it has proved so difficult. The answer is painfully simple: Interlisp is a *very* large software system and large software systems are not easy to construct. Interlisp-D has on the order of 17,000 lines of Lisp code, 6,000 lines of Bcpl, and 4,000 lines of microcode. In many ways, the more interesting question is why does it look so straightforward?

Without a doubt, the perceived ease of implementing Interlisp springs from the existence of the virtual machine (VM) specification. This admirable document purports to give a complete description of the facilities that are assumed by the higher level Interlisp software, and does a remarkable job of laying out the foundations of this very large software confederation. It is difficult to resist the implication that a straightforward implementation of this mere 120 pages of specification, much of which is already described in programmatic form, will constitute a new implementation of Interlisp. The issue is rather more complicated than that.

The VM specification looks small, but it is not. There is no simple correspondence between the size of a specification and the volume of code required to implement it. Many of the major problems of an Interlisp implementation (e.g., performance, the garbage collector, the compiler) are simply not addressed at all. We caution Interlisp implementers that the slimness of that document is misleading.

Further, while the virtual machine specification is an excellent first pass, it is far from complete. Many "incidental" functions and variables were left out (e.g. HOSTNAME). It is

occasionally ambiguous in places where the system code relies on a specific interpretation. Even though once complete, changes in the higher level code required that the VM be extended to support new facilities. Finding all these variations is an exhausting task. It is substantially easier to get 95% compatibility than 99.9%, and amazing how many programs are sensitive to the difference.

One way to look at the Lisp kernel that was written for Interlisp-D is as the definition of a new VM specification *in Lisp* code. While much of the code is specific to the Dorado environment, a great deal of it simply extends the virtual machine downwards by providing a much lower level treatment of functions such as PRINT and READ. We hope our work will provide other new implementations with a firmer foundation than the VM document alone.

Another problem for any very large software system is the existence of a long development tail. A version of Interlisp-D was "sort of running" years ago. Several other implementations of Interlisp have "sort of run" but have never reached production status. One of the key problems here is performance. The success of the PDP-10 implementation of Interlisp is due to a lot of hand tuning. Any straightforward, clean implementation will prove to be slow, and finding performance problems is difficult, even with good measurement tools. A large number of design decisions have to be made and a large amount of code has to be written. While not all of the decisions have to be optimal, none of them can be pessimal. While the Interlisp-D experience can provide some guidance, many of these decisions will be environment specific.

Finally, an important issue has been compatability with the PDP-10 implementation of Interlisp. In some ways our determination to remain compatible has helped. Ambiguities and omissions from the VM specification could always be resolved by copying the PDP-10 implementation. However, this compatibility requirement was also a burden. Complete compatibility with another implementation is hard. This is particularly so when the new implementation is in a quite different environment (a personal rather than a time-shared machine). The tension between remaining compatible versus exploring the possibilities of a personal machine environment is a continuing issue, which will probably be a focus of our further efforts on the Interlisp-D system.

Acknowledgements

Peter Deutsch was a principal designer and motivating force behind AltoLisp, of which Interlisp-D is a successor. Warren Teitelman has made major contributions to the Interlisp-D project. Martin Kay, Henry Thompson, Richard Fikes and Austin Henderson have also contributed time and effort on various aspects of the project.

REFERENCES

Bobrow, D.G. & Clark, D.W.
Compact encodings of list structure. *ACM Transactions on programming languages and systems* **1**, 1979.

Deutsch, L.P.

A Lisp machine with very compact programs. *Proceedings of the third international joint conference on artificial intelligence,* Stanford 1973.

Experience with a microprogrammed Interlisp system. *IEEE Micro-11 conference,* 1978.

Deutsch, L.P. & Bobrow, D.G.

An efficient incremental, automatic garbage collector. *CACM* 19:9, 1976.

Fiala, E.R.

The Maxc systems. *IEEE Computer* 11, May 1978.

Goldberg, A.

Smalltalk: Dreams and schemes. Xerox PARC, to appear.

Kaplan, R.M., Sheil, B.A., & Burton, R.R.

The Interlisp-D I/O system. Xerox PARC, SSL-80-4, 1980.

Lampson, B.W. & Pier, K.A.

A processor for a high-peformance personal computer. *Seventh international symposium on computer architecture*, La Baule, France, May 1980.

Masinter, L.M. & Deutsch, L.P.

Local optimization in a compiler for stack-based Lisp machines. *Proceedings of the 1980 Lisp conference*, Stanford, 1980 and Xerox PARC, SSL-80-4, 1980.

Moore, J.S.

The Interlisp *virtual machine specification.* Xerox PARC, CSL-76-5, 1976.

Teitelman, W. *et al.*

Interlisp *Reference Manual*, Xerox PARC, 1978.

Local Optimization in a Compiler for Stack-based Lisp Machines

Larry M. Masinter and L. Peter Deutsch

Abstract

We describe the local optimization phase of a compiler for translating the Interlisp dialect of Lisp into stack-architecture (0-address) instruction sets. We discuss the general organization of the compiler, and then describe the set of optimization techniques found most useful, based on empirical results gathered by compiling a large set of programs. The compiler and optimization phase are machine independent, in that they generate a stream of instructions for an abstract stack machine, which an assembler subsequently turns into the actual machine instructions. The compiler has been in successful use for several years, producing code for two different instruction sets.

INTRODUCTION

This paper describes the local optimization phase of a compiler for translating the Interlisp [Teitelman *et al.*, 1978] dialect of Lisp into stack-architecture (0-address) instruction sets [Deutsch, 1973]. We discuss the general organization of the compiler, and then the set of optimization techniques we have found most useful. The compiler and optimization phase are machine independent, in that they generate a stream of instructions for an abstract stack machine, which an assembler subsequently turns into the actual machine instructions. The compiler has been in successful use for several years, producing code both for an 8-bit Lisp instruction set for several personal computers, [Deutsch, 1978, 1980, Burton *et al.*, 1980], and a 9-bit instruction set for Maxc, a time-shared machine running the Tenex operating sytem [Fiala, 1978].

There are always tradeoffs in designing a compiler. Each additional optimization usually increases the running time of the compiler as well as its complexity. The improvement in the code generated must be weighed against the benefit gained, measured by the amount of code improvement weighted by the frequency with which the optimization is applicable. Rather than provide a multiplicity of compiler controls, which most users would not want to know about, the compiler designer should use empirical knowledge of "average" user programs and make appropriate design choices. One of the major purposes of this paper is to publish some empirical results on the relative utility of different code transformations, which can aid designers in making such choices.

Why this compiler is different

Compiling Lisp for a 0-address architecture differs from compiling other languages such as PASCAL or ALGOL for several reasons. Procedures are independently compiled, so that global

---

A revised version of a paper originally presented at the 1980 Lisp Conference, Stanford, Ca.

optimization techniques are not relevant. Compiling for a stack-based instruction set is different from compiling for more conventional machine architectures, in that register allocation is not relevant, and randomly addressable compiler-generated temporary variables other than top-of-stack are difficult to access.

In systems which provide interactive, symbolic debugging of compiled code, a compiler must not manipulate source programs too freely, since even common optimizations like tail recursion removal make it difficult or impossible to explain the dynamic state of the program in terms of the original source. However, Lisp also provides an interpreter which can be used for debugging purposes when strict faithfulness is needed; interpreted and compiled code can be mixed freely. Thus, we take the view that the compiler can rearrange the implementation of an individual function in any manner consistent with the semantics of the original program, even if fine-grained debugging information may be lost or altered (e.g., if variables that appeared in the source get eliminated).

What we did not handle

The compiler concentrates on local optimizations. More global transformations such as pulling invariants out of loops or duplicate expression elimination would probably pay off often enough to be worth the additional complication in an environment where speed was of great concern and the individual functions were large.

Related work

A few of our compiler's transformations, such as cross jumping and tail recursion removal, have been part of the literature for some time. We know of three other Lisp compilers that both compile into a machine-independent intermediate language and do substantial optimization.

The Standard Lisp project at the University of Utah has produced a transportable compiler similar to ours [Griss & Hearn, 1979]. Their intermediate language is register- rather than stack-oriented. Their report mentions a number of the optimizations in our list, plus others only applicable to register machines, but their list is shorter and not accompanied by empirical data.

Another similar compiler was the subject of a Ph.D. dissertation [Urmi, 1978]. The author in this case was more concerned with the design of instruction sets than with optimizing the use of a given architecture. His report contains extensive statistics on the opcode frequencies, and interesting suggestions for instruction set design, including a consideration of both stack- and direct-address architectures; however, his optimizations are all in the "peephole" category, being limited to a few adjacent instructions, except for the usual optimization of ANDs and ORs.

The RABBIT compiler [Steele, 1978] translates an unusual lexically scoped Lisp dialect into code for a register machine. Its optimization techniques are extremely sophisticated with regard to removal of recursions and variable bindings. However, the differences in coding

style resulting from lexical scoping are so large that a comparison between RABBIT's goals and those of our compiler would not be meaningful.

Results

Optimization in the byte compiler provides an average 5-10% speed improvement and a 10-15% space improvement over completely unoptimized code. While significant, this does not make it one of the more significant factors affecting the performance of our Lisp systems [Burton *et al.*, 1980]. The most significant effect that a reasonable optimizing compiler has for its users is a certain amount of unconcern for vagaries of syntax. Programmers can write their routines for clarity, without concern for purely syntactic devices which might otherwise affect performance. For example, while inserting assignments inside expressions is allowed and occasionally perspicuous, it generally is more readable to perform variable assignments in separate statements, and to subsequently use the variables in an unnested manner. Knowing that the compiler will do an adequate job of optimization means that a program author can make choices based on legibility, even in the most time-critical routines.

ABOUT THE COMPILER AND THE OBJECT LANGUAGE

The compiler operates in several passes. The first pass takes the S-expression definition of the function being compiled, and walks down it recursively, generating a simple intermediate code, called ByteLap, analogous to assembly code. During this first pass, the compiler expands all macros, CLISP, record accesses and iterative statements. A few optimizations are performed during this pass, but most of the optimization work is saved for later. The next pass of the compiler is a "post-optimization" phase, which performs transformations on the ByteLap to improve it. Transformations are tried repeatedly, until no further improvement is possible.

After the post-optimization phase is done, the results are passed to an assembler, which transforms the ByteLap into the actual machine instructions. We currently have two different assemblers in use, which generate code for two different instruction sets: one for the Maxc 9-bit instruction set and one for the personal machine 8-bit instruction set. The Maxc and personal machine implementations of Interlisp differ considerably; for example, the Maxc system employs shallow variable binding, while the personal machine systems employ deep binding. The translation from ByteLap to machine code is straightforward.

The structure of ByteLap

The ByteLap intermediate code generated by the compiler can be viewed as the instruction set for an abstract stack machine. The format of ByteLap is described here to simplify subsequent discussion of optimizations. There are 15 opcodes, each of which has some effect on the state of the linear temporary value stack. The instruction set is:

(VAR **var**)	Push the value of the variable **var** on the stack.
(SETQ **var**)	Store the top of the stack into the variable **var**.

(POP)	Pop the stack (i.e., throw away the top value and decrement stack depth by one).
(COPY)	Duplicate (push again) the top of the stack.
(CONST val)	Push the constant **val** on the stack (**val** may be of any Lisp data type, e.g., an atom or a number.)
(JUMP tag)	Jump to the location **tag**.
(FJUMP tag)	Jump to the indicated location if top-of-stack is NIL, otherwise continue. In either case, pop the stack.
(TJUMP tag)	Similar to FJUMP, but jump if top-of-stack is non-NIL.
(NTJUMP tag)	Similar to TJUMP, but do not pop if it jumps. This is useful when a value is tested and then subsequently used.
(NFJUMP tag)	Analogous to NTJUMP.
(FN n fn)	Call the function **fn** with **n** arguments.
(BIND (v_1...v_n)(n_1...n_k))	Bind the variables v_1,...,v_n to the n values on the top of the stack. Also bind the variables n_1,...,n_k to NIL. All bindings are done in parallel. Remember the current stack location.
(UNBIND)	Save the current top of stack. Throw away any other values on the stack since the last (stacked) BIND, and undo the bindings of that BIND. Re-push on the stack the saved value. This is used at the end of PROG or LAMBDA expressions whose value is used.
(DUNBIND)	Similar to UNBIND, but do not restore the value.
(RETURN)	Return top-of-stack as the value of the current function, throwing away any other values on the stack.

Note that a given ByteLap opcode could have one of several different translations in the actual code executed. For example, both the personal machine and Maxc implementations have a separate opcode for pushing NIL, in addition to a more general constant opcode. The final code generation phase transforms the (CONST NIL) ByteLap instruction into the appropriate opcode. Operations such as arithmetic or CAR are encoded as FN calls, even though the instruction sets have specialized instructions to perform those operations. The assemblers distinguish between the built-in operations and those that must actually perform external calls; the compiler and the optimization phase do not care. Furthermore, a sequence of ByteLap instructions can assemble into a single machine instruction; for example, both instruction sets have instructions which can do a SETQ and a POP in the same instruction. These are easily detected with a short look-ahead during code generation.

COMPILER OPTIMIZATIONS

One of the most important ground rules for the optimization phase has been that all

optimizations are conservative: they must not increase either code size or running time. Only optimizations which experience has shown to be useful are described here.

The statistics given in the text below were obtained as a result of compiling a total of about 2200 functions, producing 65000 bytes of object code. Numbers in <angle brackets> in the text indicate the number of times that a given optimizing transformation or technique was applicable.

Optimizations during code generation

A few optimizations are performed during the initial code generation phase. In particular, the compiler keeps track of the execution context of any given expression (similar to many other Lisp compilers we know of). Thus, in the recursive descent of the S-expression definition, the flag **effect** is set if the current expression is being compiled for effect only, and the flag **return** if the value is being returned as the value of the entire function.

Remove no-effect constructs when compiling for effect <162>

Compiling a variable or constant for effect results in no code generated. A call to a function with no side effects merely causes its arguments to be evaluated for effect: for example, a macro might expand into (CAR (RPLACA X Y)), which if executed for effect only performs the RPLACA, but if the value is used will return the value stored.

Remove extraneous POP <2035>

Knowledge of **return** context is used to omit extraneous POP instructions, since unused values can be left on the stack to be swept away when the frame is released by a (RETURN). For example, in the function

```
(LAMBDA (X) (PRINT X) (TERPRI))
```

the first pass emits

```
(VAR X) (FN 1 PRINT) (FN 0 TERPRI) (RETURN)
```

rather than

```
(VAR X) (FN 1 PRINT) (POP) (FN 0 TERPRI) (RETURN).
```

The compiler also uses **return** context to eliminate extraneous JUMPs after arms of a conditional to the end of the conditional code (each arm of the conditional is compiled in **return** context, which will cause it to be terminated by a (RETURN) opcode).

The compiler also removes tail recursion in **return** context <36>. In addition, constant folding is done in the first pass for functions which are constant on constant arguments (e.g. EQ and arithmetic opcodes) <34>. Constant folding is done after the code for each argument is generated, so that constant detection can be achieved by looking for CONST opcodes, rather than pre-expansion of macros.

Post-optimizations

The second pass of the compiler consists of several local transformations on the generated ByteLap code which are tried repeatedly in turn until no further improvement can be made <6461 passes total, including the final unsuccessful pass on each function>. While the compiler contains many transformations, empirical results of compiling a large number of files show that the following transformations are the most useful—we have excluded transformations which were rarely effective. For each transformation we give its name, a symbolic version of it, a brief discussion, and an example in which the optimization would be effective.

COPY introduction <1018>

val val ⇒ **val** (COPY)

This transformation reduces neither code size nor execution time; however, it often enables other optimizations. The **val** opcodes can be two identical CONST or VAR opcodes, or a SETQ followed by a VAR with the same variable. For example, the expression

```
(FOO (SETQ X (FUM)) X)
```

compiles to

```
(FN 0 FUM) (SETQ X) (VAR X) (FN 2 FOO)
```

which gets transformed to

```
(FN 0 FUM) (SETQ X) (COPY) (FN 2 FOO).
```

Variable duplication <1137>

(SETQ **var**) (POP) (VAR **var**) ⇒ (SETQ **var**)

This transformation occurs frequently after assignments. For example, the expressions

```
(SETQ X Y) (COND (X (FN)))
```

compiles to

```
(VAR Y) (SETQ X) (POP) (VAR X) (TJUMP L1) (FN 0 FN) L1:
```

which transforms into

```
(VAR Y) (SETQ X) (TJUMP L1) (FN 0 FN) L1:
```

Dead assignment <661>

(SETQ **var**) *{no subsequent use of* **var***}* ⇒

The compiler scans ahead a short distance for either a (RETURN) or subsequent (SETQ **var**) with no intervening instruction which either uses (VAR **var**) or else calls a function which might see the binding of **var**. For example, after the examples in both *COPY introduction* and *Variable duplication*, the assignment to X might well be "dead", and the (SETQ X) removed.

Unused push <734>

```
val (POP) ⇒
```

Although the first pass avoids generating values followed by POP by the **effect** mechanism, enough instances arise where subsequent optimizations uncover unused values to make this transformation worthwhile during the post-optimization phase. **val** can be a CONST, VAR, or COPY. In addition, if **val** is a (FN n fn), where **fn** is a side-effect free function, it is replaced by **n** (POP)s.

Merge POP with DUNBIND <105>

```
(POP) (DUNBIND) ⇒ (DUNBIND)
```

This simple transformation takes advantage of the fact that the DUNBIND opcode implicitly pops any values left on the stack since the last **BIND**.

JUMP OPTIMIZATIONS

Vacuous jump <1033>

```
(JUMP tag) tag: ⇒
(cJUMP tag) tag: ⇒ (POP)
```

While the first pass ByteLap generation explicitly deletes these <265 occurrences>, this transformation is useful to clean up after others. In the pattern, cJUMP is either TJUMP or FJUMP.

Invert sense of jump <488>

```
(FJUMP tag1) (JUMP tag2) tag1: ⇒ (TJUMP tag2)
```

This transformation can occur, for example, when there are explicit GO's in the source. For example, the expression

```
(COND (X (GO LABEL1)))
```

compiles to

```
(VAR X) (FJUMP L1) (JUMP LABEL1) L1:
```

which transforms into

```
(VAR X) (TJUMP LABEL1) L1:
```

COPY introduction for TJUMP <241>

```
val (NTJUMP tag) val ⇒ val (COPY) (TJUMP tag)
```

This transformation notes that, whether or not the JUMP is taken, the value **val** will remain on the stack. The transformation is effective for both NTJUMP and NFJUMP. Note that **val** will be NIL in one of the cases.

JUMP code in-line <457>

(JUMP tag) ... tag: *{code}* ⇒ *{code}* ...

This transformation moves the entire segment *{code}* in line only in the situation where the JUMP is the only way of reaching **tag**.

Jump-through <2259>

(jump tag) ... tag: (JUMP tag2) ⇒ (jump tag2) ...

One of the most common transformations in the compiler occurs when the target of a jump is itself a jump instruction. For example, the code generated for

```
(COND (A B) (T C))
```

is:

```
(VAR A) (FJUMP L1) (VAR B) (JUMP L2) L1: (VAR C) L2:
```

If the variable B is replaced by a COND clause, the target of the jump at the end of that CONDs second clause would itself be a jump instruction. The jump in the pattern above can be any of the four jump opcodes. For example,

```
(COND (A B) (T (GO TAG))
```

would result in the fragment:

```
(VAR A) (FJUMP L2) ... L2: (JUMP TAG)
```

which can be transformed into

```
(VAR A) (FJUMP TAG) ...
```

Unreachable code <1670, removed 1784 instructions>

(JUMP tag) *{code}* ⇒ **(JUMP tag)**

The code after a JUMP or RETURN which is not itself jumped to can be deleted. The first pass avoids generating any constructs of this form, but such situations can be generated by other transformations. For example, in both preceding examples, the code at L2 might well be unreachable and deleted.

NTJUMP introduction <610>

val (TJUMP tag) ... tag: val ⇒ val (NTJUMP tag+1) ...

This optimization is essentially *COPY introduction* across jumps. For example,

```
(PROG NIL LP (FOO X) (COND ((SETQ X (CDR X)) (GO LP))) ...)
```

results in

```
LP: (VAR X) (FN 1 FOO) (POP) (VAR X) (FN 1 CDR) (SETQ X) (TJUMP LP) ...
```

which is then transformed to

```
(VAR X) LP1: (FN 1 FOO) (POP) (VAR X) (FN 1 CDR) (SETQ X) (NTJUMP LP1)
```

NTJUMP introduction with code movement <506>

```
    val (FJUMP tag) val {code1} ... tag: {code2}
 ⇒ val (NTJUMP tag2) {code2} tag2: {code1}
```

This transformation is a variation of *NTJUMP introduction* where it is necessary to move code around. The two code sequences *{code1}* and *{code2}* must end with a JUMP or a RETURN. Note that this transformation moves the entire segment of code *{code2}* inline. For example, the expressions

```
(COND (X (FN1 X)) (T (FN2) (GO LAB))
```

compile to

```
(VAR X) (FJUMP L1) (VAR X) (FN 1 FN1) (JUMP L2)
L1: (FN 0 FN2) (JUMP LAB) L2:
```

which gets transformed to

```
(VAR X) (NTJUMP L3) (FN 0 FN2) (JUMP LAB) L3: (FN 1 FN1) (JUMP L2) L2:
```

Jump to NIL/POP <834>

```
(FJUMP tag) ... tag: (CONST NIL) ⇒ (NFJUMP tag+1)
(NcJUMP tag) ... tag: (POP) ⇒ (cJUMP tag+1)
```

The pattern NcJUMP stand for either flavor of N-conditional jump. In the first situation, the NIL which is being found by the FJUMP may be logically distinct from the NIL after tag. For example, the expression

```
(COND (A ...) (T (MYFN NIL))
```

compiles as

```
(VAR A) (FJUMP L1) ... L1: (CONST NIL) (FN 1 MYFN)
```

which is transformed into

```
(VAR A) (NFJUMP L2) ... L2: (FN 1 MYFN).
```

The second form of the transformation normally occurs only after other transformations, where a conditional, originally thought to be executed for value, does not need the value being preserved by the NcJUMP.

Removal of loop variables <679>

```
    (SETQ var) (POP) (JUMP tag) ... tag: (VAR var)
 ⇒ (SETQ var) (JUMP tag+1)
```

This transformation is common in loops. For example,

```
(PROG NIL LP (PROCESS X) (SETQ X (NEXT X)) (GO LP))
```

compiles as

```
LP: (VAR X) (FN 1 PROCESS) (POP) (VAR X) (FN 1 NEXT) (SETQ X) (POP)
    (JUMP LP)
```

This transforms to:

```
LP: (VAR X) LP1: (FN 1 PROCESS) (POP) (VAR X) (FN 1 NEXT) (SETQ X)
    (JUMP LP1)
```

Cross jumping <1721>

{code} (JUMP **tag**) ... *{code}* **tag:** ⇒ (JUMP **tag2**) ... **tag2:** *{code}*

This frequent transformation improves code space with no effect on running time. For example, the expression

```
(COND (A (FOO X)) (T (FOO Y))
```

compiles as

```
(VAR A) (FJUMP L1) (VAR X) (FN 1 FOO) (JUMP L2)
L1: (VAR Y) (FN 1 FOO) L2:
```

The instruction before (JUMP L2) is identical to the instruction before the label L2, and so this can be transformed into

```
(VAR A) (FJUMP L1) (VAR X) (JUMP L3) L1: (VAR Y) L3: (FN 1 FOO)
```

Jump copy test <733>

val fn1 (jump tag) val ... tag: val ⇒ val (COPY) **fn1 (jump tag+1)**

In this transformation, **fn1** is a "clean" function of one argument, e.g., (FN 1 LISTP) or (FN 1 CDR), or even (CONST **val**) (FN 2 EQ). In this case, "clean" means that the function cannot change the value of **val**. For example, the expression:

```
(COND ((LISTP X) (CAR X)) ((NUMBERP X) (ADD1 X)))
```

results in the fragments

```
(VAR X) (FN 1 LISTP) (FJUMP L1) (VAR X) ... L1: (VAR X) (FN 1 NUMBERP)...
```

which transforms into

```
(VAR X) (COPY) (FN 1 LISTP) (FJUMP L2) ... L2: (FN 1 NUMBERP)...
```

Return optimizations

Return merge

(TJUMP **tag**) *{code}* (RETURN) ... **tag2:** *{code}* (RETURN)
⇒ (FJUMP **tag2**) ... **tag2:** *{code}* (RETURN)

This is an effective code transformation which can merge completely unrelated (with regard to flow-of-control) return sequences. It does not affect speed, only space. *Return merge* is unique in not preserving the normal invarient that stack-depth is constant at any location in the code. Normal code generation only creates sequences of instructions where the stack-depth at any location is static; all other transformations preserve that property. However, the two occurrences of *{code}* in the pattern need not be at the same stack-depth, and thus, stack-depth would be ambiguous after **tag2**. This is important if the target machine language is dependent upon stack depth in the translation from ByteLap, as is the case with the Maxc instruction set. *Return merging* must be disabled if the two *{code}* sequences occur at different stack depths, and if *{code}* contains any stack-level-sensitive operations.

Needless POP before RETURN <590>

(POP) **val** (RETURN) ⇒ **val** (RETURN)

This transformation is attempted only after it is known that there is no opportunity for *Unused push*. In addition to removing POP opcodes, this transformation also removes DUNBIND and UNBIND opcodes in the same position (except when **val** is a variable which was bound in the frame corresponding to the UNBIND or DUNBIND).

Unused variable in BIND <580>

(BIND ... (.. **var** ..)) *{**var** not used}* ⇒ (BIND ... (.. ..))
(BIND (.. **var**) ...) *{**var** not used}* ⇒ (POP) (BIND (..) ...)

This transformation eliminates binds of local variables which are not used. Only the last variable bound to a value can be so removed, because of the difficulty of inserting a POP at the appropriate place back in the instruction stream. (This is an example where source level transformation might be better way of doing optimization. Unfortunately, the last use of a variable is often removed by *COPY introduction*, which has no analogue in source code transformations.) To detect unused variables, the compiler scans the code linearly for uses of each variable in every BIND. For example, the expression

(PROG (X) (SETQ X (FUM)) (FOO X X))

compiles into

(BIND () (X)) (FN 0 FUM) (SETQ X) (POP) (VAR X) (VAR X) (FN 2 FOO)

which, after several transformations, turns into

(BIND () (X)) (FN 0 FUM) (COPY) (FN 2 FOO).

Since X is no longer used, it can be eliminated. Note that this transformation is not applicable to special variables (variables which can be referenced freely by functions called from this one, e.g., FUM and FOO).

Unused BIND <2035>

(BIND (**v1** ... **vm**) (**vm+1** ... **vn**)) (VAR **v1**) ... (VAR **vm**) *{last mention of **v1**...**vm**}*
⇒ (CONST NIL) *{n-m times}*

<Of the 2035 occurrences, 440 eliminated BINDs which were generated in the compilation of mapping functions.> This transformation eliminates BINDs when the variable list is empty or when the variables bound are only mentioned, in order, immediately following the BIND. When this transformation is made, the compiler must also find all corresponding DUNBIND's for this frame and turn them into the appropriate number of POP's. In addition, for every UNBIND the stack level must be exactly one greater than it was at the BIND. If so, the UNBIND can simply be deleted; if not, this transformation cannot be made. Note, however, that where a PROG or LAMBDA expression is the value returned by a function, no UNBIND or DUNBIND opcodes are generated. For example, the expression

((LAMBDA (X) (FOO X X)) (FUM))

compiles into

(FN 0 FUM) (BIND (X) ()) (VAR X) (VAR X) (FN 2 FOO)

which, after *COPY introduction* and *Unused BIND* can be transformed into

(FN 0 FUM) (COPY) (FN 2 FOO).

CONCLUSIONS

Because our instruction sets are so well suited to the Lisp language, it is possible to write quite simple non-optimizing compilers for our Lisp machines. In fact, we have written a simple but usable compiler in less than three pages of Lisp code. However, local transformations can have an important impact on code space and running time.

As in production systems, the choice of order of application of transformations can affect the results. Without effectively trying all possible orderings, one transformation can prevent a better one from being used. In successive transformations made on a sample of user Lisp programs, however, we have not observed this to be a major problem.

The programs our compiler generates are still not optimized, in the strict sense of that term. A sample of user Lisp programs which were "hand optimized" show that code size could be compressed by as much as an additional 15% in some cases, with no speed penalty. However, the transformations involved seem to require either much special-case pattern matching or else transformations which temporarily reduce either space or speed. As usual when employing "hill-climbing" algorithms, by requiring that all transformations we employ are strict improvements, we occasionally find local optima which prevent better solutions from being found.

Optimizing on a simple intermediate language is quite effective. Many of the transformations made are not expressible as source language transformations (e.g., the COPY operator has no direct counterpart in the Lisp language). Those that would be easier to express as source transformations are often enabled by transformations which have no direct analogue. Peephole optimizers working on more complex assembly languages must be aware of more special cases, because there are many more kinds of operations.

REFERENCES

Burton, R.R. *et al.*

Overview and implementation status of Interlisp-D. *Proceedings of the 1980 Lisp conference*, Stanford, 1980 and Xerox PARC, SSL-80-4, 1980.

Deutsch, L.P.

A Lisp machine with very compact programs. *Proceedings of the third international joint conference on artificial intelligence*, Stanford 1973.

Experience with a microprogrammed Interlisp system. *IEEE Micro-11 conference*, 1978.

ByteLisp and its Alto implementation. Proceedings of the *1980 Lisp Conference*, Stanford, 1980.

Fiala, E.R.

The Maxc systems. *IEEE Computer* 11, May 1978.

Griss, M.L. & Hearn, A.C.
A portable Lisp compiler. *Department of Computer Science, University of Utah, UCP-76,* 1979.

Steele, G.L.
RABBIT: A compiler for SCHEME (A study in compiler optimization). *MIT Artificial Intelligence Laboratory, AI-TR-474,* 1978.

Teitelman, W. *et al.*
Interlisp *Reference Manual*, Xerox PARC, 1978.

Urmi, Jaak
A machine independent Lisp compiler and its implications for ideal hardware. *Linkoping studies in science and technology dissertations No. 22*, Linkoping, Sweden, 1978.

The Interlisp-D I/O system

Ronald M. Kaplan, B. A. Sheil, and Richard R. Burton

Abstract

One of the major stumbling blocks to a transportable version of Interlisp is its extensive and complex set of input/output facilities. The Interlisp virtual machine (VM) specification provides assistance by specifying a smaller set of I/O primitives in terms of which the complete set can be written. However, the primitives described in the VM were decisively shaped by features of the Tenex operating system and transferring these primitives to other operating systems has proven to be difficult. This paper describes an implementation of the VM I/O primitives in Lisp which reduces the machine dependent aspects to the device level. The design breaks the I/O task into two levels: the logical level and the physical device level. The logical level defines the notion of an abstract file device and captures the operations and information that are common to all devices, such as sequential I/O and buffer management. The interface to the physical level is through a small set of device dependent functions such as opening a file or reading a page. The implementation of a new device requires writing appropriate versions of each of these functions, but the interface to this new device from the user level programs is exactly like any other file device. One difficult problem solved in this design is the concurrent accessing of a file by both sequential I/O and by page mapping. The primitives needed in this design are fewer and much easier to implement on widely varying hardware than those defined in the VM.

BACKGROUND

Interlisp-D [Burton *et al.*, 1980] is an implementation of the Interlisp programming system [Teitelman *et al.*, 1978] on the Dolphin and Dorado, personal computers with large virtual address spaces [Lampson & Pier, 1980]. Given Interlisp's extensive and complex set of input/output facilities, it is clear that these will comprise a substantial portion of any implementation. The Interlisp virtual machine (henceforth, VM) specification [Moore, 1976] provides some assistance by describing a smaller set of I/O primitives in terms of which the complete set can be written in Lisp. Therefore, one approach is to implement the VM primitives in terms of the I/O operations provided by the host operating system. This was the approach used in the AltoLisp implementation of Interlisp [Deutsch, 1980]. Reflections on this experience have convinced us that this is not a good implementation strategy.

Although the VM primitives are sufficient to support the higher levels of Interlisp, they are by no means the only such set, nor are they necessarily the most principled. In fact, the selection and definition of primitives were shaped in large measure by the properties of the Tenex operating system [Bobrow *et al.*, 1972] for which Interlisp was originally developed, and by the interface between Lisp and the machine-language kernel of Tenex-Interlisp. It is not an easy task to reconstruct this particular set of primitives with operating system capabilities that do not match those of Tenex. For example, Interlisp assumes the existence of a mechanism, modelled after the Tenex page-mapping mechanism, to map any page of an arbitrary file into the virtual memory without disturbing ongoing sequential I/O to that file. Few other operating systems provide such a capability and retrofitting it to an existing system is non-trivial.

As it turns out, many of the VM "primitives" are not in fact primitive: they can easily be expressed in Lisp in terms of a more basic set of operations. There are several good reasons for pursuing such a lower-level decomposition. First, the result would be a more elegant and satisfying implementation than an attempt to interface two incompatible operating system designs would produce. Second, writing more of the implementation in Lisp would reduce the amount of non-Lisp expertise (e.g., knowledge of the host operating system) required to build and maintain the I/O system. Third, the lower-level decomposition would represent a more principled understanding of the structure of this part of Interlisp than the current VM specification embodies. Specifically, it would move the boundary of Interlisp (the place where different implementations are free to vary) closer to the hardware, where the environmental differences are real, rather than leaving it at the operating system level, where the differences are mostly manufactured.

The most important reason, however, is that a lower-level decomposition would offer more assistance to other implementations of Interlisp. As the declining cost of hardware makes new environments for Interlisp more attractive, transportability becomes an increasingly important consideration. Consequently, we designed a framework for the lower levels of Interlisp I/O that serves not only as a basis for the Interlisp-D implementation but also is likely to be better than the current VM specification as a foundation for the I/O facilities of other implementations. This paper reports on that design.

DESIGN OVERVIEW

The basic distinction in the Interlisp-D I/O system is between the software used to control the various physical devices and a "logical" I/O layer that bridges the gap between the devices and the VM specifications. The logical layer, which is written entirely in Lisp, provides services that are usually supplied by code in the non-Lisp kernel or the operating system. It manages the allocation of virtual memory buffers and their assignment to specific pages in particular files, maintains the state information necessary to support sequential and random access streams, and coordinates stream and page-mapped file access.

The logical I/O system rests on an interface to the physical devices which defines the notion of an *abstract file device*. This characterizes the capabilities that physical file devices must possess and defines a uniform set of operations for manipulating those devices. Since physical devices (e.g., local disks and network file servers) come with widely varying software and hardware attributes, the design does not specify how the interface functionality is to be implemented for particular devices. Separate code is written for each device, perhaps even in different programming languages, in order to take advantage of and/or to compensate for the features that the device provides.

The device operations are defined at a very low level. The device does not manage the memory buffers into which it reads and writes, nor does it directly provide operations such as sequential, byte-stream access. These are implemented in a device independent way in the logical I/O system. The low level of the device interface has two advantages. First, the uniform interface for the logical I/O system guarantees that all devices will exhibit the same behavior in terms of the higher level protocols. Second, by minimizing the amount of code that must be written for each new device, it makes it relatively easy to add devices to the

system. Thus, when we found that substantial time was being spent reading and writing the compiler's temporary files, it was a simple matter to implement a CORE device. This provides the same user program abstraction as the disk but "reads" and "writes" from temporary storage in main memory, thus avoiding the cost of the actual data transfer.

THE ABSTRACT DEVICE

The device interface prescribes a set of *properties* on which devices may vary and a set of *operations* that may be applied to any device and the files that reside on it. A device may also perform additional operations which directly reflect its physical characteristics (e.g., REWIND on a tape device). These are for programs that need to make direct use of the idiosyncratic properties of specific devices. Each physical file device is known to user programs by a *device name*, with which is associated a *file device datum* containing both its device properties and its implementation of the generic operations. The device datum represents the operations in object-oriented style: It contains a vector of Lisp callable functions, one for each of the generic operations. The generic operations are defined simply to transfer control to the corresponding element of the appropriate file device.

The use of an object oriented representation at the device level reflects the great deal of variability in the underlying physical device hardware. Each device can decide independently which properties and operations it will support and which it will not. By contrast, in the logical I/O system this variability is much less pronounced, due to the constraints imposed by the Interlisp specification, so a more conventional functional interface is used.

The generic operations defined by the device abstraction were carefully chosen to allow the flexibility of different implementations where this was required by different real devices, while providing a uniform set of building blocks for higher level facilities. Thus, the virtual device provides operations to

- translate between file names and device-dependent file specifications (a *directory*),
- create, destroy, and "open" files,
- move information in page-size units between files and memory, and
- provide information about the status of a file on that device.

The interface between the device and its clients is in terms of *file names* and *file handles*. A file name is a user sensible identification of a file. This is presented to the logical file system when some I/O service is requested. A file handle is a device-generated, unique identification of a file on that device. It specifies the device on which the file resides, a complete name for the file and any other (device-dependent) information that may be needed to reference the file. Apart from name interpretation (directory services), a file handle is the interface to all device level file operations. Thus the operations to delete a file, read and write pages, interrogate status, etc., all require file handles as arguments.

The mapping between names and handles is one of the more complex aspect of the device interface. As the I/O system was specifically designed to allow uniform reference to files on file systems over whose naming conventions we had no control (e.g., remote file servers),

user sensible file names are considered to have two separate components: a device specification in a prescribed, uniform syntax and a device-specific file specification in an arbitrary syntax. When a name is presented to the I/O system, its device specification is used to locate the corresponding device datum. The name interpretation code of that device is then applied to the *complete* name.

Interlisp's more esoteric name interpretation facilities (the names presented to operations such as opening a file can be incomplete, specify wild-carding or expansion of various forms, or be misspelled) are handled at the logical level. This sometimes imposes some complexity on these algorithms since, for example, name interpretation cannot be made indivisible with resource allocation at the device level. Thus, if two processes "simultaneously" request the I/O system to open the next version of the same file, both will compute the same value for the next version, so one request will fail due to the other having succeeded in opening the file first. The logical file system discriminates and recovers from this situation. The alternative, an *expand-file-name-and-open-it* operation, is unattractive because it would replicate the Interlisp name expansion rules within every device.

Devices also include operations which interrogate their properties and those of their files. (GETFILEINFO *filehandle attribute*) returns the value of the specified attribute (e.g., AUTHOR, WRITEDATE, LENGTH) for the given file and (SETFILEINFO *filehandle attribute newvalue*) allows (some of) those values to be modified. The functions GETDEVICEINFO and SETDEVICEINFO are the analogous functions for obtaining information (such as its natural page-size, the amount of space currently free, and its name) about the device. Both sets of properties are device dependent; each device can decide to which properties it will respond.

THE LOGICAL I/O SYSTEM

In addition to being a complete specification of a file on some device, a file handle allows certain information about that file to be obtained. When the file is *opened*, it also permits the contents of the file to be accessed and provides the capability of changing all of the file's properties (dates, author, etc.). The act of opening the file is a device dependent operation, since the device may need to establish some special state information (e.g. a network socket for a remote file server). From the point of the client Lisp program, however, only the logical file state, which is maintained in an *open file descriptor* (OFD), is important.

An OFD contains the device independent information associated with an open file. This includes the file's file handle (which in turn specifies the file device), its access mode (either INPUT, OUTPUT, APPEND, BOTH, as specified by Interlisp), and an indication of the current file byte and line positions. All of this is completely device independent; device dependent information is all stored in the file handle. An OFD is created by the logical file system file-opening operation: (OPENFILE *name access recognitionmode*), where *access* indicates the access capabilities desired and *recognitionmode* the rules for completing an incomplete name. After computing a complete file name, this operation calls the device opening routine to do the device dependent opening operations, and then initializes the logical file state. If *access* is other than INPUT, the device opening operation will cause the file to be created if it does not already exist. When the file is closed, the OFD is marked as defunct so it cannot be further used.

The primary function of the logical I/O system is to fashion sequential and random access byte level I/O from the page level transfer operations provided by the device interface. To this end, the functions BIN (read a byte) and BOUT (write a byte) are defined. These keep track of the current file page and position within that page; detect page overflow, and, for input, end of file; and call the buffer management routines to replace buffers when the sequential access or random positioning causes the logical file position to leave the current page. Also implemented at this level are functions for obtaining and setting information about the state of the open file, such as the current file length.

These functions provide the implementation primitives in terms of which the existing higher level Interlisp I/O code is directly implemented.

USER BUFFER ACCESS

As we have seen, the device layer is not responsible for managing the virtual memory buffers that file data is transferred to and from. Buffer management is common to all devices and is therefore implemented in the logical I/O layer. Buffers are allocated and assigned to a file on demand and are accessible from its OFD. When BIN or BOUT reaches the end of a page, code in the logical layer chooses a buffer for the next page. If it chooses a buffer that already contains a page of the file and if that buffer has been written onto, then the device WRITEPAGE function will be invoked to flush the contents of the buffer back to the file. The buffer is then initialized by reading in the next file page (via READPAGE, which zeroes the buffer if the page does not yet exist in the file). The BIN or BOUT operation is then completed, updating the relevant fields in the OFD as appropriate to maintain the current position in the sequential stream.

When the logical close routine is called to close the file, its currently dirty buffers are first flushed to the file device, the buffers are de-allocated, and the device close operation is called on the file handle. The OFD is then marked as defunct and the file handle is marked closed, so that future attempts to use them will result in errors.

The buffer management strategy we have outlined is quite conventional. The user interacts with the buffers only through the carefully controlled interface functions that read and write a byte and reset the file position. The system is therefore free to allocate, deallocate, and recirculate buffers according to whatever heuristics seem appropriate. However, the Interlisp I/O specifications permit the user to gain direct access to the file buffers, and this imposes strong constraints on the buffer management regime.

Interlisp's page-mapping facility is similar to that of the Tenex operating system. It allows user programs to examine and modify information in a file page as if it were part of ordinary virtual memory. Complex data structures, not just linear byte sequences, can be created on the file and manipulated using the normal machinery for referencing memory. In particular, symbolic names can be defined for the fields of these structures by overlaying a record structure on a mapped in page, so that manipulation of file-resident databases becomes a very easy thing to do. Various Interlisp system packages (e.g. HASH, WHEREIS) depend on this capability, and it supports a variety of user application including the file-resident memory system of the KRL implementation [Bobrow & Winograd, 1977].

To a certain extent, this kind of access to file information can be obtained without new system facilities. If the user wants to view information on a file in a structured fashion, he can use the ordinary sequential I/O facilities to copy the data from the file buffer into a storage array. However, the user would not only pay the extra (but possibly small) cost of the copy, he would also have the responsibility of noticing when information in the array was changed and therefore has to be copied back to the file buffer. Considered in this light, implementing a system page-mapping operation merely centralizes the bookkeeping necessary to maintain consistency between the file and the in-memory image of its pages.

There is a more crucial feature of the page-mapping mechanism that simply cannot be simulated by user programs: the page-mapping specification provides for the coordination between modifications done to the page considered as a part of the address space and those done by the sequential I/O operations. Thus, doing a BOUT to a mapped page will cause an immediate modification to the corresponding memory word, and information placed in that word by a store operation will be returned by the next BIN operation. The user really *does* have a pointer to the system buffer for that file. This means that the reading and writing of structured information on a file page can be intermixed with input and output by the complete set of sequential I/O operations, including the higher-level READ and PRINT routines. A user can maintain his own data structures as long as that is easy and necessary, but he can fall back on the facilities for transferring arbitrary s-expressions when that becomes more suitable.

Distributing buffer pointers to the user seriously undermines the simple, conventional buffer management strategy. A buffer to which the user has been given a pointer must not be re-allocated as long as the user holds onto that pointer. By asking for a pointer, the user has indicated that he intends to operate on that file page for an extended period. This flies in the face of the pressure to re-circulate buffers for new pages and files in order to minimize the address space and working set devoted to buffers. The PMAP package in Interlisp-10 defines functions by which the user can lock and unlock buffers, but this does not completely solve the problem. First, there are situations where a buffer can be quietly recirculated before the request to lock it down can be processed, and second, the user might mistakenly unlock a buffer when in fact he retains a pointer to it that he later uses.

Interlisp-D permits an interesting and unique solution to this problem. Given that buffer management is implemented in Lisp, and given that Interlisp-D includes an incremental, reference-counting garbage collector, we have a very simple way of determining whether the user is still holding a buffer pointer that has been distributed to him. When a buffer pointer is given to the user, that buffer is marked so that it will not be recirculated according to the normal heuristics. The collection phase of the garbage collector distinguishes pointers to buffers from other kinds of data objects. When the reference count of a buffer goes to zero, the buffer is unmarked so that normal de-allocation is again enabled. Buffers for most files are unmarked so that the most efficient recirculation strategies may be employed, but deallocation of a user-mapped page is postponed until the next collection phase after the last user pointer to the page has been dropped. Thus, the dangling reference problem for file buffers is solved by a completely general mechanism.

PROBLEMS AND PLANS

The major weakness of the design presented here is that it is primarily oriented to *file* devices. Devices without file name structures, displays, and conversational devices fit less well into this framework. In large part, this is because these devices are given special treatment in the Interlisp I/O specifications, and we have replicated some of this awkwardness in our design. As an example of the type of problem that arises, consider the representation of a network connection. Such a connection is usually required, by the protocols of its network, to be a *duplex* object, i.e., it must be possible to both read and write independently to its input and output channels. Unfortunately, Interlisp assumes that an open file has but one logical file position, end of file, etc.. Following Interlisp, our I/O system makes the same assumption.

For the most part, these problems do not impact the utility of the design as a basis for the existing Interlisp I/O facilities, since special code to accomodate existing problem areas such as the controlling terminal is already in place. As we extend Interlisp into a personal computing environment, we will surely revisit these issues and attempt to give them a principled treatment. In particular, as many of the facilities that make Interlisp-10 such an attractive environment involve shared resources, making these available in a personal machine environment will require a comprehensive treatment of remote I/O. Unfortunately, the major part of this work will be extending the existing semantics of Interlisp I/O.

Acknowledgements

The Interlisp-D I/O system evolved from an earlier design done by Peter Deutsch for the Alto computer. Willie Sue Haugeland navigated us through the shoals of the Alto operating system. We are also happy to acknowledge the contributions of Dan Bobrow, Richard Fikes, and Larry Masinter to the design and implementation of the system reported here.

REFERENCES

Bobrow, D.G. *et al.*
Tenex, a paged time sharing system for the PDP-10, *CACM*, March 1972.

Bobrow, D.G. & Winograd, T.
Overview of KRL0: A knowledge representation language. *Cognitive Science*, 1, 3-46, 1977.

Burton, R. R. *et al.*
Interlisp-D: Overview and status. *Proceedings of the 1980 Lisp Conference*, Stanford and Xerox PARC, SSL-80-4, 1980.

Deutsch, L.P.
A Lisp machine with very compact programs. *Proceedings of the third international joint conference on artificial intelligence,* Stanford 1973.

Experience with a microprogrammed Interlisp system. *IEEE Micro-11 conference*, 1978.

Lampson, B.W. & Pier, K.A.
A processor for a high-peformance personal computer. *Seventh international symposium on computer architecture*, La Baule, France, May 1980.

Moore, J.S.
The Interlisp *virtual machine specification*. Xerox PARC, CSL-76-5, 1976.

Teitelman, W. *et al.*
Interlisp *Reference Manual*, Xerox PARC, 1978.

Interlisp-D Display Facilities

Richard R. Burton

Abstract

As one of the goals of Interlisp-D is to make Interlisp available as a personal computing environment, it incorporates an extensive set of graphics facilities. This memo documents the abstractions and functions which comprise these facilities.

INTRODUCTION

This memo documents the abstractions and functions which have been designed to support the use of the Interlisp-D display. These functions provide the display primitives upon which DLISP is based. Their design was initially based on the ADIS primitives [Sproull, 1979] for the Alto and was later influenced by other graphics work at Xerox PARC [Warnock, 1980].

This document is intended both to document the existing facilities and to provide a framework within which extensions to the Interlisp graphics interface can be made. It is hoped that these primitives will provide a standard for Interlisp display facilities at a corresponding level in other implementations and that the framework will be extended (by ourselves and others) to more general graphics devices such as color, grey scale and high resolution printing media.

Geometric Operations

The display facilities provide three different types of geometric procedures: figure generating, transformations and clipping. Figure generating procedures include routines to place text, draw lines and curves, and fill in areas. Transformation routines allow programs to construct images with local coordinate systems, provide translation and will be extended to include scaling and rotation. The clipping routines allow an image to be clipped against a region (currently a single rectangular region, extending to a set of arbitrary polygons.)

POSITION

A Position denotes a point in a coordinate system. It is characterized by its x and y coordinates. A POSITION is an instance of a record with fields XCOORD and YCOORD. It is manipulated with the standard record package facilities.

REGION

A Region denotes a rectangular area in a coordinate system. It is primarily used to specify clipping regions which limit the areas into which figures are displayed. Regions are characterized by the coordinates of their bottom left corner and their width and height. A REGION is a record with fields LEFT, BOTTOM, WIDTH and HEIGHT. It is manipulated with the standard record package facilities. The global variable WHOLEDISPLAY is a region which covers the entire display screen.

BITMAPS

The primitives manipulate graphical images in the form of **bitmaps**. A bitmap is a rectangular array of bits. If a bit is 0, the corresponding location on the image is white. If a bit is 1, its location is black. Bitmaps use a positive integer coordinate system with the lower left corner bit being (0,0). Bitmaps are represented as instances of the datatype (BITMAP) with fields BITMAPWIDTH, BITMAPHEIGHT, BITMAPRASTERWIDTH and BITMAPBASE. Only the width and height fields are of interest to the user. (For Interlisp-D, the BITMAPRASTERWIDTH is the number of words required to hold one line of the bitmap; the BITMAPBASE is a pointer to the first word of the bits.)

To extend the display scheme to higher resolution devices, the notion of bit is changed to "pixel" and fractional parts of pixels can be darkened or greyed as required. To handle this extension, the coordinate system is represented in floating point numbers rather than integers. For expediency, the initial version of the display facility does not include this capability. It is expected that the current framework will extend in this manner.)

There are two distinguished bitmaps that are "read" by the hardware to become visible as the screen and the cursor. The screen is a bitmap SCREENWIDTH wide by SCREENHEIGHT high. (For Interlisp-D, SCREENWIDTH is 620, SCREENHEIGHT is 808.) The cursor is CURSORWIDTH by CURSORHEIGHT. (For Interlisp-D, CURSORWIDTH is 16, CURSORHEIGHT is 16.)

The functions to manipulate bitmaps are:

BITMAPCREATE[Width Height]
: Creates and returns a new bitmap which is *Width* bits wide by *Height* bits high.

BITMAPBIT[Bitmap X Y NewValue]
: *X* and *Y* are measured (as always) from the left bottom, 0 as origin. If *NewValue* is 0 or 1, the bit (*X*,*Y*) is changed to *NewValue* and the old value is returned. If *NewValue* is NIL, the *Bitmap* is not changed but the value of the bit is returned.

BITMAPCOPY[Bitmap]
: Returns a new bitmap which is a copy of *Bitmap* (same dimensions and contents).

There are two distinguished bitmaps that are "read" by the hardware to become visible as the screen and the cursor. The screen is a bitmap SCREENWIDTH wide by SCREENHEIGHT high. (For Interlisp-D, SCREENWIDTH is 620, SCREENHEIGHT is 808.) The cursor is CURSORWIDTH by CURSORHEIGHT. (For Interlisp-D, CURSORWIDTH is 16, CURSORHEIGHT is 16.) They are accessed by:

SCREENBITMAP[]
: Returns the screen bitmap.

CURSORBITMAP[]
: Returns the cursor bitmap.

BITBLT

BITBLT is the primitive function for moving bits from one bitmap to another.

BITBLT[SourceBitmap SourceLeft SourceBottom DestinationBitmap DestinationLeft
DestinationBottom Width Height SourceType Operation Texture ClippingRegion]

Width and *height* define a pair of rectangles, one in each of the *SourceBitmap* and *DestinationBitmap* whose left, bottom corners are at, respectively, (*SourceLeft*, *SourceBottom*) and (*DestinationLeft*, *DestinationBottom*). If these rectangles overlap the boundaries of either bitmap they are both reduced in size (without translation) so that they fit within their respective boundaries. If *ClippingRegion* is non-NIL it should be a Region and is interpreted as a clipping region within *DestinationBitmap*; clipping to this region may further reduce the defining rectangles. These (possibly reduced) rectangles define the source and destination rectangles for BITBLT.

The mode of transferring bits is defined by *SourceType* and *Operation*. The *SourceType* and *Operation* specify boolean functions that are used to determine, respectively, the method of combining the *SourceBitmap* bits with the *Texture* and the operation between these resultant bits and the *DestinationBitmap*. The specification given below defines the modes allowed by Interlisp-D; extensions are seen as necessary for other implementations, in particular those providing color or grey scale.

Texture is a gray pattern, as described in the section below.

Sourcetype specifies how to combine the bits of an input bitmap (in this case, the pattern specifying the character) with the bits from a texture (background pattern; see below) to produce a **source**. This is designed to allow characters and figures to be placed on a background.

SourceType	Source
SOURCEINPUT	Input
SOURCEINVERT	(NOT Input)
SOURCEMERGE	(AND Input Texture)
SOURCETEXTURE	Texture

The various SourceTypes such as SOURCEINPUT are global variables which are declared as constants. For the SOURCEINPUT case, the *Texture* argument to BITBLT is ignored. For the SOURCETEXTURE case, the *SourceBitmap* argument is ignored.

Operation specifies how this source is combined with the bits in the Destination bitmap and stored back into the Destination bitmap.

Operation	Destination
OPREPLACE	Source
OPPAINT	(OR Destination Source)
OPFLIP	(XOR Destination Source)
OPERASE	(AND Destination (NOT Source))

The various Operations such as OPREPLACE are global variables which are declared as constants.

SourceBitmap and *DestinationBitmap* default to the screen. *SourceLeft*, *SourceBottom*, *DestinationLeft* and *DestinationBottom* default to 0. *Width* and *Height* default to the width and height of the *SourceBitmap*. *Texture* defaults to WHITESHADE. *SourceType* defaults to INPUT. *Operation* defaults to REPLACE. If *ClippingRegion* is not provided, no additional clipping is done. BITBLT returns T if any bits were moved; NIL otherwise.

TEXTURE

A Texture denotes a pattern of gray which can be used by BITBLT to (conceptually) tessellate the plane to form an infinite sheet of gray. For Interlisp-D, it is a 4 by 4 pattern. Textures are created from bitmaps.

CREATETEXTUREFROMBITMAP[Bitmap]

Returns a texture object that will produce the texture given from *Bitmap*. If *Bitmap* is too large, its lower left portion is used. If *Bitmap* is too small, it is repeated to fill out the texture.

The common textures white, black and gray are available as system constants WHITESHADE, BLACKSHADE and GRAYSHADE. The alignment of the texture pattern with BITBLT is such that the origin of the destination bitmap is at an intersection of the "tiles".

SAVING BITMAPS

Bitmaps can be saved on files with the file package command BITMAPS, analogous to the file package ARRAYS command. This uses the two functions PRINTBITMAP and READBITMAP which translate bitmaps into and out of numeric representations which may be used to transfer bitmaps from other systems.

READBITMAP[Width Height BitsPerInteger]

Creates a bitmap which is *Width* by *Height* bits in size and gets values for its bits by READing an expression that should be a list of integers. (This convention is adopted from the method of saving arrays on files.) *BitsPerInteger* is the number of low order bits that should be taken from each integer in the read list. Each line of the bitmap begins on a new integer. Thus, the list of integers should be (((*Width*-1)/*BitsPerInteger*)+1)**Height* elements long. If *Width* is not a multiple of *BitsPerInteger*, the most significant of the *BitsPerInteger* bits from the last integer of each line will be used. This design allows bitmaps to be written on the files in an implementation independent way. *BitsPerInteger* should be kept small (less than 20) so that the integers can be read by the READ function on machines of smaller word size without overflow. For Interlisp-D, *BitsPerInteger* is 16.

PRINTBITMAP[Var]

Var is an atom whose value should be a bitmap. The function prints a call to the function READBITMAP with the appropriate values followed by a list of the integers representing the pattern of bits in that bitmap.

SUPPORT FOR THE MOUSE

The screen relative position at which the cursor bitmap is being displayed can be read or set using the functions:

CURSORPOSITION[Position]

This returns the present location of the cursor. If *Position* is non-NIL, it should be a position and the cursor will be positioned at *Position* relative to the whole screen.

ADJUSTCURSORPOSITION[deltax deltay]

Positions the cursor offset from its current location by deltax and deltay which are integer increments which default to 0.

The cursor can be changed like any other bitmap by BITBLTing into it or pointing a display stream at it and printing or drawing curves. However, for pointing applications, it is necessary to locate the "hot spot" within the CURSORWIDTH by CURSORHEIGHT area which is used to determine *a point* position for the cursor. The function:

SETCURSOR[Bitmap X Y]

Copies *Bitmap* into the cursor and indicates to the system that location (*X*,*Y*) within that area is used as the hot spot; i.e., the value of CURSORPOSITION. If *Bitmap* has dimensions different from CURSORWIDTH by CURSORHEIGHT, the lesser of the widths and the lesser of the heights are used to determine how many bits actually get copied into the lower left corner of the cursor. If *X* or *Y* is NIL, that coordinate is not changed. For Interlisp-D the default cursor is the uparrow and the default hot spot is (0,15), the upper left corner which is the tip of the arrow.

Reading the Mouse

The mouse can be read in either a polling or in a queued manner. For polling, use

GETMOUSESTATE[ ]

Reads the current state of the mouse and sets the variables LASTMOUSEX, LASTMOUSEY, LASTMOUSEBUTTONS, LASTMOUSETIME, and LASTKEYSET (which holds the state of the five finger keyset.) In Interlisp-D, these are all 16-bit positive integers. Since the time is in milliseconds, it rolls over every 64 seconds or so. In polling mode, the program must remember the previous state and look for changes such as a button or key going up or down or the position moving outside a region of interest.

In queuing use, the state of the mouse is saved whenever there is a transition in one of the mouse buttons or keyset keys. In this mode the button clicks are treated much like typed in keyboard strokes; saved until the program next asks for them. To coordinate the keyboard with the mouse clicks, the mouse interrupt handler will put a designated character in the keyboard input buffer. A common practice is to have this character be a read macro character that handles the mouse event. To tell the system to start queueing mouse events,

ENABLEMOUSE[#EVENTS CHARCODE]

#EVENTS gives the number of events to save (Interlisp-D limit is 49). If more than *#EVENTS* events occur, further events are ignored and the screen is flashed.

CHARCODE is the character code that will be inserted into the keyboard stream when the mouse event occurs. If *CHARCODE* is not given, nothing will be put into the keyboard buffer. If *CHARCODE* is given, CLEARBUF will clear the mouse queue as well as the input buffer. If *CHARCODE* is not given, the mouse queue can be cleared by recalling ENABLEMOUSE with the same arguments. To turn off mouse queuing, ENABLEMOUSE is called with no arguments.

GETMOUSEEVENT[FLG]

Examines the state of the mouse queue and may read the next event (set the variables mentioned above.) If *FLG* is NIL, GETMOUSEEVENT returns the time of the next event if there is one (returns NIL if there are no events) but does not read it. If *FLG* is not NIL, the time is returned and the event will be "read" into the variable LASTMOUSEX, etc.

MOUSEBUF[FLG]

If *FLG* is T, this returns ithe internal buffer of mouse events that were saved at the last CLEARBUF[T T]. If *FLG* is NIL, the internal mouse event buffer is cleared.

BKMOUSEBUF[EVENTLST]

EVENTLST is a list of mouse events. BKMOUSEBUF sets the mouse queue to this list of events. In typical usage, *EVENTLST* is a list obtained by a call to MOUSEBUF. The form of the mouse events is a list of (MouseX MouseY MouseButtons MouseTime KeySet).

DISPLAY STREAMS

Display Streams allow uniform, convenient manipulation of bitmaps. Display streams have the properties necessary to implement transformation, clipping and aspects of figure generation. One property of display streams is the bitmap they modify called its *Destination*. Changing the destination to an auxilliary bitmap can be used to construct figures, possibly save them and then display them in a single operation. Display streams have their own coordinate system and a current *Position* in that system which is changed as characters are printed or lines drawn. Having the coordinate system local to the display stream allows objects to be displayed at different places on by translating the display stream's coordinate system relative to its destination bitmap's. The translation is given by x and y *Offsets*. Display streams also have a *ClippingRegion* which limits the extent of both characters displayed and lines drawn. Display streams have a *Font* that consists of a font family, a size and faces (Italic, Bold), and that dictates how characters appear. Other properties of a Display stream are its *Operation* (how the characters or lines should be integrated with the bits that are already on the screen eg. REPLACED, FLIPed, etc.); *Texture* (of gray for background); how far the y position is advanced on a *LineFeed*; where the *LeftMargin* is; and whether or not to *Scroll* the contents when reaching the bottom of the clipping region. Display streams also have *Brush* characteristic for drawing curves.

Functions are provided for creating DisplayStreams, and manipulating both them and their component parts. The package also supports the notion of a *current* DisplayStream, which can be set and manipulated, and which is used implicitly by omitting a DisplayStream argument from functions which take such arguments.

There are two general types of figure generating procedures: character printing, and line and curve drawing. Display streams are recognized throughout the system as a legal file argument. Characters are printed using the normal Lisp print functions (PRINT, PRIN1, etc.) by giving a display stream as the file argument. Functions are provided to draw lines and curves.

MANIPULATING DISPLAY STREAMS

The attributes of a DisplayStream include:

Destination	a bitmap
XPosition	an integer which is the current x position (in the display stream's coordinate system)
YPosition	an integer which is the current y position (in the display stream's coordinate system)
XOffset	an integer which is the x translation of the display stream's coordinate system from the bitmap's.
YOffset	an integer which is the y translation of the display stream's coordinate system from the bitmap's.
ClippingRegion	a Region which limits the extent of lines and characters
SourceType	a BITBLT source type
Operation	a BITBLT operation
Texture	a Texture which is the background pattern
Font	a FontDescriptor
Italic	ON or OFF
Bold	ON or OFF
Scroll	ON or OFF (If ON, the destination is scrolled up after an EOL enough to have the next printed character appear.)
Left margin	an integer which is the X position after an EOL (in the display stream's coordinate system)
Linefeed	an integer which specifies the Y increment each linefeed
Brush	a bitmap which is used to draw curves

The default values for these characteristics are:

Destination	the screen bitmap
XPosition	0
YPosition	0
XOffset	0 (no x-coordinate translation)
YOffset	0 (no y-coordinate translation)
ClippingRegion	set so that no clipping occurs
SourceType	'INPUT
Operation	'REPLACE
Texture	WHITESHADE
Font	HELVETICA10
Italic	'OFF
Bold	'OFF
Scroll	'OFF

Left margin	0
Linefeed	minus the height of the font
Brush	a bitmap of a single bit

DisplayStreams are represented as instances of the datatype DisplayStream.

The following functions manipulate the fields of a DisplayStream. The functions return the old value (the one being replaced). A value of NIL for the new value will return the current setting without changing it. This provides a uniform way of "reading" the current setting. In the case of fields which can only be either OFF or ON, NIL returns the current setting, the special value OFF turns the feature off, and anything else turns it ON. These functions do not change the destination bitmap; just the effect of future operations done through the display stream.

DSPCREATE[ ]
: Returns a new DisplayStream, settings are copies of the initial DisplayStream (see above).

DSPDESTINATION[Destination DisplayStream]

DSPXPOSITION[XPosition DisplayStream]

DSPYPOSITION[YPosition DisplayStream]

DSPCLIPPINGREGION[Region DisplayStream]

DSPXOFFSET[XOffset DisplayStream]

DSPYOFFSET[YOffset DisplayStream]

DSPSOURCETYPE[SourceType DisplayStream]

DSPOPERATION[Operation DisplayStream]

DSPTEXTURE[Texture DisplayStream]

DSPSCROLL[SwitchSetting DisplayStream]
: Controls whether or not the bitmap contents are moved up when a linefeed would put any of the next line of characters off the bitmap.

DSPLEFTMARGIN[XPosition DisplayStream]

DSPLINEFEED[DeltaY DisplayStream]
: This is the amount the y coordinate is increased by when a linefeed is printed. It is normally a small negative number.

DSPBRUSH[Bitmap DisplayStream]

The font related functions DSPFONT, DSPITALIC and DSPBOLD are described in the section on display stream font functions below

There is a distinguished display stream, called the *current display stream*, which is used by any function which is given NIL as a DisplayStream argument. To change the current DisplayStream, use the function

CURRENTDISPLAYSTREAM[DisplayStream]

As is the case with the other functions, CURRENTDISPLAYSTREAM returns the old value of the current display stream. If DisplayStream is NIL, the current display stream is not affected.

MISCELLANEOUS OPERATIONS ON DISPLAY STREAMS

DSPFILL[Region Texture Operation DisplayStream]

Fills *Region* of the destination bitmap (within the clipping region) with *Texture* (a pattern of bits). If *Region* is NIL, the whole destination (within the clipping region) is used. If *Texture* or *Operation* are NIL, the values from *DisplayStream* are used.

DSPRESET[DisplayStream]

Sets the position of *DisplayStream* to its (0,0) position and clears its destination to its background Texture.

DSPBITBLT[SourceDisplayStream SourceLeft SourceBottom DestinationDisplayStream DestinationLeft DestinationBottom Width Height SourceType Operation Texture]

Similar to BITBLT but uses the coordinate systems of the *SourceDisplayStream* and *DestinationDisplayStream* to do the transferring. The rectangle of bits (*SourceLeft SourceBottom Width Height*) in *SourceDisplayStream*'s destination are clipped by *SourceDisplayStream*'s clipping region and tranferred to the rectangle (*DestinationLeft DestinationBottom Width Height*) in *DestinationDisplayStream*'s destination clipped by *DestinationDisplayStream*'s clipping region using the *SourceType* and *Operation* to determine the tranfer function. If *SourceLeft*, *SourceBottom*, *DestinationLeft* or *DestinationBottom* are NIL, 0 is used. If *Width* or *Height* is NIL, the positive quadrant of the *SourceDisplayStream* is used. If *SourceType* or *Operation* is NIL, the values from *DestinationDisplayStream* are used. Returns T if any bits were transferred, NIL otherwise.

DSPBIT[X Y newvalue DisplayStream]

Similar to BITMAPBIT but uses the coordinate system of *DisplayStream*. If *newvalue* is 0 or 1, the bit (*X*,*Y*) of the destination bitmap of *DisplayStream* is changed to *newvalue* and the old value is returned. If *newvalue* is NIL, the bit is not changed but the value of the bit is returned.

MOVETO[x y DisplayStream]

Changes the current position of *DisplayStream* to the point (*x*,*y*).

RELMOVETO[dx dy DisplayStream]

Changes the current position to the point (*dx*,*dy*) coordinates away from current position of *DisplayStream*.

CHARACTERS AND FONTS

Fonts are viewed as having a distinctive form or family name (such as Helvetica, Gacha or TimesRoman), a size and some face characteristics (e.g, bold and italic). Using a display stream, each of these parameters can be changed and the characters appearing on that

display stream will henceforth be in the changed font. While the specification allows any size, in practice the user will find that only certain sizes are available. Note that the display stream functions that change the font (DSPFONT, DSPBOLD and DSPITALIC) may change other attributes of the display stream, e.g. the line feed height to the height of the new font.

Most users will deal with fonts only by way of display streams and hence can skim to the next section.

A font is characterized by an ascent, descent and height (= ascent + descent), and, for each character, a width and bitpattern. The ascent is the maximum extent of any character in the font above its base line (the printing position). The descent is the maximum extent of any character below the base line such as the lower part of a "p". The width of each character is the number of bits in width used by that character and can vary in "variable pitch" fonts.

The information about a particular fully instantiated font is represented in a FontDescriptor. Functions to manipulate FontDescriptors are:

FONTCREATE[FontFamily Size Face ErrorFlg]

Returns a FontDescriptor for the specified font. *Size* is an integer indicating the width of the font in points. *Face* specifies the face characteristics and should be one of (STANDARD, BOLD, ITALIC or BOLDITALIC). If *Face* is NIL, STANDARD is used. For Interlisp-D, fonts are stored as STRIKE files. The operation of FONTCREATE is to look for a STRIKE file with the appropriate name. (In the case of (FONTCREATE 'HELVETICA 8 'BOLDITALIC), the first file looked for is HELVETICA8BI.STRIKE.) If the file is found, it is read into a FontDescriptor. If the file is not found, the function will look for fonts with less face information (in the example, HELVETICA10I.STRIKE) and "fake" the remaining faces (by doubling the bits in the pattern of each character or slanting them). If no appropriately sized font is found, the action of the function is determined by *ErrorFlg*. If *ErrorFlg* is NIL, it returns NIL. If *ErrorFlg* is non-NIL, it will generate a "file not found" error with the name of the most general file tried (in the example HELVETICA8.STRIKE) (in the example HELVETICA8.STRIKE).

FONTNAME[FontDescriptor]

Returns the font name of the described font.

FONTSIZE[FontDescriptor]

Returns the font size of the described font.

FONTFACE[FontDescriptor]

Returns the font face of the described font: STANDARD, BOLD, ITALIC or BOLDITALIC.

FONTASCENT[FontDescriptor]

Returns the ascent of the described font.

FONTDESCENT[FontDescriptor]

Returns the descent of the described font.

FONTHEIGHT[FontDescriptor]

Returns the height of the described font.

CHARACTERWIDTH[CharacterCode FontDescriptor]

CharacterCode is an integer that describes a valid character. If *FontDescriptor* is a DisplayStream, its font is used. It returns the width of the bit pattern of the character.

STRINGWIDTH[Str FontDescriptor Flg Rdtbl]

Str is a lisp object. It returns the width of the bit pattern of the printname for the object if printed in *FontDescriptor*. If *FontDescriptor* is a DisplayStream, its font is used. If *Flg* is non-NIL, the width of the PRIN2-pname with respect to the readtable *Rdtbl* is returned.

DISPLAY STREAM FONT OPERATIONS

The following functions are provided for dealing with a display stream's font characteristics:

DSPFONT[Font Size Face DisplayStream]

Sets the font of *DisplayStream*. (This also sets the linefeed to the height of the font.) *Font* is either a FontDescriptor or the name of a font family. *Size* is an integer indicating the font size. *Face* is one of STANDARD, BOLD, ITALIC or BOLDITALIC. If *Font* is a FontDescriptor, its font name and size are used. If *Face* is given and is different from the face of *Font* which is a FontDescriptor, the face of the new font is determined by combining the bold and italic attributes of *Font* with *Face*. If *Font*, *Size* or *Face* are NIL and *Font* is not a FontDescriptor, their values are not changed. The value returned is the FontDescriptor of the previous font.

DSPBOLD[SwitchSetting DisplayStream]

Sets the bold switch (= changes the font face) of *DisplayStream*.

DSPITALIC[SwitchSetting DisplayStream]

Sets the italic switch (= changes the font face) of *DisplayStream*.

BLTCHAR[CharacterCode DisplayStream]

BLTCHAR will display the bit representation of *CharacterCode* in the font of *DisplayStream* at the current position of *DisplayStream* using the face charateristics, clipping region, source type, operation and texture of *DisplayStream*. *CharacterCode* is an integer representation of a character (as returned from CHCON1). If the character is not an EOL, BLTCHAR increases the X position of *DisplayStream* by the width of the character. If the character is an EOL, BLTCHAR increases the Y position by *DisplayStream*'s Linefeed and resets the X position to its Left Margin.

DSPBACKUP[Width DisplayStream]

Backs up *DisplayStream* over a character which is *Width* points wide. DSPBACKUP fills the backed up over area to the display stream's background texture. DSPBACKUP decreases the X position by *Width*. If this would put the X position less than *DisplayStream*'s left margin, its operation is stopped at left margin. It returns T if any bits were changed, NIL otherwise.

DRAWING LINES AND SPLINES

The display facility implements the following functions for drawing lines on bitmaps:

DRAWTO[x y width operation DisplayStream]

Draws a line *width* points wide from the current position to the point (*x*,*y*) onto the destination bitmap of *DisplayStream*. The clipping region is taken from *DisplayStream*. *Width* is defaulted to 1. *Operation* is a BITBLT operation which indicates how the bits of the line should be merged with the existing bits. *Operation* is defaulted to the Operation of *DisplayStream*. The position of *DisplayStream* is left at (*x*,*y*).

RELDRAWTO[dx dy width operation DisplayStream]

Draws a line *width* points wide from the current position to the point (*dx*,*dy*) coordinates away onto the destination bitmap of DisplayStream. The clipping region is taken from *DisplayStream*. *Width* is defaulted to 1. *Operation* is a BITBLT operation which indicates how the bits of the line should be merged with the existing bits. *Operation* is defaulted to the Operation of *DisplayStream*. The position of *DisplayStream* is left at the end of the line.

DRAWLINE[x1 y1 x2 y2 width operation DisplayStream]

Draws a line *width* points wide from the point (*x1*,*y1*) to the point (*x2*,*y2*) onto the destination bitmap of *DisplayStream*. The clipping region is taken from *DisplayStream*. *Width* is defaulted to 1. *Operation* is defaulted to the Operation of *DisplayStream*. The position of *DisplayStream* is left at (*x2*,*y2*). The cases of horizontal and vertical lines are recognized so the users should not feel the need to call BITBLT directly for line drawing applications.

Curves

Curves are drawn using spline techniques. At each point along the spline, the brush bitmap is placed, positioned so that its center falls on the spline. The brush can be an arbitrary bitmap. (In Interlisp-D, a special case is made of the single point bitmap for efficiency.) However, some standard shapes and sizes are provided by the following function.

BRUSHBITMAP[BrushShape BrushWidth]

Returns a bitmap for a brush of shape *BrushShape* and size *BrushWidth*. For Interlisp-D, the recognized brush shapes are ROUND, SQUARE, HORIZONTAL, VERTICAL and DIAGONAL. *BrushWidth* indicates how wide the brush is to be. For Interlisp-D, *BrushWidth* is rounded to the nearest power of two and limited to a maximum of 16.

In the curve drawing functions, *operation* is a BITBLT operation which indicates how the curve brush bits should be merged with the existing bits. Because of the problem of overlapping brush points, in Interlisp-D only the PAINT and ERASE operations are supported. The other operations can be obtained by first drawing the curve in an auxilliary bitmap and then bitblting it (DSPBITBLT or BITBLT) with the desired operation.

DRAWSPLINE[knots closed DisplayStream]

Draws a spline curve. *Knots* is a list of positions the spline must go through. *Closed* is a flag which indicates whether or not the spline is to be closed. The operation, brush and clipping region are taken from *DisplayStream*.

DRAWCIRCLE[x y radius quadrants DisplayStream]

Draws a circle with a radius of *radius* about the point (*x*,*y*) onto the destination

bitmap of *DisplayStream*. *Quadrants*, if given, is a list of the quarter circles that should be displayed, numbered counterclockwise from 1 to 4 with +x,+y being 1. The operation, brush and clipping region are taken from *DisplayStream*.

APPENDIX

This appendix documents some parts of the Interlisp-D graphics facilities that will soon be deimplemented. These are described here both for the purpose of documenting them until they are obsolete and as a suggestion to other implementors as to how to cope with the transition to a system that is fully based on the display.

Initialization

This section is temporary until "display mode" is the only way of interacting with the display.

DISPLAYSTREAMINIT[N TTYATBOTTOMFLG]

Before you use the display, the display stream facility must be initialized. This is done with the function DISPLAYSTREAMINIT which clears the screen and leaves N lines of "teletype simulation area" at the top or bottom depending on the setting of TTYATBOTTOMFLG. The teletype region gets the normal Lisp TTY output while the rest of the screen is changed with various Display Stream functions. The variables \DisplayWidth and \DisplayHeight are set to the width and height of the display. The bottom left corner is (0,0).

To initialize the DisplayStreams facility without effecting the display, use

INITIALIZEDISPLAYSTREAMS[]

Sets up the DisplayStream mechanisms. Automatically calls InitializeDisplay if that function has not been already been called. Rreturns a "default" DisplayStream which is also made the current DisplayStream.

To couple the hardware to the special bitmaps, use

STARTDISPLAY[width height TTYAtBottom]

Height refers to how much of the screen is to be used in the bitmap (as opposed to teletype) mode. Width determines how much of the screen will be seen but the extra width is not used if this is less than \ScreenWidth (currently 620). If these are NIL, then the values from the last call are used (or reasonable defaults if this is the first call). The remainder of the screen is in teletype mode. The teletype portion is at the top of the screen normally; if TTYAtBottom is non-NIL then the TTY portion is at the bottom of the screen. This function (like InitializeDisplayStreams) will automatically call InitializeDisplay if that function has not been called before this one is. This function can be called repeatedly to change how much of the screen is to be used in bitmap mode. The bits in the screen bitmap are not cleared by this operation so that if the same width is given, the screen will be part of what was there before.

To return to the state where the full display is in teletype mode, use the function:
STOPDISPLAY[ ]

To find out whether the special display bitmaps have been initialized, use the function: DISPLAYINITIALIZEDP[ ]

To find out whether the DisplayStreams have been initialized, use the function: DISPLAYSTREAMSINITIALIZEDP[ ]

To find out whether the display is currently on, use the function: DISPLAYSTARTEDP[ ]

Preservation of state over LOGOUT and SYSOUT

All bitmaps, FontDescriptors and DisplayStreams are unaffected by SYSOUTs and LOGOUTs. The system calls STARTDISPLAY again to continue after SYSOUT automatically.

Display stream insert mode

This mode was included in an effort to speed up the DLISP editor. It is an inadequate solution which has problems with seaming of the background texture. If a better solution can be found, this will be deimplemented.

Before the character is placed, if DisplayStreamInsert is non-NIL, the region of the DisplayStream's bitmap from the left edge of the character to the right edge of the clipping region (and with height of the active font) is moved right by the width of the character. That is, the character is "inserted". This may cause seams to appear if characters are being printed on a gray background. [Implementation note: The function BltChar returns a dotted pair the CAR of which is T if any of the character fell within the clipping region, otherwise NIL; the CDR of which is T iff the insertion was to the left of the clipping region and caused unknown stuff to be "pushed" into view, NIL otherwise. Unknown stuff is denoted by reverse background.]

Acknowledgements

Warren Teitelman, Austin Henderson and Willie Sue Haugeland designed the initial version of the display facilities. This document benefited by discussions with Warren Teitelman, Peter Deutsch, and John Warnock.

REFERENCES

Sproull, R.
Raster graphics for interactive programming environments. Xerox PARC, CSL-79-6, 1979.

Warnock, J.
Personal communication, August 1980.

Further steps in the flight from time-sharing

The Interlisp-D group.

Abstract

One of the goals of the Interlisp-D effort has been to provide Interlisp's programming support tools in a personal computing environment. This report outlines the current status of the Interlisp-D implementation, and describes some of the interactive programming tools that have recently been added to the system.

BACKGROUND

The Interlisp-D project was formed to develop a personal machine implementation of Interlisp for use as an environment for research in artificial intelligence and cognitive science [Burton *et al.*, 80b]. This note describes the principal developments since our last report almost a year ago [Burton *et al.*, 80a].

Principal characteristics of Interlisp-D

Interlisp-D is an implementation of the Interlisp programming environment [Teitelman & Masinter, 81] for the Dolphin and Dorado personal computers. Both the Dolphin and Dorado are microprogrammed personal computers, with 16-bit data paths and relatively large main memories (~1 megabyte) and virtual address spaces (4M-16M 16 bit words). Both machines have a medium sized local disk, Ethernet controller, a large raster scanned display and a standard Alto keyboard and "mouse" pointing device.

Both the internal structure of Interlisp-D and an account of its development are presented in [Burton *et al*, 80b]. Briefly, Interlisp-D uses a byte-coded instruction set, deep binding, CDR encoding (in a 32 bit CONS cell) and incremental, reference counted garbage collection. The use of deep binding, together with a complete implementation of spaghetti stacks, allows very rapid context switching for both system and user processes. Virtually all of the Interlisp-D system is written in Lisp. A relatively small amount of microcode implements the Interlisp-D instruction set and provides support for a small set of other performance critical operations. The at one time quite large Bcpl kernel has been all but completely absorbed into Lisp, for the reasons outlined in [Burton *et al*, 80b].

Interlisp-D is completely upward compatible with the widely used PDP-10 version. All the Interlisp system software documented in the Interlisp Reference Manual [Teitelman *et al.*, 78] runs under Interlisp-D, excepting only a few capabilities explicitly indicated in that manual as applicable only to Interlisp-10. The completeness of the implementation has been demonstrated by the fact that several very large, independently developed, application systems, such as the KLONE knowledge representation language [Brachman, 78], have been

---

A version of this paper is to appear in SIGART Newsletter, No. 77, August 1981.

brought up in Interlisp-D with little or no modification. Interlisp-D is in active use by researchers (other than its implementors) at both Xerox PARC and Stanford University and is now approaching the level of stability and reliability of Interlisp-10.

CURRENT PERFORMANCE

The performance engineering of a large Lisp system is distinctly non-trivial. We have invested considerable effort, including the development of several performance analysis tools, on the performance of Interlisp-D and, as a result, seen its performance improve by nearly a factor of five over the last year. Although relative performance estimates can be misleading, because of variation due to choice of benchmarks and compilation strategy, the overall performance of Interlisp-D on the Dolphin currently seems to be about twice that of Interlisp-10 on an otherwise unloaded PDP KA-10. Although this level of performance makes the Dolphin a comfortable personal working environment, we have identified a number of improvements which we anticipate will further improve execution speed by 20% to 100%.

MACHINE INDEPENDENCE

Another major thrust has been to reduce the dependencies on specific features of the present environment, so as to facilitate Interlisp-D's implementation on other hardware. Dependencies on the operating system have been removed by absorbing most of the higher (generally machine independent) facilities provided by the operating system into Lisp code. Gratuitous dependencies on attributes of the hardware, such as the 16-bit word size, have been removed and inherent ones isolated. In addition to an abstract desire for transportability, our sharing of code with other Interlisp implementation projects provides a on-going motivation for this effort.

EXTENDED FUNCTIONALITY

The principal innovations in Interlisp-D, with respect to previous implementations of Interlisp, involve the extensions required to allow the Interlisp user access to a personal machine computing environment.

Network facilities

While network access is a valuable facility in any computing environment, it is of particular importance to the user of a personal machine, as it is the means by which the shared resources of the community are accessed. Over the last year, Interlisp-D has incorporated both low level Ethernet access and a collection of various higher level protocols used to communicate with the printing and file servers in use at PARC. It is now straightforward to conduct *all* file operations directly with remote file servers. This both allows the sharing of common files (e.g., for multi-person projects, such as the construction of Interlisp-D itself), permits a user to move easily from one machine to another, and eliminates any constraints of local disk size. We have also begun to investigate the possibility of paging from a remote virtual memory elsewhere on the network. This would not only allow completely transparent relocation of a user's environment from one machine to another, but would open up a variety of interesting schemes for distributing a computation across a set of machines.

High level graphics facilities

Interlisp-D has always had a complete set of raster scan graphics operations (documented in [Burton, 80b]). More recent developments include a collection of higher level user graphics facilities, akin to those found in other personal computing environments. The most important of these is the Interlisp-D window package. This facility differs in spirit from most other window systems in that, rather than imposing an elaborate structure on programs that use it, it is a self consciously *minimal* collection of facilities which allow multiple programs to share the same display. Although some mechanism is necessary to adjudicate a harmonious sharing of the display, we feel that higher level display structuring conventions are still an open research question and therefore should not yet be incorporated into a mandatory system facility. The window package *does* provide both interactive and programatic constructs for creating, moving, reshaping, overlapping and destroying windows, in such a way that a program can be embedded in a window in a completely transparent (to that program) fashion. This allows existing programs to continue to be used without change, while providing a base for experimentation with more complex window semantics in the context of individual applications.

One such existing application is a display based, structural program editor. This editor, in contrast to the character orientation of most modern display based program editors, is the result of marrying display techniques (selection and command specification by pointing, incremental reprinting, *etc*) with the structure orientation of the existing Interlisp editor. Indeed, the two editors are interfaced so that the considerable symbolic editing power of the existing editor remains available under the display based one. Although our initial experience has been positive, the user interface is under continued revision as we gain further experience with this style of editing.

FUTURE PLANS

The area in which we anticipate most future development of Interlisp-D is the personal computing facilities, such as graphics and networking, and their integration into Interlisp's rich collection of programming support tools. While radical changes to the underlying language structures are made difficult by our desire to preserve exact Interlisp compatibility, we also expect some language extensions, including some form of object oriented procedure invocation.

One of the great strengths of Interlisp has been the many contributions made by its active, critical user community. We are hopeful that the recent commercial availability of Interlisp-D to other sites, and the consequent growth of its user community, will be a similar source of long term strength and, in the short term, significantly accelerate the pace with which Interlisp evolves away from its time-shared origins into a personal computing environment.

REFERENCES

Brachman, R. *et al.*
KLONE Reference Manual. BBN Report No. 3848, 1978.

Burton, R. *et al.*
Overview and status of DoradoLisp. *Proceedings of the 1980 Lisp Conference*, Stanford, 1980a.

Burton, R. *et al.*
Papers on Interlisp-D. Xerox PARC report, SSL-80-4, 1980b.

Teitelman, W. *et al.*
The Interlisp reference manual. Xerox PARC, 1978.

Teitelman, W. and Masinter, L.
The Interlisp programming environment. *IEEE Computer*, 14:4 April 1981, pp. 25-34.

The members of the Interlisp-D group are Beau Sheil, Bill van Melle, Alan Bell, Richard Burton, Ron Kaplan and Larry Masinter.