SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 10, 489–500 (1980)

A Portable Storage Management System for the Icon Programming Language*

DAVID R. HANSON
Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, U.S.A.

SUMMARY

Icon is a new programming language designed primarily for non-numerical applications. Its roots are in SNOBOL4 and SL5; as in those languages, execution-time flexibility is an important attribute of Icon, although some aspects of programs are bound at compile time to improve efficiency. Icon, which is implemented in Ratfor, is also intended to be portable and suitable for 16-bit computers. The storage management system in Icon is designed to meet the goals of portability, flexibility and efficiency. This is accomplished by subdividing the storage management system into a set of type-specific storage management subsystems. This paper describes the implementation of these subsystems, their interaction, and their performance.

KEY WORDS Storage management Portability Icon SNOBOL4 Ratfor Fortran Garbage collection Program measurement Modularity

INTRODUCTION

Icon[1, 2] is a programming language designed for non-numeric applications with an emphasis on string and structure processing. Icon derives much of its philosophical basis from SNOBOL4[3] and SL5.[4] For example, Icon supports numerous built-in datatypes and implicit conversions at execution time.

Icon is intended to be practical for production work. As such, it lacks some of the exotic features of SNOBOL4 and SL5; in order to provide greater efficiency in the most frequently used operations, Icon restricts execution-time flexibility. In this sense, Icon follows the more traditional method of binding many language operations at compile time, attempting to strike a balance between flexibility and ease of programming and execution-time efficiency.

Icon is also intended to be portable and suitable for computers with 16-bit words. Unlike SNOBOL4 and SL5, where abstract machine modelling[5] was used to achieve portability, Icon is implemented in Ratfor,[6] a preprocessor for Fortran. For the most part, the portability of Icon is based on the ubiquity of Fortran; its efficiency relies on the existence of good Fortran compilers; and the understandability of the code itself depends on the readability of Ratfor.

The usually conflicting requirements of portability, flexibility and efficiency have a significant impact on the design of the system in general and on the storage management system in particular. This paper describes the storage management system, which was designed to accommodate these requirements and their implied

* This work was supported by the National Science Foundation under grants MCS75–01307, MCS75–21757 and MCS78–02545.

0038–0644/80/0610–0489$01.00
© 1980 by John Wiley & Sons, Ltd.
Received 6 June 1979

limitations. The performance of the storage management system has a substantial impact on the language.[7, 8] The Icon storage management system is designed to reduce this impact to an acceptable level. This is accomplished by 'modularizing' the management of storage in terms of different types of data, such as integers and strings. This permits the storage for each type of data to be managed by a scheme that is most appropriate for that type. In addition, each type of data is managed independently in order to minimize the impact of one storage management technique on the performance of another. Thus, the storage management system in Icon is essentially a set of small, semi-independent, type-specific storage management subsystems.

Although the implementation language Ratfor (and hence Fortran) aids in developing a portable system, it provides little assistance in data structuring. Fortran has such poor data typing facilities that it is essentially 'typeless'. As a result, the storage management system must deal with data structures that are represented by appropriate combinations of integers. In addition, there is no (machine-independent) mechanism for referring to fields within a word, which might contain type codes or flag bits, for example. Even if such a mechanism existed, it would be of little use in light of the requirement to run on computers with a small word size.

The following sections describe the layout of storage and the representation of various data types in Icon, allocation of storage for each type, reclamation of inaccessible storage for reuse and the performance of the storage management subsystem.

STORAGE LAYOUT AND DATA REPRESENTATION

Storage is represented by a large Fortran array of one dimension. This array is divided into five regions roughly corresponding to the division of data types into classes. These regions, depicted in Figure 1, are the string region, the string qualifier region, the integer region, the heap and the stack.

All source language values have a compact, uniform representation: a pointer into one of the storage regions. A pointer is simply an integer index into the memory array. While this representation is compact, there is no room for type codes. The locations of the storage regions provide a means of determining the type from the value of the pointer. For example, if a pointer points within the bounds specified by *intbas* and *hepbas* (see Figure 1), it points to an integer.

The structure of each region is determined by the kinds of data stored in that region. This technique is used to save space. Since the structure of each region is tailored for a specific kind of data, extra space for extraneous storage management information, such as block headers, can be eliminated. For example, there is no space required to store integers beyond that required for the integers themselves.

The integer region

Since positive integers are indistinguishable from pointers, integers are represented by pointers into the integer region. Each word in the region contains one integer. For example, the representation of the integer -5 is shown in Figure 2. This representation is less efficient than simply representing integers by their values, but is necessary in order to accommodate computers with 16-bit words. It is possible to use a flag bit to indicate a pointer, but on a 16-bit computer this reduces the magnitude of integers to 16,383 and limits the values of pointers to 15 bits.

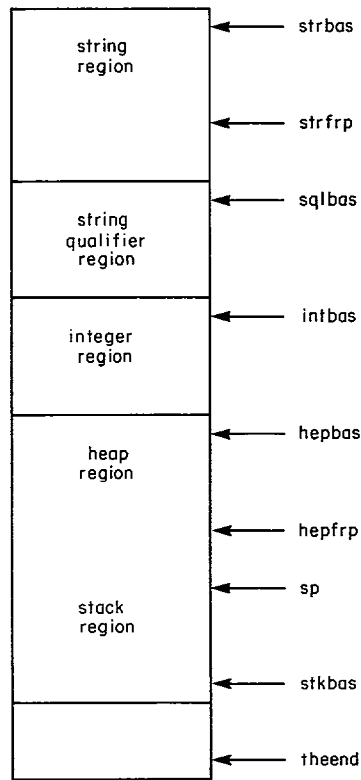


Figure 1. Storage regions

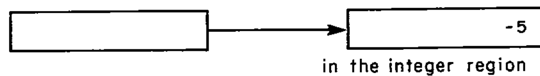


Figure 2. Representation of the integer −5

This representation does permit integers that occur frequently to be permanently allocated. Whenever a computation results in one of these integers, the appropriate pointer is returned. In the current version of Icon, the range of integers that are handled in this fashion can be specified prior to execution; the default range is −1 to 1.

The string and string qualifier regions

Strings are represented by a pointer to a two-word qualifier in the string qualifier region. A qualifier contains the length of the string in characters and the character offset to the first character in the string. The offset is the character offset from the base of the array containing the five storage regions. This representation facilitates efficient implementation of many common string operations such as substring formation, concatenation, and trimming; see References 9 and 10 for more information. Figure 3 illustrates the representation of the strings 'hippopotamus' and 'pot'.

While this representation is machine independent, it does require that the string region be located at the low end of the memory array in order to avoid very large offsets that might not fit in a single word.

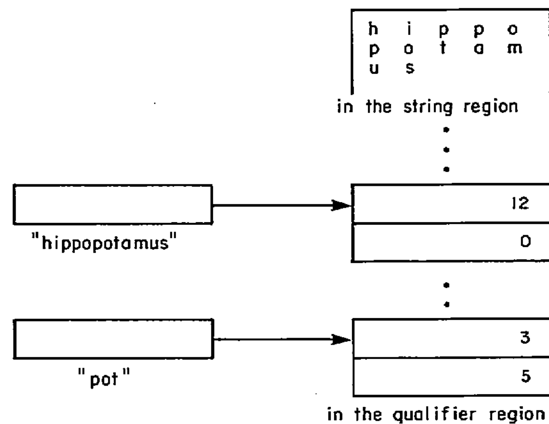


Figure 3. Representation of 'hippopotamus' and 'pot'

As for frequently occurring integers, qualifiers for frequently occurring strings can be permanently allocated. In the current version, only the null string is permanently allocated; it has a zero length and offset.

The heap

The heap provides a region in which objects of different sizes can be allocated. Storage in the heap is allocated in blocks. All blocks have a header word, which contains a block code that uniquely identifies the type of data stored in the block. All pointers pointing into the heap point to block headers. The block code serves as a type code. The layout of the remainder of a block depends on the kind of data stored in the block.

There are four basic block layouts. The most general form, shown in Figure 4, is a varying size block that may contain pointers to other blocks. This kind of block is used to house lists, tables and records, for example. The size of the block, including the header, is contained in the size field. The back reference field is used by the garbage collector in marking nested blocks; this is described in the next section.

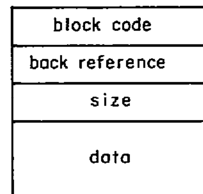


Figure 4. Layout of varying size blocks containing pointers

To save space, some objects are represented by blocks with a simpler layout. For example, if the object cannot contain pointers, the back reference field is unnecessary. If the block size is the same for all objects of that type, the size can be deduced from the block code.

For example, character sets are represented by fixed-size blocks as depicted in Figure 5. The block size is determined by table lookup using the block code as an index. The layout shown in Figure 5 applies only to blocks that do not contain pointers to other

blocks. Note that pointers to integers or strings are permitted in this layout, however. (This capability is not used in the current version of Icon). Figure 6 shows the layout for fixed-size blocks that may contain pointers to other blocks. The appearance of a back reference field is determined by the block code.

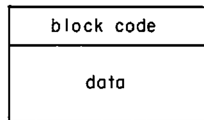


Figure 5. Layout of fixed-size blocks

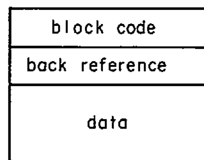


Figure 6. Layout of fixed-size blocks containing pointers

Objects whose size may vary, but that do not contain pointers, are housed in blocks whose layout is depicted in Figure 7. The value of the block code indicates that the size is contained within the block itself.

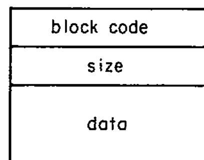


Figure 7. Layout of varying size blocks

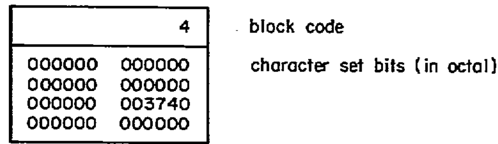
Note that the header field occurs in all blocks and is in a fixed position (the first word). Although back reference fields may be absent, depending on the block code, they are also in a fixed position (the second word). Size fields also may be absent, but may appear in the second or third word. This arrangement permits the garbage collector to find the back reference field in the same place regardless of the block code, which facilitates the processing of those blocks that contain pointers to other blocks.

Figure 8 depicts the representation of some data objects that are stored in the heap.

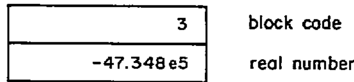
The stack

The stack is composed of single word values and grows from *stkbas* backwards towards *hepfrp* (see Figure 1). The top of the stack is indicated by the pointer *sp*.

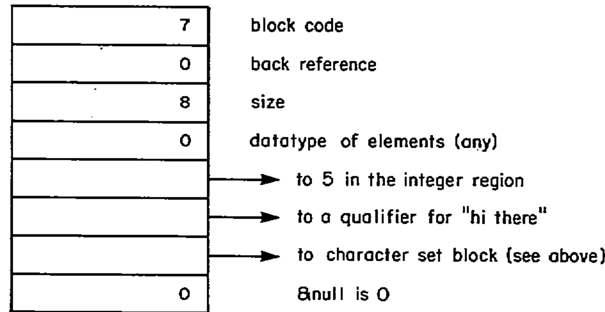
The stack may contain pointers and is examined by the garbage collector during reclamation. There are situations, however, when it is necessary to push arbitrary values onto the stack. For example, it may be necessary to push a Fortran integer onto the stack. Since positive integers are indistinguishable from pointers, the following



The Character Set "abcdef" (36-bit words).



The Real Number -47.348e5.



The List <5, "hi there", cset ("abcdef"), &null>

Figure 8. Examples of objects stored in the heap

convention is used to warn the garbage collector of non-pointer data on the stack. The appearance of a -1 on the stack indicates the presence of such data. The next word (towards *sp*) contains a count of the number of subsequent words containing non-pointer data. As an example, Figure 9 shows the stack configuration after pushing the Fortran integers 52 and -104. This convention is not necessary for integers less than -2 since they cannot be pointers.

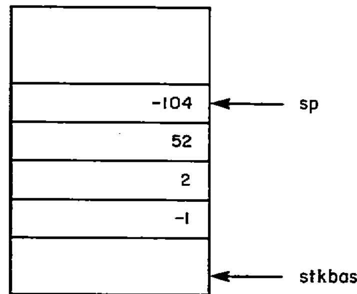


Figure 9. Stack configuration after pushing 52 and -104

ALLOCATION

There is a separate allocation strategy for each storage region. The scheme used for each region is designed for efficient allocation and to accommodate reclamation in that region.

Allocation in the qualifier and integer regions is in fixed-size units. For each region, a linked list of available units is maintained. Allocation consists of simply returning the first unit on the free list and advancing the free list pointer.

Allocation of any of the permanently allocated integers results in the return of the appropriate pointer. Similar remarks do not apply to the allocation of a string qualifier with a zero-length field, however, since it is often necessary to allocate a qualifier with zero length and compute the appropriate length during subsequent processing.

Allocation in the string and heap regions is accomplished by incrementing the appropriate free space pointer by the amount of the request. Space in the string region is allocated in characters; space in the heap is allocated in words. An allocation request in the heap is always accompanied by a block code, and the allocation routine constructs the appropriate block header. Thus, knowledge of the specific header format layout, which may include the size and back reference fields, is confined to the storage management routines.

Stack words are allocated by decrementing *sp*. Note that the amount of available stack space varies since the heap and stack grow towards each other.

RECLAMATION

Reclamation of storage in each region consists of identifying the accessible data in that region, and restructuring the region so that the space occupied by inaccessible data is made available for reuse. Allocation in one region does not affect the other regions. This is not the case for reclamation. As described below, reclamation in one region may trigger reclamation in another region.

A determination of the accessible data in any region is begun with an examination of the contents of a set of locations that may contain pointers. These locations are called *tended* locations.[11] In Icon, the set of tended locations consists of the stack and about two dozen specific locations in a Fortran labelled common block. For reclamation in the integer and qualifier regions, the entire heap is also considered tended.

The integer region

Accessible integers are determined by examining the tended areas including the heap. This examination, which consists of a single pass, is called a sweep. Prior to sweeping for pointers into the integer region, enough zero words are pushed onto the stack to form a bit map of the integer region. (The bit operations required are the same as those used for character sets and are implemented by three simple assembly language functions.) Since the bit map contains non-pointer data, it is preceded by -1 and a count as described above and illustrated in Figure 9.

For each pointer into the integer region encountered during the sweep, the corresponding bit in the map is set to 1. The bits corresponding to the permanently allocated integers are also set to 1. After sweeping, the bit map is used to construct a new free list and the map is then discarded.

Note that the sweep may mark integers pointed to from inaccessible heap blocks as accessible. This can be avoided by reclaiming space in the heap before sweeping for integer pointers. Heap reclamation involves a full compactifying garbage collection, which is time consuming. Even the marking phase of that process is usually significantly more expensive than scanning all heap blocks. In order to keep reclamation in any region as inexpensive as possible, heap reclamation is never performed when reclaiming space in the other regions.

Each reclamation routine has an argument that indicates the amount of free space desired. If reclamation fails to produce the desired amount, the region is expanded if possible. Expansion of the integer region requires adjustment of all pointers that point into the heap (see Figure 1). This adjustment is accomplished by another variation of sweeping. (Sweeps are performed by a single routine that is parameterized to handle the specific reason for sweeping).

The string qualifier region

Reclamation in the qualifier region is similar to that in the integer region, but temporary stack space is not required. The bit map is required for the integer region because there is no way to 'mark' an integer as accessible without corrupting its value. For qualifiers, however, the location field is modified temporarily to indicate accessibility without loss of information.

Pointers into the qualifier region are located by a sweep. For each pointer, the location field is set to

$$-(location+1)$$

A linear pass is then made over the qualifier region setting the location fields of accessible qualifiers back to their original values and adding inaccessible qualifiers to the free list.

The string region

Determining the accessible data in the string region is simpler than for the other regions because all pointers into the string region are in the location fields of qualifiers. A qualifier on the free list has its length field set to -1 (the linked list is formed using the location fields).

A linear pass is made over the qualifier region, pushing a pointer to each active qualifier onto the stack. The pointers on the stack are then sorted in ascending order of the values in the location fields. The sorted list is scanned and accessible strings are compacted into the lower part of the string region. The location fields of the qualifiers are adjusted during this processing to reflect the result of compaction. This technique handles 'overlapping' strings such as those depicted in Figure 3. Details of the algorithm are given in References 10 and 12.

The string allocation routine always calls the qualifier reclamation routine prior to requesting reclamation in the string region. Icon has an extensive repertoire of string manipulation operations, and it is quite common for many intermediate strings to be created during program execution. Thus, many qualifiers and the strings they point to may be inaccessible. Reclaiming this space reduces the chances that the string region will have to be expanded.

The heap

Heap reclamation is the most complex of the reclamation schemes. It is a general garbage collection scheme involving marking and compaction. The algorithm is given in detail in Reference 11.

Briefly, heap reclamation consists of three phases. The first phase consists of marking all blocks accessible from the tended areas. This phase requires a marking algorithm capable of handling circular references. Pointers to heap blocks are adjusted to their final value during the second phase. In the third and final phase, accessible blocks are compacted into the lower part of the heap. If necessary, the heap is expanded by moving the stack.

Heap reclamation was the only reclamation algorithm whose implementation was significantly affected by the use of Fortran as the implementation language. The marking algorithm used in the first phase constructs a linked list of the pointers to a block. This list begins at the type code field of each block and is constructed using the pointers themselves. The problem is that pointers are represented by indices into the memory array and the tended variables in the Fortran common cannot be easily represented by such indices.

In order to address the tended variables, their values are copied into reserved locations at the base of the stack, which is in the memory array. The marking phase uses these copies of the tended variables when constructing the linked lists of pointers. Upon completion, the updated values are copied back into the common area.

An alternative approach would be to equivalence the tended variables with a portion of the memory array, such as the first two dozen locations. The memory array is kept in unlabelled common to accommodate those systems that permit expansion of unlabelled common. Thus, unlabelled common is used only for the memory array. In addition, the tended data is in a labelled common devoted to that purpose for modularity reasons.

The stack

The stack region never requires reclamation, but may require expansion. If space in the stack region is exhausted, heap reclamation is performed in an attempt to lower *hepfrp* enough to satisfy the expansion request. If this is unsuccessful, the stack is moved up if possible. Note that movement of the stack only requires the adjustment of *sp* and *stkbas;* no sweep is necessary.

PERFORMANCE

The reclamation strategies described above are tuned for specific regions and perform well when only those regions are considered. There are, however, situations in which the reclamation routines interact in ways that degrade performance. For example, reclamation in one region may require movement of the regions that follow it. Reclamation in the integer and qualifier regions requires allocation on the stack. The effect of region movement and stack allocation is to make reclamation in the worst case substantially more costly than in the average case.

For example, in the best (and hopefully average) case, reclamation in the integer region requires a single sweep. In the worst case, however, it may require

1. Heap reclamation;
2. Movement of the stack;

3. Sweep for integer pointers;
4. Sweep for expansion of the integer region; and
5. Movement of the heap and stack.

Reclamation in the qualifier region usually consists of a sweep for pointers into the region followed by a linear pass over all qualifiers. In the worst case, qualifier reclamation requires

1. Sweep for qualifier pointers;
2. A linear pass over the qualifier region;
3. Sweep for expansion of the qualifier region; and
4. Movement of the integer region, heap, and stack.

String region reclamation requires stack allocation. In addition, a qualifier reclamation is always performed. Worst case processing for string space reclamation involves

1. Heap reclamation;
2. Movement of the stack;
3. Sweep for qualifier pointers;
4. Linear pass over the qualifier region;
5. Linear pass over the qualifier region for accessible qualifiers;
6. Sorting the pointers to accessible qualifiers; and
7. Compaction of accessible strings and adjustment of qualifier location fields.

When qualifier reclamation is performed during string region reclamation, the qualifier region is never expanded.

Following the advice given in Reference 7, the storage management system was one of the first parts of Icon to be implemented and was in continual use during the rest of the development. No attempt was made to 'tune' the initial implementation since there was no empirical evidence to support any specific improvements. As the initial version of Icon was nearing completion, instrumentation was added to the storage management system in order to monitor its performance (other instrumentation was also added to monitor other aspects of the system). The instrumentation was designed so that it was easy to obtain statistics without having to recompile programs. This encouraged spontaneous measurement of programs that occasionally exhibited poor performance.

Measurements indicated that region expansion typically consumed approximately half of the execution time of reclamation. Expansion is particularly expensive for the string and qualifier regions since it involves the movement of more data. Likewise, expansion of the integer region, heap, and stack is relatively less expensive because there is less data to move. Thus, tuning was directed towards finding heuristics that reduced the number of expansions.

This was accomplished by associating a 'dynamic breathing room' with each region.[7] Since expansion time is proportional to the amount of data moved instead of the amount of movement, it is worthwhile to expand a region more than enough to satisfy the pending allocation request. This excess—the breathing room—represents an estimate of future allocation requests.

A first-order approximation to the breathing room is computed using the difference between the current size of the accessible portion of a region and the size of that portion at the last expansion. This difference represents the 'permanent' growth of the region and is typically large during the transient part of program execution. It is small when the program reaches a steady state in which the size of the accessible portion remains essentially constant.

Specifically, the breathing room used for expansion i is given by

$$\frac{(S_i - S_{i-1})}{S_{i-1}} S_i$$

where S_i is the size of the accessible portion at expansion i, and the fraction $(S_i - S_{i-1})/S_{i-1}$ is constrained to be within the interval [0, 1].

The breathing room heuristic resulted in a drastic decrease in the number of expansions for most programs. For example, for a particular run of a word frequency program, the number of expansions of the string region was reduced from 8 to 1; this reduction is typical. The number of reclamations is also reduced slightly because a region is larger. The number of reclamations was reduced from 10 to 8 in the word frequency program, for instance.

In some cases, however, this heuristic is over-enthusiastic, causing a region to become much larger than is necessary. This can be controlled somewhat by placing bounds on the fraction $(S_i - S_{i-1})/S_{i-1}$ that are 'tighter' than [0, 1]. Lowering the upper bound dampens the effect of the heuristic; raising the lower bound is a means of providing a default breathing room in anticipation of future expansion after a period of steady state operation.

Measurements also showed that, for some programs, an excessive amount of time was spent in reclamation and expansion of the integer region. All of these programs made heavy use of integers. The dynamic breathing room heuristic reduced the number of reclamations and expansions in all of these programs. Specifying a larger range of permanently allocated integers helped some of these programs, but had little effect on other programs, such as an arbitrary precision arithmetic package.

For this latter class of programs, there is little that can be done. They represent a class for which the allocation of integers from a separate region is simply a bad design decision. Using two-word values in order to avoid integer allocation at all would be a better approach in this case. This state of affairs seems to be typical of languages that rely on dynamic storage management schemes; there are programs that will defeat the best intentions of any given scheme.

CONCLUSIONS

The Icon storage management system has been designed to balance the frequently conflicting goals of portability, flexibility, and efficiency. Modularity is the basic technique used to achieve this balance.

The modularity of the system has several beneficial consequences. Tailoring the layout of each region to the kind of data stored in that region helps save space. Second, allocation and reclamation are simplified since they are performed on a per-region basis. This saves time because each region is smaller than the total available memory and the algorithms used are designed for specific regions. Finally, the code itself is modularized along the same lines. This results in a number of small, semi-independent routines instead of several large, monolithic routines.

The modularity of the system does have its drawbacks, however. As with any system having many components, the number and complexity of the interactions among the components can cause problems. The components in the Icon storage management system are designed to minimize the amount of interaction. For the most part, there is no interaction among the allocation routines except that they may call a reclamation

routine. As described above, however, interaction among the reclamation routines can result in poor performance. The present design does not permit such interactions to be eliminated, but empirical measurements have led to heuristics that reduce the frequency with which these interactions occur and lessen their effect when they do occur.

ACKNOWLEDGEMENTS

Icon was designed by Tim Korb, Ralph Griswold and the author. Walt Hansen made a number of valuable contributions to the design. Icon was implemented by Tim Korb, Walt hansen and the author. The above-mentioned persons have also made valuable suggestions concerning the presentation of this paper.

REFERENCES

1. R. E. Griswold, D. R. Hanson and J. T. Korb, 'The icon programming language: an overview', *SIGPLAN Notices*, **14**, 18–31 (1979).
2. R. E. Griswold and D. R. Hanson, *Reference manual for the Icon Programming Language*, Tech. Rep. TR79-1, Department of Computer Science, University of Arizona, Tucson, 1979.
3. R. E. Griswold, J. F. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language*, 2nd edn., Prentice–Hall, Englewood Cliffs, N.J., 1971.
4. R. E. Griswold and D. R. Hanson, 'An overview of SL5', *SIGPLAN Notices*, **12**, 40–50 (1977).
5. M. C. Newey, P. C. Poole and W. M. Waite, 'Abstract machine modelling to produce portable software—A review and evaluation', *Software—Practice and Experience*, **2**, 107–136 (1972).
6. B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison–Wesley, Reading, P A., 1976.
7. G. D. Ripley, R. E. Griswold and D. R. Hanson, 'Performance of storage management in an implementation of SNOBOL4', *IEEE Trans. Software Engineering*, **SE-4**, 130–137 (1978).
8. D. W. Green and C. C. Green, 'An empirical study of list structures in Lisp', *Comm. ACM*, **20**, 78–86 (1977).
9. D. R. Hanson, 'A simple technique for representing strings in Fortran IV', *Comm. ACM*, **17**, 646–647 (1974).
10. D. R. Hanson, *The Manipulation of Varying-Length String Data in Fortran IV*, Tech. Rep., Department of Computer Science, University of Arizona, Tucson, 1975.
11. D. R. Hanson, 'Storage management for an implementation of SNOBOL4', *Software—Practice and Experience*, **7**, 179–192 (1977).
12. W. M. McKeeman and J. J. Horning, *A Compiler Generator*, Prentice–Hall, Englewood Cliffs, N.J., 1970, Section 8.5-6.