LYSP – A Lisp Dialect for Programming Small Applications

Paul R. Kosinski
IBM Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, New York 10598

Motivation

Some ten years ago, a small group of us[1] at IBM Research undertook to write a new kind of text editor, to run in the VM/370 environment, based on our experience with the experimental Business Definition System[2] (BDS) we had implemented previously. This new editor, called the Parametric Editor[3], or P-EDIT, was to be upwards compatible with an existing popular editor in use at that time, while providing the ability to edit multiple versions of a file simultaneously, and providing a general UNDO command which could be used to return to any previous state of the editing session, and which itself was UNDOable.

The multiple versions were to be effected by attaching to each line of text in the file a Boolean expression which determined whether that line was in the version set currently being edited. The versions set, in turn, was defined by a single Boolean expression called the mask. A file line was deemed to be in the current version set if and only if its Boolean expression was logically consistent with the mask. For example, if the mask was (DISPLAY = 3279 $\vee$ DISPLAY = 3278) $\wedge$ SYSTEM = CMS, then a line whose Boolean expression was SYSTEM = MVS would be excluded from the current version set, a line whose Boolean was SYSTEM = CMS would be strongly included in the version set (because the mask implies the Boolean), while one whose Boolean was DISPLAY = 3279 would be weakly included (because the mask doesn't imply the Boolean).

Our implementation experience with BDS suggested that Lisp was a fine language for developing complicated programs and the obvious language in which to program such things as Boolean expression manipulation. However, it was initially thought that text editors should be written in a more conventional language such as PL/I or Assembler (as the popular editor was). Furthermore, the overwhelming size of the Lisp system available apparently ruled out the possibility of programming P-EDIT in Lisp. Indeed, running Lisp required a larger virtual memory than most users were normally permitted.

After some months of very slow progress, it was decided that attempting to program the necessary Boolean expression manipulation (especially simplification) in PL/I was too difficult, and that it would be easier to implement a Lisp subset and use the existing Lisp code for Boolean expressions. It was further decided that adapting the existing popular editor, written in Assembler, to interface with a Boolean package written in Lisp, would be more difficult than rewriting the editor proper in a slightly larger Lisp subset.

Since P-EDIT was to be compatible, we did not wish to have its macros written in Lisp, unlike many Lisp based editors, but rather would use the EXEC 2 language[4], which many of the then (and now) existing editors use for their macros. (Indeed it is good design to have a single programming language for extending the capabilities of a wide range of application programs.)

We were thus faced with the task of defining and implementing a Lisp subset suitable for programming P-EDIT, but not necessarily for extending it via macros.

Application Directed Language Definition

In defining the Lisp subset (originally called DeciLisp, but now called Lysp) to be used to program P-EDIT, we took a very different approach from normal language design: we viewed the language as a tool rather than a standard. The language and its compiler were merely a means to generate executable application code, rather than the language being something sacrosanct to which the application program and compiler had to conform.

What this meant was that we would not include anything we did not feel we needed for P-EDIT, we would not balk at extending the language in idiosyncratic ways if that seemed helpful, and we would be willing go back and change the language definition if we ran into a serious problem in actually using it. Furthermore, the initial language definition was also based on what we found practical to compile and efficient to execute, rather than only on what we found 'elegant' or conventional.

The first decision we made was that, in order to save the immense amount of space taken by the compiler in full-blown Lisp systems, we would be willing to live with the old-fashioned compile, load, test and recompile mode of program development using a stand-alone compiler. This meant that Lysp would not contain functions such as COMPILE or LOAD, and new Lysp functions could not be created by the application.

We also decided to generate Assembler code as the output of the compiler, because programming in Lisp was considered to be weird at that time and we wanted to be able to show people code they code read, at least at the statement level. To aid in this, the compiler produces local comments along with the code, explaining what the instruction does in Lisp terms, with the original variable names. (The peep-hole optimizer even transforms the comments along with the code.)

We decided to forgo the possibility of separate compilation of Lysp subroutines because the format of output produced by the Assembler and accepted by the linker was not up to handling the long function and global variable names traditionally used in Lisp programs. (We were willing to write a compiler, but not an assembler and linker too!)

The next decision was that since the P-EDIT would not use macros written in Lysp, EVAL and the interpreter would not be part of the application's run-time environment (thus saving another large amount of space). However, since addresses of compiled subroutines would be needed anyway by the implementation, and adding a type code to the address to turn it into a proper Lysp datum was easy, it was simple to implement the APPLY operation and its relatives. Indeed there are quite a few cases where P-EDIT invokes one of several functions via a variable operator, for example:

```
( (ELT V I) X Y)
```

A related decision was not to support continuations (states) or closures (FUNARGs). We never had to use them in BDS, which was much more complicated than P-EDIT, and we figured (correctly) that we would not need them in P-EDIT. Leaving them out much simplified the run-time environment. We did, however, provide CATCH and THROW, as they were cheap to implement and are quite useful as an error handling mechanism; P-EDIT uses them extensively.

In Lysp, variables can either be lexically local or they can be global and dynamically re-bindable (cf. SPECIAL). Lexical variables are kept in the current stack frame, while global variables (and constants) are kept in a static area, which is actually a compiled vector of pointers. Re-binding of a global variable is done by saving its old value on the stack upon contour entry and restoring it upon exit (including a THROW-implied exit). Continuations or closures would require a more

complicated and less efficient implementation, whereas dynamically re-bindable variables are cheap to implement; P-EDIT uses quite a few of them.

Since there was no EVAL, we followed the practice of most other languages and did not preserve variable or function names in the compiled code. Even if a program contained the quoted identifier 'FOO, there would be no way for it to acquire the pointer to the compiled function FOO merely by using that identifier. To save space, there is no automatically generated run-time data structure holding the identifiers appearing in the program, much less any automatically generated mapping structure which relates identifiers to their values. Thus, the static vector is essentially the value cells without the rest of the symbol structure.

Following the lead of LISP/VM[5], we did not distinguish between ordinary values and function-values of names. Thus, the pointer to the compiled function FOO is kept in the global variable cell associated with the name FOO, rather than in a separate 'function cell' associated with the identifier-object FOO. (The latter would of course have been impossible, there not being any such data structure at run-time.) This was done so that function names could be treated just as ordinary global variable names, and implies that functions can be re-bound if their names appear appropriately in a LAMBDA-list, and they even can be 'renamed' by being the target of a SETQ. (Functions whose code is no longer pointed to are not garbage collected, however.) A LAMBDA expression appearing in operand position is treated as any other constant, except that the value of the constant is a pointer to the compiled code.

An unfortunate, but we think justifiable, example of implementation efficiency affecting the language is that builtin functions, such as CONS, do not have Lysp values (pointers) and hence cannot be APPLYed, stored into variables, passed as arguments, or tested. This is because they they are called in a much more efficient way than compiled functions (with arguments in registers rather than on the stack), or are even expanded in line. This is not much of a problem in practice, since such low-level functions are rarely useful as alternatives for a variable operator value. Furthermore, it is easy to write a compilable function, whose body consists solely of a call to the low-level builtin, to serve as an interface. (This has been done in the past, e.g. in LISP/VM.) For example, the form

```
(COMPLAMBDA CONSTRUCT (X Y) (CONS X Y))
```

causes the function CONSTRUCT to be compiled which exactly performs the builtin CONS operation, except that in addition CONSTRUCT can be APPLYed, the pointer to it stored, passed as an argument, and tested. The cost is that executing CONSTRUCT is perhaps 3 times as slow as CONS.

The functions that are builtin to Lysp, in the sense of always being there, whether or not they are used, are limited to those which it was felt had to be in the Assembler coded run-time environment (such as CONS and EQUAL), plus any that were so simple as always to be compiled inline (such as CAR and EQ). Many functions which are standard in most Lisps (such as MEMBER and INTERN) are not necessarily present, but are extracted by the compiler from a library of Lysp-coded functions if they are referred to by the program being compiled. This is safe because there is no EVAL, and thus there is no danger of calling a function 'out of the blue' by constructing its name. (One may force a library function to be included if it is to be stored into a variable yet its name appears nowhere in operator position of a Lisp expression and thus would not be automatically included.) By this means, much space is saved, yet all standard functions are available if compiled in. Although we have not added to the compiler's library any functions for which we did not forsee a need, they can be added at any time.

In order to increase run-time efficiency, few of the builtin or library functions that we provided perform type testing on their arguments. This decision has two aspects. First, most functions (such as CAR and PLUS) run unchecked, which can cause some debugging headaches, but certainly no

worse than if we had written in Assembler language. Update functions (such as RPLACA and STOREBYTE), which have a greater potential for causing serious problems, usually check the validity and range of their arguments. These functions tend to be longer anyway (and thus not inline), so the checking is relatively less onerous. The function call mechanism also can be run with checking (and usually is), so that calls on NIL, common when a function definition is omitted, will be trapped.

Second, low level functions which are generic in bigger Lisps were made data-type specific in Lysp. For example, ELT in LISP/VM serves to extract the I-th element of any data type for which indexing makes sense, including pointer vectors, character strings and lists. In Lysp, ELT is restricted to work on pointer vectors, and it assumes that its two arguments have the correct type. For indexing strings, FETCHBYTE or FETCHCHAR must be used, while NTH must be used for lists. This approach gives the run-time efficiency of declarations without their additional programming complexity or difficulty of implementation. (Remember that our goal was to program P-EDIT, not the world's best Lisp.) Yet one is not precluded from defining a generic ELEMENT function which works on all relevant data types, or even defining a set of functions in order to implement a more abstract data type in terms of lists and/or vectors.

Another decision we made, which has been made for a few other languages, was not to define any standard I/O functions for Lysp, except at the most primitive system-call level (and even those are extracted from the compiler's library on a strictly as-needed basis). In particular, P-EDIT would perform both file and terminal (screen) I/O, but in its own unique manner. Screen I/O especially was to be highly specialized to the editing application. (Later on, we added some simple higher level I/O functions to the library, but they were mainly for demonstration purposes.) This lack of automatically included I/O functions, more than almost anything else, results in modules generated by Lysp being much smaller than those generated by other 'high level' languages such as PL/I.

Since P-EDIT was to provide a general UNDO request, we decided to build into Lysp some undoable update functions. The operations we chose to have undoable variants of were SETELT (which updates a vector element), RPLACA and RPLACD, and SETQ of a global variable. These functions would record the old value of the field in a log record which was then consed onto a global log list. At the end of each P-EDIT request, the global log list would be associated with that request in the history list, and the log list variable reset to NIL. (These functions are similar to those provided in Interlisp, we discovered later.) This mechanism was was efficient, easy to implement and provided the basis for a surprisingly useful editing feature.

Data Types

The data types we decided on supporting in Lysp were exactly those that we felt we would need to implement P-EDIT. The ones supported were: integers which, when typed, could fit in the 32-bit machine register; character strings of arbitrary, but fixed length, to represent the lines of the file being edited; pointer vectors of arbitrary (but again fixed) length, to serve as the pointer blocks for the files lines, and as other control blocks; pairs (conses), to be formed into lists to represent the Boolean expressions; identifiers (symbols), to represent the Boolean variables; functions (pointers to compiled functions); and NIL, which was made a separate type so (1) it would not slow down testing for other types, (2) its type code, with a different data part, could serve other purposes (such as a CATCH/THROW marker on the stack), and (3) so NIL itself, which is permanently kept in a register, could serve as the base address of the Assembler coded, primitive functions like CONS.

We decided that we did not need any other types that are useful in full-blown Lisps, such as bignums, floating point numbers, streams, gensyms (we used distinctive identifiers), fixed point integer arrays, bit vectors and so forth.

We also decided to skimp on the use of data types, if that would increase efficiency. A prime example is that we backed off from representing characters as identifiers, but instead represented them as integers (in the range 0 through 255). We defined a FETCHBYTE builtin function, which fetches a character from the given position in a string as an integer, rather than use the FETCHCHAR function, which fetches an identifer representing that character, as we were used to in LISP/VM. The fact that the mapping from 8-bit byte to character identifer need not be done saves over 1000 bytes in the mapping table (pointer vector), plus over 2000 bytes for the character identifiers themselves! Program clarity does not suffer – there is a CHR macro which yields (at compile time) the integer representation of a character. For example, instead of writing

```
(EQ (FETCHCHAR S I) 'A)
```

one writes

```
(EQ (FETCHBYTE S I) (CHR A))
```

which does not require any additional storage, since such small integers are immediate data.

Macros and the Compiler

One of the features of Lisp that we had come to appreciate when programming BDS was the ability to write powerful macros to be evaluated at compile time. This gave us the ability to extend the basic Lisp language to an even higher level language, often in application specific ways. Since the DeciLisp compiler was written in the LISP/VM language and ran in that environment, we had access to the LISP/VM interpreter and hence its macro facility. Thus the DeciLisp language had the sort of extensibility we were used to, and this was used heavily in writing P-EDIT. In particular, macros were often used to generate efficient in-line code, using compile-time constant propagation (performed by the macro) from source which was written as a normal function application. That is, they were used to extend the compiler.

Although DeciLisp was a standalone compiler in the sense of not being present in the compiled application code, it still operated in the highly interactive environment of LISP/VM. Thus we followed the practice of making our source code files just a sequence of S-expressions to be evaluated by the LISP/VM interpreter. Many of these S-expressions, of course, invoked the compile functions to compile their bodies. But interspersed with these calls on the compiler could be macro definitions, generation of tables (such as P-EDIT's request mapping table), or just random computations of useful data.

One of the facilities that LISP/VM provides is the ability to redirect the interpreter's input stream to be from a file (via the EXF function). We use EXF to allow one source file to include another. Using the compiler's library, for example, is not implicit but explicit via EXF. In fact, since the library file was interpreted, the transitive closure part of library search is in the library file itself (via a big loop over all the compiler calls) rather than in the compiler! The compiler proper merely records the identifiers which appear in operator position (and are not builtin) and provides a COMPLAMBDAIFUSED function which compiles its body only if the given identifier has in fact been used.

When some of us began working on a new experimental operating system (EM-YMS), it was decided to write the compiler in its own language and bootstrap it so that it would be available on EM-YMS. (The 'y' in 'Lysp' comes from 'YMS'.) Bootstrapping involved more than merely rewriting the compiler, it also involved rewriting enough of the LISP/VM interpreter so that macros, EXF and source file computation could be carried over. Since compiled code does not retain function names, the interpreter we wrote had to have its own mapping mechanism and had also to

make sure that the identifier data structures (including the INTERN hash table) were present. Also, since many standard functions do not type-check their arguments, and since the interpreter could not assume correctness (since it was to be used for debugging, too), the interpreter is full of functions like CONSTRUCT (above) with added CONDs for type-checking.

The interpreter also contains the mapping table which maps the relevant identifiers (such as CONS) to their interpreting functions (such as CONSTRUCT). This mapping table is generated by a program executed at compile time which uses a hook in the compiler which maps an identifier to the global variable vector index for the slot that will hold its value at run-time. This hook was originally put in for P-EDIT, so that it could generate the table which maps request names to the Lysp-coded functions which implement them.

Data Representation

Since the EM-YMS operating system was designed to run using 32-bit addressing (the natural limit for the IBM 370), we decided that Lysp would allow full 32-bit addresses in its pointers, unlike DeciLisp, which only had 24-bit address parts (the 370 implementation limit). Going to full 32-bit addresses meant that LISP/VM's style of using the high-order byte of a word size datum as the type code would no longer work. Following the suggestion of JonL White[6]. we decided on using the low order few bits of a word as the type code. Since we didn't need very many different data types, we settled on a 3 bit type code. This meant that objects in storage would have to be aligned on 8-byte (doubleword) boundaries. If we had needed more type codes, we would have had to use coarser alignment and thus waste more storage per object on the average.

In order to make arithmetic as fast as possible, we used a type code of 000 for integers. This makes Lysp integers represented as machine integers multiplied by 8, with 28 bits of precision plus sign. With this representation, add, subtract and compare are implemented directly by the machine instructions without any masking or other adjustment needed. Multiply and divide are almost as simple, requiring only one shift in addition to the arithmetic instruction.

Since testing for pairs (or conversely, for atoms) is very common in Lisp, the pair type code was made 111 to take advantage of the machine's bit-test instruction, which reports 'all bits tested = 1' as a separate case. (Since NIL is a unique 32-bit datum, testing for it is done by 'compare'.) The other type codes were assigned to make the garbage collector's life a little easier. There are thus 3 ranges of type codes: the non-pointers integers, NIL and (from the garbage collectors point of view) functions; the pointers to objects not containing pointers, strings; and the pointers to objects containing pointers, vectors, identifiers and pairs.

Since a Lysp pointer's low order 3 bits are type code, they corrupt the address part of the pointer (the high order 29 bits, of which only 21 are used on most 370's, and 28 on XA models). Rather than paying the execution cost of removing the type bits whenever the pointer is dereferenced (e.g via CAR), a non-zero 'displacement' is used in the dereferencing machine instruction to cancel the numerical effect of the type code. This is another reason why type specific builtin functions (such as ELT) are desirable.

Integers and NIL are immediate data, and although functions require storage for the code, they have no real structure. Character strings, pairs, vectors (of pointers) and identifiers (which essentially are pairs consisting of the 'pname' and the property list) of course do have structure; they can even be updated. They are called 'stored objects', or just 'objects'.

Although it might seem that there would be 4 different storage representations for objects, there are in fact only 2: objects which contain pointers, and those which don't. The objects containing

pointers are distinguished by the type codes in the pointers to them. This makes the garbage collector's life still easier.

Each object has a 2 word header. The first word is used only by the garbage collector, while the second word contains the number of elements in the object, stored as a Lysp integer. (This makes operations such as STRLENGTH very fast – a single 'load' instruction.) The second word is redundant in the case of pair and identifier objects – they always contain 2 elements – but since object alignment is in doubleword units, a 3 word representation for pairs would have a wasted word anyway.

A further simplification of the garbage collector comes about because we made the stack uniform; only valid Lysp data are stored on the stack. For example, return addresses are not stored, since they may be too big to be stored as Lysp integers, but rather the function pointer of the calling function is stored, plus an integer which is the offset of the return point in that function. Also, the garbage collector does not see the stack as broken up into frames, but rather treats it as a linear sequence of Lysp values.

Storage Management and the Garbage Collector

Since one of our original goals was to compile code which would run in a virtual memory which was smaller than that needed by LISP/VM, we decided to use an in-place garbage collection algorithm, rather than one which copied to another heap area (as LISP/VM's does). For any given application, this allows Lysp to get by with about one-half the virtual memory that LISP/VM needs to hold data. (LISP/VM's pairs only occupy 2 words compared to Lysp's 4, but P-EDIT does not use many pairs compared to strings and vectors.)

The garbage collector operates in 4 principal phases. The garbage collector does not care, in any of its operation, which of the 3 types a pointer containing object is. It only cares whether the object contains pointers or not. To that end, one of the bits in the first header word of each object records the type class at all times.

The first phase uses the active portion of the stack as roots to visit all active objects and mark them as such. In order not to use any additonal storage, it uses a pointer reversing traversal algorithm. It uses the first header word in each pointer containing object to keep track of which pointer in that object it is currently following. It also stores the marking bit in that word.

The second phase scans heap storage linearly looking for active objects. Whenever an active object is found, its new address is stored into its first header word, preserving the mark and type-class bits, and the new-address register is advanced by the object size. In any case, the scan address register is advanced by the object size, computed from the second header word of the object and the type-class bit. (Remember that the second header word contains the number of elements in the object, not its size in bytes.)

The third phase again scans all active objects and replaces the address part of pointers in each pointer containing object by the new address for that object (obtained from its first header word).

The fourth phase scans all active objects and moves them to the location addressed by their first header word (clearing the marking bit in the process). Now all the active objects sit at one end of the heap area, making the rest of the area available for simple sequential allocation.

When we converted to EM-YMS, we decided that it would be much nicer if Lysp applications didn't grab all at once from the operating system all the storage they might possibly need. To accomplish that, we changed from a uniform, contiguous heap to a segmented one, where each seg-

ment is obtained from the operating system when the previous one becomes full, and all empty segments are returned to the operating system after garbage collection. The garbage collector algorithms remain basically the same, except that the linear scans hiccup when they cross segment boundaries.

In order still to be able to allocate arbitrarily large objects, while not having to obtain a segment for each object, the storage allocator allocates 'small' objects sequentially from standard size segments, while 'large' objects are given segments of their own. Phase 2 of the garbage collector only scans the standard size segments, so only they are compacted, and 'large' objects do not get moved by phase 4.

The garbage collection mechanism described here is a 'batch' algorithm, and it is annoying sometimes while editing to have a lengthy garbage collection interrupt ones train of thought. We had considered using an incremental algorithm, but they seem to require that any pointer fetch test that pointer to see if it points to an object which has been 'forwarded'[7]. This test, on the 370, would slow down pointer fetching by at least a factor of 4. Therefore we have stuck to a batch algorithm.

Program Size and Performance

P-EDIT, as compiled by the older DeciLisp, yields an executable module of about 188 kilobytes (KB). This includes the kernel of builtin functions (including the storage allocators and the garbage collector). (The P-EDIT which is in use, since it only runs on CMS, still has not been fully converted to Lysp.)

PEDIT is a variant of P-EDIT which does not have the parametric (multiple version) facility. The Lysp compiled EM-YMS version of PEDIT, which does not contain the kernel, occupies 89 KB, while the DeciLisp compiled CMS version, which is older and lacks a number of features, but contains the kernel, occupies 76 KB.

The Lysp kernel has been separated out from the compiled Lysp code for the EM-YMS environment. It is re-entrant and can be shared by all applications written in Lysp. It occupies about 9 KB, of which about 1200 bytes comprise the low-level storage allocator, 1900 bytes the garbage collector, 800 bytes data and initialization and the rest the builtin Lysp functions. The CMS version of the Lysp kernel (which cannot easily be shared) is also about 9 KB. They are assembled from about 6100 lines of Assembler source code (including comments), of which a little more than 200 (all in a separate file) are operating system specific.

The general Lysp library totals about 1300 lines, the YMS specific library about 1150 lines of Lysp code. (These lines are as if prettyprinted 80 columns wide.)

The Lysp compiler and interpreter give rise to an executable module of 182 KB (not including the kernel, which is shared). This is obtained by compiling about 6600 lines of Lysp source code, plus the library.

Other applications that have been written since are, a file comparison program (11 KB + shared kernel, 650 lines + library), a very simple relational data base looker-upper (5.3 KB + shared kernel, 300 lines + library), a SNOBOL style pattern matcher (10.5 KB + shared kernel, 700 lines + library), and a reduction language interpreter (58 KB + shared kernel, 1850 lines + library).

Timing tests run several years ago indicate that DeciLisp code runs 4 times faster than standard LISP/VM code on a list processing example (Boolean expression manipulation), and about 10 times faster on an arithmetic example (prime number generation). Lysp compiled code should be in about the same ratio. These figures are for LISP/VM's type-checked functions. The faster 'Q'

functions, which are (like Lysp's) unchecked, should make the first case run almost as fast as in DeciLisp or Lysp. But the more machine-like number representation that Lysp uses would make it maintain a good factor of 2 in the arithmetic example.

Lysp's garbage collector is quite fast, when not subjected to paging delays. Recent measurements indicate that on a 370 model 3081K, a full heap can be collected at a rate of more than 4 megabytes (MB) per second, where full means a PEDIT session with 71500 lines of file being edited occupying 8 MB, so that essentially no garbage is collected in the less than 2 CPU seconds spent. When the file is dropped from the edit session and the garbage collector run again, it can compact the storage down to 175 KB in less than 100 CPU milliseconds.

The real test of performance, of course, comes in day-to-day use. P-EDIT and PEDIT have been in daily use for over 5 years by a total of about a dozen people, who use one or the other of them as their principal editor. On VM on a 3081, the performance of either of them is hard to distinguish from the performance of the standard XEDIT editor, which was written in Assembler language. The only time the Lisp origins become unpleasantly apparent is when a garbage collection occurs, and there is no garbage to collect. We therefore consider the approach to have been quite successful.

Acknowledgements

The author wishes to thank Cyril Alberga, Walt Daniels, Michel Hack, Vincent Kruskal, Chris Stephenson, and JonL White for many helpful discussions.

Notes and References

1. The group consisted of Walt Daniels, Vincent Kruskal, Peter Sheridan and the author.

2. A Very High Level Programming Language for Data Processing Applications, Michael Hammer, W. Gerry Howe, Vincent J. Kruskal, Irving Wladawsky; Communications of the ACM, November 1977.

3. Managing Multi-Version Programs with an Editor, Vincent Kruskal; IBM Journal of Research and Development, January 1984.

4. EXEC 2 Reference Manual, IBM SC24-5219.

5. LISP/VM Reference Manual, IBM SH20-6477.

6. Circa 1979-1980.

7. A reviewer has pointed out that neither 'generation scavenging' nor reference counting have this problem.