Supplementary Information for the Implementation of Version 8 of Icon

Ralph E. Griswold

Department of Computer Science, The University of Arizona

1. Introduction

The Icon programming language [1] is fairly stable now, although refinements and new features are added occasionally [2]. The implementation of Icon, on the other hand, is still changing constantly. Besides changes made to support new features of the language, changes are made to correct errors, improve performance, increase portability, accommodate the requirements of new compilers and operating systems, improve readability of the source code, and so on.

The book on the implementation of Icon [3] is based on Version 6 of Icon. When the book was written, Version 6 was fairly new. The book is a snapshot of the implementation at a particular time during the final phases of the development of Version 6.2. In fact, this snapshot itself is slightly out of focus, since the implementation was changing even as the book was completed.

While the fundamental aspects of the implementation have not changed, there have been many minor changes, and in some respects the source code now differs noticeably from what appears in some places in the book.

This report is designed to be used in conjunction with the implementation book and source-code listings of Icon. Section 2 describes the most significant differences between what appears in the book and Version 8 of Icon. A list of corrections to known errors in the book appears in Section 3.

2. Recent Changes to the Implementation

2.1 Maintenance of Scanning Environments [Section 9.6, pp. 158-162]

String scanning saves, sets, and restores the scanning environment, which consists of the values of &subject and &pos. Two problems with nested string scanning have been fixed. The first problem occurs when a suspended scanning operation is resumed after the scanning environment has been changed in the outer scan. Consider the following case:

```
"abcde" ? {
   ("xyz" ? &null) & (&pos := 3) & &fail
   write(&pos)
   }
```

The change to &pos occurs in the outer scan, so 3 should be printed. Formerly, 1 was printed. This was because the inner scanning expression restored the outer scanning environment it found when it was invoked and not the one it found when it was resumed.

The second problem occurs when scanning is prematurely terminated by break, next, return, fail, or suspend. Consider the following case:

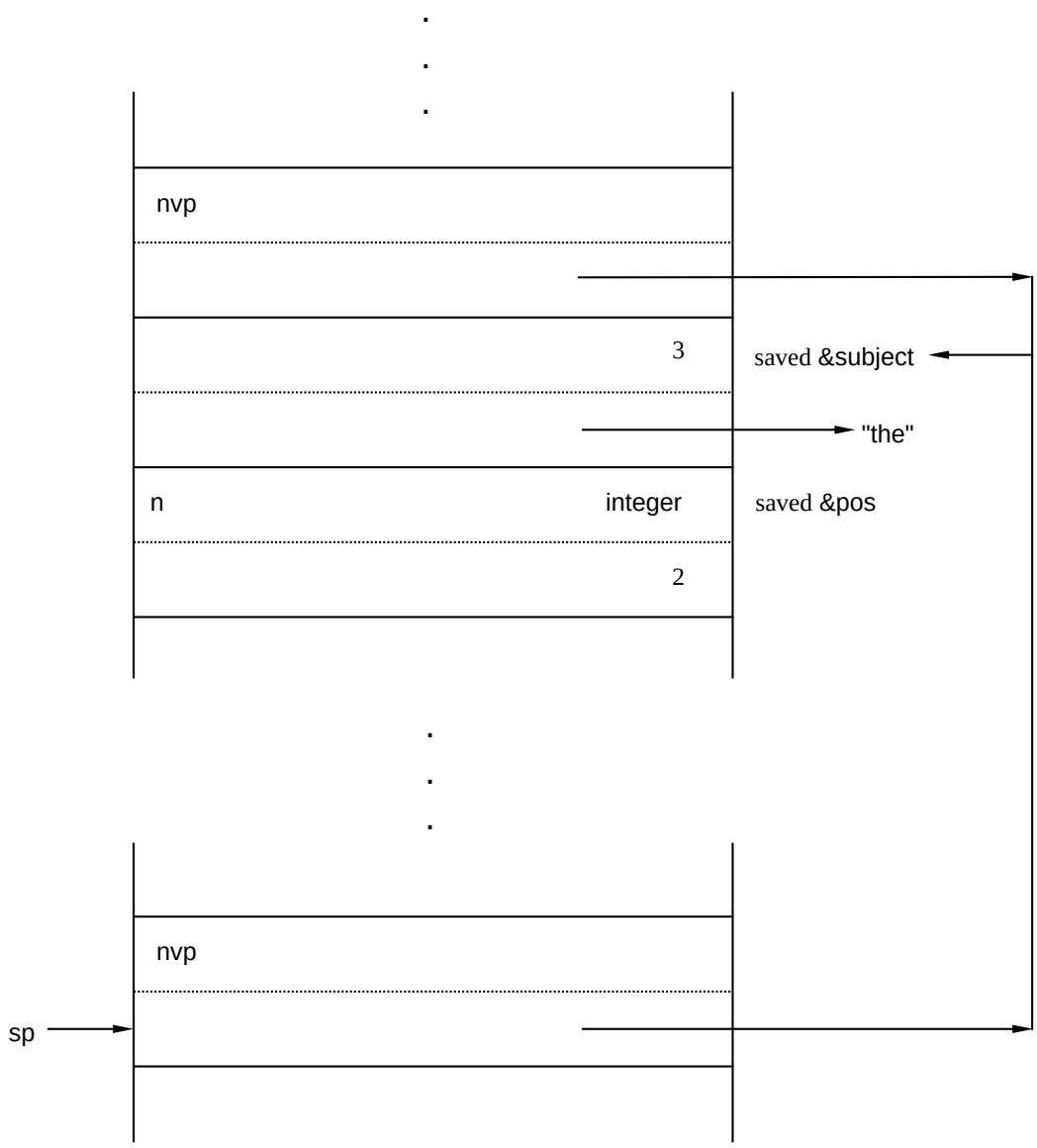
```
"abcde" ? {
   p()
   write(&subject)
   }

procedure p()
   "xyz" ? return
end
```

Logically this code should write "abcde". Formerly it wrote "xyz". This was because return did not restore the outer scanning environment when it caused the scanning expression to be exited.

Implementation

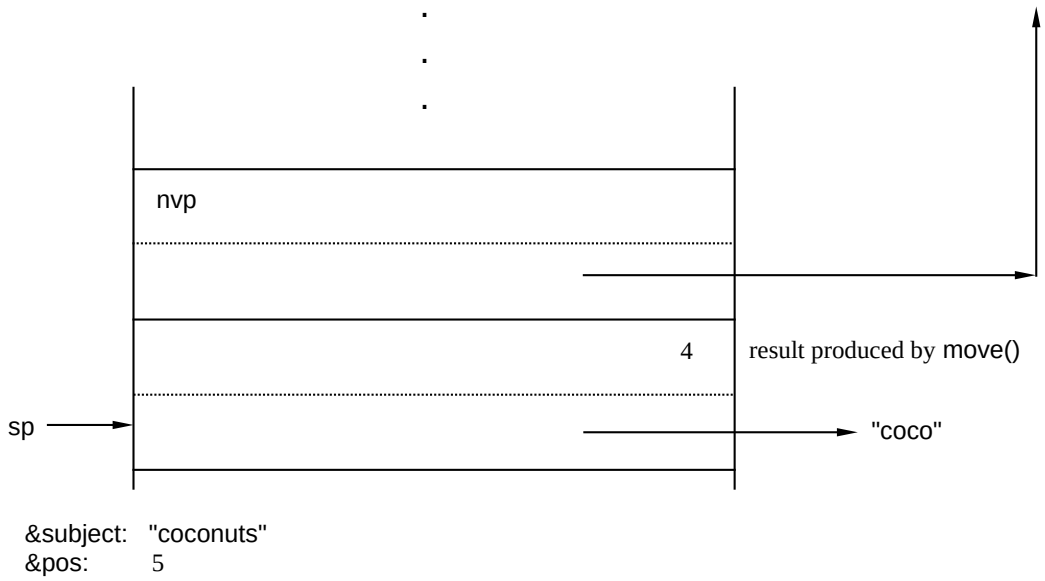
Consider the example on pages 159-162 of the implementation book. The first two diagrams are the same. On the third, the bscan instruction is executed, pushing the current values of &subject and &pos. It sets &subject to "coconuts" and &pos to 1. The bscan instruction needs to return a pointer to the saved values of &subject and &pos, so it constructs a variable on the stack and suspends.



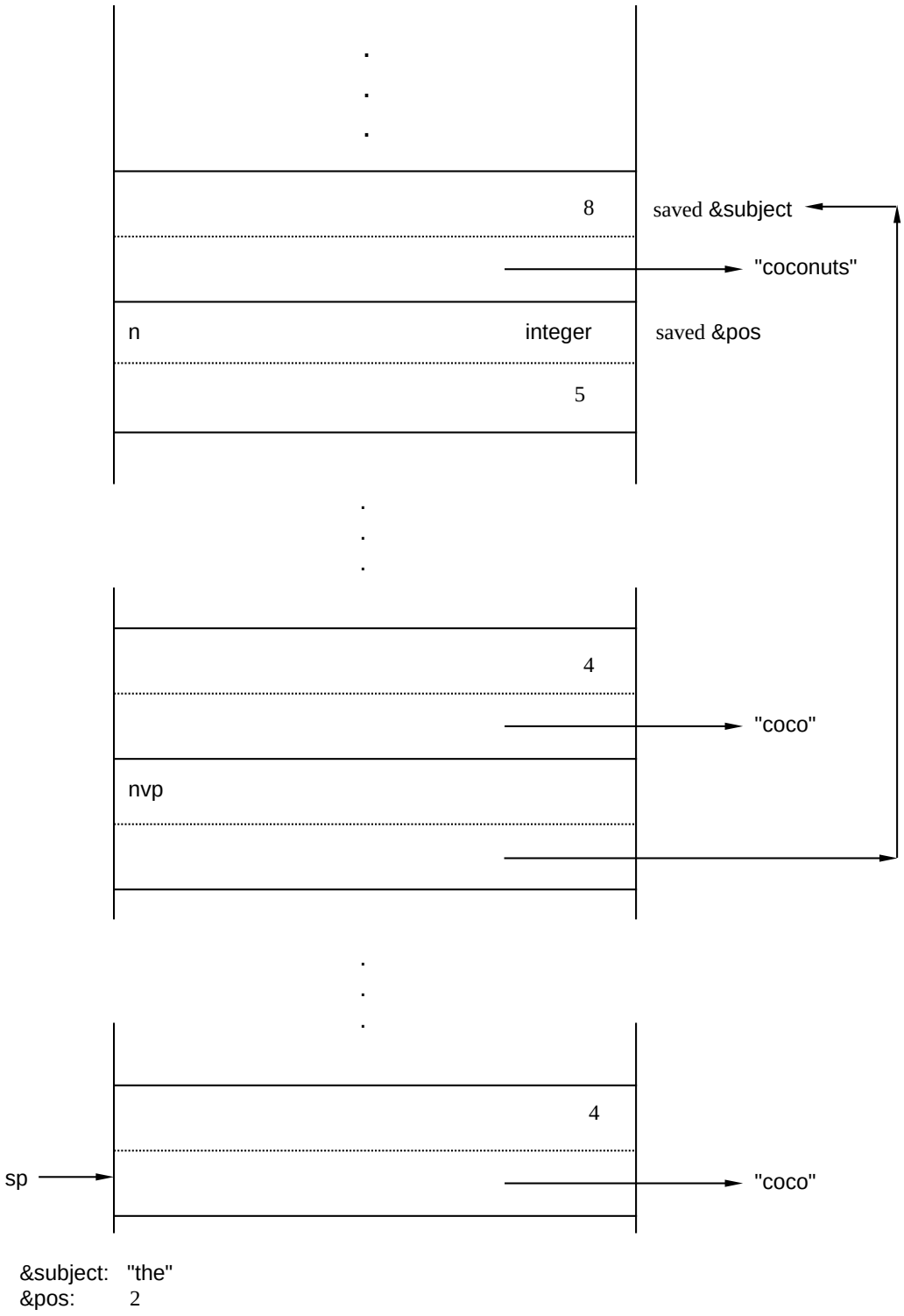
&subject: "coconuts"
&pos: 1

Since bscan suspends, the saved values of &subject and &pos are preserved in a generator frame until bscan is resumed or something causes removal of the current expression frame. This removal can be caused by break, next, fail, return, or the end of a bounded expression. In any of these cases, the interpreter returns a signal to bscan, which then restores the values of &subject and &pos before passing the signal on to the interpreter invocation below it.

Continuing the example, move(4) is evaluated, it suspends, and the top of the stack is



The escan instruction is executed next. It reverses the top two elements on the stack so that the result of scanning becomes the result of move(). It then exchanges the current values of &subject and &pos with the saved values and suspends:



Since escan suspends, the variable referencing the saved values of &subject

and &pos is preserved in a generator frame on the stack until the interpreter returns a signal to escan. When this happens, escan again switches the values of &subject and &pos with those saved on the stack. It then passes the signal on to the interpreter invocation below it.

The changes described so far handle the termination of string scanning by producing a result, expression failure, break, next, return, and fail and the removal of a suspended scanning expression on reaching the end of a bounded expression. The one case left is exiting string scanning by suspending out of a procedure. A field has been added to the procedure frame to handle this. It points to the descriptors holding the saved values of &subject and &pos that were in effect when the procedure was invoked:

```
struct descrip *pf_scan;        /* pointer to saved scanning environment */
```

When the procedure frame is constructed, this field is set to null because the scanning environment associated with the invocation is still active. If bscan determines that it is saving this environment (by seeing the null field), it fills in the field. When this environment is made active again, the field must be reset to null. Both bscan and escan participate in maintaining the field.

When a procedure suspends, the field is checked to see if the scanning environment of the invocation is active. If this environment is in a saved state, the currently active environment is exchanged with the saved environment. If the procedure is resumed, the environments must be switched back.

Having a procedure suspended in the middle of string scanning adds one additional complication. The bounded expression containing the procedure call may be removed. This results in an unwinding signal being propagated through bscan. However, bscan must not exchange the scanning environments again. This problem is solved by having bscan and escan switch environments only when a signal is received from the procedure they were called from. This is detected by comparing procedure frame pointers.

2.2 Co-Expression Activation [Section 10.4, pp. 180-182]

Previously, the implementation of co-expressions did not properly support an arbitrary sequence of activators for a given co-expression. Co-expression blocks had a descriptor that pointed to the most recent activator of the co-expression. Suppose co-expression A is activated by co-expression B. The activator descriptor of A references B and when A fails or returns, the activator descriptor of A directs it to pass control to B and this works properly. Consider, however:

```
A := create @B
B := create @C
C := create @B
@A
```

&main activates A, A activates B, and B activates C. At this point, the activator of B is A, but when C activates B, the activator of B is changed to C. At this same time, the activator of C is B. Thus, when B fails or returns, it passes control to C, which in turn passes control back to B — an endless loop.

Implementation Rationale and Overview

While a single activator works for the case of two co-expressions calling each other in a coroutine fashion, it seems to reasonable to support a more general case and allow co-expressions to have an arbitrary sequence of activators. This goal was achieved by replacing the single activator descriptor in a co-expression with a stack of activator descriptors. This stack allows for an arbitrary sequence of activators to be recorded and reversed with subsequent coret and cofail operations.

Theoretically, the sequence of activators can be of an arbitrary length and be comprised of an arbitrary number of distinct activators. Thus, the size of the stack used to record the sequence of activators should only be limited by available memory. To provide expandability, the activator stack is maintained as a linked list of blocks, each of which contain a fixed number of stack elements.

In many applications, a co-expression's activator is often the same, time after time. The most obvious example of this is two co-expressions A and B calling each other in a coroutine fashion: after the start-up sequence, the activator of A is always B, and conversely. In such a case, after *n* changes of control, each co-expression would have an activator stack containing *n* elements, all of which refer to the other co-expression. To accommodate this common use of

co-expressions, each stack element has a count field that indicates the number of consecutive activations by the associated activator. In the example, instead of having a stack of *n* identical elements, the stack consists of one element that has a count field of *n*. Although this heuristic can reduce memory throughput substantially, it should be noted that the underlying problem is one of data compression.

With a stack data structure such as this, it is only necessary to associate a stack with each co-expression, pushing activators in response to coact operations, and popping activators in response to coret and cofail operations.

Implementation Details

The activator field of the b_coexpr structure has been replaced with es_actstk, a pointer to an activator stack block, known as an astkblk. The astkblk structure is declared as:

```
struct astkblk {
    int nactivators;
    struct astkblk *astk_nxt;
    struct actrec {
        word acount;
        struct b_coexpr *activator;
        } arec[ActStkBlkEnts];
    };
```

The array arec, composed of actrec entries, is a segment of the stack. nactivators indicates the number of valid activator entries in this segment of the stack. arec[nactivators–1] is the most recent activator. astk_nxt is a pointer to the next stack block in the list. ActStkBlkEnts is a compile-time constant whose current value is 100 for systems with a large amount of memory and is 10 for systems with a small amount of memory or fixed-size memory regions.

When a co-expression is created, the es_actstk field is set to zero. Upon the first activation of the co-expression, memory for an astkblk is allocated using malloc(), and es_actstk is pointed to this block.

Four routines manipulate co-expression activator stacks:

pushact(C1,C2)	The co-expression C1 is added to the activator stack of co-expression C2.
popact(C)	The activator stack of C is popped and the top-most activator is returned.
topact(C)	The most recent activator of C is returned.
dumpact(C)	The activator stack of C is dumped (used for debugging only).

pushact(C1,C2) first determines if the most recent activator of C1 was C2 and if so, it merely increments acount in the appropriate element of arec and returns. If C2 was not the last activator of C2, a new arec element is required. If arec is full, a new astkblk is allocated and added at the front of the list based at es_actstk. The appropriate slot in arec is located, filled in with the new activator, acount is set to one, and nactivators is incremented.

pushact is called in the Op_Coact case of interp, and it is also called in init() to make &main its own activator.

popact(C) locates the arec entry that is associated with the most recent activator and holds the co-expression pointer for later return. If the acount field is one, nactivators is decremented to reflect the removal of an arec entry. If acount is greater than one, acount is merely decremented. When all the arec entries in an astkblk have been popped, the astkblk needs to be freed. An astkblk is freed when nactivators is zero and a pop is required.

popact() is called in the Op_Coret and Op_Cofail cases of interp().

topact(C) is only required for &source and merely returns the most recent activator. The only complication is to skip the first astkblk on the chain if its nactivators field is zero.

Garbage Collection Issues

A co-expression is live with respect to activators if it is the activator of a live co-expression, since the co-expression it activated may coret or cofail. Thus, all the activators in the activator stack of a live co-expression must be marked. This marking is done with special-case code in markblock(). markblock() requires the address of a descriptor, but the activators are stored as a list of addresses of b_coexpr blocks rather than as a list of descriptors. Because back-chaining is not done with co-expressions and the descriptors that reference them, it is acceptable to use a dummy descriptor, filling in its v-word with each activator address in turn and calling markblock() with the address

of this descriptor.

When a co-expression is freed in cofree(), it has a still-allocated astkblk associated with it that must be freed. It seems that there is no way to produce a dead co-expression that has a non-empty activator stack, but nonetheless, code is present to handle multiple astkblks.

When pushact(requires a new astkblk, the ensuing malloc() may cause a garbage collection. This potential allocation is handled by doing the pushact() in the Op_Coact case prior to establishing any pointers to relocatable data objects. This ensures that no C pointers are invalidated, and that there are no reachability problems, so this solution seems satisfactory. Having predictive need for the static memory region would avoid this special case, but the nature of the memory management used in the static region precludes a reasonably quick way of determining if a given amount of memory is available. This problem merits further thought.

2.3 Storage Management [Section 11.3.5, pp. 213-214]

The original implementation of storage management for Icon was predicated on the assumption that the user's memory region can be expanded using sbrk() if additional space is needed during program execution. This approach works well on some systems, but is awkward on others and some systems do not support sbrk() at all. The trend is away from the support of sbrk().

While the matter still is not completely settled (storage management remains the biggest single problem in the implementation of Icon), an alternative method of handling Icon's memory regions has been added. This method, called ''fixed regions'' uses the system's malloc() to provide separate string and block regions at the beginning of execution, as opposed to the usual ''expandable regions'', in which one large block for all regions is obtained by brk().

With fixed regions, the string and block regions are not necessarily contiguous, and neither can be expanded (although their initial sizes can be set by environment variables). Co-expressions are allocated as needed using the system's malloc(), instead of being allocated by Icon from its static region. Fixed regions do not affect the method used for garbage collection, but if there is not enough space after collection in either the string or block region, execution is terminated with an error message instead of expanding the region.

Portions of the source code that are affected by the fixed-regions option are identified by the defined symbol FixedRegions. The main module related to storage management, rmemmgt.c, now includes either rmemfix.c or rmemexp.c depending on whether or not FixedRegions is defined.

Some other changes have been made to the expandable-regions version of storage management to adjust region sizes in the event there is not enough memory to expand a region to the desired size.

Close examination of the present source code will reveal that the fixed- and expandable-regions options co-exist somewhat uncomfortably; the present situation is a work-around to allow Icon to run on systems that do not allow expansion of the user's memory region, but it is not a complete solution to the problem. It probably would be a good idea to start from scratch and completely redesign Icon's storage-management system. This would be a *major* project, however, and it is not even clear that a single method, however clever and sophisticated, can do a good job on the wide range of computer architectures presently available.

2.4 Large Integers [Section 4.1, p. 44]

Version 8 of Icon supports large-integer arithmetic in which the magnitude of integers is not limited by machine architecture. Large integers do not come into existence until integer overflow would occur. See h/rt.h and iconx/rlargint.c for details.

Overflow checking in C is now provided. See iconx/rmisc.c.

2.5 Dynamic Hashing [Chapter 7, pp. 96-109]

Earlier versions of Icon implemented sets and tables using hash tables with a fixed number of buckets (37). This worked well for small tables, but performance degraded for large tables due to the long hash chains in each bucket. Now, each hash table starts out with a fixed number buckets (typically 8), but the number of buckets doubles repeatedly as the number of elements grows. Small tables benefit from reduced memory requirements, while large tables show dramatic performance gains.

Data Structures

Sets and tables use very similar data structures, and now use common code in many places. Tables differ from sets only by having an additional descriptor in the header, for the default value, and an additional descriptor in each element, for the referenced value. The discussion that follows refers to sets, but the implementation of tables is similar.

Hash buckets are grouped in segments; each segment is a separate block in the heap region. A minimal set has a single segment containing 8 hash buckets. Successive segments hold 8, 16, 32, 64,... buckets, with each additional segment doubling the total bucket count. Segments are located via an array of pointers in the set header block. Because this array is fixed in size, there is a limit to the number of times the hash table can expand.

The set header also contains a mask that is combined with a hash value to produce its bucket number. The mask is always one less than the number of buckets, which is in turn a power of two.

Structure Growth

A set's load factor is its size divided by the number of hash buckets. When the addition of an element causes the load factor to exceed 5, an additional bucket segment is allocated. One additional bit is added to the set's hash mask, and each element having that bit set in its hash value is moved to one of the new buckets.

Deletion of elements does not reduce the number of hash buckets because of the complicating effect on element generation. However, if a sparse set is copied, the copy has a smaller and more appropriate number of buckets.

Element Generation

Element generation is usually a straightforward process: the hash chains of each bucket of each segment are traversed, generating each element in turn. Complications arise if hash buckets are added while the generator is suspended; in general, the new buckets contain some elements already generated and some not yet generated.

Each time a suspended generator regains control, it checks to see if the set has split its buckets. If so, it records the set's old mask value and the suspended element's hash value for later reference. These are saved in two arrays indexed by the number(s) of the newly created hash bucket segment(s).

When a hash bucket is about to be traversed, the arrays are checked. From the saved masks and hash values it is possible to determine whether or not the contents of the hash bucket were generated earlier. If so, the bucket is skipped. But the contents may have been partially generated, in which case those elements are skipped whose hash values do not exceed the saved value.

One additional complication comes from sequences of elements in a chain having identical hash values: the mask and hash value are not sufficient to record the position within such a sequence. The problem is avoided by deferring the detection of any bucket splits until reaching the end of the sequence. This works because the sequence can never be split into multiple buckets.

Improved Hashing

New hash functions were introduced for strings, integers, and reals.

The former string hash function did not produce enough distinct values for use with a large number of hash buckets, and was insensitive to permutations. A new function fixes both of these problems.

Integers were formerly used as their own hash values without further manipulation. That worked well with 37 hash bins but would have been unacceptable in the new scheme; for example, a set composed of even numbers would have used only half the hash bins. Integers are now hashed by multiplying by a constant based on the golden ratio.

Real numbers were formerly ''hashed'' by simple truncation to integer. Besides the even-number problem, this discarded all information from the fractional part. Now, real numbers are multiplied by a constant that scales them up and scrambles the bits.

Configuration Parameters

Four configuration parameters control the functioning of tables and sets:

HSlots — Sets the initial number of hash buckets, which must be a power of 2. The default is 4 on machines with small address spaces and 8 otherwise.

HSegs — Sets the maximum number of hash bucket segments. Consequently, he maximum number of hash buckets is $\mathsf{HSlots} \times 2^{\mathsf{HSegs}-1}$. The default is 6 on machines with small address spaces and 10 otherwise.

MaxHLoad — Sets the maximum allowable loading factor before additional buckets are created. The default is 5.

MinHLoad — Sets the minimum loading factor for new structures. The default is 1. Because splitting or combining buckets halves or doubles the load factor, MinHLoad should be no more than half MaxHLoad.

Visible Impacts

The main effects of these changes are reduced CPU requirements, better memory utilization, different results from ?X, and generator behavior that is both different and ''more random''.

Set and table element generation has always produced results in an undefined order. Generation is now ''more random'' in the sense that the sequences of results are even more unpredictable than before. First, the generated sequences are dependent on configuration parameters. Second, adding new elements can change the relative order in which existing elements are generated; this quirk even could be used to determine the configuration parameters. Finally, it is possible to construct a structure X such that !X and !copy(X) produce results in different sequence.

2.6 C Data Sizes [Sections 4.1 and 4.4, pp. 48-51, 57]

The C programming language provides no guarantees that ints and pointers are the same size, although that was true for most computers on which C was originally implemented. In fact, Icon was written on such a computer and with the assumption that ints and pointers were the same size and there was no computer available for which the assumption did not hold.

While the implementation subsequently was modified to support ''mixed sizes'', and this subject is covered in the implementation book, many changes have been made since the book was written to accommodate the idiosyncrasies of various C compilers.

To get an idea of the problem, consider the implication of the original specification in the C language that the difference of two pointers is an integer (but explicitly not whether this is an int or a long). There generally is no problem if ints and pointers are the same size, but for the so-called ''large memory model'' where ints are 16 bits and pointers are 32 bits, there are serious practical problems. Some large-memory-model C compilers take the difference of two 32-bit pointers to be a 16-bit int. It doesn't take a lot of imagination to see what happens if the difference of two pointers is larger than the largest int.

Such problems can be handled by casting the pointers to longs wherever such problems might arise. The catch is finding all the places and doing it properly. In fact, the source code for Icon presently is not completely correct in this respect. We have, however, added a *lot* of casts to the source code since the book was written. This changes the appearance of the code substantially in some places (and not for the better). The point is, if you are reading the book and the source code side-by-side, you will see many differences for this reason alone.

In a related matter, the interpreter program counter, ipc (page 128 of the implementation book) is now a union. This change was made so that space could be saved in icode files by mixing ints (for opcodes) with pointers to data (for arguments). References to the icode now specify the appropriate member of the union.

In order to provide more flexibility in configuring the Icon source code for computers with different C data sizes, the following values now are used:

ByteBits	number of bits in a byte (normally 8)
IntBits	number of bits in a C *int* (nominally 32; may also be 16 or 64)
WordBits	number of bits in an Icon ''word'' (nominally 32; may also be 64)

It's also worth noting that Icon no longer can be built as a ''small-memory-model'' program: pointers must be at least 32 bits long.

2.7 Changes in Handling of Data Objects

Integers [Section 4.1 and Section A.2.1, p. 51, 247]

Icon source-language integers are 32 (or 64) bits, regardless of the size of the C int. Consequently, on some systems Icon integers are ints, while on others, they are longs. In Version 6, there were two kinds of integers on 16-bit computers: those that fit into a C *int* and those that required a C *long*. Blocks were allocated for the latter. Now all Icon integers are kept in ''words'' which are at least 32 bits long. This is handled by a *typedef* to either *int* or *long*, depending on the size of an *int*. Descriptors are simply two words and blocks are no longer allocated for ''long integers''. Thus, the discussion and diagrams on pages 51 and 247 of the book no longer apply.

Csets [Section 5.2, p. 78]

When the implementation book was written, the size of a cset was computed when it was created. This computation is fairly expensive, but few programs ever use the sizes of csets. The size of a cset is now set to −1 when it is created and the actual size is only computed and reset if it is needed.

Pointers in Blocks [Chapter 6 and Section 11.3.2, pp. 80-109, 199, 291]

In Version 6, all pointers in blocks to other blocks were contained in descriptors. Now most of these are just pointers. The layout of blocks now is first all non-pointer and nondescriptor data, then pointers, and finally descriptors. The garbage collector uses two new tables, firstp and ptrno, to locate and process pointers.

Serialized Structures

Blocks for structures are now serialized, which adds an additional word for those types. The serial numbers are used in hashing and also appear in string images of structures.

Pointers to Variables [Section 4.3, pp. 53-54, 200-203]

In order to handle pointers in place of descriptors within blocks during garbage collection, a variable descriptor that points to a block now points to the title of the block and the offset to the corresponding value is in the d-word of the variable, rather than pointing to the value with the offset being to the title.

Lists [Section 6.1, pp. 81-82]

Formerly all newly created lists contained space for a minimum of 8 elements. Now, only empty lists contain space for extra elements.

Previously, list-element blocks that are allocated when a list is extended as the result of a push() or put() contained space for 8 elements. If large lists are created in this way, the total amount of space overhead is excessive and the time to access elements increases with the position. To avoid this, when a list-element block is allocated for extending a list, it now contains one-half the total number of elements in the entire list prior to extension or 8, which ever is larger. However, if there is not enough memory to allocate a list-element block of the desired size, the size is reduced to fit into available memory. While this may waste space in the last list-element block, the total amount of space for the list is always less than for the previous allocation strategy, and the time to access list elements toward the end of the list increases less rapidly than formerly.

2.8 Run-Time Errors [Section D.2.8, p. 288]

Most run-time errors now can be converted to expression failure under the control of a keyword. Consequently, the function runerr() may now return, whereas it formerly did not. To avoid accidentally continuing unexpectedly in code that formerly followed calls to runerr(), such calls have been replaced by instances of the macro RunErr() which is defined as

```
#define RunErr(i,dp) {\
   runerr(i,dp);\
   Fail;\
   }
```

In those cases where a run-time error cannot be converted to failure, a new function, fatalerr(), is used.

To distinguish those cases where a run-time error cannot be attributed to a specific offending value, the negative of the error number is used in calls to runerr() and fatalerr(). The actual error number reported is always positive.

2.9 Function Declaration and Definition [Section D.2, pp. 280-284, 289]

There are now two additional forms of function declaration that specify arguments are not to be dereferenced prior to function invocation:

```
FncNDcl(name,n)          /* n arguments */
FncNDclV(name)           /* variable number of arguments */
```

In addition, the function-definition macro FncDef() now has a second argument that specifies the number of arguments for the function: FncDef(name,n).

2.10 Reorganization of the Translator and Linker [Section D.1, p. 279]

At the time the implementation book was written, there were four components to the Icon source code: a command processor (icont), a translator (itran), a linker (ilink), and an executor (iconx). The first three of these have been combined into a single component so that the command processor (icont) now calls the translator and linker as functions.

2.11 Changes for the ANSI C Draft Standard [Section 4.4, p. 57]

Several changes have been made to conform to the ANSI C draft standard. For example, there is now a typedef for pointer that is void * for C compilers supporting the draft standard but char * for those that do not.

2.12 Cosmetic Changes

Most of the cosmetic changes that have been made to the source code of Icon since the implementation book was written and are obvious when comparing source code in the book to the current source code.

Two names have changed: MkInt is now MakeInt and mkreal is now makereal.

One change that should be specifically noted is the introduction of

```
typedef struct descrip *dptr;
```

3. Corrections to the Implementation Book

The following errors appear in the first printing of the implementation book. The line numbers given include captions and titles, but not running heads. Negative line numbers are relative to the bottom of the page.

Page 19, line 7: Replace ‘‘string of length i positioned’’ by ‘‘string of length i with s1 positioned’’

Page 20, line 19, last word: Replace ‘‘second’’ by ‘‘rightmost’’.

Pages 53-55: The label s next to the variable descriptor should instead be next to the descriptor it points to (three places).

Page 63, line 4: Replace ‘‘NULL’’ with ‘‘CvtFail’’.

Page 71, line 1: Replace s1[1:2] by ‘‘s1[1:3]’’

Page 72, line 10: Replace ‘‘Sec. 2.2’’ by ‘‘Sec. 4.3.2’’.

Pages 72-73: The label s next to the variable descriptor should instead be next to the descriptor it points to (two places).

Page 74, line 14: Replace ‘‘information must information must’’ by ‘‘information must’’.

Page 77, line 14: Replace ‘‘C code cset’’ by C code for cset’’.

Page 78. The indentation in the code is inconsistent and there are unnecessary braces. Since cset sizes are no longer computed when csets are created, the latter portion of the code no longer exists. The present code is:

```
/*
 * Allocate a new cset and then copy each cset word from Arg1
 *  into the new cset words, complementing each bit.
 */
bp = (union block *)alccset();
for (i = 0; i < CsetSize; i++)
    bp->cset.bits[i] = ~cs[i];
Arg0.dword = D_Cset;
BlkLoc(Arg0) = bp;
Return;
}
```

Note that alccset no longer has the argument that previously provided the cset size.

Page 87, line –2: ''put'' should be ''pop''.

Page 106, line –8: Replace ''or operation, and the result is cast as an integer'' by ''or operation.'' .

Page 140, line –11 (in the second sentence in the paragraph that starts ''Generator Frames''): ''begins with generator frame'' should be ''begins with a generator frame''.

Page 141, line –10: There should be a pnull instruction between local i and int 1.

Page 151, line 4: There should be a pnull instruction between local i and int 1.

Page 155, line 11: ''if *expr* produces a result'' should be ''if *expr* does not produce a result''.

Page 179, lines 13-14: The transmitted value is shown as a descriptor. It should be a single word that points to a descriptor. In addition, co-expression blocks now contain fields for an identifying number and for a pointer to an activator stack as described in Section 2.2.

Page 198, near end of first paragraph: Replace the sentence that starts ''The back chain is established'' by ''The back chain is established by setting the title word of the block to point to the descriptor, and setting the v-word of the descriptor to the previous contents of the title word.'' .

Page 201, line 12: There is an extraneous ''y'' at the left of the last descriptor in the diagram.

Page 210, line 10: There is another extraneous ''y'' as on page 201.

Page 215, lines 24-25 (at the beginning of the second paragraph of text): It is stated that the context for evaluation is switched to the co-expression for &main, so that a larger C stack is available. This is not done in the current implementation; it got lost somewhere in an update. (It should be done.)

Page 226, line 11 (third line after list of routines): Replace ''NULL'' by ''CvtFail''.

Page 228, line –6: Replace ''NULL'' by ''CvtFail''.

Page 256: The remarks about page 179 apply here also.

Page 276, last sentence in first paragraph: Replace the sentence that begins ''The instruction escan'' by ''The instruction bscan saves the current values of &subject and &pos and establishes their new values before $expr_2$ is evaluated. The instruction escan restores their values.'' .

Page 290, line –7: Replace ''int title'' by ''word title''

Page 291, line 2: Replace ''int title'' by ''word title''

Page 291, line 3: Replace ''int blksize'' by ''word blksize''

page 284, line 20: Replace ''type code'' by ''d-word''.

Page 294, line 18: Replace ''iconx/rt.h'' by ''h/rt.h and iconx/gc.h''.

Pages 296-297: Replace all occurrences of ''NULL'' by ''CvtFail''.

Page 327, line –1: Replace ''Yyngve'' by ''Yngve''.

Acknowledgements

Ken Walker made the changes to the maintenance of scanning environments and provided the documentation included here. Bill Mitchell made the changes to the handling of co-expression activation and provided the documentation included here. Cheyenne Wills and Kelvin Nilsen made most of the changes to support ‘‘mixed sizes’’. Bob Alexander suggested deferring the computation of cset sizes until they are needed. Dave Gudeman provided the new code for the handling of variable-sized list-element blocks. Sandy Miller changed descriptors in blocks to pointers and made the associated changes to garbage collection. Gregg Townsend implemented dynamic hashing and provided the documentation included here.

Many other persons, too numerous to list here, provided changes to the source code related to porting it to new computers.

Bob Alexander, Rick Fonorow, Dave Hanson, Robert Henry, and Janalee O’Bagy found the errors in the implementation book that are listed in Section 3.

References

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, second edition, 1990.
2. R. E. Griswold, *Version 8 of Icon*, The Univ. of Arizona Tech. Rep. 90-1, 1990.
3. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, 1986.